Ceres Handbook

A Guide for Ceres Users and Programmers

Editors:
Frank Peschel
Matthias Wille

December 1986

The type setting of this handbook was done using Lara (version 1.4) on the Lilith computer.
All figures (except bitmaps) are directly integrated in the Lara document file.

Editors: Frank Peschel, Matthias Wille
Date of Publication: December 1986
Publisher: Institut für Informatik ETH Zürich

Table of Contents

30.11.85

1. Introduction 1001
   - 1.1. Handbook Organization 1001
     - 1.1.1. Overview of the Chapters
     - 1.1.2. Page Numbers
   - 1.2. Overview of Ceres 1002
     - 1.2.1. Software
     - 1.2.2. Hardware
   - 1.3. References 1003

2. Running Ceres 2001
   - 2.1. Getting Started 2001
   - 2.2. Termination of a Session 2002

3. Running Programs 3001
   - 3.1. The Command Interpreter 3001
     - 3.1.1. Program Call
     - 3.1.2. Typing Aids
     - 3.1.3. Loading and Execution Errors
   - 3.2. Command Files 3004
   - 3.3. Program Loading 3004

4. Things to Know 4001
   - 4.1. Special Keys 4001
   - 4.2. File Names 4001
     - 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 4003
   - 4.4. The Mouse 4003

5. The Editor 5001
   - 5.1. Introduction 5001
   - 5.2. Starting the Editor 5002
   - 5.3. Keyboard and Mouse, Special Keys 5002
   - 5.4. Text Entry and Selection 5003
   - 5.5. Scrolling 5004
   - 5.6. Window Commands 5004
   - 5.7. Menu Commands 5005
   - 5.8. Error Recovery 5005

6. Utility Programs 6001
   - 6.1. directory 6002
   - 6.2. delete, protect, and unprotect 6004
   - 6.3. copy and rename 6005
   - 6.4. list 6006
   - 6.5. inspectfile 6007
   - 6.6. compare 6008
   - 6.7. xref 6009
   - 6.8. link 6010
   - 6.9. decobj 6011
   - 6.10. hermes and fhermes 6012
   - 6.11. futil 6013

6.12. asm32 6014
6.13. vfinit and vfopen 6015

7. The Compiler 7001
7.1. Glossary and Examples 7001
7.2. Compilation of a Program Module 7002
7.3. Compilation of a Definition Module 7002
7.4. Symbol Files Needed for Compilation 7002
7.5. Compiler Output Files 7002
7.6. Program Options for the Compiler 7002
7.7. Module Key 7003
7.8. Program Execution 7003
7.9. Value Ranges of the Standard Types 7003
7.10. Restrictions 7004
7.11. Compiler Error Messages 7005

8. The Debugger 8001
8.1. Introduction 8001
8.2. Starting the Debugger 8002
8.3. Global Commands 8003
8.4. Local Commands 8003
8.5. Debugger Command Summary 8004

9. The Medos-2 Interface 9001
9.1. Module FileSystem 9002
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. Creation, Opening, and Closing of Files
9.1.4.4. Permanency of Files
9.1.4.5. Protection of Files
9.1.4.6. Reading, Writing, and Modifying Files
9.1.4.7. Examples
9.1.4.8. Directory Information
9.1.5. Implementation of Files
9.1.6. File Representation
9.1.6.1. Main Characteristics and Restrictions
9.1.6.2. System Files
9.1.6.3. Error Handling
9.2. Module Processes 9020
9.2.1. Introduction
9.2.2. Definition Module Processes
9.2.3. Process Concept of Medos-2
9.2.4. Explanations
9.2.5. Implementation Notes
9.2.6. Examples
9.3. Module Program 9023
9.3.1. Introduction
9.3.2. Definition Module Program
9.3.3. Execution of Programs
9.3.4. Error Handling
9.3.5. Object Code Format
9.4. Programs 9028
9.5. Heap 9030

9.6. SEK 9031
9.7. TerminalBase 9033
9.8. Terminal 9034
9.9. Users 9035
9.10. Clock 9037

10. Screen Software 10001
10.1. Summary 10001
10.2. CursorMouse 10002
10.3. Menu 10003
10.4. Windows 10004
10.5. TextWindows 10005
10.6. GraphicWindows 10007
10.7. DisplayDriver 10009
10.8. RasterOps 10010
10.9. Fonts 10011

11. Library Modules 11001
11.1. InOut 11002
11.2. RealInOut 11004
11.3. LongInOut 11005
11.4. MathLib0 11007
11.5. ByteBlockIO 11010
11.6. FileNames 11012
11.7. Options 11014
11.8. V24 11016
11.9. Profile 11017
11.10. String 11018

12. Modula-2 on Ceres 12001
12.1. Implementation Details 12001
12.1.1. Forward References
12.1.2. Type Transfer
12.1.3. Procedure Parameters
12.1.4. Code Procedures
12.1.5. Standard Procedures and Functions
12.1.6. Numeric Constants
12.2. The Module SYSTEM 12002
12.3. Data Representation 12003

13. Hardware Problems and Maintenance 13001
13.1. What to Do if You Assume some Hardware Problems 13001
13.2. DiskCheck and DiskPatch 13002
13.3. DiskCheck 13002
13.4. DiskPatch 13004

1. Introduction

The *Ceres* workstation has been developed by Hans Eberle in a project headed by Niklaus Wirth. The name Ceres is an acronym and stands for Computing Engine for Research Enginnering and Sience. The Ceres 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.

As Ceres is a follower of the Lilith workstation, the current operating system of Ceres is an improved version of the Lilith operating system Medos-2. This implies that all software running under Medos-2 can easily be ported onto Ceres. Most of the library modules and utility programs implemented on the Lilith are available on Ceres. Thus we used the Lilith handbook as a basis for this document. Among the numerous people who have contributed to the Lilith-documentation as well as the Lilith-software are Leo Geissmann, Jürg Gutknecht, Werner Heiz, Jirka Hoppe, Svend E. Knudsen, Eliyezer Kohen, Hans-Ruedi Schär, Christian Vetterli, Thorsten v. Eicken, Bernhard Wagner, Werner Winiger and Niklaus Wirth.

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 Ceres.

Chapter 2 gives instructions on how Ceres 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 Ceres 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 Ceres-specific features of Modula-2.

Chapter 13 describes procedures to follow if Ceres 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 *thousands 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 Ceres

1.2.1. Software

The Ceres workstation provides as its major language Modula-2, which is defined in the Modula-2 Manual [1]. For time-critical system applications like I/O-drivers it is possible to write assembly code for the Ceres processor NS32032. The instruction set is defined in [2]. The specialities of the Ceres-implementation of Modula-2 are mentioned in chapter 12 of this handbook.

The resident operating system on Ceres 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. It is the same operating system as that running on the Lilith workstation. The original Lilith implementation is described in [3] while more information about the Ceres version can be found in [4].

For the programmer on Ceres a software development package is provided containing a versatile text editor, the fast Modula-2 single-pass-compiler and as an aid in program testing a post-mortem debugger. An assembler which produces Modula-2 compatible object code is also part of the package.

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 Ceres software.

1.2.2. Hardware

The Ceres hardware consists of a 32-bit processor based on the National Semiconductor Series 32000 chip set, primary memory, secondary memory and miscellaneous input and output devices. These include a high resolution display, a serial keyboard and a mouse pointing device, a RS-232-C serial line interface and a RS-485 serial line interface. A detailed description of the Ceres hardware can be found in [5]. However, the following paragraph will give a rough overview.

The heart of the Ceres computer is a National Semiconductor NS 32032 32-bit microprocessor. Two slave processors add the capabilities of virtual memory and the support of floating point arithmetic. The processor runs at a clock rate of 10 MHz and it has an addressing range of 16 MBytes.

The primary storage of Ceres contains 2 MBytes of dynamic RAM, 256 KBytes of video RAM and 32 KBytes of ROM. The ROM memory contains bootstrap and diagnostic software. The secondary storage of Ceres consists of a winchester hard disk drive and a floppy disk drive. The 5 1/4" hard disk has a formatted capacity of 40 MBytes and average access time of 40 ms. For backup purposes a 3 1/2" floppy disk is available with a formatted capacity of 720 KBytes.

The display of Ceres is a high resolution raster scan monitor. It can display 819'200 dots stored in a matrix called bitmap which is 1024 dots wide and 800 dots high. The picture is refreshed with a frequnency of 62.15 Hz (non-interlaced) which results in a flicker-free image. The bitmap information is

stored in a separate, dedicated memory implemented with video RAMs. Ceres has two such bitmaps which may be switched back and forth to allow highspeed graphics like dragging and animation.

Ceres has a standard serial ASCII keyboard with VT100 key layout. The optomechanical mouse has three push-buttons and a resolution of 380 pulses per inch. The standard RS-232-C interface works with data transfer rates ranging from 50 up to 38400 bps. A higher transmission speed can be obtained with the two RS-485 serial ports which allow data rates up to 230.4 Kbps. In a multipoint configuration this interface allows to implement a low cost computer network.

1.3. References

[1] N. Wirth: Programming in Modula-2,
3rd. Edition, Springer-Verlag, Heidelberg, NewYork, 1985.

[2] N. Wirth: The personal computer Lilith,
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.

[3] S.E. Knudsen: Medos-2: A Modula-2 Oriented Operating System for the Personal Computer Lilith,
Ph.D. thesis No. 7346, ETH Zürich 1983

[4] F.Peschel, M.Wille: Porting Medos-2 onto the Ceres workstation,
Institut für Informatik, ETH Zürich, to be published

[5] H. Eberle: The personal computer Ceres,
Institut für Informatik, ETH Zürich, to be published

2. Running Ceres

2.1. Getting Started

Ceres is switched on using the *green power switch* on the front panel of the cabinet. After power on the disk needs about 10 seconds to have the correct speed. Now the bootstrap of the machine starts automatically.

The resident operating system Medos-2 is now loaded from the winchester disk. After successful loading it first displays a *version message*, e.g.

```
Ceres/Medos V5.2C (1.12.86)
```

The operating system now makes now some *initializations*, i.e. the consistency of the file system is checked. When the last session has been properly terminated (see 2.2), this check is not needed. However, a full check lasts for about 20 seconds. When the check has been passed a *dot* appears behind the version message and the *command interpreter* is started. After telling its version it displays a login message of the form

```
Login on Tuesday, 1.12.86, 10:00:00
```

and prompts for the user id. After entering the user id which consists of two numbers denoting the group and the member number a password has to be entered. The password is a character string allowing access to various services. After the password 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.

2.2. Terminating a Session

To leave the filesystem in consistent state, a session has to be terminated using the program shutdown. This program saves the state of all memory resident administration data of the filesystem into a known area on the disk.

After running shutdown, rebooting is necessary for a next session. This is accomplished by pressing the small reset button that is located under the power switch. About three seconds after the reset the bootstrap is started following the same steps as described in 2.1.

In seldom cases a faulty program may block the whole system and thus makes rebooting necessary. In order to avoid the time consuming check of the filesystem the above mentioned administration data are periodically saved by the command interpreter during idle time.

2.3. Individual Bootstraps

A feature that is of special interest to the system designer is the possibilty to specify individual sources for the system bootstrap, i.e. the loading of other programs instead of Medos-2. For this reason, Medos-2 provides two preallocated bootfiles that are accesible using the normal filesystem utilities, namely PC.BootFile0 and PC.BootFile1. The normal, i.e. automatic, bootstrap is made using PC.BootFile0.

An individual bootstrap is initiated by pressing the reset button and selecting the bootstrap source by typing one of the following characters:

CTL-A	boot from PC.BootFile0 (same as auto boot)
CTL-B	boot from PC.BootFile1
CTL-F	boot from floppy disk (service programs only).

Note, that the character must be typed within 3 seconds after resetting Ceres. The program contained in the specified bootfile or on the floppy is immediately started after loading.

3. Running Programs

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. 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.OBN
Second file name	DK.SYS.directory.OBN

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. The dump may be inspected with program *inspect.*

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, 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*
too many modules	*maximal number of loaded modules exceeded*
heap full	*program needs too much memory space*
code version mismatch	*code of a module is not from the same generation*
syntax error in object file	*a file may be damaged*
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. In such a case several errors may have occurred, e.g. a run time overflow, the program may have called the standard procedure HALT, or the user may have killed the program by typing CTRL-C. In all of these cases, the operating system first causes the memory contents to be dumped on the *dump files* of the disk. These dump files namely PC.DumpFile0 and PC.DumpFile1 may be inspected with the post-mortem debugger *inspect*.

The following types of execution errors may be reported

killed	*program was killed by CTRL-C*
HALT called	*standard procedure HALT was called*
function exit not by RETURN	*function not terminated by a RETURN statement*
space error	*available memory space exceeded*
REAL error	
zero divide error	
CASE range error	
range error	
address error	*illegal pointer access*
priority error	*call of a procedure on lower priority*
SVC error	*an unknown SVC function code has been specified*
undefined instruction executed	*illegal instruction, i.e. the code may be overwritten*
undefined execution error code	

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

```
- DiskSystem.OpenVolume: bad pointer in directory file
```

This example indicates that procedure OpenVolume in module DiskSystem has detected that the descriptor of the directory contains a bad page pointer

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 execution 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

End command file           end of command file substitution
*
```

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 is not loaded correctly or a program terminates abnormally. Command files *must not be nested.*

The commandfile facility may also be used to execute certain initialization commands after starting a session on the Ceres. The command interpreter executes the command file 'SEK.Start.COM' after the login procedure has been finished. When the machine is idle, i.e. no keystrokes appears for about 3 minutes, the command interpreter looks for a command file 'SEK.Idle.COM' and initiates its execution.

3.3. Program Loading

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 9.3.).

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 OBN). 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.OBN
Second file name   DK.LIB.BufferPool.OBN
```

A module cannot be loaded twice. If an imported module is already loaded with the resident system (e.g. module *FileSystem*), the loader links the program to 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 6.8.

4. Things to Know

This chapter provides you with information about different things which are worth knowing if you want to get along with Ceres. 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.

?
: 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 .
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 used to access files on other media. To assign a file on the disk of your Ceres 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. The FileIdent is device-dependent, i.e. its length and the characters it contains may differ among the devices. This is necessary to allow the transparent access to filesystems with other naming conventions. A hierarchy of directories that is usually reflected in pathnames may need additional separators to distinguish between ordinary idents and directory names. However, the resident filesystem allows for FileIdents with the syntax given below:

```
FileIdent = Ident { "." Ident } .
```

The length of a FileIdent is restricted 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 Ceres, 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. OBN for object code files, SMB for symbol files), and there are programs which automatically set the extension, when they generate new files (e.g. the compiler, the editor).

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 disk of your Ceres 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.DOK	*is accepted as file name*	XY.Color.DOK

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 application programs that require the input of file names the module FileNames of the standard library is recommended (see Library).

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 application programs the module *Options* of the standard library is recommended (see Library).

4.4. The Mouse

An important input device, along with the keyboard, is the *mouse*. It allows *positioning* and *command selection*. 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 button	middle button	right 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.

Application programs may use the module CursorMouse that supports cursor tracking and the handling of the mouse buttons (see chapter 10.).

5. The Editor

5.1. Introduction

The Ceres text editor is called *Sara*. It is a full-screen text editor, specially designed to create and change programs. It imposes no restriction on file size, as it uses an incremental method (piece-list) for editing.

5.2. Starting the Editor

You start the editor by typing sara followed by <return>. Then you see a small window on the bottom of the screen, possibly giving you a default file name for editing (the last compiled file is the default). Press <return>, <space> or the left mouse button to accept the name, or enter an alternate name (to create a new text enter the empty name). If you end your file name with ".", the extension "MOD" is appended.

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

5.3. Keyboard and Mouse, Special Keys

The following notation, commands are used throughout this documentation.

ML	left mouse button
MM	middle mouse button
MR	right mouse button
ESC	escape key
EOL	return key
LF	return key (without auto indentation)
SPC	space bar
TAB	tabulator key (same effect as two SPC)
DEL	backspace key (delete character left)
Ctrl-G	delete character right
Ctrl-O	insert character by typing its number

For non-ASCII-keyboard characters use the toggle switches Ctrl-T (german) and Ctrl-S (french).

In german mode:

type:	to obtain:
@	ä
↑	ö
\	ü
`	Ä
~	Ö
|	Ü

In french mode:

type:	to obtain:
`	â
{	ê
~	î
>	ô
|	û

@	à
]	è
\	ù
[	é
}	ë
↑	ï
<	ç
$	(blank with same width as a digit)

5.4. Text Entry and Selection

The text insertion point (caret) is represented by a small triangle which can be positioned by ML (press and release). The entered text is inserted at the caret and does not overwrite a previous text. You can delete the last character by typing DEL. The next character can be deleted by typing Ctrl-G.

A text portion can be selected by moving the mouse over the text with the ML button down (dragging). If at the same time another button is active, the following commands are executed directly: *Copy* (MM), *Delete* (MR), or *Move* (MM and MR). The selection is canceled by pressing ESC. An existing text selection can be extended/shortened by pointing to the last selected character and dragging (see also command *Split* to easily select greater text portions). Selecting the same character again increases the selection level (character->word->line->lineblock->document).

5.5. Scrolling

The text of a window is scrolled by pressing a mouse button in the scroll-bar (vertikal bar) of a window:

Button	*Command*	*Explanation*
ML	scroll-up	cursor line gets top line.
MM	flip	first line -> begin of document, last line -> end of document
MR	scroll-down	cursor line gets bottom line.

5.6. Window Commands

The window commands are activated with ML and MR in the title bar of a window.

Button	*Kind*	*Command*
ML	no cursor movement	Put window on top.
MR	no cursor movement	Put window on bottom.
ML	cursor movement	Move title bar.
MR	cursor movement	Move window.

5.7. Menu Commands

The editor commands are selected with MM. They can be classified into 3 groups: *Dialog*, *Window*, and *Text*.

Group	*Location*	*Commmands*
Dialog	dialog window	open, exit.
Window	title bar	split, redefine, close.
Text	text region	copy, move, delete, find, error, save, replace, adjust, autoind.

The continuation menu of *Text* is reached by pressing MR additionally. The commands are explained in more detail below. Together with the explanation the messages produced by the commands are listed.

In order to distinct them from the rest of the text we use the *italic* font style.

Dialog

Open

Open a document. The name is read from the keyboard (the default name is the name of the last compiled file or a valid text selection). An empty name creates a new document. The document may be located on another medium (remote file).

Then the window dimensions have to be specified. When using the mouse, the start point (pressing the button) and the end point (releasing the button) are taken to compute missing parameters.

name>

SPC, EOL, ML	Terminate the input. A trailing "." is expanded to ".MOD".
DEL	Delete the last character.
ESC, MM, MR	Abort the command.

define window

ML	Window with maximal width.
MM	Window with maximal width and maximal height.
MR	Arbitrary window.
SPC, EOL	Default window.
ESC	Abort the command.

Exit

Leave the editor without updating the document(s).

exit?

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

Window

Split

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

define separation line

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

Redefine

Redefine the dimensions of a window. The new dimensions are entered in the same way as described for *Open*.

Close

Close a window. If the window is the only window to show a document, the editor asks if the document should be written on disk. The name is read from the keyboard (the default name is the name given in the *Open* command or a valid text selection). After closing the last file the *Exit* command is called automatically.

name>

SPC, EOL, ML	Terminate the input. A trailing "." is expanded to ".MOD". The document is stored on disk and the previous version of the document gets the extension ".BAK".
DEL	Delete the last character.
ESC, MM, MR	Release the document without updating.

release?

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

Text

Copy

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

Move

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

Delete

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

Find

Find a text portion starting at the caret position.

find>

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

While searching the text, ESC may be pressed to abort.

Error

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

Save

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

Replace

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

replace by buffer content?>

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

Adjust

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

adjust(-9..9)>

"0", "1", ..., "9"	Shift 0, 1, ..., 9 position(s) to the right.
"-1", ..., "-9"	Shift 1, ..., 9 position(s) to the left.
"<"	Shift 1 position to the left.
">"	Shift 1 position to the right.
"[", ML	Shift 2 positions to the left.
"]", MR	Shift 2 positions to the right.
"{"	Shift 4 positions to the left.
"}"	Shift 4 positions to the right.
ESC, MM	Abort the command.
any other key	Don't shift.

AutoInd

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

5.8. Error recovery

The editor writes the current state periodically on a file. In case of an illegal program termination, simply restart Sara (without any renaming of files!). Then the last saved state is restored.

6. Utility Programs

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

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

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 text files on screen	6.4.
inspectfile	Inspect the contents of a file	6.5.
compare	Compare textfiles	6.6.
xref	Generate a reference list of a text file	6.7.
link	Link separate modules to a program	6.8.
decobj	Disassembler of object files	6.9.
hermes/fhermes	Transfer files via V24/SCC	6.10.
futil	File-Utility	6.11.
asm32	Assembler	6.12.
vfinit/vfopen	Floppy-Medium	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.DOK`
Matching file names	`M2.DOK`
	`Modula2.DOK`
	`Modula2.Version2.DOK`

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

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

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 standard medium 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.86
  Mouse.Head        7    12.Jan.86
  Mouse.Tail        3    30.Sep.86
  Mouse             1    17.Feb.86
```

Before terminating, the program displays a summary

Number of listed files
Number of blocks used by the files
Number of free blocks on medium
Number of free files on medium

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 = 2 Kbyte)
- 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. Program will ask for the name of 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.

Continue
Program continues after execution of the operation and prompts again for a file name.

Terminate
Program terminates after execution of the operation. *Default.*

DAtesort
Information is listed in order of creation date.

TODay
Only file information of the current day is listed.

BEfore
Only file information of files elder than the given date are listed. Program will ask for the date.

AFter
Only file information of files written after the given date are listed. Program will ask for the date.

RAnge
Only files created in a given date range are listed. Program will ask for the first and last date of the range.

Capitals mark the abbreviations of the option values.

6.2. delete, protect, and unprotect

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 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.*

Continue
Program continues after execution of the operation and prompts again for a file name.

Terminate
Program terminates after execution of the operation. *Default.*

Capitals mark the abbreviations of the option values.

6.3. copy and rename

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.

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

Continue
Program continues after execution of the operation and prompts again for a file name.

Terminate
Program terminates after execution of the operation. *Default.*

Capitals mark the abbreviations of the option values.

6.4. list

Utility to list textfiles on your system's display.

The program asks for a filename which may include wildcard symbols. If there are more than one file that matches the specified filename, then the listing of every file has to be confirmed. As option you may specify whether you want to see the file page after page or scrolled up after each line.

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 "?"-key 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.

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.*

Continue
Program continues after execution of the operation and prompts again for a file name.

Header
The program asks for the number of lines which has to be displayed of the specified file(s). (Default value = 15).

6.5. inspectfile

The program *inspectfile* 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 Hexadecimal and Byte.

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 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. *Default.*

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

Ascii
Displayed values are represented as ASCII characters.

Octal
Displayed values are represented as octal numbers.

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

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. compare

Program *compare* detects differences between two text or document files. It first asks for two files to be compared. Then you are asked whether you want to change some parameters. Type *RETURN* to use default values. If you type "*y*", a window with the parameter values is opened. You may now change a value by pointing at it and clicking a mouse button. Finish modifications by pressing a mouse button in the EXIT field.

The program searches now for a difference between the two files. If there is one, it is displayed. Hit *spacebar* to search for the next difference or *ESC* to abort.

6.7. xref

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).

6.8. link

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.OBN`
Second default file name	`DK.LIB.Options.OBN`

The linker first prompts for the object file of the main module (default extension OBN). 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 V2.0 for Medos-2
 object file> delete.OBN
     Delete                          main module
 NameSearch: DK.LIB.NameSearch.OBN   second default file name
    NameSearch
 Options: DK.Options.OBN             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 OBN) 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.OBN/q          query option set
     Delete
NameSearch> NameSearch.new.OBN     own file substituted
     NameSearch    ?  yes
Options> DK.Options.OBN            default file name
     Options       ?  yes
     FileNames     ?  no          module not linked from this file
FileNames> FileNames.own.OBN
     FileNames     ?  yes
```

6.9. decobj

Program *decobj* 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 *OBN*.

```
*decobj
 in> program.OBN
 out>
```

Then the program prompts for the output file. Entering *ESC* instead of a normal file name directs the output onto the screen. The default extension of the output file is *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.

6.10. hermes/fhermes

The program *hermes* transfers files between two computers connected by a V24 (RS 232) cable, fhermes transfers via a RS 422 connection (channel A of Ceres). Both programs use the same transfer protocol.

To transfer a file do the following steps:

1) Connect both computers with a cable. For hermes be sure that the speed on both computers is set to the same speed (recommended is 19200 baud between Ceres, 9600 baud between Ceres and Lilith). The speed is changed using the program *baudrate*.
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 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 *CTRL-C*, if he is not turned off by the master.

Remarks

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.

6.11. futil

This program is a window-oriented tool for file-handling. Currently it can copy, rename, list and delete files. It can handle two media at the same time.

FUTIL divides the screen into three windows : a dialog window and two directory windows. Three menus are available through the middle mouse button : the general menu in the dialog window, the medium menu in the title bars of the directory windows and the disk menu in the directory windows as soon as a medium has been choosen.

The two directory windows display part of the directory of the medium they represent. The directory may be scrolled and sorted. A subset of the directory may be specified using file names with wildcards. Files may be picked by pointing at them with the mouse and pressing the left button. Picked files appear highlighted. Once one or more files are picked, an operation can be performed on them using the disk menu.

The key idea of FUTIL is that the user should not have to type in any file name.

Simple use

Here is an example on how to do a simple backup operation from the Ceres 'DK' medium onto a floppy disk.

1. Install floppy-medium. Start *futil*. You will be asked 'Default setup ?', answer with 'y'. This will define 'DK' as medium for the left directory window . The directory will be read immediately.

2. Open the floppy-medium in the right window (medium menu: remote). Enter the floppy-medium name. The directory will be read immediately, too.

3. The directories are sorted backwards by date such that the last modified files will appear first. If the files you want to backup do not appear in the Ceres directory window, play with the scroll, sort and select commands until you find them. Then pick the desired files by pointing at them with the mouse and pressing the left button. Depressing the right button drops ('unpicks') all files. Note that files remain picked even if they disappear from the display because of a scroll,sort or select command.

4. Once you have picked all the files you want to backup, choose the copy command in the source directory window (i.e. Ceres directory window). *futil* will display all the files you have picked in the dialog window so that you may verify your choice. Next 'Copy all files ?' will be asked. Answer 'Y' to copy all the files, 'N' to abort the command but leave all files picked.

4. If you have typed 'Y', all the files you have picked are copied. For every file, *futil* displays the operation in the dialog window. The copy operation may be aborted by hitting the escape key. Files not copied remain picked so that you may continue the copy operation easily.

To restore the files, just pick them in the floppy directory window and use the copy operation in that window. Other operations available are rename, delete and list. They also operate on all the files you have picked.

6.12. asm32

This program is an assembler for the NS32000 microprocessor family. It can handle two different "kinds" of programs

- ordinary assembler programs which may be executed on every NS 32000 based system
- assembler modules that can be used as if they were written in Modula 2. It is recommended to use it only if it is really necessary. It should be used only by programmers which are very familiar with the Ceres workstation and the Modula-2 run-time organization.

The assembler produces, on user demand, a listing file and a cross-reference of all defined symbols.

Assembly of a program or a module

The assembler is called by typing *asm32*. After displaying the string "in>", the assembler is ready to accept the filename of the program to be compiled.

Default medium is *DK* and default extension is *ASM*.

The assembler produces a relocateable code-file (extension REL) for assembler programs or an object-code file (extension OBN). The program listing and/or the cross-reference is written onto the list-file (extension LST).

Program Options

/n
The assembler does not generate a listing file.

/x
The assembler generates a listing file that includes a cross-reference of all user defined symbols.

6.13. vfopen and vfinit

The two programs described in this section allow the use of the floppy disk of Ceres. The program *vfinit* formats diskettes and initializes the file system on the floppy. The program *vfopen* installs the floppy disk as an ordinary medium in Medos-2. The floppy may then be used with all Medos-2 utility programs. However, it is substantially slower than the winchester and therefore better suited as a backup medium.

The programs allow to work with both single- and double sided 3 1/2" inch floppy disks. The capacities are 360 KBytes resp. 720 KBytes. The file organization on the floppy disk is described in [1]. The following sections describe the two programs in more detail.

The program *vfinit* initializes and formats 3 1/2" floppy disks for the use as medium under Medos-2. It allows the transparent use of single- and double sided floppy disks. Because the file organization on the floppy uses preallocated space for the directory and allocation information, the number of files per floppy disk is fixed. A single sided floppy allows a maximum of 64 files, while 128 files are possible on a double sided floppy disk.

After *vfinit* is started the user is asked for the medium name and the size of the diskette. It is possible just to initialize the file system information on the floppy without physical reformatting. The following example shows the dialog. The characters typed by the user are printed in the *italic* font style.

```
*vfinit
vfinit V3.0 28.11.86
VF medium initialization
mediumName: backup
s(ingle sided), d(ouble sided) ? single sided
format floppy (y/-)? yes
formatting floppy...done
initialize file directory.....done
initialize name directory...done
*
```

When there are problems during the initialization they are reported by vfinit. However, when problems occur during the formatting of a diskette the physical media must be seriously damaged.

Once a floppy disk has been initialized with *vfinit* it can be made accessible as a medium in Medos-2 using the program *vfopen*. This program contains the implementation of the two procedures necessary to announce a medium to the module FileSystem (see section 9.1.5. of this document). *vfopen* introduces a new level and calls the command interpreter on top of it. Before calling *vfopen* you must be sure that the floppy disk is installed in the drive. After using the floppy disk the program *vfopen* must be left properly by typing ESC in order to leave the floppy disk in a consistent state. During one run of vfopen the diskette must not be changed for the same reason. However, when the floppy disk has not been closed properly the main data structures, i.e. the allocation tables are rebuild when it is used next. The following example shows how to use *vfopen*.

```
*vfopen
vfopen 28.11.86
...#backup opened, press ESC to leave
*
..... now the floppy is accessible as #backup
*ESC
Close #backup (y/-)?y

#backup closed
*
```

We recommend to use the floppy utilities together with the program *futil*, which is described in section 6.11.

[1] F.Peschel, M. Wille: Porting Medos-2 onto the Ceres workstation,
Institut für Informatik, to be published

7. The Compiler

This chapter describes the use of the Modula-2 compiler. For the language definition refer to the Modula-2 manual [1]. Ceres 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).

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*.

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

reference file
Compiler output file with debugger information, generated during compilation of a program module. Assigned extension is *RFN*.

object file
Compiler output file with the generated code in loader format. Assigned extension is *OBN*.

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;
  ...
END Prog3.

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

7.2. Compilation of a Program Module

The compiler is called by typing *compile*. After displaying the string "in>", the compiler is ready to accept the filename of the compilation unit to be compiled.

Default device is *DK* and default extension is *MOD*.

After compilation, the compiler again requests a file name for a further compilation. Terminate by typing ESC.

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.

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.

The filenames of imported modules are shown by the compiler.

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

7.5. Compiler Output Files

Several files are generated by the compiler. Their file names are taken as the compilation unit's module name with the appropriate file name extension:

SMB symbol file
RFN reference file
OBN object file

The reference file is used by the symbolic debugger. The compiler produces two error listing files (*err.LST* and *err.DAT*). The latter is interpreted by other programs, e.g. the program editor *sara*.

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:

/r
The compiler does generate instructions to check subrange values.

/v
The compiler does generate instructions to check against overflow.

/x
The compiler does not generate instructions to check index bounds.

7.7. Module Key

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

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

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

WARNING

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

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

7.8. Program Execution

Programs are normally executed on 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 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 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.

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 module *Program* (see chapter 9.4.).

7.9. Value Ranges of the Standard Types

The value ranges of the Modula-2 standard types on Ceres are defined as follows

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.

LONGINT
: The value range of type INTEGER is [-2147483648..2147483647]. The smallest LONGINT can be defined indirectly (see INTEGER).

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

REAL
Values of type REAL are represented in 4 bytes. 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..31]. Consider that sets are represented from the low order bits to the high order bits, i.e. {0} corresponds to the ordinal value 1.

7.10. Restrictions

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

Forward References
No forward references are permitted, except in definitions of pointer types and in forward procedure declarations.

Sets
Maximal ordinal value for set elements is 31. If, in {m..n}, m is a constant, then n must also be a constant.

FOR statement
The values of both expressions of the for statement must not be greater than 32767 (77777B).

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

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

Standard functions, procedures, and types
The functions VAL, the procedures NEW, DISPOSE, TRANSFER, NEWPROCESS, and the type PROCESS are not predeclared (although defined as standard objects in earlier reports on the language).
The function SIZE is a standard procedure (globally defined), and it is identical to the function TSIZE defined in module SYSTEM. Its argument is a type or a variable; the result is the size of the type (of the variable) in number of bytes required for the variable or for a variable of that type.

Subranges
The bounds of a subrange must be less than 2^{15} in absolute value.

Opaque types
If a type T is declared in a definition module to be opaque, it cannot (in the corresponding implementation module) be declared as equal to another, named type.

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

7.11. Compiler Error Messages

Syntax errors

10 identifier expected
11 , comma expected
12 ; semicolon expected
13 : colon expected
14 . period expected
15) right parenthesis expected
16] right bracket expected
17 } right brace expected
18 = equal sign expected
19 := assignment expected
20 END expected
21 .. ellipsis expected
22 (left parenthesis expected
23 OF expected
24 TO expected
25 DO expected
26 UNTIL expected
27 THEN expected
28 MODULE expected
29 illegal digit, or number too large
30 IMPORT expected
31 factor starts with illegal symbol
32 identifier, (, or [expected
33 identifier, ARRAY, RECORD, SET, POINTER, PROCEDURE, (, or [expected
34 Type followed by illegal symbol
35 statement starts with illegal symbol
36 declaration followed by illegal symbol
37 statement part is not allowed in definition module
38 export list not allowed in program module
39 EXIT not inside a LOOP statement
40 illegal character in number
41 number too large
42 comment without closing *)
43
44 expression must contain constant operands only
45 control character within string

Undefined

50 identifier not declared or not visible

Class and type errors

51 object should be a constant
52 object should be a type
53 object should be a variable
54 object should be a procedure
55 object should be a module
56 type should be a subrange
57 type should be a record
58 type should be an array
59 type should be a set
60 illegal base type of set
61 incompatible type of label or of subrange bound
62 multiply defined case (label)
63 low bound > high bound

64 more actual than formal parameters
65 fewer actual than formal parameters

66 - 73 mismatch between parameter list D in definition and I in implementation module
66 more parameters in I than in D
67 parameters with equal types in I have different types in D
68 mismatch between VAR specifications
69 mismatch between type specifications
70 more parameters in D than in I
71 mismatch between result type specifications
72 function in D, pure procedure in I
73 procedure in D has parameters, but not in I

74 code procedure cannot be declared in definition module
75 illegal type of control variable in FOR statement
76 procedure call of a function
77 identifiers in heading and at end do not match
78 redefinition of a type that is declared in definition part
79 imported module not found
80 unsatisfied export list entry
81 illegal type of procedure result
82 illegal base type of subrange
83 illegal type of case expression
84 writing of symbol file failed
85 keys of imported symbol files do not match
86 error in format of symbol file
88 symbol file not successfully opened
89 procedure declared in definition module, but not in implementation

Implementation restrictions of compiler

90 in {a..b}, if a is a constant, b must also be a constant
91 code procedure can have at most 8 bytes of code
92 too many cases
93 too many exit statements
94 index type of array must be a subrange
95 subrange bound must be less than 2↑15
96 too many global modules
97 too many procedure in definition module
98 too many structure elements in definition module
99 too many variables, or record too large

Multiple definition

100 multiple definition within the same scope

Class and type incompatibilities

101 illegal use of type
102 illegal use of procedure
103 illegal use of constant
104 illegal use of type
105 illegal use of procedure
106 illegal use of expression
107 illegal use of module
108 constant index out of range
109 indexed variable is not an array, or the index has the wrong type
110 record selector is not a field identifier
111 dereferenced variable is not a pointer
112 operand type incompatible with sign inversion
113 operand type incompatible with NOT
114 x IN y: type(x) # basetype(y)

115 type of x cannot be the basetype of a set, or y is not a set
116 {a..b}: type of either a or b is not equal to the base type of the set
117 incompatible operand types
118 operand type incompatible with *
119 operand type incompatible with /
120 operand type incompatible with DIV
121 operand type incompatible with MOD
122 operand type incompatible with AND
123 operand type incompatible with +
124 operand type incompatible with -
125 operand type incompatible with OR
126 operand type incompatible with relation

127-131: assignment of a procedure P to a variable of type T
127 procedure must have level 0
128 result type of P does not match that of T
129 mismatch of a parameter of P with the formal type list of T
130 procedure has fewer parameters than the formal type list
131 procedure has more parameters than the formal type list

132 assignment of a negative integer to a cardinal variable
133 incompatible assignment
134 assignment to non-variable
135 type of expression in IF, WHILE, UNTIL clause must be BOOLEAN
136 call of an object which is not a procedure
137 type of VAR parameter is not identical to that of actual parameter
138 value assigned to subrange variable is out of bounds
139 type of RETURN expression differs from procedure type
140 illegal type of CASE expression
141 step in FOR clause cannot be 0
142 illegal type of control variable
143
144 incorrect type of parameter of standard procedure
145 this parameter should be a type identifier
146 string is too long
147 incorrect priority specification
148
149

Name collision

150 exported identifier collides with declared identifier

Implementation restrictions of system

200 character assigned to array (not yet implemented)
201 integer too small for sign inversion
202 set element outside word range
203 overflow in multipliaction
204 overfolow in division
205 division by zero, or modulus with negative value
206 overflow in addition
207 overflow in subtraction
208 cardinal value assigned to integer variable too large
209 set size too large
210 array size too large
211 address too large (compiler error?)
212 character array component cannot correspond to VAR parameter
213 illegal store operation (compiler error?)
214 set range bounds must be constants
215 expression too complex (stack overflow)
216 double precision multiply and divide are not implemented

222 output file not opened (directory full?)
223 output incomplete (disk full?)
224 too many external references
225 too many strings
226 program too long

8. The Debugger

8.1. Introduction

Program execution errors cause the operating system to make a complete dump of the main memory to the disk, thus preserving the computer's state at error time. The debugger is an interpreter of this dump file. It allows you to search for execution errors on the same abstraction level that you originally created your program (see Fig. 1). The debugger is of great help in finding the reasons for your program's crash.

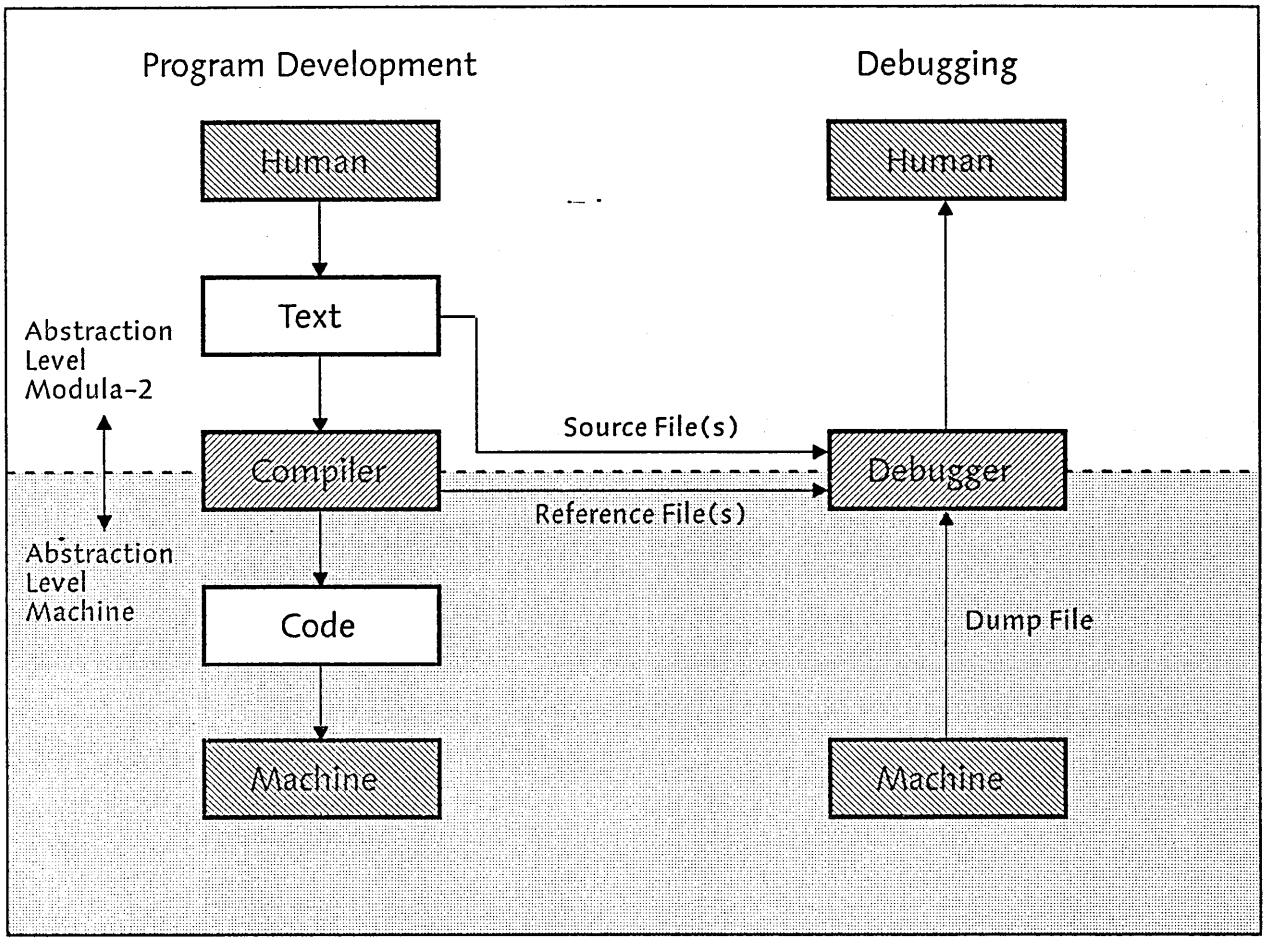


Fig. 1: Debugger Environment

The debugger gets its information from three sources:

- Source File(s)	source text of your program
- Reference File(s)	compiler-generated file with names, types, etc. of your program
- Dump File	operating system generated dump of the computer's main memory

If your program consists of several modules, the debugger needs the source and reference file of each module.

8.2. Starting the Debugger

After a runtime error, you start the debugger by typing *inspect*. The debugger now reads the dump file, the applicable source file and all the necessary reference files and presents you with the following screen image:

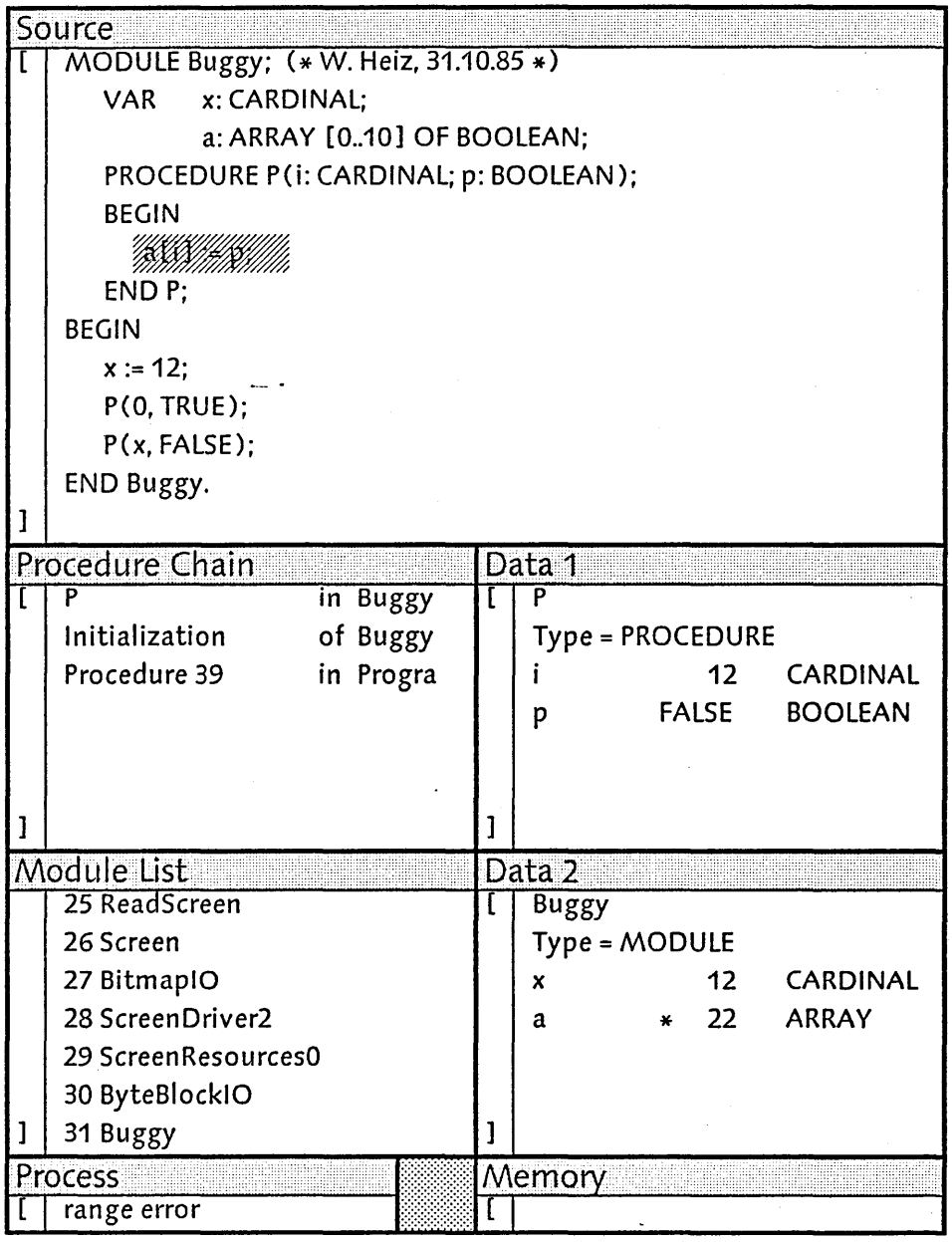


Fig. 2: Debugger Screen Image

The seven windows are:

- **Source Window**: Contains the source listing with the erroneous statement marked.
- **Procedure Chain Window**: Shows the calling sequence of the active procedures.
- **Data 1 Window**: Contains all the variables of the last called procedure.
- **Module List Window**: Lists all loaded modules.
- **Data 2 Window**: Contains all the variables of the active module.
- **Process Window**: Displays the reason for the program crash.
- **Memory Window**: Shows the memory in an uninterpreted form.

Subsequently we shall take a closer look at each window.

8.3. Global Commands

Each Window has a scroll bar at the left side. Clicking inside the scroll bar causes the contents to scroll as in the editor. Three marks in the scroll bar ("[", "]" and "|") give additional information on the window state. The marks "[" and "]" tell that the beginning or end, respectively, of the information is visible. The mark "|" gives feedback of the relative position of the window in the information being displayed. The three mouse buttons (ML, MM and MR) are used as follows:

ML: Display data of selected object.
MR: Display source or memory of selected object.
MM: Display menu.

8.4. Local Commands

The Source Window

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

ML: -
MR: -
MM: Menu: Ask On — Ask user for alternate filenames if source file cannot be found.
Ask Off — Don't ask user for alternate filenames.

The Data Windows

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

ML: Show further data of selected object if there is such.
Objects with further data are marked with "*".
The top line shows a path through which the present structure was reached.
The path can be retraced in reverse order by selecting a previous component.
MR: Show memory corresponding to a selected object. This includes objects in the structure path.
MM: Menu: Expand — Expand window.
Shrink — Shrink window.

The Procedure Chain Window

The procedure chain window shows the calling sequence of the active procedures in the current selected process.

ML: Show local data of selected procedure.
MR: Show source listing of selected procedure.
MM: Menu: Data 1 — Use data window 1 to display local data of procedures.
Data 2 — Use data window 2 to display local data of procedures.

The Module List Window

The module list window shows the list of loaded modules.

ML: Show global data of selected module.
MR: Show source listing of selected module.
MM: Menu: Data 1 — Use data window 1 to display global data of modules.
Data 2 — Use data window 2 to display global data of modules.
DEF — Show the definition listing of future selected modules.
MOD — Show the implementation listing of future selected modules.

The Process Window

The process window shows the process status.

ML: -
MR: -
MM: Menu:

Process	After selecting this option, a process can be selected from the data window by pointing to a variable of type ADDRESS or from the memory window by pointing to any memory cell.
Main Pr	Return to the main process.
Exit	Leave the debugger.

The Memory Window

The memory window shows the memory in an uninterpreted form.

ML: Interpret selected memory cell as an address for further memory dump.
MR: -
MM: Menu:

Expand	Expand window.
Shrink	Shrink window.
Char	Display mode is character.
Byte	Display mode is byte (in hex notation).
Word	Display mode is word (in hex notation).
Double	Display mode is double word (in hex notation).

8.5. Debugger Command Summary

Mouse Button Pressed	Mouse Cursor is in:						
	scroll bar of any window	source window	data window	procedure window	module window	process window	memory window
left button	scroll up	-	show further data of selected object	show local data of selected procedure	show global data of selected module	-	show memory dump of selected address
right button	scroll down	-	show memory of selected object	show source of selected procedure	show source of selected module	-	-
middle button	position window contents	Menu: Ask On Ask Off	Menu: Expand Shrink	Menu: Data 1 Data 2	Menu: Data 1 Data 2 DEF MOD	Menu: Process Main Pr Exit	Menu: Expand Shrink Char Byte Word Double

Fig. 3: Debugger Command Summary

9. The Medos-2 Interface

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

FileSystem	Module for the use of files	9.1.
Processes	Module for the handling of processes	9.2.
Program	Facilities for the execution of programs	9.3.
Programs	Module for handling programs and resources	9.4.
Heap	Module for storage allocation and deallocation	9.5.
SEK	Module for activation of the command interpreter	9.6.
TerminalBase	Module for substitution of terminal input and output	9.7.
Terminal	Module for terminal input and output	9.8.
Users	Module for handling of user identifications	9.9.
Clock	Module for setting and requesting the current time	9.10.

9.1. Module FileSystem

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 Ceres 16 Mbyte of the winchester disk drive act as 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;

  FROM SYSTEM IMPORT ADDRESS, WORD;

  TYPE
    Flag          = (er, ef, rd, wr, ag, bm);
    FlagSet       = SET OF Flag;

    Response      = (done, notdone, lockerror, permissionerror,
                        notsupported, callerror,
                        unknownmedium, unknownfile, filenameerror,
                        toomanyfiles, mediumfull,
                        deviceoff, parityerror, harderror);

    Command       = (create, open, opendir, close, rename,
                        setread, setwrite, setmodify, setopen, doio,
                        setpos, getpos, length,
                        setpermission, getpermission,
                        setpermanent, getpermanent);

    Lock          = (nolock, sharedlock, exclusivelock);

    Permission    = (noperm, ownerperm, groupperm, allperm);

    MediumType    = ARRAY [0..1] OF CHAR;

    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:  new: BOOLEAN; lock: Lock
                      | opendir:  selections: BITSET;
                      | setpos, getpos, length:  highpos, lowpos: CARDINAL
                      | setpermission, getpermission:
                          readpermission: Permission;
                          modifypermission: Permission
                      | setpermanent, getpermanent: on: BOOLEAN
                      END;
                      mt: MediumType; mediumno: CARDINAL;
                      fileno: CARDINAL; versionno: CARDINAL;
                      openedfile: ADDRESS;
                    END;

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

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

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

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


  PROCEDURE 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 SetPosL(VAR f: File; pos: LONGINT);
  PROCEDURE GetPosL(VAR f: File; VAR pos: LONGINT);
  PROCEDURE LengthL(VAR f: File; VAR pos: LONGINT);

  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 DeleteMedium(mt: MediumType; mediumno: CARDINAL;
                         VAR done: BOOLEAN);

  PROCEDURE AssignName(mt: MediumType; mediumno: CARDINAL;
                       mediumname: ARRAY OF CHAR; VAR done: BOOLEAN);
  PROCEDURE DeassignName(mediumname: ARRAY OF CHAR; VAR done: BOOLEAN);

  PROCEDURE ReadMedium(index: CARDINAL;
                       VAR mt: MediumType; VAR mediumno: CARDINAL;
                       VAR mediumname: ARRAY OF CHAR; VAR original: BOOLEAN;
                       VAR done: BOOLEAN);
  PROCEDURE LookupMedium(VAR mt: MediumType; VAR mediumno: CARDINAL;
                         mediumname: ARRAY OF CHAR; 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.OBN

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

Create(f, filename)

Procedure *Create* creates a new file. The created file is temporary (and nameless) if the substituted filename parameter is a medium name. The created file is permanent if the substituted filename parameter contains a local name. After the call

f.res = done	if file f is created,
f.res = notdone	if a file with the given name already exists,
f.res = ...	if some error occured.

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

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.

Delete(f)

Procedure *Delete* terminates any actual input or output operation on file f and disconnects the variable f from the actual file. The actual file is deleted hereafter. Procedure Delete is equivalent to:

```
Rename(f, "");      (* See next description *)
Close(f);
```

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 9.8.) and to handle defaults (see chapter 11.6.).

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* unreads the last read byte (e.g. character) on file f. As a consequence, the last byte read just before the call of *Again* will be read again as the first byte the next (sequentially) read-operation.

Related Modules

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. For Ceres there also exist the procedures *GetPosL*, *LengthL* and *SetPosL* which have only one position argument of type LONGINT.

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*. The commands *create*, *open*, *opendir*, *close*, and *rename* (constants of type *Command*) are executed by procedure *DirectoryCommand*, all other by procedure *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; f.lock := nolock;
  DirectoryCommand(f, filename)
END Lookup;
```

Commands may be executed either by procedure *FileCommand* or by procedure *DirectoryCommand*. Unless the command is *create*, *open*, *opendir*, *close*, or *rename*, a call to *DirectoryCommand* will be converted to a call to *FileCommand* by the file system. Similarely, the file system converts calls to procedure *FileCommand* with command *create*, *open*, *opendir*, *close*, or *rename* to *DirectoryCommand* calls with empty file name parameters..

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)
opendir	open a directory file by wildcard filename	(9.1.4.6)
close	close a file	(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)
setpermission	change the *protection* of the file	(9.1.4.4)
getpermission	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)

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 9.2.4.5. *) 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 current state of the file, whenever it is accessed. The "flag" *ag* is set by routine *Again* and cleared by routines, which changes the current position of a file (e.g. read routines) or which removes the *rd*-flag field *flags* of the file (command *setopen* and *setwrite*)..

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 stored in the fields *mt* and *mediumno* of the file variable. The LFI is stored in the fields *fileno* and *versionno* of the file variable.

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;
f.new := ...; f.lock := ...;
DirectoryCommand(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 looked up and replaced by 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.).

9.1.4.3. Creation, Opening, and Closing of Files

The most convenient way to create, open, and close files have already been mentioned in chapter 9.1.3.1. This is done by the procedures *Create*, *Lookup*, and *Close*. For special problems, the command *create* or *open* may be invoked directly. Concurrent accesses to files and nameless files may be handled that way.

A file *lock* specify, how concurrent accesses to a file has to be handled. This is interesting, when there are several access-paths to one file at the same time. E.g. a file on a file-server may be accessed from several client machines at the same time. In Medos-2, the wanted handeling of concurrent accesses is specified when a file is created or opened. This is done by assigning the field *lock* in the file-variable a value of type *Lock*.

```
Lock      = (nolock, sharedlock, exclusivelock)

nolock:        concurrent accesses are unrestrictedly allowed
sharedlock:    concurrent non-modifying accesses are unrestrictedly allowed
exclusivelock:no concurrent accesses are allowed
```

The specific lock holds for the time the file is opened. This implies that a file is opened with *sharedlock* cannot be changed during the time the file is opened. It can, however, be opened several times with sharedlock at the same time, if the underlaying system permits it (i.e. if no other conditions forbid it). If a file is created or opened with *exclusivelock* it is guaranteed that the file is not opened a second time until the first opened file is closed again.

Create a nameless (and temporary) file on medium "DK"

```
f.com := create;
f.new := TRUE; f.lock := nolock;
 DirectoryCommand(f, "DK")
```

The field *new* must always be set TRUE for command *create*. An existing database "DK.EvR1.WS84.students" is opened for exclusive access the following way:

```
f.com := open;
f.new := FALSE; f.lock := exclusivelock;
DirectoryCommand(f, "DK.EvR1.WS84.students");
IF f.res = done THEN
  .....
END
```

Closing of files is done by command *close*:

```
f.com := close;
DirectoryCommand(f, "")
```

9.1.4.4. 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 *)
```

9.1.4.5. Protection of Files

It is possible to control accesses to a file by changing its access-permissions. A file has a read- and a modify-permission. A permission can be given to nobody, to the owner of the file only, to group members, and to everybody. The read-permission allows a user to read the information stored in the file and to get information (e.g. statistics) about the file. A user need the modify-permission on a file in order to be able to change it. The only exception to these rules is that the owner of a file may always inspect and change the permissions of the file. A certain permission is indicated by a constant of type Permission:

Permission = (noperm, ownerperm, groupperm, allperm).

In Medos-2, the modification of a file implies reading of the file. A file's read-permission must thereforalways be less restrictive or the sames as its modify-permission. This is given by the following rule:

read-permission of file >= modify-permission of file

Set Permissions

```
f.readpermission := ...perm; f.modifypermission := ...perm;
f.com := setpermission; FileCommand(f)
```

Protect File against changes

```
f.readpermission := allperm; f.modifypermission := noperm;
f.com := setpermission; FileCommand(f)
```

Unprotect File

```
f.readpermission := allperm; f.modifypermission := allperm;
f.com := setpermission; FileCommand(f)
```

Get File Permissions

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

(* now f.readpermission and f.modifypermission describe permissions *)
```

9.1.4.6. Reading, Writing, and Modifying Files

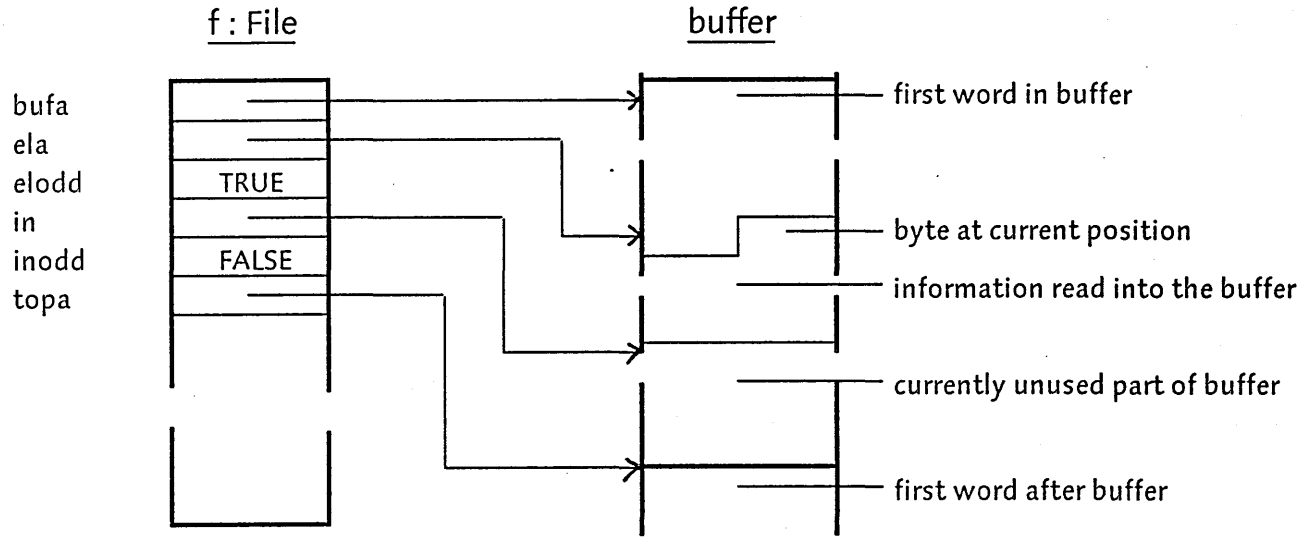
A file can be in one of four possible *I/O states* (or simply, *states*), namely in state *opened*, *reading*, *writing* or *modifying*. Just after a file has been connected (e.g. by a call to procedure *Create*), a file is in state opened, and its current position is zero. The state of a file can only be changed by a direct or indirect call to one of the routines *SetOpen*, *SetRead*, *SetWrite*, and *SetModify* or by executing one of the commands *setopen*, *setread*, *setwrite*, and *setmodify*. The actual state of a file may be inspected in field *flags* of the file:

opened	flags * FlagSet{rd, wr} = FlagSet{}
reading	flags * FlagSet{rd, wr} = FlagSet{rd}
writing	flags * FlagSet{rd, wr} = FlagSet{wr}
modifying	flags * FlagSet{rd, wr} = FlagSet{rd, wr}

The buffers needed for the transfer of data to and from files are supplied and managed by the file system. The changes of a file's I/O state and normally the command *doio* (or procedure *Doio* resp.) control the system's buffering. The commands *setread*, *setwrite*, *setmodify*, *setopen*, and *doio* (and the corresponding routines) do, however, *not* change the *current position* of a file as a side effect.

In state *opened*, no buffer is assigned to a file (seen from a user's point of view). Any internal buffer with new or changed information has been written back onto the medium on which the file is physically stored. The addresses describing the buffer in the file variable (*bufa*, *ela*, *ina*, and *topa*) are all equal to NIL. Any written or changed information within a file can therefore be forced out (flushed) to the corresponding medium by a call to *SetOpen*.

In the other three states (*reading*, *writing* and *modifying*), a buffer is assigned to the file. The following figure shows how *bufa*, *ela*, *elodd*, *ina*, *inodd*, and *topa* describe the buffer supplied by the system:



bufa	address of the first byte of the buffer
topa	address of the first byte behind the buffer
ina	address of the first byte behind the data read from the file
inodd	TRUE, if the *last read byte* is a high order byte
ela	address of the word containing the *byte at the current position*
elodd	TRUE, if the *byte at the current position* is a low order byte

The following two assertions should always hold for *bufa*, *ela*, *ina*, and *topa*:

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

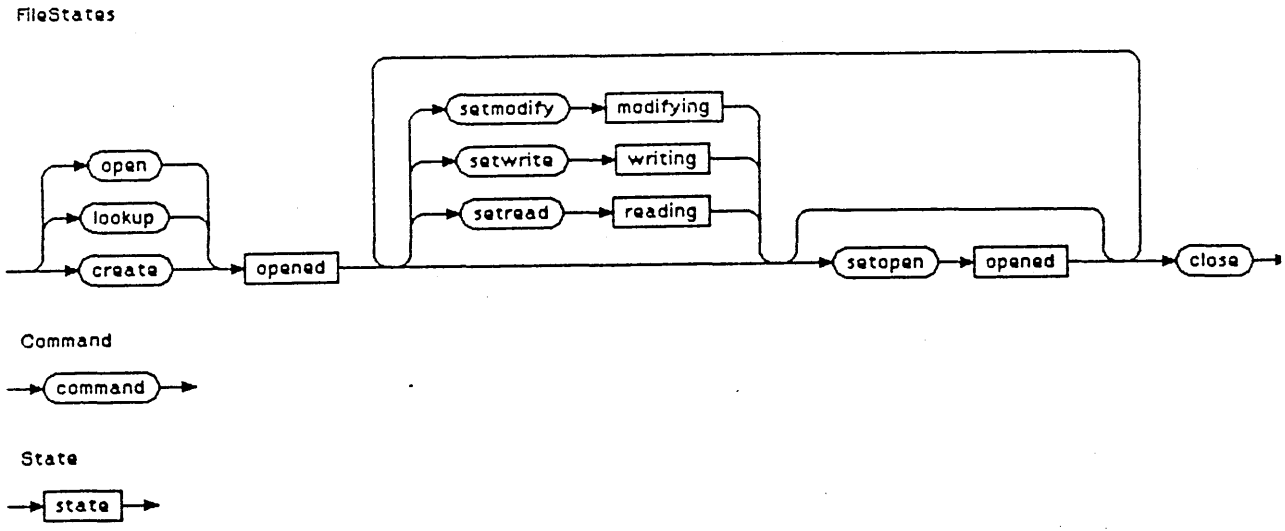
The fields *bufa*, *ina*, *inodd*, and *topa* are *read-only*, as they contain information which must never be changed by any user of a file.

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.

9.1.4.7. 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);
  VAR chPtr: CharPointer;
BEGIN
  WITH f DO
    LOOP
      IF flags * FlagSet{wr, bm, er} <> FlagSet{wr, bm} THEN
        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;
        IF er IN flags THEN RETURN END;
        INCL(flags, bm)
      ELSIF ela >= topa THEN Doio(f)
      ELS
        chPtr := ela; chPtr↑ := ch;
        INC(ela); elodd := NOT elodd;
        RETURN
      END
    END
  END
END WriteChar;
```

9.1.4.8. Directory Information

Information about the files stored on a certain medium may be obtained by executing command *opendir*. This command is activated by a call to procedure *DirectoryCommand*. The command causes the system to create a new temporary file containing the desired directory information. The "filename" supplied to *DirectoryCommand* may contain wildcard symbols, and such a wildcard filename selects hereby the set of files for which information is desired. The field *selections* in the file variable will further specify the expected extent of the desired information. The generated file may be read and interpreted afterwards, e.g. by a directory or copy program.

The following wildcard characters may be part of a wildcard filename:

*	{ letter | digit | "." }	any (including the empty) sequence of legal file name characters
$	{ letter | digit }	any (including the empty) sequence of identifier characters
%	(letter | digit | ".")	exactly one legal file name character

The wildcatd characters may not be used in the mediumname of a filename, i.e. on execution of opendir only information can be delivered about files on one single medium.

The following elements in selections are assigned:

0	equal	Capital and lower case letters are treated as being equal
1	dirstat	Provide statistics about the directory
2	dironly	Only directory information requested
3	filestat	Provide statistics about the selected files
6	groupmem	Select desired information among files belonging members in the same group
7	allmem	Select desired information among all files on the selected medium

Currently no futher elements should be present in selections. They may be assigned later on if necessary.

The directory information is encoded on the generated file in the following format. The information consists of a sequence of blocks, each containing information according to the type of the block. The type and the size of each block is specified in the first word of the block.

```
File =          { Block } .
Block =         TypeAndSize { Words } .
TypeAndSize =   Number .                    \ Type + number of words after header * 256\
```

The format of a directory file is specified as follows:

```
DirectoryFile =  DirInfo | FileInfo .
DirInfo =        { DirName } DirIdent { StatisticInfo } { DirInfo | FileInfo } DIREND .
DirName =        DIRNAME String .
DirIdent =       DIRID CharWord Number .    \ medium type, medium number \
FileInfo =       { Filename } FileIdent { StatisticInfo } .
FileName =       FILENAME String .
FileIdent =      FILEID Number Number .     \ file number, version number \
StatisticInfo =  FILES FlexNumber | FREEFILES FlexNumber |
                 SIZE ByteSize | FREESIZE ByteSize |
                 CREATEDATE Date |
                 MODIFYDATE Date | MODIFYCOUNT Number |
                 ACCESSDATE Date  | ACCESSCOUNT Number |
                 OWNER UserIdent |
                 PERMISSION Permission Permission |          \ read-perm, modify-perm \
                 MESSAGE String .
String =         { CharWord } .
CharWord =       Number .                   \ ORD(leftChar) + ORD(rightChar) * 256\
ByteSize =       FlexNumber .               \ Size in bytes \
FlexNumber =     Number [ Number ] .        \ [ long ] cardinal value \
Date =           Number Number Number .     \ day, minute, millisecond as in module Clock \
```

```
UserIdent =     Number Number.      \ group member \
Permission =    Number.             \ ORD(Permission-value) \

DIRNAME =       0 + n * 256.
DIRID =         1 + 2 * 256.
DIREND =        2 + 256.

FILNAME =       16 + n * 256.
FILID =         17 + 2 * 256.

FILES =         32 + n * 256.
FREEFILES =     33 + n * 256.
SIZE =          34 + n * 256.
FREEFILES =     35 + n * 256.

CREATEDATE =    48 + 3 * 256.
MODIFYDATE =    49 + 3 * 256.
MODIFYCOUNT =   50 + 1 * 256.
ACCESSDATE =    51 + 3 * 256.
ACCESSCOUNT =   52 + 1 * 256.

OWNER =         64 + 3 * 256.
PERMISSION =    65 + 1 * 256.

MESSAGE =       80 + n * 256.
```

9.1.5. Implementation of Files

A program may implement files on a certain medium and make these files accessible through the file system (i.e. through module *FileSystem*). This is done with 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.

After a call to procedure *DeleteMedium*, 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.

The external name of a medium is computed from the medium's type and number: One or two letters are taken from argument *mt* to procedure *CreateMedium* and the argument *mediumno* is apended to the letter(s) as decimal digits if mediumno is less than 177777B. This mediumname is the original mediumname of the created medium.

It is possible to assign additional names to a known medium. This is done by procedure *AssignName*. The assigned mediumname must a letter followed by zero to seven letters or digits, and it must not be known to the filesystem. An assigned mediumname may be removed again by procedure *DeassignName*.

Two additional procedures are provided for the management of mediumnames. Procedure *ReadMedium* returns known mediumnames and information about them. Procedure *LookupMedium* converts a known mediumname to the corresponding internal medium identification IMI.

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 *create, open, opendir, close,* 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 normally only the *local file name* part of the original file name. The corresponding IMI is stored in the file variable.

If the medium was "created" with MediumType = "", the string supplied as parameter to procedure *directoryproc* contains the whole filename (i.e. including its mediumname), and the field *mt* in the affected file is equal "". The medium "" is used in (direct or indirect) calls of procedure *DirectoryCommand* where an unknown mediumname is specified in its filename-parameter. Only one medium may be "created" with name "".

The field *openedfile* in the file variable may be used freely by the module implementing files (e.g. as an index into a table of connected files).

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 *Programs*, chapter 9.4.).

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.

DeleteMedium(mediumtype, mediumnumber, done)

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

AssignName(mt, mediumno, mediumname, done)

> Procedure *AssignName* assigns an additional mediumname to the given medium. It must be an identifier of at most eight characters. *done* is TRUE if the assignment is made. If the mediumname exists already, if there is no space available in the internal tables, or if any of the arguments are not valid, no new mediumname is assigned and *done* is set FALSE.

DeassignName(mediumname, done)

Procedure *DeassignName* deassigns a mediumname previously assigned be procedure AssignName. The original name of a medium, i.e. the one generated by procedure CreateMedium, cannot be deassigned. *done* is TRUE if the assignment is made. Automatic deassignment of a mediumname occurs when the corresponding medium deleted.

ReadMedium(index, mt, mediumno, mediumname, original, done)

Procedure *ReadMedium* returns for the given index a mediumname and the corresponding medium identification. *original* indicates whether or not the returned name is the implicitly generated name of the medium. *done* is TRUE if a mediumname was returned. The index is understood as an index into an internal table of module FileSystem in which the mediumnames are stored. The lowest index is 0, the highest depends upon the number of known mediumnames. By calling ReadMedium several times with incremented indices, all known mediumnames will be returned.

LookupMedium(mt, mediumno, mediumname, done)

Procedure *LookupMedium* converts a known mediumname into the corresponding medium identification IMI. *done* is TRUE if the mediumname was known.

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. File Representation

9.1.6.1. Main Characteristics and Restrictions

Files are implemented by the module *DiskSystem*. The module is accompained by the disk driver module, i.e. the module *WinDisk* for the winchester disk drive. The common characteristics of the current implementation of files are listed below:

max. file length	192 kbyte
local file name length	1 - 24 characters
max. number of opened files	14 (16)
medium name	"WD"
internal medium identification	("WD", 65535)
maximum number of files	2048
disk capacity	16 MByte

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.

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 disk. The remaining 14 files may be used freely by ordinary programs.

9.1.6.2. System Files

The space on a disk is allocated to actual files in *pages* of 2 kbyte each (or 4 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 winchester disk (file directory). The local file names of all files on a disk are stored in another file on the disk (name directory). When a disk is initialized, seven (system-)files are allocated on the disk. These preallocated files can not be truncated or removed. Except for the two directory files and the file containing the disk's bad sectors, all files can be read and written (modified). The preallocated files are:

`FS.FileDirectory`	File with file directory
`FS.NameDirectory`	File with name directory
`FS.BadPages`	File with unusable sectors
`PC.BootFile0`	Normal boot file (size = 64k)
`PC.BootFile1`	Alternate boot file (size = 128k)
`PC.DumpFile0`	File onto which main memory is dumped (size = 64k)
`PC.DumpFile1`	File onto which main memory is dumped (size = 64k)

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	Page in an affected file
page	Disk address of page = page DIV 7 * 4
file number	Number of the affected file
local file name	Local file name of affected file
response	Text describing the *response*
statusbits	Status from disk interface
disk address	"Logical" address of sector on disk

Note, that all inserted values are displayed as *hexadecimal* numbers.

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

- `DiskSystem.ShortSetToBITSET:` bad bitmap entry found
  Module DiskSystem has found an incorrect entry in the allocation bitmap.

- `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.

The following error messages are produced by the disk driver modules.

- `WinDisk.DiskRead:` *response*
- `diskadr =` *disk address*, `statusbits =` *statusbits*

  The driver detected an error, which did not disappear after three retries.

- `WinDisk.DiskWrite:` *response*
- `diskadr =` *disk address*, `statusbits =` *statusbits*

  The disk driver detected an error, which did not disappear after three retries.

9.2. Processes

9.2.1. Introduction

In Medos-2, a Modula-2 program is executed sequentially by one process only, by the so-called *main process*. Module Processes makes it possible to execute programs with (pseudo-) concurrency, i.e. to execute programs by several processes. Module Processes makes it also possible to handle interrupts from devices connected to the Ceres computer. The definition module is given in 9.2.2. The process concept of Medos-2 is explained briefly in 9.2.3. Explanations of the provided routines are given in 9.2.4. and some implementation notes are given in 9.2.5.

9.2.2. Definition Module Processes

```
DEFINITION MODULE Processes;

  PROCEDURE CreateProcess(p: PROC; size: LONGINT; VAR done: BOOLEAN);
  PROCEDURE Pass;
  PROCEDURE Delay(ms: CARDINAL);

  TYPE Signal;
  PROCEDURE InitSignal(VAR s: Signal);
  PROCEDURE Send(VAR s: Signal);
  PROCEDURE SendAll(VAR s: Signal);
  PROCEDURE Wait(VAR s: Signal);
  PROCEDURE TimedWait(VAR s: Signal; ms: CARDINAL);

  TYPE Device = [0..15];
  PROCEDURE CreateDriver(p: PROC; size: LONGINT; dev: Device;
                         VAR done: BOOLEAN);
  PROCEDURE WaitInterrupt;
  PROCEDURE TimedWaitInterrupt(ms: CARDINAL);

END Processes.
```

9.2.3. Process Concept of Medos-2

Concurrency

A program may be executed by one or several (pseudo-concurrent) sequential processes. A program may start a process by calling either procedure *CreateProcess* or procedure *CreateDriver*. These procedures creates and starts a process, which executes the parameterless procedure given as argument to CreateProcess or CreateDriver. A process terminates and removes itself after it has executed the parameterless procedure. The lifetime of a process created by *CreateProcess* or *CreateDriver* is restricted to the lifetime of the program, which started the process.

The *owner* of a new process is the program, for which the creator of the process executes. The terms *owner*, *program*, and *activated program* are explained in the description of module *Program* and *Programs*. Within Medos-2 V5, a process always executes for its owner (with the exception of the main process, which may change the program for which its work by calling a program or returning from a program).

Mutual Exclusion

Medos-2 provides mutual exclusion by so-called *priority monitors* (or simply *monitors*). A priority monitor is identified by the priority number of a Modula-2 module. It is guaranteed that upmost one process executes statements inside a certain monitor. Medos-2 for the Ceres computer distinguishes 16 such monitors by priority numbers from 0 to 15.
A process executing code inside a monitor may only call routines declared outside monitors, inside the same monitor or inside monitors with a higher priority number.
A monitor may be opened (for other processes) by a (direct or indirect) activation of one of the procedures defined by module Processes.

Synchronization

Synchronization between processes has to be programmed explicitly by the user, i.e. the program has to wait for the desired event (or condition). In such situations it is recommended to give a hint to the scheduler, that this process may only waste its time-slice. This is done by calling one of the procedures Pass, Delay, Wait, TimedWait, WaitInterrupt or TimedWaitInterrupt.

Interrupts

On occurance of an interrupt, the currently process is suspended if it is running without priority or with a priority that is lower than the number assigned to that interrupt. An interrupt handler will only be interrupted by an interrupt with a higher number.

Scheduling

The descriptors of the processes which can be activated are collected in the so-called *readylist*. Each process gets at its creation time a so-called *base priority* which is a parameter for the scheduler.
If the clock interrupt occurs and the currently running process runs without any priority it will be preempted. Processes running on priority for a longer timeperiod has to call explicitly the scheduler. Otherwise no other process will get the CPU.

9.2.4. Explanations

CreateProcess(p, size, done)
: Procedure *CreateProcess* creates a new process that will execute procedure *p*. It reserves a memory area of size *size* in the heap area of that program. This area is used by the new process as its stack area. The return parameter *done* is set TRUE, if the creation of the process was successfull.

Pass
: Procedure *Pass* is the explicit call to the scheduler. The descriptor of the calling process is appended at the end of the *readylist*.

Delay(ms)
: The calling process won't be activated for at least *ms* milliseconds.

Send(s)
: Procedure *Send* includes the descriptor of a process that is waiting for *s* into the *readylist*. If no process is currently waiting for *s* *Send* has no effect.

SendAll(s)
: Procedure *SendAll* includes the descriptors of all processes that are waiting for *s* into the *readylist*. If no process is currently waiting for *s* *SendAll* has no effect.

Wait(s)
: Procedure *Wait* deactivates the calling process and excludes its descriptor from the *readylist*. The descriptor will only be reincluded into that list due to a call of an other process to *Send* or *SendAll* with *s* as parameter.

TimedWait(s, ms)
: Procedure *TimedWait* deactivates the calling process and excludes its descriptor from the *readylist*. The descriptor will be reincluded into that list due to a call of an other process to *Send* or *SendAll* with *s* as parameter or the process is waiting for that event as long as *ms* milliseconds.

CreateDriver(p, size, dev, done)
: Procedure *CreateDriver* creates a new process that will execute procedure *p*. It reserves a memory area of size *size* in the heap area of that program. This area is used by the new process as its stack area. The priority of the caller has to be at least a high as the specified interrupt number specified in parameter *dev*. The return parameter *done* is set TRUE, if the creation of the process was successful.

WaitInterrupt
: Procedure *WaitInterrupt* installs the calling process as interrupt handler of that interrupt that has been specified in the parameter *dev* of *CreateDriver*. Only processes that are created by *CreateDriver* may call *WaitInterrupt*. The process will be reactivated as soon as the according interrupt will occur.

TimedWaitInterrupt(ms)
: Procedure *TimedWaitInterrupt* installs the calling process as interrupt handler of that interrupt that has been specified in the parameter *dev* of *CreateDriver*. Only processes that are created by *CreateDriver* may call *TimedWaitInterrupt*. The handler will wait for that interrupt only *ms* milliseconds, i.e. it will be activated after that period has passed by.

9.2.5. Implementation Notes

The minimal working space for a process is as long as 200 bytes.
The number of processes is restricted to 16. Medos itself installs 4 processes.
The system clock is assigned to device number 15. If a process is running with this priority the system clock is disabled. Processes which run with that priority can't be stopped by the user.

9.2.6. Examples

An interrupt handler is typically written according to the following scheme:

```
MODULE DevHandle[13]; (* priority >= interrupt number *)
  FROM Processes IMPORT CreateDriver, WaitInterrupt;
  CONST wsp = ... ; (* wsp >= 200 *)
  VAR ok: BOOLEAN; wsp: CARDINAL;

  PROCEDURE DevDriver;
  BEGIN
   (* initialization of driver and device *)
    LOOP
      WaitInterrupt;
      (* interrupt handling *)
    END
  END DevDriver;
  ...
BEGIN
  CreateDriver(DevDriver, wsp, 13, ok);
  IF NOT ok THEN ... END;
END DevHandle
```

9.3. Program

9.3.1. 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 9.3.2. The program concept and explanations needed for the activation of a program are given in chapter 9.3.3. Possible error messages are listed in 9.3.4. The object file format may be inspected in 9.3.5.

9.3.2 Definition Module Program

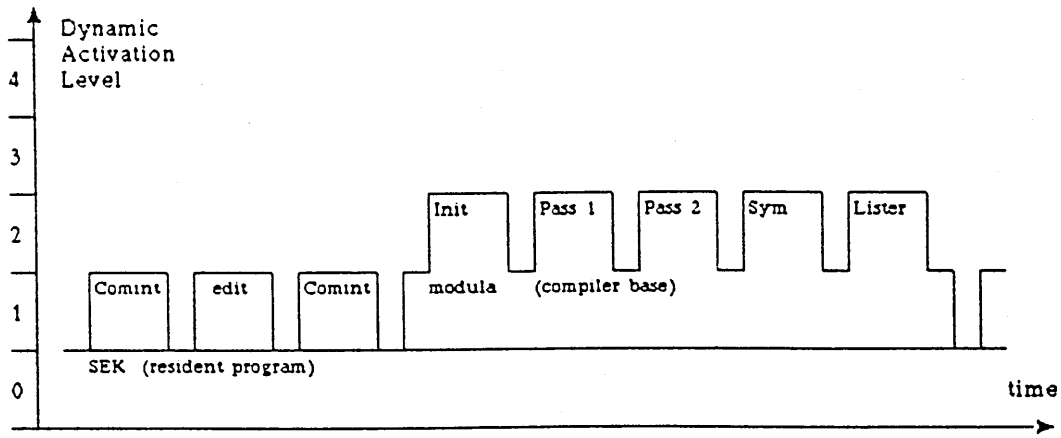
```
DEFINITION MODULE Program;

  FROM SYSTEM IMPORT ADDRESS;

  TYPE
    Status = (normal, warned, halted, killed, executionError, callError,
              moduleNotFound, moduleAlreadyLoaded, loadError);

  PROCEDURE Call(programName: ARRAY OF CHAR; shared: BOOLEAN; VAR st: Status);
  PROCEDURE Include(programName: ARRAY OF CHAR; VAR st: Status);
  PROCEDURE Terminate(warn : BOOLEAN);

END Program.
```

9.3.3. 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 *.OBN* 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 *.OBN* 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.OBN`
Second searched file	`DK.SYS.directory.OBN`
Module name	`Storage`
First searched file	`DK.Storage.OBN`
Second searched file	`DK.LIB.Storage.OBN`

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 = warned	Terminate(TRUE) called in program
status = halted	HALT called in program
status = killed	Program terminated by CTRL c from keyboard
status = executionError	Some execution error detected
status IN {callError .. loadError}	Some load error detected

Include(programname, status)

Procedure *Include* loads and initializes the program (or library-module) *programname*. The loaded program is part of the callers program, and it remains loaded as long as the caller of Include remains loaded. *status* indicates the success of the program inclusion. The meanings of status are given above.

Terminate(warn)

The execution of a program may be terminated by a call to *Terminate*. If *warn* is FALSE, *normal* is returned as status to the calling program, otherwise *warned* is returned as status.

Implementation Notes

The current implementation of procedure *Call* may only be called from the *mainprocess*, i.e. the process within which function *MainProcess* of module *Programs* returns TRUE.

Only up to 127 modules may be loaded at any time. The resident part of Medos-2 consists of 23 modules. The loader can handle up to 40 already imported but not yet loaded modules.

The maximum number of active programs is 8.

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;
  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.
```

9.3.4. 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/Include: incompatible module`
  'module name' `on file` *'file name'*

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

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

  The listed modules were not found.

9.3.5. Object Code Format

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

```
Module      = HeaderBlock ImportBlock EntryBlock LinkBlock
              {CodeBlock | DataBlock | AlignBlock}.
HeaderBlock= MODULE BlockSize
              VersionNumber Flags LinkSize VarSize ConstSize CodeSize ModuleName.
ImportBlock= IMPORT BlockSize NofImports {ModuleName}.
EntryBlock = ENTRY BlockSize {EntryAddress}.
                                (*BlockSize/2 = no. of entries*)
LinkBlock  = LINK BlockSize {ModuleNumber ProcNumber}.
                                (*BlockSize/2 = no. of links*)
CodeBlock  = CODE BlockSize Offset {Byte}.
DataBlock  = DATA BlockSize Offset {Byte}.
AlignBlock = ALIGN BlockSize {Byte}.

BlockSize     = Word.
VersionNumber = Word.
Flags         = Word.

ModuleName       = ModuleKey ModuleIdentifier.
ModuleIdentifier = String.
ModuleKey        = Word Word Word.

VarSize    = Word.
ConstSize  = Word.
CodeSize   = Word.
LinkSize   = Word.
NofImports = Word.

EntryAddress = Word.
ModuleNumber = Byte.
ProcNumber   = Byte.
Offset       = DoubleWord.

MODULE     = 81H.
IMPORT     = 82H.
```

```
ENTRY       = 83H.
LINK        = 84H.
CODE        = 85H.
DATA        = 86H.
ALIGN       = 87H.
```

BlockSize, *VarSize*, *ConstSize*, *CodeSize*, *LinkSize*, and *Offset* are numbers of bytes. The block size does not include itself, nor its preceding specifier. The first character of a string indicates the length of the string (including itself).
The fixup frame of the Lilith version is replaced by two new blocks: the entry block and the link block. The former contains the list of entry addresses of the module's exported procedures, and the link block establishes the link table. Each pair <module number, procedure number> is translated into the corresponding MOD / PC pair, where the PC value is taken from the entry block of the referenced module. The "procedure number" 255 is an exception: the link table entry is loaded with the referenced module's data address (SB).
The size of the data area is the sum of *VarSize* and *ConstSize*; the area for constants follows that of the variables, and the SB register points to the beginning of the constant area, and thereby also to the end of the variable area. The loader places the link table address into the first 4 bytes of the constant area. Bit 0 of byte 4 is used as an initialization flag and is cleared by the loader. Bytes 4 - 7 are reserved for the system. The link table, the data area, the workspace, and the code are allocated at addresses which are multiples of 4. The lengths of the data and code segments are multiples of 4, and they are properly aligned in the file.

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

9.4. Programs

Module Programs enables to activate (call) and terminate execution of programs.

NOTE: This module should only be used by users who are very familiar with Medos-2.

```
DEFINITION MODULE Programs;

  TYPE
    Status = (normal, warned, halted, killed, executionError, callError);

  PROCEDURE Call(p: PROC; shared: BOOLEAN; VAR st: Status);
  PROCEDURE Terminate(warn: BOOLEAN);
  PROCEDURE Kill;
  PROCEDURE CurrentStatus(): Status;

  PROCEDURE MainProcess(): BOOLEAN;

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

  PROCEDURE InitProcedure(init: PROC; VAR done: BOOLEAN);
  PROCEDURE TermProcedure(term: PROC; VAR done: BOOLEAN);

END Programs.
```

Explanations

Call(p, shared, st)
: Procedure *Call* activates a new program that executes procedure *p*. The current level is incremented by one before the execution of the program. If shared is TRUE, the shared level of the new program remains unchanged, otherwise it is set equal to the respective current level. After the execution of the program *st* indicates the termination cause.

Terminate(warn)
: The currently running process may be terminated by a call to *Terminate*. If *warn* is FALSE the process terminates normally. If *warn* is TRUE the process terminates with status *warned*.

Kill
: A call to procedure *Kill* terminates the currently running program.

CurrentStatus(): Status
: Function *CurrentStatus* returns the state of the program for which the current process runs.

MainProcess(): BOOLEAN
: Function *MainProcess* returns TRUE if the currently executed process is the one which executes the initialization part of the main module in the running program.

CurrentLevel(): CARDINAL
: Function *CurrentLevel* returns the program level of the calling process.

SharedLevel(): CARDINAL
: Function *SharedLevel* returns the level number of the lowest program sharing resources with the calling process's level

InitProcedure(init, done)
: A call to procedure *InitProcedure* causes that the procedure *init* will be called whenever a program on a higher level will be started. These so-called initialization procedures are called just after the installation of the new execution level in question (i.e. after current level and shared level are incremented) and in order of their announcement. The result parameter *done* is set TRUE if the assignment was done.

TermProcedure(term, done)

A call to procedure *TermProcedure* causes that the procedure *term* will be called whenever a program on a higher level is terminated. These so-called termination procedures are called just before the termination of the execution level in question (i.e. before current level and shared level are reset) and in reverse order of their announcement. The result parameter *done* is set TRUE if the assignment was done.

Implementation Note

The totally number of installed initialization and termination procedures is limited to 32.

9.5. Heap

This module handles for each shared level its own heap.

NOTE: This module should only be used by users who are very familiar with Medos-2.

```
DEFINITION MODULE Heap;

  FROM SYSTEM IMPORT ADDRESS;

  PROCEDURE Allocate(VAR a: ADDRESS; size: LONGINT);
  PROCEDURE Deallocate(VAR a: ADDRESS; size: LONGINT);

END Heap.
```

Explanations

Allocate(a, size)
: Procedure *Allocate* tries to allocate a memory area of the given size *size* in the heap that belongs to the shared level of the caller. If that space is not available, *a* returns NIL otherwise it returns the address of the reserved area.

Deallocate(a, size)
: Procedure *Deallocate* releases the memory area given by address *a* and size *size*. Deallocate checks if the given memory area resides inside a heap. If the check failed the program is halted. *a* returns the value NIL.

Implementation Note

The minimal memory area that will be reserved by *Allocate* is as big as 8 bytes.
Only the heap of the program that runs on the topmost level can be expanded. All other heaps are frozen at that moment a new program level is activated.

9.6. SEK

Module SEK (Sequentiel Executive Kernel) is the main program of the operating system Medos-2. The module is actually the resident part of the standard command interpreter. Currently the two nonresident parts of the command interpreter are the program Comint and CommandFile. The module also serves the configuration of the system by importing (directly or indirectly) the needed modules.

```
DEFINITION MODULE SEK;

  FROM Program IMPORT Status;

  PROCEDURE CallComint(loop: BOOLEAN; VAR st: Status);
  PROCEDURE PreviousStatus(): State;

  PROCEDURE NextProgram(programname: ARRAY OF CHAR);
  PROCEDURE SetParameter(param: ARRAY OF CHAR);
  PROCEDURE GetParameter(VAR param: ARRAY OF CHAR);

  PROCEDURE Login(): BOOLEAN;
  PROCEDURE LeaveLogin;

  PROCEDURE TestDK(actualstate: BOOLEAN): BOOLEAN;

END SEK.
```

Explanations

CallComint(loop, st)
: A call to procedure *CallComint* activates the standard command interpreter. If *loop* is TRUE, the command interpreter repeatedly reads in commands and activates the corresponding programs. The loop is terminated when the command interpreter reads an ESC character. If *loop* is FALSE only one single command is interpreted. The return parameter *st* reflects the success of the most recently executed program.

PreviousStatus(): Status;
: Function *PreviousStatus* returns the status of the most recently executed program.

NextProgram(programname)
: A program activated by module *SEK* may by call to procedure *NextProgram* define, which program should be executed after its own termination. If a program makes no call to *NextProgram*, the command interpreter will be executed after the termination of the program.

SetParameter(param)
: By a call to procedure *SetParameter*, a program may pass over a textual parameter to the following program.

GetParameter(param)
: By a call to procedure *GetParameter*, a program receives the parameter passed over to it from the previous program.

Login(): BOOLEAN
: The function *Login* is TRUE during the login period, i.e. from system intialization time until procedure *LeaveLogin* has been called.

LeaveLogin
: A call to procedure *LeaveLogin* terminates the login period, i.e. the period in which the function *Login* is TRUE.

TestDK(actualstate): BOOLEAN
: Function *TestDK* senses the state of the disk drive. If the argument *actualstate* is TRUE, the value of the function is the sensed state, otherwise *TestDK* is only TRUE, if the current state of the drive is ok and all activations of *TestDK* since the most recent system initialization found the state of the drive to be ok.

Implementation Notes

Procedure *GetParameter* acceptes only the first 64 characters of the argument *param*. If the argument to *param* is "smaller" than the string handled over by *SetParameter* was, the actual argument is truncated to the length of the argument to *GetParam*.
If the disk drive has been in a not ready state since the last system initialization, and this has been detected by a call to *TestDK*, the system is reinitialized when the program on level one is terminated.

9.7. TerminalBase

Module TerminalBase makes it possible for programs to define their own read and write procedures for module Terminal. For example, this facility is needed, if the normal keyboard input has to be substituted by the text in a command file or if the terminal output has to be written to a log file.

```
DEFINITION MODULE TerminalBase;

  TYPE ReadProcedure = PROCEDURE(VAR CHAR);
  PROCEDURE AssignRead(rp: ReadProcedure; VAR done:BOOLEAN);
  PROCEDURE Read(VAR ch: CHAR);

  TYPE WriteProcedure = PROCEDURE(CHAR);
  PROCEDURE AssignWrite(wp: WriteProcedure; VAR done:BOOLEAN);
  PROCEDURE Write(ch: CHAR);

END TerminalBase.
```

Explanations

AssignRead(rp, done)
: By a call to *AssignRead*, the terminal input procedure for the current program is set to be procedure *rp*. The procedure *rp* must be similar to procedure *BusyRead* in module *Terminal*, i.e. it must return character 0C, if no input is available. A previous assignment of a read procedure will be overwritten by a new call to *AssignRead* in the same program. The result paramater *done* is set TRUE, if the assignment was done.

Read(ch)
: Procedure *Read* reads in the next character. If no character is available, character 0C is returned. *Read* normally activates the read procedure belonging to the highest program level, within which *AssignRead* has been called. If, however, *Read* is called from an assigned read procedure, that read procedure is activated, which was assigned on the highest program level below the level, on which the current executing read procedure was assigned.

AssignWrite(wp, done)
: By a call to *AssignWrite*, the terminal output procedure for the current program level is set to be procedure *wp*. A previous assignment of a write procedure will be overwritten by a new call to *AssignWrite* on the same program level. The result parameter *done* is set TRUE, if the assignment was done.

Write(ch)
: Procedure *Write* writes out the next character. *Write* normally activates the write procedure belonging to the highest program level, within which *AssignWrite* has been called. If, however, *Write* is called from an assigned write procedure, that write procedure is activated, which was assigned on the highest program level below the level, on which the current executing write procedure was assigned.

Implementation Restrictions

Read procedures and write procedures can "only" be assigned on five different program levels. The return parameter *done* is set FALSE, if a further assignment would exceed this limit.

9.8. Terminal

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;

  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 "echo" 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 this character

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

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

9.9. Users

The module *Users* serves the identification of (not necessarily) human users within Medos-2. A user is uniquely identified by a pair of numbers, namely the *group* and the *member-of-group* number. A user-chosen password is also encoded into a pair of numbers.

```
DEFINITION MODULE Users;

  TYPE
    User = RECORD
             group, member: CARDINAL;
             password1, password2: CARDINAL
           END;

  PROCEDURE GetUser(VAR u: User);
  PROCEDURE SetUser(u: User; VAR done: BOOLEAN);
  PROCEDURE ResetUser

END Users.
```

Explanations

A program is executed on behalf of a certain user, the so-called *real user* of a running program. Each process executes, however, on behalf of a so-called *current user*. The *current user* and the *real user* are handled according to the following rules:

1) The real user of a program is set when the program is called (activated) and cannot be changed. It is set equal to the current user of the calling process.

2) The current user of a process activating a program is not changed by the activated program (i.e. the current user of a process just after a program activation is equal to its current user just before the program activation).

3) The current user of a new process is initially set equal to the current user of its creator process.

GetUser(u)
: Procedure *GetUser* returns the current user of the current process.

SetUser(u, done)
: Procedure *SetUser* sets the *current user* of the current process. If the assignment contradicts a security rule, the assignment is not done and the parameter *done* is set FALSE.

ResetUser
: Procedure *ResetUser* sets the current user of the current process equal to the *real user* of the program, within which it executes.

The Assignment of Group and Member

The (group, member) pair has the following semantic:

```
          group =     0B        no user
   1B  <= group <= 77777B       normal user groups   (* clients *)
100000B<= group <=177777B       trusted user groups  (* servers *)
          group = 100000B       os group
```

Members of a group may have any member number. The *current user* cannot be set to a user of a trusted group, if the *real user* is not from a trusted group.

The Assignment of Password1 and Password2

The (password1, password2) pair has the following semantic:

```
          password1 =    0      no password
    1  <= password1 <177777B normal password
          password1 =177777B special password
```

Password2 may have any value. The internal password (password1, password2) is encoded from a string by the following algorithm. The password string should be restricted to an identifier beginning with a letter and followed by letters or digits. Note, that the procedure ConvertPassword does not generate a special password!

```
PROCEDURE ConvertPassword(password: ARRAY OF CHAR;
                          VAR pw1, pw2: CARDINAL);
  VAR c, h: CARDINAL;
BEGIN c := 0;
  pw1 := 0; pw2 := 0;
  WHILE (c <= HIGH(password)) & (password[c] # 0C) DO
    h := pw2; pw2 := pw1;
    pw1 := (h MOD 509 + 1) * 127 + ORD(password[c]);
    INC(c)
  END
END ConvertPassword
```

9.10. Module Clock

Module Clock is used to obtain the current date and time of day stored in a real-time clock chip. The module is also used to initialize (or correct) the internal clock setting.

```
DEFINITION MODULE Clock;

  TYPE
    Time = RECORD
             weekday:     CARDINAL; (* mo = 0, .., so = 6 *)
             day:         CARDINAL; (* ((year - 1900)*20B + month)*40B + day *)
             minute:      CARDINAL; (* hour*60 + minute *)
             millisecond: CARDINAL; (* second*1000 + millisecond *)
           END;

  PROCEDURE SetTime(t: Time);
  PROCEDURE GetTime(VAR t: Time);

END Clock.
```

Explanations

Time

The field *day* is defined to be equal to (year - 1900) * 512 + month * 32 + day. The field *minute* is the minute within the given day and must therefore be in the range 0 to 24 * 60 - 1. The field *millisecond* indicates the millisecond within the given minute of the day; its value must consequently be in the range 0 to 60 * 1000 - 1. The field *weekday* reflects the day of the week.

SetTime(time)

The internally maintained clock can be set by calls of procedure *SetTime*. If the argument to *SetTime* is apparently incorrect, the internal clock is not adjusted. The correct execution of procedure *SetTime* can be validated by geting the time just after the clock has been set and a comparison of the new time with the set time.

GetTime(time)

The internally maintained clock can be read by calls of procedure *GetTime*. The field day of the variable *time* has the value zero as long as the clock has not been set.

Implementation Notes

The field millisecond is actualized at a granularity of about 20 millisecond.

Module Clock is programmed with priority 7. Consequently, procedure SetTime and GetTime cannot be called by processes executing on a priority higher than 7 (i.e. executing with priority 8 to 15).

10. Screen Software

The screen software chapter describes the following modules:

CursorMouse	(10.2.)
Menu	(10.3.)
Windows	(10.4.)
TextWindows	(10.5.)
GraphicWindows	(10.6.)
DisplayDriver	(10.7.)
RasterOps	(10.8.)
Fonts	(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 (9.8.) and is not explained in the screen software chapter. Formatting modules (e.g. *InOut*) are found in the chapter 11.

The modules *CursorMouse*, *Menu* and *Windows* provide a de-facto standard package for window applications under Medos-2. *TextWindows* and *GraphicWindows* are modules based on Windows that define routines for character output and simple graphics within windows. The whole package is described in [1].

The modules *DisplayDriver*, *RasterOps* and *Fonts* form the interface to the display hardware of Ceres.

Throughout this chapter some articles are referenced which contain more information about the screen software. They are summarized below.

[1] J. Gutknecht: Mouse and Bitmap Display - System Programming in Modula-2
Institut für Informatik, report 56, ETH Zürich 1983

[2] F.Peschel, M.Wille: Porting Medos-2 onto the Ceres workstation,
Institut für Informatik, ETH Zürich, to be published

[3] J. Gutknecht: Ceres Font Machinery
Institut für Informatik, internal paper, ETH Zürich 1986

10.2. CursorMouse

This module can be used to request the mouse buttons and the mouse position. A cursor can be displayed on the screen as a visual feedback of the mouse movements. A description of this module is included in [1].

Definition Module

```
DEFINITION MODULE CursorMouse;

  IMPORT RasterOps;

  CONST ML = 2; MM = 1; MR = 0;

  TYPE
    Pattern = RasterOps.Pattern;

  PROCEDURE SetMouse(x, y: CARDINAL);
  (*Set Mouse to point (x, y)*)

  PROCEDURE GetMouse(VAR s: BITSET; VAR x, y: CARDINAL);
  (*Get current mouse state
    ML IN s = "Left mouseKey pressed";
    MM IN s = "Middle mouseKey pressed";
    MR IN s = "Right mouseKey pressed"*)

  PROCEDURE MoveCursor(x, y: CARDINAL);
  (*Move cursor to specified location*)

  PROCEDURE EraseCursor;

  PROCEDURE SetPattern(VAR p: Pattern);
  (*Activate private cursor pattern*)

  PROCEDURE ResetPattern
  (*Reactivate standard arrow pattern*)

END CursorMouse.
```

Explanations

Procedure *SetMouse* sets the mouse coordinates to the point *(x,y)* on the screen.
GetMouse returns the state of the buttons and the position of the mouse.

MoveCursor moves the cursor to screen position (x, y).
EraseCursor deletes the cursor on the screen.

SetPattern is used to install a private cursor pattern. Subsequent calls of *MoveCursor* will use this pattern. The cursor need not to be erased before calling SetPattern.
ResetPattern returns to the default pattern (arrow) after *SetPattern* has been called.

Imported Modules

Resident system modules

10.3. Menu

This module implements so called pop-up menus that allow a graphical selection of commands using the mouse. A description of this module is included in [1].

Definition Module

```
DEFINITION MODULE Menu;

  PROCEDURE ShowMenu(X, Y: CARDINAL;
    VAR menu: ARRAY OF CHAR; VAR cmd : CARDINAL)

  (*menu  = title {"|" item} .
    item  = name ["(" menu ")"] .
    name  = {char} .
    char  = 'any character except 0C, "|", "(", ")"' .
    title = name .

    Nonprintable characters and characters exceeding
    maximum namelength are ignored.

    The input value of "cmd" specifies the command initially
    to be selected.

    The sequence of selected items is returned via
    the digits of "cmd" (from right to left),
    interpreted as an octal number*)

END Menu.
```

Explanations

Procedure *ShowMenu* draws a menu at screen position (X, Y), waits for a command selection and returns the selected command code as soon as all mouse buttons are released. Variable *cmd* contains the (coded) command number. *cmd = 0* --> no command has been selected at all.

Submenus are specified in paranthesis after an item name, a continuation menu by an empty item name followed by a menu in paranthesis.

Imported Modules

CursorMouse
Resident system modules

10.4. Windows

This is a library module for handling windows on the display. A window is a rectangle on the screen where text and graphic operations may be performed like on a regular display. *Windows* can be compared to pieces of paper on a desk. They can overlay each other in arbitrary order and can be moved. A complete description of the implementation of the Ceres window package is given in [1].

Definition Module

```
DEFINITION MODULE Windows;

  CONST Background = 0; FirstWindow = 1; LastWindow = 15;

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

    RestoreProc = PROCEDURE(Window);

  PROCEDURE OpenWindow(VAR u: Window; x,y,w,h: CARDINAL;
    Repaint: RestoreProc; VAR done: BOOLEAN);
  (*Open a new window u and initialize its rectangle
    by clearing it and drawing a frame of width 1;
    procedure Repaint will be called when restoration
    becomes necessary*)

  PROCEDURE DrawTitle(u: Window; title: ARRAY OF CHAR);
  (*Draw title bar; height = lineHeight of default font*)

  PROCEDURE RedefineWindow(u: Window; x,y,w,h: CARDINAL;
    VAR done: BOOLEAN);
  (*Change and reinitialize rectangle of window u*)

  PROCEDURE CloseWindow(u: Window);

  PROCEDURE OnTop(u: Window): BOOLEAN;

  PROCEDURE PlaceOnTop(u: Window);

  PROCEDURE PlaceOnBottom(u: Window);

  PROCEDURE UpWindow(x,y: CARDINAL): Window
  (*Return the uppermost opened window containing (x,y),
    if there is any, and take the value Background otherwise*)

  PROCEDURE GetWindowFrame(u: Window; VAR x, y, w, h: CARDINAL);

END Windows.
```

Imported Modules

Resident system modules

10.5. TextWindows

TextWindows is a library module for writing non-proportional text into a window. Text windows may be operated as scrolling displays or as forms (with positioning). A description of this module is included in [1].

Definition Module

```
DEFINITION MODULE TextWindows;

  IMPORT Windows;

  TYPE Window = Windows.Window;
       RestoreProc = Windows.RestoreProc;

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

  PROCEDURE OpenTextWindow(VAR u: Window; x, y, w, h: CARDINAL;
                           name: ARRAY OF CHAR);
    (*Open new standard text window. Draw title bar iff "name" is not empty*)

  PROCEDURE RedefTextWindow(u: Window; x,y,w,h: CARDINAL);
    (*Redefine and reinitialize window u*)

  PROCEDURE CloseTextWindow(u: Window);

  PROCEDURE TextWindowHeight(lines: CARDINAL):CARDINAL;
    (*returns the window height needed to have the given
      number of lines when the default font is used*)

  PROCEDURE TextWindowSize(u: Window; VAR lines, columns: CARDINAL);
    (*returns the actual size of the window u according to
      the default font size*)

  PROCEDURE AssignFont(u: Window; frame, charW, lineH: CARDINAL);
    (*Assign non-proportional font at frame-address f to window u.
      Character width = w, Character height = h*)

  PROCEDURE AssignRestoreProc(u: Window; r: RestoreProc);
    (*Assign procedure to restore window u*)

  PROCEDURE AssignEOWAction(u: Window; r: RestoreProc);
    (*Assign procedure to react on "end of window" condition
      for window u*)

  PROCEDURE ScrollUp(u: Window);
    (*Scroll one line up in window u (standard EOW-action)*)

  PROCEDURE DrawTitle(u: Window; name: ARRAY OF CHAR);

  PROCEDURE DrawLine(u: Window; line, col: CARDINAL);
    (*col  = 0: draw horizontal line at line;
      line = 0: draw vertical line at col*)

  PROCEDURE SetCaret(u: Window; on: BOOLEAN);
    (*on TRUE (FALSE): turn caret on (off)*)

  PROCEDURE Invert(u: Window; on: BOOLEAN);
    (*on TRUE (FALSE): set to inverse video (normal) mode*)

  PROCEDURE IdentifyPos(u: Window; x, y: CARDINAL; VAR line, col: CARDINAL);

  PROCEDURE GetPos(u: Window; VAR line, col: CARDINAL);
    (*Get current caret position of window u*)

  PROCEDURE SetPos(u: Window; line, col: CARDINAL);
    (*Set position within window u*)
```

```
  PROCEDURE ReadString(u: Window; VAR a: ARRAY OF CHAR);
    (*Read string and echo to window u*)

  PROCEDURE ReadCard(u: Window; VAR x: CARDINAL);
    (*Read cardinal and echo to window u.
      Syntax: cardinal = digit {digit}.
      Leading blanks are ignored*)

  PROCEDURE ReadInt(u: Window; VAR x: INTEGER);

  PROCEDURE Write(u: Window; ch: CHAR);
    (*Write character ch at current position.
      Interpret BS, LF, FF, CR, CAN, EOL and DEL characters*)

  PROCEDURE WriteLn(u: Window);

  PROCEDURE WriteString(u: Window; a: ARRAY OF CHAR);

  PROCEDURE WriteCard(u: Window; x, n: CARDINAL);
    (*Write x with (at least) n characters.
      If n is greater than the number of digits needed,
      blanks are added preceding the number*)

  PROCEDURE WriteInt(u: Window; x: INTEGER; n: CARDINAL);

  PROCEDURE WriteOct(u: Window; x, n: CARDINAL);

  PROCEDURE WriteHex(u: Window; x, n: CARDINAL);

END TextWindows.
```

Imported Modules

Windows
Resident system modules

10.6. GraphicWindows

GraphicWindows is a library module for drawing lines, circles and filled rectangles in a window (supports "turtle graphics"). A description of this module is included in [1].

Definition Module

```
DEFINITION MODULE GraphicWindows;
  IMPORT Windows, RasterOps;

  TYPE Window = Windows.Window;
    RestoreProc = Windows.RestoreProc;

    Mode = RasterOps.Mode;

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

  PROCEDURE OpenGraphicWindow(VAR u: Window; x,y,w,h: CARDINAL;
    name: ARRAY OF CHAR; Repaint: RestoreProc);
    (* Open new graphic window. Draw title bar if "name" not empty *)

  PROCEDURE RedefGraphicWindow(u: Window; x,y,w,h: CARDINAL);
    (* Change rectangle and reinitialize graphic window u *)

  PROCEDURE Clear(u: Window);
    (* clear window u *)

  PROCEDURE CloseGraphicWindow(u: Window);
    (* close window u *)

  PROCEDURE SetMode(u: Window; m: Mode);
    (* set mode "replace, paint, invert or erase" *)

  PROCEDURE Dot(u: Window; x, y: CARDINAL);
    (* place dot of current mode at coordinate x,y *)

  PROCEDURE SetPen(u: Window; x,y: CARDINAL);
    (* set pen at (x, y); window need not be on top *)

  PROCEDURE TurnTo(u: Window; d: INTEGER);
    (* turn direction of the pen to d degrees *)

  PROCEDURE Turn(u: Window; d: INTEGER);
    (* turn direction of the pen by d degrees *)

  PROCEDURE Move(u: Window; n: CARDINAL);
    (* move pen and draw line of length n in direction specified before *)

  PROCEDURE MoveTo(u: Window; x,y: CARDINAL);
    (* move pen and draw line to (x, y) *)

  PROCEDURE Circle(u: Window; x, y, r: CARDINAL);
    (* draw circle with center at (x, y) and radius r *)

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

  PROCEDURE CopyArea(u: Window; sx,sy,dx,dy,dw,dh: CARDINAL);
    (* copy rectangular area at (sx, sy) into rectangle at (dx, dy)
       of width dw and height dh *)

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

  PROCEDURE WriteString(u: Window; s: ARRAY OF CHAR);
    (* write string s at pen's position *)

  PROCEDURE IdentifyPos(VAR u: Window; VAR x,y: CARDINAL);
```

```
    (* return uppermost opened window and the
       window oriented coordinates (x, y)
       for given screen coordinates (x, y) *)

END GraphicWindows.
```

Restrictions

Graphic objects (e.g. a circle) are only drawn if they will lie completely in the specified window. If something has to be drawed in a window overlaid by another window, it is first put on top. In all other cases nothing is displayed and *Done* is set to FALSE.

Imported Modules

Windows
Resident system modules

10.7. DisplayDriver

Module *DisplayDriver* gives access to the display hardware. A description is included in [2].

Definition Module

```
DEFINITION MODULE DisplayDriver;

  FROM RasterOps IMPORT BitMap;
  FROM SYSTEM IMPORT ADDRESS;

  VAR BMD, BMDB: BitMap;                    (* BMD  = default bitmap desc.
                                               BMDB = background bitmap desc.*)

  PROCEDURE BuildBmd(ptr: ADDRESS; width, height: CARDINAL; VAR bmd: BitMap);
    (* build a bitmap desc. for a bitmap starting at address 'ptr'
       with size width x height. Note: width must be a multiple of 32! *)

  PROCEDURE ScreenWidth(): CARDINAL;
  PROCEDURE ScreenHeight(): CARDINAL;

  PROCEDURE CharWidth(): CARDINAL;
  PROCEDURE LineHeight(): CARDINAL;
  PROCEDURE MapHeight(): CARDINAL;

  PROCEDURE ShowBitmap(on, inv: BOOLEAN);
    (* on  -> show or suppress the display of the current bitmap
       inv -> invert the display, i.e. display black on white *)

  PROCEDURE SwitchBitmap(bank: CARDINAL; VAR done: BOOLEAN);
    (* switch between forground and backgound bitmap;
       bank = 0 denotes the default bitmap BMD;
       bank = 1 denotes the background bitmap BMDB;
       Note: The user has to manage which bitmap is currently
             displayed. However, after termination of a user
             application the display is reset to BMD *)

  PROCEDURE Write(ch: CHAR);
    (* display a character on the systems default bitmap BMD.
       the following control characters are interpreted:
       10C BS  backspace one character
       12C LF  next line, same x position
       14C FF  clear page
       15C CR  return to start of line
       30C CAN clear line
       36C EOL next line
      177C DEL backspace one character and clear it.

      Note: Write supports up to 256 characters per line and
            allows deletion only within one line. *)

END DisplayDriver.
```

Imported modules

Resident system modules

10.8. RasterOps

The module RasterOps defines the primitive operations on the Ceres raster scan display. Because the operations work on the whole screen instead of windows they should be used carefully. The module is described in [2].

Definition Module

```
DEFINITION MODULE RasterOps;

  FROM SYSTEM IMPORT ADDRESS;
  FROM Fonts IMPORT Font;

  CONST maxheight = 16;

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

       Block  = RECORD
                  x, y, w, h : CARDINAL
                END;

       Rect   = RECORD
                  xd, yd, xu, yu: CARDINAL
                END;

       BitMap = RECORD
                  bmPtr : ADDRESS;
                  width, height : CARDINAL;
                  clipB: Block; (* for further use *)
                  clipR: Rect (* for further use *)
                END;

       Pattern = RECORD
                   pheight : CARDINAL;
                   pat : ARRAY [0..maxheight-1] OF LONGINT
                 END;

  PROCEDURE DDT (VAR bmd: BitMap; x, y: CARDINAL; m: Mode);
  PROCEDURE DCH (VAR bmd: BitMap; x, y: CARDINAL; font: Font;
                 ch: CHAR; m: Mode);
  PROCEDURE REPL (VAR bmd: BitMap; VAR blk: Block; VAR p: Pattern;
                  m: Mode);
  PROCEDURE BBLT (VAR sBmd: BitMap; VAR sBlk: Block;
                  VAR dBmd: BitMap; VAR dBlk: Block; m: Mode);

  PROCEDURE LIN (VAR bmd: BitMap; x1, y1, x2, y2: CARDINAL; m: Mode);
  PROCEDURE SCR (VAR bmd: BitMap; VAR blk: Block; lineheight: CARDINAL);
  PROCEDURE RECT (VAR bmd: BitMap; VAR blk: Block; m: Mode);

END RasterOps.
```

Restrictions

The fields *clipB* and *clipR* in the type *Bitmap* are restricted to the use by Medos-2. Changing these values is considered harmful.

Imported modules

Resident system modules

10.8. Fonts

The module *Fonts* allows the use of multiple different fonts on Ceres. It defines the default font which is used by the resident system as well as procedures to load arbitary fonts from the disk. Some routines provide information about particular characters. The font file format on Ceres is defined in [3]. The module Fonts itself is described in detail in [2].

Definition Module

```
DEFINITION MODULE Fonts;

  TYPE Font;

  PROCEDURE LoadFont(VAR fnt: Font; fn: ARRAY OF CHAR;
                     abstr: INTEGER; VAR err: INTEGER);
    (* loads the font in file 'fn' with abstraction 'abstr'
       into memory yielding a handle 'fnt' to be passed to
       DCH; errors are reported in 'err' *)
  PROCEDURE UnloadFont(VAR fnt: Font; VAR err: INTEGER);
    (* unloads the font 'fnt' by releasing its resources;
       errors are reported in 'err' *)

  PROCEDURE SysFont():Font;
    (* returns a handle to the current system font *)
  PROCEDURE DefFont():Font;
    (* returns a handle to the default font *)
  PROCEDURE SetDefFont(fnt: Font);
    (* allows to set the default font *)

  PROCEDURE ChW(fnt: Font; ch: CHAR):INTEGER;
    (* returns the width of a character 'ch' in font 'fnt' *)
  PROCEDURE ChBox(fnt: Font; ch: CHAR; VAR x, y, w, h: INTEGER);
    (* returns the box of character 'ch' in font 'fnt';
       x and y are relative to a baseline *)
  PROCEDURE FontBox(fnt: Font; VAR fnX, fnY, fnW, fnH: INTEGER);
    (* returns the maximal box of font 'fnt', i.e. the
       union of all character boxes in 'fnt' *)

END Fonts.
```

Imported modules

Resident system modules

11. Library Modules

This chapter is a collection of some commonly used library modules on Ceres. For each library module a *symbol file* and an *object file* is stored on the disk. The file names are derived from (the first 16 characters of) the module name, beginning with the prefix LIB and ending with the extension SMB for symbol files and the extension OBN 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.SMB
Object file name	DK.LIB.FileNames.OBN

List of the Library Modules

InOut	Simple handling of formatted input/output	11.1.
RealInOut	Formatted input/output of real numbers	11.2.
LongInOut	Formatted input/output of long integers	11.3
MathLib0	Basic mathematical functions	11.4.
ByteBlockIO	Input/output of byte blocks on files	11.5.
FileNames	Input of file names from the terminal	11.6.
Options	Input of program options and file names	11.7.
V24	Driver for the RS-232 (V24) line interface	11.8.
Profile	Reading from the user profile	11.9.
String	String handling	11.10.

11.1. InOut

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;
  FROM SYSTEM IMPORT WORD;
  FROM FileSystem IMPORT File;

  CONST EOL = 36C;
  VAR Done:  BOOLEAN;
    termCH:  CHAR;  (*terminating character in ReadInt, ReadCard*)
    in, out: File;  (*for exceptional cases only*)

  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 ReadWrd(VAR w: WORD);
    (*Done := NOT in.eof*)

  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);

  PROCEDURE WriteWrd(w: WORD);

END InOut.
```

11.2. RealInOut

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;
  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. LongInOut

This module can be used for formatted input/output of real numbers on terminal or files. It works together with the module InOut.

Imported Library Module

InOut

Definition Module

```
DEFINITION MODULE LongInOut;

  PROCEDURE WriteLongInt(x: LONGINT; n: INTEGER);
    (*Write x on current active stream of module InOut.
      n characters will be used, with x right justified.*)

  PROCEDURE ReadLongInt(VAR x: LONGINT);
    (*Read x from the active stream of module InOut.*)

END LongInOut.
```

Explanations

WriteLongInt(x, n)
: WriteLongInt writes *x* on the current output stream of module InOut. It uses *n* characters with x right justified.

ReadLongInt*(x)*
: ReadLongInt reads *x* from the active stream of module InOut.

11.4. MathLib0

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

Definition Module

```
DEFINITION MODULE MathLib0;

  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.
```

11.5. ByteBlockIO

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;

  FROM SYSTEM IMPORT BYTE, ADDRESS;

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

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

  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 to or from the file f. The number of the actually read bytes is assigned to *actualcount*.

Example

```
MODULE ByteBlockIODemo;

  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 $2^{15} - 1$ bytes.

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 the MOVE function provided by the module SYSTEM.

11.6. FileNames

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;

  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, must 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. In order to allow filenames for the hierarchical filesystem of the future operating system Vamos the ':' is accepted as an additional separator. Of course these names are invalid in the context of the Medos-2 filesystem.

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 accepted

Error Message

ReadFileName called with incorrect default

Imported Module

Terminal

11.7. Options

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;

  TYPE Termination = (normal, empty, cancel, escape);

  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 first reads a file name by calling procedure ReadFileName of module FileNames. If acceptOption is TRUE, it afterwads reads program options from the terminal. The procedure 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
cancel	CTRL-X was typed, input line is cancelled
escape	ESC was typed, no file is specified.

Consider that *name[0] = 0C* when *term <> normal*.

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.

Example

```
LOOP
  Terminal.WriteString("file > ");
  Options.FileNameAndOptions(default, name, termstat, TRUE);
  IF termstat = Options.escape THEN
    Terminal.WriteString(" -- no file");
    Terminal.WriteLn;
    continue := FALSE;
    EXIT;
  ELSIF termstat = Options.cancel THEN
    Terminal.WriteString(" -- cancelled");
    Terminal.WriteLn;
  ELSE (* normal OR empty *)
    IF termstat = Options.empty THEN
      Terminal.WriteString(default);
      name := default;
    END;
    Terminal.WriteLn;
    FileSystem.Lookup(file, name, FALSE);
    IF file.res = FileSystem.done THEN
      Options.GetOption(opttext, optlength);
      WHILE optlength > 0 DO
```

```
        (* ... interprete program option ... *)
        Options.GetOption(opttext, optlength);
      END;
      continue := TRUE;
      EXIT;
    ELSE
      Terminal.WriteString(" ---- file not found");
      Terminal.WriteLn;
    END;
  END;
END; (* LOOP *)
```

11.8. V24

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;

  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

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.

11.9. Profile

Module *Profile* reads entries from the file *User.Profile*. This file is used to specify user dependent defaults for different programs.

Definition Module

```
DEFINITION MODULE Profile;

  PROCEDURE OpenProfile(title: ARRAY OF CHAR);

  PROCEDURE FindKey(key: ARRAY OF CHAR);

  PROCEDURE GetString(VAR s: ARRAY OF CHAR);

  PROCEDURE GetFileName(VAR name: ARRAY OF CHAR; ext: ARRAY OF CHAR);

  PROCEDURE GetNumber(VAR x: CARDINAL);

  PROCEDURE CloseProfile;
END Profile.
```

Explanations

OpenProfile(title)
: Opens the file *User.Profile* and searches for the title entry "title".

FindKey(key)
: Searches for 'key' starting from the position of the previously found 'title'. Position the file after the entry 'key' so that further calls of GetString, GetFileName and GetNumber return the values of 'key'.

GetString(s)
: Procedure GetString reads a string at the current position.

GetFileName(name, ext)
: Reads file name with default extension *ext*.

GetNumber(x)
: Reads a number.

CloseProfile
: Closes the file *User.Profile*.

Imported Module

FileSystem

11.10. String

This module provides a set of primitves for character string manipulations.

Definition Module

```
DEFINITION MODULE String;

  CONST
    first =      0B;
    last  = 177777B;

  PROCEDURE Length(VAR string: ARRAY OF CHAR): CARDINAL;

  PROCEDURE Occurs(VAR s: ARRAY OF CHAR; start: CARDINAL;
                   w: ARRAY OF CHAR): CARDINAL;

  PROCEDURE Insert(VAR s: ARRAY OF CHAR; at: CARDINAL; w: ARRAY OF CHAR);

  PROCEDURE Append(VAR s: ARRAY OF CHAR; w: ARRAY OF CHAR);

  PROCEDURE InsertCh(VAR s: ARRAY OF CHAR; at: CARDINAL; ch: CHAR);

  PROCEDURE AppendCh(VAR s: ARRAY OF CHAR; ch: CHAR);

  PROCEDURE Delete(VAR s: ARRAY OF CHAR; start, length: CARDINAL);

  PROCEDURE Copy(VAR s: ARRAY OF CHAR;
                 source: ARRAY OF CHAR; start, length: CARDINAL);

  PROCEDURE Assign(VAR s: ARRAY OF CHAR; source: ARRAY OF CHAR);

  PROCEDURE Same(VAR s: ARRAY OF CHAR; start, length: CARDINAL;
                 w: ARRAY OF CHAR): BOOLEAN;

  PROCEDURE Equal(s, w: ARRAY OF CHAR): BOOLEAN;

END String.
```

Explanations

Character strings are represented the same way as the compiler generates string constants:

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

Either all elements (characters) are used or the string is terminated by a (single) 0C character;

The first (leftmost) character has position 0; the last (rightmost) character has position length-1.

Length(string)
: Function *Length* returns the number of characters in *string* (without counting a terminating 0C).

Occurs(string, start, substring)
: Function *Occurs* checks if *substring* occurs as a substring of *string* starting at the character at position *start* or more to the right; The function returns the index of the first character, or 177777B if it does not occur.

Insert(string, at, substring)
: Procedure *Insert* inserts *substring* into *string* left of the character at position *at*. If the resulting *string* is to long to be stored in its memory available, it is clipped;

  If *at* > length(*string*) blanks are inserted in *string*.

  If *at* = last *substring* is appended to *string*.

Append(string, substring)
Insert(string, substring)

InsertCh(string, at, ch)
Procedure *InsertCh* inserts *ch* into *string* left of the character at position *at*.

If the resulting *string* is to long to be stored in its memory available, it is clipped;

If *at* > length(*string*) blanks are inserted.

If *at* = last *ch* is appended to *string*.

0C is considered as being no character.

AppendCh(string, ch)
InsertCh(string, last, ch)

Delete(string, start, length)
Procedure *Delete* removes the substring in *string* starting at position *start* with length *length*.

If *start* points to the right of the end, nothing happens.

Copy(string, source, start, length)
Procedure *Copy* copies a substring from *source* to *string*; the substring starts with the character at the position *start* (inclusive) and is *length* characters long.

If *start* and *length* denote non-existent characters (characters to the right of the last character) then the non-existent characters are ignored. If the resulting string does not fit into the memory for *string*, then it is clipped.

Assign(string, source)
Copy(string, source, first, last)

Same(string, start, length, substring)
Function *Same* denotes if the substring of *string* is equal to *substring*; the substring starts at position *start* (inclusive) and is *length* characters long.

If *start* and *length* denote non-existent characters (characters to the right of the last character) then the non-existent characters are not part of the substring used to compare.

All characters inside the string and the used substring are compared (including blanks).

Equal(s, w)
Same(s, first, last, w)

Imported Modules

none

12. Modula-2 on Ceres

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 *"system-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 Ceres specific features.

12.1. Implementation Details

12.1.1. Forward References

Due to the nature of a single-pass compiler an object has to be defined textually before it is referenced. In most cases this rule poses no problems. Only in the case of cyclic recursive procedures this rule cannot be satisfied. Therefore a forward declaration of procedures is introduced. This is done by simply appending the keyword *FORWARD* to a complete procedure header.

12.1.2. Type Transfer

Type transfer functions are not accepted by the standard compiler. (See VAL in 12.2.)
The types INTEGER and CARDINAL are assignment compatible with LONGINT, REAL with LONGREAL, and LONGREAL with REAL.

12.1.3. Procedure Parameters

Procedures can be used as parameters, or can be assigned, only if they are declared on the global level. This restriction also holds for Modula on Ceres. Furthermore, they must be declared in a definition module or in a foward declaration, or their heading must be followed by an asterisk.

```
PROCEDURE Assignable(parameters)*;  ...
```

This not very pleasing rule is necessary, because the NS processor uses different return instructions for external procedures and others, and they must correspond with the call instruction used. The compiler cannot determine the kind when generating a formal call. Hence we postulate the external mode for all formal and assigned procedures. Those defined in a definition module or defined in a forward declaration are automatically "external".

12.1.4. Code Procedures

Modula on Ceres also offers a code procedure declaration. In contrast to Lilith-Modula, however, it is used in definition modules only and serves to introduce procedures implemented by supervisor calls. The code number n specifies the identification inserted as a byte after the SVC instruction. Evidently, such definitions are provided with the operating system used. The format is

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

variables. INC and DEC accept a single parameter only.
The new, standard function LONG converts an argument of type INTEGER or REAL to the types LONGINT or LONGREAL, and the function SHORT performs the inverse transformation (in the case of integers without range check). The two additional standard functions FLOATD and TRUNCD are analogous to FLOAT and TRUNC; they yield results of types LONGREAL and LONGINT respectively.

12.1.6. Numeric constants

Constant values as well as variables are of a specific type. Character constants can be denoted by numbers appended by a capital C ('A' = 101C in the ASCII character set).
Constants of type INTEGER (or CARDINAL) can be denoted as decimal, hexadecimal or octal numbers (4096 = 1000H = 10000B). Those of type LONGINT or ADDRESS are denoted by decimal or hexdecimal numbers (4096D = 1000L). Constants of type LONGREAL are distinguished by the use of the letter D in place of E in the scale factor (or simply a suffix D if the scale factor is missing). Examples: 1.0D, 37.82D-7.

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*. 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.

Finally, there are the explicitly system-dependent features imported from module SYSTEM, to which we also count the declaration of code procedures. On Lilith, the module SYSTEM contains the types ADDRESS and WORD, and the procedures ADR, TSIZE, and LONG. On Ceres, the type ADDRESS is not compatible with CARDINAL, but rather with LONGINT for address arithmetic. The new type BYTE represents the unit of addressable storage. The type WORD is eliminated from the Standard Compiler, but retained in the Ceres-Medos Compiler for obvious reasons. Programmers should aim at its elimination.

On Ceres, the module SYSTEM contains a larger number of objects. This is a reflection of the fact that machine code cannot be defined by code procedures as on Lilith. The following definition is an attempt to summarize the facilities contained:

```
DEFINITION MODULE SYSTEM;
  (*FOR NS32032.  for details of instructions refer to manual*)

  TYPE ADDRESS;  (*compatible with type LONGINT and with all pointer types*)
  TYPE BYTE;
  TYPE WORD;     (*16 bit entity; in Medos Compiler only*)

  PROCEDURE ADR(VAR x: T): ADDRESS;  (*address of variable x*)
  PROCEDURE TSIZE(T): INTEGER;    (*size in bytes of variables of type T*)

  (*T subsequently denotes any type of size <= 4 bytes;
    T0 stands for either INTEGER or LONGINT*)

  PROCEDURE ASH(x: T; n: T0): T;  (* x * 2↑n *)
  PROCEDURE LSH(x: T; n: T0): T;  (* x shifted by n positions*)
  PROCEDURE ROT(x: T; n: T0): T;  (* x rotated by n positions*)
  PROCEDURE COM(x: T): T;         (*binary complement of x*)
  PROCEDURE FFS(VAR x: T; VAR n: INTEGER): BOOLEAN;
    (*assign to n the position of the first one bit of x with position >= n;
      FFS = "a one-bit was found" *)
  PROCEDURE GET(a: LONGINT; VAR x: T); (*assign value at address a to x*)
  PROCEDURE PUT(a: LONGINT; x: T); (*assign x to storage at address a*)
  PROCEDURE MOVE(VAR src, dest: ARRAY OF BYTE; count: T0);
  PROCEDURE VAL(T; x: T1): T;
END SYSTEM.
```

The procedures GET and PUT are used to access device registers. The absolute addressing mode, i.e.

variable declarations specifying an absolute address, is not available. All these facilities must be used only in few, low-level modules.

The generic function procedure VAL(T, x) is effectively a replacement for type transfer functions T(x). Its value is x, interpreted as type T. No code is generated for this "procedure". Its function is to make the uses of machine-dependent type transfers more explicit and more readily locatable.

12.3. Data Representation

1. Ceres uses byte addressing. However, data are transferred to and from memory in 32-bit words. Each type has an alignment factor k. Variables are aligned by the compiler to lie at an address a, such that a MOD k = 0. Since allocation is sequential, i.e. variables are allocated in the order of their textual occurrence, the least amount of storage gets waisted through alignment, if declarations are grouped according to size. The same holds for record fields, and in this case is even more important. The following are the sizes and alignment factors of types:

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

13. Disk Problems and Maintenance

13.1. Introduction

The programs *DiskCheck* and *DiskPatch* are used to maintain and repair errors of the file system on the fixed disk of Ceres. Both programs share the same base code, the user interface is, however, different. The program *DiskCheck* can be run by a casual user. It checks the structure of the disk and tries to fix automatically some simple errors. The program *DiskPatch* should be used by experienced users only. This program allows manual changes and initialization of a disk. In this way even harder problems can be fixed, on the other hand it is possible to destroy the entire disk information by a few key strokes.
The manual changes should be done only by users that are familiar with the structure of the disk system. It is not purpose of this description to discuss this topic, interested readers are invited to read [1] for the structure of the disk system and [2] for the details about its implementation on Ceres. A very simple description of some fields of the file descriptor is given in the description of DiskPatch (chapter 13.2.2).
Both programs provide procedures to recover from some simple error situations. It happens in three ways:

1) by reading and writing back a damaged sector. In this way sectors with recovered parity and partly with an unrecovered parity may be rescued. The sectors with other hardware problems are, however, definitely lost.
2) by computing a new probable value and setting it into the directory. This fixes errors like inconsistent version numbers, wrong file numbers, etc
3) by giving you hints what to do. An example of such a situation is the following: When one sector is allocated to two files, one of these file must be deleted. The user must decide which file should be deleted.

It is indicated in the following description of possible errors which method is used to fix the problem. The user should be aware that the program cannot repair all possible error situations, it tries with the best effort to fix them, but one cannot expect miracles.

Both programs are provided together with the operating system on the winchester disk of Ceres. However, when is disk is damaged seriously, i.e. the operating system does not start,, the program DiskPatch may be loaded using a special boot floppy.

13.2. DiskCheck

If some problems are encountered with a disk, the program *DiskCheck* should be run. This program checks the directory for consistency and on demand all sectors of the disk for hardware problems. Initially the program is set in a default state enabling to check the disk without a possibility to repair errors. This state may be changed such that the program tries some fixes.

If some problems are encountered, error messages are displayed. A long chain of error messages may be stopped by hitting a key and may be continued by hitting a key another time. ESC may be used to stop the checking procedure.

The program gives the following 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 a read or write operation. The program asks for confirmation. The sector will be written back and read again. If the read operation still produces an error, the program asks whether this sector should be linked into the file FS.BadPages.

- bad file number = nnnnn on sector mmmmm

the sector number of the data directory sector mmmm is destroyed. The program computes a correct number and replaces it.

- wrong page pointer, dir sector= nnnnn, pointer= mmmmm index=oooo
file name = yyyyy

a data directory pointer has an illegal value.

- double allocated page, page = nnnnn

file name = yyyyy file#: nnnn file name = zzzzz file#: mmmm
delete one of these files and immediately boot

After the program DiskCheck finished, delete either yyyyy or zzzzz and immediately bootstrap. If you forget to bootstrap, the same error will occur soon!!!!

- bad pointer: name->data directory, sector = nnnnn

pointer between name and data directory is destroyed. The program computes a correct pointer and replaces it.

- name dir points to a free block, name dir sect = nnnnn,
block dir sect = mmmmm, file name = yyyyy

the pointer from name to data directory points to an unused block.

- version # conflict, name dir sect = nnnnn, version = mmmmm,
data dir sect = ooooo, version = ppppp
file name = yyyyy

the version number of data directory is set equal to the version number of the name directory.

- illegal name dir entry kind, filenr = nnnn, kind = mmmm

The kind of the name directory sector has an illegal value.

If some fixes are made on the disk, the computer should be bootstraped and program DiskCheck should be run again to be sure that the disk is all right now.

13.3. DiskPatch

13.3.1. Introduction

The program DiskPatch allows you to initiate the winchester disk of Ceres 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. 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 hexadecimal 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 hexadecimal.

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 '.'

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.

H hex dump

The current sector (opened by *R* command) will be displayed in hexadecimal mode either byte- or word-wise.

I inspect

The last read sector may be inspected and changed in a hexadecimal representation either byte- or word-wise. 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. 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 bad sectors and their position if they are located within the fixed files. 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 16 file entries (0..F). Each entry consists of:

name - 24 characters; left adjusted, 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..F), 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.

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.

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 4 sectors. To compute the physical address make the following computation

physical.address := (page DIV 7) * 4 decimal /hexadecimal

A pointer to an unused page has a value D5B8 (hexadecimal).

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!!!!

+ hexadecimal calculator

This is a simple calculator able to add, subtract, multiply and divide two hexadecimal numbers. Type ESC to exit to the main menu.

13.3.3. The most Frequent Problems

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.

13.3.3.1. It is Impossible to Boot Ceres

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 (hexadecimal). The file name will be found on the position nnnn MOD 10 (hexadecimal). 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 divisible by 7 (hexadecimal). 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 D5B8 (hexadecimal). 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.

13.3.3.2. Consistency Problems

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 (hexadecimal), and set the *kind* of the file on the position nnnn MOD 10 (hexadecimal) 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.

13.4. References

[1] S.E. Knudsen: Medos-2: A Modula-2 Oriented Operating System for the Personal Computer Lilith, Ph.D. thesis No. 7346, ETH Zürich 1983

[2] F.Peschel, M.Wille: Porting Medos-2 onto the Ceres workstation, Institut für Informatik, to be published, ETH Zürich, Nov. 1986

Modula-2 for Ceres

N. Wirth 1. 1. 86 / 1. 2. 86

A Modula-2 compiler is now available for Ceres. The accepted language differs in some details from Modula for Lilith. This memo describes these differences, and it serves two purposes: First, it is intended for programmers who wish to port their software to Ceres. Second, it is a reminder for machine-independent programming and points out which points have to be observed, if new programs are to be easily transferrable either to Ceres or other machines. A summary of hints that should be of interest to all Modula programmers is given at the end.

New Data Types

The primary differences lie in the fact that Ceres is a 32-bit machine. This becomes manifest in the definition of some standard data types:

BITSET =	SET OF [0 .. 31]
LONGINT	integers in the range -2147483648 .. 2147483647
LONGREAL	a new type representing real numbers by 64 bits

All set and pointer types use 32 bits. These are extensions, and therefore do not require changes to existing programs, unless one wishes to profit from the new definitions. For example, routines operating on arrays of sets might be reprogrammed to save storage, since only half as many elements are needed, if effective use is made of their extended range.

Constants of type LONGINT are integers with a suffix letter D (e.g. 1986D). Constants of type LONGREAL are distinguished by the use of the letter D in place of E in the scale factor (or simply a suffix D if the scale factor is missing). Examples: 1.0D, 37.82D-7.

The new, standard function LONG converts an argument of type INTEGER or REAL to the types LONGINT or LONGREAL, and the function SHORT performs the inverse transformation (in the case of integers without range check). Also, the types INTEGER and CARDINAL are assignment compatible with LONGINT, REAL with LONGREAL, and LONGREAL with REAL. The two additional standard functions FLOATD and TRUNCD are analogous to FLOAT and TRUNC; they yield results of types LONGREAL and LONGINT respectively. Given the declarations below, the following correct assignments summarize these new facilities:

```
i: INTEGER; k: LONGINT; x: REAL; z: LONGREAL;

i := SHORT(k); i := TRUNC(x); i := TRUNC(z);
k := i; k := LONG(i); k := TRUNCD(x); k := TRUNCD(z);
x := z; x := SHORT(z); x := FLOAT(i); x := FLOAT(k);
z := x; z := LONG(x); z := FLOATD(i); z := FLOATD(k);
```

The Type CARDINAL

A more subtle change concerns the type CARDINAL. The NS32000 processor supports unsigned arithmetic, but multiplication and division are cumbersome, requiring double registers. We must also recognize that through the availability of a type LONGINT the primary justification for the type CARDINAL, namely to enlarge the range of positive integers to cover all address values, has vanished. The

second reason, namely to express that a variable assumes only natural numbers as values, has lost in attractivity, simply because the NS processor does not provide for convenient means to check against overflow. One is therefore tempted to abolish the type CARDINAL. However, our goal to make Lilith software easily portable to Ceres requires that Medos be available as an operating system. Medos makes heavy use of the type CARDINAL, and its elimination would require a very substantial rewrite of Medos. Our solution to this dilemma is to provide two compilers:

The Standard Compiler treats the type CARDINAL as the subrange [0 .. 32767] with base type INTEGER. The implementation offers the standard range check for assignment. The welcome benefit of this solution is that the nasty incompatibility of the types INTEGER and CARDINAL in expressions disappears. (The result type of the functions ORD and HIGH is now INTEGER, and so is the base type of subranges, even if the lower bound is not negative).

The Medos Compiler retains the type CARDINAL as on Lilith ([0 .. 65535]), but does not provide any checks against overflow or assignment of illegal, negative values of type INTEGER.

We strongly recommend to adapt programs to the Standard Compiler, unless compelling reasons exist against it, and in particular to design new programs without the use of the type CARDINAL. The Medos Compiler will not be distributed outside, and we hope that it can be eliminated after some time.

Type Conversions

So far, conversion seems to pose no severe problems. In fact, genuine difficulties appear only where machine-dependent features of Modula were used. They are supposedly highlighted by imports from module SYSTEM, an import that everybody knows should only be taken as a last resort. Programmers who have imported from SYSTEM too generously are now receiving the bill. Another, much less obvious and therefore easily abused machine-dependence is the type transfer function. I strongly recommend to abstain from using type transfer functions; in fact, they are not accepted by the Standard Compiler (see below). Particularly frequent cases of their use are the packing of characters to and the unpacking from a word file:

```
VAR n: CARDINAL;  ch0, ch1: CHAR;
n := 256*CARDINAL(ch0) + CARDINAL(ch1); WriteWord(wf, n)
```

On Ceres, the transfer function CARDINAL is inapplicable to values of type CHAR. The use of ORD saves the situation. It becomes more difficult, if we also wish to eliminate the type CARDINAL. Simply replacing it by INTEGER creates two pitfalls. First, overflow may occur in multiplication. Second, n is then not compatible with ORD, when using the Lilith or the Ceres-Medos compiler. The suggested solution is to use a shift function imported from SYSTEM, and to make the machine-dependence explicitly visible: n := LSH(ORD(ch0),8) + ORD(ch1).

```
ReadWord(wf, n); ch0 := CHAR(n DIV 256); ch1 := CHAR(n MOD 256)
```

In this case, the transfer function CHAR will not be applicable, because variables of types CARDINAL and CHAR require different amounts of storage, whereas on Lilith they both use 1 word. Replacing CHAR by CHR solves that problem. Note, however, that it is customary to store the "right byte" first on files on Ceres, since byte numbering in words proceeds from right to left, like bit numbering. Hence, the two assignments should be interchanged. Here again, the use of an explicitly system dependent function is recommended: ch1 := CHR(LSH(n,-8)); ch0 := CHR(n). A much better solution, however, is to refrain from word files alltogether, and to treat all files as byte files.

Integer Arithmetic

Here we point out a discrepancy between the DIV and MOD operators on Lilith and Ceres, when applying them to negative operands. In fact, DIV is wrong on Lilith (and probably most other computers) for negative arguments, as it represents the zero-symmetric integer division, generally expressed by /. Therefore, the unpacking cannot be accomplished on Lilith with DIV/MOD, if the arguments are of type INTEGER. On Ceres, this is possible, and DIV is implemeted by a right shift, whenever the divisor is a power of 2.

Lilith:	-10 DIV 3 = -3	Ceres:	-10 DIV 3 = -4
	-10 MOD 3 not allowed		-10 MOD 3 = 2

Module SYSTEM

Finally, there are the explicitly system-dependent features imported from module SYSTEM, to which we also count the declaration of code procedures. On Lilith, the module SYSTEM contains the types ADDRESS and WORD, and the procedures ADR, TSIZE, and LONG. On Ceres, the type ADDRESS is not compatible with CARDINAL, but rather with LONGINT for address arithmetic. The new type BYTE represents the unit of addressable storage. The type WORD is eliminated from the Standard Compiler, but retained in the Ceres-Medos Compiler for obvious reasons. Programmers should aim at its elimination.

On Ceres, the module SYSTEM contains a larger number of objects. This is a reflection of the fact that machine code cannot be defined by code procedures as on Lilith. The following definition is an attempt to summarize the facilities contained:

```
DEFINITION MODULE SYSTEM; (*NW 7.12.85*)
 (*FOR NS32032. for details of instructions refer to manual*)

 TYPE ADDRESS; (*compatible with type LONGINT and with all pointer types*)
 TYPE BYTE;
 TYPE WORD;    (*16 bit entity; in Medos Compiler only*)

 PROCEDURE ADR(VAR x: T): ADDRESS; (*address of variable x*)
 PROCEDURE TSIZE(T): INTEGER;   (*size in bytes of variables of type T*)

 (*T subsequently denotes any type of size <= 4 bytes;
   T0 stands for either INTEGER or LONGINT*)

 PROCEDURE ASH(x: T; n: T0): T;  (* x * 2↑n *)
 PROCEDURE LSH(x: T; n: T0): T;  (* x shifted by n positions*)
 PROCEDURE ROT(x: T; n: T0): T;  (* x rotated by n positions*)
 PROCEDURE COM(x: T): T;        (*binary complement of x*)
 PROCEDURE FFS(VAR x: T; VAR n: INTEGER): BOOLEAN;
  (*assign to n the position of the first one bit of x with position >= n;
    FFS = "a one-bit was found" *)
 PROCEDURE GET(a: LONGINT; VAR x: T); (*assign value at address a to x*)
 PROCEDURE PUT(a: LONGINT; x: T); (*assign x to storage at address a*)
 PROCEDURE MOVE(VAR src, dest: ARRAY OF BYTE; count: T0);
 PROCEDURE VAL(T; x: T1): T;
END SYSTEM.
```

The procedures GET and PUT are used to access device registers. The absolute addressing mode, i.e. variable declarations specifying an absolute address, is not available. All these facilities must be used only in few, low-level modules.

The generic function procedure VAL(T, x) is effectively a replacement for type transfer functions T(x). Its value is x, interpreted as type T. No code is generated for this "procedure". Its function is to make the uses of machine-dependent type transfers more explicit and more readily locatable.

Another source of potential problems is the inhomogeneous store on Lilith, manifest in the form of frames. Ceres offers a single, linear address space, and all programs making use of frames should be changed by eliminating frames.

Modula on Ceres also offers a code procedure declaration. In contrast to Lilith-Modula, however, it is used in definition modules only and serves to introduce procedures implemented by supervisor calls. The code number n specifies the identification inserted as a byte after the SVC instruction. Evidently, such definitions are provided with the operating system used. The format is

PROCEDURE P(parameter list) CODE n;

On Lilith, procedures can be used as parameters, or can be assigned, only if they are declared on the global level. This restriction also holds for Modula on Ceres. Furthermore, they must be declared in a definition module, or their heading must be followed by an asterisk.

PROCEDURE Assignable(parameters)*; ...

This not very pleasing rule is necessary, because the NS processor uses different return instructions for external procedures and others, and they must correspond with the call instruction used. The compiler cannot determine the kind when generating a formal call. Hence we postulate the external mode for all formal and assigned procedures. Those defined in a definition module are automatically "external".

Compiling Options

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

x	array index bound check	default = on
r	subrange assignment check	default = off
v	arithmetic overflow check	default = off

Example: SomeName.MOD/rv (all checks on)

Some programming hints

1. Ceres uses byte addressing. However, data are transferred to and from memory in 32-bit words. Each type has an alignment factor k. Variables are aligned by the compiler to lie at an address a, such that a MOD k = 0. Since allocation is sequential, i.e. variables are allocated in the order of their textual occurrence, the least amount of storage gets waisted through alignment, if declarations are grouped according to size. The same holds for record fields, and in this case is even more important. The following are the sizes and alignment factors of types:

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

2. The statements INC(n), DEC(n), INCL(s,n), EXCL(s,n) generate considerably denser code than their equivalents n := n+1, n := n-1, s := s + {n}, s := s - {n}, even in the case that n or s are simple variables. INC and DEC accept a single parameter only.

3. As on Lilith, access to so-called intermediate-level variables is slower and requires more code than access to local or global variables. Such accesses should therefore be made only after careful justification. Intermediate-level variables should be considered as implicit, additional procedure parameters.

4. Unlike Lilith, Ceres does not use indirect addressing for structured variables; all variables are allocated in sequence with ascending addresses. Frequently accessed variables should be placed at the beginning of the declaration list in order to obtain small addresses. This reduces the size of the code.

Judging from my own experience in porting the compiler, there are usually more hidden machine-dependent features in a program than one is likely to assume, even in one's own concoctions. The event of porting a program is a good occasion to become aware of them and to eliminate (at least some of) them. I recommend to first write a version eliminating CARDINALs, word files, and type transfer functions, still operating on Lilith. The constructs that make use of features in which Lilith-Modula and Ceres-Modula genuinely differ can then be tackled in a separate, second step.

The worst kind of machine-dependence, because it is so well hidden, is the use of an (untaged) variant record and its misuse by accessing a value as type T0 which was stored (to an overlaid field) as type T1. The only valid recommendation is to reprogram the algorithm.

Some General Hints for Programming in Modula

One of the main purposes of using a high-level language is to eliminate dependence on a particular implementation and computer. Even if only a single computer (type) is ever used, it is advisable to refrain from using machine-specific features of a language. The following are suggestions for programming on Lilith in general; they obtain additional relevance, if later on the use of Ceres is envisaged.

1. Refrain from using the type CARDINAL, unless use of values >= 32768 is relevant.
2. Refrain from the use of type transfer functions.
3. Refrain from the use of untagged variant records.
   Make sure that only fields of the variant indicated by the current tag field value are accessed.
4. Use byte (character) files rather than word files.
5. Make reference to modula SYSTEM only in carefully isolated places (modules).

Format of the compiler error data file ("err.DAT"):

H.R.Schär, 26.11.86

The error data file is a byte file generated by the Modula-2 compiler.
It contains the information of "err.LST" (error listing file) in machine
readable form to be interpreted by other programs (mainly the text editor
Sara with its "error" command).

```
ErrorFile     ::= [ListNameBlock]{CompUnit}.
CompUnit      ::= FileNameBlock {ErrorBlock}.
ListNameBlock ::= LNM {Char} NUL.
FileNameBlock ::= FNM {Char} NUL.
ErrorBlock    ::= ERR FilePos ErrorNumber.
FilePos       ::= LongInt.  (* >= 0 *)
ErrorNumber   ::= Integer.  (* >= 0 *)
NUL           ::= 0C.
FNM           ::= 300C.
ERR           ::= 301C.
LNM           ::= 302C.
Char          ::= byte.
Integer       ::= byte byte.  (* least significant byte first *)
LongInt       ::= byte byte byte byte. (* least significant byte first *)
```

The error text of each error number is looked up in the error listing file.
Its default name "DK.ErrList.DOK" may be replaced by a ListNameBlock.
Other applications (mainly other compilers) may therefore use their
own error listing file.

NS16032 Assembler

Benutzeranleitung

J.E. Wanner 15. Juni 1984

1. Aufruf des Assemblers

Der Assembler ist unter dem Namen 'asm16000' verfügbar. Nach dem Aufruf erscheint der Prompt für das Inputfile. Die Defaultextension für ein Source File ist '.ASM'. Es gibt die Möglichkeit die Option 'Nolist' anzugeben um ein Assembly ohne Listing durchzuführen. Andernfalls kreiert der Assembler ein Listingfile mit demselben Namen wie das Sourcefile und der Extension '.LST'. Des weiteren wird ein Codefile kreiert mit Extension '.REL'.

2. Assembler Syntax

Ein Assembler Programm besteht aus einer Sequenz von Statements, die je eine Zeile umfassen. Die Syntax für ein Statement sieht in EBNF folgendermassen aus :

```
statement = [ [label] opcode [operanden] [kommentar] |
            '|' [kommentar] ] 'eol'.
('eol' bezeichnet ein Zeilenende)
```

Ein Statement kann in völlig freiem Format eingegeben werden, d.h. die Position innerhalb einer Zeile ist irrelevant. Leerzeilen und Kommentarzeilen sind zulässig, allerdings müssen letztere mit einem vertikalen Strich '|' beginnen. Nach einem vollständigen Statement wird alles auf derselben Zeile als Kommentar interpretiert. Eine Ausnahme bilden die Pseudoinstruktionen BYTE, WORD und DOUBLE, bei denen allfälliger Kommentar auch mit dem vertikalen Strich beginnen muss.

2.1 Labels

Ein Label ist ein vom Benutzer definiertes Symbol, dem der momentane Wert des Programmzählers zugeordnet wird. Mit der Pseudoinstruktion 'EQU' kann man allerdings einem Label einen beliebigen Wert zuordnen.

2.2 Opcode

Der Opcode muss entweder eine gültige Maschinen- oder Pseudoinstruktion sein. Die Funktion jeder Maschineninstruktion ist im NS16000 Instruction Set Reference Manual beschrieben. Für eine Beschreibung der Pseudoinstruktionen siehe Abschnitt x.x.
Gross- und Kleinschreibung wird nicht unterschieden für Mnemonics und Assemblerdirektiven. Das heisst

```
movd    aPtr, R0
word    $ce00
```

ist äquivalent zu

```
MOVD    aPtr, R0
Word    $ce00
```

2.3 Operanden

Jede Maschinen- bzw. Pseudoinstruktion besitzt eine gewisse Anzahl Operanden, die nach dem betreffenden Mnemonic durch Kommata getrennt aufzuführen sind. Einzelne Pseudoinstruktionen besitzen eine variable Anzahl von Operanden (BYTE, WORD und

DOUBLE).

2.4 Kommentar

Als Kommentar wird alles interpretiert, das zwischen dem Ende einer gültigen Instruktion und dem nächstfolgenden Zeilenende steht. Ausserdem kann Kommentar auch mittels eines vertikalen Strichs '|' eingeleitet werden. Auch in diesem Falle gilt das Zeilenende als Kommentarende. Innerhalb des Kommentars sind beliebige Zeichen ausser natürlich einem Zeilenende zulässig.

3. Symbole und Ausdrücke

3.1 Symbole

Ein vom Benutzer definiertes Symbol bzw. Label kann aus einer beliebigen Anzahl von Zeichen bestehen. Es gelten folgende Regeln :

Gültige Zeichen sind 'A'-'Z', 'a'-'z', '0'-'9' und '.'. Jedes Symbol muss mit einem Buchstaben beginnen und nur die ersten 16 Zeichen sind relevant. Gross- und Kleinschreibung wird unterschieden, also 'LOOP' <> 'loop'.

Symbole können auf zwei verschiedene Arten definiert werden :

1. Als Label eines Statements. Das betreffende Symbol erhält den momentanen Wert des Programmzählers.
2. Mit der Pseudoinstruktion 'EQU'. Hier kann einem Symbol ein beliebiger Wert zugeordnet werden, z.B. können so Konstanten definiert werden.

3.2 Konstanten

Alle Konstanten werden als absolute Werte interpretiert. Konstanten sind zulässig, solange sie als 32 Bit Grösse darstellbar sind. Es können sowohl dezimale als auch hexadezimale Zahlen benutzt werden. Eine hexadezimale Zahl muss aber immer durch ein unmittelbar vorausgehendes Dollarzeichen gekennzeichnet werden (z.B. $fe00, $20ED). ASCII Zeichen können als Strings eingegeben werden, wobei diese in Apostroph-Zeichen eingeschlossen werden müssen. Es sind sowohl der Apostroph (') als auch das Gänsefüsschen (") zulässig.

3.3 Operatoren

Es gibt genau einen unären Operator, nämlich die Negation dargestellt durch das Minuszeichen. Binäre Operatoren sind :

```
+        Addition
-        Subtraktion
*        Multiplikation
/        Division
```

Alle binären Operatoren arbeiten mit 32-Bit Grössen, allerdings muss bei der Division der Divisor in 16 Bit darstellbar sein.

3.4 Terme

Terme sind Komponenten von Ausdrücken, sie können folgendermassen aussehen :

1. Eine Konstante, wie in 3.2 definiert worden ist.
2. Ein Symbol, wie in 3.1
3. Ein Ausdruck oder Term eingeschlossen in Klammern
4. Ein Minuszeichen gefolgt von einem Term (Negation, z.B. -a)

3.5 Ausdrücke

Ausdrücke sind Kombinationen von Termen, zusammengefügt durch binäre Operatoren. Jeder Ausdruck wird als 32 Bit Zahl interpretiert. Die Auswertung eines Ausdrucks geschieht mit der üblichen Operatorenpräzedenz (also Multiplikation und Division werden vor Addition und

Subtrakion ausgewertet.
Jeder Ausdruck ist nach der Auswertung entweder absolut oder relozierbar :

1. Ein Ausdruck ist absolut, wenn sein Wert eine fixe Grösse ist. Ein Ausdruck dessen Terme Konstanten, oder Symbole dessen Werte Konstanten sind, ist absolut. Ein relozierbarer Ausdruck subtrahiert von einem relozierbaren Ausdruck ist auch absolut.
2. Ein Ausdruck ist relozierbar, wenn sein Wert fix relativ zu einer Basisadresse ist. Alle Labels eines Programms sind relozierbare Terme und jeder Ausdruck, der solche enthält kann nur noch Konstanten addieren oder subtrahieren. Nehmen wir zum Beispiel an, dass 'loop' als Label in einem Programm definiert wurde. Dann gelten folgende Regeln :

```
loop          relozierbar
loop+5        relozierbar
loop-$20      relozierbar
20-loop       nicht relozierbar
loop-endloop  absolut, da sich die Offsets von loop und endloop
              gegenseitig aufheben
```

4. Instruktionen und Adressiermodi

4.1 Instruktionsmnemonics

Die Mnemonics sind im vorher schon erwähnten NS16000 Instruction Set Reference Manual beschrieben. Die meisten Instruktionen können auf verschieden lange Operanden angewendet werden, d.h. an den Mnemonic wird ein B, W oder D als Längenangabe angehängt.
Displacements in Branch Instruktionen oder bei PC relativer adressierung können 1, 2 oder 4 Bytes lang sein. Falls die referenzierte Adresse kleiner als die gegenwärtige ist, so wird einfach das kürzest mögliche Displacement generiert. Bei Vorwärtsreferenzen kann optional eine Länge für das Displacement angegeben werden, andernfalls wird ein 4-Byte Displacement generiert. Beispiel :

```
loop     movb  0(R0), 0(R2)
         addqd 1, R2
         acbb  -1, R1, loop   | hier wird ein 1-Byte Displacement
                              | generiert
         cmpqb 0, 0(R2)
         beq   endloop:B      | durch die Angabe :B oder :D kann
                              | ein 1- bzw. 2-Byte Displacement
                              | erzwungen werden
         sprb  US, R0
         sbitb 2, R0
         lprb  US, R0
endloop  ret   0
```

4.2 Adressiermodi

Die folgende Tabelle zeigt eine Übersicht über die Adressiermodi die vom Assembler erkannt werden. *displ1* und *disp2* bezeichnen Displacements, dies müssen absolute Ausdrücke sein. ***Rx*** bezeichnet ein CPU Register, ***Fx*** ein FPU Register, ***dreg*** ein 'dedicated register' in der CPU und *mreg* ein MMU Register.

Adressmodus	Notation	Beispiel
Register	Fx, Rx	MOVL F0, F4 MOVD R0, R2
Register Relative	disp1(Rx)	MOVB 1(R0), R3
Memory Relative	disp2(disp1(FP)) disp2(disp1(SB)) disp2(disp1(SP))	MULW R0, 10(2(FP))
Immediate	absoluter Ausdruck	MOVW $2000, R0
Absolute	@absoluter Ausdruck	JUMP @$fe00
External	EXT(disp1)+disp2	DIVD EXT(1)+20, 10(R0)
Top Of Stack	TOS	CMPQB 5, TOS
Memory Space	disp1(FP) disp1(SB) disp1(SP)	MOVQB 2, 2(FP)
	relozierbarer Ausdruck	COMB R0, data COMB R0, data:W
Scaled Index	BaseMode[Rx:B] BaseMode[Rx:W] BaseMode[Rx:D] BaseMode[Rx:Q]	ROTW 2, 10(R0)[R1:W]

Bei der Adressierart 'Memory Space' wird, falls einfach ein relozierbarer Ausdruck erkannt wird, relativ zum Programmzähler adressiert. Der 'BaseMode' in 'Scaled Index' ist ein beliebiger Adressiermodus ausser 'Scaled Index'.

5. Assemblerdirektiven (Pseudoinstruktionen)

5.1 BLOCK

Mit der Instruktion BLOCK kann Platz für Daten reserviert werden. Wie bei den meisten normalen Mnemonics muss unmittelbar nach der Direktive eine Längenangabe folgen, d.h. die folgenden Versionen sind gültig : BLOCKB, BLOCKW und BLOCKD. Als Operand wird ein absoluter Ausdruck erwartet. Beispiele :

```
JumpTable  BLOCKD  20              | Dies reserviert 20 Doppelworte
                                   | also 80 Bytes
           BLOCKB  2*tableLength
```

5.2 BYTE, WORD, DOUBLE

Mit diesen Instruktionen können Datenfelder initialisiert werden. Zugleich wird der dazu benötigte Platz reserviert. Alle drei Instruktionen akzeptieren eine beliebige Anzahl Operanden und jeder wird in der angegebenen Grösse abgespeichert. Eine Ausnahme bilden Stringkonstanten, die als aufeinanderfolgende Bytes abgelegt werden. Die Operanden dürfen auch Vorwärtsreferenzen enthalten (z.B. um Sprungtabellen für die CASE Instruktion zu definieren). Beispiele :

```
message   BYTE     "Fehler in Adressiermodus", $0d, 0
```

```
TabPtr    DOUBLE    | so wird einfach ein Doppelwort reserviert
          WORD      $fff8, 23, $100

s1        CASEB   casetab[R2:B]
casetab   BYTE    case1 - s1
          BYTE    case2 - s1
case1     MOVF    F0, 1(SP)
          BR      endcase
case2     MOVF    F1, 1(SP)
endcase
```

5.3 EQU

Die Instruktion weist einem Symbol einen beliebigen Wert zu. Falls der Ausdruck der zugewiesen wird relozierbar ist, so hat auch dieses Symbol einen relozierbaren Wert. Der Ausdruck darf keine Vorwärtsreferenzen enthalten. Beispiele :

```
EntrySize       EQU   10
NumberOfEntries EQU   100
TableLength     EQU   EntrySize * NumberOfEntries
```

The Ceres-Assembler

Frank Peschel
1.3.1987
Institut für Informatik
ETH Zürich

This paper describes the use of the assembler on Ceres. It produces code for the NS 32000 Series CPU, which is the heart of this workstation. It can handle two different "kinds" of programs

- ordinary assembler programs which may be executed on every NS 32000 based system
- assembler modules that can be used as if they were written in Modula 2. It is recommended to use only use if it is really necessary. It should be only used by programmers which are very familiar with the Ceres workstation and the Modula-2 run-time organization.

The assembler produces, on user demand, a listing and a cross-reference of all defined symbols. The first version of the assembler was written by Jürg Wanner as cross-assembler on Lilith in his diploma work.

1. Assemblation of a program or a module

The assembler is called by typing *asm32*. After displaying the string "in>", the assembler is ready to accept the filename of the program to be compiled.

Default medium is **DK** and default extension is **ASM**.

The assembler produces a relocateable code-file (extension REL) for assembler programs or an object-code file (extension OBN). The program listing and/or the cross-reference is written onto the list-file (extension LST).

2. Program Options for the Assembler

/n

The assembler does not generate a listing file..

/x

The assembler generates a listing file that includes a cross-reference of all user defined symbols.

3. Syntax of an Assembler Program

An assembler program consists of a statement sequence. The syntax of a statement is given below. Each statement has to be defined on one line. The rest of the line (behind a valid statement) is interpreted as comment. Empty and comment lines are allowed, the latter have to start with a vertical bar ('|').

```
statement= [[label] opcode [operands][comment] | '|' [comment] 'eol'.
operands = operand {',' operand}.
```

3.1. Labels

A label is a user defined symbol that represents a constant or a memory location (data or code). Each label has a specific type. The assembler distinguishes the types absolut, program-counter relative (PC-relative), static-base relative (SB-relative) and external. Each

identifier in front of a valid mnemonic is interpreted as a label and is included into the internal symbol-table.

3.2. Opcodes

Each opcode is a processor instruction or a so-called pseudo instruction (assembler directive). The semantics of the processor instructions is described in the Instruction Set Reference Manual [1]. A description of the pseudo instructions will follow in paragraph 6. Mnemonics and assembler directives are not case sensitive, i.e.

```
movd  aPtr,R0  is equivalent to MOVD  aPtr,R0
word  $ce00    is equivalent to Word  $ce00
```

3.3. Operands

Every instruction has a specific number of operands. The operands are separated by a colon. Some pseudo instructions may have a variable number of operands (see paragraph 6.).

3.4. Comments

The starting symbol of a comment is the vertical bar character. The rest of this line is interpreted as comment. There is no explicit closing symbol. Everything between the end of a valid instruction and the following end-of-line is interpreted as comment.

4. Symbols and Expressions

4.1. Symbol

Each user defined symbol consists of an arbitrary number of characters with respect to the following rules:

The set of valid characters consists of: 'A' - 'Z', 'a' - 'z', '0' - '9' and '.'.
The first character must be a letter.
Only the first 16 characters are relevant.
Symbols are case sensitive, i.e. 'loop' <> 'LOOP'.

4.2. Constant

Constants are absolute values. Constant numbers must be representable in 32 bits. Constants can be defined in decimal as well as in hexadecimal form. Hexadecimal numbers are identified by a preceeding dollar character (eg $fe00). String constants are enclosed in quote marks (quote or double quotes). The opening and closing marks must be the same character.

4.3. Operator

There are four operators defined:

'+'	addition	'-'	subtraction
'*'	multiplication	'/'	division

Each operator apply to two operands. '-' with a single operand only denotes sign inversion. All operands are internally represented as 32bit values.

4.4. Expression

```
Term       = Symbol | Constant | "(" Expressions ")" | "-" Term.
Expression= Term {Operator Term}.
```

The evaluation of an expression is done according to the usual operator precedence. The assembler distinguishes three types of expressions:

- absolute
  Constants are absolute expressions
- PC-relative
  All labels, except those defined in the static base segment and those defined by EQU, are PC-relative expressions.
- SB-relative
  All labels inside the static base segment, except those defined by EQU, are SB-relative expressions.

In the following section relocatable expression denotes both, PC-relative as well as SB-relative, expressions. Labels defined by EQU get their type from the operand of their definition. The following rules apply to the evaluation of expressions:

- multiplication and division is restricted to operands of the type absolute
- relocatable operands of one expression must be of the same type
- sum of a relocatable and an absolute Term is of type relocatable
- a reclocatable Term subtracted by an absolute Term gives a relocatable Expression
- an absolute Term subtracted by a reclocatable Term gives an absolute Expression
- addition or subtraction of two relocatable Terms gives an absolute Expression

5. Instructions and Addressing Modes

5.1. Mnemonics

As mentioned above all mnemonics are described in the Instruction Set Reference Manual. Most of the instructions apply to different types of operands. Those mnemonics are suffixed be the characters 'B', 'W', 'D', 'F' or 'L'.

Displacements may have the length of 1, 2 or 4 bytes. Forward references (e.g. forward jumps) are stored in 4 bytes unless the programmer forces the assembler to use a smaller space. This is done by appending ':B' for 1 byte and ':W' for 2 byte displacements.

Example:

```
loop      MOVB   0(R0),0(R2)
          ADDQ   D1,R2
          ACBB   -1,R1,loop   | generates 1 byte displacement
          BEQ    endloop:B    | forces 1 byte displacement
          ...
endloop
```

5.2. Addressing Modes

The following table shows the addressing modes known by the assembler. disp1 and disp2 denote displacements. Displacements are always absolute expressions. Rx denotes a CPU-register and Fx a FPU-register.

Adressmode	Notation	Example
Register	Fx, Rx	MOVL F0, F4 MOVD R0, R2
Register Relative	disp1(Rx)	MOVB 1(R0), R3
Memory Relative	disp2(disp1(FP)) disp2(disp1(SB)) disp2(disp1(SP))	MULW R0, 10(2(FP))

```
                    disp2(SB-rel.Ausdr.)     LSHW  5, 3(dPointer)

Immediate           absolute expression      MOVW  $2000, R0

Absolute            @absolute expression     JUMP  @$fe00

External            EXT(disp1)+disp2         DIVD  EXT(1)+20, 10(R0)

Top Of Stack        TOS                      CMPQB 5, TOS

Memory Space        disp1(FP)                MOVQB 2, 2(FP)
                    disp1(SB)
                    disp1(SP)
                    rel. expression          COMB  R0, data
                                             COMB  R0, data:W

Scaled Index        BaseMode[Rx:B]           ROTW  2, 10(R0)[R1:W]
                    BaseMode[Rx:W]
                    BaseMode[Rx:D]
                    BaseMode[Rx:Q]
```

In addressing mode "Memory Space" the assembler choose the base register according to the type of the relocatable expression. "Basemode" in "scaled index" denotes any mode with the exception of "scaled index" and "immdiate".

6. Pseudo Instructions

Pseudo Instructions are used to reserve memory space for variables, assign values to user defined identifiers and to force the assembler to give some support for the "realization" of implementation modules by an assembler program.

6.1. BLOCK

This instruction reserves space for data. This instruction has a length attribute (restricted to 'B', 'W', 'D'). It has one operand of type absolute.

Example:

```
JumpTable BLOCKD 20             | 80 bytes
          BLOCKB 2*tableLength  | 2*tableLength bytes
```

6.2. BYTE, WORD, DOUBLE

These instructions reserves memory space and initializes it to given values. All three instructions accepts operand lists. Each operand is stored in the size given by the instruction. This do not hold for strings, which are always stored as sequence of bytes. The operands may contain forward references (e.g. to produce jump tables for the CASE instruction).

Examples:

```
message   BYTE     "invalid addressing mode", $0d, 0
TabPtr    DOUBLE   | reserves 4 bytes
          WORD     $fff8, 23, $100

s1        CASEB  casetab:B[R2:B]
casetab   BYTE   case1 - s1
          BYTE   case2 - s1
case1     MOVF   F0, 1(SP)
```

```
            BR      endcase:B
case2       MOVF    F1, 1(SP)
endcase
```

6.3. EQU

This instruction assigns a value and its type to a user given identifier. The right hand side expression must not contain forward references.

Example:

```
EntrySize        EQU   10
NumberOfEntries  EQU   100
TableLength      EQU   EntrySize * NumberOfEntries
```

6.4. STABEG, STATEND

Labels defined by BLOCK, BYTE, WORD, DOUBLE between the STATBEG and STATEND statements denote variables, i.e. the labels have the type SB-relative. It is allowed to have more than one pair of these brackets. All labels outside these brackets are PC-relative.

6.5. Assembler Modules

As already mentioned above the assembler can produce object files, i.e. the realization of an implementation module can be done in assembly language. This sometimes useful, but only necessary in very rare cases, for modules on a very low level.

6.5.1. Identification of assembler modules

Assembler modules have to start with the keywords MODULE or IMPLEMENTATION MODULE appended by the module name. If the module is an implementation module, the assembler scans the module's symbol file. All exported identifiers can be referenced in this module. Offsets of exported procedures are stored in the entry table. If the module defines a record type in its export list, a field identifier can be used as displacement, but this identifier is only known in qualified mode.

6.5.2. INIT, ENDINIT

INIT labels the body of the pseudo module. The assembler produces code for the normal Modula-2 initialization and stores the program offset of the body into the first entry of the module's entry-table. ENDINIT produces the correct return instruction.

7. Error messages

- missing right paranthesis
- error in factor (unexpected symbol)
- illegal displacement length modifier
- illegal scale option in scaled index mode
- missing right bracket
- colon expected
- error in external addressing
- illegal symbol in memory relative addressing
- 'until' and 'while' options are mutually exclusive
- missing left bracket
- only 4-bit constant ins short value
- comma expected
- symbol defined twice
- ident or opcode expected, line ignored
- opcode expected, line ignored
- constant too long
- illegal address mode
- illegal forward reference
- undefined symbol : 'xxxxxx'
- relocatable expression illegal for displacements
- displacement expected in fixup
- fixup failed (displacement requires more space)
- basemode in scaled index must not be immediate
- integer overflow
- string exceeds end of line
- static data can't be initialized
- regular mnemonic encountered, end of static segment
- end of file encountered, end of static segment
- incompatible types of operands in expression
- multiplication or division not allowed with relocatable operands
- illegal module name
- module name expected
- name conflict in import export list
- error in import list
- ident not exported
- ident expected
- implementation module construct
- undefined exported procedure
- not expected pseudo instruction
- strings are restricted to BYTE
- inconsistent symbol file (no of procedures)

Reference

[1] NS3000 Series Instruction Set Manual
National Semiconductor, November 1984

Lara 2.0 on Ceres

H.R. Schär, 1.3.87

Version 2.0 of the document editor *Lara* is also running on Ceres. Use the "Editor"-floppy of the Software Ditribution Box (I.Noack,RZ H2) to get the necessary files.

Differences to the Lilith version:

1) The functional extensions:
   - Commands to send and receive electronic mail
   - Spelling checker

   are not yet available.
2) Remote printing in *laraprint* is not yet available.
3) The font data files with extension WID (and PSW) are replaced by MD2.FNT (and MDP.FNT) files. The description files for generic symbols in formulas have therefore the new extensions SCR.FNTD, MD2.FNTD and MDP.FNTD.
4) The file FormulaKeys.KMC contains some function key definitions to enter formulas.
5) *Lara* accepts also pressing a mouse button to confirm commands as in the editor *Sara*.

Lara 2.0

H.R. Schär, 1.2.87

A new version of the document editor *Lara* is available on the file server. Use the commandfile *#ma.ED.GetLara2.COM* to get the necessary files.

Changes:

1) The interactive editor Lara is available in different variants. Each variant consists of a common Lara-kernel and a (possibly empty) set of functional extensions.

   Functional extensions:

   - Mathematical formulas (see the next chapter for a description).
   - Commands to send and receive electronic mail (see *PUB.LaraMail.DOK*).
   - Spelling checker (see *PUB.LaraSpell.DOK*).

   Each variant is identified by a letter appended to its program name ("f"=formula, "m"=mail, "s"=spelling checker, "n"=none). For example, the program *SYS.larafm.OBJ* denotes a variant with mathematical formulas and mail commands.
   I recommend copying the most frequently used variant to *SYS.lara.OBJ*. Some variants may be too big to run on the "small" Lilith machines (with memory size 128 kWords). The smallest variant *SYS.laran.OBJ* should cause no problems.

2) The file *Lara.UserSection* contains modified standard user profile entries. The font SYNTAX12 is now used as the default font. If only fonts of the families SYNTAX and BARBEDOR are used, the entry 'Resolution' (header entry "Screen") should be set to 80 to get a ratio of 1:3 (instead of 1:2.4) on 240dpi printers.
   The default values for (unformatted) ASCII-files have been changed to improve printer output for program listings.

3) The print program *laraprint* has an improved page breaking algorithm to avoid single lines at the end or at the beginning of a page. For each paragraph at least two lines are left at the end of the current page or at the beginning of the next page. A paragraph with one line is recognized as title belonging to the next paragraph, if the following line is indented or has some defined difference in its font.

4) *laraprint* may also be used to generate a local output file. Press "f" instead <RETURN> when asked to select the name of the printer.
   PostScript format for the Apple LaserWriter is also supported. *Lara.UserSection* contains the relevant entries required for your user profile.

Mathematical Formulas in Lara

Lara provides an easy way to generate and modify mathematical formulas. The following short description shows the major concepts and illustrates the usage of formulas in documents. For a more detailed description, see [1].

Definition of a formula in Lara

To integrate formulas into normal text, the definition of a paragraph as a sequence of characters has been extended in the following way:

- A paragraph is a sequence of cells.
- A cell contains a character or a formula.
- A formula is a fixed sequence of operands with an optional generic symbol.
- Generic symbols are: Integral symbol, summation symbol, parenthesis, etc.
- An operand is a sequence of cells.

The definition of formulas is recursive, i.e. a formula may contain other formulas.

Example: $\sum_{i=1}^{n-1} \frac{2i+1}{i+n}$

If your Lara does not include mathematical formulas, a □ symbol instead of a summation formula is displayed.
The summation formula has three operands: "i=1" (lower bound), "n-1" (upper bound) and $\frac{2i+1}{i+n}$ (sum content).

The user cannot define new kinds of formulas. This would contradict the basic Lara concept that documents are self-contained.

How to select formulas

Formulas are selected in the same way as ordinary characters. Repeated selection of the same character first extends the selection to the formula operand, then to the whole formula. This way of extending a selection is repeated until the selected formula is no more part of another operand. After that the whole paragraph is selected.

Dragging a selection inside a formula is restricted by the end of the current operand.

Selected formulas can be copied or deleted with the normal text commands *copy* and *delete*.

How to insert text into formulas

The insertion point (caret) can be positioned inside a formula. Clicking the left mouse button on the same character again positions the caret at the beginning of the operand, then at the beginning of the formula. This way of positioning can be repeated until the caret is no longer inside a formula. Then the beginning of the paragraph is taken as caret position.
If the caret is adjacent to a formula, a dotted frame surrounding the formula appears. It helps to recognize different caret positions with nearly identical locations on the screen.

Example:

1) $\int_0^1 f(x)dx\,\Delta$ 2) $\int_0^1 f(x)dx\,\Delta$

In the first case the caret is at the end of the third operand of the integral. In the second case it is at the end of the integral.

Keyboard, Special Keys

The following keys are interpreted differently inside a formula.

<RETURN>	return key: skip operand.
<Ctrl-J>,<LINE FEED>	end of paragraph command: skip formula(s), then enter <Ctrl-J>.
<Ctrl-L>	end of chapter command: skip formula(s), then enter <Ctrl-F>.
DEL	backspace key: ignored after a formula or before an operand.

Initial entry of formulas

Formulas can be entered either by copying them from other documents or by using "macros". A macro is a "\" followed by a defined name. As soon as a macro is typed in (terminated with " " or <RETURN>), it expands to a corresponding text sequence representing a formula.

Macros are interpreted by the separate program *KeyboardMacros*, i.e. you have to start *KeyboardMacros* before starting *lara*.

If you want to enter a "\" (or "ü" or "ù") in a document, press "\" twice.

The file *Formulas.DOK* containes all defined formulas and their correspondig macro names.

Example:

The formula $\int_0^1 \frac{dx}{\sqrt[3]{1+x^2}}$ is entered by the following keyboard input:

	Display	Keyboard Entry		Display	Keyboard Entry
1)	Δ	\int<RETURN>	2)	$\int_{\Delta}$	0<RETURN>
3)	$\int_0^{\Delta}$	1<RETURN>	4)	$\int_0^1 \Delta$	\frac<RETURN>
5)	$\int_0^1 \frac{\Delta}{}$	dx<RETURN>	6)	$\int_0^1 \frac{dx}{\Delta}$	\root<RETURN>
7)	$\int_0^1 \frac{dx}{\Delta}$	3<RETURN>	8)	$\int_0^1 \frac{dx}{\sqrt[3]{\Delta}}$	1 +

9) $\int_0^1 \frac{dx}{\sqrt[3]{1+\Delta}}$ \sup<RETURN>

10) $\int_0^1 \frac{dx}{\sqrt[3]{1+\Delta}}$ x<RETURN>

11) $\int_0^1 \frac{dx}{\sqrt[3]{1+x^{\Delta}}}$ 2<RETURN>

12) $\int_0^1 \frac{dx}{\sqrt[3]{1+x^{2}_{\Delta}}}$ <RETURN>

13) $\int_0^1 \frac{dx}{\sqrt[3]{1+x^{2}}_{\Delta}}$ <RETURN>

14) $\int_0^1 \frac{dx}{\sqrt[3]{1+x^{2}}}_{\Delta}$ <RETURN>

15) $\int_0^1 \frac{dx}{\sqrt[3]{1+x^{2}}}_{\Delta}$

This example shows that a formula is entered (without repositioning the caret) top-down, i.e. from the outside to the inner parts.

It is very important to always leave a formula before continuing with ordinary text.

Generic symbols

Generic symbols like $\int$ or $\sum$ as parts of formulas are automatically generated (in their required size) by Lara. They are usually characters taken from a special font. Currently the two fonts *GENSYMA* and *GENSYMB* (and *Symbol* for *PostScript* output) are used.
Character codes are assigned (installed) to generic symbols by a textual entry in a description file. The currently used decription files are *GENSYM.SCFD* (Screen), *GENSYM.WIDD* (print file) and *GENSYM.PSWD* (*PostScript*).
If a symbol is not installed by an entry in a description file, Lara draws a simple symbol using a fixed sequence of straight lines.

Some formulas without generic symbol

There are some formulas without explicit generic symbol. They only define how some operands have to be grouped together.

1) Matrix

A matrix is a sequence of operands arranged in a rectangular form, e.g.: $\begin{matrix} 3 & 4 \\ 7 & 10 \end{matrix}$ (\mat)

Usually a matrix is used inside parenthesis: $\begin{pmatrix} 3 & 4 \\ 7 & 10 \end{pmatrix}$, $\begin{vmatrix} 3 & 4 \\ 7 & 10 \end{vmatrix}$, $\begin{bmatrix} 3 & 4 \\ 7 & 10 \end{bmatrix}$
Enter first the parenthesis formula, then the matrix formula.

2) Sub- and Superscription

The general case has a base operand and six circularly arranged operands: $\overset{c}{d}{}^{b}_{e}F^{}_{f}{}_{a}$ (\circum)

Special cases are: a_i (\sub), e^x (\sup), $T_{i,j}^{k}$ (\subsup), $\ddot{X}$ (\upper), $\lim_{n\to\infty}$ (\lower)

3) Elementary formula

The elementary formula has only one operand, e.g.: $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ (\fma)

References

[1] Schär, H.R. Die Integration mathematischer Formeln in den Dokumenteneditor Lara. Informationstechnik (it), Oldenbourg Verlag München, Dezember 1986.

NrOp	Nr	Type	Symb	Macro	Comment
any	0	any	□	\anyfma	unknown formula (e.g. extended file format)
any	1	any	⊞	\errfma	error symbol (NrOp or Type wrong)
1	2	1	(F	\lpar	left parenthesis
1	2	2	F)	\rpar	right parenthesis
1	2	3	(F)	\pars	parentheses
1	3	1	[F	\lbrack	left bracket
1	3	2	F]	\rbrack	right bracket
1	3	3	[F]	\bracks	brackets
1	4	1	{F	\lbrace	left brace
1	4	2	F}	\rbrace	right brace
1	4	3	{F}	\braces	braces
1	5	1	|F	\lvert	left vertical line
1	5	2	F|	\rvert	right vertical line
1	5	3	|F|	\verts	vertical lines
1	5	6	‖F‖	\Verts	double vertical lines
1	6	1	⟨F	\langle	left angle
1	6	2	F⟩	\rangle	right angle
1	6	3	⟨F⟩	\angles	angles
1	9	1	F	\fma	formula
1	9	2	F	\frame	frame
1	9	3	F	\dframe	dotted frame
1	10	1	$\hat{F}$	\hat	hat
1	11	1	$\tilde{F}$	\tilde	tilde
1	12	1	$\overline{F}$	\ol	overline
1	13	1	$\vec{F}$	\vec	vector
2	17	1	$\sqrt[F]{G}$	\root, \sqrt	root
2	19	1	$\frac{F}{G}$	\frac, \over	fraction
2	20	1	$\cfrac{F\|}{\|G}$	\cfrac	continued fraction
2	21	1	$\xrightarrow[G]{F}$	\rarrow	right arrow
2	22	1	$\xmapsto[G]{F}$	\mapsto	maps to
2	23	1	$\underbrace{F}_{G}$	\ubrace	underbrace

2	23	2	$\overbrace{F}^{G}$	\obrace	overbrace
3	25	1	$\prod_{a}^{b} F$	\prod	product
3	26	1	$\sum_{a}^{b} F$	\sum	sum
3	27	1	$\bigcap_{a}^{b} F$	\cap	cap (intersection)
3	28	1	$\bigcup_{a}^{b} F$	\cup	cup (union)
3	29	1	$\forall_{a}^{b} F$	\forall	for all
3	30	1	$\exists_{a}^{b} F$	\exists	exists
3	31	1	$\mathbb{N}_{a}^{b} F$	\number	number
3	35	4	$\int_{a}^{b} F$	\int	integral
3	35	8	$\iint_{a}^{b} F$	\int2	double integral
3	35	12	$\iiint_{a}^{b} F$	\int3	triple integral
3	35	5	$\oint_{a}^{b} F$	\oint	contour integral
3	35	6	$\fint_{a}^{b} F$	\pvint	principle value integral
2	40	1	F_{i}	\sub	subscripted
2	40	2	F^{i}	\sup	superscripted

3	40	3	F_i^j	\subsup	sub- and superscripted
2	40	4	$\overset{i}{F}$	\upper	upper
2	40	32	F_i	\lower	lower
7	40	63	$d^{cb}_{e\,f\,F\,a}$	\circum	circumscripted
n*m	50	m	a b c d	\matu22	matrix (uniform spacing, n rows, m columns)
n*m	51	m	a b c d	\mat22	matrix (spacing for each row, column)
n*m	52	m	a b c d	\mati22	matrix (individual spacing for each element)

Some mathematical formulas

p.

42 $$\int_0^1 \frac{dx}{\sqrt[3]{1+x^2}}$$

45 $$\lim_{n\to\infty}\left(1+\frac{x}{n}\right)^n = e^x \qquad z_{1.2} = \frac{-b+\sqrt[2]{b^2-4ac}}{2a}$$

47 $$\|A\| = \sqrt{\sum_{i,j=1}^{n} a_{ij}^2}$$

57 $$F = \frac{p+q}{2}\sqrt{pq} \qquad F = \frac{p+q}{2}\sqrt{pq}$$

59 $$f'(x) = \frac{1}{2}\sqrt{2x} + \frac{2+x}{2}\frac{1}{\sqrt{2x}} f'(x) = \frac{1}{2}\sqrt{2x} + \frac{2+x}{2}\frac{1}{\sqrt{2x}}$$

71 $$k > \frac{\log\left(\frac{\varepsilon}{|e_0|}\right)}{\log|F'(s)|}$$

72 a) $1 + \cfrac{1}{\sqrt{1 + \cfrac{1}{\sqrt{1+\ldots}}}}$ b) $\sqrt[3]{1 + \sqrt[3]{3\sqrt{1 + \ldots}}}$

74 $$\lim_{k\to\infty} \frac{e_{k+1}}{e_k^2} = \frac{F''(s)}{2} \quad e_{k+1} \sim \frac{F''(s)}{2} e_k^2$$

81 $$\frac{f'(x_k)^3}{f''(x_k)^2} = -\frac{x_k + b}{2} \Rightarrow a = -\frac{4f'(x_k)^3}{f''(x_k)^2} \quad x_{k+1} = x_k - 2\frac{f'(x_k)}{f''(x_k)}\left(\frac{1}{1 - \frac{1}{2}\frac{f(x_k)f''(x_k)}{f'(x_k)^2}}\right)$$

82 $$|g(x)| = \exp\left(\int \frac{dx}{x - F(x)}\right) \qquad \int \frac{dx}{x - F(x)} = \int \frac{1 + e^x}{xe^x - 1} dx$$

86 $$x = F(x) = -\frac{f(x)}{f'(x)} G\left(\frac{f(x)f'(x)}{f'(x)^2}\right) \qquad x = F(x) = -\frac{f(x)}{f'(x)} G\left(\frac{f(x)f'(x)}{f'(x)^2}\right)$$

98 $$P_n(x) = \sum_{k=0}^{n} P_{n-k}(x-z)^k \qquad P_n(x) = \sum_{k=0}^{n} \frac{P_n^{(k)}(z)}{k!}(x-z)^k$$

101 $$r = 2 \max_{1 \le k \le n} \sqrt[k]{\left|\frac{a_{n-k}}{a_n}\right|}$$

142

x	1	2	3	4
y	5	7	5	1

151 $$f(x) = \int_0^x e^{\sin t} dt$$

161 $$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{z^2}{2}} dz$$

163 $$T_{ij} = \frac{h_i T_{i-1,j-1} - h_{i-j} T_{i,j-1}}{h_i - h_{i-j}}$$

166 $$f(x) = x^2 \ln\left(\frac{\sqrt{x^3 + 1}\, e^x (x^3 + \sin x^2 + 1)}{2(\sin x + \cos^2 x + 3) + \ln x} \right)$$

194 $$\sum_{i=0}^{n-1} \left(\int_{x_i}^{x_i+1} f(x)dx - T_i(h) \right) = -\frac{h^3}{12} \sum_{i=0}^{n-1} f''(\xi_i)$$

199 $$\left| \int_a^b f(x)dx - S(h) \right| = \frac{(b-a)h^4}{180} \left| f^{(4)}(\xi) \right|$$

202 $$\frac{x}{e^x - 1} = \sum_{k=0}^{\infty} \frac{B_k}{k!} x^k$$

209 $$\text{abs(i2)} < \eta \left| \int_a^b f(x)dx \right| \qquad S(h) = \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right) f(\frac{a+b}{2})$$

213 $$\int_1^{\infty} \frac{1}{x} e^{-\frac{x}{2}} dx$$

216 $$x''(t) = -\frac{cx(t)}{\left(\sqrt{x(t)^2 + y(t)^2}\right)^3}$$

232 $$h_{neu} < h \sqrt[5]{\frac{\varepsilon \left| y_k^* \right|}{\left| y_k - y_k^* \right|}}$$

Some formulas from the TEX Book

p.

129 $$\left(\left(x^2\right)^3\right)^4$$

142 $$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4}}}}$$

180 $$\prod_{j>0}\left(\sum_{k>0} a_{ij}z^k\right) = \sum_{k>0} z^n \left(\sum_{\substack{k_0,k_1,\ldots>0 \\ k_0+k_1+\ldots=n}} a_{0k_o}a_{k_1}\cdots \right)$$

186 $$|z| = < \left(\limsup_{n \to \infty} \sqrt[n]{|a_n|} \right)^{-1}$$

192 $$\left(\int_{-\infty}^{\infty} e^{-x^2} dx \right)^2 = \int_0^{2\pi} \left(-\frac{e^{-r^2}}{2} \Big|_{r=0}^{r=\infty} \right) d\theta$$

$$\lim \frac{a_n}{a'_n} = 2 \frac{\ln(n)}{\log(n)} = 2 \ln(2) \doteq 1.386... \qquad (4.59)$$

LaraPrint 2.0

Hansrudolf Schär, 1.2.87

LaraPrint is a program to produce hardcopy for Lara documents. Its output is a file in the Lilith print file format (further called *print file*), or a *PostScript* file. A print file serves as input for a Lilith computer accessing a laserbeam printer to produce the hardcopy. Normally, LaraPrint sends the print file to the printer server over the local area network. If the output file must remain on the local disk, e.g. for inspection by the program *pisa*, press "f" when selecting the name of the printer station. Furthermore, an Apple LaserWriter connected to a Lilith can be used.

LaraPrint requires a user profile entry *Printer*.

Example:

```
"System"
'Printer'          PRISMA HAMPEL LaserWriter
```

Usage

After starting *laraprint*, the name of the printer is requested. Type *RETURN* to send the file to the printer who's name is currently displayed. Type "/" to display the next printer name. Type "f" to generate a local output file. Pressing *ESC* here lets you leave the program.

After checking the availability of the printer and the permission to print, the program prompts for the name of a file to be printed (default extension is *DOK*). Enter the filename terminating with *spacebar* or *RETURN* (or "/c" and "y" for a confidental printout) or *ESC* to cancel the name entered.

Now, the number of copies is requested. Type one digit to print from 0 to 9 copies or *spacebar* or *RETURN* for exactly 1 copy.

Following this, *laraprint* asks for the number to be printed on the first page. If you type *spacebar* or *RETURN*, the pages will not be numbered.

The prompt *pages>* allows you to restrict printing a consecutive range of pages. Enter one page number or two numbers separated by "-" (e.g. 14-19). *Spacebar* or *RETURN* prints all pages.

The required fonts are now loaded and the file is subsequently printed. Each processed page is indicated on the screen by a "." character.

After completion the message *in>* appears again for a next file to be printed. Press *ESC* if you want to leave the program.

In case of local output, the name of the output file is requested.

Example: (characters entered by the user are underlined)

```
*laraprint
Printer: PRISMA_
print file format (PRT)
remote printing
CIPON 21.12.84
installation of printer service PRISMA Prisma: printer ready
User: Schär Hans-Rudolf

laraprint 2.0, 1-Feb-87, HRS, ETHZ
in> PUB.Letter.DOK   copies> 1 pageno> 1 pages> _
font TIMESROMAN12
font GACHA14
font TIMESROMAN12B
font HELVETICA10
font HELVETICA12
font TIMESROMAN12I
formatting.
in>_
```

```
 1 page to print
*
```

Installation

The following files are needed:

SYS.laraprint.OBJ	Print Program.
LIB.PrDr.PRT.OBJ	Program module to generate a print file.
LIB.PrDr.PRTC.OBJ	Program module to generate a Ceres print file.
LIB.PrDr.PSR.OBJ	Program module to generate a PostScript file.
LIB.PrCh.Rem.OBJ	Program module for remote printing.
LIB.PrCh.Loc.OBJ	Program module for local printing.
LIB.PrCh.V24.OBJ	Program module for printing over the serial link (9600 Baud).
LIB.PrDr.PSRprologue	PostScript header.
SYS.setprinter.OBJ	Program to select a current printer.
lasercopy.OBJ	Program to transmit a PostScript file over the serial link.
LaserPrint.OBJ	Program to transmit a PostScript file over the serial link.
GENSYM*	Data files for generic symbols of mathematical formulas.
*.WID	Font data files used to generate print files.
*.PSW	Font data files used to generate PostScript files.
SILtemplate.DOK	Template document to print SIL-files directly.
DRAWtemplate.DOK	Template document to print DSP-files directly.
Lara.User.Section	Standard user profile entries.
PUB.LaraPrint.V20.DOK	This documentation.

User profile entries

Each Printer may have a user profile entry to specify a number of print parameters.

Example:

```
"LaserWriter"
'Type'              PSR
'Version'           0
'Channel'           V24
'Resolution'        2400
'X'                 0
'Y'                 0
'W'                 20480
'H'                 30720
'X0'                0
'Y0'                0
'FontScale'         0.81
'MDext'             PSW
```

The following table contains all possible attributes of the user profile and their values. If an attribute name is not in the user profile, the corresponding default value is taken.

Attribute Name	Default value	Possible values
'Type'	PRT	PRT, PRTC, PSR
'Version'	0	*positive number*
'Channel'	Rem	Rem, Loc, V24
'Resolution'	240	*positive number*
'X'	0	*positive number*
'Y'	0	*positive number*
'W'	2048	*positive number*
'H'	3072	*positive number*
'X0'	0	*number*
'Y0'	0	*number*
'FontScale'	1.0	*real number greater than 0*
'MDext'	WID	WID, PSW
'MDprx'		*string*

'Type' (printer type) specifies the output file format (PRT = print file, PRTC = Ceres print file, PSR = PostScript file).

'Version' specifies the version of the file format. For print files, the value 0 indicates that the printer cannot print arbitrary curves. It only prints horizontal and vertical lines.

'Channel' defines the output channel of the program (Rem = remote, Loc = local, V24 = serial link).

'Resolution' specifies the resolution on the printed page in pixels per inch.

'X', 'Y', 'W', 'H': size of the printable region in pixels for the given resolution. The origin is the top left corner. X increases to the right, Y increases to the bottom.

'X0', 'Y0': displacement of the origin in pixels for the given resolution. Increasing X0 shifts the content of a page to the right. Increasing Y0 shifts the content of a page to the bottom.

'FontScale' is used to scale the size of each font. For print files, values other than 1.0 are not interpreted.

'MDext': extension of font metric data files.
'MDprx': prefix of font metric data files. Mainly used to identify WID-files for a different resolution.

Standard user profile entries

The following initial entries may be used in the user profile.

```
"System"
'Printer'           PRISMA HAMPEL OMEGA OMEGA3 LaserWriter CERES

"OMEGA"
'Version'           1

"OMEGA3"
'Version'           1
'Resolution'        300
'X0'                -200
'Y0'                -100
'MDprx'             R3

"LaserWriter"
'Type'              PSR
'Version'           0
'Channel'           V24
'Resolution'        2400
'X'                 0
'Y'                 0
'W'                 20480
'H'                 30720
'X0'                0
'Y0'                0
'FontScale'         0.81
'MDext'             PSW

"CERES"
'Type'              PRTC
'Version'           0
'Channel'           Loc
'Resolution'        240

"Lara"
'Font'              SYNTAX12
'Underline'         0
'Voffset'           0
'OverWrite'         no
'LeftMargin'        2.01
'RightMargin'       168
'FirstLineIndent'   0
'LineSpacing'       4.24
'Pspace'            0.96
'Mode'              adjust
'TabStops'          10,20,30,40,50,60,70,80,90,100,110,120,130,140,150
```

```
'Wspace'            0
'Figure'            NoFig
'FigDelayed'        yes
'TopMargin'         11
'BottomMargin'      255
'Header'            NoHeader
'PnoFont'           SYNTAX12
'PnoMode'           right
'PnoLoc'            top
'PageLeft'          20
'PageRight'         188
'PageTop'           12
'PageBottom'        285

"Ascii"
'Font'              GACHA10L
'Underline'         0
'Voffset'           0
'OverWrite'         no
'LeftMargin'        1.02
'RightMargin'       181
'FirstLineIndent'   0
'LineSpacing'       3.4
'Pspace'            0
'Mode'              leftadjust
'TabStops'          10,20,30,40,50,60,70,80,90,100,110,120,130,140,150
'Wspace'            0
'Figure'            NoFig
'FigDelayed'        yes
'TopMargin'         10.5
'BottomMargin'      262
'Header'            NoHeader
'PnoFont'           GACHA10L
'PnoMode'           right
'PnoLoc'            top
'PageLeft'          11.5
'PageRight'         192.5
'PageTop'           13.5
'PageBottom'        285
```

KeyboardMacros

Hansrudolf Schär, 1.2.87

Introduction

KeyboardMacros, a program that runs on Lilith and Ceres, adds some functionality to the keyboard. It is loaded prior to starting other applications. Its main use is to enter mathematical formulas in Lara, but it also offers a service useful to other applications. The following things can be done with it:

1) Entering an arbitrary text string by typing a short name.
2) Entering ASCII characters which are not available on the keyboard.
3) Definition of function keys.
4) Input the content of a file to another application (like executing a command file).

This documentation describes version 1.0 (1-Feb-87) of that program.

Installation

The following files are needed to run *KeyboardMacros*:

KeyboardMacros.OBJ	program running on Lilith.
KeyboardMacros.OBN	program running on Ceres.
KeyboardMacros.KMC	Startup file.
Lara.KMC	Definitions usable in Lara.
Formula.KMC	Definitions to enter mathematical formulas in Lara.
Sara.KMC	Definitions usable in Sara.

Keyboard, Special Keys

The following notation is used throughout this documentation.

ESC	escape key (aborts the current command)
EOL	return key (end of line)
SP	space bar
DEL	backspace key (deletes previous character)
CAN	<Ctrl-X>, cancel key (deletes current line)
NUL	null character (0C)

Using KeyboardMacros

The program is started by typing *KeyboardMacros* followed by *EOL*. After some initializations the command interpreter is called to allow you to start other programs. From now on, the character "\" (backslash) is used to call a macro (see next paragraph). If you need to enter the backslash character without interpretation, type in "\" twice.
To leave the program, press *ESC* while in the command interpreter and confirm with "y".

Calling a Macro

To call a macro type *name* followed by *EOL* or *SP*. A macro is a defined name referring to a built in command (*standard macro*) or referring to a defined text (*user defined macro*). The standard macros are (see following description of all commands):

def, dir, reset, chdef, chredef, chdir, chrem, chreset, text, read, echo, noecho.

If the macro name has a defined text, the corresponding text is now entered. To define a text, the standard macro *def* is used. A text itself may contain macro calls to allow more complex macros by using existing ones.
If a given name has no defined text, the macro call is ignored.

When typing in *name*, you may use *DEL* or *CAN* to correct or *ESC* to abort. All characters are shown, but not passed to the application. As soon as *EOL* or *SP* is typed, the string *name* disappears.

Example:

Assume the text "programming language" has been assigned to the macro name "pl". Typing *\pl* will then immediately expand to

programming language

The text "Modula-2 \pl environment" assigned to "m2" will expand to

Modula-2 programming language environment

whenever *\m2* is typed.

Function Keys

A Function Key is a character with a macro assigned to it. See *\chdef*, *\chredef* and *\chrem* on how to assign (deassign) a macro to a character. As soon as the character is typed in, the corresponding macro is called.

Example:

Assume the macro "pl" has been assigned to the character "↑". Typing "↑" will then immediately expand to

programming language

Function keys are neither interpreted while typing the name of a macro to be called nor during all standard macro calls.
Function keys are also called "character macros".

Error Messages

The program shows error messages as soon as some (known) error occurs. Every error message starts with "----". This helps to locate errors on log-files and to identify a message as one from KeyboardMacros during other applications. The following errors may generally occur:

`---- recursive macro call`	recursive calls are not allowed
`---- macro stack overflow`	too many nested macro calls
`---- fatal error in KeyboardMacros`	some unrecoverable error

\def define a macro

The standard macro *\def* associates a text with a name. A text is a sequence of characters not containing *NUL*. It may contain up to 300 characters. A text is entered in symbolic form. The following EBNF definition illustrates the format.

```
symbolic text  = {symbol}.
symbol         = string|charconst|macrocall.
string         = "'" {char} "'" | '"' {char} '"'.
charconst      = octalDigit {octalDigit} "C" | digit {digit} ["N"].
macrocall      = "\"char{char} (" "|EOL).
```

A *string* is defined as in Modula-2. A *charconst* can be a character constant as in Modula-2 (octal code) or a decimal number (decimal code). For convenience in the description of mathematical formulas used in Lara, an unconventional decimal code starting at "0" was also introduced. In this case the decimal number is followed by "N".
A symbolic text may also contain comments as in Modula-2.

Example:

The character "8" may be specified in the following ways: "8", '8', 70C, 56, 8N.

The name of a macro is a sequence of characters not containing *ESC*, *EOL*, *SP*, *DEL*, *CAN*, *NUL* or "\". Its maximal length is 15 characters.

The *\def* macro first asks for the name of the macro, then requests the text (in symbolic form). To allow the definition of longer texts, the program asks for more than one text line. Type *EOL* at the beginning of a line to terminate an entered text.

Examples:

```
\def
macro name> pl
macro text> "programming language"
macro text>
macro pl defined

\def
macro name> m2
macro text> "Modula-2 \pl  environment"
macro text>
macro m2 defined
```

Note that no macro is called (expanded) and function keys are not interpreted while typing in a symbolic text.
A text entered symbolically is automatically converted into actual text. An already defined name may not be redefined with another text. Macros specified inside a text need not be defined in advance.

Error messages:

```
---- name too long
---- text too long
---- predefined macro name
---- macro already defined
---- too many macros (table overflow)
```

\dir directory of defined macros

The *dir* macro shows already defined macro names. Specify a search string (wildcard characters "*" and "%" are allowed) to list particular macros or simply press *EOL* to get a list of all names.

Examples:

```
\dir
dir>
def     dir     reset   chdef   chredef
chdir   chrem   chreset text    read
mat     echo    noecho
13 of 13 macros listed (7000 bytes free)

\dir
dir> *ch*
chdef   chredef chdir   chrem   chreset
echo    noecho
7 of 13 macros listed (7000 bytes free)

\dir
dir> %%%%*
reset   chdef   chredef chdir   chrem
chreset text    read    echo    noecho
10 of 13 macros listed (7000 bytes free)
```

All macro names and texts are stored internally in a buffer. Messages of the type

```
7000 bytes free
```

show the amount of unused space.

\reset initialize, remove all macro definitions

This macro initializes all macro definitions, which means that all user defined macros are cancelled. Also all function keys are cancelled, since they refer to defined macros. If *reset* is called from a macro, the calling macro is aborted.
reset is primarily used before reloading macro definitions from a file that has been modified (see also *read*).

\chdef define a function key

The *chdef* macro assigns a macro to a function key. A function key may be specified in two ways. It can be typed in directly or a symbolic text (see *dir*) of length 1 may be entered.

Examples:

```
\chdef
enter ch by typing a key? (y|n) > y
type a key> ↑
macro name> p1
character macro defined

\chdef
enter ch by typing a key? (y|n) > n
character (e.g. "a" or 201C)> "~"
macro name> m2
character macro defined
```

The characters *ESC*, *EOL*, *DEL*, *CAN*, *NUL* and "\" may not be used as function keys. Note that characters in existing macro texts may become function keys.

Error messages:

```
---- text too long
---- only one character expected
---- unknown macro
---- character already used
---- this character cannot not be used
```

\chredef redefine a function key

chredef is like *chdef*, but additionally allows the replacement of a function key definition.

Error messages:

```
---- text too long
---- only one character expected
---- unknown macro
---- this character cannot not be used
```

\chdir directory of defined function keys

The *chdir* macro lists the names assigned to all function keys.

Example:

```
\chdir
"↑":p1
"~":m2
2 character macros listed
```

\chrem remove a function key definition

Cancel a function key definition. The function key is specified in the same way as in the *chdef* macro.

Example:

```
\chrem
enter ch by typing a key? (y|n) > y
type a key> ~
character macro removed
```

Error messages:

```
---- text too long
---- only one character expected
---- character macro not used
---- this character cannot not be used
```

\chreset initialize function keys, remove all function keys

All defined function keys are cancelled.

\text enter a (symbolic) text

The *text* macro asks for a text to be entered immediately. See *def* on how to enter a symbolic text. *text* is like defining and calling a macro with no name. It is mainly used to enter control characters.

Example:

```
\text
text> 14C (* form feed *)
```

Error messages:

```
---- text too long
```

\read read a file

The *read* macro takes the content of a file as a text to be entered. Type in the name of the file. If the name ends with a ".", the extension KMC is appended automatically.

Example:

```
\read
macro file name> Lara.KMC
```

If the file contains calls of the *def* macro, *read* can be used to install a set of frequently used macros. To ease the installation of frequently used definitions, the file KeyboardMacros.KMC is read automatically at the beginning of the program *KeyboardMacros*.
If *read* is called in the command interpreter, a file without *def* macros is like a commandfile.
A file may contain calls of *read* to include other files.

Error messages:

```
---- file not found
---- file already opened
---- too many files opened
```

\mat enter a matrix (used in Lara)

The control sequence of a matrix is entered. The editor Lara interpretes this sequence as a mathematical formula with type matrix.

Example:

```
\mat
mat: mode(left,center,right)> l
mat: spacing(uniform,normal,individual)> n
mat: lines> 3
mat: columns> 4
```

Error messages:

```
---- unknown mode
---- unknown spacing
---- nr of lines must be >= 1
---- too many lines
---- nr of columns must be >= 1
---- too many columns
```

\noecho switch dialog off

The *noecho* macro switches the display of dialog messages off. From now on the call of a macro is no longer echoed on the display. Use *noecho* at the beginning of a KMC file (see *read*).
The display of error messages is not surpressed by *noecho*.

\echo switch dialog on

The *echo* macro switches the display of dialog messages on.

LaraMail

M. Grieder, 11.11.86

Kurze Benutzungsanleitung für LaraMail

Einleitung:

Lara wurde um insgesamt 5 Mail-Befehle erweitert, die zusammen mit den üblichen Editierfunktionen ein benutzerfreundliches Mail-System bilden. Die expliziten Operationen sind: Mailbox-Inhalt anzeigen, eine Meldung anzeigen oder löschen, ein Dokument formatiert oder unformatiert absenden. Funktionen, wie z.B. beantworten, weiterleiten, abspeichern usw., sind (implizit) sehr rasch möglich.

mail-Befehl:

Bewirkt das lesen des Mailbox-Inhalts. Ist keine Meldung vorhanden, wird dies im Dialog-Fenster vermerkt, andernfalls muss ein Fenster eröffnet werden um die Mailbox-Einträge als Dokument anzuzeigen. Dieses Mailbox-Dokument kann nicht verändert werden, es wird als read-only Lara-Dokument betrachtet. Die als Paragraph dargestellten Meldungen können mit Doppelklick selektiert werden; darauf wirken dann die Befehle *show* und *remove* im Mail-Menu (statt Text-Menu).

show-Befehl:

Die entsprechend selektierte Meldung wird als Laradokument aufbereitet und kann angezeigt werden. Bei unlesbaren Files (Larafiles für welche Fonts fehlen, Module, binäre Daten usw.) wird zumindest der Header mit einer entsprechenden Meldung dargestellt, zum Übertragen der Mail muss "mailtransfer" verwendet werden (s.u.).

remove-Befehl:

Löscht eine selektierte Meldung im Mail-Fenster und in der Mail-Box, wobei als einzige Sicherheit eine Bestätigung verlangt wird. Es ist hier auch möglich mehrere Meldungen gleichzeitig zu löschen.

send- und *send u*-Befehle:

Erlauben jedes Laradokument (als Meldung) zu verschicken. Dazu schreibt man vorne im Dokument einen Header. Er besteht aus zwei Rubriken, die mit den Schlüsseln 'Subject:' und 'To:' eingeleitet werden. Als Titel sind maximal 63 Zeichen erlaubt. Die Adressen der externen Empfänger entsprechen der heutigen uucp-Norm an der ETH. Für die internen Teilnehmer genügt meist der Name, ist dieser nicht eindeutig, bedarf es Name und Vorname. Der Titel muss mit EOL, jede Adresse mit Komma oder EOL und der Header als ganzes durch ein LF (Paragraph) abgeschlossen sein.

Beispiel:

```
Subject:    LaraMail .....
To:         Gutknecht, Grieder
            Meier

..... Meldung .....
```

Beim Aufruf von *send oder send u* im Balken-Menu wird der Header untersucht. Dabei schickt *send* die Meldung als Larafile (formatiert). *send u* hingegen sendet die Meldungen als Asciifiles (unformatiert), was für externe Empfänger unerlässlich ist.

Adress-Liste:

Das File "#mbx.PUB.MailUser.LST" beinhaltet alle berechtigten Mail-Benutzer. Es wird automatisch auf dem neuesten Stand gehalten und kann zum Erstellen eigener Adress-Listen verwendet werden.

Eine Adressen-Liste "DK.MailUser.MAP" kann sich jeder Benutzer selbst erstellen. In diesem 'Mapping'-File steht: Listenname '=' Adresse ',' Adresse ',' usw. ';'.
Hier sind verschiedene Adressen durch Komma oder EOL gtrennt und der Strichpunkt schliesst eine Liste ab. Kommentare können in (Klammern) hinter jedem Namen stehen. Auch hier existiert ein Hilfs-File "#mbx.PUB.MailUser.MAP" als Grundlage zum Erstellen des eigenen 'Mapping'-Files.

Im To-Feld können die Listen durch Pfeil und Name spezifiziert werden (s.u.). Beim senden werden die entsprechenden Namen eingefügt; dabei ist Rekursion zugelassen und bei geöffnetem 'Mapping'-File wird der aktuelle Zustand verarbeitet.

Beispiel:

DK.MailUser.MAP: Grp.Gutkn = Gutknecht, Grieder, ...,
.....;
(* = Gruppe Gutknecht *)

Header: Subject: LaraMail .....
To: ↑Grp.Gutkn

Hilfslisten:

#mbx.PUB.MailUser.LST beinhaltet alle berechtigten Mail-Benutzer.
#mbx.PUB.MailUser.MAP ist ein Grundstock an Adress-Listen.

Replayfile:
Mailbox-Befehle im Replayfile könnten zu irreparablen Inkonsistenzen führen, dies darf also nicht unterstützt werden:
Das Replayfile ist nur bis zum ersten Aufruf eines Mailbox–Befehls aktiv !

SYS.mailtransfer.OBJ:
Dieses Modul überträgt Meldungen beliebigen Formats von der Mailbox auf die lokale Disk. Der Dialog ist sehr einfach. Bejaht man "read ?", so wird Header und Body (auf Anfrage; "show body ?") angezeigt und man kann auf "write to>" den Filenamen (Default: DK.*.MSG) spezifizieren, existiert das File bereits, muss "replace?" beantwortet werden. Ist eine Mail übertragen ("done"), kann sie auf der Mailbox gelöscht werden ("delete mail ?"), was ebenfalls mit "done" quittiert wird. Im übrigen kann die Maus als Antwort-Beschleuniger eingesetzt werden (ML = 'yes', MM = ESC, MR = 'no').

Fragen, Anregungen:
Für Fragen, allfällige Fehler oder Anregungen, wende man sich bitte direkt an:
Grieder Markus resp. Grieder@ifi.ethz.chunet

LaraSpell

Benutzeranleitung

M. Grieder 31.1.87

1. Vorbemerkung

"Spell" ist ein auf 4 Sprachen ausgerichteter Dictionary-Lookup-Spelling-Checker. Er ist im Lara als 'spell'-Befehl eingebaut und eine Batch-Version ist unter dem Namen "Spell.OBJ" verfügbar.

Das Programm vergleicht den Text mit einem codierten Dictionary, wobei ein Wort als Buchstabenkombination inklusive eingeschlossener Apostrophs definiert ist. Insbesondere werden mit Bindestrich unterteilte oder getrennte Worte ebenfalls aufgeteilt.

Die Online-Version ist auf dem File-Server verfügbar:
#ma.ED.SYS.laras.OBJoder
#ma.ED.SYS.larafms.OBJ
(s: spell, f: formula, m: mail; wie das File lokal heisst, ist unwesentlich).

Die zentral verwalteten Dictionary-Files (sprachabhängig) sind ebenfalls auf Maple.
english: #ma.ED.*.DIC
deutsch: #ma.ED.*.DIX
(Lokal sind die Files *.*, also gleich zu nennen; Erklärungen s.u.).

Momentan steht ein deutscher Dictionary mit 16'000 Worten und ein englischer mit 8'000 Worten auf Maple. Ich habe weitere 16'000 deutsche Worte in unkodierter Form bereit und 40'000 englische Worte in Aussicht, die bei Gelegenheit hinzu kommen sollen. Um die Dictionarys zu vergrössern, können Sie sich mit Listen unbekannter Worte oder umfangreichen Texten aller Art (auch französisch) an mich wenden (mail: grieder@ifi.ethz.uucp).

2. Benutzung der Online-Version

Im Lara hat das "TEXT"-Menu einen Befehl 'spell' mit Folge-Menu:
"SPELL" 'english' 'german' 'french' 'others' 'clear'

Beim ersten Aufruf wählt man die Sprache (resp. Default 'english') und der entsprechende Dictionary wird eröffnet. Eventuelle Fehler oder Meldungen werden im Dialog-Fenster angezeigt (s. u.) ansonsten beginnt der Prüfvorgang sofort.

Der Spelling-Checker sucht ab der momentanen Caret-Position das nächste unbekannte Wort und positioniert das Caret neu an die gefundene Stelle (wie 'find') oder es erscheint im Dialog-Fenster die Meldung "no unknown words found".

Ist die Sprache initialisiert, so gilt sie bis man eine andere oder 'clear' wählt, d.h. für weitere Aufrufe genügt 'spell' ohne Folge-Menu.

Ist eine Textstelle selektiert, sucht der Spelling-Checker nur innerhalb dieser Auswahl, wobei die Start-Position durch das Caret bestimmt wird: ist das Caret ausserhalb, wird vom Beginn der Selektion, sonst von der momentanen Caret-Position geprüft.

Da beim Prüfen viele File-Opeationen gemacht werden, ist es sinnvoll ein Laradokument (nur) paragraphen-weise (Bsp: MR MR ML ML 'spell') und ganze Dokumente mit der Batch-Version zu testen. Auf den kleinen Liliths (128 kWorte Hauptspeicher) ist es auch angezeigt, nach dem prüfen mit 'clear' den Speicher wieder zu entlasten.

3. Benötigte Files

Der Spelling-Checker benötigt einige Dictionary-Files, die im folgenden aufgelistet sind. Die Extensions sind sprachabhängig: ".DIX" für Deutsch, ".DIC" für Englisch, ".VOC" für Französisch und ".VOK" für die vierte Sprache. Diese zusätzliche Sprache ist nicht festgelegt, der Spelling-Checker verwendet einfach alle Files mit Extension ".VOK", was es dann auch immer sei, insbesondere wird dazu auch kein zentraler Dictionary verwaltet.

*Dictionary.**
ist das eigentliche Dictionary-File. Es muss zum Prüfen auf der Disk sein und kann periodisch vom File-Server (z.B. als verbesserte Version) kopiert werden.

*Structural.**
beinhaltet die Strukturworte (der, die, und, ...) und am Institut sehr häufig verwendete Worte (Computer, Prof., Zentrum, ...). Diese Worte werden eingelesen und schnell geprüft. Fehlt dieses File, funktioniert der Spelling-Checker trotzdem, aber etwas langsamer.

*Project.**
ist ebenfalls ein Wortlisten-File, das z.B. pro Forschungsgruppe einmal erstellt werden kann. Auch diese Worte werden wie die Strukturwörter eingelesen und rasch geprüft. Der Name "Project" ist ein Default-Name, der mit einem Eintrag im "User.Profile" geändert werden kann (siehe unten).

*User.**
Dieses dritte Wortlisten-File ist, wie schon der Name sagt, ganz speziell für den Benutzer. Es kann im Editor oder Lara erstellt werden und wird wie oben behandelt. Auch "User" ist ein Default-Name der im "User.Profile" umbenannt werden kann (siehe unten).

*Temp.**
Für spezielle Anwendungen ist es sinnvoll, auch eine spezielle Wortliste bereit zu haben. Sie kann Fachausdrücke des momentanen Teilprojekts oder kurzfristig häufig verwendete Namen enthalten. Es ist auch möglich, dieses File während der Session im Lara zu schreiben oder zu verändern, die neuen Worte sind aber erst verfügbar, nachdem das File geschlossen und der Dictionary neu geladen ist.

User.Profile
Hier stehen (eventuell) unter dem Titel "spell" die Namen für Projekt- und Benutzerfile.
Beispiel: "spell"

'project'	SpellCheck	ergibt z.B. "DK.SpellCheck.DIX"
'user'	MarkusG	ergibt z.B. "DK.MarkusG.DIX"

Bemerkungen:
- Die Wortlisten-Files können klein und gross geschriebene Worte beinhalten.

Steht nur das kleine Wort, so wird die grosse Form trotzdem akzeptiert.
- Mischformen (zT, uAwg, GmbH, ...) akzeptiert der Spelling-Checker nur zweibuchstabig, d.h. längere Mischformen müssen in den Wortlisten-Files stehen, damit sie erkannt werden.
- Einbuchstabige "Worte" sind immer korrekt!

4. Fehler- und System-Meldungen

Eine Fehlermeldung wird beim Eröffnen des Dictionary ausgegeben, falls der Speicherplatz nicht genügt oder File-Operationen unkorrekt ausgeführt wurden. Erscheint eine Fehlermeldung, kann der Spelling-Checker auch nicht prüfen. Möglich sind:

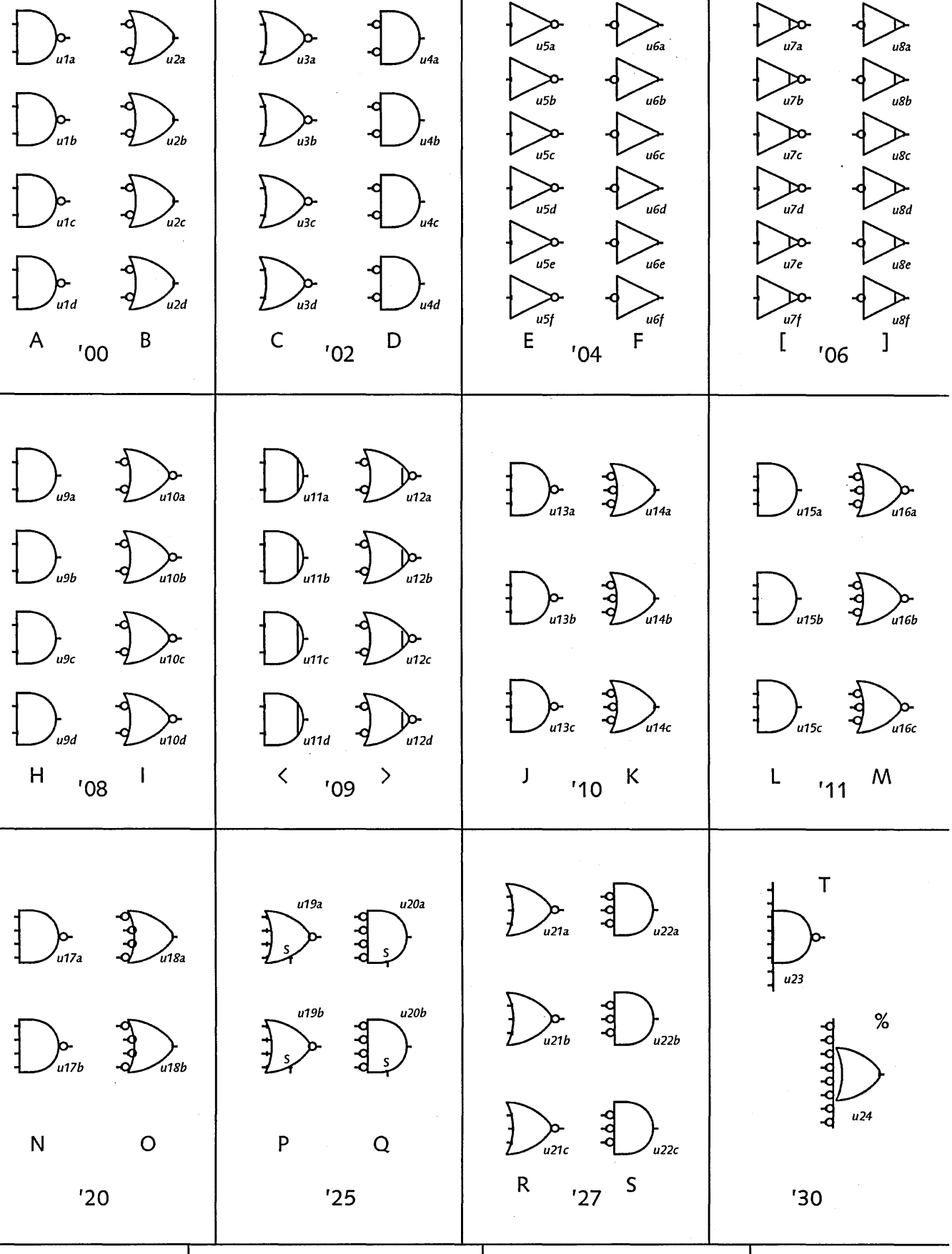
not enuogh space	zum Eröffnen des Dictionary
not found	"Dictionary.*" ist nicht auf der Disk
not opened	Fehler beim Eröffnen oder Lesen auf "DK.Dictionary.*"
not loaded	Fehler beim Einlesen von Wortlisten-Files
notdone	Nicht lokalisierter Fehler

Systemmeldungen sind rein informativ, das Prüfen geht trotzdem, und sie stehen deshalb auch nur in Klammern:

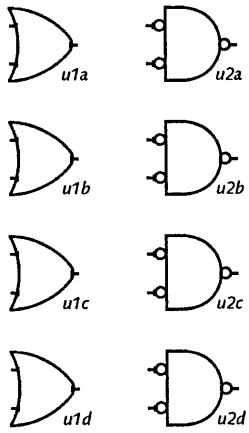
(no wordlist)	Speicherplatz zu klein für die interne Wortliste
(no wordbuf)	Speicherplatz zu klein um Worte zu lesen
(no strc-WL)	"Srtuctural.*" nicht oder unvollständig eingespeichert
(no proj-WL)	"Project.*" nicht oder unvollständig eingespeichert (obiges geladen)
(no user-WL)	"User.*" nicht oder unvollständig eingespeichert (obige geladen)
(no temp-WL)	"Temp.*" nicht oder unvollständig eingespeichert (obige geladen)
(not all)	Nicht genügend Speicherplatz

Im Normalfall und bei genügendem Speicherplatz steht aber:

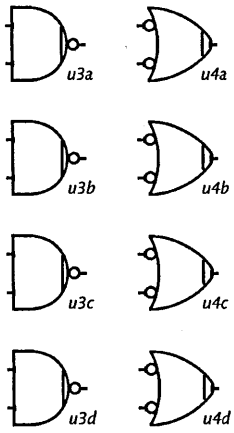
done	alles korrekt (alle Wortlisten geladen)



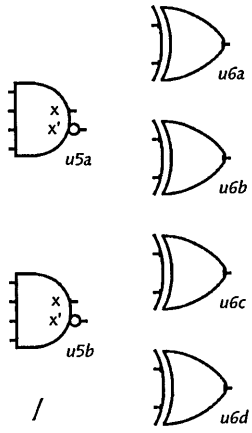
Zurich	mac1a.PIN	Author: N.Wirth	Date: 5.12.81



U '32 V

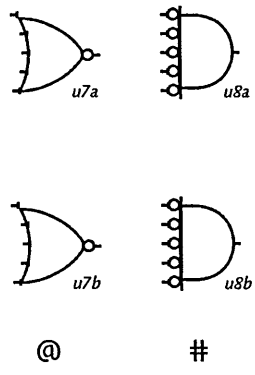


('38)



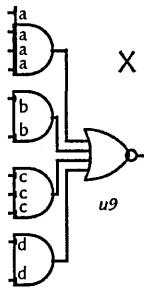
/ Z

'60 '86



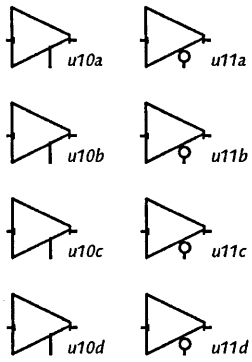
@ #

'260



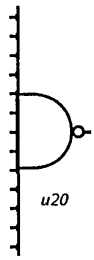
X

'64



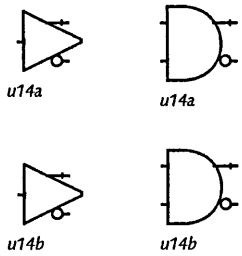
*6 *8

'126 '125



W

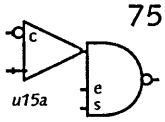
'133



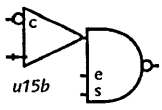
*[*]

'265

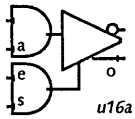
75107



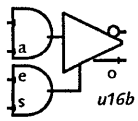
*q



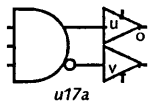
75110



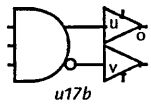
*r

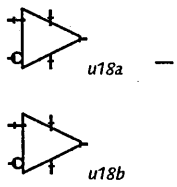


75114



:

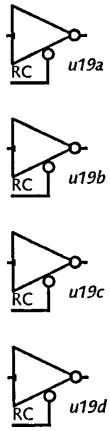




–

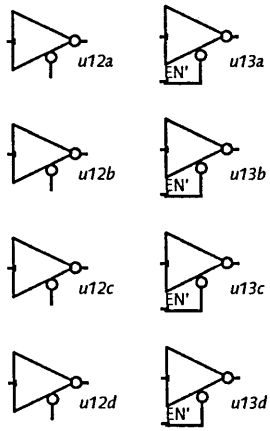
75115

*~



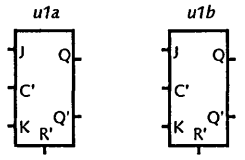
75189

*Z ~



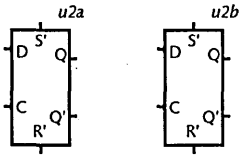
8T09 8T96

Zurich	mac1a	Author: N.Wirth	Date: 9.1.82



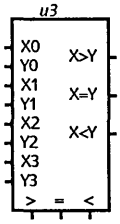
0

'73



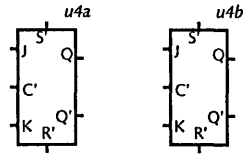
1

'74



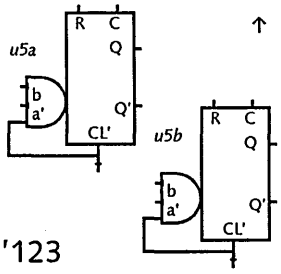
a

'85

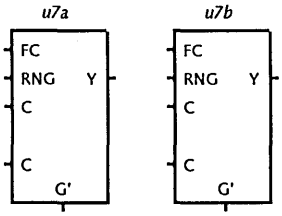


=

'112

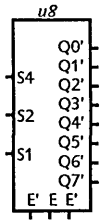


'123



'124

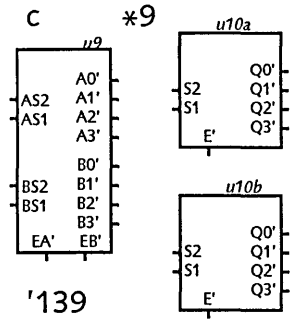
*B



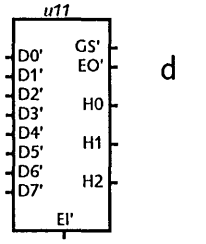
b

'138

c *9



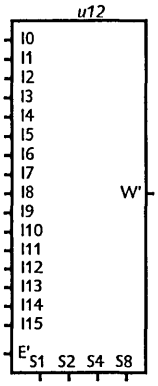
'139



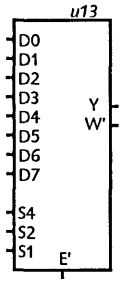
d

'148

e



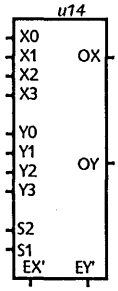
'150



f

'151

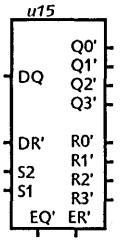
'251



g

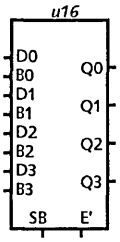
'153

'253



h

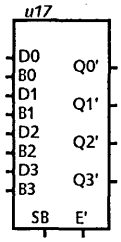
'155



i

'157

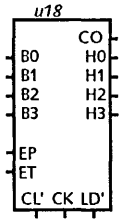
'257



j

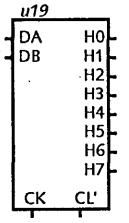
'158

'258



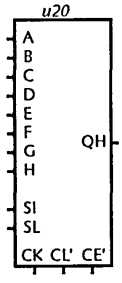
k

'160 – '163



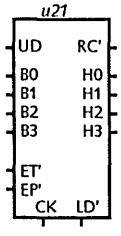
l

'164



7

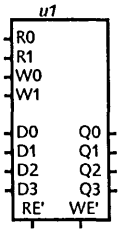
'166



*d

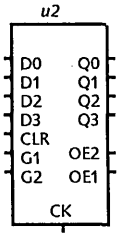
'169

Zurich	mac2a	Author: N.Wirth	Date: 6.12.81



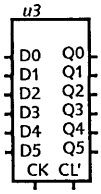
m

'170 '670



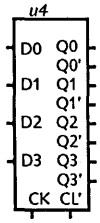
*G

'173



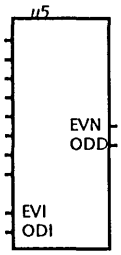
n

'174



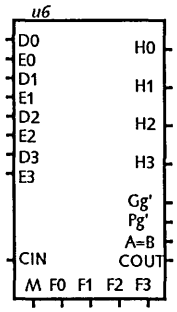
o

'175



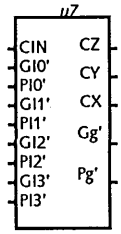
p

'180



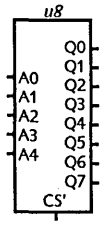
q

'181



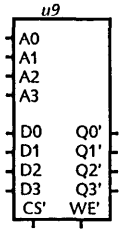
r

'182



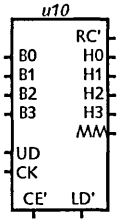
?

'188



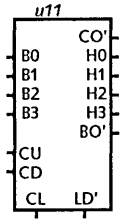
4

'189



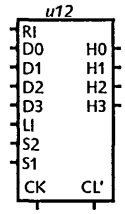
s

'190 '191



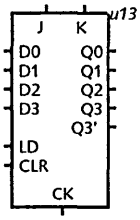
t

'192 '193



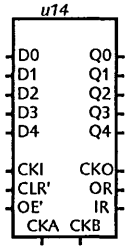
u

'194



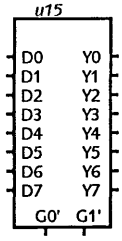
*H

'195



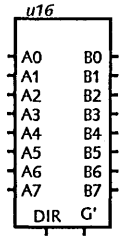
*g

'225



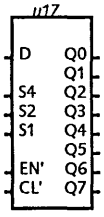
*C

'240 '244



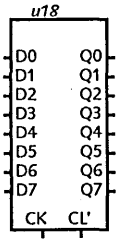
*D

'245



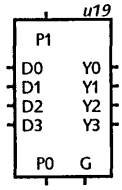
*k

'259



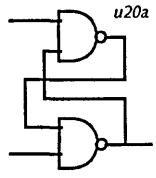
*e

'273



*I

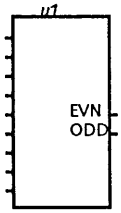
'278



*K

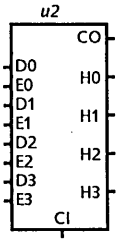
'279

Zurich	mac3a	Author: N.Wirth	Date: 6.12.81



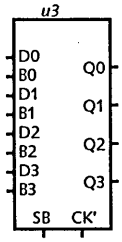
v

'280



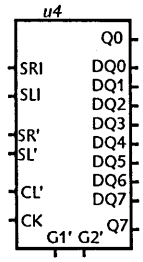
5

'283



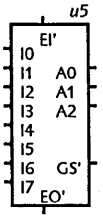
2

'298



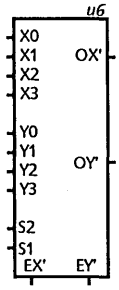
*o

'299



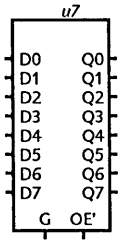
*Y

'348



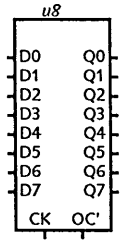
;

'352



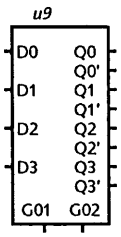
*E

'373



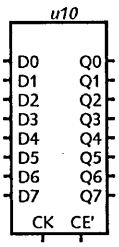
*f

'374



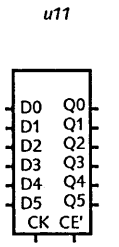
*A

'375



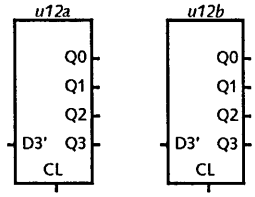
*F

'377

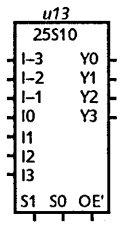


*X

'378

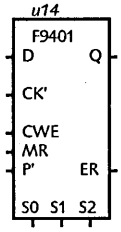


'393



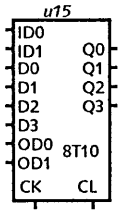
*a

25S10



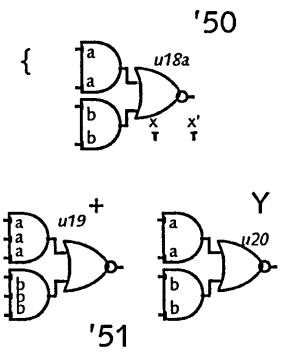
*%

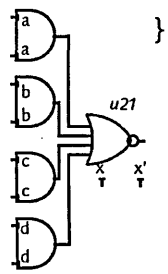
F9401



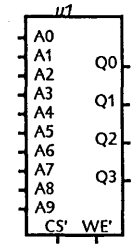
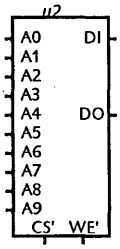
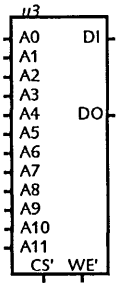
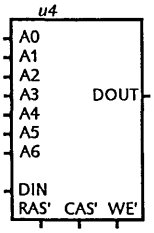
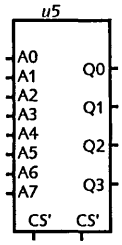
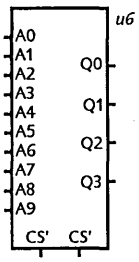
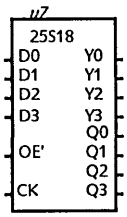
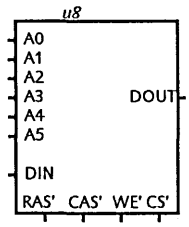
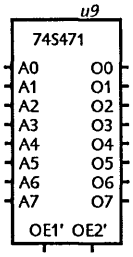
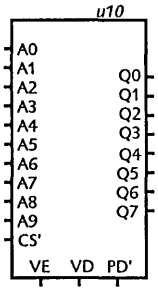
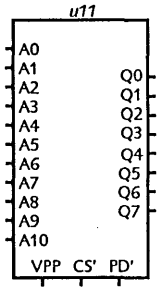
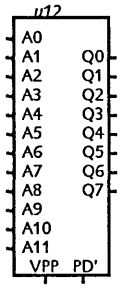
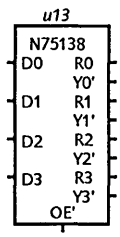
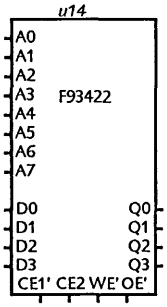
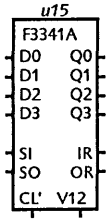
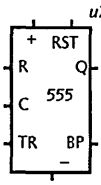
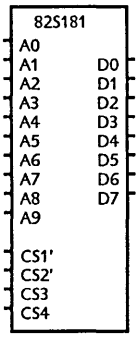
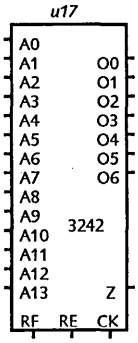
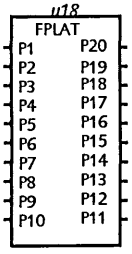
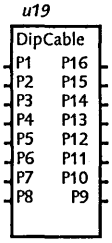
$

8T10



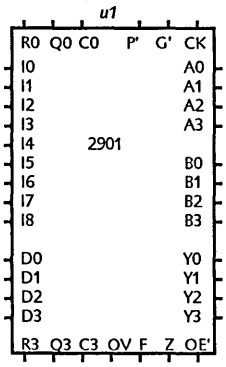


Zurich	mac4a	Author: N.Wirth	Date: 6.12.81

2114 2148  *j	2115  3	2147  *<	 4116 !
 82S129 *	 3625 *i	 25S18 9	 4096 8
 74S471 *J	*v  2708	*n  2716	*m  2732
 75138 *c	*h  	 F3341A *b	 555 *↑
 82S181 *O	 3242 *X	 *0	 *l

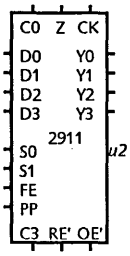
Zurich	mac5a	Author: N.Wirth	Date: 6.12.81

*L



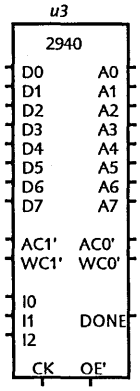
2901

*M



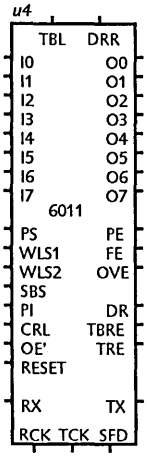
2911

*N



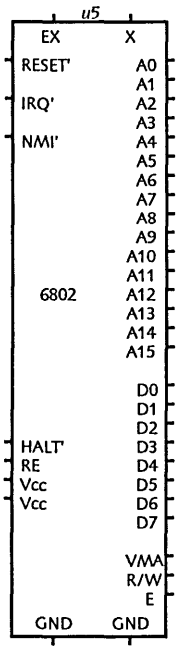
2940

*P



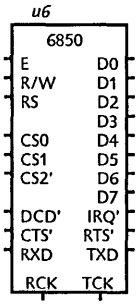
6011

*Q



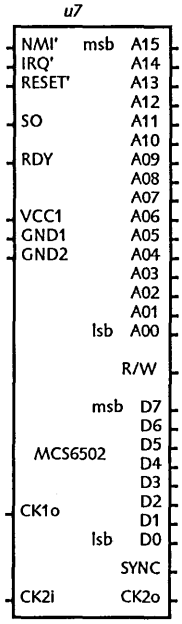
6802

*R



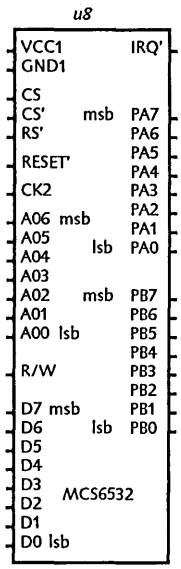
6850

*(



6502

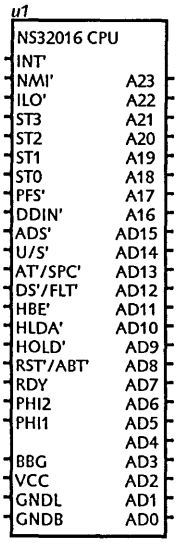
*)



6532

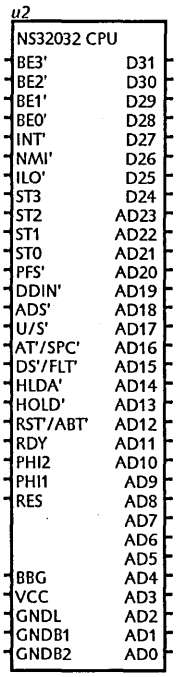
Zurich	mac6a	Author: N.Wirth	Date: 6.12.81

*!



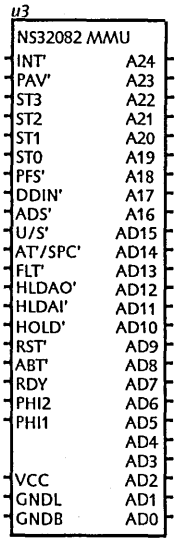
32016 CPU

*?



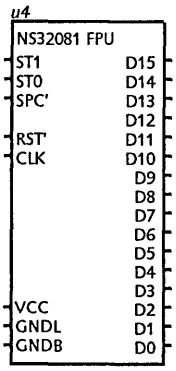
32032 CPU

*"



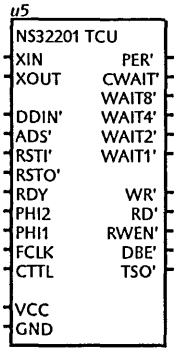
32082 MMU

*#



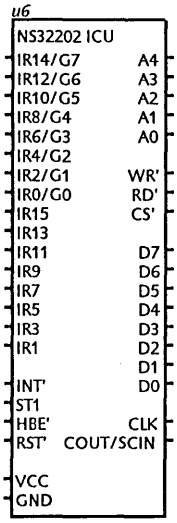
32801 FPU

*$



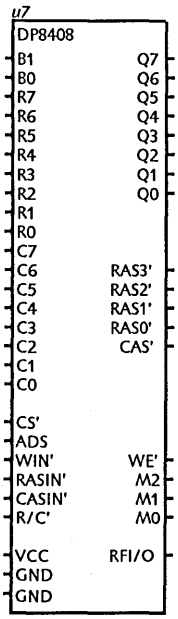
32201 TCU

*&



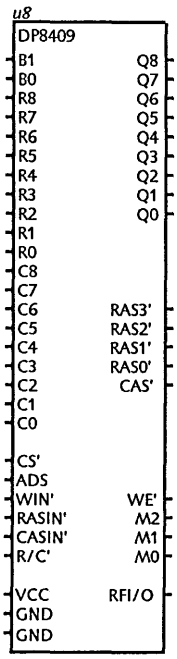
32202 ICU

*3



DP8408

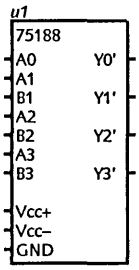
*>



DP8409

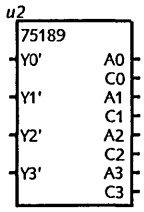
Zurich	mac7a	Author: H.Eberle	Date: 13.1.85 REV. 24.4.85

*5



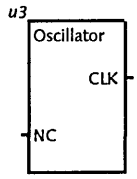
75188

*7



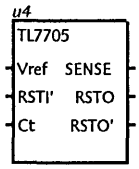
75189

*:



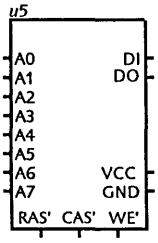
Oscillator

*;



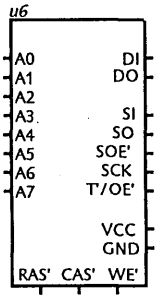
TL7705

*1



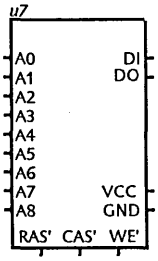
4164

*2



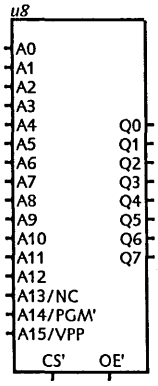
4161

*=



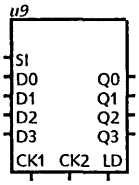
41256

*{



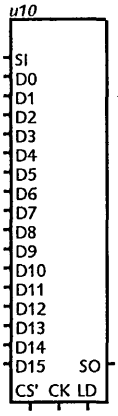
eprom
2764/128/256

*'



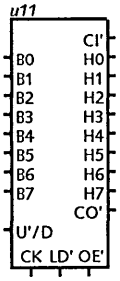
'95

*}



'676

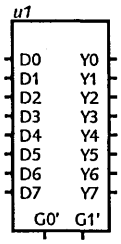
*@



u/d counter

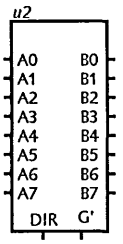
Zurich	mac8a	Author: H.Eberle	Date: 20.6.85

*+



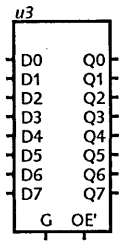
'540 '541

*,



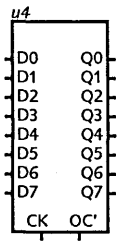
'645

*-



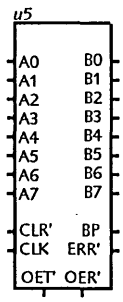
'573

*.



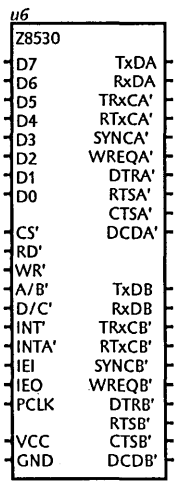
'574

**



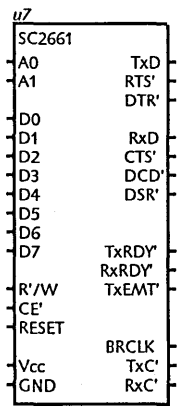
Am29833

*|



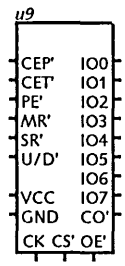
Z8530 SCC

*4



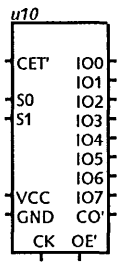
SC2661

*Z



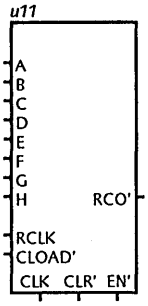
'579

*W



'779

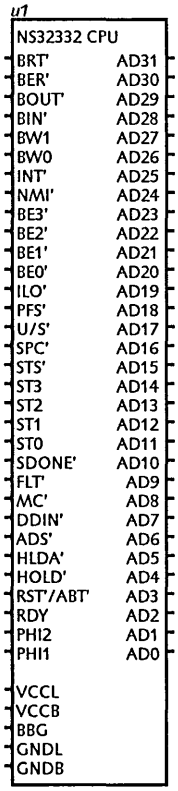
*/



'592

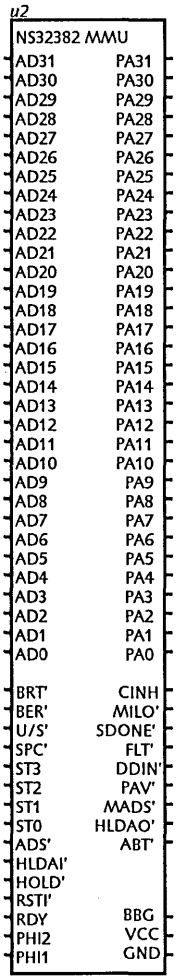
Zurich	mac9a	Author: H.Eberle	Date: 6.11.87

*S



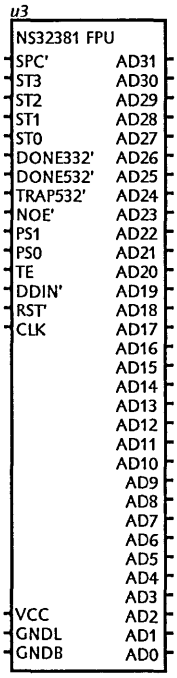
32332 CPU

*T



32382 MMU

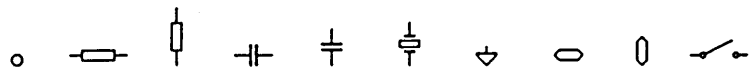
*U



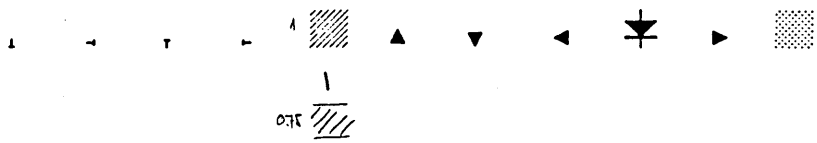
32381 FPU

Zurich	mac10a	Author: H.Eberle	Date: 6.11.87

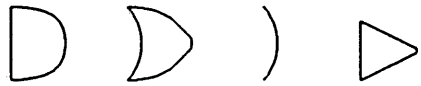
! " # $ % & ' ( ) *



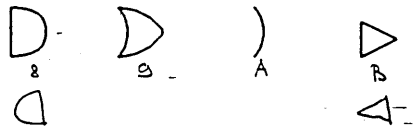
+ , - . / : ; < = > ?



0 1 2 3



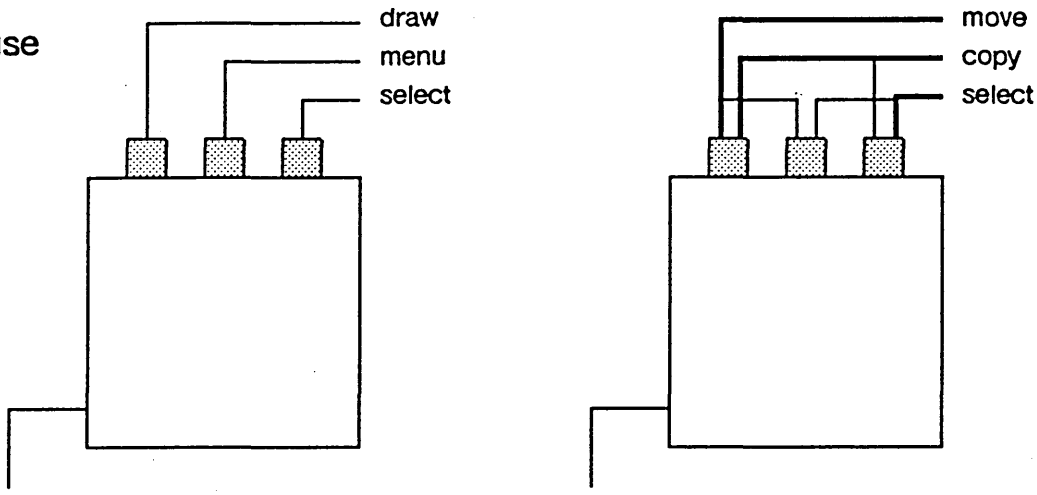
4 5 6 7



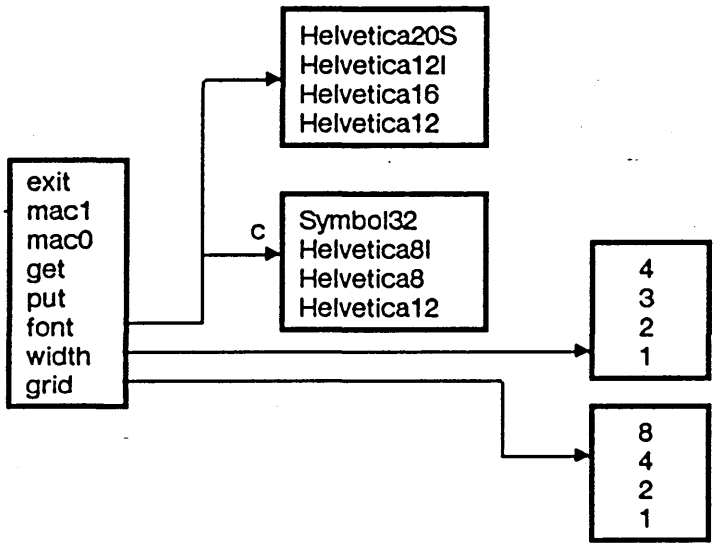
E T H

E T H

Mouse



Menu



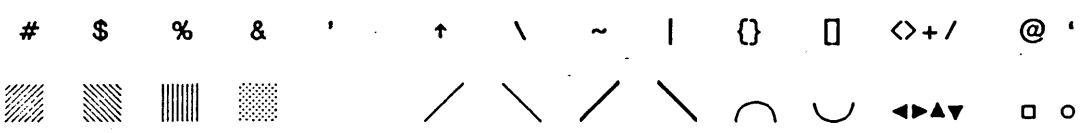
Keyboard

Text input; terminate with RETURN; DEL for corrections

Special commands:

ESC	delete
DEL	delete
LINEFEED	restore
<cntl>t	ticks on/off
<cntl>u	Umlaut on/off
<cntl>n	upper/lower part

Special characters in HELVETICA20S:



Decimal	Octal	Hex	ASCII	Decimal	Octal	Hex	ASCII	Decimal	Octal	Hex	ASCII	Decimal	Octal	Hex	ASCII
0	000	00	NUL	32	040	20	SP	64	100	40	@	96	140	60	`
1	001	01	SOH	33	041	21	!	65	101	41	A	97	141	61	a
2	002	02	STX	34	042	22	"	66	102	42	B	98	142	62	b
3	003	03	ETX	35	043	23	#	67	103	43	C	99	143	63	c
4	004	04	EOT	36	044	24	$	68	104	44	D	100	144	64	d
5	005	05	ENQ	37	045	25	%	69	105	45	E	101	145	65	e
6	006	06	ACK	38	046	26	&	70	106	46	F	102	146	66	f
7	007	07	BEL	39	047	27	'	71	107	47	G	103	147	67	g
8	010	08	BS	40	050	28	(	72	110	48	H	104	150	68	h
9	011	09	HT	41	051	29	)	73	111	49	I	105	151	69	i
10	012	0A	LF	42	052	2A	*	74	112	4A	J	106	152	6A	j
11	013	0B	VT	43	053	2B	+	75	113	4B	K	107	153	6B	k
12	014	0C	FF	44	054	2C	,	76	114	4C	L	108	154	6C	l
13	015	0D	CR	45	055	2D	-	77	115	4D	M	109	155	6D	m
14	016	0E	SO	46	056	2E	.	78	116	4E	N	110	156	6E	n
15	017	0F	SI	47	057	2F	/	79	117	4F	O	111	157	6F	o
16	020	10	DLE	48	060	30	0	80	120	50	P	112	160	70	p
17	021	11	DC1	49	061	31	1	81	121	51	Q	113	161	71	q
18	022	12	DC2	50	062	32	2	82	122	52	R	114	162	72	r
19	023	13	DC3	51	063	33	3	83	123	53	S	115	163	73	s
20	024	14	DC4	52	064	34	4	84	124	54	T	116	164	74	t
21	025	15	NAK	53	065	35	5	85	125	55	U	117	165	75	u
22	026	16	SYN	54	066	36	6	86	126	56	V	118	166	76	v
23	027	17	ETB	55	067	37	7	87	127	57	W	119	167	77	w
24	030	18	CAN	56	070	38	8	88	130	58	X	120	170	78	x
25	031	19	EM	57	071	39	9	89	131	59	Y	121	171	79	y
26	032	1A	SUB	58	072	3A	:	90	132	5A	Z	122	172	7A	z
27	033	1B	ESC	59	073	3B	;	91	133	5B	[	123	173	7B	{
28	034	1C	FS	60	074	3C	<	92	134	5C	\	124	174	7C	|
29	035	1D	GS	61	075	3D	=	93	135	5D	]	125	175	7D	}
30	036	1E	RS	62	076	3E	>	94	136	5E	^	126	176	7E	~
31	037	1F	US	63	077	3F	?	95	137	5F	_	127	177	7F	DEL

VI commands

...iser

:Q	exit ohne zurückschreiben
:w fname	schreiben auf ein File
ZZ	exit mit zurückschreiben

...it:

i text <ESC>	insert
a text <ESC>	append
ns text <ESC>	substitute next n chars by text
A text <ESC>	append at end of line
:r fname	read file (fname) and insert at cursor pos
d..	delete (.. steht für irgendwas; n steht für n times
dw	delete next word
db	delete last word
d pos	delete 1 Char pos steht für Positionierung

...sitioning

h	: links left
j	: down
u	: up
l	: right
B	: at begin of txt ?
G	: at end of txt

...neral:

u : undo last command.