Lilith Handbook

A Guide for Lilith Users and Programmers

Leo Geissmann
Jiri Hoppe
Christian Jacobi
Svend Erik Knudsen
Werner Winiger
Niklaus Wirth

August 1982

Table of Contents

21.5.82

1. Introduction
   - 1.1. Handbook Organization
     - 1.1.1. Overview of the Chapters
     - 1.1.2. Page Numbers
   - 1.2. Overview of Lilith
     - 1.2.1. Software
     - 1.2.2. Hardware
   - 1.3. References

2. Running Lilith
   - 2.1. Getting Started
   - 2.2. Exchange of the Disk Cartridge
   - 2.3. Termination of a Session

3. Running Programs
   - 3.1. The Command Interpreter
     - 3.1.1. Program Call
     - 3.1.2. Typing Aids
     - 3.1.3. Loading and Execution Errors
   - 3.2. Command Files
   - 3.3. Program Loading

4. Things to Know
   - 4.1. Special Keys
   - 4.2. File Names
     - 4.2.1. File Names Accepted by the Module FileSystem
     - 4.2.2. File Name Extensions
     - 4.2.3. File Name Input from Keyboard
   - 4.3. Program Options
   - 4.4. The Mouse

5. The Editor
   5.1. Introduction
   5.2. Starting the Editor and Entry of New Text
   5.3. Positioning the Document
      5.3.1. Scrolling
      5.3.2. Flipping
   5.4. Insert Characters at Different Locations
   5.5. Activating the Menu and Making a Selection
   5.6. Delete, Move, or Copy Text
   5.7. Termination of an Editing Session
   5.8. Rescue from Abnormal Termination and Other Errors
   5.9. Searching
   5.10. Working with Windows
      5.10.1. Open a Window
      5.10.2. Open a Document
      5.10.3. Change the Size of a Window
      5.10.4. Close a Window or a Document
   5.11. Accelerators
   5.12. User Profile and User Guidance
   5.13. Special Characters

6. Utility Programs
   6.1. directory
   6.2. delete, protect, and unprotect
   6.3. copy and rename
   6.4. list
   6.5. inspect
   6.6. xref
   6.7. link
   6.8. decode
   6.9. layout
   6.10. hermes
   6.11. hpcopy
   6.12. backup and restore
   6.13. boot and altboot

7. The Compiler
   7.1. Glossary and Examples
   7.2. Compilation of a Program Module
   7.3. Compilation of a Definition Module
   7.4. Symbol Files Needed for Compilation
   7.5. Compiler Output Files
   7.6. Program Options for the Compiler
   7.7. Compilation Options in Compilation Units
   7.8. Module Key
   7.9. Program Execution
   7.10. Value Ranges of the Standard Types
   7.11. Differences and Restrictions
   7.12. Compiler Error Messages

8. The Debugger
8.1. Starting the Debugger
8.2. General Debugging Dialog
8.3. The Debugger Windows and Their Commands
8.3.1. The Procedure Chain Window
8.3.2. The Program Window
8.3.3. The Data Window
8.3.4. The Dialog Window
8.3.5. The Memory Window
8.3.6. The Load Map Window
8.3.7. The Process Window
8.3.8. The Screen Window
8.3.9. The Background Commands
8.4. An Example
8.4.1. Screen with Default Layout
8.4.2. Screen with Additional Opened Windows

9. The Medos-2 Interface
9.1. Module FileSystem
9.1.1. Introduction
9.1.2. Definition Module FileSystem
9.1.3. Simple Use of Files
9.1.3.1. Opening, Closing, and Renaming of Files
9.1.3.2. Reading and Writing of Files
9.1.3.3. Positioning of Files
9.1.3.4. Examples
9.1.4. Advanced Use of Files
9.1.4.1. The Procedures FileCommand and DirectoryCommand
9.1.4.2. Internal File Identification and External File Name
9.1.4.3. Permanency of Files
9.1.4.4. Protection of Files
9.1.4.5. Reading, Writing, and Modifying Files
9.1.4.6. Examples
9.1.5. Implementation of Files
9.1.6. Files on Cartridges for Honeywell Bull D120/D140 Disk Drives
9.1.6.1. Main Characteristics and Restrictions
9.1.6.2. System Files
9.1.6.3. Error Handling
9.2. Module Program
9.2.1. Introduction
9.2.2. Definition Module Program
9.2.3. Execution of Programs
9.2.4. Heap
9.2.5. Error Handling
9.2.6. Object Code Format
9.3. Storage
9.4. Terminal

10. Screen Software
    10.1. Summary
    10.2. Screen
    10.3. TextScreen
    10.4. WindowHandler
    10.5. CursorStuff
    10.6. CursorRelations
    10.7. WindowDialogue
    10.8. ScreenResources0
    10.9. BitmapVars

11. Library Modules
    11.1. InOut
    11.2. RealInOut
    11.3. Mouse
    11.4. LineDrawing
    11.5. MathLib0
    11.6. OutTerminal
    11.7. OutFile
    11.8. OutWindow
    11.9. ByteIO
    11.10. ByteBlockIO
    11.11. FileNames
    11.12. Options
    11.13. Line
    11.14. V24

12. Modula-2 on Lilith
    12.1. Code Procedures
    12.2. The Module SYSTEM
    12.3. Data Representation and Parameter Transfer
        12.3.1. Data Representation
        12.3.2. Parameter Transfer

13. Hardware Problems and Maintenance
    13.1. What to Do if You Assume some Hardware Problems
    13.2. DiskCheck
    13.3. DiskPatch

1. Introduction

Leo Geissmann 15.5.82

The *Lilith* computer is intended to be used as a flexible workstation by individual users. This guide will give an introduction to the use of the machine and the basic software environment running on 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 6.

1.1.1. Overview of the Chapters

Chapter 1 gives introductional comments on the handbook and on Lilith.

Chapter 2 gives instructions on how Lilith is started.

Chapter 3 describes how programs are called with the command interpreter.

Chapter 4 provides information about the general behaviour of programs.

Chapter 5 describes the use of the text editor.

Chapter 6 is a collection of important utility programs, needed by all Lilith users.

Chapter 7 describes the use of the Modula-2 compiler.

Chapter 8 describes the use of the post-mortem debugger.

Chapter 9 is a collection of library modules constituting the Medos-2 interface.

Chapter 10 is a collection of library modules constituting the screen software interface.

Chapter 11 is a collection of further commonly used library modules.

Chapter 12 describes the Lilith-specific features of Modula-2.

Chapter 13 describes procedures to follow if Lilith is not working as expected.

1.1.2. Page Numbers

It is intended that the page numbers facilitate the use of the handbook. It should be possible to find a chapter quickly, because the chapter number is encoded within the page number. The pages belonging to a chapter are enumerated in the *thousends digit* of the chapter number, i.e. in the first chapter the page numbers start with 1001, in the second chapter with 2001, etc. As a chapter has less than one hundred pages, the chapter number is always separated from actual page number within the chapter by a zero.

1.2. Overview of Lilith

1.2.1. Software

Lilith is programmed in the language *Modula-2*, which is defined in the Modula-2 manual [1]. Some specialities of Modula-2 on Lilith are mentioned in chapter 12 of this handbook.

The resident operating system on Lilith 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.

One of the most frequently used program is the *text editor*. It is used for writing and modifying text and programs. Programmers also need the Modula-2 *compiler* and, in the case that program execution should fail, the post-mortem *debugger*.

The handling of the screen display is provided by the *screen software* package. It enables writing and drawing at any place on the screen. A window handler provides the subdivision of the screen into smaller independent parts, called *windows*.

Further, there exists a large number of utility programs and library modules. The most commonly used subset is described in this handbook; the handbook should never be considered to give a complete overview of the Lilith software.

1.2.2. Hardware

The *Lilith* computer consists of a processing unit, which includes the main store, peripheral devices, and a power supply. The store has a capacity of 128K (131'072) *words* of 16 bits each. The standard peripheral devices are a *display* for visual output, a *keyboard* and a so-called *mouse* for manual input. Furthermore, there is a secondary store consisting of a *magnetic cartridge disk* with a capacity of 10 MByte. It is used to store and retain files. A description of Lilith is given in the Lilith report [2].

The display uses the raster scan technique with 592 lines and 768 dots per line. The total number of dots is 454'656, and each dot is represented in the main store by a bit. This representation is called the *bitmap*; if the full screen is represented, it occupies 28'416 words. The display controller allows to reduce the bitmap's size and to use part of the screen only, or even to discard it altogether.

The direct representation of the screen as a bitmap gives the programmer a high degree of freedom for manipulation of the displayed information. Diagrams and pictures can be shown as well as text. In fact, each character of a text is a picture itself, represented by an array of bits computed by the program from the character's internal (ASCII) encoding. This offers the possibility to use different visual styles (i.e. *fonts*) for characters.

The keyboard uses the standard ASCII character set with 96 printing characters (plus a few extra keys for control characters, which may be ignored for almost all uses). The mouse allows movements of the user's hand holding the mouse to be read by the processor. These movements can be translated by appropriate programs into corresponding movements of a cursor displayed on the screen. The mouse also features three pushbuttons (keys) used to indicate commands.

In addition, the computer also provides a standard *serial line interface* (V24, RS232). It can be used to connect to printer terminals or other devices, including of course other computers.

1.3. References

[1] **Programming in Modula-2**
N. Wirth, Springer-Verlag, Heidelberg, NewYork, 1982.

[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 Lilith

Leo Geissmann 15.5.82

2.1. Getting Started

The computer is switched on by pushing the *red power switch* on the cabinet. As soon as the *white disk switch* lights up, the square, black *disk cartridge* may be inserted into the disk drive. Afterwards, push the *white disk switch* on the cabinet (to start the disk) and the *reset button* at the rear of the keyboard (to set the computer ready for a bootstrap). Finally, hit the *space bar* or CTRL-A on the keyboard.

As soon as you hear a short, clicking noise, the disk is ready for operation, and the bootstrap of the computer is started. The resident operating system Medos-2 is loaded from the disk. After successful loading, it first displays a *version number* in the top left corner of the screen.

```
V4
```

Sometimes a bootstrap is not successful the first time, i.e. the version number does not appear. In this case retry the bootstrap: Push the reset button again and hit the space bar or CTRL-A.

The operating system now makes some initializations. If everything is all right, a *dot* appears behind the version number.

```
V4.
```

It is possible that there appears an error message instead of a dot. This indicates that something may be wrong with the computer or with the disk cartridge. Chapter 13 describes what to do in this case.

The *command interpreter* now displays a version message. Afterwards, the command interpeter displays the date of the last use of the disk cartridge and prompts for the current date.

```
old date = 25.8.81
new date >
```

The new date is expected in the same format as shown by old date, whereby only the changed numbers must be specified. The input of the new date is terminated by hitting the RETURN key or the *space bar*. Hitting only the RETURN key means that the old date is still valid. After the date is accepted, an asterisk * is displayed. The command interpreter is now ready to accept the name of a program which should be executed next. How programs are called is described in chapter 3.

For advanced users: The disk cartridge contains two so-called *boot files*. According to the key pressed when a bootstrap is started, one of these files is loaded into the memory. File `PC.BootFile` is read when the space bar is pressed, and file `PC.BootFile.Back` is read when CTRL-A is typed. This allows to substitute, with utmost care, a different bootstrap program. Boot files are linked in an absolute format and cannot be executed like other programs. They are generated by a special bootlinker program.

2.2. Exchange of the Disk Cartridge

If you want to work with another disk cartridge, you have to exchange it. First, be sure that an asterisk was last displayed, i.e. the command interpreter is active to accept a program name. Afterwards, switch off the disk (white disk switch) and wait until the disk has stopped. This is signalled by a light on the disk switch and by a short clicking noise. Now, the disk cartridge in the disk drive may be replaced by another one. Restart the disk by pushing the white disk switch again.

While the disk cartridge is exchanged, a waiting message is displayed on the screen.

```
****************
*              *
 * I'm waiting  *
*              *
****************
```

As soon as the disk is ready again, the message disappears and the operating system is caused to make some reinitializations. This "soft bootstrap" is important because the operating system stores some information about the loaded disk cartridge in the memory. The reinitialization is indicated by the version number and the dot on the top left corner of the screen. Finally, an asterisk * is displayed and a program name is accepted again.

WARNING

If the disk cartridge is exchanged during execution of a program, this exchange will not be detected by the operating system and therefore *no reinitialization* takes place. In this case a real *bootstrap* of the computer is *mandatory* (see 2.1.). Otherwise, there might be problems with the disk cartridge sooner or later.

2.3. Termination of a Session

For termination of a Lilith sesson, exchange the disk cartridge with the grey *dummy cartridge* (do not restart the disk). The dummy cartridge is very important to protect the disk drive from dust.

The computer may now be switched off with the red power switch. This, however, is not necessary if somebody else wants to work with the machine. In this case leave it in the waiting state, or start the program `HardwareTest` before switching off the disk. This program will use the idle time to run some checks on the computer's hardware.

3. Running Programs

Svend Erik Knudsen 15.5.82

This chapter describes, how programs are called with the *command interpreter* of the Medos-2 operating system. An often used sequence of program calls may be controlled by a *command file.*

3.1. 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 interprete 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. But, this fact is transparent to most users of Medos-2.

3.1.1. 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 commands: *program calls.*

To call a program, type a program name on the keyboard and terminate the input by either hitting the RETURN key or pressing the space bar.

```
*directory
```

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.

```
*direx
 program not found
*directory
                                        directory 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. It inserts the medium name DK at the beginning of the program name, appends an 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 program name	`directory`
First file name	`DK.directory.OBJ`
Second file name	`DK.SYS.directory.OBJ`

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

3.1.2. 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.* Further, there are the *automatic extension* of a typed character sequence and some *special keys.*

Automatic Extension

The command interpreter automatically extends an initially typed character sequence to the name of an *existing* program. This means that a long program name may be identified by a few characters. If several programs exist whose names start with the typed character sequence, the sequence is only extended up to the point where the names start to differ. In this case, further characters are needed for identification. The input of a program name must be terminated by either hitting the RETURN key or pressing the space bar.

The command interpreter needs a few seconds to find all the names of available programs. Therefore, automatic extension is only possible after that time. If a command is typed very fast (or probably before the asterisk is displayed), the meaning of the termination character may be different. Termination with RETURN means that the command should be accepted as it is, termination with the space bar means that the command interpreter should try to extend the character sequence to a program name before accepting it.

Special Keys

While typing a program name, the command interpreter also accepts some special keys which are executed immediately.

?

HELP character. It causes the display of a list of all programs, whose names start with the same character sequence as the typed one. At the end of the list, the already typed part of the program name is displayed again, and the rest of the program name is accepted.

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 just *behind an asterisk.* It is not accepted within a character sequence.

ESC

Terminate the execution of the command interpreter.

CTRL·C

Kill character. This key may be typed at any time. The currently executed program will be *killed* and a *dump* will be written on the disk cartridge. The dump may be inspected with program *debug.* Obviously, the CTRL-C key is built into Medos-2 in order to help the programmer during unavoidable debugging activities. But, CTRL-C is **NOT THE NORMAL WAY TO LEAVE A PROGRAM.**

3.1.3. 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 on the disk cartridge, 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. There may occur a run time overflow, the program may call the standard procedure HALT, or the user may even kill the program by typing CTRL-C on the keyboard. In all of these cases, the operating system first causes the memory contents to be dumped on the *dump files* of the disk cartridge. The dump files may be inspected with program *debug*.

The following types of execution errors may be reported

stopped	*program was killed by CTRL-C*
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-explaining. 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 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. In this case, try to understand the displayed error message. If the memory image has been dumped on the dump files, it is also possible to find the reason for the failure with the debugger.

3.2. 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.

Except for 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.

3.3. 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 9.2.).

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 first 16 characters of) the module names. 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	DK.BufferPool.OBJ
Second file name	DK.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 7.7.) 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 6.7.

4. Things to Know

Leo Geissmann 15.5.82

This chapter provides you with information about different things which are worth knowing if you want to get along with Lilith. 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 behaviour of the programs.

4.1. 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; and so on. 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. This key is always active, even if no keyboard input is awaited. 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.

?

Key to ask a program for a list of all active commands.

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.

4.2. File Names

4.2.1. 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     = MediumName [ "." FileIdent ].
MediumName   = Ident.
FileIdent    = Ident { "." Ident }.
Ident        = Letter { Letter | Digit }.
```

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

MediumName means the device on which a file is allocated. This name must be an identifier with at most 7 characters. It is designed in view of a coming network. To assign a file on the HB disk cartridge of your Lilith computer, the medium name DK must be used.

FileIdent means the name of a file under which it is registered in the name directory of the device. For files on the HB disk cartridge the length of FileIdent is limited to 24 characters.

A file name consisting solely of a MediumName means a temporary file on the device, i.e. the file is not registered in the name directory and will be deleted automatically when it is closed.

4.2.2. 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 to categorize files of specific types (e.g. OBJ for object code files, SYM for symbol files), and there are programs which automatically set the extension, when they generate new files (e.g. the compiler).

4.2.3. 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 keyboard according to following syntax:

```
InputFileName = FileIdent | "#" MediumName [ "." FileIdent ].
```

Normally you want to specify a file on the HB disk cartridge of your Lilith computer and therefore it is more convenient, to type FileIdent only. MediumName DK is then added internally. If you want to specify another MediumName, then you must start with a # character.

```
Harmony.MOD        is accepted as file name    DK.Harmony.MOD
#XY.Color.TEXT     is accepted as file name    XY.Color.TEXT
```

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.

4.3. 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 which is 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.

4.4. The Mouse

An important input device, along with the keyboard, is the *mouse*. It allows *positioning* and *command selection*. The mouse is connected with the keyboard by a cable. It has three *pushbuttons* on its front and a *ball* embedded in its bottom. The ball rotates when the mouse is moved around on the desktop.

To use the mouse, take it in your hand with the middle three fingers in position to press the three pushbuttons and the thumb and little finger apply slight pressure from the sides.

For positioning, e.g. for tracking a cursor, the mouse is moved around on the desktop. The movements are translated by the programs into movements on the screen:

mouse	*screen*
forward	up
backward	down
left	left
right	right

The mouse indicates movements only if it is driven on the table. If it is lifted and set down at another place on the table, no movement is indicated. This allows to reposition the mouse without changing the actual position on the screen.

The pushbuttons on the front of the mouse are pressed for sending commands to programs. They are named according to their position:

left	*middle*	*right*
button	button	button

Generally it may be assumed that a *menu selection* becomes active when the middle button is used. In *scroll bars* usually the left button is used for *scrolling* a text *up*, the right button for *scrolling* a text *down*, and the middle button for *flipping* on the text.

The actual meaning of the mouse buttons is given in the program descriptions. Some programs also display it on the screen.

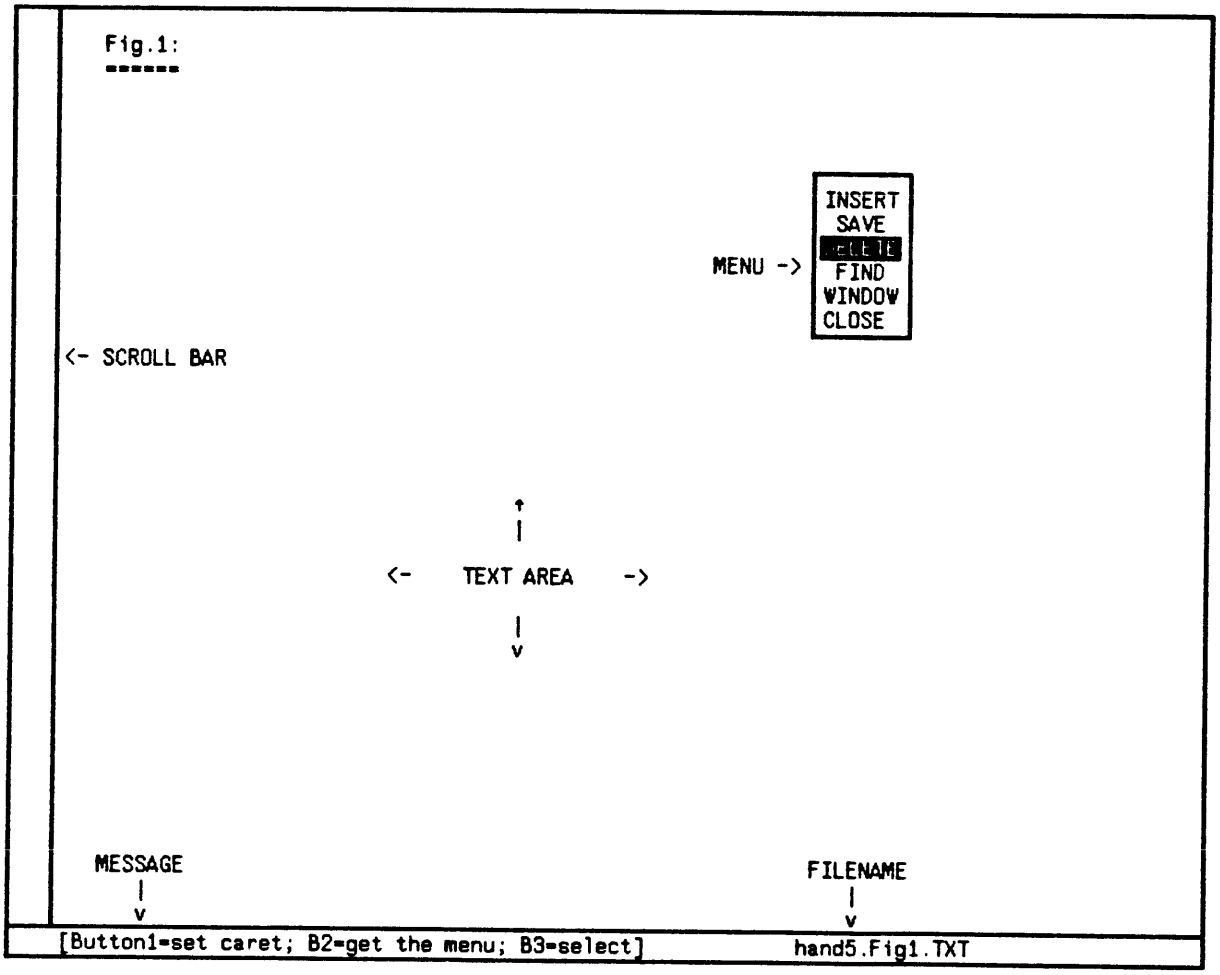
For programmers: The modules *Mouse* and *CursorStuff* support cursor tracking and the handling of the mouse buttons.

5. The Editor

Werner Winiger 29.4.84 (Version I.9)

5.1. Introduction

The Lilith text editor is started by typing its name "edit" followed by RETURN. It presents you the screen image shown in figure 5.1.



The screen is divided into a text area (38 lines of 91 characters), a scroll bar at the left hand side, and a communication bar at the bottom. The text area serves to show you parts of the document you are working on. With the help of the scroll bar you may position this document such that you see the desired sections. In the communication line you get messages from the editor, you have to enter file names and search strings and you see the file name of the document being edited. Not always visible is a menu. It appears as a box containing the keywords of the available commands.

The input devices which allow you to give your commands to the program, are the keyboard and the mouse. Typing on the keyboard usually means that you want to insert new characters into your text. Exceptions are

- when the editor asks for a file name,
- or for a search string,
- or prompts you for an assertion such as `write the document? [Y/N]`.

The mouse is used to move a cursor around the screen (just by moving the mouse on the desk) and to indicate

- where you want to insert text,
- which characters are to be deleted,
- which line should be scrolled to the top of the screen,

and so on. Furthermore, the three buttons of the mouse allow you to specify what action the editor should perform at the position the mouse points to. In the editor's prompts, the left button is referred to as B1, the middle one as B2, and the right button as B3.

5.2. Starting the Editor and Entry of New Text

Let's follow now the procedure of preparing a new document (updating an old one respectively):
If you start the editor it prompts you for a file name. Press RETURN to create a new document. You see now as the first and only "character" of the first line an end-of-file mark in the form of a small (8 by 8 dot) square. The blinking wedge shaped mark to its left is called the *caret* and denotes the location where newly typed text is inserted.
You may now type in your text using

- RETURN to start a new line,
- DEL to erase the character just keyed in,
- TAB to get two blanks,
- CAPS to capitalize the following alphabetic characters.

On the current Lilith keyboard you have to hit CTRL-I or the key labeled ON LINE to get a TAB. The CAPS function is called by pressing CTRL-N or HOME, and HELP, which is mentioned later, is the name of CTRL-S, TAB or "?".

If you reach the bottom of the text-window your document is scrolled up automatically. And after a while you will have a text, just as if you had entered a file name at the beginning of the session instead of RETURN only. So now we can discuss the general case of dealing with an existing document.

When typing the file name you don't have to enter the extension, if you want to edit a Modula program. MOD is the default extension which is appended automatically, if you terminate the file name with a period.

5.3. Positioning the Document

When you enter a text, it is scrolled up such that you always see the last 38 lines of the document; if you get an old document from the disk you automatically see the first page of that text. But you may control which portion of the document you want to be visible through the available text window. [Refer to 5.10. to find out how to split the text-area into several windows.]

Positioning is done with the help of the mouse and the so called scroll bar: Move the mouse to the left until the cursor changes its form from an arrow "pointing to north-west" to a double-arrow. Attention: if you go too far the cursor wraps around, but you may use the same effect to come back again. The special form of the cursor in the scroll bar indicates the special mode of the mouse: it can now be used for positioning.

5.3.1. Scrolling

The cursor's location in the scroll bar corresponds to the text line on the same hight. It is possible to make this line the first (top) of the window (*to scroll up*) by pushing the left button, or to make it the bottom line (*scroll down*) by pushing the right button. Note that you still may adjust the mouse position while the button is pressed because scrolling is not performed until you release it.

As a consequence you may scroll up your document by one line by pressing the left button near the second line from the top, or scroll it down by 37 lines (go to the previous page) by pressing the right button near the top line.

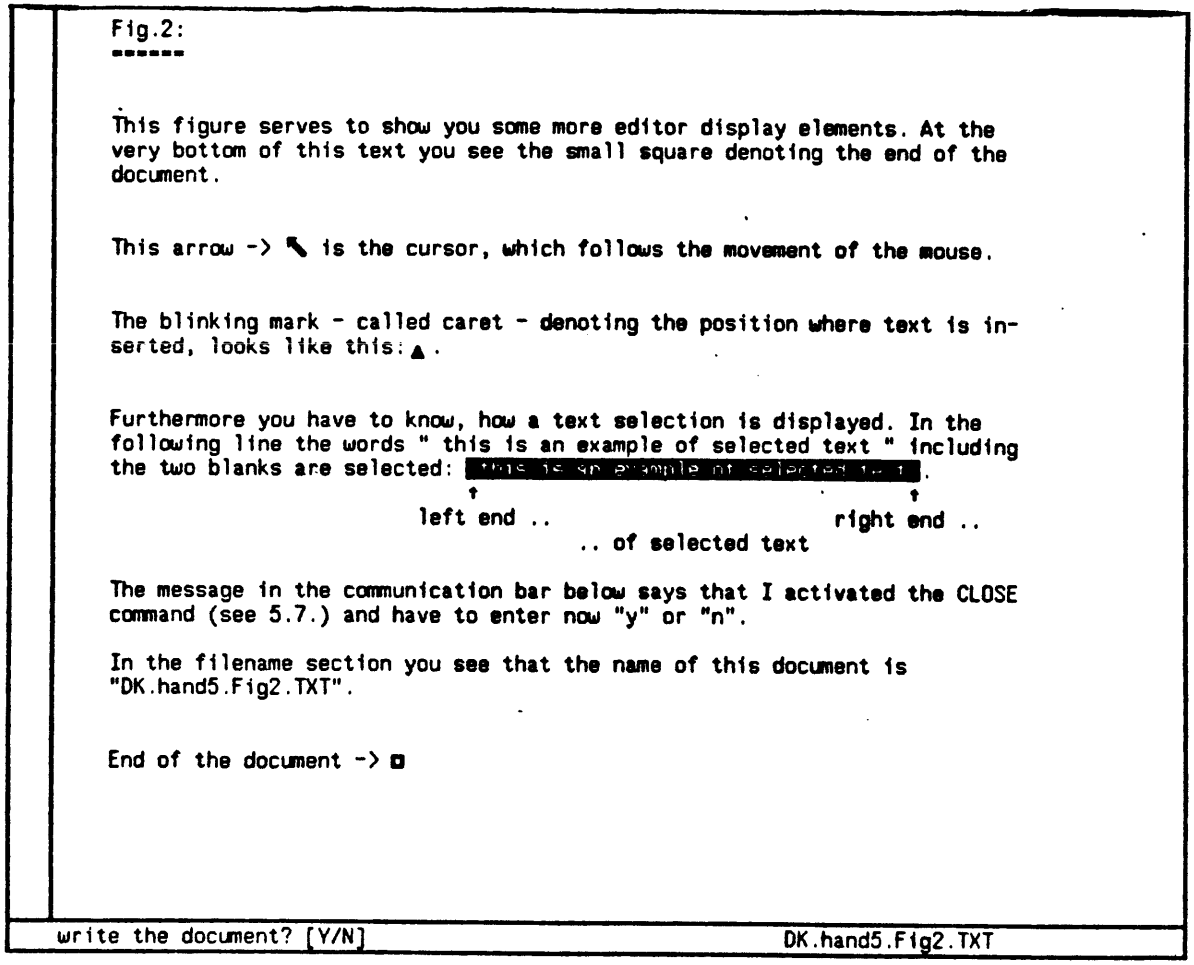
5.3.2. Flipping

By means of scrolling you may reach only the next (or previous) few lines of your document easily from the

current position. But it is also desirable to make longer jumps, like to go to the end of the text directly, or to the beginning, and so on. The editor offers such a facility: *flipping*. If you push the middle button while the cursor is in the scroll bar, the document is repositioned as follows: the number of lines of the text window beside the cursor is considered to represent the whole document. The cursor position within these lines indicates which relative position within the document you want to select. That means: the cursor on the last line indicates a jump to the end of the document whereas the cursor on the 13th line of the 38 indicates a jump to the first third of the document. Again, the jump is executed only after you release the key. [Refer to 5.9. to read how to position the document at a searched string.]

5.4. Insert Characters at Different Locations

Until now we have learned to inspect an old document or to enter a new one just by typing its text sequentially. Now we also want to be able to change a document. First of all, we would like to insert new text not only at the end of the document but at arbitrary locations. The blinking caret denotes the location where typed characters are inserted. Use the mouse to position the caret. Move the arrow to the desired place, press the left button and release it again. If you didn't hit well just try again (and again). It is even allowed to hold down the left button during the movement of the mouse. Then you will see the caret tracking the arrow. And if you have led it to the correct position you release the left button. Now the editor is ready to receive the characters to be inserted. If the caret sits in the middle of a word or a line and you type new text, then the rest of the line is shifted to the right. There is no so-called "paint mode" since deleting is easy enough, as you see from the next section. If a line becomes longer than 91 characters the superfluous ones are wrapped around, and displayed on the following line. But they are viewed only as two lines, they still form one single line with no end-of-line inbetween.



5.5. Activating the Menu and Making a Selection

This editor doesn't expect you to type commands but lets you point to them. Since you would like to see as much text as possible the commands are not always visible but only when you need them. Press the middle button (having the cursor anywhere in the text area) and a list of commands will be displayed near the location of the arrow overwriting temporarily your text. This so-called *menu* will disappear as soon as you release the middle button. So if you want to think or make a choice keep the button depressed. One of the commands is shown in reverse video. In order to control which command should be selected you have to move the mouse. Invoking a command is achieved by releasing the middle button when the associated menu item is inverted. If you wish to release the middle button without causing any action, just move the cursor outside the menu. In any case the menu disappears and the (eventually) hidden information appears again. In case of an activated command it is executed immediately.

For the operations described in the following section you need to point to a specific portion of your document. In the editor's terminology, you have to make a *selection*. Point with the cursor to the first character which should be deleted and press the right button. This character is now shown in reverse video. This visual feedback denotes the selected part of the document. You may select one character or whole words or even several lines: Press the right button at the leftmost character to be selected, hold it down, and move the cursor to the rightmost one. Now the reversed portion tracks the cursor until you release the button. Making a selection is, like moving around the caret, an activity which you may use anytime and as often as you like. The editor lets you *edit* the operands of its operations at your convenience unless you are satisfied by the setup.

5.6. Delete, Move, or Copy Text

I explained how to erase accidentally typed characters, namely with the DEL key. But this is not convenient for more than a few characters. If you wish to correct something longer you have to select this item first with the mouse. Once the text is selected as desired, it may be deleted. This is achieved with the help of the menu. Choose the DELETE command, the selected text gets deleted, and the rest of the text is shifted (and scrolled if necessary) in order to fill the gap.

The string you have deleted is kept in a buffer (until you delete another one). It is possible to reinsert the contents of that buffer anywhere in the document. This may be convenient in the following cases:

- If you have deleted something erroneously.
  You may patch the situation by inserting the buffer again at the same location from where you have deleted it.
- If you like to move some part of your text to another place.
  You just delete it at the source and insert it at the destination
- If you want to copy a string from one place to another.

How, exactly do you insert? INSERT is just another command of the menu. This means that you have to use the middle button to get the menu and then move the mouse to select the insert command. When using the menu, you notice that it is always shown with that command exhibited which you used last. So executing the same command several times is done by simply pushing the middle button shortly - once the desired command has been selected.

And where does the buffer get inserted? Remember the blinking caret denoting the location where typed characters are placed. The same mark is also used here. After deletion of a string, the caret is located by the editor at the position of the resulting gap. So, restoring the string at the same place is very simple. For the purpose of moving text, however, you have to position the caret (as described in 5.4.) prior to activation of the insert command.

Let's consider now copying a portion of a document. Having in mind only the delete and the insert facility it would be necessary to delete a string and then to insert it twice: once at the origin and once at the destination. To accomodate this, there is an abbreviating command in the menu: SAVE. It takes the selected text (which you defined using the right button) and stores it in the buffer as if it had been deleted. But the text remains in the document and may now be copied with the help of the insert command. There is an even easier way to copy a string which may be used if the destination is visible on the screen: set the

caret to the destination and select the text to be copied; now activate INSERT. The string will implicitly be saved into the buffer and also inserted.

5.7. Termination of an Editing Session

If you have entered a new document or updated an old one, you sometimes wish to stop your work. For that purpose the editor provides the CLOSE command. It prompts you with the question: `write the document? [Y/N]`. The *No* case is chosen if you want to exit the editor without saving the document on the disk (e.g. if you used the editor to inspect a file only). To make sure that no work is lost this way the program asks `exit without writing the document? [Y/N]`. And the editor stops only if you insist by typing a "y".

If you have confirmed that you wish the document being saved, the editor displays the message `what filename? [RETURN=backup input file]` and the file name of the document you worked with. (This name is by default DK.Temp.MOD if you entered a new document.) You have now the choice to give the document a new name (what you normally will do if it is a new one) or to store the current version with the same name as the old one; in this case the old version is renamed: the extension of its file name is changed to BAK. In any case you will not lose the inputfile (if there is any) of the edit process, so you may retrieve your 'edited' data again! Note, however, that when renaming the old document to ... .BAK the file with this name (the grandfather of your current document) is deleted.

According to the principle of ***not typing again what you have on your display already*** the editor uses the selected text as file name if there is a selection when you terminate the session. Reading the file name from the selection stops, however, when encountering a terminator such as you get control again before the file is actually written: complete or correct the name from the keyboard and confirm with RETURN that you are done.

5.8. Rescue from Abnormal Termination and Other Errors

From 5.7. it follows that our new, edited document is not written onto the disk while you work with it. The document is indeed represented only by a data structure in main memory. This makes the case of a hard- or software error dangerous. Is all your work lost in such a case? Of course not. You may start the editor again and watch what happens: It remembers all you did and repeats the whole editing process again. You may stop this so-called *replay* by typing any key, continue by typing some key again, or terminate the interrupted replay process and switch to normal editing with ESC. This feature is implemented by writing all commands which you give either with the mouse or the keyboard to a file (Edit.I10.RPL) and deleting this file if you terminate editing normally. Hint: it is wise to save the document onto the disk after a successful replay.

Delete Edit.I11.RPL ← edit replay file if edit is hung up

5.9. Searching

The FIND command allows you to search for a string which you entered from the keyboard or which you selected on the screen with the mouse. When activating the command you have the following three possibilities:

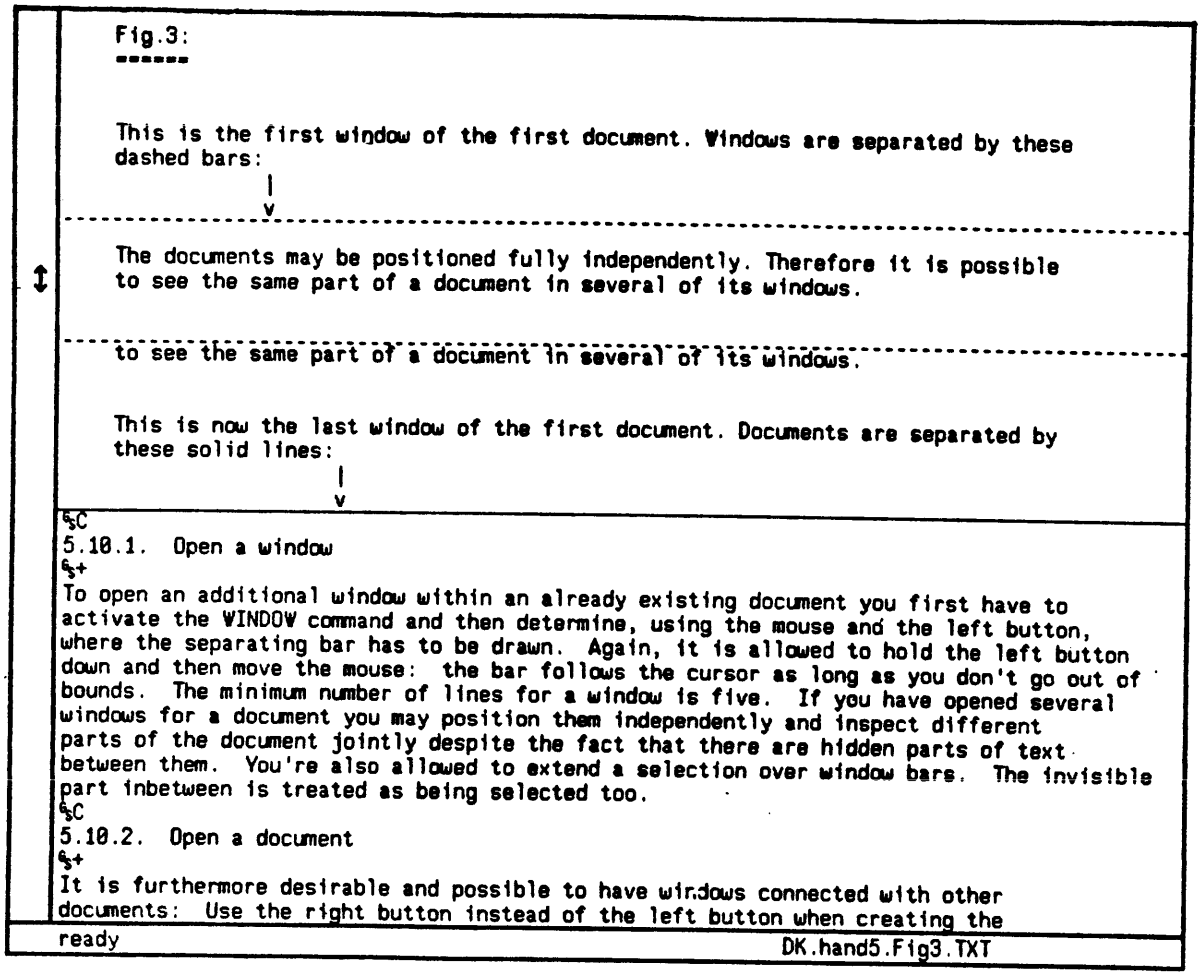
> You may search for the same string as used before: Press the left button. This is convenient for finding all occurrences of a string.
>
> You want to search for another string of which you have a copy somewhere on the screen: Select this string, by using the right button, before activating the find command. This way you don't have to type the searched string.
>
> You enter the search string from the keyboard, terminating it with RETURN. The editor finds the next occurrence of the string (not containing RETURN).

If you forgot what the previous search string was you may ask the editor by typing the HELP key instead of a new search string. Searching is performed sequentially forward through the document starting from the position of the caret. If the string can't be found a message is displayed; else the document is positioned

such as the line containing the string is visible in the middle of the text window. The caret is positioned right after the found string. Therefore, it may be used as starting point to search for the next occurrence.

5.10. Working with Windows

In the previous chapters we have considered the whole text area as one window through which you may inspect 38 adjacent lines of your document. The editor features, however, multiple windows for the same document and multiple documents in parallel. Fig. 3 shows a screen with two documents of which the first has three windows.



5.10.1. Open a Window

To open an additional window within an already existing document you first have to activate the WINDOW command and then determine, using the mouse and the left button, where the separating bar has to be drawn. Again, it is allowed to hold the left button down and then move the mouse: the bar follows the cursor as long as you don't go out of bounds. A window must contain at least five lines. If you have opened several windows for a document you may position them independently, and inspect different parts of the document jointly despite the fact that there are hidden parts of text between them. You're also allowed to extend a selection over window bars. The invisible part inbetween is treated as being selected too.

5.10.2. Open a Document

It is furthermore desirable and possible to have windows connected with other documents: Use the right button instead of the left button when creating the window. The bar, which is drawn then, is not dashed

like for a normal window but it is a solid line. And the program asks you for a file name, just like at the beginning (where you implicitly created the first document). For the second and the subsequent documents the default extension is no longer MOD but LST. And there is also a default name, namely the name of the document opened before. The editor has the same conventions for entering file names as described in section 4.2.3. Having a string selected when opening a new document causes these characters to be interpreted as the file name.

5.10.3. Change the Size of a Window

In the window command, the left button has another meaning if it is pressed while the cursor is located above the last line of a window. The cursor then picks up the associated bar and moves it up or down following the mouse. This implies that you may change the size of an existing window.

5.10.4. Close a Window or a Document

If you want to remove a window you have to activate the CLOSE command. If there is more than one window the editor prompts you `close which window? [B1]`. You have to denote the window by moving the cursor inside it and pressing the left button.

Closing the uppermost window of a document, even if there are some more open, is associated with closing the whole document. You have then to perform the necessary dialog as described in 5.7. in order to define whether that document should be stored on the disk and, if so, which name it should have. If the document to be closed is not the topmost one then the editor doesn't doublecheck in case you indicate closing without writing.

Care has to be taken if there are documents which have, besides the extension, the same file name (e.g. x.MOD and x.DEF). They cannot all be renamed to get the extension BAK. Therefore, the editor uses the extension BAK for the first which you back up, BAL for the second, and so on.

Closing the topmost document, even if there are some more open, is associated with leaving the editor. If you want to do this without writing the document you have to assert your intention. Writing is only applicable to the document you close explicitly. So, if there are further documents when you close the topmost one, they are implicitly closed too, but without writing.

After you have closed a document using the default backup scheme, which renamed the inputfile to ... .BAK, you must not try to do a replay in case of a crash without renaming the files to the names they had at the beginning of the interrupted editing session.

5.11. Accelerators

Editors are usually too simple and not handsome enough for experts *or* too complex for beginners or casual users. The method to delete, copy, or move text as described in 5.6. is intended to be easily understandable but it requires invoking the menu once or twice after the selection is made and the caret positioned. For skilled users, the editor features a faster way to achieve the same results. While holding down the right button to define a selection you may also associate a *type* with the selection. There are delete-, copy-, and move-selections. If you release the right button and the selection has such a type, the editor performs the appropriate operation: with a delete-selection the selected text just disappears, with a copy-selection it is copied to the location of the caret, and with a move-selection it is deleted at the source and inserted at the destination (denoted by the caret again).

There are two ways to define the type of a selection: from the keyboard or with mouse buttons. If you prefer to use your left hand and the keyboard proceed as follows: Hit "d" for *delete*, "c" for *copy*, or "s" (because of its neighborhood to d and c) for *move*. You will receive a visual feedback. A delete-selection has a dashed bottom edge, a copy-selection a dashed top, and a move-selection has both (because you delete and copy). The selection type is a toggle, so typing "d" twice turns on and off again the delete-selection. Furthermore you are allowed to switch from one type to another. Hitting "d" followed by "c" produces a copy-selection. Remember to keep depressing the right button all the time. If you prefer to use the left and the middle button of the mouse instead you have to know the following encoding: the left button corresponds to the delete-selection, the middle one to the copy-selection, and both together

to the move-selection.

But before you can do some practising, your user profile must be edited in order to direct the editor to support one way of getting selection types or the other. Read the following section on the function and format of the user profile.

5.12. User Profile and User Guidance

A key word of today's editors is user tailorability. The Lilith editor has a few functions which may be turned on or off via the file *User.Profile*. This text file is mainly a list of options which should be active for a given user. It may be edited as desired and when starting the editor the next time the options will be set accordingly. If no profile is available the editor generates a default file containing:

```
"Editor"
'Version'           I.10
'SelectionFeedBack' inverted
'Caret'             after insertion
'Umlaut'            per default
```

The meaning of these entries is that you are using version I.10 of the editor, that you want to see selected text in reverse video, that you would like the caret to be moved behind an inserted (or copied or moved) string, and that the editor should start in the umlaut mode. If you don't like a particular option (e.g. those bright rectangles for selection feedback) you may delete the associated entry from the user profile and the editor will behave the way you specify. Note, however, that it is not possible to get version 1.8 of the editor running by editing the version number! This entry merely serves to indicate to a newly released editor that you just have started it the first time and that it should give you some advice on the differences between this and the previous version. Section 5.13. tells you more about umlauts. The other options will be discussed here.

The alternative to selection feedback by inverse video is a triangular mark at the left end of the selection and a similar mark at the right end thus avoiding large differences in the brightness level of your screen. As a side effect it is no longer possible to have the selection types which are described in 5.11.

Not moving the caret behind inserted portions means leaving it where it is, in front of them. This location is at least visible on the screen in all cases, whereas the caret would disappear in the other mode if you insert more than n lines having the insertion point located less than n lines from the bottom of the window. In such a case the editor moves the caret only to the end of the last line of the window!

Further user profile entries known by the editor are:

```
'SelectionTypes'    from keyboard
'SelectionTypes'    from buttons
'HardCopy'          enabled
'Font'              GACHA20
```

The function of the first two has already been explained in 5.11. You are allowed to have both options enabled together.

The hardcopy entry enables the CTRL-P key. This command writes the current bitmap on the file Editi.PICT, i = 0,1,... Such pictures may be processed by hardcopy programs. Pressing the middle and then also the left button is equivalent to CTRL-P.

The font feature is mainly for demonstration purposes. The editor may be used with any fixed-pitch font. Having a larger font but fewer lines and less characters is useful if you want to project the screen image and still get readable information. As a side effect of changing the font, the editor assumes selection feedback by reverse video and ignores selection types.

User guidance is tentatively achieved by the dialog with the user through the message window which is done in a somewhat systematic manner: There are messages indicating the state of the system during long lasting operations (like FIND). There are messages prompting the user for an action (like entering a filename). The possible options for such a situation are enumerated in square brackets. There are error

messages informing the user about an action the system cannot handle. And finally there are messages indicating the last operation performed. This last type of messages appears only on request (to be given with the HELP key).

The other general purpose key is ESC. ESC lets you escape from almost any situation. It cancels the selection or the selection type, it cancels entering a file name or a search string, and it lets you also return from opening or closing a document.

5.13. Special Characters

The editor supports six umlaut characters: Ä, Ö, Ü, ä, ö, and ü. The following table shows their encoding on a disk file and by which keys they may be generated:

Ä	200B	SHIFT-"@"
Ö	201B	SHIFT-"↑"
Ü	202B	SHIFT-"\"
ä	203B	"@"
ö	204B	"↑"
ü	205B	"\"

But as you see it is also possible, to generate the ASCII characters `, ~, |, @, ↑, and \. The three keys "@", "↑", and "\" may be switched to ASCII mode back and forth with CTRL-T or the key labeled FORMAT. You may control the mode in which the editor should start by an entry in the user profile.

Fig.4: Representation of the Nonprintable ASCII Characters in the Editor Font

0c	CTRL @	null	
1c	CTRL A	start of heading	
2c	CTRL B	start of text	
3c	CTRL C	end of text	
4c	CTRL D	end of tansmission	
5c	CTRL E	enquiry	
6c	CTRL F	acknowledge	
7c	CTRL G	bell	
10c	CTRL H	backspace	
11c	CTRL I	horizontal tabulation	
12c	CTRL J	line feed	(interpreted by the editor)
13c	CTRL K	vertical tabulation	
14c	CTRL L	form feed	
15c	CTRL ↑	carriage return	
16c	CTRL N	shift out	
17c	CTRL O	shift in	
20c	CTRL P	data link escape	
21c	CTRL Q	device control 1	
22c	CTRL R	device control 2	
23c	CTRL S	device control 3	
24c	CTRL T	device control 4	
25c	CTRL U	negative acknowledge	
26c	CTRL V	synchronous idle	
27c	CTRL W	end of transmission block	
30c	CTRL X	cancel line	
31c	CTRL Y	end of medium	
32c	CTRL Z	substitute	
33c	CTRL [	escape	
35c	CTRL]	group separator	
36c	CTRL M	record separator	(interpreted by the editor)
37c	CTRL _	unit separator	
177c	DEL	delete	
34c	CTRL \	file separator	

ready DK.hand5.Fig4.TXT

6. Utility Programs

15.5.82

This chapter gives an overview of some utility programs which provide the most important services on Lilith. Utility programs are usually stored on the disk cartridge (medium name DK). The file name is derived from the program name, beginning with the prefix SYS and ending with the extension OBJ. Programs are called for execution by their name.

Program name	`directory`
File name	`DK.SYS.directory.OBJ`

List of the Programs

directory	Give a list of file names	6.1.
delete	Remove files from the disk	6.2.
protect	Set protection on files	6.2.
unprotect	Cancel protection on files	6.2.
copy	Make copies of the file contents	6.3.
rename	Change file names	6.3.
list	List a text file on screen	6.4.
inspect	Inspect the contents of a file	6.5.
xref	Generate a reference list of a text file	6.6.
link	Link separate modules to a program	6.7.
decode	Disassembler of object files	6.8.
layout	Tool for screen layout	6.9.
hermes	Transfer files to another Lilith	6.10.
hpcopy	Transfer files to a Hewlett Packard terminal	6.11.
backup	Save files on an Apple diskette	6.12.
restore	Restore files from an Apple diskette	6.12.
boot	Bootstrap of the computer	6.13.
altboot	Bootstrap the computer with the alternate boot file	6.13.

Most programs are operating on files, and they therefore will prompt for a *file name* and probably will also accept *program options*. The syntax of file names and program options is given in chapter 4.

There are some programs which may operate on a group of files. For this purpose the accepted file name may contain *asterisk* * and *percent* % characters as *wildcard symbols*. An asterisk stands for any (including the empty) sequence of legal characters (letters, digits, periods), a percent for just one legal character. This allows to select all file names that match the pseudo file name.

Pseudo file name	`M*2.TXT`
Matching file names	`M2.TXT`
	`Modula2.TXT`
	`Modula2.Version2.TXT`

Pseudo file name	`M%.TXT`
Matching file names	`M1.TXT`
	`M2.TXT`

Pseudo file name	`Mouse.*`
Matching file names	`Mouse.Head`
	`Mouse.Tail`
	`Mouse.old.Head`

An asterisk enclosed by two periods would match a single period and a leading or trailing asterisk with a separating period would match the empty sequence. Therefore `Mouse` would also be a matching name in the third example.

6.1. directory

Leo Geissmann 15.5.82

The program *directory* shows directory information of the selected files. It accepts a file name with wildcard symbols. Hitting the **RETURN** key instead of specifying a name means that all files on the cartridge should be selected. For all files with a name matching the specified name, the program displays directory information in the following sequence:

- Protection symbol: A # if the file is protected
- File name
- Length in blocks (1 block = 1 K Word)
- Date of creation or last modification

Example

```
# Mouse.old.Head    6   22.Sep.80
  Mouse.Head        7   12.Jan.81
  Mouse.Tail        3   30.Sep.80
  Mouse             1   17.Feb.81
```

Before terminating, the program displays a summary

Number of listed files
Number of blocks used by the files
Number of free blocks on disk cartridge

If the information fills more than one screen page, the string ... is displayed and the program is waiting until any key is pressed on the keyboard (ESC would stop the program immediately).

With program option *EXtra*, supplementary information is displayed for each file. In this case the information sequence is as follows

- Protection symbol: A # if the file is protected
- File name
- Length in blocks (1 block = 1 K Word)
- Exact length in Bytes
- Number of the directory entry
- Date of creation or last modification
- If modified: Modification version
- If modified: Date of creation

To get the directory information on a file instead on screen, the program option *Output* must be specified. In this case the program asks for a file name and writes the information on a file with this name.

```
*directory
 directory> Mouse.*/output
 output file> Mouse.Directory
```

Program Options

Alpha
Information is listed in the alphabetic order of the file names.

NOAlpha
Information is listed in the order of the directory. *Default.*

Equal
Capital and lower case letters are treated as equal.

NOEqual

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

Output

Information is listed on an output file.

Page

Information displayed on screen page by page. A key must be pressed after each page. *Default.*

Scroll

Information displayed on screen continuously. After a key is pressed, output is stopped until a second key is pressed.

SHort

Short information. Only file names are listed.

NORMal

Normal information, as described above. *Default.*

EXtra

With supplementary information, i.e. exact length in bytes, the file number and modification information.

Capitals mark the abbreviations of the option values.

6.2. delete, protect, and unprotect

Leo Geissmann 15.5.82

The program *delete* allows to remove the selected files, *protect* and *unprotect* handle the protection of the files. In the current implementation of the file system, protection means that a file cannot be changed.

The programs accept a file name with wildcard symbols. For all files with a matching name on the disk cartridge the programs display the file name and prompt for an assertion (**y** for *yes*; **n** or **RETURN** for *no*) before doing the desired operation.

```
Mouse.Head   delete? yes
Mouse.Tail   delete? no
```

Each program skips those files on which the operation cannot be applied, i.e. protected files are skipped by delete and protect, and unprotected files are skipped by unprotect.

Program Options

Query
Operation on file must be asserted. *Default.*

NOQuery
Operation on file without assertion.

Equal
Capital and lower case letters are treated as equal.

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

Capitals mark the abbreviations of the option values.

6.3. copy and rename

Leo Geissmann 15.5.82

The program *copy* handles the copying of files, *rename* the change of file names.

Two file names with wildcard symbols are accepted by the programs: a *from-name* and a *to-name*. The from-name specifies the selected files, to-name the corresponding new names. In to-name only asterisks are accepted as wildcard symbols, and these must be separated by periods from other identifiers.

The *compatibility* of from-name and to-name is checked and an error message is displayed, if a projection is impossible. Be aware that the projection of the names *is not always clear* and that the programs might come to an *interpretation which differs from the intended one.*

For all files with a name matching from-name, this name and the new generated name are displayed and the programs prompt for an assertion (**y** for *yes*; **n** or **RETURN** for *no*) before copying or renaming the file. If a file with the new generated name already exists, the replacement of the existing file must be asserted.

```
Mouse.Head  to Mice.Head  copy? yes  replace? yes
Mouse.Tail  to Mice.Tail  copy? no
```

Program Options

Query
Operation on file must be asserted. *Default.*

NOQuery
Operation on file without assertion.

Equal
Capital and lower case letters are treated as equal.

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

Replace
Existing files with new name are replaced without assertion. *Default, if Query is specified.*

NOReplace
Existing files with new name must not be replaced. *Default, if NOQuery is specified.*

Capitals mark the abbreviations of the option values.

6.4. list

Werner Winiger 7.5.82

Utility to list a textfile on your system's display.

The program asks for a filename. Default extension is LST. As options you may specify whether you want to see the file page after page or scrolled up after each line. Furthermore you may choose a different font.

Example:

```
*list
 list> Example.MOD/P
MODULE Example;
    ...
```

In the scroll mode the program may be interrupted temporarily by typing any key. The ESC-key then will terminate it, any other key resumes displaying of the file.

With the HELP-key ("?", CTRL-S, or TAB) instead of a filename you may ask the program for the available options.

The following options are available:

Paging
: The file is displayed pagewise. The program writes "..." on the bottom line of the display and waits until you press a key. ESC in this situation terminates the program.

Font
: The program asks you for the filename of a font (default extension: FONT). The file is displayed using the specified character style.

6.5. inspect

Peter Lamb 15.5.82

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. 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.

6.6. xref

Leo Geissmann 15.5.82

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).

6.7. link

Svend Erik Knudsen 15.5.82

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 on an *object file*. An object file may be loaded by the loader of Medos-2 directly.

As a program usually consists of several separate modules, the loader has to read the code of the modules from several object files which are searched according to a *default strategy*. On the one hand, this is time consuming because several files must be searched, on the other hand, it could be useful to subsitute a module from a file with a non-default name. These are some reasons for having a linker program.

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 first 16 characters of) the module name. 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	`DK.Options.OBJ`
Second default file name	`DK.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: DK.LIB.NameSearch.OBJ            second default file name
      NameSearch
 Options: DK.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> DK.Options.OBJ                       default file name
     Options       ?  yes
     FileNames     ?  no                      module not linked from this file
FileNames> FileNames.own.OBJ
     FileNames     ?  yes
```

6.8. decode

Christian Jacobi 10.5.82

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

6.9. layout

Christian Jacobi 20.2.82

Program *layout* is used to design the screen layout for programs which use windows.

The program allows creation of windows and to query interactively the coordinates of the windows. When the program is started, a window *LAYOUT* is created. Other windows can be opened, the windows can be moved and changed. At any time it is possible to write the coordinates of the windows.

The program is interactively driven with the mouse. Pressing a mouse button calls either the window menu or the (standard) background menu, depending on where the cursor points to. To exit the program use the exit command of the background menu.

When you type on the keyboard, the text is written in the *active* window. That window is active, where a mouse button was clicked most recently.

Window Menu Commands

alfa
: Writes the alphabet into the window.

clear
: Clears the window.

border
: Writes the outside (border) coordinates of the window into the window.

inside
: Writes the coordinates of the inside area of the window into the window.

layout
: Asks for a filename, creates a file and writes the coordinates of all windows to the file.

open
: Opens another window. Type the header line of the window, terminated by RETURN. Point the diagonal of the new window with the mouse.

Background Menu Commands

This is the standard background menu. It allows to delete, to change, to move ... windows, to call the command interpreter and to exit. This background menu is described with the module WindowDialogue.

Format of the Layout File

Every window gets a 3 line entry. The first line shows the title of the window. The second line shows the outer coordinates of the window in decimal notation. The third line shows again the outer coordinates of the window, but in octal notation.

Example

```
procedure call chain
  5 255 537 147
  5 377 1031 223
```

6.10. hermes

Jirka Hoppe 14.5.82

The program *hermes* transfers files between two Lilith computers connected by a V24 (RS 232) cable.

To transfer a file do the following steps:

1/ Connect both computers with a cable. Be sure that the RS232 switch is in the 'CPU-PORT' position (up) and that the speed switch on both computers is set to the same speed (recommended is 9600 baud). This is the normal (default) situation, preset by the hardware technician.

2/ Start the *hermes* program on both computers.

3/ The both programs ask you: `are you a master?` Answer with **y** on the computer where you will give the transfer commands (master), answer with **n** on the other computer (slave). It is recommended to start first the slave computer and later the master. The master will respond with `opening line...`. When the connection with the slave is established, a message `line opened` will be displayed.

4/ The master now asks you for the names of the files that should be transferred. The syntax of a file name is the standard syntax with a prefix allowing to distinguish between both computers. The prefix *ME:* identifies the master, the prefix *YOU:* identifies the slave. Type **M** for *ME:*; type **Y** for *YOU:*.

If you try to write a file that already exists, the program asks you if you would like to replace the old file. Answer with **y** to replace the file or with **n** to abandon this file transmission. This query may be turned off by an option *n (no query)* following the *from* file.

When specifying the name of the **to:** file you may type a RETURN only. In that case the name of the **from:** file will be taken as default.

Examples

```
from>ME:MyMoney.All<cr>
to>YOU:debt<cr>
```

This transfers a file 'MyMoney.All' from the master computer to the file 'debt' on the slave computer. If the file already exists on the master computer, you will be asked if the file should be overwritten.

```
from>YOU:hundred.francs/n<cr>
to>ME:<cr>
```

transfers a file 'hundred.francs' on the slave computer to the file 'hundred.francs' on the master computer. If the file already exists on the master computer, it will be overwritten without any notice.

5/ At the end of the transmission you could exit from the master with typing ESC. The program asks you in which way you like to exit. Type:

K or **ESC** - to exit and turn off the slave process

S - to exit but let the slave process active to reopen the session next time the master starts hermes.

R - to return back to the hermes program

If the slave process is turned off by the master, a message `the transmission is finished, you may use your machine again` will be displayed on the his screen.

The slave must be killed by *CRTL-C*, if he is not turned off by the master.

Remarks

Since the program works without interrupts, there may arise a situation, where the protocol gets out of the synchronization. If the program reports the `lost line` or if the line could not be opened at the beginning, please restart both programs. In the normal case checksum of each packet is computed and packets with any kind of troubles are retransmitted. The transmission speed is about 1Kword in 2.6 seconds.

6.11. hpcopy

Werner Winiger 6.5.82

Utility to transfer a textfile from/to the tape cartridge of a Hewlett Packard 26XY terminal.

The program checks whether a terminal is connected to your system's V24 connector and whether it is ready to transmit data.

If so, the program asks for a file name. Default extension is MOD. The source device of the file is indicated (as for the program hermes) with a prefix to the file name. The prefix *ME:* stands for your Lilith (i.e. the file is transferred from disk to tape), *YOU:* denotes the terminal (i.e. the file is copied from the tape cartridge). You only have to type the first character of the prefix (lower case), the rest will be supplied by the program.

Examples:

```
*hpcopy
 copy from> ME:Example.MOD
        to> YOU:
 done
*

*hpcopy
 copy from> YOU:
        to> ME:Example.TXT
 done
*
```

The program doesn't know about positioning the tape. This means that the cartridge has to be positioned appropriately before transferring the file. After copying a file, however, hpcopy writes a file mark on the tape.

6.12. backup and restore

Richard Ohran 21.5.82

Programs for *saving* and *restoring* Lilith files on an Apple diskette.

The program `backup` allows you to backup your files on an Apple mini floppy diskette; the program `restore` loads them from an Apple floppy back to your Lilith.

To back up your files you first have to prepare a floppy with the following files (you may once have prepared a master copy):

```
SYSTEM.PASCAL
SYSTEM.APPLE
SYSTEM.MISCINFO
T.CODE
```

Maybe you would also like to include the file

```
SYSTEM.FILER
```

Now boot your Apple diskette, set the prefix to the name of your diskette and execute the Apple program `T`. The Apple computer is now ready for both backup and restore.

Start either a backup if you want to save your file or restore if you want to reload your files. The programs will ask you for two file names: a *from-name* and a *to-name.* In the program backup, the from-name specifies the file on your Lilith computer that should be saved and the to-name specifies the name of the copy on an Apple diskette. In the program restore, the from-name specifies the Apple file and the to-name is the name of the reloaded file on your Lilith.

If you type just RETURN to specify the to-name, the from-name will be taken as the default for the to-name.

To exit the program type a single RETURN when the program asks for the from-name.

Example

```
*backup
 from>MyImportantFile.TXT
 to>MyImpFil
```

Saves the Lilith file `MyImportantFile.TXT` on the Apple diskette under the name `MyImpFil`.

```
*restore
 from>MyImpFil
 to> <RETURN>
```

Loads the Apple file `MyImpFil` to Lilith under the name `MyImpFil`.

Caution

The Apple program `T` requires correct interactive input. The program crashes if any error occurs. This especially, if the file names on the Apple are too long (at least 11 characters are allowed for an identifier within the file name), or if the Apple diskette gets full.

The Apple converts lower case letters in the file name to capitals.

File Format on the Apple

On the Apple diskette the files are stored as data files. The first 4 bytes on the files are used to encode the length of the file. This header is followed by the saved information.

6.13. boot and altboot

Svend Erik Knudsen 15.5.82

The programs *boot* and *altboot* initiate a *soft boot* of Medos-2 on Lilith.

The program *boot* initiates a bootstrap from the normal boot file *(PC.BootFile)*, whereas the program *altboot* initiates a bootstrap from the alternate boot file *(PC.BootFile.Back)*. The programs need no input from the keyboard.

Warning

The devices connected to Lilith are not reset by a *soft boot*, as this is not generally possible from software. Whenever you have to boot and you are not sure, whether or not a device using direct memory access is running, *press the reset button* on the rear of the keyboard *and hit the space bar* or type CTRL-A.

7. The Compiler

Leo Geissmann 15.5.82

This chapter describes the use of the Modula-2 compiler. For the language definition refer to the Modula-2 manual [1]. Lilith specific language features are mentioned in chapter 12 of this handbook.

7.1. 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. Default extension is **MOD**.

listing file
Compiler output file with list of the compiled unit. Assigned extension is **LST**.

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

reference file
Compiler output file with debugger information, generated during compilation of a program module. 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 the 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.
```

7.2. 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 DK.Prog1.MOD is accepted
p1
p2                              the succession of the activated
p3                              compiler passes is indicated
p4
lister
end compilation
*
```

Default device is **DK** and 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
*
```

7.3. Compilation of a Definition Module

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

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

7.4. 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 7.6.), 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`
First file name	`DK.Storage.SYM`
Second file name	`DK.LIB.Storage.SYM`

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

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

7.5. 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.

7.6. 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 4.).

The compiler accepts the option values:

LISTing
: A listing file must be generated. *Default.*

Nolisting
: No listing file must be generated.

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

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

SMall
: A small program is compiled. Work files of the compiler may be allocated in memory. *Default.*

LArge

A large program is compiled. Work files of the compiler must be allocated on the disk.

Version

Compiler has to display information about the running version, e.g. processor and operating system flags.

Capitals mark the abbreviations of the option values.

7.7. 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
```

7.8. 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.

7.9. 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. Besides of the *main module*, the module which is called to be executed and therefore constitutes the main program, all modules which are directly or indirectly imported are 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 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 3.

Although it is not necessary to link programs explicitly, it is sometimes more appropriate to *previously* collect all modules, which are to be loaded together, and to write them on the same file. This will accelerate the loading. Linking is provided by the program *link* (see chapter 6.7.).

Medos-2 also supports some kind of a *program stack*. A program may call another program, which will be executed on the top of the calling program. After termination of the called program, control will be returned to the calling program. For more details refer to the library module *Program* (see chapter 9.2.).

7.10. Value Ranges of the Standard Types

The value ranges of the Modula-2 standard types on Lilith are defined according to the word size of 16 bit.

`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, e.g. `-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.

7.11. Differences and Restrictions

For the implementation of Modula-2 on Lilith some differences and restrictions must be considered.

Constants expressions with real numbers

Constants 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 *must 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], except the value 0.

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.

7.12. 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

8. The Debugger

Christian Jacobi 15.5.82

If an error occurs while executing a program, the operating system will make a complete dump of the main memory to the disk. The debugger is an aid to inspect this dumpfile.

Even though the debugger seems to have a large number of commands, its use is mostly self-explaining. The commands are well structured.

8.1. Starting the Debugger

Type in `debug` to start the debugger. The debugger asks for the dumpfile to analyze. Type RETURN to take the default dump files, otherwise the filename. If the default filenames are not used, some debugger versions also ask for the file where the upper memory bank is dumped. The debugger asks if the default files for reference and listing information should be taken. (RETURN or blank is interpreted as yes).

Next, the debugger constructs its internal data structures. If the default files are not taken, it may ask for several filenames. To use the default files type RETURN. Now the debugger is ready to start the interactive inspection of the saved dump.

8.2. General Debugging Dialog

The debugger shows several windows, having different active commands. The mouse is used to control the debugger. Clicking the middle button of the mouse within a window will show the active commands inside that window with a *menu*. To select a command, move the cursor to that command in the menu and release the button. The left button is usually used to execute the most important of the active commands immediately (i.e. without going through the menu).

Some windows have a scroll bar at the left. In the scroll bar the buttons have different meanings:

The left button scrolls the current line to the top; the right button scrolls the current line to the bottom and the middle button is used for relative positioning.

Most debugging interactions are done at the source level. Additionally, the debugger gives some machine-level information. This information allows debugging to be continued in cases where the source level information is not (now) complete. The debugger should be useful not only for debugging simple programs, but also for system programs, where binary information, process descriptors and other low-level data is used.

8.3. The Debugger Windows and Their Commands

8.3.1. The Procedure Chain Window

The procedure chain window shows the calling sequence of active procedures.

Commands

`list:`

Shows the listing of a program unit (a procedure or a module) in the program window. After giving the list command, select a procedure by pointing at it with the mouse.

`data:`

Shows the data of a program unit in the data window. After giving the data command, select a procedure by pointing at it with the mouse.

The left button combines the two commands for listing and data. The data and the listing of the selected procedure are shown in their respective windows.

Each procedure in the procedure chain is described by one line. The procedure name is followed by the name of the module, the actual program counter value and the base address of the procedure frame. For modules with a missing reference file, the procedure name is replaced by its internal number.

Some Procedures Have a Mark:

`<==` This is the most recent procedure. It is the top procedure in the window.

`xxx` This procedure is the main procedure of the process, usually the main module initialization code. It is the bottom procedure in the window.

`pv` This procedure has been called as a procedure variable.

`...` Length of procedure chain exceeds maximum length displayed by the debugger.

8.3.2. The Program Window

The program window is used for inspection of the text of the program to be debugged. The window shows the program text prefixed with a line number and an octal location counter. Lines which do not fit into the window are clipped.

Commands

`pc:`
Asks for an octal number and searches for this number as program counter in the program.

`line:`
Asks for a line number and searches for that line.

If default filenames are not used, the debugger asks for the listing file every time a new module is shown in the program window. A RETURN input will use the default listing file whose name corresponds to the module name. Press ESC if no listing file exists.

8.3.3. The Data Window

The data window shows the data of the inspected modules.

Commands

`select:`
This is the default command which is executed if the left mouse button is pressed inside the data window.
Applied to a data element: Shows its value (and address) in the memory window.
Applied to a local module: Shows the contents of that module in the data window.
Applied to the shown program unit itself: If the program unit is a module and is local to another one, the embedding program unit is shown.

`father:`
This command is only active when a local module is shown. Executing it causes the embedding program unit to be shown.

If default filenames are not used, the debugger asks for the reference file when ever a new module is referenced for the first time in the data window. A RETURN input will use the default reference file, whose name is equal to the module name. Press ESC if no reference file exists.

The first line in the window shows the name of the module or procedure currently inspected, together with the name of the embedding program unit (filename for global modules). After this, the local modules and variables are shown. Simple variables are shown with name, value, type and address. For structured variables, the value is replaced by the size. The actual value can be inspected in the memory window. The next version of the debugger will show the value of the structured variable in the "structured variables" window.

8.3.4. The Dialog Window

The dialog window shows messages and prompts user input. During the initialization of the debugger, the trap cause of the debugged program is written in the dialog window.

Commands

`exit:`
Exit the debugger.

`map:`
Install and show the load map window.

`memory:`
Install and show the memory window.

`screen:`
Install and show the screen window. (Not always possible).

`process:`
Install and show the process window.

8.3.5. The Memory Window

The memory window is used for inspection of the main memory. This window is shown on demand only.

The first column contains the octal address. The rest of the line shows the contents of memory in any selected mode.

Commands

`ind:`
Demands selection of a memory word and uses the contents as address of the memory area to be displayed. This is the default command executed when the left mouse button is pressed.

`addr:`
Asks for an (octal) address from the keyboard and shows that memory portion.

`process:`
Demands selection of a memory word with the mouse and shows the interpretation of this and the following words as a process descriptor in the process window.

`mode:`
Changes the representation of the displayed memory area. Another menu is shown, which lets you select the data representation either as octal, cardinal, integer, byte, hexadecimal or character.

8.3.6. The Load Map Window

The load map window shows the load map. This window is shown on demand only.

Commands

`list:`
Shows the listing of a program unit in the program window. After giving the list command, select a module by pointing at it with the mouse.

`data:`
Shows the data of a program unit in the data window. After giving the data command, select a procedure or a module by pointing at it with the mouse.

The left button combines the two commands for listing and data. The data and the listing of the selected module are shown in their respective windows.

The module names in the load map are shown with corresponding data and code frame pointer and internal module number.

8.3.7. The Process Window

The process window allows inspection of process descriptors. The cause of the error trap is described in this window.

This window is especially useful for debugging programs with coroutines (PROCESS). The displayed values correspond to the machine registers saved in the process descriptor.

This window is shown on demand only.

Commands

`debug:`

Debugs the process whose process descriptor is shown in the process window.

`normal:`

Switch the process window to show the normal process descriptor. This is the process descriptor of the process which caused the termination.

`caller:`

Switch the process window to show the process descriptor of the calling process. This feature is used mainly when a program which uses the loader (procedure Call of module Program) is debugged.

8.3.8. The Screen Window

The screen window shows the screen of the debugged program. Not all debugger versions support this window. This window is shown on demand only.

This window has in addition to the standard vertical scroll bar also a horizontal scroll bar. The left button shows the relative location of the shown parts of the screen. The middle button shows, as usual, a menu.

Restriction (for the current version of the debugger): The screen of a program can be shown only when that program imports the module Screen.

8.3.9. The Background Commands

A group of commands can also be activated when the cursor is outside all windows, i.e. in the background. These commands are used to modify the debugging environment, like the size and location of the windows or the fonts used.

The commands of this menu are applied to the debugger itself; they have no connection to the debugged program. After selecting a command a window has to be designated; the command is relative to that window.

Commands

`exit:`

Exits the debugger like the exit command in the dialog window.

`call:`

Shows another menu. Selects a utility program and executes it. (Either directory, copy, delete, rename, list or other important routines). If the utility-menu is discarded and the keyboard is pressed, it starts a command interpreter, reads a program name and executes that program inside the window. (Press ESC to quit the command interpreter). None of the called programs is part

of the debugger; it is possible to run out of memory or to get other loader or execution error messages. The call command allows creation of a new, temporary window if no window is selected.

This command is dangerous. When the called program traps or is halted, it will produce a dump. The new dump may overwrite the old dump, which currently is being inspected by the debugger, causing the debugger to inspect inconsistent data.

Press CALLS instead of CALL to show stack & call nesting etc.

remove:

Removes the Window. The memory, process, screen and load map windows are the only ones which can be removed.

move:

Moves the window, asking the user to select the new location.

change:

Changes the window size or location. Asks the user to point out the diagonal of the new window. Size changes have different effects for the different windows. It may be useful to have a smaller program window, but the lines will be clipped. Losing information of the procedure call chain window is normally not tolerated. The memory window will always display contiguous areas of memory, unless it is too small to display anything.

font:

Set font used for the window. When positioning is needed (memory window!) a non-proportional (fixed width) font should be used.

order:

Shows another menu with names of windows. Select a window to be placed on top (made it visible).

When no menu command is selected, but the mouse button is released while the cursor square points to a window, this window is put on top (made visible).

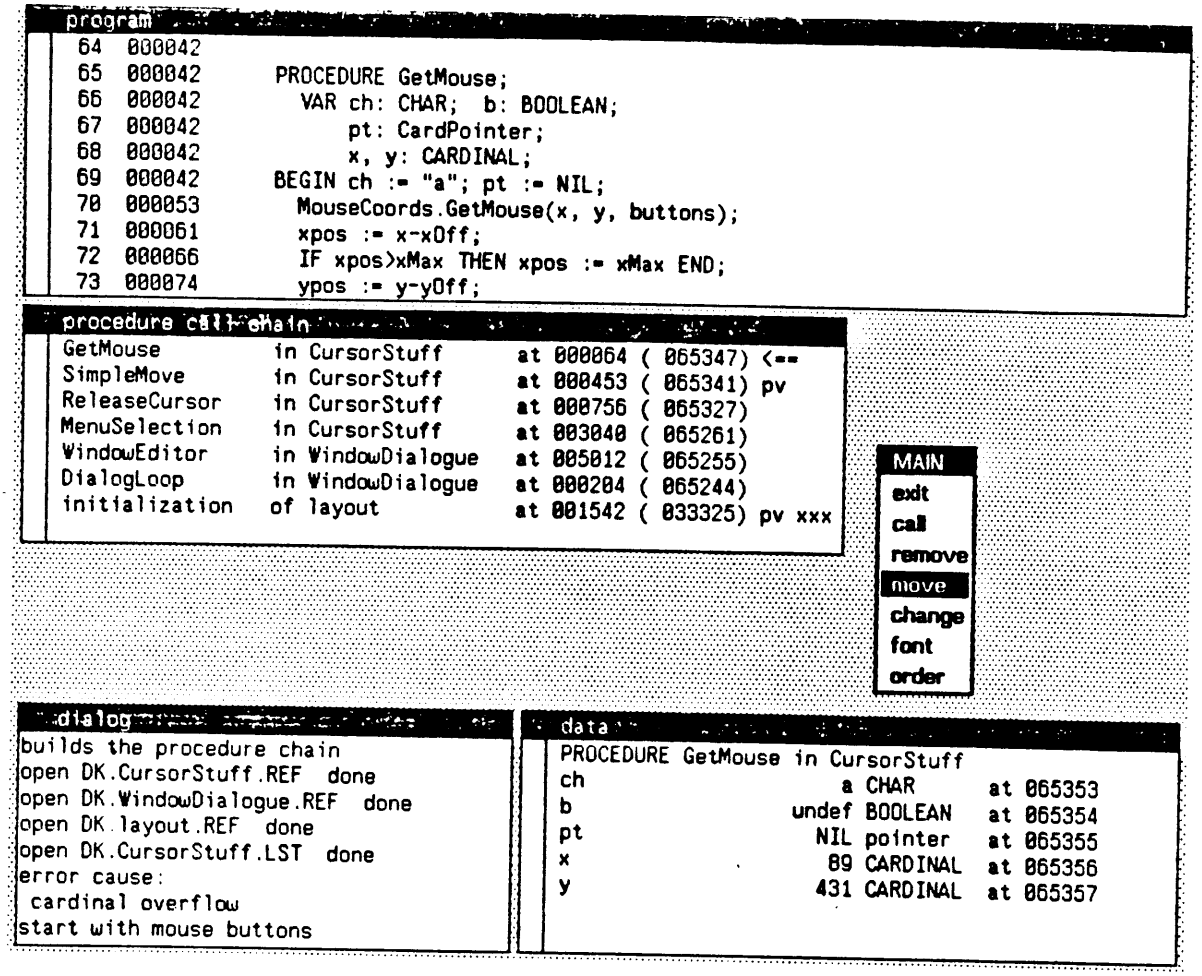
Fine points:

The top line of the screen is considered to be background. This allows selection of the background commands when the background is not visible at all.

To escape from a started command, press ESC.

8.4. An Example

8.4.1. Screen with Default Layout



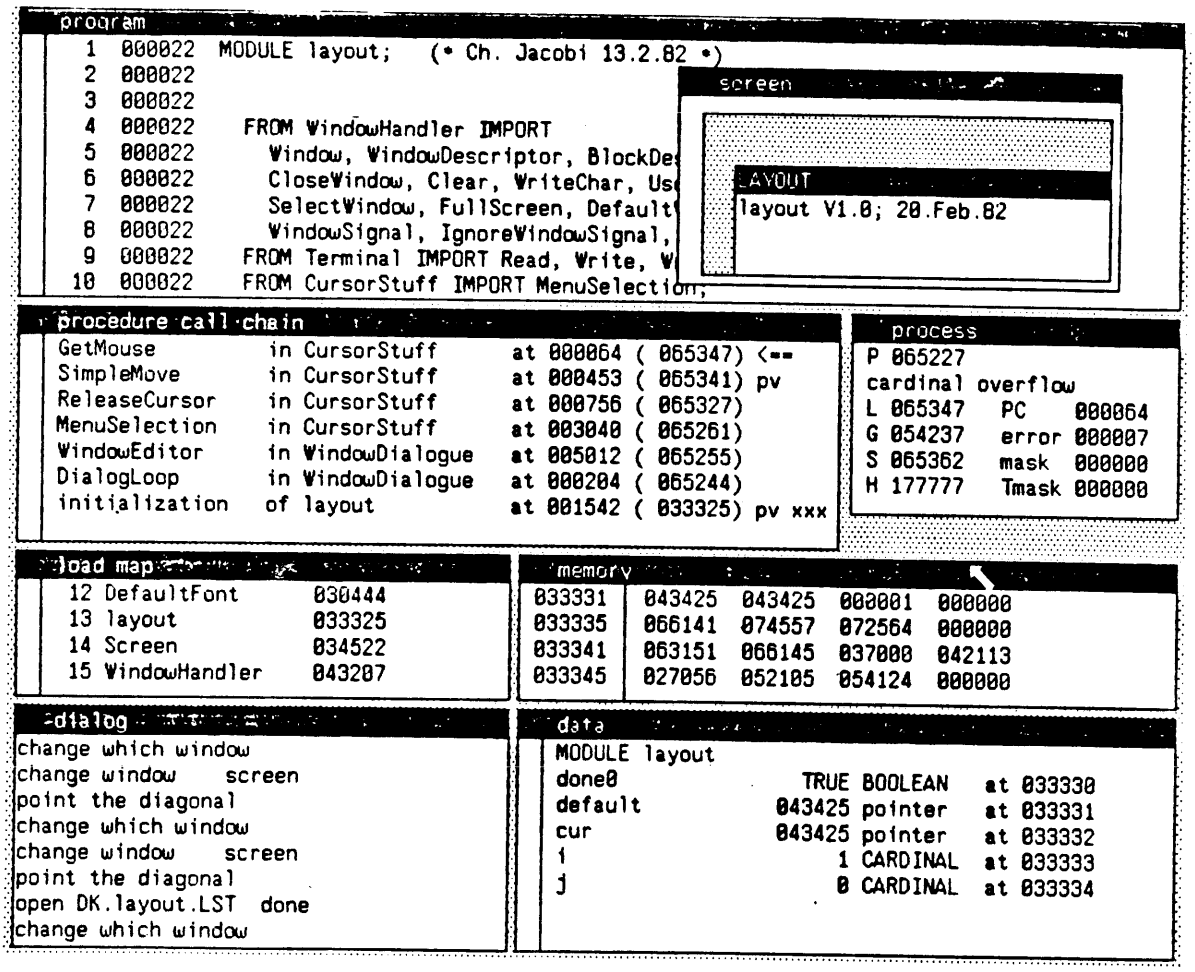
1 program window
2 procedure call chain window
3 dialog window
4 data window
5 background menu with move command selected
6 scroll bar
7 line number
8 approximate location; byte offset to module base. (displayed octal).
9 program text
10 procedure name
11 name of embedding module
12 program counter (like 8)
13 base address of data
14 execution stopped in this procedure
15 this procedure is called as procedure variable
16 error cause
17 memory address
18 message: file has been successfully opened
19 name of a variable
20 value of a variable
21 type description

22 name of embedding module or file

In the dialog window the trap cause *cardinal overflow*, is displayed. The procedure call chain window shows, that the process terminated at location 64B in the procedure GetMouse of the module CursorStuff. This location is approximately found on line 71 in the program window. On that line the cardinal expression x-xOff is found. In this program example xOff is declared as a constant of value 90 (Not currently shown in the debugger). Inspection of the data window shows that x has the value 89, which was the cause of the overflow.

(Note: The actual library module CursorStuff will NOT cause such cardinal overflows.)

8.4.2. Screen with Additional Opened Windows



In this figure, the additional windows have been opened. The module layout in the load map window has been selected, causing the modules program and data to be shown.

30 load map window
31 process window
32 memory window
33 screen window
34 registers saved in processdescriptor
35 process pointer
36 error cause
37 module number
38 module name
39 module base address (data segment, in octal)

40 screen of debugged program (shows again a window)
41 horizontal scroll bar of the screen window
42 cursor
43 interactive dialog; the screen window had been changed
44 memory address (inside the memory scroll bar)
45 memory data; current mode is octal

9. The Medos-2 Interface

Svend Erik Knudsen 15.5.82

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

FileSystem	Standard module for the use of files	9.1.
Program	Facilities for the execution of programs upon Medos-2	9.2.
Storage	Standard module for storage allocation in the heap	9.3.
Terminal	Standard module for sequential terminal input/output	9.4.

9.1. Module FileSystem

Svend Erik Knudsen 15.5.82

9.1.1. 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 9.1.2. The explanations needed for simple usage of sequential (text or binary) files are given in chapter 9.1.3. More demanding users of files should also consult chapter 9.1.4. The file system supports several implementations of files. At execution time a program may declare that it implements files on a certain named medium. How this is achieved is mentioned in chapter 9.1.5. On Lilith the 10 Mbyte cartridge for the Honeywell Bull D120/D140 disk drive is the standard medium for files. Some characteristics and restrictions of the current implementation, as well as a list of possible error messages, are given in chapter 9.1.6.

9.1.2. 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,
    ReadWord, WriteWord, ReadChar, WriteChar,
    Reset, Again, SetPos, GetPos, Length,
    Command, MediumType, FileCommand, DirectoryCommand,
    Flag, FlagSet,
    SetRead, SetWrite, SetModify, SetOpen, Doio,
    FileProc, DirectoryProc, CreateMedium, RemoveMedium;


  TYPE
    MediumType     = ARRAY [0..1] OF CHAR;
    MediumHint;
    Flag           = (er, ef, rd, wr, ag, bytemode);
    FlagSet        = SET OF Flag;

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

    Command        = (create, open, close, lookup, rename,
                      setread, setwrite, setmodify, setopen,
                      doio,
                      setpos, getpos, length,
                      setprotect, getprotect,
                      setpermanent, getpermanent,
                      getinternal);

    File           = RECORD
                       bufa: ADDRESS;
```

```
                        ela: ADDRESS; elodd: BOOLEAN;
                        ina: ADDRESS; inodd: BOOLEAN;
                        topa: ADDRESS;
                        flags: FlagSet;
                        eof: BOOLEAN;
                        res: Response;
                        CASE com: Command OF
                          create, open, getinternal:
                            fileno, versionno: CARDINAL
                        | lookup: new: BOOLEAN
                        | setpos, getpos, length: highpos, lowpos: CARDINAL
                        | setprotect, getprotect: wrprotect: BOOLEAN
                        | setpermanent, getpermanent: on: BOOLEAN
                        END;
                        mt: MediumType; mediumno: CARDINAL;
                        mh: MediumHint;
                        submedium: ADDRESS;
                      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 FileCommand(VAR f: File);
  PROCEDURE DirectoryCommand(VAR f: File; filename: ARRAY OF CHAR);

  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);


  TYPE
    FileProc      = PROCEDURE (VAR File);
    DirectoryProc = PROCEDURE (VAR File, ARRAY OF CHAR);

  PROCEDURE CreateMedium(mt: MediumType; mediumno: CARDINAL;
                         fp: FileProc; dp: DirectoryProc; VAR done: BOOLEAN);
  PROCEDURE RemoveMedium(mt: MediumType; mediumno: CARDINAL;
                         VAR done: BOOLEAN);

END FileSystem.
```

9.1.3. Simple Use of Files

9.1.3.1. 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* on a named medium or *looked up* by its file name. The syntax of *medium name* and *file name* is

```
medium name  = [ identifier ] .
identifier   = letter { letter | digit } .

file name    = medium name [ "." local name ] .
local name   = identifier { "." identifier } .
```

Capital and lower case letters are treated as being different. The medium name is the name of the medium, upon which a file is (expected to be) stored. 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.).

DK.SYS.directory.OBJ

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

Create(f, mediumname)

Procedure *Create* creates a new temporary (and nameless) file on the given medium. After the call

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

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

If file f is connected, the field f.new indicates:

f.new = FALSE	File f existed already
f.new = TRUE	File f has been created by this call

Rename(f, filename)

Procedure *Rename* changes the name of file f to filename. If filename is empty or contains only the medium name, 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

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 9.4.) and to handle defaults (see chapter 11.11.).

9.1.3.2. 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 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)

A call of procedure *Again* prevents a following call to procedure *ReadChar* (*ReadWord*) from reading the next character (word) on file f. Instead, the character (word) read just before the call of *Again* will be read again.

Implementation Note

The current versions of the routines *ReadWord* and *WriteWord* do not support reading and writing of words at odd positions (for more information on *current position*, see 9.1.3.3).

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). This module also transfers words correctly if the current position of the file is odd (see note above)!

9.1.3.3. 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 behind 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.***

9.1.3.4. Examples

Writing a Text File

```
.
VAR
  f: File;
  ch: CHAR; endoftext: BOOLEAN;
.
.
Lookup(f, "DK.newfile", TRUE);
IF (f.res <> done) OR NOT f.new 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, "DK.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
.
.
```

9.1.4. Advanced Use of Files

9.1.4.1. The Procedures FileCommand and DirectoryCommand

In the previous sections, the file variable served, with few exceptions, simply as a reference to a file. The exceptions were the fields *eof*, *res* and *new* within a file variable. Generally, however, all operations on a file are implemented by either inspecting or changing fields within the file variable directly and/or by encoding the needed operation (*command*) into the file variable followed by a call to either routine *FileCommand* or *DirectoryCommand*. *Commands* requiring (part of) a filename as parameter are executed by *DirectoryCommand*, all others by *FileCommand*. An implementation of *SetPos* and *Lookup* should illustrate this:

```
PROCEDURE SetPos(VAR f: File; highpos, lowpos: CARDINAL);
BEGIN
  f.com := setpos;
  f.highpos := highpos; f.lowpos := lowpos;
  FileCommand(f);
END SetPos;

PROCEDURE Lookup(VAR f: File; filename: ARRAY OF CHAR; new: BOOLEAN);
BEGIN
  f.com := lookup;
  f.new := new;
  DirectoryCommand(f, filename)
END Lookup;
```

The commands *lookup* and *rename* must be executed by *DirectoryCommand*, other commands may be executed either by *FileCommand* or by *DirectoryCommand*. Unless the command is *lookup* or *rename*, a call to *DirectoryCommand* will be converted by the file system to a call to *FileCommand*. This facility is only useful for the commands *create* and *open* (see also 9.1.4.2).

Below is a list of all commands and a reference to the section where each is explained:

create	create a new temporary (and nameless) file	(9.1.3.1)
open	open an existing file by *IFI*	(9.1.4.2)
close	close a file	(9.1.3.1)
lookup	look up (or create) a file by file name	(9.1.3.1)
rename	rename a file	(9.1.3.1)
setread	set a file into state *reading*	(9.1.4.5)
setwrite	set a file into state *writing*	(9.1.4.5)
setmodify	set a file into state *modifying*	(9.1.4.5)
setopen	set a file into state *opened*	(9.1.4.5)
doio	get next buffer	(9.1.4.5)
setpos	change the *current position* of the file	(9.1.3.3)
getpos	get the *current position* of the file	(9.1.3.3)
length	get the *length* of the file	(9.1.3.3)
setprotect	change the *protection* of the file	(9.1.4.4)
getprotect	get the current *protection* of the file	(9.1.4.4)
setpermanent	change the *permanency* of the file	(9.1.4.3)
getpermanent	get the *permanency* of the file	(9.1.4.3)
getinternal	get the *LFI* of the file	(9.1.4.2)

After the execution of a command, field *res* of the file reflects the success of the operation. Other fields of the file variable might, however, contain additional return values, depending on the executed command and the *state* of the file (see 9.1.4.5). Here, the normal way of setting the fields before a return from procedure *FileCommand* is given:

```
WITH f DO
  (* set other fields *)
  res := "...";
```

```
  flags := flags - FlagSet{er, ef, rd, wr};
  IF "state = opened" (* see 9.1.4.5 *) THEN
    bufa  := NIL;                  (* no buffer assigned *)
    ela   := NIL; elodd := FALSE;
    ina   := NIL; inodd := FALSE;
    topa  := NIL;
    eof   := TRUE
  ELSE
    bufa  := ADR("buffer");        (* buffer at current position of file *)
    ela   := ADR("word in buffer at current position");
    elodd := ODD("current position");
    ina   := ADR("first not (completely) read word in buffer");
    inodd := "word at ina contains one byte";
    topa  := ADR("first word after buffer");
    eof   := "current position = length";
    IF "(state = reading) OR (state = modifying)" THEN INCL(flags, rd) END;
    IF "(state = writing) OR (state = modifying)" THEN INCL(flags, wr) END;
    IF elodd OR ODD("length") THEN INCL(flags, bytemode) END;
  END;
  IF res <> done THEN eof := TRUE; INCL(flags, er) END;
  IF eof THEN INCL(flags, ef) END
END
```

The *states* of a file and the file buffering are explained in 9.1.4.5. The field *flags* enables a simple (and therefore efficient) test of the state of the file, whenever it is accessed. The "flag" *ag* is set by routine *Again* and cleared by read routines.

9.1.4.2. Internal File Identification and External File Name

All files supported by the file system have a unique identification, the so called *internal file identification (IFI)* and might also have an external (or symbolic) *file name.*

Both the internal file identification and the file name consist of two parts, namely a part identifying the medium upon which a file is (expected to be) stored, and a part identifying the file on the selected medium.

The two parts of an internal file identification are called the *internal medium identification (IMI)* and the *local file identification (LFI).* The two parts of a file name are called the *medium name* and the *local file name.*

The IFI of a connected (opened) file may be obtained at any time: The IMI is always stored in the fields *mt* and *mediumno* of the file variable. The LFI is stored in the fields *fileno* and *versionno* after the execution of command *create* or *getinternal.*

A file f can be opened, if it exists and its IFI is known:

```
f.mt := ...; f.mediumno := ...;
f.fileno := ...; f.versionno := ...;
f.com := open;
FileCommand(f)
```

The identification of a file by a user selected or computed name (a string) is however both commonly accepted and convenient. The syntax of a *file name* is given in 9.1.3.1. The routines *Create, Lookup, Rename* and *DirectoryCommand* all have a parameter specifying the file name.

If the *medium name* is contained in the file name, it is "converted" into an IMI and stored into the file variable, except when the rename command is used. In this case, the "converted" IMI is checked against the IMI stored in the file variable. If the medium name is missing in the actual file name parameter, it is assumed that the corresponding IMI is already stored in the file variable.

The *local file name* part of the file name will be handled by the routine implementing *DirectoryCommand* for the medium given by the IMI (see also 9.1.5.).

Implementation Notes

The current version of module ***FileSystem*** supports only ***medium names*** according to the following syntax:

```
medium name = letter [ letter ] { digit } .
```

When a ***medium name*** is "converted" to an ***internal medium identification***, the letter(s) is (are) copied to the *MediumType* part (field *mt*), and the digits are considered as a decimal number whose value is assigned to the ***medium number*** part (field ***mediumno***). If the medium name contains no digits, medium number 65535 (=177777B) is assumed.

```
"DK"      =>   ( "DK", 65535)
"DK0"     =>   ( "DK", 0)
"DK007"   =>   ( "DK", 7)
```

9.1.4.3. Permanency of Files

As explained in 9.1.3.1, a file is either ***temporary*** or ***permanent***. The rule is that, when a file is closed (explicitly, implicitly, or in a system crash), a temporary file is deleted and a permanent file will remain on the medium for later use. Normally, a "nameless" file is temporary, and a "named" file is permanent. It is, however, possible to control the permanency of a file explicitly. This is useful, if for some reason, it is better to reference a file by its IFI instead of its file name (e.g. in data base systems, other directory systems).

Set File Permanent

```
f.on := TRUE; f.com := setpermanent;
FileCommand(f)
```

Set File Temporary

```
f.on := FALSE; f.com := setpermanent;
FileCommand(f)
```

Get File Permanency

```
f.com := getpermanent;
FileCommand(f);

(* f.on = TRUE if and only if f is permanent *)
```

9.1.4.4. Protection of Files

A file can be protected against any changes only (length, information, name etc.). The only exception to this rule is, of course, that the protection of a protected file may be changed.

Protect File

```
f.wrprotect := TRUE; f.com := setprotect;
FileCommand(f)
```

Unprotect File

```
f.wrprotect := FALSE; f.com := setprotect;
FileCommand(f)
```

```
bufa <= ela <= topa
bufa <= ina <= topa
```

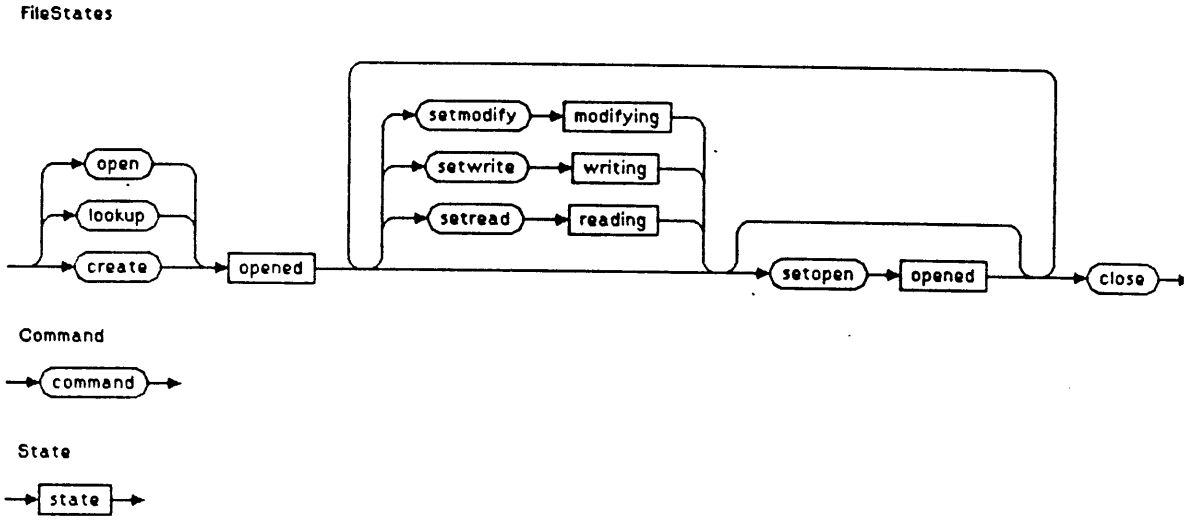
The fields *bufa*, *ina*, *inodd*, and *topa* are *read-only*, as they contain information which must never be changed by any user of a file.

If the file is not in state *opened*, the byte at the current position will be in the buffer after procedure *FileCommand* has been executed. The read information is stored in the buffer between *bufa* and *(ina, inodd)*. The pair *(ela, elodd)* always points to the byte at the current position of the file, i.e. to the byte (or to the first byte of the element) to read, write, or modify next in the file. If *(ela, elodd)* points outside the buffer, and no other command has to be executed, the byte at the current position can be brought into the buffer by a call to *Doio* or by the execution of command *doio* respectively.

The following two assertions also hold after a call to *FileCommand*, if the state of the file is *reading*, *writing*, or *modifying*.

```
(ela, elodd) <= (ina, inodd)
ela          <  topa
```

The current position of a (connected) file can only be changed by either an (explicit or implicit) execution of command *setpos* or by changing *ela* and/or *elodd* (implicitly or explicitly). In the latter case of course, the file system "knows" the exact value of the current position only after an activation of the routine *FileCommand*.



This figure shows how the I/O state of a file is changed when different commands are executed. Commands not shown in the figure do not affect the I/O state of a file. Whenever the command *setopen* is omitted, the system might execute *setopen* before executing the following command.

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. A buffer is assigned to the file, and the information between the beginning of the buffer and the current position (= length) is read into the buffer. Information in the buffer up to the location denoted by *(ela, elodd)* is considered as belonging to the file and will be written back onto the actual file.

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. In this state, information in the buffer up to *MAX((ela,elodd), (ina,inodd))* is considered as belonging to the file and will therefore be written back onto the actual file. The length of the file might hereby be increased but never decreased!

Doio(f)

If the state of the file is *reading*, *writing* or *modifying*, the buffer with the byte at current position is assigned to the file after a call to *Doio*. A call to *Doio* is essentially needed, if *(ela,elodd)* points outside the buffer and no other command has to be executed.

9.1.4.6. Examples

Procedure *Reset(f)*

```
PROCEDURE Reset(VAR f: File);
BEGIN
  SetOpen(f);
  SetPos(f, 0, 0);
END Reset;
```

Write File f

```
.
(* assume, that file f is positioned correctly *)
SetWrite(f);
WHILE "word to write" DO
  IF ela = topa THEN Doio(f) END;
  ela↑ := "next word to write";
  INC(ela);
END;
SetOpen(f);
IF f.res <> done THEN
  (* some write error occured *)
END;
.
```

Read File f

```
.
(* assume, that file f is positioned correctly *)
SetRead(f);
WHILE NOT f.eof DO
  WHILE ela < ina DO
    "use ela↑";
```

```
    INC(ela);
  END;
  Doio(f);
END;
SetOpen(f);
IF f.res <> done THEN
  (* Some read error occured *)
END;
.
```

Procedure *WriteChar*

```
PROCEDURE WriteChar(VAR f: File; ch: CHAR);      (* SEK  15.5.82 *)

  PROCEDURE SXB(a: ADDRESS; oddpos: BOOLEAN; ch: CHAR);
    (* Store indeXed Byte *)
  CODE 225B END SXB;

BEGIN
  WITH f DO
    LOOP
      IF flags * FlagSet{wr, bytemode, er} <> FlagSet{wr, bytemode} THEN
        IF er IN flags THEN RETURN END;
        IF NOT (wr IN flags) THEN
          IF rd IN flags THEN
            (* Forbid to change directly from reading to writing! *)
            res := callerror; eof := TRUE;
            flags := flags + FlagSet{er, ef}
          ELSE SetWrite(f)
          END
        END;
        INCL(flags, bytemode)
      ELSIF ela >= topa THEN Doio(f)
      ELSIF elodd THEN
        SXB(ela, TRUE, ch);
        INC(ela, TSIZE(WORD)); elodd := FALSE;
        RETURN
      ELSE
        SXB(ela, FALSE, ch);
        elodd := TRUE;
        RETURN
      END
    END
  END
END WriteChar;
```

9.1.5. Implementation of Files

A program may implement files on a certain medium and make these files accessible through the file system (that is, through module *FileSystem*). This is done with a call to procedure *CreateMedium*. The medium which the calling module will support, is identified by its *internal medium identification (IMI)*. The two procedures given as parameters should essentially implement procedure *FileCommand (fileproc)* and *DirectoryCommand (directoryproc)* for the corresponding medium. Whenever a command is executed on a file, module *FileSystem* activates the procedure which handles the command for the medium upon which the file is (expected to be) stored. The commands *lookup* and *rename* will cause procedure *directoryproc* to be called; all other commands will cause procedure *fileproc* to be called. The string supplied as parameter to procedure *directoryproc* contains only the *local file name* part of the original file name. The corresponding IMI is stored in the file variable. The field *submedium* in the file variable may be used freely by the module implementing files (e.g. as an index into a table of connected files).

After a call to procedure *RemoveMedium*, the indicated medium is no longer known by the file system. This procedure can, however, be called only from the program which "created" the medium. A medium will automatically be removed, if the program within which it was "created" is removed.

As connected files should have "lifetimes" like Modula-2 pointers (dynamically created variables), a medium should only be declared from an unshared program (i.e. if *SharedLevel() = CurrentLevel()*, see module *Program*, chapter 9.2.).

CreateMedium(mediumtype, mediumnumber, fileproc, directoryproc, done)

Procedure *CreateMedium* announces a new medium to the file system. *done* is TRUE if the new medium was accepted.

RemoveMedium(mediumtype, mediumnumber, done)

After a call to *RemoveMedium*, the given medium is no longer known to the file system. *done* is TRUE if the medium was removed.

Implementation Note

Eight is the highest number of media that the current version of module *FileSystem* can support at the same time.

9.1.6. Files on Cartridges for Honeywell Bull D120/D140 Disk Drives

9.1.6.1. Main Characteristics and Restrictions

Modules *DiskSystem* and *D140Disk* implement files on cartridges for the Honeywell Bull D120/D140 disk drives. The main characteristics of the current implementation are listed below:

maximum number of files	768/cartridge
maximum file length	192 kbyte
cartridge capacity	9408 kbyte
typical transfer rates	3 - 30 kbyte/sec
minimum transfer rate	< 10 byte/sec
maximum transfer rate	> 50 kbyte/sec
local file name length	1 - 24 characters
maximum number of opened files	14 (16)
medium name	"DK"
internal medium identification	("DK", 65535)

Each actual file can be connected to *only one* file variable at the same time. As long as essentially only a single program runs on the machine, this should be acceptable, as it is more an aid than a restriction.

The transfer rates depend mostly on the number of disk head movements needed for the actual transfer. The positioning of a file for each transfer of one or a few bytes might decrease the transfer rate to some few bytes per second. On the other hand, sequential transfers of larger elements (>= 16 byte/element) are performed with the maximum transfer rate (50 - 60 kbyte/sec).

Actually 16 files can be connected at the same time. Module *DiskSystem* uses two of them internally for access to the two directories on the cartridge. The remaining 14 files may be used freely by ordinary programs.

The current version of module *DiskSystem* does not distinguish between cartridges. All cartridges are simply given the same internal medium identification ("DK", 65535).

9.1.6.2. System Files

The space on a cartridge is allocated to actual files in *pages* of 2 kbyte each (or 8 sectors). The pages belonging to a file as well as its length and other information is stored in a file descriptor, which itself is stored in a file on the cartridge (file directory). The local file names of all files on a cartridge are stored in another file on the cartridge (name directory). When a cartridge is initialized, nine (system-)files are allocated on the cartridge. These preallocated files can not be truncated or removed. Except for the two directory files and the file containing the cartridge's bad sectors, all files can be read and written (modified). The preallocated files are:

FS.FileDirectory	File with file directory
FS.FileDirectory.Back	Back up of file directory (not implemented)
FS.NameDirectory	File with name directory
FS.NameDirectory.Back	Back up of file with name directory (not implemented)
FS.BadPages	File with unusable sectors
PC.BootFile	Normal boot file
PC.BootFile.Back	Alternate boot file
PC.DumpFile	File onto which the main memory (0 .. 64k-1) is dumped
PC.Dump1File	File onto which the main memory (64k .. 128k-1) is dumped

9.1.6.3. Error Handling

Normally all detected errors are handled by assigning a *Response* indicating the error to field *res* in the file variable. Whenever a detected error cannot be related to a file or if a more serious error is detected, an error message is written on the display. This is done according to the following format:

"- " module name ["." procedure name] ":" error indicating text

module name and *procedure name* are the names of the module and the procedure within the module, where the error was detected. In the explanations of the messages, the following terms are used for inserted values:

page number	octal number (0 .. 137B)	Page in an affected file
page	octal number (0..167340B)	Disk address of page = page DIV 13 * 8
file number	octal number (0 .. 1377B)	Number of the affected file
local file name	string(1 .. 24)	Local file name of affected file
response	string	Text describing the *response*
statusbits	octal number (177400 .. 177777B)	Status from disk interface
disk address	octal number (0 .. 111377B)	"Logical" address of sector on disk

If some of the following error messages are displayed, please consult the description of program *DiskCheck*!

- `DiskSystem.PutBuf: bad page: pageno =` ***page number*** `fno =` ***file number***
  Page indicates a disk address which is allocated to a "system file", but the file is not a "system file", or the page indicates a disk address for normal files, but the file is a "system file".

- `DiskSystem.GetBuf: bad buffering while reading ahead`
  The disk address of a certain allocated sector was not found.

- `DiskSystem.FileCommand: bad directory entry: fno =` ***file number*** `read fno =` ***file number***
  An inconsistency in the file directory was detected.

- `DiskSystem.OpenVolume: bad page pointer:`
  `fno =` ***file number*** `pageno =` ***page number*** `page =` ***page***
  An inconsistency in the file directory was detected during the initialisation of Medos.

- `DiskSystem.(ReadName, WriteName or SearchName): bad file number in name entry`
  `file name =` ***local file name***
  `found fno =` ***file number***`, expected fno =` ***file number***
  An inconsistency in the name directory was detected.

- `D140Disk: soft timeout in wait`
  A disk operation was timed out by software. This error occurs mainly, if the disk is switched off while a disk operation is processed. Usually, this has no harmful consequences.

- `D140Disk.DiskRead:` ***response***
- `diskadr =` ***disk address***`, statusbits =` ***statusbits***
  The driver detected an error, which did not disappear after three retries.

- `D140Disk.DiskWrite:` ***response***
- `diskadr =` ***disk address***`, statusbits =` ***statusbits***
  The disk driver detected an error, which did not disappear after three retries.

Warning

It must be mentioned here that among *the best ways to get some of these error messages on the screen* is this one: Switch off the drive while a "harmless" program is running, exchange the cartridge in the drive, and switch on the drive again. A cartridge exchange is simply *not* detected by module DiskSystem which does *not*, therefore, initialize its local information about the mounted cartridge from the new cartridge.

9.2. Module Program

Svend Erik Knudsen 15.5.82

9.2.1. 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 9.2.2. The program concept and explanations needed for the activation of a program are given in chapter 9.2.3. The ***heap*** and two routines handling the heap are explained in chapter 9.2.4. Possible error messages are listed in 9.2.5. The object file format may be inspected in 9.2.6.

9.2.2 Definition Module Program

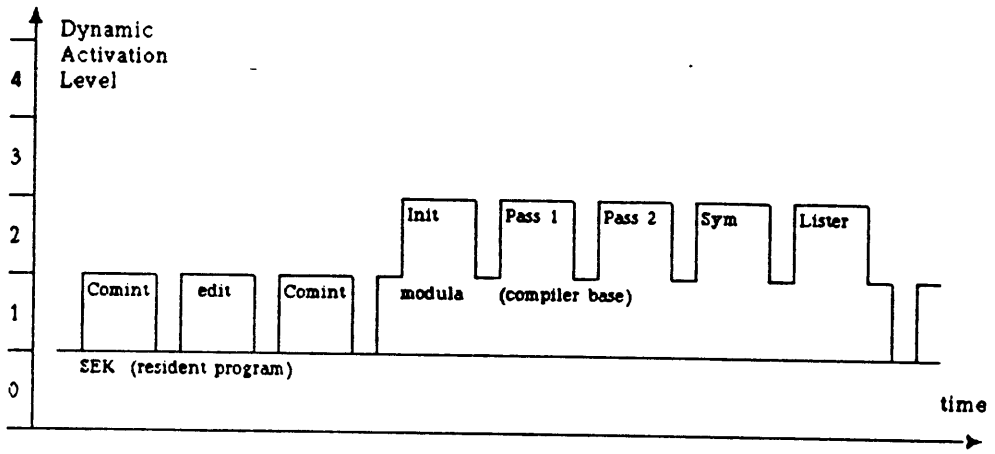
```
DEFINITION MODULE Program;      (* Medos-2 V3  S. E. Knudsen  1.6.81 *)

  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.
```

9.2.3. 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.

Typical Execution of Programs



The figure illustrates, how programs may be activated. At a certain moment, the *dynamic activation level* or simply the *level* identifies an 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).

A program can only be active once at the same time (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 analog 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 actually are 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 sharable. 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. At most 16 characters are allowed for program names. Capital and lower case letters are treated as being different.

```
Program name    = Identifier { "." Identifier } . / At most 16 characters/
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 prefix *DK.* is inserted before the program name, and the extension *.OBJ* is appended. If no such file exists, the prefix *DK.* is replaced by the prefix *DK.SYS.*, 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 (first 16 characters of the) module name is converted to a file name by inserting *DK.* at the beginning of the module name and appending the extension *.OBJ* to it. If the file is not found, a second search is made after the prefix *DK.* has been replaced by the prefix *DK.LIB.*. 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	directory
First searched file	DK.directory.OBJ
Second searched file	DK.SYS.directory.OBJ
Module name	Storage
First searched file	DK.Storage.OBJ
Second searched file	DK.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 13 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;

  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.
```

9.2.4. Heap

The main memory of Lilith 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;
```

9.2.5. 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.

9.2.6. 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" }. /exactly 16 characters/
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.

9.3. 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, procedures which are either explicitly programmed or imported from a separate module (see Modula-2 report in [1], chapter 10.2). 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 9.2.).

Loading of Module Storage

Module *Storage* may be loaded once for each heap it should handle. For more details see module *Program*, chapter 9.2.

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.

9.4. 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 ReadAgain;

  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*.

ReadAgain

A call to *ReadAgain* prevents the next call to *Read* or *BusyRead* from getting the next typed character. Instead, the last character read before the call to *ReadAgain* will be returned again.

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

10. Screen Software

Chrisitan Jacobi 15.5.82

The screen software chapter describes the following modules:

Screen	(10.2.)
TextScreen	(10.3.)
WindowHandler	(10.4.)
CursorStuff	(10.5.)
CursorRelations	(10.6.)
WindowDialogue	(10.7.)
ScreenResources0	(10.8.)
BitmapVars	(10.9.)

10.1. Summary

For default sequential output the use of module *Terminal* is recommended; the use of the higher level modules should be reserved to the case, when the output is not strictly sequential. Terminal is described in an operating system description (9.4.) and is not explained in the screen software chapter.

Formatting modules (e.g. *OutTerminal, OutWindow*) are found in the chapter 11.

The module *TextScreen* supports positioning (and output to) the display screen.

The module *Screen* is the base for all bit directed operations on the display terminal. It contains only a minimum of features: the basic display operations, loading of fonts, creation of bitmaps and subbitmaps, and the default settings.

The module *WindowHandler* allows the use of windows. A window is a visible rectangle on the screen where text and binary operations are possible. Such a rectangle simulates a complete display. The windows can be compared to pieces of paper lying on a table. They can be moved, an overlaid window may be put on top, and of course, programs may write or paint into windows.

The module *CursorStuff* implements a cursor and allows selection of commands with a menu.

CursorRelations imports the cursor position of CursorStuff and computes the relative cursor position to windows.

With the module *WindowDialogue* it is possible to build programs with independent modules. The module WindowDialogue collects the interactive input commands and directs them to the specific windows and activities. The module implements a so called window editor, which allows interactive modifications of the windows.

ScreenResources0 is used to get access to resources used by the module Screen.

BitmapVars can be used by programs which completely ignore the screen software system around "Screen"; BitmapVars delivers the addresses of the standard bitmap, standard bitmap descriptor and standard font.

10.2. Screen

The module Screen implements the basic operations with display instructions. It allocates the resources and it implements a subbitmap feature. It exports the hardware functions in a controlled manner to the user, i.e. provides additional validity checks for parameters.

The Lilith has a raster display as output medium. The value of each picture element is independently stored as a bit in a two dimensional array, called *bitmap*. This bitmap is stored in the main memory. All ordinary data manipulation instructions may therefore be used to build pictures on the display terminal. This involves great flexibility; the normal instructions are however usually inadequate for handling images. Four specially microcoded instructions are used for that purpose (DCH, REPL, BBLT, DDT). These display instructions are represented by the display procedures DisplayChar, Replicate, BlockTransfer and DisplayDot.

The procedure CreateBitmap allocates a bitmap, and allows use of these operations. Calling the procedure ShowBitmap shows such a bitmap onto the display terminal. Most applications will use a bitmap which is already allocated by the system, the procedure GetDefaultBitmap returns this bitmap in its parameter. This *default* bitmap is usually shown on the terminal, there is no need to call ShowBitmap with this bitmap.

The display procedures work on rectangles within bitmaps. Variables of type BlockDescriptor are used to describe such rectangles. A block-descriptor describes a rectangle by giving its coordinates of the lower left corner, its width and its heights. It is possible to treat such a rectangle like a bitmap again. The procedure CreateSubBitmap takes a rectangle inside a bitmap and generates an abstract bitmap, denoting that rectangle. This abstract bitmap is called a *subbitmap*. All operations on bitmaps, except for ShowBitmap, may also be applied to subbitmaps. ShowBitmap is, however, restricted from the hardware and displays therefore it's "father"bitmap.

To be more exact in the former, also the default bitmap could have been changed to be such a subbitmap. When several windows are shown, one of these windows will be used as the default also. The procedure GetSystemBitmap is used to get the real *system* bitmap. The system bitmap may differ from the default bitmap, it is the actual bitmap which is allocated by the system, it cannot be changed and denotes the whole screen. The default bitmap is the rectangle to which the default output will be directed. Usually, the default bitmap is the whole or a part of the system bitmap.

To write a character, the procedure DisplayChar paints the picture of the character into a bitmap. The procedure copies the character's picture from a font table into the bitmap. The procedure LoadFont loads a font into a memory area and prepares it for use by DisplayChar. A great number of such fonts exist.

Types Used for Screen Resources

```
TYPE
  Bitmap;
  Font;
  BlockDescriptor =
    RECORD
      x, y, w, h: CARDINAL;
    END;
```

Bitmap: A bitmap which can be displayed at the display terminal, or a rectangular part of such a bitmap. To the programmer, there is no difference between such a rectangle (a subbitmap) and the real bitmap, which is the one displayed by hardware; the hardware displays only complete bitmaps. The origin of the coordinate system is at the lower left corner of the bitmap.

Font: Pictures of the characters in a character set.

BlockDescriptor: Describes a block which is a rectangle inside a bitmap. x,y are the coordinates of the lower left corner of the block, whereas w is the width and h the height of the block.

The structure of the types Bitmap and Font are hidden to the user. It is good programming style, not to assign variables of font or bitmap type.

The Elementary Display Procedures

In general, the display procedures work on two rectangles, called the source and the destination.

destination := F(destination, source)

The function F(d, s) depends on the procedure and on a mode parameter.

Mode: Mode of operation of the basic display instructions.

```
TYPE
  Mode = (replace,    (* d := s          , s      *)
          paint,      (* d := d OR s     , d+s    *)
          invert,     (* d := d XOR s    , d/s    *)
          erase);     (* d := d AND NOT s, d*(-s) *)
```

replace: replaces the destination by the source.
paint: the source is overlaid (added) to the destination.
invert: the destination is inverted, where the source contains ones.
erase: the destination is cleared, where the source contains ones.

BlockTransfer: is the most general display procedure. It copies the source block into the destination block according to the mode.

Replicate: Replicates the bitpattern (pattern) over a rectangle (the destination block) of the bitmap according to the mode.

pattern: This is not a declared type; any variable may be used as pattern, but the first word of the pattern must contain the number of words following. This number is the height of the pattern, its width is 16.

DisplayDot: Writes a single dot at the coordinates x, y of the bitmap according to a mode.

DisplayChar: Paints the picture of a character from a font table into a block of a bitmap. The position is given by the block (the block denotes the line). DisplayChar updates the block to exclude the just-painted character. DisplayChar works with paint-mode only; the block is not previously erased.

```
PROCEDURE DisplayChar(VAR bm: Bitmap; VAR lineBlk: BlockDescriptor;
          VAR f: Font; ch: CHAR);
PROCEDURE Replicate(VAR bm: Bitmap; VAR destBlk: BlockDescriptor;
          m: Mode; VAR pattern: ARRAY OF WORD);
PROCEDURE BlockTransfer(VAR dbm: Bitmap; VAR destBlk: BlockDescriptor;
          m: Mode; VAR sbm: Bitmap; VAR sourceBlk: BlockDescriptor);
PROCEDURE DisplayDot(VAR bm: Bitmap; x, y: CARDINAL; m: Mode);
```

Resource Handling

The following procedures allocate bitmaps or fonts:

```
PROCEDURE CreateBitmap(VAR bm: Bitmap; w, h: CARDINAL; VAR done: BOOLEAN);
PROCEDURE CreateSubBitmap(VAR bm: Bitmap; VAR father: Bitmap;
                          location: BlockDescriptor; VAR done: BOOLEAN);
PROCEDURE LoadFont(VAR f: Font; name: ARRAY OF CHAR; VAR done: BOOLEAN);
```

Creating bitmaps needs their size w and h in dots. w is augmented to a multiple of 16, since the lines of a bitmap must start on a word boundary in the memory. The procedures have a return parameter done. There are several reasons for malfunction: not enough memory, file for a font not found, too many resources allocated, etc. The allocation procedures always return to the calling program, there are no halts. Calling Loadfont for the same font a second time will return a different font value, however the actual font table is loaded only once. To unload the font table, all font variables pointing to that table must be returned. To use a subbitmap whose "father" bitmap is returned is considered an error.

For the procedures to return resources, to set or to query the defaults, see the definition module. The procedures SetDotPos and GetDotPos handle the insertion point, where the default output (from module Terminal) appears on the screen.

Furtheron, the specific properties of fonts may be requested:

```
PROCEDURE Proportional(VAR f: Font): BOOLEAN;
PROCEDURE FontHeight(VAR f: Font): CARDINAL;
PROCEDURE FontWidth(VAR f: Font): CARDINAL;
PROCEDURE FontBaseLine(VAR f: Font): CARDINAL;
PROCEDURE CharWidth(VAR f: Font; ch: CHAR): CARDINAL;
```

A proportional font is a font with characters of different widths. FontWidth returns the maximum character width for proportional fonts. The procedure FontBaseLine returns the interval from the bottom of the line used in DisplayChar to the baseline of the text.

Size of the Screen

Programs should not use knowledge about the screen size, in order to work properly using different screen formats. Further, if a windowhandler is used, a window could be denoted to simulate the default screen. To get the actual *size* of the *screen* (compatible to WindowHandler, but without importing it), use the following statements:

```
GetDefaultBitmap(bm);
GetMaxBlock(b, bm)
b.w may be used as screen width, b.h as height.
```

Two different hardware displays are used currently:
1) width 768; height 592 ("standard" horizontal display)
2) width 640; height 832 (new vertical display)

Clipping

Using blocks which do not fit completely into their bitmap is considered as an error, if the bitmap is a real bitmap. This error will cause a trap. For subbitmaps the operation is done with clipped blocks and no error occurs.

To achieve clipping within a bitmap, it is possible to create a subbitmap of the same size and to use the subbitmap instead of the bitmap.

Restrictions

The procedure ShowBitmap requires that a bitmap fulfills some further hardware restrictions, which are not necessarily granted by creating and using bitmaps without displaying them:

The width must be a multiple of 64 (dots). The height must be a multiple of 2.

In DisplayChar the firmware cannot recognize whether or not the character fits into the line block. However, it traps if the character would be painted outside the bitmap.

The procedure SetDefaultBitmap is disabled when the WindowHandler is loaded; in such cases, the WindowHandler takes care of the default bitmap.

The actual definition module exports some additional private stuff, needed to implement some system programs. This stuff will change on the next version of the operative system. Therefore avoid using it, even if you would know the declarations. The module ScreenResources0 gives you indirect access to that private stuff, until its improved version will be included in the module Screen.

Definition Module

```
DEFINITION MODULE Screen;  (* Ch. Jacobi  28.10.81*)
  FROM SYSTEM IMPORT ADDRESS, WORD;
  EXPORT QUALIFIED
```

```
  Bitmap, Font, Mode, BlockDescriptor,
  DisplayChar, Replicate, BlockTransfer, DisplayDot,
  Proportional, FontHeight, FontWidth, FontBaseLine,
  CharWidth,
  GetSystemBitmap, GetSystemFont,
  GetDefaultBitmap, GetDefaultFont, GetDotPos,
  SetDefaultBitmap, SetDefaultFont, SetDotPos,
  CreateBitmap, CreateSubBitmap, ReturnBitmap, ShowBitmap,
  LoadFont, ReturnFont, GetFontName,
  IsSubBitmap, GetRealFather, GetRealBlock, GetMaxBlock;

TYPE
  Bitmap;
  Font;
  Mode = (replace,  (* d := s           , s      *)
          paint,    (* d := d OR s      , d+s    *)
          invert,   (* d := d XOR s     , d/s    *)
          erase);   (* d := d AND NOT s , d*(-s) *)
  BlockDescriptor =
    RECORD
      x, y, w, h: CARDINAL;
    END;

PROCEDURE DisplayChar(VAR bm: Bitmap; VAR lineBlk: BlockDescriptor;
                      VAR f: Font; ch: CHAR);
PROCEDURE Replicate(VAR bm: Bitmap; VAR destBlk: BlockDescriptor;
                    m: Mode; VAR pattern: ARRAY OF WORD);
PROCEDURE BlockTransfer(VAR dbm: Bitmap; VAR destBlk: BlockDescriptor;
                        m: Mode;
                        VAR sbm: Bitmap; VAR sourceBlk: BlockDescriptor);
PROCEDURE DisplayDot(VAR bm: Bitmap; x, y: CARDINAL; m: Mode);

PROCEDURE Proportional(VAR f: Font): BOOLEAN;
PROCEDURE FontHeight(VAR f: Font): CARDINAL;
PROCEDURE FontWidth(VAR f: Font): CARDINAL;
PROCEDURE FontBaseLine(VAR f: Font): CARDINAL;
PROCEDURE CharWidth(VAR f: Font; ch: CHAR): CARDINAL;

PROCEDURE GetSystemBitmap(VAR bm: Bitmap);
PROCEDURE GetSystemFont(VAR f: Font);
PROCEDURE GetDefaultBitmap(VAR bm: Bitmap);
PROCEDURE GetDefaultFont(VAR f: Font);
PROCEDURE GetDotPos(VAR x, y: CARDINAL);
PROCEDURE SetDefaultBitmap(VAR bm: Bitmap);
PROCEDURE SetDefaultFont(VAR f: Font);
PROCEDURE SetDotPos(x, y: CARDINAL);

PROCEDURE CreateBitmap(VAR bm: Bitmap; w,h: CARDINAL; VAR done: BOOLEAN);
PROCEDURE CreateSubBitmap(VAR bm: Bitmap; VAR father: Bitmap;
                          location: BlockDescriptor; VAR done: BOOLEAN);
PROCEDURE ReturnBitmap(VAR bm: Bitmap);
PROCEDURE ShowBitmap(VAR bm: Bitmap);

PROCEDURE LoadFont(VAR f: Font; name: ARRAY OF CHAR; VAR done: BOOLEAN);
PROCEDURE ReturnFont(VAR f: Font);
PROCEDURE GetFontName(VAR name: ARRAY OF CHAR; VAR f: Font);
```

```
  PROCEDURE IsSubBitmap(VAR bm: Bitmap): BOOLEAN;
  PROCEDURE GetRealFather(VAR fbm, bm: Bitmap);
  PROCEDURE GetRealBlock(VAR blk: BlockDescriptor; VAR bm: Bitmap);
    (* maximal block in coordinates of the real father bitmap of bm*)
  PROCEDURE GetMaxBlock(VAR blk: BlockDescriptor; VAR bm: Bitmap);
    (* maximal block in coordinates of the bitmap bm itself*)

END Screen.
```

Implementation Details

The modules Screen, Terminal, TextScreen, and WindowHandler use common resources. Your version of Medos uses either module TextScreenDriver or module ScreenDriver to coordinate the access to these resources. Depending on which communication module is used, the system does not remember the insertion point on the display terminal after a user program which imports Screen terminates (That insertion point is stored in one of the communication modules).

A hidden (empty) module PrivatScreen is imported to force version conflicts when the semantics of some private feature changes. Therefore, any system program using the private features also imports the module PrivatScreen to prevent unknown runtime errors.

Imported Modules

The module is distributed in linked form only.

10.3. TextScreen

The module TextScreen allows writing text at arbitrary positions on the display terminal. The definition is made in such a way, that the use of the module is also possible when the default output is directed to a window. For proper functionality a fixed width font has to be used. With proportional fonts, however, the horizontal positioning (SetPos, GetPos) is done dotwise. FreeChars and ClearChars denote space for maximum width characters if a proportional font is used.

The use of this module is recommended if positioning is necessary. In cases where only a scrolling terminal is needed the module Terminal is preferred. The procedures of the module TextScreen may be used intermixed with the procedures of module Terminal.

The coordinate system used for TextScreen is different to the system used in module Screen. TextScreen uses character width and height as units. The origin (0, 0) is the first character of the top line. (Compared to Screen: unit = dots, origin is bottom, left edge of the screen). The module TextScreen tries to be machine independent, if fixed width (non proportional) fonts are used.

The font and the default rectangle used for output are modified with procedures either from module Screen or from module WindowHandler.

Definition Module

```
DEFINITION MODULE TextScreen;   (* Ch. Jacobi  30.10.80*)
  EXPORT QUALIFIED Write, FreeChars, FreeLines,
    GetPos, SetPos, ClearChars, ClearLines;
  (* Text output and positioning relative to the default rectangle used
     for output. The origin is at the upper left corner of the rectangle.
     Units of the coordinates are characters and lines, starting with 0.
     Uses fixed-width font.  *)
  PROCEDURE Write(ch: CHAR);
  PROCEDURE FreeChars(): CARDINAL;
    (* returns number of free characters in the current line *)
  PROCEDURE FreeLines(): CARDINAL;
    (* returns number of empty lines *)
  PROCEDURE GetPos(VAR line, pos: CARDINAL);
  PROCEDURE SetPos(line, pos: CARDINAL);
  PROCEDURE ClearChars(n: CARDINAL);
    (* clears n positions but at most the rest of the current line *)
  PROCEDURE ClearLines(n: CARDINAL);
    (* clears n full lines but at least the rest of the current line *)
END TextScreen.
```

Imported Modules

The module is distributed in linked form only.

10.4. WindowHandler

This module allows the use of windows. A window is a visible rectangle on the screen where text and binary operations are possible. Such a rectangle simulates a complete display. The windows may be considered as pieces of paper lying on a table. They may be moved; an overlaid window may be put on top.

Typical text operations are writing and positioning; typical binary operations are replicate and blocktransfer. In addition to the procedures for manipulations inside windows, the module supports creation, deletion... of windows and it handles their overlapping.

Window Output Handling Procedures

The output procedures for binary output correspond exactly to the fullscreen output procedures of the module Screen. The text handling procedures correspond to those of the module TextScreen. Further some procedures allow combination of binary and text operations.

- for binary output:
  Replicate, BlockTransfer,
  DisplayDot, DisplayChar

- for text handling:
  WriteChar,
  FreeChars, FreeLines,
  GetPos, SetPos,
  ClearChars, ClearLines

- for combination:
  Clear,
  GetDotPos, SetDotPos

Window Management Procedures

Procedures to create, move, eliminate, ... windows and to handle fonts.

- window management:
  CreateWindow, ChangeWindow,
  OpenWindow, CloseWindow

- font handling:
  LoadFont, SetFont

- visibility operations:
  PutOnTop, SaveWindow,
  NextDown

- dummy procedure used for creation:
  IgnoreWindowSignal

Context Management Procedures

- handling of default output:
  UseForDefault, DefaultWindow

- full screen operations:
  SelectWindow, FullScreen

- use of (Screen) bitmaps as windows:
  OpenBitmapWindow

Remark

The windowhandler may be used without import of the module Screen. Using the same types for Font, Bitmap, BlockDescriptor and Mode serves only for compatibility of *intended* common use of these two modules. So does the procedure OpenBitmapWindow. If these type declarations would be replaced by their actual definition and the procedure OpenBitmapWindow would be eliminated, the module WindowHandler would be independent; it would not be necessary to explain references to other modules.

```
TYPE
  Bitmap;                    (*used for compatibility with module Screen*)
  Font;                      (*picture of the characters*)
  BlockDescriptor =          (*rectangle inside a window*)
    RECORD
      x, y, w, h: CARDINAL;
    END;
```

The Window Type

Windows are represented by an abstract data type. The type Window itself is a pointer to a descriptor. This descriptor describes some public features of the window.

```
TYPE
  Window = POINTER TO WindowDescriptor;
  WindowDescriptor =
    RECORD
      wptr:      WindowHint;       (* do not access *)
      bm:        Bitmap;
      font:      Font;
      overlaid:  BOOLEAN;          (* window not completely visible *)
      outerblk:  BlockDescriptor;  (* in coordinates of the original bitmap *)
      innerblk:  BlockDescriptor;  (* in coordinates of the original bitmap *)
      header:    ARRAY [0..N-1] OF CHAR;
    END;
```

All fields are *read-only.*

`wptr`: for use by the implementation module only.

`bm`: allows combination of operations of module Screen with module WindowHandler. `bm` is the bitmap which corresponds to the inner (writable) area of a window. `bm` is not valid while the window is overlaid.

`font`: this font will be used on "Write" operations for the window.

`overlaid`: flag, if some part of the window is overlaid. A window which is overlaid is put on top before it is written.

`outerblk`: coordinates of the window inclusive its border.

`innerblk`: coordinates of the inner area of the window. Only this area can be modified with write and paint procedures. Note: The coordinates used are relative to the full screen. Windowhandler operations use window-coordinates: 0,0 is at the lower left point inside the window.

`header`: title, written on top of the window.

It is important, that the type Window is a pointer. Window variables may be assigned, function procedures may return windows. The descriptor is allocated by the windowhandler and must not be copied (since the implementation may be position dependent). The fields are read-only and may change at any time.

Allocation of Windows

The simplest method to open (create) a new window is to call the procedure OpenWindow:

```
PROCEDURE OpenWindow(VAR w: Window; x, y, width, height: CARDINAL;
                     name: ARRAY OF CHAR; VAR done: BOOLEAN);
```

`w`: is the new window

`x, y`: coordinates of the (left, bottom) edge of the border

`width, height`: of the window (including the border)

name: title of the window
done: returns success

It is necessary to check the done parameter, there exist several reasons for failures of the operation: not enough memory, too many windows, bad coordinates, etc.

The more complete procedure CreateWindow has two additional parameters:

```
PROCEDURE CreateWindow(VAR w: Window;
            x, y, width, height: CARDINAL;
            name: ARRAY OF CHAR;
            savecontents: BOOLEAN;
            signal: WindowProc;
            VAR done: BOOLEAN);
```

savecontents: selects the method of handling the window when it is overlaid.
signal: is a procedure which is called when some operations occur.

(OpenWindow corresponds to CreateWindow where the signal's are ignored, and the windowhandler saves the contents of the window).

WindowSignals

When windows are modified, the modification is notified to the "owner" program (the program which called CreateWindow). Upon creation of the window a "signal" procedure (of type WindowProc) is specified. The signal procedure is called when the window is modified. The signal procedure may inspect its parameter to get information about which modification of which window occured.

```
TYPE
  WindowSignal = (redraw, save,
                  moved, changed, fontchanged, opened, closed,
                  usedfordefault, enddefault, ...);
  SignalSet  = SET OF WindowSignal;
  WindowProc = PROCEDURE(Window, WindowSignal);
```

Note: The "signal" procedure is not only called during the creation of the window; it is also called later, when some action happens. Signal procedures may be compared to asynchronous interrupts.

Overlapping

Windows may freely overlap each other. It is, however, possible to put any window on the top at any time. There are two methods to redraw a window when it is put on the top (savecontents parameter of CreateWindow).

If the windowhandler saves the contents of the window itself, then it redraws the window itself. The signal procedures may then be used to notify the action.

If the windowhandler does not saves the contents, then the owner of the window has to redraw it whenever the windowhandler requests redrawing by a call of the corresponding signal procedure.

Handling WindowSignals

```
- redraw, save:
        if restoring is done automatically (savecontents=TRUE)
                the signal procedure is called after the window has been saved,
                respectively redrawn;
                (No signal of redraw when Clear puts the window on top)
        if window is not restored automatically (savecontents=FALSE)
                this is a request to the owner to save or to redraw the window;
                save:   when called, the window is visible
                redraw: when called, the window is visible but cleared
```

- moved, changed:
  if restoring is done automatically
    called after the action has been done
  if not restored automatically
    this call replaces a redraw signal; it is a request to draw the window on its new position
- fontchanged, opened:
  called after the action has been done
- closed:
  called before the window disappears
- usedfordefault, enddefault:
  notifies the action
- other values: should be ignored by user programs.

If a window should not signal its operation, it may use IgnoreWindowSignal as its signal procedure.

The Binary Display Procedures

In general, the display procedures work on two rectangles within windows, called the source and the destination.

destination := F(destination, source)

The function F(d, s) depends on the procedure and on a mode parameter.

These blocks are described with the BlockDescriptor type. The coordinate system used for these blocks has its origin at the lower left corner of the window.

Mode: Mode of operation of the basic display procedures.

```
TYPE
  Mode = (replace,    (* d := s          , s      *)
          paint,      (* d := d OR s     , d+s    *)
          invert,     (* d := d XOR s    , d/s    *)
          erase);     (* d := d AND NOT s, d*(-s) *)
```

replace: replaces the destination by the source.
paint: the source is overlaid (added) to the destination.
invert: the destination is inverted, where the source contains ones.
erase: the destination is cleared, where the source contains ones.

Clear: Clears the window.

Replicate: Replicates the bitpattern (pattern) over a rectangle (the destination block) of the window according to the mode.

pattern: This is not a declared type; any variable may be used as pattern, but the first word of the pattern must contain the number of words following. This number is the height of the pattern, its width is 16.

BlockTransfer: is the most general display procedure. It copies the source block into the destination block according to the mode.

DisplayDot: Writes a single dot at the coordinates x, y of the window according to a mode.

DisplayChar: Paints the picture of a character from a font table into a block of a window. This block represents the position (line). DisplayChar updates the block to exclude the just-painted character.

```
PROCEDURE Clear(w: Window);
PROCEDURE Replicate(w: Window; VAR dest: BlockDescriptor;
          m: Mode; VAR pattern: ARRAY OF WORD);
PROCEDURE BlockTransfer(dw: Window; VAR dest: BlockDescriptor;
          m: Mode; sw: Window; VAR source: BlockDescriptor);
```

```
PROCEDURE DisplayDot(w: Window; x, y: CARDINAL; m: Mode);
PROCEDURE DisplayChar(w: Window; VAR lineBlk: BlockDescriptor;
          VAR f: Font; ch: CHAR);
```

Implementation Restrictions

The current implementation does not guarantee correct blocktransfer, if the source and the destination windows overlap each other.

The destination blocks will be clipped to the window's size. The procedure DisplayChar does not recognize whether or not the character fits into the line block.

Text Output

The text output procedures are defined in such a way that they could be used without any respect to the binary procedures. These procedures have their own coordinate system for positioning: The origin (0, 0) is the first character of the first line in the window. Units are the line height and the character width (if the font is not proportional).

For the procedure declarations see the definition module; the procedures correspond to those of the module TextScreen.

The output and the positioning is done with the default font associated to the window. The procedures LoadFont and SetFont are used to exchange the default font of a window, the font which is used by the procedure WriteChar. It is up to the user's responsibility not to return the font explicitly (Procedure ReturnFont of module Screen). The windowhandler's implementation may (but need not) return fonts which were loaded with the procedure LoadFont of the windowhandler.

If a proportional font is used the positioning is a bit clumsy. The procedure GetPos and SetPos will use dot coordinates for the horizontal coordinate. The procedures ClearChars and FreeChars will use maximum width characters.

The procedures GetDotPos and SetDotPos allow positioning text output with the binary (dot) coordinate system.

Default Output

When the windowhandler is initialized, the first window which is created will be used as *default window*. The procedure UseForDefault allows to exchange the default window. Library modules must not call this procedure, since their callers may make assumptions where the default output will be written. The procedure DefaultWindow returns the window where default output will be directed to.

Whether the windowhandler is used or not, need not be known by library modules. If it is used, the default window is also used to display the default output written with module Terminal. Default output should not use knowledge about screen size, because it may be directed to a window. It is possible to get the size of the default window without importing the module WindowHandler (See description of module Screen).

Miscellaneous Procedures

The procedure SelectWindow is used in many applications.

```
PROCEDURE SelectWindow(VAR w: Window; x, y: CARDINAL; VAR found: BOOLEAN);
  (* Returns the window (w) in which the point x,y is visible;
     found: x, y points to a window *)
```

If a point is selected (e.g. with mouse actions) the procedure returns which window contains the point. If an object in a partly overlaid window gets selected, this procedure allows detection whether or not the object was visible: Since the actual Window type is a pointer, it is allowed to compare the returned window with another window and to check if they are the same. (This was the real cause to design the type Window to be a pointer).

The procedure FullScreen returns the *dummy* window which denotes the whole screen. This dummy

window enables drawing outside the windows, it is, however, up to the user's responsibility not to modify other windows.

The procedure PutOnTop moves an overlaid window on the top, this window then may lie over other windows. Do not assume a window to remain on top, any output of any module may cause a change of the window order.

For further procedures see the definition module.

Reference

The Modula-2 manual [1] shows a subset of the windowhandler. This subset fulfills the need of most applications. There is also an example of the use of the windowhandler.

Definition Module

```
DEFINITION MODULE WindowHandler;  (* Ch. Jacobi  9.12.81*)
  FROM SYSTEM IMPORT WORD, ADDRESS;
  IMPORT Screen;

  EXPORT QUALIFIED
    Window,
    Bitmap, Font, BlockDescriptor, Mode,
    WindowDescriptor, WindowSignal, SignalSet, WindowProc,
    CreateWindow, OpenWindow, CloseWindow, ChangeWindow,
    Clear,
    Replicate, BlockTransfer, DisplayDot, DisplayChar,
    WriteChar,
    FreeChars, FreeLines, GetPos, SetPos, ClearChars, ClearLines,
    SetDotPos, GetDotPos,
    LoadFont, SetFont,
    UseForDefault, DefaultWindow,
    PutOnTop, NextDown, SaveWindow,
    SelectWindow, FullScreen,
    OpenBitmapWindow, IgnoreWindowSignal;

  CONST N = 24;
  TYPE
    Window          = POINTER TO WindowDescriptor;
    Bitmap          = Screen.Bitmap;          (* Bitmap; *)
    Font            = Screen.Font;            (* Font; *)
    BlockDescriptor = Screen.BlockDescriptor; (* RECORD x,y,w,h: CARDINAL END; *)
    Mode            = Screen.Mode;            (* (replace, paint, invert, erase);
    WindowHint;
    WindowDescriptor =
      RECORD
        wptr:      WindowHint;       (* do not access *)
        bm:        Bitmap;
        font:      Font;
        overlaid:  BOOLEAN;          (* window not completely visible *)
        outerblk:  BlockDescriptor;  (* in coordinates of the original bitmap *)
        innerblk:  BlockDescriptor;  (* in coordinates of the original bitmap *)
        header:    ARRAY [0..N-1] OF CHAR;
      END;
      (* do not make copies of windowdescriptors;
           all fields are considered read-only,
           the fields may dynamically change values *)
    WindowSignal = (redraw, save,
                    moved, changed, fontchanged, opened, closed,
```

```
                usedfordefault, enddefault, ...);
  (* - redraw, save:
          if not restored automatically
              requests on the owner;
              save: when called, the window is visible
              redraw: when called, the window is visible but cleared
          if restoring is done automatically
              is called after the action has been done;
              No call of redraw when Clear puts the window on top
     - moved, changed:
          if restoring is done automatically
              called after the action has been done;
          if not restored automatically
              this call replaces a redraw signal; it is a request
     - fontchanged, opened:
          called after the action has been done
     - closed:
          called before the window disappears
     - usedfordefault, enddefault:
          notifies the action
     - other values: should be ignored by user programs. *)
  SignalSet  = SET OF WindowSignal;
  WindowProc = PROCEDURE(Window, WindowSignal);

PROCEDURE CreateWindow(VAR w: Window;
          x, y, width, height: CARDINAL;
          name: ARRAY OF CHAR;
          savecontents: BOOLEAN;
          signal: WindowProc;
          VAR done: BOOLEAN);
  (* w: new created window
     x, y: coordinates of the (left, bottom) edge of the border
     width, height: of the border
     name: title of the window
     savecontents: pointwise saved and restored on overlapping
     signal:   procedure called when an event to signal occurs
               Warning: signal must not cause operations on other windows;
                        otherwise infinite loops may be programmed
     done: returns success *)

PROCEDURE OpenWindow(VAR w: Window; x, y, width, height: CARDINAL;
                     name: ARRAY OF CHAR; VAR done: BOOLEAN);
  (* w: new created window
     x, y: coordinates of the (left, bottom) edge of the border
     width, height: of the border
     name: title of the window
     done: returns success
     [OpenWindow is short form of CreateWindow with
      savecontents  := TRUE;  signal  := IgnoreWindowSignal] *)

PROCEDURE CloseWindow(VAR w: Window);

PROCEDURE ChangeWindow(w: Window; x, y, width, height: CARDINAL;
                       VAR done: BOOLEAN);

(* operators *)
PROCEDURE Clear(w: Window);
```

```
PROCEDURE Replicate(w: Window; VAR dest: BlockDescriptor;
                        m: Mode; VAR pattern: ARRAY OF WORD);
  (* a pattern contains the size of its image in the first word,
     followed by the image *)
PROCEDURE BlockTransfer(dw: Window; VAR dest: BlockDescriptor;
                        m: Mode; sw: Window; VAR source: BlockDescriptor);
  (* transferring between overlapping windows is not guaranteed
     in the current implementation*)
PROCEDURE DisplayDot(w: Window; x, y: CARDINAL; m: Mode);
PROCEDURE DisplayChar(w: Window; VAR lineBlk: BlockDescriptor;
                      VAR f: Font; ch: CHAR);

(* Text Windows; positioning only with non proportional fonts *)
PROCEDURE WriteChar(w: Window; ch: CHAR);
PROCEDURE FreeChars(w: Window): CARDINAL;
  (* returns number of free characters in the current line *)
PROCEDURE FreeLines(w: Window): CARDINAL;
  (* returns number of empty lines; the current line not counted *)
PROCEDURE GetPos(w: Window; VAR line, pos: CARDINAL);
PROCEDURE SetPos(w: Window; line, pos: CARDINAL);
PROCEDURE ClearChars(w: Window; n: CARDINAL);
  (* clears n positions but at most the rest of the current line *)
PROCEDURE ClearLines(w: Window; n: CARDINAL);
  (* clears n full lines but at least the rest of the current line *)

(* positioning with dot coordinates *)
PROCEDURE GetDotPos(w: Window; VAR x, y: CARDINAL);
PROCEDURE SetDotPos(w: Window; x, y: CARDINAL);

PROCEDURE LoadFont(w: Window; fontName: ARRAY OF CHAR; VAR ok: BOOLEAN);
PROCEDURE SetFont(w: Window; VAR f: Font);

(* defaults *)
PROCEDURE UseForDefault(w: Window);
  (* denote the system to use this w as default Window*)
PROCEDURE DefaultWindow(): Window;

(* visibility *)
PROCEDURE PutOnTop(w: Window);
PROCEDURE NextDown(w: Window): Window;
  (* w=NIL:  gets the window on top
     w<>NIL: gets the window below of w
             if w is bottom window: returns NIL.
     The window list is ordered according to visibility
     (overlaying of windows); when the visibility changes while
     following the list, windows may be overseen or seen twice ! *)
PROCEDURE SaveWindow(w: Window): BOOLEAN;
  (* the window w will be saved,
     this allows to overlay the visible look of the window on the
     screen, on further WindowHandler operations the window will
     first be redrawn. *)

(* special *)
PROCEDURE SelectWindow(VAR w: Window; x, y: CARDINAL; VAR found: BOOLEAN);
  (* Returns the window in which the point x,y is visible*)
PROCEDURE FullScreen(): Window;
  (* the dummy window which denotes the whole screen; *)
```

```
  PROCEDURE OpenBitmapWindow(VAR w: Window; VAR bm: Bitmap; VAR done: BOOLEAN);
    (* enables WindowHandler operations onto bitmaps (of module Screen).
       A bitmap window cannot be changed or moved;
       it is not included in the window list available
       by calling NextDown;
       it is not tested against visibility nor overlapping *)
  PROCEDURE IgnoreWindowSignal(w: Window; s: WindowSignal);

END WindowHandler.
```

Imported Modules

Screen
Terminal
FileSystem
Others, which either are resident, or linked to the module WindowHandler.

10.5. CursorStuff

Simple *cursor handling* and *menu input* technique. The module supports three different levels of implementation.

The normal, high level use is provided by the procedures TrackCursor, ReleaseCursor, and MenuSelection. The position of the mouse is saved in global variables.

The unsymmetry between these procedures is necessary! TrackCursor returns whenever a button is pressed. Always one button is pressed first, this button is specified in the return value. ReleaseCursor returns when no button is pressed, i.e. after a release of the buttons. This allows to return the set of all buttons, which have been pressed after the procedure was called.

MenuSelection draws a menu and waits until the user selects one of the commands of the menu (by releasing the mouse buttons when the cursor lies on the corresponding command), or does another manipulation. Normally the mouse buttons are pressed before the procedure is called, and the command are selected by releasing the buttons. But, if MenuSelection should first wait till a button is pressed and only after that may exit, the commandstring should start with a "*".

The procedure GetMouse (lower level) simply gets the mouse coordinates into the variables xpos, ypos and reads the state of the buttons. In fact, the procedure gives not only the hardware coordinates, it also maps the hardware coordinates onto the screen size. The moving of the cursor is delayed at the border of the screen (this allows exact positioning onto the border). However, moving the mouse further causes the cursor to wrap around.

On a middle level, the procedures TrackCursor and ReleaseCursor are used again, but an own cursor procedure is used. Calling InstallCursor tells the module to install the users cursor tracking procedure for the next call of TrackCursor or ReleaseCursor. It is not recommended to use this middle level for simple programs, it is, however, used to implement the menu selection procedure. Further, the module exports the procedures ArrowInvert and SimpleMove, which are installed, when no user installations are done.

Definition Module

```
DEFINITION MODULE CursorStuff;  (* Ch. Jacobi  2.8.81*)
  EXPORT QUALIFIED
    MenuSelection, TrackCursor, ReleaseCursor, xpos, ypos,
    InstallCursor, ArrowInvert, SimpleMove,
    buttons, GetMouse;

  (* high level features *)

  VAR xpos, ypos: CARDINAL;

  PROCEDURE MenuSelection(s: ARRAY OF CHAR): CARDINAL;
     (* The menu is painted near to the position (xpos, ypos).
        s: max 9 commands
            "  title | comm-2 | comm-3 | ...| comm-n"     OR
            "* title | comm-2 | comm-3 | ...| comm-n"    (waits first)
        the "*" is used when no button was pressed previously.
        returns:
        0:    not selected:
                a key on the keyboard has been pressed.  The key
                pressed is put back into the input buffer and
                can be read with Terminal.Read;
        1:    not selected:
                tried to select title or outside the menu;
        2..n: the corresponding command has been selected  *)

  PROCEDURE TrackCursor(): CARDINAL;
    (*returns 0: keyboard 1: left button; 2: middle; 3: right;
```

```
      does cursor tracking; returns when the first button (or key)
      is pressed while cursor tracking *)

  PROCEDURE ReleaseCursor(wait: BOOLEAN; VAR but: BITSET);
    (* {1}: left button; {2}: middle; {3}: right; {0}: keyboard;
       if wait is set and no button is pressed at initialization time, then
           ReleaseCursor does cursor tracking till any button
           is pressed, then continues cursor tracking and returns when
           all buttons are released.
           (but returns as soon a key is pressed)
        else
           does cursor tracking; returns when all buttons are
           released (or a key is pressed) *)

  (* lower level features *)

  VAR buttons: CARDINAL;

  PROCEDURE GetMouse;
    (* reads the mouse status; sets xpos, ypos, buttons *)
  PROCEDURE InstallCursor(invertproc, moveproc: PROC);
    (* the next call of TrackCursor or ReleaseCursor uses invertproc to draw
       and to erase the cursor; moveproc to move the cursor *)
  PROCEDURE ArrowInvert;
  PROCEDURE SimpleMove;
    (* does the move through inverting twice *)

END CursorStuff.
```

Restrictions

The global variables are read-only. (Assgning a value to them may destroy information used by other utilities or by the user; the module CursorStuff is not affected.)

Example

```
  LOOP
    (* assume no button is pressed *)
    CASE MenuSelection("* MENU | comm-A | comm-B | EXIT") OF
      0: Read(ch); Write(ch) |
      1: WriteString("please select a command") |
      2: ProcA |
      3: ProcB |
      4: EXIT
    END;
  END;
```

or:

```
  i := TrackCursor(); (* draws and moves the cursor *)
                      (* a button is pressed down *)
  IF i=1 THEN
    (* the (left) button is probably still pressed down *)
    CASE MenuSelection(" MENU | comm-A | comm-B") OF
      0: BusyRead(ch)|
      1: WriteString("no command selected")|
      2: ProcA |
      3: ProcB
```

```
    END;
  ELSE.....
```

Imported Modules

Terminal
Screen
Resident system modules

10.6. CursorRelations

Cursor-positions relative to windows

All procedures take the position where the mousebuttons have been pressed the last time and compute offsets to the window used as parameter. For the characterwise relative positions the default font of the window is used (The font which would be used by the procedure WriteChar). For the procedure DownCharPos and RightCharpos, (0, 0) is the position of the first character of the top line. The procedure UpDotPos and RightDotPos use the dot coordinate system of the window; (0, 0) denotes the position of the leftmost point of the bottom line.

Definition Module

```
DEFINITION MODULE CursorRelations;  (* Ch. Jacobi 10.6.81*)
  FROM WindowHandler IMPORT Window;
  (*IMPORT CursorStuff*)
  EXPORT QUALIFIED
    Inside,
    DownCharPos, RightCharPos,
    UpDotPos, RightDotPos;

  (* all coordinates are relative to
     CursorStuff.xpos, CursorStuff.ypos *)

  PROCEDURE Inside(w: Window): BOOLEAN;
  PROCEDURE DownCharPos(w: Window): CARDINAL;
  PROCEDURE RightCharPos(w: Window): CARDINAL;
  PROCEDURE UpDotPos(w: Window): CARDINAL;
  PROCEDURE RightDotPos(w: Window): CARDINAL;

END CursorRelations.
```

Restrictions

Relations are taken between the cursor position at cursor tracking time and the window position at current time; Changes of the window in the mean time would cause wrong results.

The windows have to be properly initialized.

The results are unpredictable, if the cursorposition is not inside the inner area of the window (Not so for the procedure Inside).

Imported Modules

Screen
CursorStuff
WindowHandler

10.7. WindowDialogue

Window Editor and Input Scheduler.

This module collects the interactive input commands and directs them to the specific windows and their activities. The module implements a so-called *window editor*, which allows interactive modifications of the windows.

The Window Editor

If a mouse button is clicked while the cursor shows to the background, a control menu is shown. The commands shown allow to modify windows. After selecting a command, a window has to be pointed at; the command is relative to that window. Note: Only those actions will happen to a window, which explicitly were installed by a previous call of InstallWindow.

The call feature is dangerous, an interactively selected and executed erroneous program may cause the calling program to abort. To avoid this, the call feature is discarded when the procedure DialogLoop was called with its parameter `comint=FALSE`.

Interactive Commands of the Window Editor

`exit:`
: Exits the most recent invocation of DialogLoop; usually programs then will terminate.

`call:`
: Shows another menu. Selects a utility program and executes it. (Either directory, copy, delete, rename, list or other important routines) If the utility-menu is discarded and the keyboard is pressed, it starts a command interpreter, reads a program name and executes that program inside the window. (Press `ESC` to quit the command interpreter). It is possible to run out of memory or to get other loader or execution error messages. The call command allows creation of an own, temporary window if no window is selected.

`remove:`
: Removes the Window.

`move:`
: Moves the window. Asks the user to point the new location.

`change:`
: Changes the window size or location. Asks the user to point the new diagonal of the window.

`font:`
: Set font used for the window.

`order:`
: Shows another menu with names of windows. Select a window to be placed on top (made visible).

If no menu command is selected, but the mouse button is released while the cursor square points to a window, this window is placed on top (made visible).

To escape from a started command press `ESC`.

Warning

This module is a step forward to an integrated system. In spite of this, the use of such a module is not freely recommended. Actions are executed on the top program level. All actions which return or require some resources will do this on that top level. When the operating system will allow resource allocation on another level and will support some kind of multiprogramming, then the operating system will take over the job of distributing the input to the proper program. It will do it in a nicer way than with these handler-procedures, eg. with processes.

However, if use for combination of independent *programs* is not without problems, this module is very useful for importing independent (library) *modules.*

Definition Module

```
DEFINITION MODULE WindowDialogue;  (* Ch. Jacobi  14.12.81*)

  FROM WindowHandler IMPORT
    Window, WindowSignal, SignalSet, WindowProc;
  EXPORT QUALIFIED
    DialogProc,
    InstallWindow, CreateInstallWindow, RemoveWindow,
    InstallKeyboard,
    DialogLoop, EndLoop;

  (* procedures to install "handlers" which are called on specific
     actions done with or inside a window;
     [the handlers are only called, if the action is caused from within this modul
     all installations are active only while the procedure DialogLoop

    DialogActions:
        1 - left   button pressed inside the window
        2 - middle    "      "       "    "    "
        3 - right     "      "       "    "    "
        4 - commandinterpreter called inside the window
              (executing programs elsewhere is not recorded)
        5 - remove command interactively called
       15 - enables dialog actions caused from higher, non shared levels *)

  TYPE DialogProc = PROCEDURE(Window, CARDINAL);
         (* the window parameter describes with which window an action happened;
            the CARDINAL describes the DialogAction which happened *)

  PROCEDURE InstallWindow(w: Window;
              dialogActions: BITSET;          (*interactive allowed dialog actions
              dialogHandler: DialogProc;      (*called on dialog actions*)
              windowHandlerActions: SignalSet;(*interactive allowed WH actions*)
              VAR done: BOOLEAN);

  PROCEDURE CreateInstallWindow(VAR w: Window; name: ARRAY OF CHAR;
              dialogActions: BITSET;          (*interactive allowed dialog actions
              dialogHandler: DialogProc;      (*called on dialog actions*)
              windowHandlerActions: SignalSet;(*interactive allowed WH actions*)
              savebitwise: BOOLEAN;           (*WindowHandler*)
              signal: WindowProc;             (*WindowHandler; called on WH action
              VAR done: BOOLEAN);
    (* corresponds to :
         interactive position query; CreateWindow; InstallWindow
         dialogActions:        interactive allowance by this module
         windowHandlerActions: INTERACTIVE allowance by this module;
                               (if called explicitly all actions are allowed
                                since the WindowHandler ignores this module)
         dialogHandler:        handler called on all dialog actions
         signal:               handler called on all WindowHandler actions;
                               (don't care about interaction or not)
                               for exact definition see WindowHandler
         savebitwise:          the WindowHandler saves the image on overlapping
```

```
                                for exact definition see WindowHandler*)

  PROCEDURE RemoveWindow(w: Window);
    (* cancel the installation, does not close the window*)

  PROCEDURE InstallKeyboard(p: PROC);
    (* p will be called when a key (keyboard) is pressed *)

  PROCEDURE DialogLoop(comint: BOOLEAN);
    (* starts a main loop which calls the installed handler procedures
       according to the interactive actions of the user.
       it may be stopped interactively or
       programmed by calling EndLoop;
       comint: allows calling the command interpreter interactively*)

  PROCEDURE EndLoop;
    (* stops the dialog loop *)

END WindowDialogue.
```

Fine point: The procedure DialogLoop considers the top line of the screen to be background. This allows selection of the window-editor, if the background is not visible at all.

Imported Modules

Terminal
WindowHandler
FileNames
Program
ProgramMessage
CursorStuff
Screen
Some resident system modules

10.8. ScreenResources0

Gives access to actual resources used by the module Screen.

The module will be replaced in the next version of the operating system. It allows to hide the private exports of the module Screen. Because of this indirection, normal users of Screen will not suffer from the intended changes, the module Screen stays stable (Upward compatible).

If the module Screen is not used at all, the module BitmapVars gives cheaper access to the resident resources used for fonts and bitmaps.

Restrictions

This module will be changed on the next operating system release.

No checks.

Font name of font created with UseFont will not be remembered for a subsequent call of LoadFont.

Definition Module

```
DEFINITION MODULE ScreenResources0;  (*Ch. Jacobi 10.10.81*)
  FROM SYSTEM IMPORT ADDRESS;
  FROM Screen IMPORT Bitmap, Font;
  EXPORT QUALIFIED PToBMD, PToFontFramePointer, UseBitmap, UseFont;

  (* this module is temporary and will freely be deleted *)

  PROCEDURE PToBMD(b: Bitmap): ADDRESS;
    (* returns pointer to (firmware) bitmapdescriptor;
       however this bitmapdescriptor needs NOT be aligned.
       Do not copy the bitmapdescriptor, a future version of module Screen
       may move the actual bitmap. Parameter b must not be subbitmap *)

  PROCEDURE PToFontFramePointer(f: Font): ADDRESS;
    (* returns pointer to font-framepointer (which ignores 4-word descriptor)
       Do not copy the font-framepointer, a future version module Screen
       may move the actual font*)

  PROCEDURE UseBitmap(VAR bm: Bitmap; w,h: CARDINAL;
                      at, size: CARDINAL; VAR done: BOOLEAN);
    (* like CreateBitmap, but uses user specified memory*)
    (* at: (hard)framepointer to bitmap*)

  PROCEDURE UseFont(VAR f: Font; name: ARRAY OF CHAR;
                    at, size: CARDINAL; VAR done: BOOLEAN);
    (* like LoadFont, but uses user specified memory; no checks*)
    (* at: (hard)framepointer to 4-word soft font descriptor*)

END ScreenResources0.
```

Imported Modules

Screen
Other modules, which are linked with module Screen already.

Font Format Conventions

The microcoded DCH instructions needs a framepointer to denote the font. This framepointer points to the actual font table, which is a table of offsets (one for each character), followed by the pictures of the

characters.

The following pseudo declaration should explain the format of the font table as assumed by the firmware.

```
TYPE ChPointer = SELF-RELATIVE OFFSET TO ChDesc.w;
TYPE ChDesc = RECORD
                pat: ARRAY [1..height] OF BITSET;
                w:   ONE'S-COMPLEMENT INTEGER;
                       (* w>=0: width;
                          w <0: -w is index to extend *)
                skip, height: [0..255] (* packed;
                                          skip means empty bottom lines *)
              END;
TYPE fonttable = RECORD
                   chtable: ARRAY [0..255] OF ChPointer;
                   extend:  ARRAY [256..   ] OF ChPointer;
                   characterInfo: ARRAY [..] OF WORD
                 END;
```

On the file and in memory a 4-word header is prefixed to the fonts. This is, however, a software convention only.

This should explain why with PToFontFramePointer a frame pointer is returned, which points behind the 4-word header, but with UseFont, a frame pointer pointing TO the 4-word header must be given.

4-word header:

```
length       (16-bit)
checksum     (16-bit)                          over 4-word header and firm font
baseline     ( 8-bit) | height   ( 8-bits)
proportional ( 1-bit) | maxwidth (15-bits)     two's complement
```

The software allocates also some trailing information in the memory, residing after the actual font table.

Bitmap Descriptor

The following declaration shows the format of bitmap descriptors, as is assumed by the firmware instructions, and returned by the procedure PToBMD. This BitmapDescriptor type is not exported, it is for documentation only. For the old display processor this type is used also to drive the hardware.

```
TYPE
  BitmapDescriptor =
    RECORD
      bitmapAddress: ADDRESS;
      width:         CARDINAL;
      heigth:        CARDINAL;
      position:      CARDINAL;
    END;
```

`bitmapAddress`: is a rotated frame pointer to the memory area used for bitmap. (framepointer rotated 2 bits to left). Therefore, the two least significant bits of `bitmapAddress` denote the memory bank.

`width`: Width of the bitmap in words.

`heigth`: Heigth of the bitmap - 2. Unit is dots.

`position`: Used only in old fashioned display hardware: denotes location of the displayed bitmap on the screen. One bit drives inverting the picture.

Avoid using knowledge of these hardware descriptions. Either the hardware level or the screen software level may be used, but intermixing these levels needs too much knowledge of implementation details.

10.9. BitmapVars

Module BitmapVars can be used by stand alone programs which completely ignore the screen software system around "Screen"; it delivers the addresses of the bitmap, bitmap descriptor and standard font.

All variables fulfill the hardware constraints of the machine; e.g. the bitmapdescriptor is an aligned one, if this is necessary to the machine. Consider the variables to be read-only. Do not make copies of the variables.

Definition Module

```
DEFINITION MODULE BitmapVars; (* Ch. Jacobi 22.2.81*)
  FROM SYSTEM IMPORT ADDRESS;
  EXPORT QUALIFIED BMD, BMA, FA;
  (* this module is freely changed if future implementations of
     screen software would conflict with its definition *)
  VAR
    BMD: ADDRESS; (* address of the bitmapdescriptor seen by the
                     display hardware *)
    BMA: ADDRESS; (* frame pointer to the (resident) bitmap *)
    FA:  ADDRESS; (* frame pointer to the standard font;
                     (points to the firm font; behind 4-word header) *)
    (* read only; do not make copies *)
END BitmapVars.
```

Imported Modules

Resident system modules

11. Library Modules

15.5.82

This chapter is a collection of some commonly used library modules on Lilith. For each library module a *symbol file* and an *object file* is stored on the disk cartridge. 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	`DK.LIB.FileNames.SYM`
Object file name	`DK.LIB.FileNames.OBJ`

List of the Library Modules

InOut	Simple handling of formatted input/output	11.1.
RealInOut	Formatted input/output of real numbers	11.2.
Mouse	Mouse handling and cursor tracking	11.3.
LineDrawing	Line drawing on the screen	11.4.
MathLib0	Basic mathematical functions	11.5.
OutTerminal	Formatted output to the terminal	11.6.
OutFile	Formatted output to files	11.7.
OutWindow	Formatted output to windows	11.8.
ByteIO	Input/output of bytes on files	11.9.
ByteBlockIO	Input/output of byte blocks on files	11.10.
FileNames	Input of file names from the terminal	11.11.
Options	Input of program options and file names	11.12.
Line	Driver for the RS-232 (V24) line interface	11.13.
V24	Driver for the RS-232 (V24) line interface	11.14.

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

11.1. InOut

Niklaus Wirth 15.5.82

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

11.2. 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 to the Modula-2 manual [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.
```

11.3. Mouse

Niklaus Wirth 15.5.82

Library module for handling the mouse and tracking a cursor on the screen. A description of this module is included to the Modula-2 manual [1]. This module does not work together with the screen software package described in chapter 10, but it may be used with module LineDrawing. All exported variables must be considered as *read-only* variables.

Imported Library Module

BitmapVars

Definition Module

```
DEFINITION MODULE Mouse;  (*NW 2.1.82*)
  EXPORT QUALIFIED keys, Mx, My, curOn,
      TrackMouse, FlipCursor, ShowMenu;

  VAR keys:   BITSET;    (*Mouse keys*)
      Mx, My: INTEGER;   (*Mouse and cursor coordinates*)
      curOn:  BOOLEAN;   (*cursor toggle switch; initial value = FALSE*)

  PROCEDURE TrackMouse;
    (*read Mouse coordinates Mx, My, and keys;
      move cursor accordingly*)

  PROCEDURE FlipCursor;
    (*toggle switch for cursor*)

  PROCEDURE ShowMenu(text: ARRAY OF CHAR; VAR selection: INTEGER);
    (*show menu text at current cursor position, then follow the Mouse's
      movements for command selection until menu key is released.
      Selection = 0 means that no command was selected. In the text, command
      lines are separated by "|". Command word have at most 7 characters,
      and there must be at most 8 commands *)

END Mouse.
```

11.4. LineDrawing

Niklaus Wirth 15.5.82

Library module for drawing lines ans writing strings on the screen. A description of this module is included to the Modula-2 manual [1]. This module does not work together with the screen software package described in chapter 10.

Imported Library Modules

FileSystem
BitmapVars
ByteBlockIO

Definition Module

```
DEFINITION MODULE LineDrawing;   (*NW 15.1.82*)
  FROM FileSystem IMPORT File;
  EXPORT QUALIFIED
    width, height, PaintMode,
    Px, Py, mode, CharWidth, CharHeight,
    dot, line, area, copyArea, clear, Write, WriteString, WriteBitmap;

  TYPE PaintMode = (replace, paint, invert, erase);

  VAR Px, Py:     INTEGER;    (*current coordinates of pen*)
      mode:       PaintMode;  (*current mode for paint and copy*)
      width:      INTEGER;    (*width of picture area, read-only*)
      height:     INTEGER;    (*height of picture area, read-only*)
      CharWidth:  INTEGER;    (*width of a character, read-only*)
      CharHeight: INTEGER;    (*height of a character, read-only*)

  PROCEDURE dot(c: CARDINAL; x,y: INTEGER);
    (*place dot at coordinate x,y*)

  PROCEDURE line(d,n: CARDINAL);
    (*draw a line of length n in direction d (angle = 45*d degrees) *)

  PROCEDURE area(color: CARDINAL; x,y,w,h: INTEGER);
    (*paint the rectangular area at x,y of width w and height h in color c
      0 = white, 1 = light grey, 2 = dark grey, 3 = black*)

  PROCEDURE copyArea(sx,sy,dx,dy,dw,dh: INTEGER);
    (*copy rectangular area at sx,sy into rectangle at dx,dy of
      width dw and height dh *)

  PROCEDURE clear;  (*clear the screen*)

  PROCEDURE Write(ch: CHAR);  (*write ch at pen's position*)

  PROCEDURE WriteString(s: ARRAY OF CHAR);

  PROCEDURE WriteBitmap(VAR f: File);

END LineDrawing.
```

11.5. MathLib0

Niklaus Wirth 15.5.82

Library module providing some basic mathematical functions. A description of this module is included to the Modula-2 manual [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.
```

11.6. 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

11.7. 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 procedures of the modules OutTerminal and OutWindow. 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

11.8. OutWindow

Christian Jacobi 15.5.82

This module contains a small collection of output conversion routines for numbers and strings into windows.

Procedures for formatted output onto windows:

Write	writes a character
WriteLn	writes an end of line
WriteS	writes a string
WriteT	writes a string (T = text) with format
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 OutWindow; (* Ch. Jacobi 11.1.81 *)
  FROM SYSTEM IMPORT WORD;
  FROM WindowHandler IMPORT Window;
  EXPORT QUALIFIED
    Write, WriteLn, WriteS, WriteT,
    WriteI, WriteC, WriteO;
  PROCEDURE Write(w: Window; ch: CHAR);
  PROCEDURE WriteLn(w: Window);
  PROCEDURE WriteS(w: Window; s: ARRAY OF CHAR);
  PROCEDURE WriteT(w: Window; s: ARRAY OF CHAR; length: CARDINAL);
  PROCEDURE WriteI(w: Window; value: INTEGER; length: CARDINAL);
  PROCEDURE WriteC(w: Window; value: CARDINAL; length: CARDINAL);
  PROCEDURE WriteO(w: Window; value: WORD; length: CARDINAL);
END OutWindow.
```

Imported Module

WindowHandler

11.9. 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, 'DK.Demo.from', FALSE);
      Lookup(outf, 'DK.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

11.10. 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 *(actual-)count* 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, 'DK.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).

11.11. 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          = MediumName [ "." LocalFileName ] [ 0C | " " ] .
MediumName        = Identifier .
LocalFileName     = [ QualIdentifier "." ] Extension .
QualIdentifier    = Identifier { "." Identifier } .
Extension         = Identifier .
Identifier        = WildcardLetter { WildcardLetter | Digit } .
WildcardLetter    = Letter | "*" | "%" .

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.

Wildcard characters ("*", "%") are only accepted, if the default file name contains wildcard characters.

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.

Note: As all file names contain at least a medium name, don't forget the default medium name in a call to *ReadFileName*.

Examples

`ReadFileName(fn, "DK")`	Default for medium name
`ReadFileName(fn, "DK..MOD")`	Defaults for medium name and extension
`ReadFileName(fn, "DK.Temp.MOD")`	Defaults for all three parts of a file name
`ReadFileName(fn, "DK.*")`	Defaults for medium name and extension, wildcards accepte

Error Message

```
ReadFileName called with incorrect default
```

Imported Module

Terminal

11.12. 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.

11.13. Line

Svend Erik Knudsen 15.5.82

Module *Line* is used for reading or writing characters over the RS-232 asynchronous line adapter. Character 36C (= *eol*) is converted into CR LF if it is written to the line and vice versa.

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

  EXPORT QUALIFIED Read, BusyRead, Write;

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

  PROCEDURE Write(ch: CHAR);

END Line.
```

Explanations

Read(ch)

Procedure *Read* gets the next character from the line and assigns it to *ch*. The character sequence CR LF (15C 12C) is converted to *eol*.

BusyRead(ch)

Procedure *BusyRead* assigns 0C to *ch* if no character is received on the line. Otherwise, the received character is assigned to *ch*.

Write(ch)

Procedure *Write* writes the given character on the line. Character *eol* is hereby converted to CR LF (15C 12C).

Restrictions

Module *Line* must not be used together with module *V24* (see chapter 11.14).

The received characters might be lost, if procedure *Read* or *BusyRead* are not called frequently enough. Buffering cannot easily be provided because the line adapter generates no interrupts.

11.14. V24

Svend Erik Knudsen 15.5.82

Module *V24* is used for reading or writing characters over the RS-232 asynchronous line adapter. No character conversions are implied in the routines.

```
DEFINITION MODULE V24;        (* Medos-2 V3  S. E. Knudsen  1.6.81 *)

  EXPORT QUALIFIED BusyRead, Read, Write;

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

  PROCEDURE Write(ch: CHAR);

END V24.
```

Explanations

Read(ch)

Procedure *Read* gets the next character from the line and assigns it to *ch.*

BusyRead(ch, got)

Procedure *BusyRead* assigns FALSE to *got* if no character is received on the line. Otherwise, the received character is assigned to *ch.*

Write(ch)

Procedure *Write* writes the given character on the line.

Restrictions

Module *V24* must not be used together with module *Line* (see chapter 11.13).

The received characters might be lost, if procedure *Read* or *BusyRead* are not called frequently enough. Buffering cannot easily be provided because the line adapter generates no interrupts.

12. Modula-2 on Lilith

Leo Geissmann 15.5.82

Differences between programming for 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 Lilith specific low-level features.

WARNING

It must be considered that all these features should be applied with **utmost care**, and that their use might be in *opposition* to the basic software, e.g. Medos-2, screen software, etc.

12.1. Code Procedures

The only extension of Modula-2 for Lilith is the addition of so-called *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 available routines that are micro-coded at the level of Modula-2.

This facility is reflected by the following extension to the syntax of the procedure declaration (see Modula-2 report in [1], chapter 10):

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

The following are typical examples of code procedure declarations:

```
PROCEDURE get(channel: CARDINAL; VAR info: WORD);
  (* input info from channel *)
CODE 240B (* READ *)
END get

PROCEDURE put(channel: CARDINAL; info: WORD);
  (* output info to channel *)
CODE 241B (* WRITE *)
END put
```

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 that the parameters correspond to the instructions. **Responsibility is left to the programmer.**

12.2. The Module SYSTEM

The module *SYSTEM* offers some further tools of Modula-2. Most of them are implementation dependent and/or refer to the given processor. Such kind of tools are sometimes necessary for the so

called *low-level programming*. SYSTEM contains also types and procedures which allow a very basic coroutine handling.

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

For more detailed information see Modula-2 report in [1], chapter 12.

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 no or not all tag constants are specified, then the remaining variant with maximal size is assumed.

12.3. Data Representation and Parameter Transfer

12.3.1. Data Representation

The basic memory unit for data is the word. One word contains 16 bit. 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

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

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

BOOLEAN

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

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..255] (octal [0B..377B]). The compiler accepts character constants in the range [0C..377C].

REAL

Represented in two memory words (32 bit). 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

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

Represented in one memory word.

WORD

Represented in one memory word.

ADDRESS

Represented in one memory word. The value is an absolute address.

PROCESS

Represented in one memory word. The value is an absolute address pointing to a process descriptor.

12.3.2. 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 (no matter of size). The procedure allocates the memory space and copies the parameter.

For *dynamic arrays* also 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.

13. Hardware Problems and Maintenance

Jirka Hoppe 19.5.82

13.1. What to Do if You Assume some Hardware Problems

There is one significant difference between software and hardware. A piece of software keeps the same quality, if left untouched, a piece of hardware will die sooner or later (the mean time between failure for the HB D120 disk drives was estimated to be about 5000 hours). For this reason you must expect at any time a hardware failure of your system. Any user can influence the reliability of Lilith by observing some basic rules:

1/ The computer must be cooled properly so that the inner temperature never exceeds 40 degrees Celsius. Keep therefore the bottom of the computer (entry of the air flow) uncovered. Place your computer so that there will be a minimal distance between the 'fan side' of the computer (exit of the air flow) and the next wall of at least 15 cm. Adjust your air conditioning to a lower temperature.

2/ The disk cartridges are extremely sensitive to dust. Keep your cartridges always in a plastic bag, avoid any dirt and wipe the bottom of your cartridge before inserting it into the drive. Leave your cartridge in the drive, avoid unnecessary removing. Put always a dummy grey cartridge into the drive when you remove the disk cartridge. The dummy cartridge protects your drive from dust.

The hardware problems may be classified into four groups:

a/ it is impossible to boot the computer
b/ one peripheral device (mouse, network,..) does not work
c/ everything 'works' but unreliable
d/ computer works, but there are some problems with your disk

If you have some troubles of the class a/ or b/ please check that all connectors (power, keyboard, mouse, display, network) are plugged in correctly and that all control lights on the computer cabinet (red for power and white for a "not" active disk) and on the keyboard (keys **ON LINE** and **FORMAT**) are on.

There is a possibility that the boot file on your disk is damaged. Put your disk into another Lilith and try a bootstrap. If the bootstrap is still not successful, check your disk using the *DiskCheck* program (load *DiskCheck* from another disk and exchange cartridges when the program is waiting for input) and get a new boot file.

If everything seems to be all right but you have still problems => call the maintenance group. If you have any problems where you believe it may be caused by an unreliable hardware, please check first your software before you start any panic action.

Your system disk contains a hardware test program *SYS.HardwareTest* that tests thoroughly the processor and the main memory. You should run this program not only if you assume some hardware problems but always when you drink coffee or whisky on the rocks instead of working. The command file *SEK.Idle.COM* will be executed if you leave the computer idle for more than 3 minutes. Put the command *SYS.HardwareTest* on this file and the hardware will be tested regularly. If the test program runs for a long time without giving a single error message, there is a great chance that the hardware is OK. If the test program displays error messages, please note them and call the maintenance group.

If you assume that not the processor but any peripheral device is causing reliability problems, call the maintenance group. They have a collection of test programs for all devices.

Generally: handle your computer like you handle your girl (or boy) friend. Don't kick it, don't use a hammer, don't spill your coffee into the processor, etc. Please talk to the computer tenderly and pet him.

13.2. DiskCheck

If you have any problems with your disk cartridge, run program *DiskCheck*. It checks the directory and possibly all sectors of your disk. The program asks you if you would like to try any fixes. Answer with Y if you want. If you try any fixes, the program will ask you if you would like to confirm them. Answer always with Y unless you have a large number of damaged sectors. There will be some errors that you cannot fix using program *DiskCheck*. In this case find somebody who knows how to use the *DiskPatch* program.

There are the following possible error messages:

- `disk error on sect = nnnnn xxxxx`
  `file name = yyyyy`

  nnnnn is the sector number
  xxxxx is a further description of the error(e.g. time out, parity err, ..)
  yyyyy is the name of the file that contains this sector

  A hardware error occured during the read or write operation. The program asks you if you want to fix it, answer with 'Y' unless the sector belongs to the 'FS.Directory' file; for the 'FS.Directory' file => call DiskPatch

- `bad fno = nnnnn on sector mmmmm`
  => call DiskPatch

- `wrong page pointer, dir sector nnnnn, pointer mmmmm`
  `file name = yyyyy`
  try to delete yyyyy and immediately bootstrap, if not successful => call DiskPatch

- `double allocated page, page = nnnnn, dir entries mmmmm ooooo`
  `file name = yyyyy  file name = zzzzz`
  delete yyyyy and zzzzz and immediately bootstrap

- `bad pointer: name->block directory, sector = nnnnn`
  => call DiskPatch

- `name dir points to a free block, name dir sect = nnnnn,`
  `block dir sect = mmmmm  file name = yyyyy`
  try to delete yyyyy and immediately bootstrap, if not successful => call DiskPatch

- `version # conflict, name dir sect = nnnnn, version = mmmmm,`
  `block dir sect = ooooo, version = ppppp`
  `file name = yyyyy`
  => call DiskPatch

After the program has checked the directory, it asks whether you would like to check all sectors on hardware errors. Answer with Y if you want.

If you made any fixes on your disk, bootstrap the computer and run program DiskCheck again to be sure that your disk is all right now.

13.3. DiskPatch

Jirka Hoppe 14.5.82

Warning: The program DiskPatch should be used by professional users only. You can destroy by a few key strokes the entire disk and there will be no possibility to recover this disaster. If you are not completely sure how to use this program => keep hands off.

13.3.1. Introduction

The program DiskPatch allows you to initiate disk cartridges for D120 and D140 Honeywell Bull drives and to recover from some crashes of either the file system or the hardware. All actions to fix a disk are done manually and the user must know the structure of the file system.

After DiskPatch is started, a greeting message is displayed on the screen and you are asked whether you know how to use this program. Answer with Y if you are a professional user. (If you type anything else than Y the program will stop!!!). Next, you are asked to exchange the current disk for the disk you would like to fix. If the current disk should be fixed, switch the white disk-ready-button off and back on. Next, the program asks you, whether you are definitly sure that it is the right disk. Check again the cartridge number and answer with Y if everything is ok. Now a set of commands is available. You may get the menu by typing '?'. Every command is activated by typing a key character from the menu. The octal representation is used for all numbers .

13.3.2. Commands

B bad block link

This command is used to insert a damaged sector into a 'FS.BadPages' file. Type the sector number in octal.

In case of problems there may be the following error messages:

`problems...not done` => hardware problems when reading or writing directory

`too many bad blocks` => the BadPages file is already full

`already linked` => the sector is already inserted in the BadPages file

C character dump

The last read sector is displayed in ASCII characters. Nonprintable characters are displayed as '.'

D disk switch; removable/fixed

This command may be used only for the D140 disk. It switches the cartridges used for next commands. First the type of the current disk is displayed. Next you can change the disk. Type 'r' if the next operations should be performed on the Removable cartridge, type 'f' if the fixed disk should be used.

G get file to sector

This command finds a name of a file containing a specified data sector. If the sector is not in use a message *not allocated* is displayed.

F find name

This command finds an internal file number of specified file name. Type the full file name, (you may use DEL) and close the string by RETURN. If the file is found, the internal file number is displayed, if the file does not exists, `not found` is displayed.

I inspect

The last read sector may be inspected and changed in a octal representation. Type the address and the content of the specified word will be displayed. Now you may type ':' to enter the change mode and to specify the new value. By typing ',' the next address will be displayed, typing any other key will terminate the inspect command.

K consistency check

The constistency of the directory is checked. For detailed information see the description of the

DiskCheck program.

L illegal block build up

This command is used to write and to read the entire disk and to find damaged sectors and situations, where more than one invalid sector is located on a single track. Such sectors will be inserted later into the FS.BadPages file using the 'S' command. (After you have run the 'Z' command: no problems, the program will help you.) The execution of the 'L' command takes about 15 minutes. At the end a statistic about your disk is displayed. It contains the number of single bad sectors (they will be handled by the disk driver so that you will not notice any difference), and their position if they are located within the fixed files. The same information is displayed for ***double bad sectors*** (see above). If there are some ***double bad sectors*** located in fixed files you may not use this disk cartridge at all!!!. When in doubt consult some specialist.

WARNING: THIS COMMAND DESTROYS THE ENTIRE DISK INFORMATION!!!!

N name directory update

This command is used to inspect and change the name directory and has 5 subcommands. Type 'ESC' to get back to the main menue:

display The current directory sector is displayed. Such a sector contains 8 file entries (0..7). Each entry consists of:

name - 24 characters; left adjusted, right filled with **BLANKs**,

kind - (0=> file is not in use, 1=> file is used),

file number - acting as a pointer to the 'block directory',

version number - which must be the same as the version number in the block directory

The number in parentheses gives the address of the information, that may be used by the inspect command.

read sector This procedure reads a name directory sector that will be used for next operations. You may either type a sector number or '=' to get the same sector again or ',' to get the next sector.

inspect The same as Inspect from the main menu.

name change The name entry will be changed by this command. The procedure asks you for the index of the name (0..7), displays the old name and asks for the new name. The name input is closed by **RETURN**.

write sector The sector is written back onto the disk. You may either type a sector number or '=' to write to the same sector as read. The program asks you to confirm the sector number. Type 'y' if you agree that this sector should be written.

O octal dump

The current sector (opened by the **R** command) will be displayed in octal mode.

R read sector

This command reads a disk sector that will be used for next operations. You may either type a sector number or '=' to get the same sector again or ',' to get the next sector.

S set illegal blocks into directory

The information as computed by the 'L' command is inserted into the file FS.BadPages. If no double illegal blocks were found by the 'L' command, a message `no double illegal blocks` is displayed.

T transfer

This procedure is used on D140 disk drives only to transfer a number of sectors between two disks. You first have to specify the source disk. Type 'f' for fixed disk or 'r' for removable disk. Now specify in octal numbers the low and high limit of the region to be transferred.

WARNING: If you would like to transfer the entire disk and the target disk contains any bad blocks (see 'L' command) you will destroy the BadPages file and you may run into troubles!!

U update directory

This command is used to inspect and change the block directory and has 4 subcommands; Type 'ESC' to get back to the main menue:

display The current block directory sector is displayed.

There is the following relevant information being displayed.

file# - is the internal index of the file; it must be the same as the relative sector number in the directory

version Nr - version number of the file, it must be the same as the version number of the same file in the name directory

kind - 0 => file is free; 1=> file is in use

length.block - how many page blocks are used

length.byte - number of bytes in the last used page

page table - pointers to data sectors. Data sectors are assigned in blocks of 8 sectors. To compute the physical address make the following computation

physical.address := (page DIV 13) * 8 decimal

physical.address := (page DIV 15) * 10 octal

A pointer to an unused page has a value 167340 (octal).

The number in parentheses of each identifier of the display command gives the address of the information, that may be used by the inspect command.

read sector This command reads a directory sector that will be used for next operations. You may either type a sector number or '=' to get the same sector again or ',' to get the next sector.

inspect The same as Inspect from the main menu

write sector The sector is written back on the disk. You may either type a sector number or '=' to write to the same sector as read. The programs asks you to confirm the sector number. Type 'y' if you agree that this sector should be written.

W write sector

The sector is written back on the disk. You may either type a sector number or '=' to write to the same sector as read. The program asks you to confirm the sector number. Type 'y' if you agree that this sector should be written.

Z zero directory

The directory is initialized.

WARNING: THIS COMMAND DESTROYS THE ENTIRE DISK INFORMATION!!!!

+ octal calculator

This is a simple calculator able to add, subtract, multiply and divide two octal numbers. Type ESC to exit to the main menu.

13.3.3. How to Initialize a New Cartridge

Each cartridge must be initialized in order to find damaged sectors or situations where more than one illegal block is encountered on a single track.

To initialize a cartridge perform the following steps:

Run the 'L' command

Run the 'Z' command

If the 'L' command displays any double bad blocks run the 'S' command.

Record the *bad block* information in a log book in order to have an overview of your cartridges. If there are some 'double bad sectors' located in fixed files you may not use this disk cartridge at all!!!. When in doubt consult some specialists.

13.3.4. The most Frequent Problems with Your Disk

The following section gives you an overview of the most frequent problems with your disk and gives proposals how to fix them. Some of these problems will be encountered during the bootstrap sequence, where the file system refuses to complete the bootstrap, since the directory is out of order; some other

problems will be detected either by the *K consistency* command or the *DiskCheck* program.

Most problems can be solved, when the damaged file is entirely removed by setting the *kind* field of the damaged file in both directories (name and block) to **0**. This is however a rather brutal, but simple method.

It is Impossible to Boot the Machine

There is message on the screen

`DiskSystem.FileCommand: bad directory entry: fno= nnnn; read fno = mmmm`

Solution: read the directory sector nnnn using the commands 'U' (update directory) and 'R'; correct the file number (address 1) to the value nnnn and write the sector back using the 'W' command'. Find the name of the corresponding file by entering the 'N' (name directory) command and reading the sector nnnn DIV 10 (octal). The file name will be found on the position nnnn MOD 10 (octal). This file may contain garbage. Boot the system and check this file.

There is message on the screen

`DiskSystem.OpenVolume:bad page pointer; fno = nnnn, pageno = mmmm, page = oooo`

Solution: All page pointers must be dividable by 15 (octal). Enter 'U'(update directory) command, read the sector nnnn and check the pointer mmmm using the '+' (calculator). Replace the bad pointer by the NIL value 167340 (octal). If too many pointers are damaged, read another directory sector, change the file number, set length to zero, put all page pointers to NIL and write the sector on nnnn.

Find the name of the file as described above and check the file for garbage.

Problems Found by Program DiskCheck or Command Consistency:

double allocated page

A single page belongs to two files. Delete both files and immediately bootstrap!!!!

name dir points to free block, name dir sect=nnnn

Enter the 'N' (name directory) command; read the sector nnnn DIV 10 (octal), and set the *kind* of the file on the position nnnn MOD 10 (octal) to zero. Write the sector back.

version# conflict

The version number of a file in the *name* and *block* directory must be the same. Change one version number so that they match.