ETH

Eidgenössische Technische Hochschule
Zürich

Institut für Informatik

Frank Peschel, Matthias Wille

Porting Medos-2 onto the Ceres Workstation

April 1987

Address of the authors:

Institut für Informatik
ETH-Zentrum
CH-8092 Zürich / Switzerland

Porting Medos-2 onto the Ceres Workstation

Frank Peschel, Matthias Wille

Abstract

Medos-2 is a single-user operating system that is entirely written in the language Modula-2. It has been developed for the Lilith workstation at the Institut für Informatik, of ETH Zürich. Ceres is a personal workstation based on the 32-bit microprocessor NS32000. This paper describes the tools for the cross-development of Modula-2 software as well as the actual porting of Medos-2.

1. Introduction

This paper describes the software part of the Ceres-Project which has been launched at the Swiss Federal Institute of Technology (ETH) in 1984. Ceres stands for Computing Engine for Research, Engineering and Science.

The project started in spring 1984 when the concept of a new personal workstation has been proposed by N. Wirth and H. Eberle. The goal was the development of a simple and yet powerful workstation, cheap enough to be sold as a kit to each student at the computer science department. The concept was not based on a proprietary processor as the Lilith [Wir81] but on a standard microprocessor. Note, that the Lilith is a microprogrammed implementation of a stack machine based on the AMD2900 bit-slice family. The choice of the processor for Ceres was influenced by the need of an architecture which is well suited for the compilation of high-level languages namely the language Modula-2. For this reason the NS32000 family of microprocessors was chosen. With respect to the support of high-level languages it has the best instruction set among the available processors [Wir86]. After building the first prototype with the NS32016 processor the idea of the student machine has been buried due to the high costs compared to competitors in the 16-bit market. The new aim was now the development of a true 32-bit machine using the software compatible NS32032 processor. This machine should be a follower of the Lilith. The goal has been reached in summer 1986 when the first 5 Ceres workstations have been operational. A complete description of the Ceres hardware design is given in [Ebe87].

The work presented herein concerns the operating system that has been created for Ceres. The software project started in summer 1984 and comprised the tools for the cross development as well as the actual porting of the Lilith operating system Medos-2 [Knu83]. The paper is divided into four parts and an appendix. First we present some theory about portability. The second part describes the tools for the cross development of Modula-2 programs. The third part gives a detailed description of the Medos-2 implementation and some performance measurements. The last part contains some concluding remarks about the porting of Medos-2.

2. About Porting

2.1. Software Portability

As Tanenbaum stated in [Tan84], since the building of the second computer it has been a matter of research to write a program that runs on more than one machine without being rewritten. Thus the portability of programs is a major goal of software design. The need for portability is still due to the variety of machines and machine architectures used today and the need for standardized software packages. Although is has been a research topic for lots of years we would like to sketch a few ideas about portability.

Why should programs be portable? As the costs of hardware have dramatically declined in the last years, it turned out that the software is the most critical part in a computer system. The value of a new system heavily depends on the programs that are provided. Another reason for programs to be portable are the high costs and error prone rewriting. While in the first days of computer programming each tool was in some sense unique, because it was written for a special machine, the evolution of the high-level languages opened the doors towards portability. Starting with FORTRAN in 1956 dozends of high-level languages have been created to aid the development of portable software.

Because of the growing complexity of systems and software the need of reusable program parts, so called libraries emerged. Programs and interfaces should be available not on just one machine. Totally different machines should run the same programs and thus make them easier to use because of known user interfaces. Even whole operating systems, like Unix have been made portable.

2.2. Steps during Porting

This section describes not only the steps necessary to port a program but shows how a language, in our case Modula-2, can be made available on a new hardware. First we introduce some terminology. Porting involves two computers. The computer for which a program was originally written, is called the *host machine* or *host*. The computer onto which the program is to be transferred is referred to as the *target machine* or *target*. To illustrate the porting steps we use the common method of T-diagrams [Ear70] as shown in Figure 2.1.

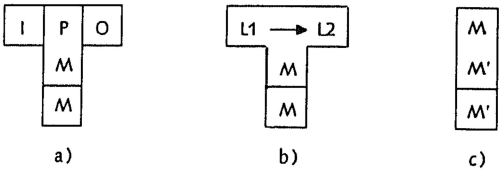


Figure 2.1 Basic T-Diagrams

The diagram 2.1a denotes a certain program *P* that is written in language *M* and thus executable on machine *M*. The program transforms an input *I* into an output *O*. A special case of such a program is a compiler, which is shown in 2.1b. It is itself written in language

M and translates a program written in *L1* into a program written in *L2*. The figure 2.1c denotes an interpreter running on a machine *M'* interpreting instructions of a machine *M*.

2.2.1. Making the Language Available

The first step is to make the implementation language available for the new hardware. There are different methods to achieve this goal. The first method uses a compiler that is already present on the host machine. On the target machine an interpreter is needed for the code produced by the compiler. This interpreter is mostly written in the machine language of the target machine. Note, that for most microprocessors there exists at least an assembler. The compiled program is loaded together with the interpreter into the target machine and then executed by simulating each host instruction. Figure 2.2 expresses this method using T-diagrams. The host is the Lilith (Li) while the target machine is the NS32000 processor (NS). The language to be implemented is Modula-2 (M2).

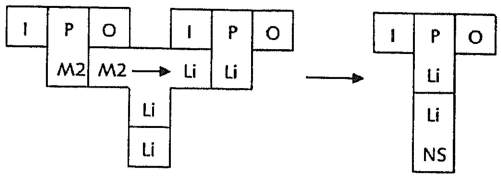


Figure 2.2 Language Interpretation

Obviously efficiency is bad compared to native code but this is the fastest way to get programs running on a new target machine, especially when the host code model is rather simple, like the Pascal P-code or Lilith's M-code. However, this method has a major drawback, because the advantages of a better hardware and a comfortable instruction set cannot be exploited when there are significant differences between the host and target architectures.

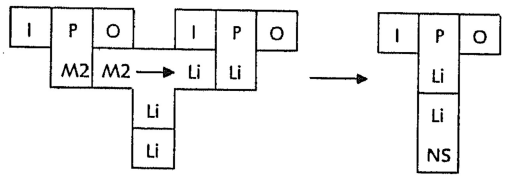


Figure 2.3 Native Code Generation

The second method needs a change of the compiler's backend to produce code for the target machine. The compiler may produce either assembly code or binary code (Figure 2.3). Although the first alternative needs one more step and assumes the availability of an assembler it supports much better the task of debugging the generated code. However, the same can be achieved by having a decoder for the target machine's native language.

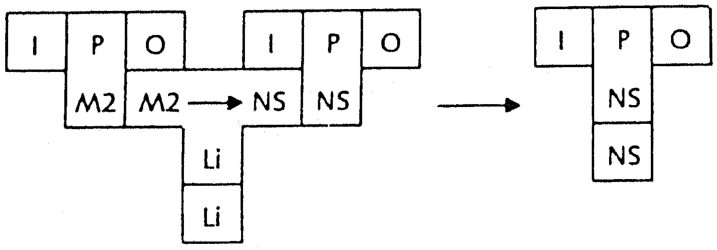
In our project we followed the second method. The backend of the one-pass compiler for the Lilith has been changed to produce NS binary code instead of M-code.

M and translates a program written in *L1* into a program written in *L2*. The figure 2.1c denotes an interpreter running on a machine *M'* interpreting instructions of a machine *M*.

2.2.1. Making the Language Available

The first step is to make the implementation language available for the new hardware. There are different methods to achieve this goal. The first method uses a compiler that is already present on the host machine. On the target machine an interpreter is needed for the code produced by the compiler. This interpreter is mostly written in the machine language of the target machine. Note, that for most microprocessors there exists at least an assembler. The compiled program is loaded together with the interpreter into the target machine and then executed by simulating each host inst

Correction:
the following figure is to be inserted for Fig. 2.3 on page 5



machine. The compiler may produce either assembly code or binary code (Figure 2.3). Although the first alternative needs one more step and assumes the availability of an assembler it supports much better the task of debugging the generated code. However, the same can be achieved by having a decoder for the target machine's native language.

In our project we followed the second method. The backend of the one-pass compiler for the Lilith has been changed to produce NS binary code instead of M-code.

2.2.2. Making the Program Portable

Assuming that code can be produced, another not less important step is to make the program portable. Of course one may say that the use of a high-level language alone guarantees this quality of a program, but most language implementations have some hidden pitfalls. The more a program is adapted to its original host machine, the more pits may open up. Although the problem of the hardware dependencies like wordlength, byte order and arithmetic etc. exists with most programs, the level of abstraction in a system is a measure for the degree of these dependencies. While a program doing simple I/O and arithmetics is easy to transfer the porting of an operating system seems to be a harder task. Hence an operating system makes heavy use of the specific resources and facilities of the host machine to execute its tasks as efficient as possible. A commented list of the problems mentioned above as well as guidelines to achieve portable programs can be found in [Tan83].

Modula-2 itself has some constructs which allow to make assumptions about the underlying hardware. The following list sketches some of them:

- all type transfers and especially the relations of the standard types to *SYSTEM* types like *BYTE*, *WORD* and *ADDRESS*
- *BITSET* and the assumptions about the exact location of the highest and lowest bit, when interpreting a *BITSET* as an integer value
- variant records, especially untagged variants, where assumptions are made about the memory allocation of the record fields
- the order of bits and bytes in memory, especially in type transfers
- value ranges and object sizes
- possible inline instructions within the code, e.g. code procedures on Lilith

One can say that the above step can be omitted or at least minimized, when software is designed from the first step with porting in mind. This is unfortunately not true for an operating system, because it is itself a basis for further implementations and is expected to be as efficient as possible. However, the dependencies can be isolated in the lowest levels of the operating system and are thus easier to exchange for a new target machine.

Concerning higher level programs written in Modula-2 a lot of effort has been invested in creating a library standard [Mod85, Hei86]. Today no such standard is widely accepted, because there is no agreement about the features to incorporate.

3. The Medos-2 Bootstrap

This chapter describes the bootstrap of the Medos-2 operating system onto the Ceres workstation as a an exercise in porting a large program. The first section gives a rough specification of the Ceres. The second section describes the tools needed for the bootstrap, while the last gives a historical overview over the project.

3.1. Target Machine

3.1.1. The NS32000 Processor

The heart of Ceres is a NS32000 series CPU manufactured by National Semiconductor. The main advantage against its commercial competitors is its architecture and the corresponding instruction set [NSC83]. It has some properties that are heavily used by programs defined in a high-level language [Wir86]. In addition to the usual registers like program counter (PC), condition code register, general purpose registers, and stack pointers (SP) it incorporates base registers for global data (static base, SB) and for local data (frame pointer, FP). The latter is used in procedural languages to address local variables and parameters.

In most systems separate compilation of modules, which is an integral property of Modula-2, leads to a separate linking phase prior to program loading. This is not true for the NS32000 processor, because it supports the separation of programs into modules. Each module has its own environment defined by a *module descriptor*. It contains the base addresses of data, code and link information of the corresponding module. The address of the current module's descriptor is stored in the processor's module register (MOD).

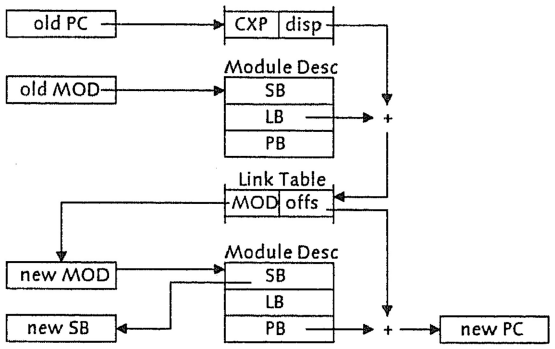


Figure 3.1 Call of an External Procedure (CXP)

The binding between modules is done at load-time by using the link tables, which are constructed by the loader. Therefore there is no need for fix-ups inside the code segment. This mechanism of accessing external objects is used for variable access as well as for procedure calls across module boundaries. The instructions for calling external procedures and return form external call provide this mechanism including reloading of the involved CPU

registers (MOD, SB). Figure 3.1 shows the memory locations and registers involved in an external procedure call.

In general the processor offers an orthogonal architecture with respect to the instructions and the addressing modes. Two addressing modes are especially well suited for high-level language programs: the indexed and the so-called memory relative addressing mode. The former is used for array access, the latter has two displacements and is especially useful to access record fields via pointers.

3.1.2. The Ceres Workstation

This paragraph introduces the Ceres workstation. It only sketches some of its highlights to give the reader a notion of its internal structure. A detailed technical description of Ceres is given in [Ebe87].

Ceres is a single-user workstation. It incorporates a 32 bit CPU (NS32032), primary and secondary memory, user-I/O devices and some communication devices. One of the design goals of the Ceres hardware was its extensibility. Therefore Ceres is a bus-system with a motherboard. Six slots are available, the basic system occupies three of them.

The NS32032 chip is the main processor of Ceres. It is accompanied by the Floating-Point coprocessor and the Memory-Management Unit (which is not used by Medos). The address range of the processor is 16MByte. The system operates at a clock rate of 10MHz.

The main memory consists of up to 256 kByte ROM, at least 2 MByte dynamic RAM and 256 kByte VRAM. The latter is used as refresh memory for the raster scan display. The boot loader and some hardware diagnostic programs are stored in the ROM. Secondary storage consists of a 40MByte (formatted) 5.25" winchester disk drive and a 3.5" floppy-disk drive (1 MByte unformatted).

The user-I/O devices are a keyboard, a mouse as pointing device and a high resolution raster scan display. The display has a resolution of 800 lines and 1024 dots per line (landscape format) and a refresh rate of 62 frames per second. This leads to a flicker-free screen. There is no hardware support for the bitmap operations, everything is done by the main processor.

For communication purposes Ceres has one RS-232-C and two RS-485 serial ports. The latter provides a transmission speed of up to 234 kbit/s. They can be used to implement a low cost network.

3.2. Development Tools for the Bootstrap

3.2.1. Overview

The components of the Ceres Cross Development System, i.e. the tools that are needed to produce programs on the Lilith which are then executed and tested on the Ceres are shown in Figure 3.1. Each box in the figure represents a tool, the arrows indicate the flow of information. The labels show the types of files exchanged between the tools.

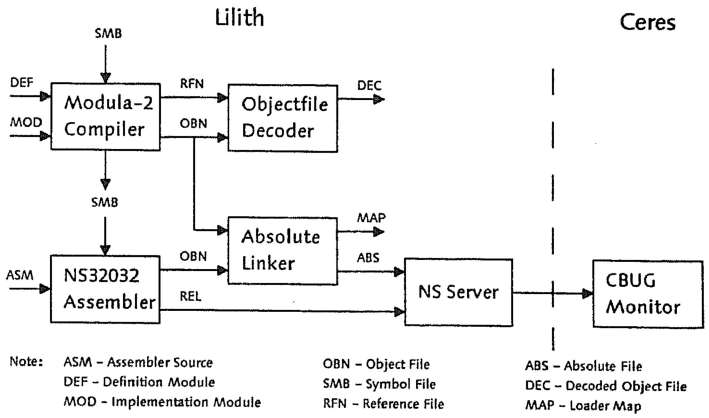


Figure 3.2 Ceres Cross Development System

The software for Ceres may be written in either assembly language or in Modula-2. While the latter is the major implementation language the former is needed to implement time critical program parts as well as program parts that need special instructions not supported in Modula-2. The compiler and the assembler generate a common object file format, that allows the linkage of Modula-2 and assembly objects. Linkage is done on the host system using an absolute linker that reads all objects files belonging to the program, resolves the external references, and creates a binary load image, the so-called absolute file. The binary load image is then loaded into the Ceres memory using the Lilith host server and the Ceres ROM monitor. After loading the program it can be started using the ROM monitor with the Lilith functioning as its terminal.

The following paragraphs give more detailed information about the particular tools mentioned above. The description follows the figure from right to left and bottom-up.

3.2.2. The Ceres ROM Monitor

The Ceres ROM monitor is called CBUG which is an acronym for Ceres deBUG. It is both a versatile program loader and a low-level debugging tool for the development of programs on the Ceres workstation. Of course the test facilities are restricted to the testing of low-level software like the kernel and device drivers in an operating system. Once the higher-levels are

established CBUG stands in the background leaving the work to more sophisticated debugging tools.

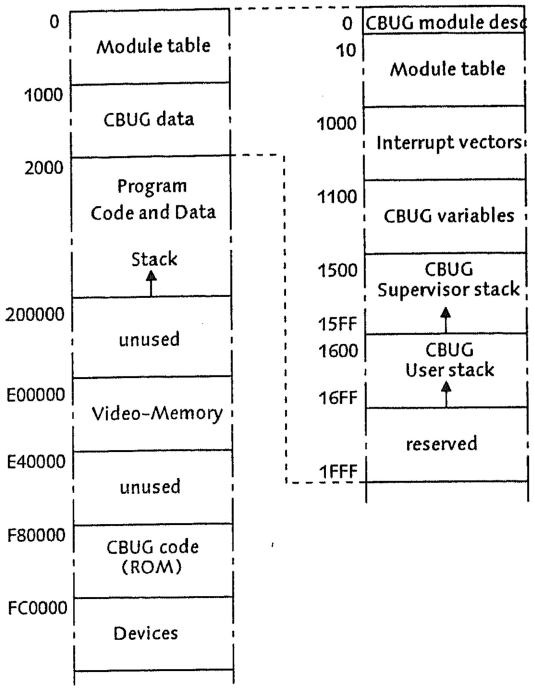


Figure 3.3 CBUG memory layout

CBUG can be seen as a quite simple operating system. It provides a small and comprehensive set of commands that allow testing of both assembler and Modula-2 programs. It contains rudimentary device drivers for the keyboard, the external RS-232-C interface, a general purpose parallel interface and the disk. The latter is used by the monitor exclusively, while the other devices are accessible to user programs through a small set of routines. Note that the parallel interface was implemented in the Ceres prototypes only.

The memory may be inspected as a sequence of bytes, words or doublewords. CBUG allows to access all processor registers as well as those of the coprocessors, namely Floating-Point Unit (FPU) and Memory Management Unit (MMU).

Programs can be downloaded either from the serial or the parallel interface in a special format that is produced by the absolute linker. Additionally, programs may be written as bootfiles onto the winchester disk or a 3.5" floppy disk. When a program traps or an interrupt occurs control is transferred to CBUG. After having inspected and/or changed the registers and/or memory a program may be resumed.

The monitor assumes a memory layout that is shown in Figure 3.3. CBUG allows for max. 127 modules whose descriptors are located in the module table. Each loaded module has a certain layout that is described in more detail in the paragraph about the absolute linker. The module table contains the module descriptors as described in 3.1.

The private data of CBUG contain variables of the monitor, the interrupt vectors [NSC83] and two stacks, one for the user mode and one for the supervisor mode. The actual program starts at address 2000H. The stack for the user program grows downwards and begins therefore at the highest physical address.

The I/O routines of CBUG mentioned above are callable using the CXPD instruction with ROM based procedure descriptors. They provide a functionality similar to those of the module *Terminal* in Medos-2. The routines are callable only from assembler programs because they do not have the same entry/exit conditions as Modula-2 procedures. A list of the routines together with the specification of their entry/exit conditions can be found in [Wil86].

3.2.3. The Host Server

An important companion of the ROM monitor is the host server program. It is subsequently called *NServer*. During the development of the Ceres hardware this program has provided several different services. Among them are

- terminal for CBUG
- program loader
- device simulation

The first and most important service has been that of a terminal to communicate with CBUG. Note, that a first version of Medos-2 used this terminal as its output medium due to the lack of a screen. A service of no less importance is the loading of programs. Based on a simple protocol NServer transfers programs from the Lilith's disk to Ceres using the fast parallel link or the slower RS-232-C. Programs are either transferred directly into Ceres' memory or written onto the bootfile on hard disk or floppy disk.

The development of the Ceres hardware has been done stepwise starting with the CPU-board. Due to this fact some devices had not been operational when they were needed, e.g. the disk and the display. For these cases the NServer provided the Lilith devices to Ceres using the two communication interfaces. Generally, the RS-232-C has been used for sending control information while the higher bandwidth of the parallel interface has been used for the bulk of data transfers.

An interesting example for the device simulation has been the display. All graphical operations on Ceres are done by the CPU in a memory mapped bitmap. To test the raster operations the bitmap was simply placed into the already available RAM. After performing a screen operation the test program sent the whole bitmap to the NServer where it was prepared and displayed in a dedicated window on the Lilith screen.

3.2.4. The Ceres Assembler

The Ceres Assembler was first designed to produce relocatable code sequences [Wan85]. This was needed for writing the first short test programs and, above all, the CBUG ROM-Monitor. The assembler implements the complete instruction set (excluding the custom slave instructions) and all addressing modes.

In contrast to Lilith, Ceres has no special instructions for process switching and bitmap operations. On Ceres, this has to be done in software. Although Modula-2 is a system-implementation language, these operations should or even must be written in assembly language. The coroutine management operations as well as the interrupt/trap handler part need certain processor-specific instructions that cannot be generated by the Modula-2 compiler.

Therefore, we had to solve the integration of assembler-written code into the Modula-2 world. We have not done this by introducing special code procedures consisting of a sequence of numerical constants (as on Lilith), or by introducing an INLINE procedure exported by the pseudo-module SYSTEM (as in the MC68000 compiler). We decided to extend the assembler in such a way that an implementation module can be defined in assembler code. This preserves the distinction between interface description and realization of a module. Assembler modules have an ordinary interface description (definition module as well as symbol file). Therefore our integration of assembler programs maintains type checking of Modula-2. Furthermore the programmer is forced to separate this low-level programming from the Modula-2 code.

By introducing new pseudo-ops the assembler supports the programmer to write such modules. An assembler module starts with a Modula-2 module header. If it is an implementation module the assembler reads the corresponding symbol file and inserts all declared objects in the local symbol table. Therefore the names of these objects can be used in the assembler program. A code sequence representing an exported procedure is labelled by its procedure name. Additionally the programmer can define an import list. This is mainly used for data definitions of imported objects. Record fields can be used as offsets to base addresses. Due to the lack of a WITH-statement all record fields are known in qualified mode only.

The pseudo-ops *INIT* and *ENDINIT* denote the entry and exit point of the initialization part of an assembler module. The assembler produces the same entry/exit code as the Modula-2 compiler. *STATBEG* and *STATEND* are the brackets for the variable declarations. All data-objects defined inside that section are referenced via the SB-register.

For more infomation about the assembler the reader is referred to [Pes87].

Nevertheless, our intention was not to introduce "seperate compilation of assembler programs". Assembler modules should only be used if it is impossible to express the program in Modula-2 or if there are genuine performance reasons.

3.2.5. The Modula-2 Cross Compiler

The Modula-2 Cross compiler is a member of the single-pass compiler family introduced in 1985 [Wir85]. A comparison with the compiler for Lilith and for the MC68000 is done in [Wir86]. For this reason the description is restricted to particular features of the NS compiler.

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.

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 format is

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

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.

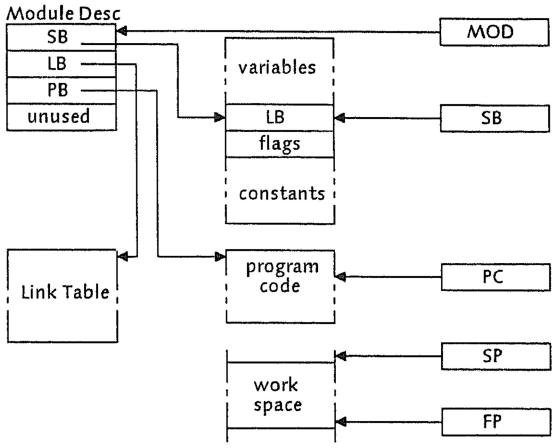


Figure 3.4 Runtime Allocation of a Module

Ceres uses byte addressing. However, data are transferred to and from memory in 32-bit words. The compiler aligns data such that the number of memory accesses is minimized. Each type *T* has an alignment factor k. Variables of type *T* are aligned by the compiler to lie at an address *a*, such that *a* MOD *k* = 0.

Each Modula-2 module is represented in one module in the sense of the NS architecture as shown in Figure 3.4. The module's variables and constants are accessed via the SB-register. The FP register is the base address for local variables.

The compiler generates an object code file with 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 *BlockSize* 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.

3.2.6. The Ceres Absolute Linker

The object files generated by the Modula-2 compiler and the assembler are not yet executable. Even a standalone module must first be brought into a form that allows the processor to load and execute it. The tool that is used to make an executable core image out of one or several object files is called the *absolute linker*. As input it takes the object files of all the modules that belong to a program and binds them. It resolves the external references and computes exact locations for code and data of each module. The generated output is an absolute core image of the program, i.e. merely a one-to-one image of the program in memory. Furthermore, the absolute linker optionally generates a load map containing the sizes and addresses of all loaded modules and an import reference dictionary. In the following paragraphs we give a rough description of the linker's data structures and sketch the algorithm used to link a program.

The data maintained by the absolute linker is contained in two global tables called the *module table* and the *import table*. Internally modules are identified by a number, the so-called *module number*, which is the index of an entry within the module table. Basically, an entry in the module table contains a module name and key and some other information, depending on the module's current state, i.e. loaded or to-be-loaded. A module with state to-be-loaded has a list with the numbers of all referencing, i.e. importing modules, such that forward calls may be fixed. A loaded module's entry contains the exact, i.e. absolute, addresses of code and data as well as a *link table* and an *entry table*. The latter two tables contain the module's external link information and the entry offsets of the exported procedures. The information in these tables is created from the corresponding blocks in the object file. For a description of the object file format we refer to the previous section. The import table always contains the numbers of those modules that are imported but not yet loaded. It is filled from the import block in the object file. At the end of the linkage process the import table is empty.

The algorithm to link a module is straight-forward. It reflects the structure of the object file, i.e. it reads all blocks sequentially, filling its tables with the appropriate information. In the following we give a skeleton of the linking algorithm.

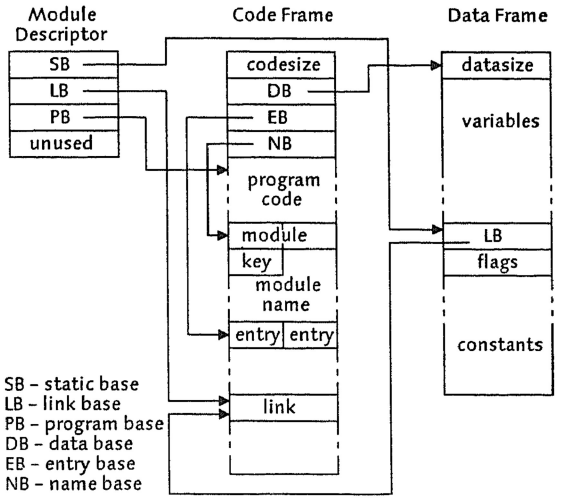
```
PROCEDURE LoadObjectFile(VAR mnr: ModuleNumber);
   :
BEGIN
   ModuleHeader(mnr); Imports(mnr);
   EntryTable(mnr);
   AddressComputation(mnr);
   SatisfyReferences(mnr);
   LinkTable(mnr);
   CodeHeader(mnr); DataHeader(mnr);
   WHILE ~ eof DO
      IF codeBlock THEN Code(mnr)
      ELSIF dataBlock THEN Data(mnr)
      END
   END
END LoadObjectFile;
```

ModuleHeader reads the information contained in the header block and creates a new entry in the module list if necessary, returning its number in the parameter *mnr*. *Imports* reads module names and keys in the import block. Each imported module is searched first in the import list and second among the already loaded modules. When a module is not found, a new module entry is created and its number is inserted into the import table. During separate compilation, each imported module gets a local module number. *Imports* establishes the mapping from local to global numbers. *EntryTable* reads the entry block and creates a table that is attached to the module table entry. *AddressComputation* determines the location of a module's code and data and prepares the module descriptor. *SatisfyReferences* follows the list of references attached to the module list entry and fixes the link tables of the contained modules. *LinkTable* reads the link block of the object file and expands it to form a link table as defined in 3.1. All entries referring to already loaded modules are fixed. *CodeHeader* and *DataHeader* produce the module environment desc and linker tables as well as the header of the data frame (see Figure 3.5). *Code* and *Data* copy the contents of the corresponding blocks directly to the core image file. No modification of the code is necessary because all information for accessing external procedures and variables is contained in the link tables.

During linkage a file is created, the so-called *ABS-file*. The ABS-file contains a compressed form of the core image. In fact, it is a sequence of blocks containing a 32-bit absolute address, a 16-bit byte count and the actual data to be loaded at the prespecified address. The last block in the file has a byte count of zero. This format has two advantages. On the one hand it allows an efficient storage of the core image, because the uninitialized data must not be part of the ABS-file. On the other hand it enables the linker to write the ABS-file strictly sequentially, because the output of yet incomplete information like the linktables may be delayed.

The absolute linker generates a special memory layout for a loaded module that is described below. Generally a loaded module consists of two memory frames as in the Lilith implementation, namely a *code frame* and a *data frame*. Whereas the code frame contains a module's code and linker information, the data frame consists of both initialized and

uninitialized data, i.e. variables and string constants. The structure of the two frames is shown in Figure 3.5.



Note: All tables are double-word aligned

Figure 3.5 Frame structure of a loaded module

Each module has a 32-byte descriptor consisting of the *NS module descriptor* which is located in the module table of CBUG (see section 3.2.2) and an *environment descriptor* of 16 byte which is located immediately in front of the module's code.

The NS module descriptor contains three pointers used by the processor to perform external procedure calls and external data access. The pointer *SB* points to the static data of the module, i.e. the variables and constants. The pointer *LB* holds the address of the module's link table. The third pointer *PB* points to the start address of the code of the module. The last double word in the descriptor is not yet used by the processor. We leave it unused to be compatible with further versions of the processor.

The *code frame* consists of the environment descriptor, the module code, the module name, the entry table and the link table. All parts except the code are necessary to allow the dynamic linkage of modules to the operating system or other already loaded modules. The pointers EB and NB in the environment descriptor support this task by giving fast access to the module name and to the entry table. The code frame size and the pointer DB, as well as the data frame size are needed to later release the frame space used by a module.

The *data frame* contains the data of a module divided into uninitialized data and initialized data. Both kinds of data are accessed using the pointer SB register as base. Unitialized data has negative offsets while constants are addressed using positive offsets. The LB pointer is

used to access external variables. The flags contain e.g. a module initialization flag that is set upon execution of the module's main part.

The described memory layout is identical to that of Medos-2. It enables the dynamic linking capability by storing the essential linkage information together with the module. Part of the above format is also known to the ROM monitor, i.e. the location of the module name. This allows to give meaningful error messages when a trap occurs that is not handled by the user program.

3.2.7. The Object File Decoder

The object file decoder was originally designed to be a help for writing the code generator of the cross compiler. Thus the first version simply analyzed the object file and disassembled the generated code. During the porting of the first Modula-2 programs onto Ceres it helped to find the reasons for certain runtime errors. However, the larger the ported programs the more clumsy it was to find the place of an error in the source file. Thus the decoder has been upgraded to include source information by using the compiler-generated reference files. Note, that these files contain so-called *ref points* that provide a mapping between code addresses and source positions. The ref points are also used by the Medos-2 debugger *inspect* to find the location of an error in the source. Although the decoder does not provide the interactive capabilities of the debugger, it turned out to be a useful tool for the cross development.

3.3. From the First Program to Medos-2

In order to describe the process from porting the first Modula-2 program up to the Ceres version of the Medos-2 operating system, we present a table that shows both the development of the Ceres hardware and the milestones of the software development.

Date	Hardware	Software
spring 1984	start of CPU-board development	
summer 1984	CPU-board with NS32016	small ROM test programs for the CPU-board
	RAM-board with 256 kByte	CBUG 1.0 for downloading programs via RS-232-C
	parallel interface (16-bit)	faster program loading first version of the cross compiler for Modula-2 first version of the absolute linker porting of DiskPatch as first reasonable Modula-2 program using the Lilith-disk with NServer
autumn 1984	adding the disk controller	DiskPatch working on local disk Kernel for coroutines and interrupt handling CBUG upgrade allowing to abort and resume programs and to inspect the memory and the registers

In winter 1984 the 16-bit prototype of Ceres was operational with the exception of the display and the mouse interface. In December 1984 we started to port the Medos-2 operating system, beginning with its file system.

Date	Hardware	Software
Dec. 1984 - Jan. 1985		porting & testing of the Medos-2 file system
Feb. 1985		I/O drivers for keyboard & display
		porting Medos-2 Processes and Programs
March 1985		testing the raster operations with NServer on the Lilith
		porting the Medos-2 program loader
April 1985		first version of Medos-2 without display
May 1985	display controller & mouse interface available	porting of the Debugger and the Modula-2 compiler
June 1985		porting of the Medos-2 utilities and the Maple file system
July 1985		improved version of Medos-2

All steps have been accompanied by a continuous improvement of the tools described above. It must be mentioned that we did not have any debugger until the Medos-2 operating system was fully available, i.e. by the end May 1985. Program testing has been done solely with the ROM Monitor CBUG and an improved object file decoder that incorporated source code.

4. Characteristics of Medos-2 on Ceres

4.1. System Overview

Before we start to describe the particular implementation of Medos-2 on the Ceres, we will give an overview of its structure. We present a view somewhat different from that presented in [Knu83]. It is not primarily based on the import/export relations but on a logical grouping of modules that implement certain parts of the whole system. Dependencies are shown only between the modules of the same logical group. With these assumptions the structure of Medos-2 is described in Figure 4.1.

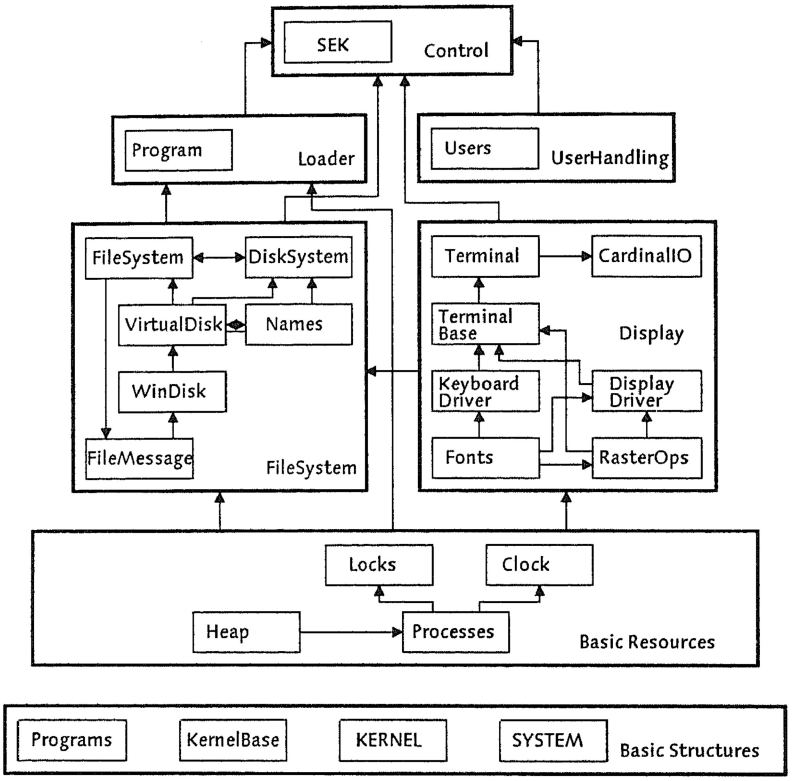


Figure 4.1 Medos-2 Overview

Note, that there are no arrows leading from the lowest module group to the higher ones. This is in consideration of the fact that just these imports from the lowest level of a program make a module graph unreadable. Thus we decided to leave these arcs out, saying that there

may be imports from the lowest level in any of the groups above. The subsequent paragraphs follow the structure in Figure 4.1 in bottom-up manner.

4.2. Basic Structures

The current version of Medos-2 was designed for the Lilith [Knu83, Sch85]. Some of its parts, especially the implementation of processes and interrupt handling depend on the special instruction set of this machine (M-Code) and its hardware architecture [Jac82]. (The notion *process* is used in this chapter in the sense of a Modula 2 process, i.e. of a coroutine). The purpose of this chapter is to describe the implementation of these features as a basis for Medos-2 on Ceres.

The module *KERNEL* described in this chapter is normally not used by programmers. Its description is included to show the way concurrency is implemented on Ceres only.

At first we should have a look at the features of the Lilith architecture which support handling of processes:

- P-Register

  This processor register is a pointer to the process descriptor (*pcd*) of the currently running coroutine.

- TRANSFER

  This M-Code instruction performs a switch from the current to a dedicated process, given by the address of a process variable.

- interrupt handling mechanism

  On occurence of an interrupt the system performs a transfer operation to the interrupt handler process. The address of the process variable of the interrupted process is stored in the interrupt vector.

Because Lilith is a stack machine, it has to save only a few special registers on switching between processes. Due to this fact the transfer operation is very efficient. Interrupts can be handled by a process switch. The interrupt vector entries are not addresses of routines but of process variables. All these features are not directly supported by the NS32000 processor. Subsequently we describe their implementation on Ceres.

On occurence of a process switch the complete status of the interrupted process has to be saved, i.e. all relevant registers must be saved in the *pcd* (see Figure 4.2). In our case this incorporates all registers of the NS-chip set with the exception of

- INTBASE because it is a global value in Medos-2
- SB because it will be automatically updated after realoading the MOD register
- MMU-registers because they are not used in Medos

There are three additional entries in each *pcd*. In the first entry (error) the reason for termination of a process is stored. The second entry (mask) holds the current interrupt (priority) mask of the process. The third entry is the stack-limit, i.e. the upper bound (lowest possible address) of this stack.

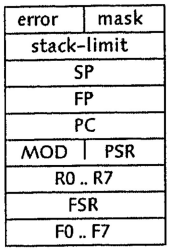


Figure 4.2 Image of a Process Descriptor

In general programs are executed in user mode. Only the routines for process switching and changing of the process priority are executed in supervisor mode. These routines are collected in a monitor, and they are uninterruptable. For that reason there must be only one global supervisor stack.

These monitor routines are called via a supervisor call instruction (SVC). The value of the byte following the instruction determines the called routine. The possible parameters of such a call are on top of the caller's stack (like in ordinary procedure calls).

```
VAR currentP: PROCESS;
PROCEDURE NEWPROCESS(p: PROC; a: ADDRESS; size: LONGINT; VAR pr: PROCESS);
PROCEDURE TRANSFER(VAR from, to: PROCESS);
```

KERNEL exports the type *PROCESS* and the procedures *NEWPROCESS*, *TRANSFER* formerly exported by the pseudo-module *SYSTEM*. The variable *currentP* plays the role of Lilith's P-register. Due to the Modula-2 compiler register allocation strategy the general purpose registers (including the FPU-registers) are not saved.

```
PROCEDURE IOTRANSFER(VAR from, to: PROCESS; dev: CARDINAL);
PROCEDURE SetDriver(p: PROC; dev: CARDINAL);
PROCEDURE Trap(error: CARDINAL);
```

There are two different interrupt handling schemes. The program can choose whether it installs an ordinary interrupt handler or a coroutine which will be resumed on occurence of the desired interrupt. Note, that the first method is only possible with an assembler-written handler.

After calling *IOTRANSFER* an interrupt dispatcher is installed in the corresponding entry of the processor's interrupt vector. The dispatcher performs the desired process switch. The choice between the two ways of interrupt handling is done individually for each device. There is no protection against overriding of handlers; this coordination has to be done at a higher level.

All error traps of the CPU result in an interrupt of the pseudo device 7. Call of procedure *Trap* terminates the caller with the specified error code, i.e. it is "interrupted' by device 7.

4.3. Basic Resources

4.3.1. Storage Management

Due to hardware restrictions Medos-2 offers on Lilith three areas of dynamic memory: stack, heap and frame. Heap and stack area have to reside in the first 64kWords (Lilith addresses words), whereas the frame area is accessed by special instructions to extend the 16-bit address range. The code has to be located in the first 128kWords. This hardware driven restrictions influenced the Medos-2 storage management.

On Ceres, the processor does not force such separations. It has a linear address space of 16MByte. Therefore the frame area is discarded (for compability reasons the module Frames still exists as library module, using the heap). The stack grows from the highest valid memory (RAM) address, the heap starts on top of Medos. Both areas grow against each other. Program code as well as static data are loaded into the heap area.

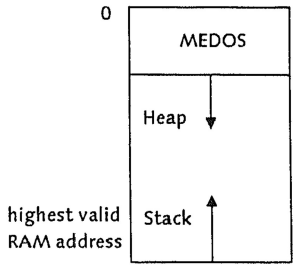


Figure 3.3 Memory Areas

Note, that Ceres has no hardware stack-overflow check. This error is only recognized during a process switch, but prior to its detection the program may have damaged big portions of memory. We felt that in future microprocessor designs a stacklimit register is indispensable for safe computer systems.

The bitmap memory is not managed by the storage management, because it is completely reserved for the system. Programs should access that area only by using the display operations.

4.3.2. Timer handling

Ceres offers a real-time clock chip. Therefore, unlike on Lilith, time need not be computed by counting internal timer interrupts. The module *Clock* is is the standard interface to the clock chip. Due to the clock chip this time has only a granularity of 1 second.

4.4. Display

This section describes the implementation of the display software in Medos-2 on Ceres. The module structure is shown in Figure 4.4.

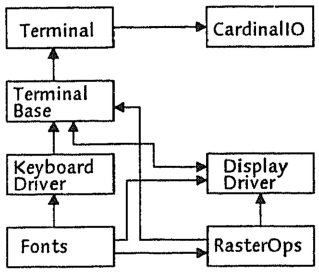


Figure 4.4 Structure of the Display Software

We confine our description to the two low level modules *Fonts* and *RasterOps*. The modules above are nearly the same as on Lilith [Knu83]. The next paragraphs describe the graphical objects and operations and the handling of fonts. They are followed by some remarks about the implementation and a performance analysis of the Ceres raster operations.

4.4.1. Raster Operations

A *bitmap* is a two-dimensional entity that stores boolean informations, so called pixels or picture elements. It is defined by a bitmap descriptor of the form:

```
TYPE BitMap=RECORD
                bmPtr: ADDRESS;
                width, height: CARDINAL
             END;
```

The field *bmPtr* contains a pointer to the bitmap's data. The field *width* specifies the length of one line in bytes, while *height* is the number of lines. A bitmap may have any dimension with the exception that the width must be a multiple of 4. Note, that a double word (4 byte) is the basic unit handled by the raster ops due to the data bus width of 32 bits. A bitmap may lie anywhere in memory. The bitmap which representation lies within the current refresh memory of the display controller appears on the screen.

A bitmap defines the first quarter of a cartesian coordinate system. The coordinates are in the range 0..*width*-1 and 0..*height*-1. When a bitmap is displayed the origin is in the lower left corner of the screen. The pointer *bmPtr* points to the upper left corner such that the address of a certain pixel (x, y) is computed by

```
address := bmPtr + (height - y - 1) * width + x DIV 32
bitoffset := x MOD 32.
```

A *Block* defines a rectangular area (with sides parallel to the coordinate axes) within a bitmap by specifying its lower left corner and the width and height in pixels.

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

A *pattern* is a small area of raster data. Its height is fixed to a maximum of 16 lines while a lines contains a maximum of 32 pixels. A pattern is appropriately described by

```
TYPE Pattern=RECORD
                pheight: CARDINAL;
                pat: ARRAY [0..15] OF LONGINT
             END;
```

The representation of the pattern, contained in *pat* starts in the upper left corner of the area, i.e. the bit 0 of the doubleword *pat[0]* contains the leftmost pixel on the topmost line. Only the doublewords 0..(*pheight* - 1) MOD 16 contain valid pattern data.

The basic operations on objects presented above:

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

Each operation specifies a source, a destination, and a mode, i.e. logical operation to combine the two operands. The current implementation knows four different modes:

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

The operands are combined as follows:

mode	result
replace	dest := source
paint	dest := dest OR source
invert	dest := dest XOR source
erase	dest := dest AND NOT source

For convenience two additional operations have been implemented. The first is called *SCR* and represents an optimized, special case of *BBLT*, namely the vertical scrolling of a block within a bitmap. The second operation *LIN* draws arbitrary lines within a bitmap. For lines that are parallel to the coordinate axes it is an optimization of *REPL*. The two operations are specified by

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

Note, that the sign of the parameter *lineHeight* in *SCR* controls the direction of the scrolling, i.e. up/down for a positive/negative value.

All operations clip their result into a predefined rectangular clipping region. This region is defined in the bitmap descriptor. For this reason the descriptor shown above gets an additional field. Its complete definition is now:

```
TYPE BitMap=RECORD
               bmPtr: ADDRESS;
```

```
    width, height: CARDINAL;
    clipR: Rect
END;
```

The field *clipR* describes the clipping region as a rectangle defined by its lower left and upper right corner coordinates.

```
TYPE Rect = RECORD xl, yl, xu, yu: CARDINAL END;
```

The choice of this method is primarily based on efficiency. The clipping mechanism should not slow down the case where an operation may be executed directly on the screen. For this reason the test for clipping must be as fast as possible. Using only a block as description of the clipping area implies the expression

```
(x >= clipB.x) & (clipB.x + clipB.w > x) & (y >= clipB.y) & (clipB.y + clipB.h > y)
```

to check if the point (x, y) lies within *clipB*. Note, that for each test the coordinates of the right and upper boundary of *clipB* are recomputed. Using the rectangle *clipR*, the test for inclusion is

```
(x >= clipR.xl) & (clipR.xu > x) & (y >= clipR.yl) & (clipR.yu > y).
```

This is much shorter in execution time compared to the first method, because the additions are removed.

4.4.2. Fonts

Ceres allows for different fonts to be displayed simultaneously. Basically a font is a collection of character descriptions consisting of metric data and raster data.

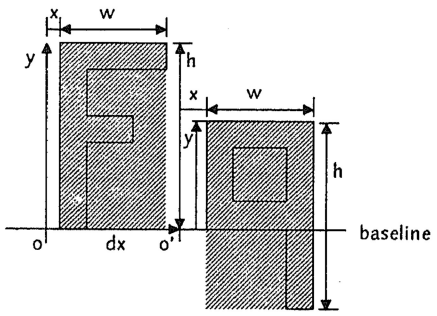


Figure 4.5 Character Box Information

The metric data describes a virtual box that contains the character image and the minimal horizontal distance to the next character. The raster data contains the actual image information, i.e. a raster. Figure 4.5 shows two characters and their box information. Each box defines a local coordinate system for a character. The origin (*o* and *o'*) of such a local

system is relative to the origin of the bitmap coordinate system. The values *x* and *y* are relative to their corresponding origin.

All data for a font is contained in a so-called font file. The font file format is described in EBNF as follows [Gut86].

```
FontFile    = FileType FontName Header Directory MetricData [RasterData].
FileType    = 330C Abstraction.
Abstraction = 0C | 1C.
FontName    = fontID fontType.
Header      = fontHeight minX maxX minY maxY.
Directory   = nofRuns {Run}.
Run         = firstChar nofChars.
MetricData  = {dx x y w h}.
RasterData  = {pattern}.
```

The symbols *fontHeight, minX, maxX, minY, maxY, dx, x, y, w, h* denote two byte integers (low order byte first). The value *Abstraction* tells if the font file contains both metric and raster data (0C) or metric data only (1C). The *pattern* is split into pattern lines describing the character from top to bottom. Each line is made up of $(w + 7)$ DIV 8 byte.

A character is displayed using the procedure DCH of module RasterOps.

```
PROCEDURE DCH(VAR bmd: Bitmap; X, Y: INTEGER;
              f: Font; ch: CHAR; m: Mode);
```

The parameters *X* and *Y* specify the origin of the local coordinate system of the character relative to the origin of the bitmap *bmd* (Figure 4.5). As with the basic raster operations, the parameter *m* defines a logical operation performed between the bits in the character pattern and the pixels on the screen. As stated above the data of a font is contained in a font file. When a font should be used by DCH it must be loaded into main memory. The following procedure manages this task.

```
PROCEDURE LoadFont(VAR fnt: Font; fn: ARRAY OF CHAR;
                   abstr: INTEGER; VAR err: INTEGER);
PROCEDURE UnloadFont(VAR fnt: Font; VAR err: INTEGER);
```

The procedure *LoadFont* creates a memory image out of the font file, which may then be used by *DCH*. The parameter *fn* is the name of the font file, while *abstr* determines whether only metric data or both metric and raster data should be loaded (see above). A font is unloaded using the procedure *UnloadFont*. Figure 4.6 shows the memory layout of a loaded font.

The font header resembles the header information in the font file. The box indices define the mapping from the set of possible character codes, i.e. 0C to 377C onto the set of implemented characters. The implemented characters are represented by their box descriptors. All unimplemented characters map onto the box descriptor 0, i.e. the empty box. Note, that the character code 0C may never be used for a printable character.

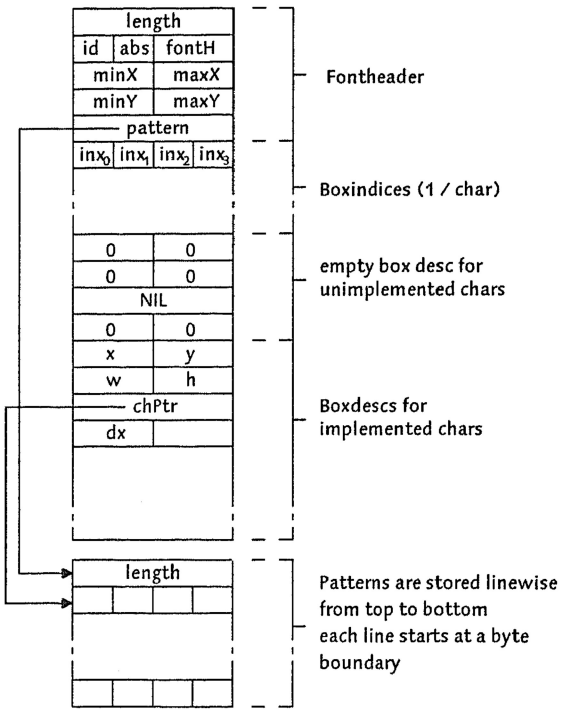


Figure 4.6 Memory Image of a Font

Medos-2 defines a *default font* that is used for all display I/O. Initially this default font is set to the so-called *system font* that is contained in the boot file. However, it may be changed. Some procedures are provided to handle the default font.

```
PROCEDURE SysFont():Font;
PROCEDURE DefFont():Font;
PROCEDURE SetDefFont(fnt: Font);
```

The metric data may be extracted from a loaded font by the following procedures:

```
PROCEDURE ChW(fnt: Font; ch: CHAR):INTEGER;
PROCEDURE ChBox(fnt: Font; ch: CHAR; VAR x, y, w, h: INTEGER);
PROCEDURE FontBox(fnt: Font; VAR fnX, fnY, fnW, fnH: INTEGER);
```

4.4.3. Implementation and Performance

The type definitions and procedures specified in section 4.4.1 and the procedure DCH are implemented in the module *RasterOps*. The font loader, i.e. the procedure LoadFont and the other font operations are contained in the module *Fonts*. *RasterOps* is a module which is

implemented in assembler and benefits from the clean integration of assembly code and Modula-2.

In the following we provide some measurements about the performance of the raster operations on Ceres. Some of the tests have been done both on Lilith and Ceres. For most operations the Lilith is faster. However, a comparison of the speed of Ceres with other workstations that have no special display hardware, namely the Sun/1 and Sun/2 and the Blit [Pik85] shows that our raster ops are at least equally efficient.

Test	Ceres	Lilith
BBLT		
block 512 x 512	130 ms	75 ms
scroll screen	281 ms	163 ms
REPL		
horizontal line l = 700	0.33 ms	0.2 ms
vertical line l = 320	18.6 ms	2.8 ms
cursor pattern 16 x 16	0.82 ms	0.21 ms
fill screen	108 ms	109 ms
DCH		
GACHA14	0.29 ms	0.09 ms
SYNTAX24	0.32 ms	0.21 ms
DDT	73 us	60 us

One major reason that slows down the execution of the raster ops is the lack of a barrel shifter in the NS32032 processor because shifting is an often performed operation. Another reason are the nested loops that are inherent to the raster ops. Note, that the NS32032 processor has no instruction cache but only a 8-byte prefetch queue. While a cache would allow to keep small loops in fast memory, the prefetch queue is emptied on each loop cycle. The effect of throwing out an inner loop has been shown in the comparision of *BBLT* and *SCR*, as well as *REPL* and *LIN*.

Test	BBLT	SCR
scroll block up by 16 lines		
size 1024 x 800	342.0 ms	88.8 ms
size 512 x 800	192.0 ms	55.0 ms
size 256 x 400	56.0 ms	22.3 ms

Test	REPL	LIN
horizontal line l = 700	0.33 ms	0.3 ms
vertical line l = 320	18.6 ms	4.8 ms

Even for small blocks the replacing of the inner loop that copies one line from the source to the destination block by a single MOVS instruction caused speed increases of about 100 %. Another interesting example was made by implementing the scroll operation such that the actual code to copy one line was produced at runtime. This on-the-fly code generation showed nearly the same speed improvements, although additional time was needed to compile the code. A detailed discussion of this method can be found in [Pik85].

The presented model for the raster operations turned out to be useful for porting the Medos-2 software available on Lilith onto Ceres. For applications like the editors *sara* and *lara* which highly depend on good response times the speed seems to be sufficient. But for

other projects like font design or 3D-graphics where more complex graphics are involved it would be worth having a dedicated graphics processor.

4.5. Loader

The program loader of Medos-2 is realized by the module *Program* (see figure 4.1). Basically it has the same task as the absolute linker (see 3.2.6). Its input is the name of a program module to be loaded. The program is linked and loaded to the current environment of already loaded modules. When the whole program is in memory the loader calls the nitialization procedure of the program module.

)uring the loading phase nearly the same data structures and algorithms can be used as lescribed for the absolute linker. The major difference is, that the absolute linker runs as a tand-alone program and may thus use the whole available memory for its tables, because .he linked program is never in real memory. In the program loader both the administrative data and the program code and data compete for the available memory. Thus the emphasis has been put on minimizing the administrative data, i.e. to put as much data to its definitive location in memory and not to duplicate information. In order not to fragmentize the heap, the loader's data is put on the stack and thus vanishes when loading is finished. Basically the same two global tables import list and module list exist. However, due to the above memory constraints the module list contains only information that is discarded after loading the program. *Link table* and *entry table* as well as the module name, module key and the descriptor data are all put into their final locations.

e algorithm to load a module is the same with the exception that the search strategy for ported modules is different. Where the absolute linker searches a module in its module t, the program loader searches in memory using the module table.

1.6. File System

The file system of Medos-2 is implemented by several modules.

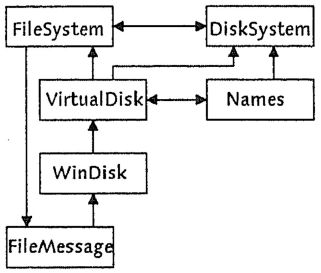


Figure 4.7 File System Structure

The structure shown in figure 4.7 is nearly the same as in the Lilith implementation. The module *FileSystem* provides the programming interface. The modules *DiskSystem*, *VirtualDisk*,

Names and *WinDisk* implement files on the Winchester disk of Ceres. Two major changes of the implementation are outlined in the following paragraphs.

4.6.1. Searching of File Names

One major drawback of the linear directory structure of Medos-2 [Knu83] is the long time needed for searching. Although the separation of file name and file allocation information allows for a dense packing in the name directory, i.e. 16 names per disk block, the checking of 2048 names implies the reading of 64 kByte in the worst case. One way to solve this problem would be a lexical ordering of the file names plus a memory based data structure to make binary search possible.

Our solution to the problem takes into account that a user normally works only on a relatively small subset of his files at one time. Working with the document editor involves one or more document files and a couple of fonts. The development of a Modula-2 program is usually a cycle of calling the editor, the compiler and the debugger. In this case there is also a limited number of files needed. Due to this situation the faster access is realized with a small cache holding the 64 least recently used file names. Each name is represented by a hash code in the cache. When the code for a name is found in the cache the corresponding directory entry is loaded to authenticate the name. The latter is done to detect collisions. The cache is implemented in module *Names*.

4.6.2. Initialization Speedup

One of the most annoying things with the Medos-2 file system is the long startup time, especially when the directory is large. Although the booting of the machine happens only a few times per day, it wastes time. Note that on Ceres the initialization of Medos-2 requires about 30 seconds. The major amount of time is needed for the establishment of the allocation bitmaps of the file system. This includes the reading of all file descriptor, i.e. 512 kByte. One advantage of this strategy is the robustness of the file system against failures. This seems to be adequate in the original environment this concept was designed for. Remember that Lilith had a removable disk cartridge, where a lot of 'hard' situations may cause a failure, e.g. disfigured disk spindle, different temperature conditions etc.

Because of the more trustworthy fixed disk of Ceres we use a different strategy for the initialization. A copy of the allocation bitmaps for file descriptors and disk pages is always stored on the disk. As long as this copy is valid, it is read at system startup instead of rebuilding the bitmaps. The crucial point is the saving of the bitmaps. To be sure that the disk always contains the correct bitmaps they have to be written after each allocation or deallocation of a block or file. Of course this is not feasible because of the enormous overhead of disk I/O. Therefore the most recent bitmaps are always those in memory. The copy on the disk is valid as long as no change occured in the memory bitmaps. Because it is not possible to determine the best moment of creating the copy, the file system is passive in the sense that it invalidates the current copy of the bitmaps upon the first change. The user itself has to initiate a new copy. This needs some discipline when the machine is to be switched off. A special command *shutdown* has to be executed before powering off Ceres. In order to help the user, the command interpreter makes a disk copy of the bitmaps after 3 minutes in the idle state, i.e. when it waits for keyboard input.

4.6.3. File Implementation

Along with the change of the initialization strategy, the administration files, i.e. the directory files, boot files and dump files have been changed in size. The most important difference to Lilith is the absence of back-up directories which saves about 600 kByte of disk space. The names and sizes of the administration files can be found in [Pes86]. Another change concerns the larger disk sectors, resulting in a higher throughput in the disk I/O.

One major complaint about the Medos-2 file system is the limited file size. This problem is not solved in our implementation. The justification for this decision is twofold. On the one hand larger files can be achieved by making the file descriptor larger. On the other hand the basic allocation unit can be made larger.

Making the file descriptor larger is only a fix. The old limit of 192 kByte per file is due to storing all allocation information within the descriptor. Making the descriptor twice as long replaces the old limit by a new one, i.e. 478 kByte. Without sacrificing the redundancy and security of the Medos-2 file system, the number of pages cannot be increased far beyond 16 MByte. All in all, a larger medium requires additional space for more file descriptors. The directory becomes unmanageable larger because no substructuring is supported. The invention of a different directory structure for Medos-2 was not taken into account.

The file system implementation uses only the first 16 MByte of the 40 MByte on the Winchester disk. However, the remaining disk space may be managed in different ways. Generally a new file implementation must be installed using the module *FileSystem*. Currently a test version of *VF* the file system of Vamos, the future multitask operating system is available. VF was originally based on the Maple file system [Ost85] which is used on the file server of the Lilith network. VF is also used to implement files on the 3.5" floppy disk.

Another implementation of files is provided by the program *ramdisk*. The program organizes a part of the memory as a file system. This implementation is thought as an accelerator for program loading. Thus the installed medium is known to the program loader.

5. Conclusions

The operating system Medos-2 has been ported from a totally Modula-2 oriented environment to the new workstation Ceres. Although Medos-2 has been written completely in Modula-2, some facts prevented a simple recompilation of the program.

First of all, Lilith and Ceres have different architectures. As expected, some low-level parts especially device drivers have to be rewritten. The Lilith-instructions for bitmap operations and for coroutine handling are emulated by new modules. This category of problems is inherent to the porting of an operating system.

Another source of troubles are the 'covered' machine dependencies. They are incorporated by the so-called CODE procedures and by type transfers (both too often used on Lilith). Absolute variables lead to another problem, if they are misused for hidden communcation between modules. This scheme was unfortunately used once in Medos-2.

Medos-2 makes heavy use of its knowledge about the allocation of data structures by the Modula-2 compiler. With the new compiler this strategy changed. This influenced especially those data structures that have to fit into a certain amount of memory, e.g. the file descriptors. Reordering of the record fields of the file descriptor helped in this case.

During the project a number of errors have been found in the Modula-2 compiler. Because of the lack of adequate test software some of the errors were extremly hard to find. In our view it seems to be worth investing more time to test the compiler before using it for larger programs. However, some seldom situations occur under real-life conditions only.

Some of the program parts were so critical in execution time that writing them in Modula-2 was not adequate. To avoid scattering of machine dependencies mentioned above a method has been implemented to allow the interfacing of assembler-coded modules with ordinary Modula-2 programs retaining the benefits of strong type checking and structured interfaces.

A number of tools has been built throughout the project that facilitated the cross development of software for Ceres. The most important ones, i.e. the compiler, the assembler, the absolute linker and the ROM monitor are still in use.

After the project has been finished one may ask wether it would have been a better decision to adapt Medos-2 more to the hardware. However, the current solution has shown that it was possible to transport most of the utility programs and the whole cross development software with a minimal effort.

Acknowledgements

We wish to thank N. Wirth for conceiving and guiding the Ceres project and for his comments on this report. He wrote the one-pass Modula-2 compiler for the Ceres. Hans Eberle supported us with the outstanding implementation of the two Ceres prototypes. Svend Erik Knudsen the 'father of Medos-2' helped us with lots of discussions. We thank Jürg Wanner for the first versions of the Ceres assembler and of the raster ops. The handling of fonts on Ceres is a result of discussions with Hans-Ruedi Schär and Jürg Gutknecht. We express our thanks also to Werner Heiz for carefully reading the report.

It was a worthwile adventure to be involved in the development of a new computer from the very beginning.

References

[Ear70] J. Early, H. Sturgis: A formalism for translator interactions
Comm. of the ACM, Vol. 13(10), October 1970, 607-617

[Ebe87] H. Eberle: Hardware Description of the Workstation Ceres,
Institut für Informatik, ETH Zürich, report No. 70, January 1987

[Gut86] J. Gutknecht: The Ceres Font Machinery
Institut für Informatik, ETH Zürich, internal document

[Hei86] G. Heiser et al.:
OSSI - A portable Operating System Interface and Utility Library for Modula-2
Institut für Informatik, ETH Zürich, report No. 67, June 1986

[Jac82] C. Jacobi: Code Generation and the Lilith Architecture
Ph.D. thesis No. 7195, ETH Zürich 1982

[Knu83] S.E. Knudsen:
Medos-2: A Modula-2 Oriented Operating System for the Personal Computer Lilith,
Ph.D. thesis No. 7346, ETH Zürich 1983

[Mod85] Modula-2 Standard Library Definition Modules
in Modula-2 News, Issue #1, January 1985, Modus (Modula-2 Users Association)

[NSC83] National Semiconductor Corp.: NS32000 Instruction Set Reference Manual
Publ. No. 420010099-001A

[Ost85] F. L. Ostler: Maple: A Modula-2 File Server for the Lilith Environment
Ph.D. thesis No. 7936, ETH Zürich 1985

[Pes86] F. Peschel, M. Wille (Ed.): Ceres Handbook
Institut für Informatik, ETH Zürich, 1986

[Pes87] F. Peschel: The Ceres Assembler
Institut für Informatik, ETH Zürich, 1987, internal document

[Pik85] Rob Pike, Bart Locanthi: Hardware/Software Trade-offs for Bitmap Graphics on the Blit
in Software-Practice and Experience, Vol. 15(2), 131-151, February 1985

[Sch85] H. R. Schär (Ed.): Lilith Handbook
Institut für Informatik, ETH Zürich

[Tan78] A.S. Tanenbaum et al.: Guidelines for Software Portability
in Software-Practice and Experience, Vol. 8(3), 681-698, March 1978

[Tan84] A.S. Tanenbaum: Structured Computer Organization
Prentice Hall, Englewood Cliffs, 1984, p. 396

[Wan85] J. Wanner: Assembler und Raster-Operationen für den NS32016
Institut für Informatik, ETH Zürich, diploma thesis

[Wir81] N. Wirth: The personal computer Lilith,
in Proc. 5th International Conf. on Software Engineering,
IEEE Computer Society Press, 1981.

[Wir86] N. Wirth:
Microprocessor Architectures: A comparision based on code generated by compiler,
in Comm. of the ACM, Vol. 29(10), October 1986

[Wil86] M. Wille: CBUG - User Manual
Institut für Informatik, ETH Zürich, internal document