Diss. ETH No. 7346

Medos-2:
A Modula-2 Oriented Operating System for the Personal Computer Lilith

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by
SVEND ERIK KNUDSEN
Dipl. Phys. ETH
born October 19, 1947
citizen of Thalwil (Zürich)

accepted to the recommendation of
Prof. Dr. N. Wirth, examiner
Prof. Dr. C.A. Zehnder, co-examiner

N. Wirth

Zürich 1983

Preface

In the fall of 1977, Prof. N. Wirth started an integrated hardware and software project aiming at the construction of a modern personal computer, a highly desirable tool for the creative software engineer. The principal parts of the so-called Lilith project were the design of the programming language Modula-2 and the construction of a multipass compiler for it, the design of a suitable machine architecture for the execution of compiled programs, the design and construction of a computer capable of efficiently interpreting the defined machine architecture and equipped with the desired peripheral devices, the design and the implementation of the single-user operating system Medos-2, the development of modern text and graphic editors taking full advantage of the computer's capabilities, and the development of a whole range of utility programs and library modules supporting the preparation of documents and the development of programs. After successful prototyping a series of 20 computers was built.

During the first year of the project, I was mainly involved in the programming language part of the project. Since then, I have designed, implemented, and maintained the operating system. This thesis grew out of my participation in the Lilith project and concentrates on the single-user operating system. The thesis shows that it is feasible to implement a substantial single-user operating system in Modula-2, that good support of and by a modern programming language providing separate compilation may considerably influence the design of an operating system, and that the area of personal computing made possible by recent progresses in the computer hardware area will call for well-suited operating systems. Medos-2 is an attempt to create such a system.

I am indebted to Prof. N. Wirth for conceiving and coordinating the Lilith project, for giving me the opportunity to design and implement the operating system Medos-2, and for supervising this thesis.

I also thank Prof. C.A. Zehnder for his advice on this thesis.

Thanks are also going to all my colleagues who parcipated in the Lilith project and thereby contributed to Lilith's success. The thanks are in particular going to L. Geissmann, Ch. Jacobi, and W. Winiger for many valuable discussions and to A. Gorrengourt for the implementation of the file-buffering. I would also like to thank H. Hinterberger and F. Ostler for their careful reading of my thesis.

The patience and encouragement of my mother have, however, above all, made it possible for me to finish this work.

Contents

Appendices

Abstract

Medos-2 is a single-user operating system designed and implemented for the personal computer Lilith. It is an *object-oriented* operating system conceived according to the concept of *open* systems. Its principal parts are the file system, the linking-loader, the part providing a "standard" terminal, and the command interpreter. All basic software for the Lilith computer is programmed in Modula-2. The operating system's interface to programs is therefore presented as a collection of separate Modula-2 modules and the operating system is a memory-resident Modula-2 program.

The purpose of this thesis is to show how Medos-2 provides both an easy to use and comfortable environment for developing and running Modula-2 programs by use of simple but powerful concepts for managing resources, for executing programs, and for handling files. The successful implementation, the efficiency, and the small size of the system are also due to this. The thesis also shows that it is possible to implement a realistic single-user operating system *completely* in Modula-2.

Medos-2 executes programs like "super-procedures". Any running program may activate another program. The linking-loader links a called program to its actual environment and checks the compatibility of separate modules. Both Lilith's architecture and the format of *object code files* contribute to the simplicity of the linking-loader.

Resources are managed by separate modules, typically one module for each kind of resource. A resource (e.g. an open file, main memory space) is *owned* by an activated program. Medos-2 provides several routines helping in implementing resource-managers, which may be non-resident.

The effectiveness of the file system may heavily influence the performance of a computer system. Medos-2's file system allocates disk sectors to files almost contiguously, and makes use of this fact by reading ahead on files. The buffering strategy is different for sequentially and randomly accessed files. This is enabled by simple statistics over file accesses and by the management of buffers by the generalized clock-algorithm.

Kurzfassung

Medos-2 ist ein Einbenützer-Betriebssystem, welches für den Arbeitsplatzrechner Lilith entworfen und implementiert wurde. Es ist ein *objektorientiertes* Betriebssystem gestalltet gemäss dem Konzept von *offenen* Systemen. Die Hauptteile des Betriebssystems sind das Dateisystem, der Linking-Loader, der Teil, welcher ein "Standard" Terminal zur Verfügung stellt und der Commandinterpreter. Alle Grundprogramme für den Lilith-Rechner sind in Modula-2 programmiert. Die Schnittstellen des Betriebsystems für Programme sind deshalb eine Sammlung von separaten Modula-2 Modulen, und das Betriebssystem selbst ist ein speicherresidentes Modula-2 Programm.

Der Zweck dieser Dissertation ist zu zeigen, wie Medos-2 eine sowohl einfach zu benützende als auch komfortable Umgebung für die Entwicklung und Ausführung von Modula-2 Programmen zur Verfügung stellt. Dies wird durch einfache aber wirksame Konzepte für das Verwalten von Betriebsmitteln, für das Ausführen von Programmen und für die Behandlung von Dateien erreicht. Die erfolgreiche Implementierung, die Effizienz, und der kleine Umfang des Betriebssystems ist ebenfalls diesen Konzepten zuzuschreiben. Die Dissertation zeigt ebenfalls, dass es möglich ist, ein realistisches Einbenützer-Betriebssystem komplett in Modula-2 zu implementieren.

Medos-2 führt Programme wie "Super-Prozeduren" aus. Jedes laufende Programm kann ein anderes Programm aktivieren. Der Linking-Loader bindet ein aufgerufenes Programm zu seiner aktuellen Umgebung und prüft die Kompatibilität der separaten Module. Sowohl Liliths Architektur als auch das Format des *object code file* trägt zur Einfachheit des Linking-Loaders bei.

Betriebsmittel werden durch separate Module verwaltt, typischerweise ein Modul für jede Betriebsmittelart. Ein Betriebsmittel (z.B. ein Datei, ein Hauptspeicherbereich) gehört einem aktivierten Programm. Medos-2 stellt mehrere Routinen zur Verfügung, die das Implementieren von eventuell nichtresidenten Betriebsmittelverwaltern erleichtern.

Die Effektivität des Dateisystems kann die Leistung eines Rechners gravierend beeinflussen. Das Dateisystem von Medos-2 alloziert, wenn möglich, benachbarte Plattensektoren zu Dateien und nützt diese Tatsache durch Vorauslesen auf Dateien aus. Verschiedene Pufferstrategie werden für sequentiell und direkt zugegriffene Dateien angewandt. Dies wird ermöglicht durch eine einfache Statistik über Dateizugriffe und die Verwaltung der Puffer mit dem generalisierten "clock"-Algorithmus.

Seite Leer /
Blank leaf

1 Introduction

1.1 Similarities and Dissimilarities of Operating Systems

Medos-2 is a relatively small operating system developed for the personal computer *Lilith* [Wir81]. A clear definition of what an *operating system* is cannot be found in the literature. Terms like *monitor, supervisor,* and *executive system* are essentially synomyms to operating system. Most authors, however, agree to the need of an operating system and argue that the main reason for having an operating system is the need for sharing resources. P. Brinch Hansen wrote about a decade ago, that "an operating system is a set of manual and automatic procedures that enables a group of people to share a computer installation efficiently" [BH73]. At another place it is stated that "the operating system is that part of the software which supports the sharing of resources" [TB74]. The sharing of a computer installation among several (or many) users was, and still is, one of the main problems solved by the operating systems in expensive installations. Personal computers should, however, by definition not be shared among different users, at least not at the same time. Like terminals, they are used exclusively by one person at a time. So, two questions have to be answered before building a new operating system for a personal computer, namely the first: Why is an operating system needed at all? And the second: If an operating system is needed, why is a new operating system needed? The answers to these two questions are also an important answer to the question about the similarities and dissimilarities of operating systems.

The answer to the first question is that an operating system helps in sharing resources in a very general sense. Although the traditional sharing of computers among users is no longer needed for a purely personal computer, resource sharing provided by operating systems in other areas is still very important for the user. A few examples illustrate this: File systems support the storage of many files on a single or a few disks, i.e. many files may share one or a few disks. More or less unrelated and even unreliable programs may be executed one after the other, without hampering each other, i.e. the main memory and the central processing unit are shared by several computations in a convenient way. In most cases however, the sharing of a resource among otherwise unrelated computations makes it necessary to provide some sort of protection in order to guarantee the integrity of the resource. The sharing of secondary storage media among several files makes it possible, for example, to communicate information not only among users but, at least as important, also among different computations of one user. The standardized environment (*abstract machine*) provided by operating systems makes the exchange of programs and routines among users feasible, i.e. the sharing of programs and routines is made possible. A standard environment also hides configuration differences, at least in the most often used aspects. Also, it helps to make software more generally usable. A standardized encoding of information on files increases the usefulness of programs. Many operating systems recommend programmers to encode a text into a sequence of ASCII-characters. As most programmers follow this advice, it turns out that text written by one program (e.g. an editor) may be read by many other programs (compilers, cross reference generators, text formaters, print programs, etc.) and not only by the program itself.

What can be expected from a *new* operating system that no existing one does provide? The worst property of a new operating system is that it is *new*. Old programs have to be adapted or even reprogrammed, new systems tend to be less reliable, etc. Why not adapt an existing interactive (time-sharing) system (e.g. UNIX [RT78]) or a real-time system to the personal computer Lilith? The answer to this question is similar to the answer to questions like: Why not buy an existing mini- or micro-computer system and use it as a personal computer? Why propose and implement still another programming language? Many assumptions made one decade ago when the now existing operating systems were designed have since changed. Most operating systems (and indeed the most successful of them) are special purpose systems in order to provide the desired service in a convenient and economical way. The changed basic assumptions imply that better operating systems might be written for our current (and special) needs. The interface of a specialized operating system and, even to a higher degree, its implementation are of course influenced by the assumptions made for its design. Over the last decade, the size of hardware components and their prices have dropped so much that it is now nearly a must to "give" each serious programmer his own *real* (and personal) computer and not only a *virtual* machine in a centralized, shared computer system. Consequences are that the user may be offered a much better interactive service (shorter response time, "intelligent" responses to each key-stroke, etc.) and that the operating system no longer has to share resources among different users. Developments in programming languages (Ada [Ich80], Concurrent Pascal [BH75], Mesa [MMS78], Modula-2 [Wir82], Pascal [Wir71, JW75], and many others) as well as the fact that each user has his private computer, change the view on the whole programming environment considerably. An operating system for a modern work station should, as an example, no longer merely provide an improved (and hardware protected) assembly language machine as its main building block.

1.2 Development History

A short history of the development of Medos-2 is given here: The main design was done between spring 1979 and autumn 1979 as a part-time task in addition to educational duties and work on the Modula-2 compiler. The programming task started in October 1979, and the first test executed on the Lilith computer was made December 23, 1979. In those days, the reliability of Lilith was very poor. The mean time between two main memory failures was around 10 minutes, the time to load the test-program from a small casette-tape about the same! The first release of Medos-2 (version 1) was in April 1980. In summer 1980, the Modula-2 cross compiler was transported from a PDP 11/35 to Lilith within one week. The second version of Medos-2 was released in October 1980. It had a few minor changes in the interface to the file system. The third release, June 1981, included an improved mechanism for recovery from errors in user-programs. In this period, several machines were equiped with 128 kword main memory. The normal address space of Lilith is, however, only 65536. Medos-2 was adapted not to store the bitmap for the display in the normally addressable part of main memory. Version four of the system was released in June 1982. Its interface to the display was changed, it provided a more general allocation method for main memories larger than 64 kword, and it uses an

improved buffering technique for disk-files. The most recent release of the system (Medos-2 version 4.2) was in December 1982. It supports the identification of users. This appears to be needed if several work-stations are integrated into distributed system by a (local area) network.

Meanwhile, many packages and programs have been developed for Lilith, all based on Medos-2. The list of available programs includes the more common things like editors, compilers, a debugger, and file utilities as well as a relational database system and a new operating system built on top of Medos-2.

2 Medos-2 and the Given Environment

The development of the operating system Medos-2 is merely one part of a larger project: the construction of a relatively powerful personal computer [Wir81]. Roughly speaking, each operating system has to provide facilities which support the desired applications of the target machine, in this case the area of personal computing. Some of the initial goals of Medos-2 are enumerated in the next section. However, an operating system has also to fit well on top of the target machine and has to be well implemented. Two major pillars, on top of which Medos-2 is built, have to be characterized before the design of Medos-2 is discussed in more details, namely the hardware of Lilith (2.2) and the programming language Modula-2 (2.3). Both, the design of the hardware and of the programming language, were parts of the project aiming at the development of the single-user work-station Lilith.

2.1 Goals of Medos-2

When the design of Medos-2 started in the spring of 1979, Lilith was mainly seen as a work-bench computer in a software-oriented computer science department. Two essential application areas were envisaged for a personal work-station in such an environment, namely assistance by the machine in doing repetitive office work such as preparing a document, writing a letter, and sending a message to a colleague, or as a tool for developing, debugging, and executing programs.

The first class of problems may be handled by letting users execute one or several application-specific "standard" program(s) (e.g. editors, formatters, print-programs, etc.). The main advantage of a personal computer for these applications lies in its potentially better user interface compared to that of an ordinary time-sharing system. Shorter response times, more and better information on the screen (graphics), response to each key-stroke, and the availability of a pointing device (mouse) are some of the key-points. In order to have an even better tool at hand for this class of applications, it is desireable to have work-stations connected to each other and to some server computers by a local area network. It is still convenient and more economical to place expensive, large, noisy, and/or rarely used devices like printers, large disk drives, tape units, and communication multiplexers at central locations and share them among several users.

The programmers, whom Medos-2 is intended to help, are mainly computer scientists, software engineers, and students. Such people like to be free in the use of their personal computer. The operating system should not hinder a programmer to program and test even hardware oriented software like a driver for a new interface, should not prescribe a certain lay-out on the display, nor should it fill the memory up with resident code. What is needed is a relatively *small* resident executive system which can load a program from a file on the disk and execute it. No absolute protection mechanisms are necessary. Hardware supported protection mechanisms tend to be of the category allowing "everything or nothing" which appeared not to be of much help if it is desirable, from time to time, to use special or new features of a certain machine. The desired "openness" aspect of the operating system is important in an experimental environment. When Medos-2 was designed, we had, however, no clear idea of the consequences this aspect has on an operating

system, although B. W. Lampson argued for open single-user operating system in 1974 [Lam74]. Pilot, an operating system developed for personal computers at Xerox, is a modern example of a system providing no absolute protection mechanisms [Red80]. Pilot is, however, by no means small.

Relatively early it was also decided that both the operating system and all user programs had to be programmed in the same programming language, namely Modula-2. The rule of having only one single programming language available was not considered to be a restriction. On the contrary, there are many advantages to programming all programs in the same high level programming langague. Here are some examples: Only one compiler must be provided. The operating system calls can be (at least syntactically) checked at compile time. The interface to the operating system can be defined such that there is no distinction between user-written routines, library routines, and operating system provided routines. It was also expected that the module concept of Modula-2 would help in structuring programs into manageable units and would provide a protection mechanism, no doubt circumventable, but with a much finer granularity than the absolute mechanisms usually employed by operating systems.

The small resident executive had to include a file system, simply because the resident loader should load programs from files. The file system must be very robust against all hardware, software, and user faults. This robustness is desirable mainly because the system provides no absolute protection mechanism and because a loss of information on a disk pack often may turn out to be irreversible. Of course, the file system also has to be efficient both in terms of memory space used and in terms of access-times, although, no real-time applications were envisaged for Medos-2. The general performance of a single-user work-station (and of other general purpose computer systems), however, depends heavily on the efficiency of its file system.

Another problem, which had to be anticipated from the beginning, was that several application programs may turn out to be too large to be stored in main memory. The first critical program was the Modula-2 compiler. The compiler was already programmed for a minicomputer with a relatively small address space (PDP 11), and therefore partitioned into several passes. A good support of loading and execution of the compiler passes and of communication between the passes was felt to be adequate from the beginning, especially since Lilith does not support the implementation of virtual memory.

To conclude the section, some key-sentences characterizing the goals of Medos-2 are listed:

- It should be a single-user system.
- It should support the execution of programs written in Modula-2.
- It should itself be programmed in Modula-2.
- It should enable an efficient use of main memory.
- It should be open.
- It should include a robust and efficient file system.

Other operating system topics were explicitly not egoals of Medos-2, for example: Concurreency in user programs were not considered important at the beginning (no real-time applications are supported). Multi-user support and multi-programming

were also not aimed at. The file system did not have to support general data-base applications (e.g. atomic transactions, stable storage, etc.).

2.2 The Target Computer Lilith

The operating system Medos-2 is designed for the Lilith computer hardware. The hardware of Lilith is described in a report by N. Wirth [Wir81]. This section gives a short overview of a typical Lilith configuration and enumerates some of its main hardware characteristics.

The Lilith Hardware

The hardware is a 16-bit machine and consists of a micro-coded central processing unit, a multi-port memory with initially 64 kword(16), interfaces for a raster scan display, a disk drive, a keyboard, a cursor tracking device (mouse), a V-24 (RS-232) asynchronous communication port, and a line clock. (See Figure 2.1) Later, the main memory was enlarged to 128 kword and a controller for a 3 MHz Ethernet-type local area network has been developed for the work stations [Hop83]. Interfaces for a laser beam printer (Canon LBP 10), for an about 450 MByte disk drive (Fujitsu M 2351 A), for a X.25 network, and for other devices have also been developed.

The central processor (CPU) has a 16 bit wide arithmetic and logic unit (ALU) based on four AMD 2901 bit-slice units. A register stack for up to 16 entries supports the evaluation of expressions. The micro-code memory is a PROM-store for 2048 instructions of 40 bits each. The CPU operates at a basic clock cycle of 150 ns, the time required to interpret a micro-instruction. The most frequently occuring macro-code instructions correspond to about 5 micro-instructions on the average. No hardware mechanism has been provided for address translations (virtual addresses to physical addresses) nor is a protection mechanism supported. Devices are not memory-mapped, the interfaces are connected to a separate I/O bus.

The display is based on the raster scan technique using 592 lines of 768 dots each. Each of the 454'656 dots is represented in main memory by one bit. The entire bitmap occupies therefore 28'416 word or about 43 % of the initial main memory. A second type of display providing 832 lines of 640 dots each has also been developed. The bitmap for this display is a little larger (33'280 word), but as the main memory typically contains 128 kword now, the fraction used for the display bitmap is reasonable (22 - 23 %). The representation of each dot (picture element) in the program accessible main memory makes the display equally suitable for text, technical diagrams, and graphics in general. In the case of text, each character is generated by copying the dot-pattern of the character into the appropiate place of the entire screen's bitmap. This is done by software, supported by microcoded routines, corresponding to special instructions. This solution offers the possibility to display characters according to different fonts (i.e. the size, the thickness, the slope, and the style of the displayed characters is changeable by software).

A Honeywell Bull D120 disk drive for removable disks is provided. The capacity of one disk is about 9.8 MByte. The disk cartridges are formatted by the manufacturer with 50 sectors per track and with 392 tracks on each of its two surfaces. 256 Byte

may be stored in each sector. The average rotational positioning time of the drive is 8.3 ms, the average head positioning time about 65 ms.

The mouse is a device that transmits to the computer signals which represent the mouse's movements on a flat surface (e.g. a table). These movements are translated (by software) into a cursor-position displayed on the screen. The accuracy of position may be as high as the resolution of the screen. The mouse also contains three pushbuttons (keys) which are convenient for giving commands after positioning the mouse.

The Lilith Architecture

The instruction set of the Lilith computer is based on a stack architecture. This so-called *M-code* was designed such that it can be generated easily by a Modula-2 compiler (or compilers for other Pascal-like programming languages) and can also be executed efficiently. The efficiency of the execution is partly due to the dense encoding of the instructions and partly due to the fact that the instruction set enables a heavy use of CPU-internal base registers and of the CPU-internal short register stack. The high density of the code is achieved not only by implicit addressing of intermediate results in expressions, but mainly by providing different address lengths and suitable addressing modes. Most addresses in instructions are relatively small offsets to one of the base registers.

Unlike most other implemented architectures, the M-code architecture also supports the execution of programs partitioned into several modules. The word *module* here is to be understood as synonym to compilation unit. A separately compiled Modula-2 module is a typical example. To each loaded module belongs a so-called *data frame* for its global data and a so-called *code frame* for the code of its procedures. (See Figure 2.2) A table at a fixed location in main memory, the *data frame table*, holds the addresses of the data frames of loaded modules. A reference to the corresponding code frame is stored in the first word of its data frame. All modules are accessible via the data frame table. The index of the entry in this table, the *module number*, is used for the addressing of a certain module in the code. M-code instructions that access data in other modules or call of procedures in other modules typically use the module number as reference.

During the execution of a procedure declared in a certain module, the base addresses of the corresponding data frame and code frame are stored in two base registers (called G and F). Code and global data within the procedure's own module may therefore be addressed efficiently by offsets only.

The next instruction to be executed is addressed by the register PC, a byte-offset relative to the beginning of the code frame (i.e. relative to F). A call to a procedure of another module (or a transfer of control from one coroutine to another coroutine) implicitly assigns new values to the registers F, G, and PC.

M-code instructions for procedure calls do not contain the offsets of the procedures' entry points within the corresponding code frame. Instead, an index to the *procedure entry table*, the so-called *procedure number*, is specified in the instruction. The procedure entry table is allocated at the beginning of the code

frame and contains the byte-offsets of the entry points of all procedures in the corresponding module.

Local data of procedures are allocated in a stack of procedure activation records. Each coroutine is allocated a contiguous working area in store, called a *stack frame*. The stack frame contains the so-called *process-descriptor* at its beginning. The rest of the stack frame contains the working space of the process. Four address registers point to the stack frame of the currently executed process. They are called P, L, S, and H. P points to the process descriptor at the beginning of the stack frame (P for Process pointer), L ponts to the activation record on top of the stack (L for Local data), S points to the first free location in the stack(S for Stack pointer), and H points to the upper end of the stack (H for High limit).

The addresses of stack frames are used when control is transferred from one coroutine to another. This so-called *coroutine transfer* may either be explicitly programmed, or be implicitly invoked by an execution error detected by the microcoded M-code interpreter, or be caused by an interrupt. Eight interrupt handlers may be defined, one for each interrupt line. The addresses of the corresponding coroutines (driver processes) are stored in a table, the *interrupt vector*, at a fixed location in main memory and are therefore known to the M-code interpreter. A priority scheme and an interrupt-enable scheme control the way interrupts are handled by the M-code interpreter.

The M-code architecture includes several instructions for machine-specific operations. These instructions may be classified into I/O-instructions, instructions for operations on (screen-) bitmaps, and instructions for moving of blocks in main memory. In the Lilith computer, interfaces are controlled by several M-code instructions (and not by accessing fixed memory locations). Four M-code instructions operate on bitmaps. The desired efficiency of these operations forced them to be microcoded. Data can be moved around in the *whole* main memory by a single M-code instruction.

Three further concepts of the M-code have to be mentioned:

Many instructions use absolute addresses as reference to the accessed data (variables). It is therefore impossible to move data frames or stack frames around in the address space after their loading or creation.

The M-code provides only a direct mapping from virtual memory addresses to physical memory addresses. The M-code provides no absolute protection mechanisms.

Addresses provided by the M-code are 16 bit wide word-addresses. The resulting 64 kword address space is, however, too small to address the whole physical memory which typically contains 128 kword. The consequence is that all data referenced by arbitrary M-code instructions have to reside in the generally addressable 64 kword of the main memory (i.e. in the contigous part of main memory starting at address zero). Only data referenced by few specialized instructions or referenced in a controllable way may be stored at any main memory location. In particular only fonts, bitmaps, code frames, and data not accessed by arbitrary M-code instructions may be stored at any location in main memory.

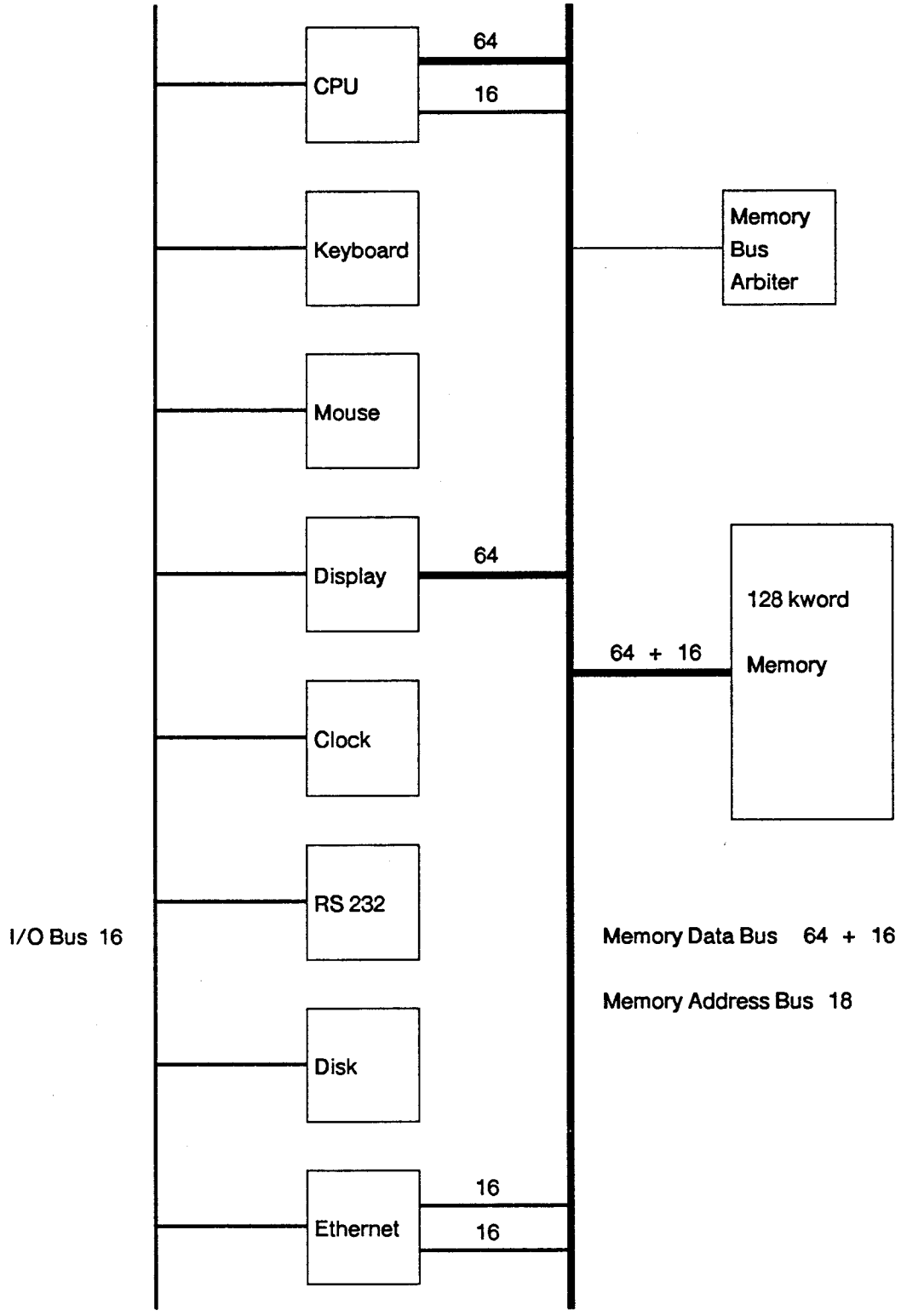


Figure 2.1 Lilith Hardware

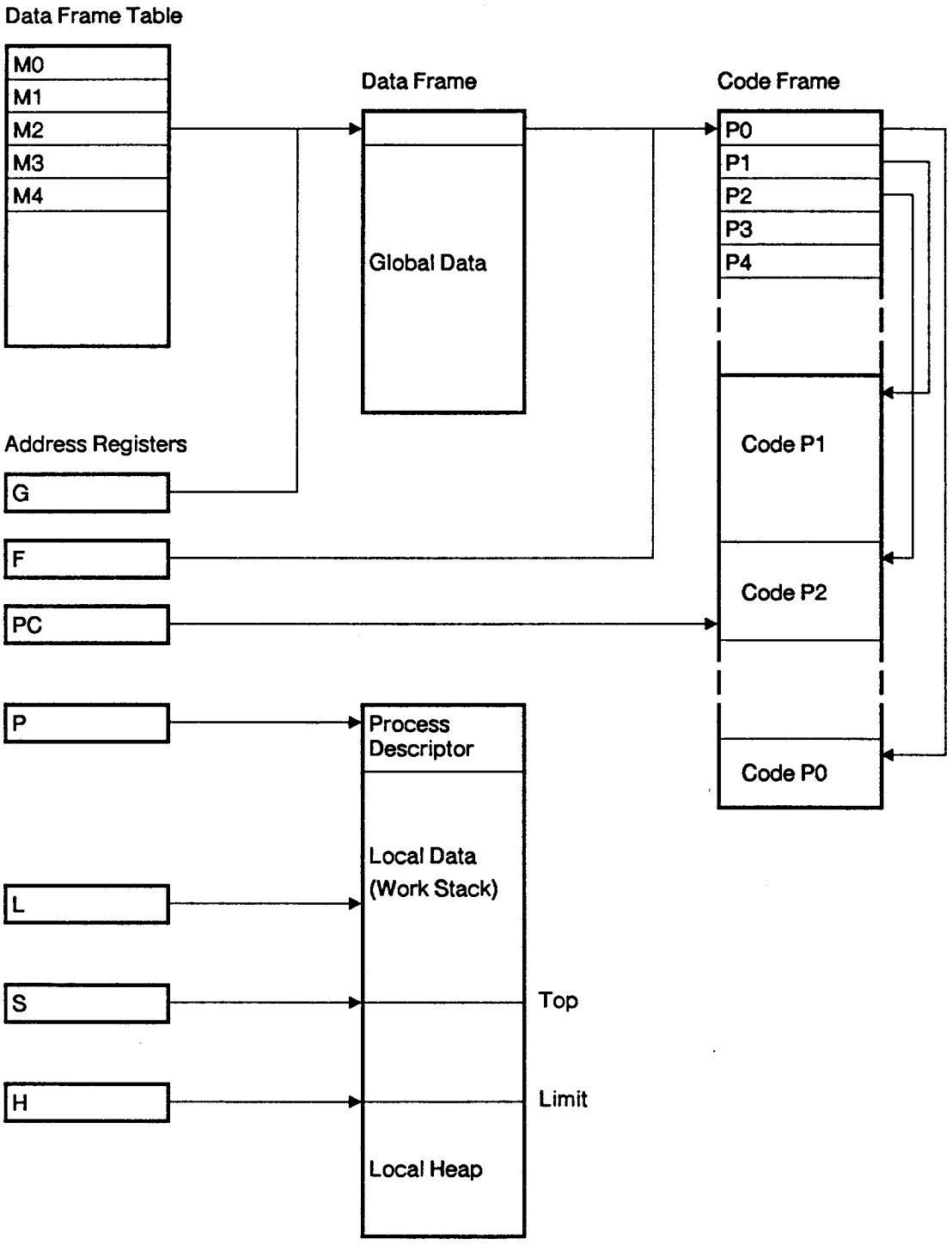


Figure 2.2 Lilith Architecture

2.3 The Programming Language Modula-2

The programming language *Modula-2* is a descendent of its direct ancestors Pascal [Wir71, JW75] and Modula [Wir77]. It is designed to satisfy requirements of high-level systems design as well as those of low-level programming of parts that directly interact with the given hardware. Roughly, Modula-2 includes all aspects of Pascal except files. The important module concept, the modern syntax, the multiprogramming aspect, and the low-level features have been influenced by Modula. In the rest of this chapter, the most important additions, compared to Pascal, are explained. These are grouped in the module concept, separate modules and separate compilation, coroutines, procedure types, and low-level (implementation dependent) facilities. The definition of Modula-2 may be found in [Wir82], more details on separate modules and their compilation may be found in [Gei83], and the code-generation of the Modula-2 compiler for Lilith is explained in [Jac82].

Modules

The module in Modula-2 is a syntactical structure which supports the modularization of programs. A module declaration is almost identical to the declaration of a parameter-less procedure. Three essential differences exist between procedures and modules:

The rules controlling the visibility (or validity) of objects, the so-called *scope rules*, are different. An object visible outside a module is only visible within the module if it is explicitly imported, and an object declared inside the scope of a module may be made visible outside the module by an explicit export.

The lifetime of objects declared inside a module is equal to the life-time of the objects in the scope enclosing the module (i.e. of the procedure enclosing the module).

The module's body (statement part) is executed when the environment of the module is instantiated. The body typically serves for the initialization of objects declared inside the module.

An example from Medos-2, a small coroutine scheduler in program *Comint*, should illustrate the module concept:

```
MODULE Scheduler;

  FROM SYSTEM IMPORT ADDRESS, PROCESS, NEWPROCESS, TRANSFER;
  EXPORT CreateProcess, Pass;

  CONST procs = 3;

  VAR ptab: ARRAY [0..procs-1] OF PROCESS;
      cur, top: [0..procs];
```

```
  PROCEDURE CreateProcess(proc: PROC; addr: ADDRESS; size: CARDINAL);
  BEGIN
    IF top < procs THEN
      NEWPROCESS(proc, addr, size, ptab[top]); INC(top)
    ELSE HALT
    END
  END CreateProcess;

  PROCEDURE Pass;
    VAR old: CARDINAL;
  BEGIN
    old := cur;
    cur := (cur+1) MOD top;
    IF old <> cur THEN TRANSFER(ptab[old], ptab[cur]) END
  END Pass;

BEGIN
  cur := 0; top := 1
END Scheduler.
```

The module *Scheduler* owns six local objects, namely a constant *procs*, three variables *ptab, cur,* and *top*, and two procedures *CreateProcess* and *Pass*. It exports the two procedures and hides the constant and the variables. Four objects are imported from module *SYSTEM*, namely *ADDRESS, PROCESS, NEWPROCESS,* and *TRANSFER*. The body of the module initializes the variables cur and top. The example shows how objects may be hidden and how access from outside is restricted to explicitly exported objects, in this case to the procedures *CreateProcess* and *Pass*. This makes it possible to guarantee invariants on the local objects of the module independent of possible errors in the environment and thereby makes it possible to understand the module without first studing its complete environment (in this case about 830 additional lines of Modula-2 text).

Separate Modules and Separate Compilation

A Modula-2 program consists of either a single (and independent) module or it is split into several *separate modules*. The environment of each separate module is considered as the "universe" in which the separately compiled modules are known. A program consists in this case of a main module together with all the modules which are directly or indirectly imported by it from the "universe". Mainly two concepts support the decomposition of programs into several modules on the global level, namely the so-called *qualified export* mode, and the splitting of a separate module into a part specifying the interface to it, its *definition module*, and a part specifying the realization, its *implementation module*.

Qualified export serves to avoid clashes between identically named identifiers exported from different modules into the same enclosing scope. This is especially a problem when a separate module is written because its writer may not know all exported objects. If the procedure *Pass* is exported in qualified mode from module

Scheduler, then the procedure needs to be denoted as *Scheduler.Pass* in the environment of module Scheduler. All exports from separate modules must be made in qualified mode. Therefore, a module writer has merely to choose a module name not already existing in his "universe" in order to avoid name clashes. (This rule also makes life much easier for the Modula-2 compiler, as it has only to search complete modules in the "universe".)

A separate module providing (exporting) objects for other modules must be split into a definition and an implementation module. The definition module describes the interface to the separate module at least syntactically and may therefore be considered as a contract of the module with its importers (clients). The definition module contains all declarations needed for a complete specification of the interface. Procedures are, for example, defined by their complete headings.

The following example shows module Scheduler as a separate module.

```
DEFINITION MODULE Scheduler;

  FROM SYSTEM IMPORT ADDRESS, PROCESS;
  EXPORT QUALIFIED CreateProcess, Pass;

  PROCEDURE CreateProcess(proc: PROC; addr: ADDRESS; size: CARDINAL);
  PROCEDURE Pass;

END Scheduler.
```

The corresponding implementation module contains the realization of the separate module, i.e. "things" which need not be known to the clients of the module. Generally, all objects declared in the definition module are implictly defined within the implementation module. Procedures are the exception; they must be declared again inside the implementation module, this time, however, including their bodies.

```
IMPLEMENTATION MODULE Scheduler;

  FROM SYSTEM IMPORT ADDRESS, PROCESS, NEWPROCESS, TRANSFER;

  CONST procs = 3;

  VAR ptab: ARRAY [0..procs-1] OF PROCESS;
      cur, top: [0..procs];

  PROCEDURE CreateProcess(proc: PROC; addr: ADDRESS; size: CARDINAL);
  BEGIN
    IF top < procs THEN
      NEWPROCESS(proc, addr, size, ptab[top]); INC(top)
    ELSE HALT
    END
  END CreateProcess;
```

```
  PROCEDURE Pass;
    VAR old: CARDINAL;
  BEGIN
    old := cur;
    cur := (cur+1) MOD top;
    IF old <> cur THEN TRANSFER(ptab[old], ptab[cur]) END
  END Pass;

BEGIN
  cur := 0; top := 1
END Scheduler.
```

The splitting of a separate module into an interface description and an implementation part is advantageous and crucial. It allows changing the implementation of a module whenever needed, or even having several implementations of the same module.

Separate compilation of modules implies full type checking across module boundaries, in particular among separate modules of the same program. The Modula-2 compiler for Lilith generates a so-called *symbol file* upon compilation of a definition module. The symbol file contains a symbolic encoding of the definition module and is considered to represent the separate module in compilations later on. The compiler reads the symbol file when the implementation of the separate module or another compilation unit importing the module (a definition or an implementation module) is compiled.

It is obvious that the compiler, maybe assisted by a program linker, has to check that all references to a separate module are based on the same interface description, i.e. on the same symbol file. For this check, the compiler generates a time stamp, called *module key*, when a definition module is compiled and includes it on the symbol file. The name of the module together with the module key identifies thereafter a (certain version of a) separate module uniquely.

Upon compilation of an implementation module the compiler writes the generated code on an *object file*. It also copies the module name and its key on the file as well as the name-key pairs of all modules referenced by the module. This allows a linker or linking-loader (as provided by Medos-2) to check by simple name and key matching tests, whether or not all references to a certain module are based on the same interface description (i.e. symbol file). The format of the symbol file is given in [Gei83], the format of the object file in Appendix 2.

Coroutines

Modula-2 does not provide "general" processes as do many other real-time or systems programming languages (Ada, Concurrent Pascal, PORTAL [Nae79], etc.). It includes instead, roughly speaking, the mechanisms needed for the implementation of processes, namely a simple mechanism to handle coroutines and the possibility to encapsulate a user-defined scheduler in a separate (library) module.

The examples used to illustrate the module concept show a very simple user-defined scheduler. It schedules processes with the *round-robin* algorithm. Coroutines are referenced by variables of type PROCESS. Procedure NEWPROCESS creates a new coroutine, given a parameterless procedure and a memory segment. Procedure TRANSFER transfers the "control" from one coroutine to another.

In Modula-2, an interrupt is considered as a transfer of control at an unpredictable moment. It can be regarded as equivalent to a statement TRANSFER(interrupted, driver) that is effectively inserted in the program whenever an interrupt request is accepted. The variable *driver* denotes the coroutine (process) that handles the request, whereas the variable *interrupted* will be assigned the suspended coroutine.

In the Lilith M-code architecture, each of the eight interrupt signals is associated with its own variables *interrupted* and *driver* at fixed locations (in the *interrupt vector*). A priority scheme and an interrupt-enable scheme allow disabling further interrupts while an interrupt is handled.

Procedure Types

The concept of *procedure types* has, although rarely used, shown up to be both powerful and important for providing the intended openness in Medos-2. Normally, procedures are simply considered as program parts or texts that specify actions on variables. A procedure may, however, also be considered an object of a certain type. From this point of view, a procedure declaration is a special kind of constant declaration, the value of this constant being a procedure. Modula-2 allows the definition of types whose values are procedures, so-called *procedure types*. Both variables and procedure parameters of procedure types may be declared. The procedure type declaration specifies the number and types of parameters, and if it is a function procedure, also the type of the result. Thereby, (globally declared) procedures may be assigned to variables (of a compatible procedure type) or be passed as an actual parameter to a procedure.

Low-Level Facilities

Modula-2 and its compiler for Lilith provide some low-level (i.e. implementation and/or machine dependent) facilities which are important for the implementation of Medos-2. The most important of them will be enumerated, together with a short explanation. Further explanations may be found in [Wir81, Wir82, and Han82].

The specification of the *absolute address* of a variable in its declaration helps in accessing variables at fixed locations in the memory. The M-code defines some variables to be at fixed locations (e.g. the interrupt vector and the data frame table).

Arithmetic on addresses is made possible with the type ADDRESS. Variables of type ADDRESS are compatible with any pointer type and with CARDINAL. The functions SIZE, TSIZE, and ADR also support address computation.

A formal procedure parameter of type ARRAY OF WORD may be substituted by an actual parameter of any type. Inside the called procedure, the actual parameter would typically not be inspected, but rather, for example be copied to or from a

secondary storage medium.

Type transfer functions make it easily possible to give the value of an expression another interpretation, i.e. its bit-pattern is assumed to be the value of another type. In Medos-2, a type transfer function is used, for example, to assign the value of an expression of type CARDINAL describing the initialization code of a program to a variable of type PROC. (See Chapter 5.2.)

Last but not least, *code-procedures* stand for M-code instructions which cannot be generated by ordinary Modula-2 statements. A call of such a code-procedure results in in-line code. Examples of such instuctions are instructions controling devices and instructions operating on display bitmaps.

3 Design Objectives and Provided Concepts

Section 3.1 enumerates some predicates of good programming. These have significantly influenced the development of Medos-2 and are therefore mentioned here. Medos-2 presents itself to the normal programmer as a collection of separate Modula-2 modules. The object-orientation aspect of these modules (and of the system) is presented in 3.2. Medos-2 provides a simple but powerful concept for the execution of programs. It is explained in 3.3. The mechanism for managing resources is explained in 3.4, and section 3.5 enumerates concepts which contribute to the openness of Medos-2.

3.1 Design Objectives

Several objectives have guided the development of Medos-2. From the beginning, it was the intention to make a good operating system of modest size. But what does the term *a good operating system* or generally *a good program* mean? The answer is different from individual to individual.

Among the set of predicates specifying the quality of a program, (e.g. modular, portable, efficient, general, well structured, etc.) the most important is that a program is *reliable*. One cannot depend on an unreliable program. A reliable program must both be correct and understandable, i.e. its behaviour must be foreseeable for its users. A reliable program must, however, also be robust to commonly occuring errors. Generally, the correctness of a program cannot be proved formally. By use of high-level programing languages, proof-reading, debugging, run-time tests, and other techniques, one might sooner or later get the conviction that a (certain piece of a) program is correct. Robustness against errors is generally achieved by redundancy in both the stored data and the programs accessing the data. The redundancy makes it possible to detect state inconsistencies and thereby to prevent one single failure to cause many succeeding errors as a consequence (i.e. a disaster). Redundancy may of course also help to reestablish a consistent state after an erroneous situation, without any loss of information.

A program should, however, also be *simple, adaptable,* and, whenever possible, *efficient*. Simplicity is needed in order to make a program understandable to its users and to its implementor(s). If an implementor does not understand an algorithm to be programmed, he cannot be expected to deliver a correct (and reliable) program. It is, however, generally not easy to find simple solutions solving a huge number of (more or less) complex problems. Both theory and experience appear to be very important. Most larger programs are written to be used for several (or even many) years. It is therefore important that a program is structured such that it can easily be adapted to "small" changes in the preconditions of the program, as they may change over the years. A program should, of course, also be efficient, if the efficiency is not conflicting with the previously mentioned attributes of good programs. It is cumbersome to live with unnecessarily slow programs. Fortunately, most of the time simple (not simple-minded!) solutions are also efficient solutions.

If in a certain case, several of the mentioned objectives are conflicting goals,

reliability should be considered as the most important objective, certainly the one which should not be subjected to compromise.

3.2 Medos-2 as a Collection of Modules

Medos-2 presents itself to the programmer as a collection of library modules which may be imported from any user-written module whenever desired. This modularization of the operating system interface and its passive nature (a module typically exports only several routines) is very desirable. It simply changes the programmer's view on the operating system from being *one supervisor*, i.e. a monolith which the programmer has to accept, into being a set of provided facilities (services, resource managers, device handlers, etc.) which in principle may easily be changed and which a programmer does not have to know completely. In the following, it is described how this *object-oriented* interface is provided by separate modules in Medos-2.

Following the so-called *object model* [Jon78], an operating system can be described as a set of *object types*, each of which can be thought of as a kind of resource. Some resources have a direct physical realization (I/O devices), others are "only" artificial (processes, files, etc.). Each resource is described by an instance of the object type, i.e. by an *object*. The object describes the state of the resource, and an operation on the object corresponds to a change of the resource's state. For example, consider type *File* and the operations on an instance of a file: *Create, Delete, Write, Reset, Read*, etc. In this model a resource (an object) is passive, and a user of a resource has only to understand the fixed set of operations defined for the resource. The representation of the resource (e.g. as stored information or as hardware) and the implementation of the operations on the resource are not essential for its user. Many recently developed operating systems are object-oriented, for example CAP [WN79], iMAX [Kah81], StarOS [Jon79], Pilot [Red80], and Solo [BH77]. The encapsulation of the objects (i.e. the protection of the resources) is supported by hardware in the first three examples. Medos-2, Pilot, and Solo merely provide no absolute (i.e. no uncircumventable) protection of resources by software.

The object model view of an operating system may easily be expressed in high-level programming languages supporting abstract data types (i.e. data encapsulation) [Par72]. In programs written in such languages, an object type (a resource kind resp.) is expressed as an abstract type, and an object (a resource resp.) is expressed as an instance of the corresponding abstract type. Abstract data types are, however, directly supported by few high-level programming languages only, and these have not been widely accepted. CLU [LSAS77], Concurrent Pascal [BH75], Euler [BH81], PORTAL [Nae79], and Simula 67 [DMN68] are examples of such languages.

Separate modules of Modula-2, however, may be used to provide interfaces with many of the desired data encapsulation aspects. Essentially three cases may be distinguished if a separate module is used this way:

A module exports routines operating on *one* set of data (representing one resource).

As exactly one instance of the type exists, the instance need not be identified for each called routine. The data needed for the representation of the instance is simply declared globally in the corresponding implementation module and initialized by the initialization code of the module.

A module exports a type and a collection of routines operating on variables (instances) of this type. A variable of the exported type passed as parameter to an exported routine identifies the instance, upon which the corresponding operation should be performed. Modula-2 allows to export *opaque* types. In this case, the structure of the exported type is unknown to the module's clients.

A module may provide an interface to several data abstractions, presented in one of the two variants mentioned above.

A separate Modula-2 module providing operations on exactly one set of data (one object) has the best correspondence with the data encapsulation idea. The data representing the instance of an abstract type can totally be hidden in the implementation part of the separate module and the initialization of the data can be guaranteed. Furthermore, it is impossible to issue operations on non-existing data because the data exist as long as the module is accessible (loaded).

A module exporting a type and a collection of routines operating on variables of the exported type supports the data encapsulation aspect less well. A variable of the exported type corresponds to an instance of the abstract type, but in order to keep the hiding aspect, the variable is in most cases effectively a reference to a data element representing an instance of the abstract type. The structure of the data element may thereby be hidden in the implementation part of the module. The allocation and deallocation (creation and deletion) of instances of the abstract data type must explicitly be done by procedure calls. But the lifetime of an object is ideally the same as the existence of the variable representing the object. Operations on uninitialized variables may occur if the first operation executed on a variable of the exported type is not an operation creating the corresponding object. The missing automatic deallocation of no longer referenced data elements makes it difficult to limit the lifetime of objects which erroneous programs "forget" to deallocate explicitly. How this deallocation problem is handled in Medos-2 is described in section 3.4.

The reason for providing an interface to several object types in one separate module is typically either a desire to provide convenience or a need for hiding (protecting) "something" which would otherwise be freely available to clients.

From Medos-2, module *Terminal* provides routines reading from or writing to the standard terminal. The module provides an interface to exactly one resource. Module *Frames* provides routines for the allocation and deallocation of a main memory segment, a so-called *frame*. A parameter of type *FramePointer* identifies the frame to be operated upon for the provided routines. Module *DisplayDriver* is of the third kind. The resources managed by the module are the display interface, the default bitmap, and the default font. Descriptions of module *Terminal, Frames,* and *DisplayDriver* may be found in Appendix 1.

3.3 Execution of Programs

A *program* is the formulation of an algorithm in a formalized notation (programming language). The definition almost always used for a *process*, that of being the execution of a (sequential) program makes sense only partly to programmers looking at "real-world" computer programs. Many computer programs make explicit use of concurrency for the description of algorithms. The concurrency is thereby expressed by either programming language or operating system primitives. Many high-level programming languages provide primitives for expressing concurrency (e.g. Ada, Concurrent Pascal, Portal, Edison, Mesa), mostly by offering the possibility to declare processes. Many operating systems provide primitives for executing several programs by one process, either one after each other or (less often) by a program call mechanism.

In the following text, the term *program* (or source program) stands for a textual description of an algorithm formulated according to the rules of a programming language. As Medos-2 essentially only supports the execution of Modula-2 programs, the programs mentioned will typically be Modula-2 programs. (See Chapter 2.3.) If no confusion is possible, the term program may also mean an activated (running) program.

A *process* is a unit of activity sequentially performing operations on objects, for example by sequentially executing statements of a Modula-2 program. Generally, a program may be executed by one process or several concurrent processes, and a process may possibly execute part of one or several programs. As a computer most often has only one central processor and several processes may be required to execute programs, most operating systems include a so-called process scheduler which multiplexes the processor among the processes.

Medos-2 does not support explicit concurrency in user-written programs. From the operating system point of view, one single process executes all user-written programs. The process does this by executing so-called *program calls* or *program activations.* The execution of a program may be considered as a generalization of the execution of a procedure. Any program may contain a statement (a program call) causing the activation of a program like a procedure call. During the execution of the called program, the caller of the program is suspended, and it is resumed when the called program has terminated.

All activated 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 program on top of the stack is the currently *running program*. Figure 3.1 illustrates how programs may be activated. In Medos-2, some essential differences exist, however, between the activation of a program and the activation of a Modula-2 procedure.

A program is identified by a *computable* program name (a string).

The calling program is also resumed, if a program terminates by a crash (*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 activated once at any one time, i.e. there are no instances, no recursion (programs are *not* reentrant).

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

At any moment, the *dynamic activation level* or simply the program *level* uniquely identifies an activated program. Program level 0 is the first activated program, i.e. the resident part of Medos-2. If program level *l* calls a program, the activated program is on level *l + 1*. The level of the running program, i.e. the activated program at the top of the activation stack, is the so-called *current level*. The caller of a program may indicate by a parameter to the program-call procedure that the calling program and the called may share resources whenever possible. The lowest program level sharing resources with the current level is the so-called *shared level*.

An activated program is represented by a *program activation record* in the stack. Currently, the activation record contains a *working stack* and the code and global data of all modules loaded for the called program.

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

After the execution of a program, its activation record is removed from the stack. The modules that were loaded when the program was activated are removed. At this time, all resources (objects) owned by the activated program are returned as well. Futher details of the management of resources will be described in the next section.

The main advantages of the described concept for the execution of programs are to be found in the following three areas:

The stack of activated programs enables a very high memory utilization (without any fragmentation). Generally, the architecture of Lilith does not support the relocation of used data in memory. Under this circumstance, memory allocation strategies more general than the stack scheme do not guarantee a comparably high memory utilization because of possible memory fragmentation. From the beginning, the available memory space (65536 word) was a critical factor. The operating system occupied about 12 kword and the refresh memory for the display (the bitmap) used 28.5 kword. Only 23.5 kword remained for the execution of user-written programs. This was little more than needed to run the Modula-2 compiler and the standard editor. (Later, another 65'536 word of main memory were added to all machines, and the bitmap is now allocated in this (not generally addressable) half of the memory.)

The binding of an activated program to activated programs at lower levels is very desirable. This mechanism allows every user to dynamically expand the set of facilities (modules) provided by the operating system. For example, this can be done

by executing a program importing (directly or indirectly) the additional modules. Its main module may for example be the standard command interpreter. Ordinary user-written programs will then be executed as level-2 programs, and they can also make use of the facilities provided by the program on level-1 (e.g. a module enabling communication over the local area network). The Modula-2 multi-pass compiler makes use of the same facility. The part common to all passes is executed as a program called *modula*. This program subsequently activates the passes in *shared* mode. Modules in program *modula* are used for the communication among the various passes, and the execution of a pass in shared mode guarantees that the heap and the inter-pass files are not deallocated when a compiler-pass program terminates.

The concept of activating a program just like a procedure is very convenient for programmers. The binding of the activated program to the already activated programs is quite natural, especially for programmers used to high-level programming languages allowing nested procedure declarations. To let activated programs be owners of resources (objects), just like activated procedures are "owners" of locally declared variables is both a convenient and very powerful concept. The management of resources is, however, subject to the next section.

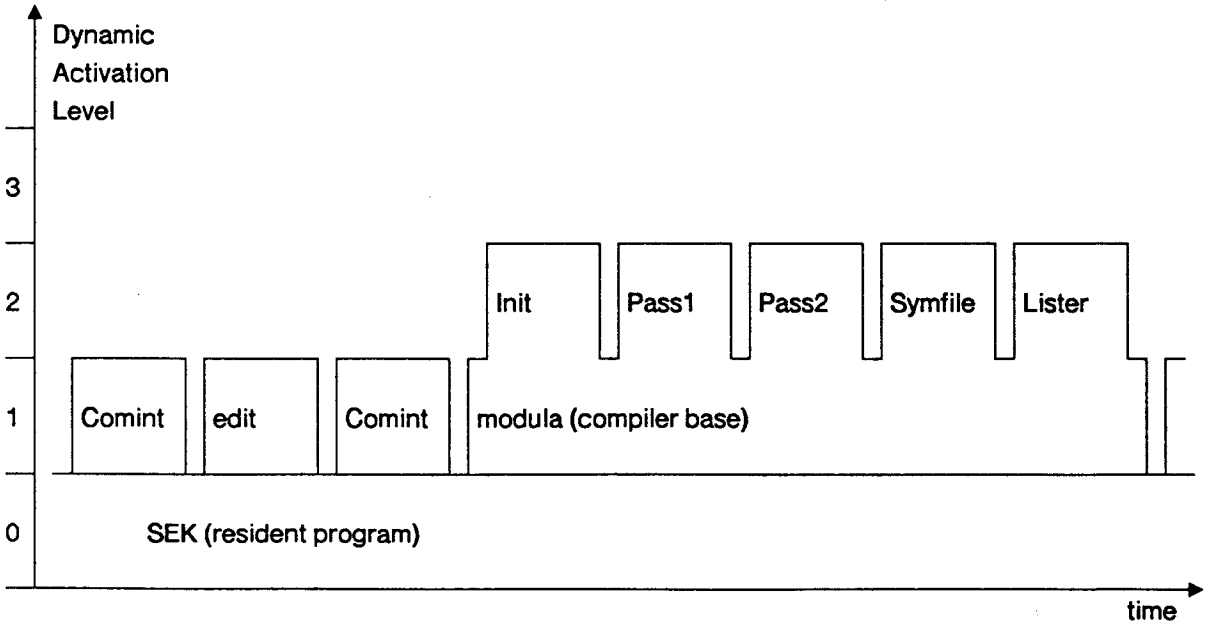


Figure 3.1 Execution of Programs

3.4 Management of Resources

In section 3.2, it was mentioned that objects are managed by separate modules, i.e. a module may provide routines which create, remove and perform other operations on a certain type of objects. In Medos-2, the allocation and deallocation strategy for objects as well as the rules governing the access to the objects can be chosen freely by the designer of the module providing the objects (the *object-handler*). Medos-2 is in this respect a very "liberal" or *open* single-user operating system. However, it is generally not without cost for a module to provide objects. The maximal number of provideable objects is therefore limited, either to a fixed or to a changing maximal number of objects. The allocation strategy for objects causes only few worries. As long as an object-handler can allocate an object, it will generally do it so. The rules controlling the access to an object is in most cases also very liberal: A program may issue an operation on an object if the object (and therefore also its handler) exists. The normal scope rules of Modula-2 may of course limit the visibility of existing objects and thereby hinder an access to a specific object, and the object-handler may introduce special rules on the usage of a certain type of objects. Difficulties arises, however, if an erroneous program "forgets" to deallocate an object it allocated. Object-handlers have to limit the lifetime of objects such that the following two conflicting goals are satisfied: It should be guaranteed that an object exists when it can be legally used, i.e. referenced objects should exist as long as references to these objects exist in order to avoid dangling references. On the other hand, the lifetime of objects should be as short as possible, in order to reduce or eliminate the costs of unused objects. If no longer accessible objects remain infinitely allocated, a system will sooner or later (virtually) crash because of lack of resources needed for the creation of new objects.

In Pascal-like programming languages, the lifetime of an object (e.g. type, variable) is with few exceptions equal to the execution time of the procedure, within which the object is declared. The local variables of a procedure are instantiated when the procedure is called and are removed when the procedure terminates. Scope rules guarantee that objects can only be accessed as long as they exist. Programming languages supporting abstract data types typically control the lifetime of instances of abstract types by the same rules.

Modula-2 also supports explicit allocation and deallocation of variables. Such variables are referenced by so-called pointers. It is, however, generally impossible to know the number of pointers referencing a dynamically allocated variable. The consequence is that a no longer referenced variable cannot automatically be retrieved when the last reference to it is removed. The no longer referenced variable has to remain allocated as long as the corresponding pointer type declaration exists. As most pointer types are declared globally in programs, a garbage collector is normally not included in a "storage allocator" for Modula-2 (or Pascal) programs. An allocated variable remains allocated until it is explicitly deallocated or the program terminates. In the latter case, all pointer type declarations have normally disappeared.

If a Modula-2 program is executed by Medos-2, a type declaration may survive the execution of a program, because not all modules imported by the program are

"thrown away" when the program terminates. Modules used for the execution of lower program levels remain activated. In order to, nevertheless, be able to automatically remove no longer used objects the following concept has been introduced:

An object is *owned* by an activated program.

An object owned by program level *l* may only be accessed from the same or higher program levels. A reference to an object owned by program level *l* is assumed to be stored in an activated program at level *k* greater or equal to *l* or in an object owned by such a program. An object may, therefore, be removed when its owner program terminates, and no other object holds a reference to the object.

In section 3.2 it was stated that each type of object (each kind of resource) is managed by one separate module. If exactly one object is provided by a module and this object exists as long as the module is activated, then there are no extraordinary problems. The object exists as long as it can be accessed. When, after termination of a program, the module is no longer imported, it is removed together with the provided object. Module *Terminal* in Medos-2 is an example of a module of this class (see Appendix 1).

More problematic are modules providing one or several objects which may be created explicitly by different program activations. Such modules typically export a type and a set of routines performing operations on variables of the exported type. In rare cases, an object may be completely described by a variable of the exported type. In such cases, the lifetime of an object is of course not longer than the lifetime of the variable describing it. Typically, however, a variable of the exported type serves only to identify (contain a reference to) the provided object. The following module illustrates the case:

```
DEFINITION MODULE Streams;

  EXPORT QUALIFIED
    Stream, Create, Remove, Read, EOS (*, ... *);

  TYPE Stream;

  PROCEDURE Create(VAR s: Stream; name: ARRAY OF CHAR; new: BOOLEAN);
  PROCEDURE Remove(VAR s: Stream);

  PROCEDURE Read(VAR s: Stream; VAR ch: CHAR);
  PROCEDURE EOS(VAR s: Stream): BOOLEAN;

  (* ... *)

END Streams.
```

A program may use module *Streams* for reading characters. A variable of type *Stream* represents a stream when it has been created and until it is removed. The following program module illustrates the usage of a stream.

```
MODULE ListText;

  FROM Streams IMPORT Stream, Create, Remove, Read, EOS;
  FROM Terminal IMPORT Write;

  VAR s: Stream; ch: CHAR;

BEGIN
  Create(s, "DK.Example.TEXT", FALSE);
  Read(s, ch);
  WHILE NOT EOS(s) DO Write(ch); Read(s, ch) END;
  Remove(s)
END ListText.
```

The program activation owning an explicitly created object must typically be determined when the object is created. In most cases, it will either be the currently running program or the shared program (the program at the lowest dynamic activation level sharing resources with the current program).

Medos-2 provides three routines (in module *Monitor*), which help modules to control the lifetime of objects: the function procedures *CurrentLevel* and *SharedLevel*, and the proper procedure *TermProcedure*. *CurrentLevel* returns the level-number of the running program and *SharedLevel* returns the level-number of the shared program. By a call to procedure *TermProcedure* in the initialization part of a module, the module declares a parameterless routine to be its *reset-routine*. The reset-routine will be called whenever a program terminates, provided the program was activated on a higher level than the level within which the object-handler is loaded. All defined reset-routines are called *before* the actual removal of the program, and in the *reversed* order of their declaration as being reset-procedures.

This simple implementation of module Streams illustrates how an object-handler may use these three routines for the automatic removal of objects when a program terminates.

```
IMPLEMENTATION MODULE Streams;

  FROM FileSystem IMPORT   (* See also Appendix 1.7 *)
    File, Response, Lookup, Close, ReadChar, WriteChar;

  FROM Monitor IMPORT CurrentLevel, SharedLevel, TermProcedure;

  CONST streams = 16;

  TYPE
    Stream = CARDINAL;
    StreamDesc = RECORD free: BOOLEAN; owner: CARDINAL; f: File END;

  VAR streamTab: ARRAY [0..streams-1] OF StreamDesc;
```

```
PROCEDURE Create(VAR s: Stream; name: ARRAY OF CHAR; new: BOOLEAN);
BEGIN
  s := 0; (* search a free entry in streamTab *)
  LOOP
    IF s >= streams THEN (* ... *) HALT END;
    IF streamTab[s].free THEN EXIT END;
    INC(s)
  END;
  WITH streamTab[s] DO
    Lookup(f, name, new);
    IF f.res <> done THEN (* ... *) HALT END;
    owner := SharedLevel(); (* a stream is sharable *)
    free := FALSE;
  END
END Create;

PROCEDURE Remove(VAR s: Stream);
BEGIN
  IF (s >= streams) OR streamTab[s].free THEN RETURN END;
  WITH streamTab[s] DO Close(f); free := TRUE END
END Remove;

PROCEDURE Read(VAR s: Stream; VAR ch: CHAR);
BEGIN
  IF (s >= streams) OR streamTab[s].free THEN (* ... *) HALT END;
  ReadChar(streamTab[s].f, ch)
END Read;

PROCEDURE EOS(VAR s: Stream): BOOLEAN;
BEGIN
  IF (s >= streams) OR streamTab[s].free THEN (* ... *) HALT END;
  RETURN streamTab[s].f.eof
END EOS;

(* ... *)

PROCEDURE ResetLevel;
  VAR s: Stream;
BEGIN
  FOR s := 0 TO streams-1 DO
    WITH streamTab[s] DO
      IF NOT free AND (owner >= CurrentLevel()) THEN Remove(s) END
    END
  END
END ResetLevel;
```

```
  VAR  s: Stream;

BEGIN
  FOR s := 0 TO streams-1 DO streamTab[s].free := TRUE END;
  TermProcedure(ResetLevel)
END Streams.
```

Streams are sharable. Procedure *Create* sets the owner of a new stream equal to the currently shared level, i.e. to *SharedLevel()*. (An object-handler can provide sharable objects only if all objects referenced by it in its implementation are also shareable. Files provided by module *FileSystem* are shareable!)

Procedure *ResetLevel* is called whenever a program terminates. *ResetLevel* removes all streams owned by the terminating program or owned by even higher program levels. The file needed for the implementation of a stream is owned by the same program level as the stream. The fact that module *Streams* imports module *FileSystem* and that there are no circular imports among modules in Medos-2 guarantee that the initialization part of module *Streams* is executed *after* the execution of the initialization part of module *FileSystem*. The reset-routine of module *Streams* is, therefore, defined *after* the reset-routine (i.e. procedure *ResetLevel*) of module *FileSystem*. As the reset-routines are called in the reverse order of their announcement to module *Monitor*, procedure *ResetLevel* of module *Streams* will be called *before* an eventually defined reset-routine of module *FileSystem*. This ordering of the reset-routine calls prevents that an object is automatically removed before an object containing a reference to it is removed.

By the example above, it has informally been shown that the three routines *CurrentLevel, SharedLevel*, and *TermProcedure* provided in Medos-2 both simply and powerfully support the automatic removal of no longer used objects. They are one of the pillars for Medos-2's ability to recover resources after program crashes.

Another pillars of the provided recoverability is to be found in the careful programming of routines performing operations on objects. An operation on an object should ideally be indivisibly executed in order to maintain consistent data. Serious problems may occour when a program crashes during the execution of a procedure exported by a module which remains activated after the removal of the terminated program. Medos-2 recovers from most program crashes in "unhappy moments" by making at least the remove-operation, i.e. the reset-routine, of an object-handler reexecutable and insensitive to inconsistent data, and by ordering statements performing changes on an object properly. Detected inconsistent states, from which the system cannot recover, however, lead to the system's suicide.

3.5 Concepts Providing Openness

The desired openness of Medos-2 is basaed on several concepts:

The fact that any program may execute programs and that activated programs are bound not only to the resident system makes the system extensible. A level-1 program may for example include a collection of modules effectively providing a

totally changed environment for programs on higher levels. The programs executed on level-2 or even higher levels will effectively run on top of another operating system. The XS-1 system [BBF82] is implemented this way. XS-1 is a single-user system providing an integrated user-interface and hierarchically structured files.

The binding of activated programs at load-time, and the method for searching not already loaded modules on files, makes it easy to substitute user-written implementations for public modules, to make additions and corrections to Medos-2, and to provide new modules. Generally, no distinction is made between user-provided and system-provided programs or modules. Therefore, any user is free to develop the modules needed for his special application.

The idea of extending the standard operating system by a non-resident program providing additional facilities is also made feasible by the fact that non-resident object-handlers may recover from program crashes the same way resident modules do, namely by use of reset-routines. The number of object-handlers may change during the lifetime of the system.

Several resident modules allow non-resident programs to provide routines which are in effect new or additional implementations of the resident modules. The possibility to declare procedure types and variables of procedure types in Modula-2 makes it easy to provide this facility. For example, the *Read* and the *Write* routines in module *Terminal* may be substituted by user-written routines. (See Appendix 1.13 and Appendix 1.14.) A non-resident program may substitute the input from the keyboard by characters read from a file and thereby introduce *commandfiles*. Another place in the system, where this concept is provided, is in the file system. All operations on a file are encoded into a call of one of two procedures, namely procedure *FileCommand* and *DirectoryCommand* exported from module *FileSystem*. A (resident or non-resident) module may implement files on a medium and make them accessible via module *FileSystem* by a call of the "create medium" routine. This routine needs as parameter the name of the medium on which the module provides files, and the two routines corresponding to *FileCommand* and *DirectoryCommand*. (See Appendix 1.7, section 5.)

Openness of the system is also directly supported by the module interfaces. An example showing this is module *FileSystem*. The unexperienced user may use files in a very ordinary manner. The routines *Lookup, Close, ReadChar, WriteChar,* etc. enable a simple use of files. More demanding users (e.g. systems programmers) may get higher performance and/or flexibility by directly accessing the file-buffers, by building file-directories themselves, by making ramdom accesses to files, etc. The same module *FileSystem* also provides routines enabling this kind of "low-level" programming. (See Appendix 1.7, section 4.)

The possibility to access all separate modules in the resident system, and even to access the hardware if needed, also makes the system more open. This kind of openness is only relevant or useful for relatively few experienced programmers. Direct access to the hardware may of course cause the system to crash, if it is not carefully programmed. Nevertheless, this freedom is felt to be essential by users of the work-station.

4 System Overview from the User's Side

This chapter gives an overview of Medos-2 as it presents itself to a user. In section 4.1 it is briefly explained how a program may be activated from the keyboard (job control). 4.2 presents the operating system interface as it may be seen by programmers. In 4.3 the structure of the resident part of Medos-2 is shown, and some statistical information about the system is given.

4.1 The Command Interpreter

After Medos-2 has been loaded (booted) from disk, the main program of Medos-2, i.e. the standard command interpreter, executes. It first initializes the operating system and asks its user to log in. Thereafter the command interpreter repeatedly executes the following tasks:

- read the next command (i.e. read the name of the program to execute next),
- interpret the command (i.e. call the corresponding program), and
- report errors (i.e. write program load or execution errors).

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

The command interpreter indicates with an asterisk (*) that it is ready to accept the next command. Actually, there is only one type of command: the *program call*.

To call a program, the name of the program has to be typed in, for example

```
*directory
```

The program with the typed in name is called. If some load or execution error occured, an error message is displayed. Thereafter, the command interpreter is ready to accept the next comand, and therefore it displays an astennrisk again.

```
*direx
 program not found
*directory
 display output from the directory program
*
```

A *program name* is an identifier or a sequence of identifiers separated by periods. (An identifier begins with a letter followed by further letters or digits. Capital and lower case letters are treated as distinct.)

In order to asist the user in typing a program name, 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 name start with the typed character sequence, the sequence is only extended up to the point where the program names start to differ.

The command interpreter accepts several *special keys* as commands to be interpreted immediately.

?	Display a list of all programs starting with the typed characters.
DEL	Delete the last typed character.
CRTL-X	*Cancel.* Delete all typed characters.
CRTL-L	*Form Feed.* Clear the screen. (Accepted after an asterisk.)
CRTL-C	*Kill character.* Stop the execution of the running program.
ESC	Terminate the execution of the command interpreter.

It is possible that a sequence of programs must be called several times. In this case, instead of typing all commands interactively, it is more appropiate to substitute these commands as a batch. For this purpose the operating system allows the substitution of *command files.* A command file must contain exactly the same sequence of characters as normally would be typed on the keyboard, i.e. the names of the programs to execute next and the answers given in the expected dialog with the called programs. Program *commandfile* initializes the substitution of keyboard input with the text stored on a command file. This program prompts for the name of a command file. After the whole text has been read from the command file, the input is read again from the keyboard.

```
*commandfile
 Command file> transfer.COM
*input is now read from the command file
 instead of from the keyboard

 End command file
*
```

Programs are loaded from the user's disk cartridge. In order to find the file from which the program should be loaded, the linking-loader converts the program name into a file name. It inserts the medium name DK. at the beginning of the program name, appends the extension .OBJ, and searches for a file with this given name. If no such file exists on the user's cartridge, the loader inserts SYS. into the file name after the medium name and searches for a file with this name.

Accepted program name	`directory`
Firest file name	`DK.directory.OBJ`
Second file name	`DK.SYS.directory.OBJ`

A more detailed description of the user's options to control execution of programs may be found in the *Lilith Handbook*, Chapter 2 and 3 [Han82].

4.2 The Program Interface

For the (inexperienced) programmer, Medos-2 presents itself as a collection of library modules. The programmer essentially sees no difference between operating system modules, ordinary (non-resident) library modules, and modules written by the programmer himself. Thus, the *mechanism for checking* the syntactically correct use of separate modules at compile and linking-load time are *also provided if operating system modules are imported* [Han82 Chapter 7, Gei83]. The main difference between modules provided by the operating system and other modules is to be found in the fact that the former are always loaded when Medos-2 is running

(because it makes use of them itself).

When a program is activated, its main module and all directly or indirectly imported and not already loaded modules are loaded and linked to each other *and* to the already loaded modules. This way, the activated program is linked to the operating system at load time, if it imports any of the operating system's modules.

The resident part of Medos-2 consists of 15 modules, one of them is the *run-time system* needed to execute any Modula-2 program on Lilith. The remaining 14 modules provide an interface to operating system facilities. Oormally only three of these modules are of major interest, namely the modules *Terminal, FileSystem*, and *Program*. The purpose of these three modules is illustrated here. Short descriptions of the purpose of the other modules are given at the end of this section. Appendix 1 contains the definition modules of all 15 modules together with explanations needed by programmers to use them. (The Modula-2 run-time system, i.e. module *System*, is included for comleteness.)

Module *Terminal* provides routines to read from the standard input and write to the standard output. Normally, the standard input is connected to the keyboard and the standard output to the display. The routines provided for reading characters are *Read* and *BusyRead*, the routines provided for writing characters are *Write, WriteLn,* and *WriteString*.

Module *FileSystem* provides an open interface to files. Thus, a client of module *FileSystem* may use files on several abstraction levels. Medos-2 files are, however, always byte-sequences stored on a certain storage medium (e.g. a disk cartridge, a magnetic tape, or the main memory).

Files may be used much like sequential files in Pascal. The routines *Lookup, Close, Reset, ReadChar, WriteChar*, etc. are provided for this purpose.

Module *FileSystem* also provides routines enabling a program to position a file at any stored Byte, to directly access a file's buffer, to explicitly control the buffering, to protect files against modifications, and to control the permanency of files.

Module *FileSystem* does *not* include any device drivers. Instead, a resident or non-resident module may, by a call to procedure *CreateMedium* in module *FileSystem*, specify that it provides an implementation of files on a certain, uniquely identified, medium. Several such so-called *file implementations* may coexist, which makes it possible for a program to access files stored on several different media.

The following example illustrates the use of module *Terminal* and module *FileSystem*. The program lists a text stored on a file.

```
MODULE ListText;

  FROM FileSystem IMPORT File, Response, Lookup, Close, ReadChar;
  FROM Terminal IMPORT Read, Write, WriteLn, WriteString;

  CONST eol = 36C; fnlength = 32;

  VAR
```

```
    f: File;
    fn: ARRAY [0..fnlength] OF CHAR;
    ch: CHAR; c: CARDINAL;

BEGIN
  WriteString(" name of file to list > ");
  c := 0;
  LOOP
    Read(ch);
    IF (ch = " ") OR (ch = eol) THEN fn[c] := 0C; WriteLn; EXIT
    ELSIF c < fnlength THEN Write(ch); fn[c] := ch; INC(c)
    END
  END;
  Lookup(f, fn, FALSE);
  IF f.res <> done THEN WriteString("- file not found"); WriteLn
  ELSE
    LOOP
      ReadChar(f, ch);
      IF f.eof THEN EXIT END;
      Write(ch)
    END;
    Close(f)
  END
END ListText.
```

Module *Program* controls the execution of (sequential) programs and manages the generally addressable 65'536 word part of Lilith's main memory. Any program may activate another program by calling procedure *Call* in module *Program*. This facility, for example, enables users to provide their own command interpreter (job control language) and/or environment for programs. A program may be executed in either *shared* or *unshared* mode. The caller of a program specifies the actual execution mode by a parameter to procedure *Call*. In shared mode, the called program shares resources with the caller program whenever possible (i.e. if the resources are shareable). The heap and files are the most commonly used shareable resources provided by Medos-2.

The routines *AllocateHeap* and *DeallocateHeap* enable a client module to dynamically increase and reduce the size of the heap provided for the running program. However, modules typically allocate and deallocate memory space by use of the (non-resident) library module *Storage* [Han82 Chapter 9.3]. The following example illustrates the use of module *Program*:

```
MODULE MiniComint;

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

  CONST eol = 36C; pnlength = 16;
```

```
  VAR
    pn: ARRAY [0..pnlength] OF CHAR;
    st: Status;
    ch: CHAR; c: CARDINAL;

BEGIN
  LOOP
    Write("*");
    c := 0;   (* read program name *)
    LOOP
      Read(ch);
      IF (ch = " ") OR (ch = eol) THEN pn[c] := 0C; WriteLn; EXIT
      ELSIF c < pnlength THEN Write(ch); pn[c] := ch; INC(c)
      END
    END;
    Call(pn, FALSE, st);
    IF st <> normal THEN
      WriteString("- some load or execution error occured"); WriteLn;
    END
  END
END MiniComint.
```

Whenever a program is activated, its main module is loaded from an object code file. The name of the object code file is generated from the program name given as argument to procedure *Call* as explained in 4.1. All directly or indirectly imported modules are also loaded from files if they are not already used by activated programs. In the latter case, the just called program is bound to the already loaded modules. 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 an imported module is searched similarly as with programs. A file name is generated from the module name by inserting DK. at the beginning of the module name and appending the extension .OBJ to it. If the file does not exist, a second search is made after the prefix DK. has been replaced by the prefix DK.LIB. If the object code file is still not found, the object code for another missing module is searched. This is tried once for each imported but still unloaded module.

Module name	Storage
First file name	DK.Storage.OBJ
Second file name	DK.LIB.Storage.OBJ

Modules not Mentioned in the Resident System

Module *SEK (Sequential Executive Kernel)* is the main program of the resident program. It is the resident part of the standard command interpreter. It also configures the system by importing the needed modules and initializes the system after its boot and whenever needed later on.

Module *DiskSystem* implements files on disk cartridges for the Honeywell Bull D120/D140 disk drive connected to Lilith.

Module *D140Disk* is the driver for the Honeywell Bull D120/D140 disk drive connected to Lilith.

Module *TerminalBase* enables programs to substitute the read- and/or write-routines of module *Terminal* with program-specific routines.

Module *DisplayDriver* provides the lowest level interface to the display controller, the default bitmap for the display, and the default font. A routine writing characters into the default bitmap is also provided by this module.

Module *DefaultFont* provides the default font used by module *DisplayDriver* as a global variable. The reason for the existence of this module is to be found in the way the default font is loaded. This is done by initialization of global data in the absolutely linked operating system. In the actual implementation of the absolute linker, this is much simpler to do for a module with only one or a few globally declared variables.

Module *Frames* manages the "upper" not generally addressable part of the main memory. Client modules may allocate and deallocate contiguous memory segments, so-called *frames*.

Module *Monitor* provides interfaces to three different facilities, namely to the keyboard, the clock, and a low-level program execution facility. The routines of the program execution facility include three routines supporting object management in client modules (see Chapter 3.4). Module *Monitor* hides all concurrency in the system, i.e. the existence of two interrupt-driven driver processes (the line-clock process and the trap-handler) and the normal process.

Module *UserIdentification* enables a process to get the identification of the (usually) human user, for whom it executes. This is essential if several work-stations and so-called servers are connected together by a communication network.

Module *FileMessage* provides a routine displaying a file system error-message.

Module *CardinalIO* provides two routines for reading and writing numbers in octal form. This module is used for writing out error-messages

Module *System* is the run-time system needed for executing any Modula-2 program on Lilith. It essentially implements procedure *NEWPROCESS* exported by the standard module *SYSTEM* defined in Modula-2. The absolute code linker automaticly includes this module in the generated absolute code executable on a bare Lilith. The module is not explicitly imported by any module in the resident part of Medos-2.

4.3 The Structure of the Resident System

The bare operating system Medos-2 consists of a (memory-) resident Modula-2 program called *SEK* and two non-resident programs called *Comint* and *CommandFile*. The non-resident programs are effectively the non-resident parts of

the main program module *SEK*. The resident program consists of 14 modules hierarchically ordered by their mutual imports. The run-time system for Modula-2 programs is included in the list of modules in the resident system, because it is also resident when the system runs. Figure 4.1 shows this hierarchy of modules. The main module of the resident system, module *SEK*, is placed at the top of the figure. A module imports only modules placed lower in the figure. An import is indicated with a line.

The module hierarchy not only makes the system easier to understand, it also helps to verify it. The simplicity of the concept used for managing objects in the system is mainly due to the strict hierarchical ordering of the system (see Chapter 3.4).

The following table contains for each module in the resident system the number of source code lines (definition module and implementation module separately), the number of needed memory word (data space and code space separately, as well as their sum), and the fraction of memory space needed by the module compared to the memory space needed by the whole system. The statistics were taken on October 23, 1982.

Module Name	Lines Def.	Lines Impl.	Word Data	Word Code	Word C+D	C+D of Total C+D
SEK	25	168	84	266	350	2.4 %
Program	46	871	132	1352	1485	10.1 %
Monitor	49	506	333	576	909	6.2 %
FileSystem	228	577	65	1001	1066	7.2 %
DiskSystem	109	2002	3471	3518	6989	47.4 %
D140Disk	39	266	92	373	465	3.2 %
Terminal	21	55	17	74	91	.6 %
TerminalBase	15	94	41	177	218	1.5 %
DisplayDriver	46	205	46	344	390	2.6 %
DefaultFont	11	32	1439	42	1481	10.0 %
Frames	16	244	21	429	450	3.0 %
UserIdentification	16	75	58	127	185	1.3 %
CardinalIO	9	52	14	129	143	1.0 %
FileMessage	9	37	153	115	268	1.8 %
System	25	168	172	98	270	1.8 %
Total	729	5322	6138	8622	14760	

From this table it can be seen that the modules needed to execute programs including the linking-loader (i.e. modules *Program* and *Monitor*) needs 16.3 %, the modules providing files (i.e. module *FileSystem*, *DiskSystem*, and *D140Disk*) needs 57.8 %, the modules providing a "terminal" (i.e. module *Terminal*, *TerminalBase*, *DisplayDriver*, and *DefaultFont*) need 14.7 %, and the remaining modules 11.3 % of the memory space used by the entire operating system.

The source text of the non-resident program *Comint* is 860 lines long, and the program *CommandFile* is 52 lines long.

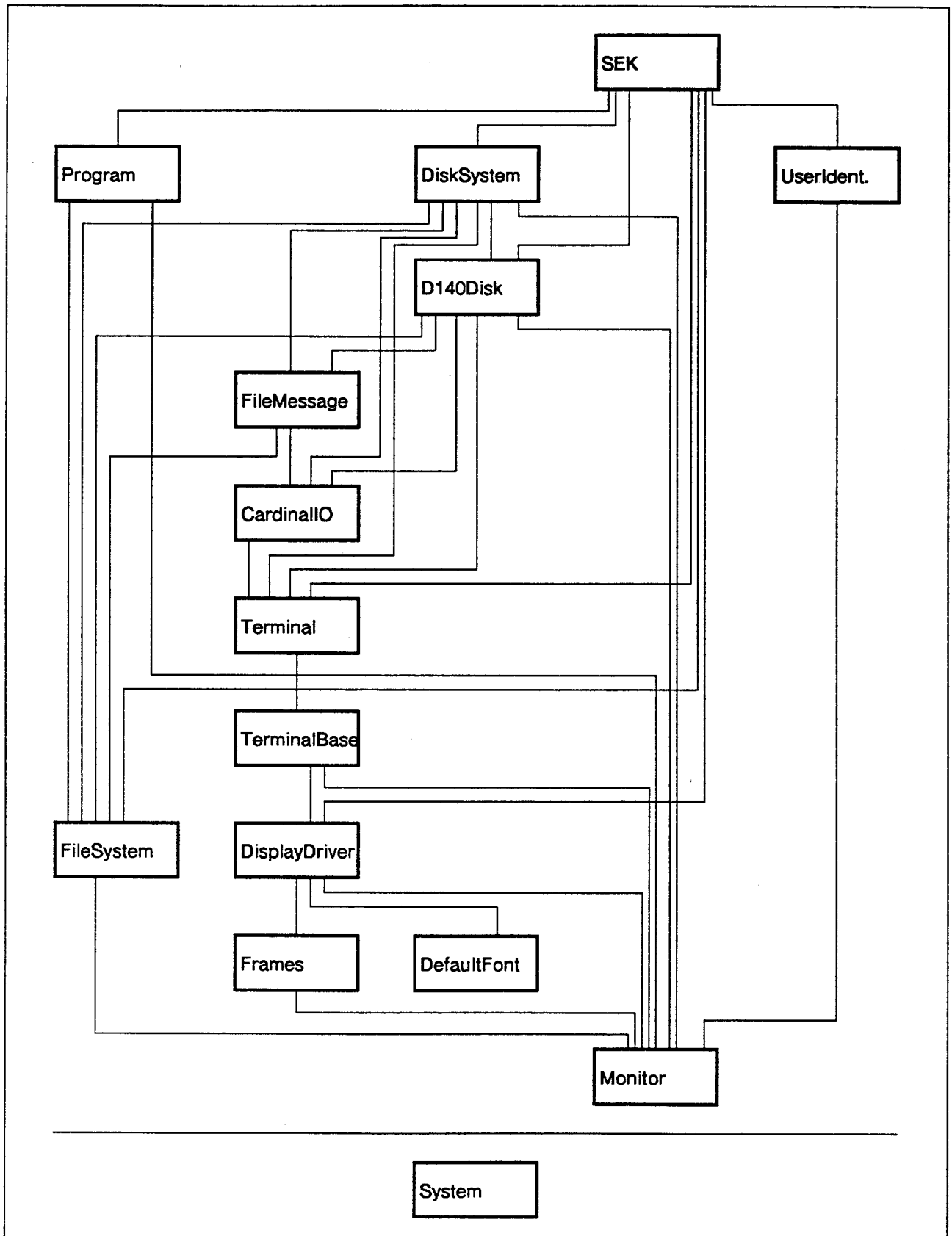


Figure 4.1 Resident Part of Medos-2

5 Implementation of Programs

Mainly three facilities support the execution of programs, namely the main memory management, the linking-loader provided by module *Program*, and a low-lewel executive provided by module *Monitor*. In section 5.1 the main memory management is presented. Section 5.2 deals with the linking-loader, and the part of module *Monitor* which supports the execution of programs is presented in 5.3.

5.1 The Management of Main Memory

The generally addressable part of Lilith's main memory (i.e. the main memory from address 0 to address 65'535) is managed by module *Program*. The not generally addressable part of the main memory, the so-called *upper bank*, is managed by the separate module *Frames*. (See Appendix 1.8 and Appendix 1.10.)

Management of the Generally Addressable Part of the Main Memory

Module *Program* divides the generally addressable part of the main memory into three parts, the *program activation stack*, the *free area*, and the *heap stack*. Figure 5.1 illustrates the memory layout. In the following text, the program activation stack will simply be called *the stack* and the heap stack simply *the heap*. The program activation stack grows from address 0 towards the *top of stack*, the free area is the memory segment between top of stack and the *stack limit*, and the heap stack is the memory segment from stack limit to the highest possible generally useable address. The stack limit is also called *top of heap*.

(Parts of) the free area may on demand be allocated for both the stack and the heap.

Whenever a program is called, a *program activation record* for the activated program is pushed on top of the stack by procedure *Call*. Currently, the program activation record contains the code and global data of the modules, loaded for the called program, and the working stack (i.e. the coroutine) needed for the execution of the called program. The activation record of the *running program* is limited at the high end by top of stack. (See also Figure 5.2 and 5.3.) How the activation record is formatted, and how the linking-loader loads modules onto the stack is described in section 5.2. During the execution of the running program, top of stack is implicitly incremented or decremented by the running program, for example when a procedure is called and returned from. The M-code instructions incrementing top of stack check that top of stack remains less than or equal to stack limit. When a program terminates, its activation record is simply popped from the stack again.

Module *Program* also handles the area reserved for heaps in the simplest possible way, namely as a "reverse" stack of heaps. The heap at the top of the heap stack, the *current heap*, may be enlarged by decrementing the stack limit and reduced by incrementing the stack limit. Clients of module *Program* may, with the exported routines *AllocateHeap* and *DeallocateHeap*, change the size of the current heap. If a program is activated in shared mode, i.e. if the parameter *shared* of procedure *Call* is set TRUE, the heaps are not affected The current heap may grow and shrink, as if no program had been activated. If a program is called in unshared mode (parameter

shared set FALSE), a new current heap is created for the called program on top of the heap stack. When an unshared program terminates execution, its current heap is simply popped from the heap stack, and the heap previously used by the calling program is the current heap.

The handling of the program activation stack will be further documentated in section 5.2. The implementation of the routines *AllocateHeap* and *DeallocateHeap* is straight forward and will therefore not be shown in detail. Top of stack and stack limit of the currently executed coroutine are kept in two processor-registers (called *S* and *H*, see Figure 2.2). The value of stack limit can easily be obtained and set by code-procedures. (See Chapter 2.3.) The value of top of stack is also obtainable by a small routine. With the aid of these three routines, *AllocateHeap* and *DeallocateHeap* are very simple to implement. The removal of the current heap after the termination of an unshared program is done by procedure *Call* in module *Program*.

Rationales for the Stack-Orientation of the Heap

There are several reasons for providing "only" a simple stack-oriented management of the heap:

When the system was designed, it was felt to be important that programmers could choose their private algorithm for the allocation and deallocation of dynamically allocated variables in the heap. Medos-2 provides a standard non-resident library module *Storage* which enables programmers to allocate and deallocate pointer-referenced variables in an easy way [Han82, Chapter 9.3]. Programmers may implement both simpler or more sophisticated algorithms whenever they feel this to be necessary.

The mechanism for managing the heap stack is very reliable and robust against program crashes. It is very easy for module *Program* to remove the current heap from the heap stack. It is essentially done by an assignment to the stack limit register *H*.

The removal of the whole heap previously provided for an (unshared) program guarantees that allocated memory space for a program is freed when the execution of the program is terminated. It also prevents that a fragmented heap survives the program that caused its fragmentation. Such fragmentation would generally cause a suboptimal memory utilization and (even worse) an almost unpredictable system behaviour.

The reason for managing the main memory in module *Program*, the second highest module in the hierarchy of modules in the resident system (see Figure 4.1), and not for example by module *Monitor*, is due to the linking-loader. The linking-loader stores loaded modules in the program activation stack and not in the heap. The mechanisms for explicitly pushing data onto and popping data from the program activation stack are provided for the linking-loader and, therefore, are better hidden in the module containing the linking-loader. It is desiriable for the linking-loader not to store loaded modules in the ordinary heap for two reasons:

The current heap may not be released after the execution of an unshared program. In order to be able to free the space needed for the modules loaded for the terminated program, the resident system had to include a more general algorithm for the allocation and deallocation of memory segments in the current heap. But even the best algorithm could generally not avoid fragmentation problems if an allocated heap segment must remain at the allocated location during its lifetime. Consider the Modula-2 multi-pass compiler. The passes of the compiler are executed in shared mode. If pass one includes a module whose code requires a 2000 word large memory segment to be loaded, a 2000 word large free segment may remain in the heap after the execution of pass one. If a module in pass two needs 2010 word for its execution, the consequence could be that the size of the current heap must be enlarged by these 2010 word, and that a 2000 word unused segment would remain in the heap. It can easily be imagined that the fragmentation problem gets worse the more programs (passes) are executed after each other in shared mode. The current Modula-2 compiler executes up to six programs after each other in shared mode!

Until recently, the Modula-2 compiler did not include the size of the module code at the beginning of the module's object code file. This information is, however, needed for the allocation of a memory segment in the heap (or in the *upper bank*). Under this circumstance, the "trick" of loading modules into the program activation stack helps also to simplify the resident linking-loader.

Management of the Not Generally Addressable Part of the Main Memory

The management of the *upper bank* will not be discussed at length. Currently, the *upper bank* is used for the storage of display bitmaps, fonts, and of memory-resident files. In rare cases, application programs also keep information in this part of memory. Thus, the main idea for the usage of the *upper bank* is to let the operating system modules and non-resident library modules use this not generally addressable memory for the storage of "large", machine-dependent objects. The normal programmer may, therefore, generally execute larger programs on a Lilith with 131072 word of main memory than would otherwise be possible on a machine with 65535 word of main memory. Module *Frames* provides routines for the allocation and deallocation of *frames*, fixed allocated memory segments addressed via so-called *frame pointers*. The module allocates frames with a straight forward first fit algorithm. Further details about this module can be derived from its description in Appendix 1.8.

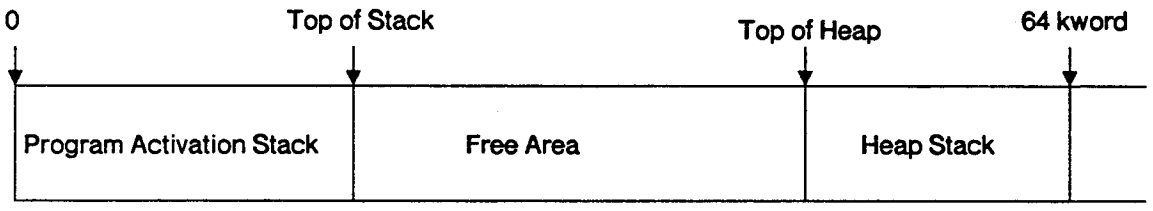


Figure 5.1 Normally Addressable Main Memory

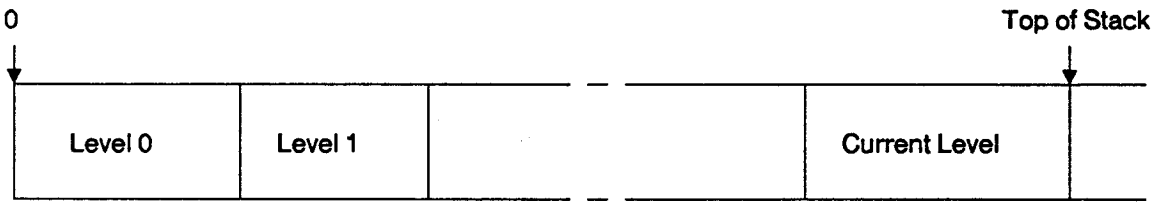


Figure 5.2 Program Activation Stack

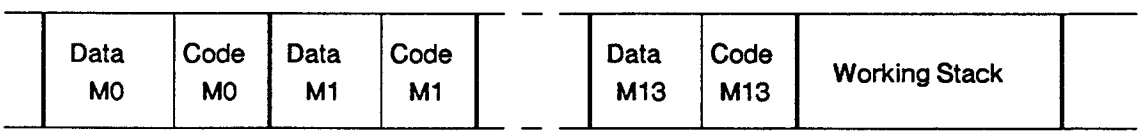


Figure 5.3 Program Activation Record

5.2 The Linking-Loader

Overview

Due to the concept of treating the activation of a program similar to the activation of a procedure, due to the support of separately compiled modules in Lilith's M-code architecture, and due to the format of object code files, Medos-2's linking-loader turned out to be relatively straight forward to implement. The actual implementation is simple and therefore efficient. In contrast to linkers and linking-loaders in most other systems, Medos-2's linking-loader supports the linking of separate modules by simple compatibility checks.

The program activation concept is presented in Chapter 3.3. When a program is called, a program activation record is pushed on top of the program activation stack, and it is removed, when the called program terminates. A procedure activation record contains essentially only the procedure-mark, the variables declared locally to the procedure, and some temporarily used variables. The program activation record contains, in addition to the information corresponding to the information in the procedure activation record (i.e. the variables declared globally to modules loaded for the program and a coroutine used as working stack for the execution of the main program), the *code* of the modules loaded for the program. Figure 5.2 and Figure 5.3 show the memory layout of the program activation stack and the program activation record.

The concepts in the M-code architecture which simplify the compilation of separate modules, turn out also to simplify the linking-loader's job. The addressing of global data and all procedures relative to base-addresses, the indirect addressing of imported modules by aid of the *data frame table*, and the position independent code simplify both the linking of object code modules and the relocation of object code when it is loaded into main memory. Imported modules are only referenced by their (actual) *module numbers* in the linked code. A module number is the index of the corresponding module's entry in the *data frame table*. Thus, the linking of a program is reduced to inserting module numbers of referenced modules into the referencing instructions. The location of a module in main memory is stored at only two places in main memory. The module's entry in the data frame table contains the address of the module's data frame. The first word of the data frame contains the address of the module's code frame divided by two. Figure 2.2 illustrates this.

The format of the object code file (and the possibilities provided by it) influences the complexity of a linker or linking-loader considerably. The format of the object code file for Medos-2 makes it possible to provide both a simple and efficient resident linking-loader. Its syntax is given in Appendix 2.

The object code file syntactically consists of a sequence of tables. The first two words of each table describe the table's type and the number of words following the first two words in the table. This simple and uniform encoding of the object code information into tables enables the linking-loader to scan object code files by small routines. Six types of tables are defined: *Version, Header, Import, DataText, CodeText,* and *Fixup*. The *version table* identifies the M-code version which is used in the object code file. The *header table* contains the module name of the module

stored on the file and the size of its data frame. The *import table* contains a mapping of module names to local module numbers. The *data-text table* and the *code-text table* contain information to be loaded into the module's data frame and code frame, respectively. The *fixup table* contains a list of code locations where local module numbers must be replaced by actual module numbers.

The object code file supports the linking of separately compiled modules by only one simple but very powerful concept. In order to guarantee that linked together modules make at least a syntactically correct use of each other, the linking-loader may only link together modules whose compatibility has been asserted by a compiler. The provided mechanism is based on the possibility to uniquely identify a module's interface in the object code. This is done by so-called *module names*. A module name consists of the (first 16 characters of the) identifier, which specifies the module's name in the program text, and a 48 bit unique time-stamp, the so-called *key*. The key is generated by the compiler when the module's interface part (i.e. definition module) is compiled. The generated value of the key is not essential if the module provides no interface to other modules, i.e. if it is not split into a definition part and an implementation part. The linking-loader in Medos-2 links only modules if the module name in a reference (i.e. in an import table) is equal to the module name of the referenced module (i.e. the module name in the referenced module's header table).

The initialization of each separate module of a Modula-2 program must adhere to two rules: Each module of a program should be initialized exactly once and, with the exception of modules which directly or indirectly import each other, imported modules should be initialized before the importing module is initialized. These rules defined by the programming language cause the initialization code of a program's modules to be executed in an order which does not violate the two rules. This may be guaranteed by either linking-loader (linker) or compiler generated code. For Medos-2, the latter solution was chosen. Although the generated code for a module is a little longer (actually: 3 * (imported modules + 1) Byte), the reduced complexity of the resident linking-loader compensates for the increased code-size in most cases on the Lilith computer.

A third problem concerning the topic of linking of Modula-2 programs is the definition of a program's main module. From the collection of separate modules of a Modula-2 program, it generally cannot be determined which of the modules is the main module of the program. This information, however, is needed when the loaded program has to be executed. It is therefore defined by Medos-2's linking-loader that a program's main module is the module which the linking-loader reads (and loads) first from an object code file when the program is activated.

Module Executer

The linking-loader is programmed in module *Executer* declared locally to module *Program*. Its structure is illustrated below. Only declarations are shown which are essential to understanding the linking-loader and the way it is executed.

```
MODULE Executer;

  FROM SYSTEM IMPORT PROCESS, NEWPROCESS, TRANSFER, ADDRESS, ... ;
  FROM FileSystem IMPORT File, ... ;

  IMPORT
    Monitor, Status, MainProcess,
    GetStackLimit, SetStackLimit, GetStackTop,
    heaptop, heapbottom, ... ;

  EXPORT Call;

  CONST
    pnlength = 16;
    modules = 128;        (* maximal number of loaded modules *)
    imports = 40;         (* maximal number of imported and   *)
                          (* not already loaded modules       *)
    callworkspace = 40;
    loaderworkspace = 480 + 12 * imports;

  TYPE
    ProgramName  = ARRAY [0..pnlength-1] OF CHAR;
    ModuleNumber = [0..modules];
    ModuleIndex  = [0..modules-1];
    FrameTable   = ARRAY ModuleIndex OF ADDRESS;

  VAR
    moduletop: ModuleNumber;          (* loaded (imported) modules *)
    dft[40B]: FrameTable;             (* data frame table *)

    programname: ProgramName;         (* parameter to ProgramLoader *)
    loadbase, maxloadadr: ADDRESS;    (* variables used by *)
    state: Status;                    (*   ProgramLoader *)
    programproc: PROC;                (* result to Call *)

    caller, loader: PROCESS;

  PROCEDURE Error(st: Status);
  BEGIN state := st END Error;

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

    VAR
      oldmoduletop: ModuleNumber;
      oldheapbottom: ADDRESS;
      monst: Monitor.Status;
      son: PROCESS;
```

```
  BEGIN
    state := normal;
    (* programname := pn (with checks) *)
    IF (state = normal) AND MainProcess() THEN
      loadbase := GetStackTop() + callworkspace;
      IF heaptop - loadbase >= loaderworkspace THEN
        maxloadadr := heaptop - loaderworkspace;
        oldmoduletop := moduletop;
        IF NOT shared THEN
          oldheapbottom := heapbottom;
          heapbottom := heaptop;
        END;
        NEWPROCESS(ProgramLoader,maxloadadr,loaderworkspace,loader);
        TRANSFER(caller, loader);
        IF state = normal THEN
          NEWPROCESS(programproc, loadbase, heaptop-loadbase, son);
          Monitor.Call(son, shared, monst);
          state := VAL(Status, ORD(monst));
        END;
        IF shared THEN
          SetStackLimit(heaptop);
        ELSE
          heaptop := heapbottom;
          heapbottom := oldheapbottom;
        END;
        moduletop := oldmoduletop;
      ELSE Error(maxspaceerr)
      END
    ELSE Error(callerr)
    END;
    st := state
  END Call;

  PROCEDURE ProgramLoader;

    CONST
      keysize = 3;
      modidsize = 16;

    TYPE
      ModuleIdent = ARRAY [0..modidsize-1] OF CHAR;
      ModuleName =  RECORD
                      key: ARRAY [0..keysize-1] OF CARDINAL;
                      ident: ModuleIdent;
                    END;

      ImportedModule =  RECORD
                          mn: ModuleName;
```

```
                    mnr: ModuleIndex;
                  END;

  ImportNumber = [0..imports];
  ImportIndex  = [0..imports-1];

VAR
  importlist: ARRAY ImportIndex OF ImportedModule;
  importtop, importsearchtop: ImportNumber;

  programmodule: ModuleIndex;  (* Module number of just
                                  loaded program *)
  f: File;

PROCEDURE InsertModule(VAR modnam: ModuleName;
                       VAR mnr: ModuleNumber); ...
PROCEDURE DeleteModule(importnr: ImportIndex); ...

PROCEDURE LoadFile(VAR f: File; prog: BOOLEAN);

  TYPE
    Symbol = (eofsy, codekeysy, modulesy, importsy,
              ctext, dtext, fixup);

  MODULE InCode;

    IMPORT Symbol, ... ;

    EXPORT Getnum, Getsy, Skip, sy, fs;

    VAR sy: Symbol; fs: CARDINAL;

    PROCEDURE Getnum(VAR n: CARDINAL); ...
    PROCEDURE Skip; ...
    PROCEDURE Getsy; ...

  END InCode;

  PROCEDURE LoadModule(prog: BOOLEAN);

    CONST maxarea = 77777B;

    TYPE
      AreaPointer = POINTER TO ARRAY [0..maxarea] OF CARDINAL;
      Area = RECORD
               ap: AreaPointer;  (* pointer to the actual area *)
               atop: CARDINAL;   (* uppermost used index + 1   *)
               top: CARDINAL;    (* uppermost usable index + 1 *)
```

```
                END;

        VAR
          loctab: ARRAY ModuleIndex OF ModuleIndex;
          loctabtop: ModuleNumber;

          data, code: Area;

        PROCEDURE Equal(VAR mn1, mn2: ModuleName;
                        VAR eq, comp: BOOLEAN); ...
        PROCEDURE Find(VAR mn: ModuleName; ... ); ...

        PROCEDURE SkipModule; ...
        PROCEDURE Getmn(VAR mn: ModuleName); ...

        PROCEDURE CodeKey; ...
        PROCEDURE ModuleHeader; ...
        PROCEDURE Imports; ...
        PROCEDURE LoadText(VAR a: Area); ...
        PROCEDURE Fixups; ...

      END LoadModule;

    END LoadFile;

  BEGIN
    ...
    (* Assign the procedure-variable programproc the initialization
       procedure of the program's main module by aid of
       typeconversion. *)
    programproc := PROC(programmodule * 400B);
    ...
    TRANSFER(loader, caller);
  END  ProgramLoader;

END Executer
```

Procedure *Call* determines the memory space available for loading modules, controls the execution of the linking-loader, starts the execution of the loaded program, and returns stack-space, heap-space, and module numbers provided for the executed program. The linking-loader is executed as a coroutine to the running program's main coroutine (working stack) in order to enable the linking-loader to load modules on top of the program activation stack and *not* because of a need for concurrency. How this is done in detail can best be explained by looking at the actions performed by procedure *Call*. The linking-loader is procedure *ProgramLoader*. The structure of this routine is also sketched in the following section.

Procedure Call

Procedure *Call* first copies the name of the called program into a globally declared variable (*programname*). The memory space available for loading modules is determined and also passed to the linking-loader in globally declared variables (*loadbase, maxloadadr*). The number of modules actually loaded (*moduletop*) is saved. If the program is called in unshared mode, also the state of the current heap (*oldheapbottom*) is saved and a new current heap is created. Thereafter, *Call* creates the coroutine which is needed for the execution of the linking-loader (i.e. for the execution of procedure *ProgramLoader*) and transfers control to it (*TRANSFER(caller, loader)*). This coroutine is allocated between *maxloadadr* and *heaptop* (= *stack limit*). The memory layout at this moment is illustrated by Figure 5.4. After the program has been loaded, the linking-loader transfers control back to the caller's coroutine (*TRANSFER(loader, caller)*), i.e. procedure *Call* resumes execution.

Assuming the called program is loaded now, the loaded program may be executed by activating the procedure-variable *programproc* to which the linking-loader has assigned the initialization procedure of the loaded program's main module. Procedure *Call* does, however, *not* activate the procedure-variable directly. An erroneous program could cause the whole system to crash, if it were executed this way. Instead, a new coroutine is created for the execution of the loaded program and this coroutine is handed over to module *Monitor* in order to be executed. The memory layout of the program ready to start is shown in Figure 5.5. How module *Monitor* executes the program will be explained in the next section. After the execution of the program, i.e. by return from procedure *Monitor.Call*, the number of loaded modules and the current heaplimit are adjusted as required.

Procedure ProgramLoader

Object code files are parsed by a simple parser consisting of the procedures *LoadFile, LoadModule, CodeKey, ModuleHeader, Imports, LoadText*, and *Fixups*. Each of these routines parses a syntactical unit of the object code file. The names of the routines describe their purpose, apart from procedure *CodeKey* which parses a version table and procedure *LoadText* which is used to parse both a data text table and a code text table. The syntax of the object file is given in Appendix 2. Two other routines help parsing the object code file: *SkipModule* and *Getmn* (get module name).

The routines *Getsy, Getnum*, and *Skip* defined in module *InCode* local to procedure *LoadFile* are used to scan the object code file. Procedure *Getsy* reads a table-header and returns the table's type in variable *sy* (symbol) and the number of information words in variable *fs* (frame size). Procedure *Getnum* reads one of the information words in a table and decrements *fs* by one. Procedure *Skip* "jumps over" the rest of a table. All these routines check of course the syntax of the read object code file.

Procedure *LoadProgram* lets procedure *LoadFile* parse the object code files needed for the loading of the called program. The globally defined procedure-variable

programproc is assigned the initialization procedure of the loaded program's main module if the linking-load operation is successful, and the procedure *ProgramLoader* returns thereafter the control to the running program's main coroutine, i.e. to procedure *Call*.

The linking-loader operates on several essential data structures, namely a list of imported but not already loaded modules, a translation table from local module numbers to actual module numbers, the data frame table at a fixed location in memory, the loaded module's module names, and the data frame and code frame of the module currently being loaded.

The list of imported but not already loaded modules is represented by the variables *importlist* and *importtop*, declared locally to procedure *LoadProgram*. An entry in array *importlist* consists of an imported module's module name and its (actual) module number. *importtop* modules are contained in the list and described by the entries 0 to *importtop*-1. Procedure *InsertModule* inserts a module in this list, assigns it its module number, and initializes the modules's data frame entry to *NIL*. A certain module may be deleted from the list by a call of procedure *DeleteModule*.

Procedure *Imports* inserts the modules in *importlist* which are contained in the import table of the currently loaded module's object code and are not already loaded or entered in the importlist. A module is deleted from *importlist* when the module is loaded, i.e. when procedure *ModuleHeader* encounters the module during the parsing of an object code file.

Imported modules are referenced by so-called *local module numbers* within the object code of a module. A module's import table is the mapping between module names and local module numbers: Local module number 0 means the importing module itself, the *i*'th module name in the import table is the module with local module number *i*. The array *loctab* declared locally to procedure *LoadModule* is a translation table from local module numbers to (actual) module numbers. 0 to *loctabtop*-1 are valid local module numbers for the module just being loaded. The translation table is set up by procedure *Imports* during the pasrsing of the import table. This is possible, because any imported module has a module number: It is either already loaded or in the importlist. The translation table is used by procedure *Fixups* to replace local module numbers by module numbers in the loaded code.

The *data frame table (dft)* is an array of addresses allocated at a fixed memory location. One entry is reserved for each loaded or imported module. A module's number is the index of the module's entry in this table. The address of a module's data frame is stored in this table if the module is loaded. Entries of modules which are imported but not yet loaded are initialized to *NIL*. The entries in the data frame table are managed like a stack: The entries with indices from 0 to *moduletop*-1 are allocated. Variable *moduletop* is incremented by one whenever a (free) module number is required for a module, i.e. when a module is inserted into *importlist* or when a program's main module is loaded. After the execution of a program, procedure *Call* resets the value of *moduletop* to its value before the corresponding program was activated.

The linking-loader uses the module name and the module number of all loaded modules in order to be able to link the called program to the already activated

programs. The module name of a loaded module is stored in the main memory just before the module's code frame, i.e. with a small negative offset relative to the first word of the corresponding module's code frame. The actual memory layout is illustrated in Figure 5.6.

The linking-loader pushes all the modules which have to be loaded for the called program on top of the program activation stack. At most one module is in the state of being loaded (pushed) at any time. The data frame and code frame of the module being loaded are described by two area descriptors, called *code* and *data*, declared in procedure *LoadModule*. Each area descriptor contains a pointer to an array of CARDINALs (positive numbers from 0 to 65'535) indexed from 0 to 32'767 (field *ap*). The pointer is initialized such that the element with index 0 in the array is allocated in the first word of the frame described by the area descriptor. Procedure *LoadText* loads the data frame (the code frame respectively) by storing information into the array pointed to by field *ap* in the corresponding area descriptor. Only the memory space corresponding to indices from 0 to *top*-1 of the array is allocated for the frame and may therefore be used to load information into it. The memory space corresponding to indices 0 to *atop*-1 is the actually used part of the reserved memory space for the frame. The latter information is used for the storage allocation for code frames because the size of the code frame of a module is not stored in the object code file. The loading of code and data frame of a module is illustrated by Figure 5.7.

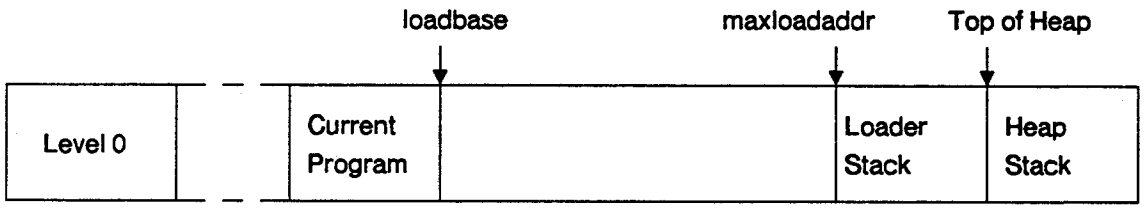


Figure 5.4 Layout after Activation of Loader

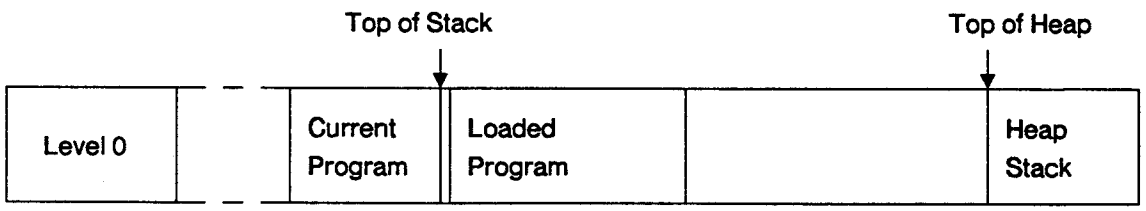


Figure 5.5 Layout before Activation of Loaded Program

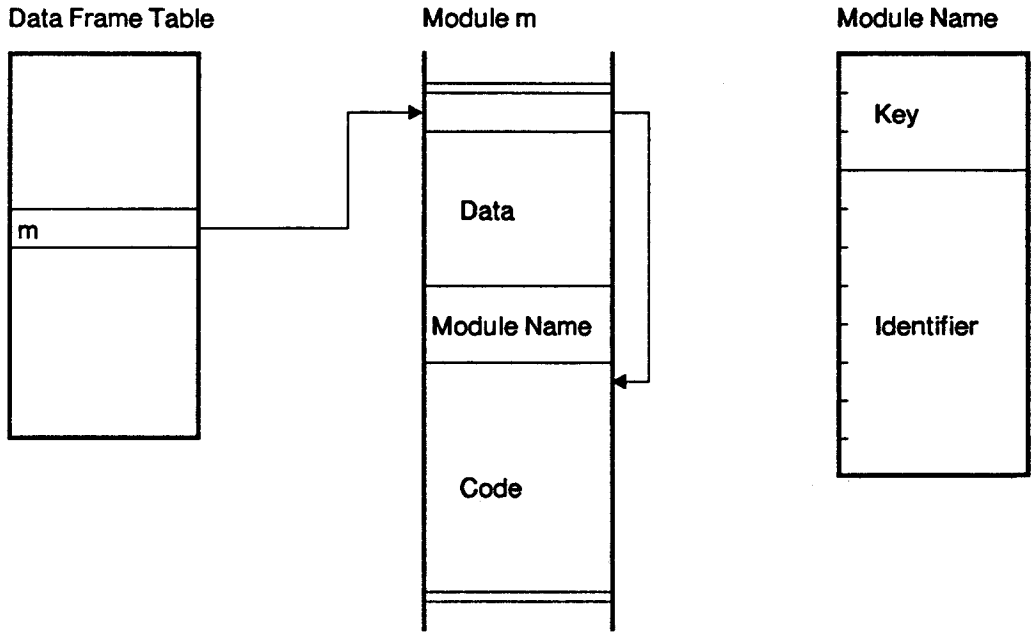


Figure 5.6 A Loaded Module

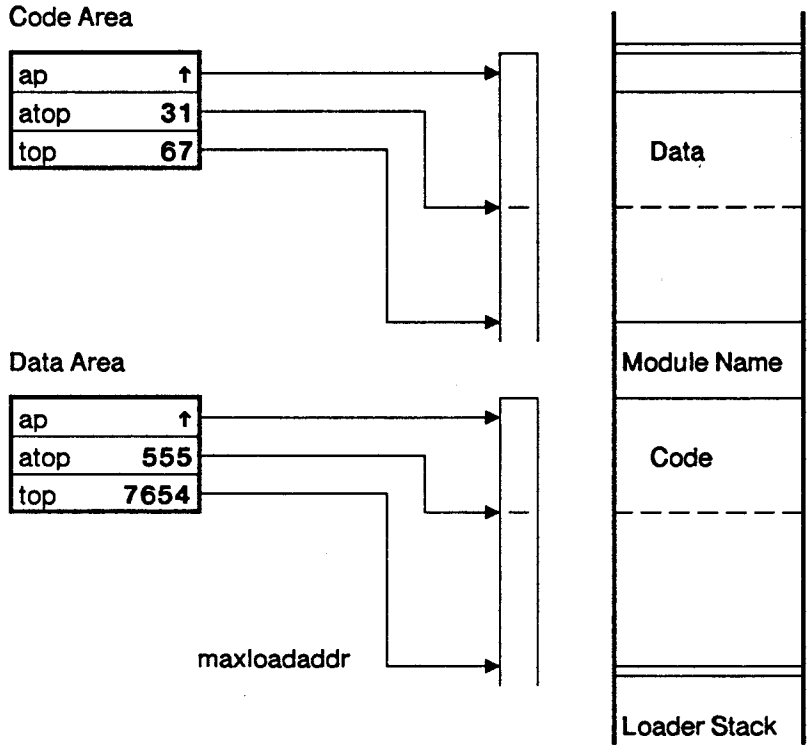


Figure 5.7 Loading a Module

5.3 Execution and Termination of Programs

Module *Monitor* controls the execution of a program, i.e. it activates the coroutine which is set up for the execution of the initialization part of the program's main module. (See Appendix 1.9.) A simple exception handling is provided which enables the caller of a program to resume execution after the termination of the called program. Medos-2 cannot guarantee that the caller of a program always resumes execution after the occurrence of any "fatal" error as the system provides no uncircumventable protection mechanisms against bad behaviour of programs. Experience shows, however, that the system recovers from the most frequently occurring errors.

Module *Monitor* also provides routines which return the *current level* and the *shared level*, a routine which enables a program to terminate execution at any place in the program and with any status (*Terminate*), and a routine for the announcement of termination procedures (see Chapter 3.4). Other provided routines control the clock and enable the reading of characters from the connected keyboard. The implementation of these facilities will not be discussed here. It must, however, be known that the keyboard is periodically scanned for input by the clock process, as the keyboard interface does not send interrupt to the Lilith's processor. The clock process is implemented in module *Monitor*.

In the M-code architecture, a *trap* is invoked when an erroneous condition is detected by the processor (M-code interpreter). A trap may also be unconditionally invoked by an M-code instruction in the executed code. A trap causes a transfer of control to another process (or coroutine) just like an (unmaskable) interrupt. The condition causing the trap may be derived from the suspended coroutine's descriptor. To better understand this mechanism, consider the processor as a peripheral device causing interrupts. The processor (-device) is controlled by the operating system. The trap handler is an *interrupt-driven* process handling the traps caused by the process executing programs. (It is assumed that the other two processes in the operating system, namely the trap handler and the clock process, do not cause traps. Both are programmed in module *Monitor* and are assumed to be correct. The operating system treats all traps the same way: A trap terminates the execution of the running program and lets the caller of that program resume execution.

It is expected from an ordinary process scheduler that suspended processes are resumed at the place where they were suspended. Since the trap handler starts to execute because an erroneous condition occurred in the suspended process, it makes no sense to restart the process unchanged. The *trapped* process must either never be resumed, or it must be changed such that it makes sense to resume it. In Medos-2, the latter alternative is chosen for conceptual reasons. The trapped process is changed by popping the uppermost program activation record from the program activation stack. Then the trapped process resumes execution where the popped program was called, i.e. in the program which called the trapped program at the place where it was called. This operation is easy to perform because the working stack of an activated program is kept in a separate coroutine. The algorithm used is illustrated in the following piece of program:

```
MODULE ExceptionHandler[15];

  FROM SYSTEM IMPORT PROCESS, NEWPROCESS, TRANSFER, ADR, SIZE, WORD;

  EXPORT Call;

  TYPE InterruptVector = RECORD driver, interrupted: PROCESS END;

  VAR
    cpuIV[14]: InterruptVector;
    father, son: PROCESS;

    stack: ARRAY [0..99] OF WORD;

  PROCEDURE Call(VAR p: PROCESS);
    VAR grandfather: PROCESS;
  BEGIN
    grandfather := father;
    TRANSFER(father, p);
    p := son;
    father := grandfather
  END Call;

  PROCEDURE TrapHandler;
  BEGIN
    LOOP
      TRANSFER(cpuIV.driver, cpuIV.interrupted);
      son := cpuIV.interrupted;
      cpuIV.interrupted := father
    END
  END TrapHandler;

BEGIN
  NEWPROCESS(TrapHandler, ADR(stack), SIZE(stack), cpuIV.interrupted)
  TRANSFER(cpuIV.interrupted, cpuIV.driver)
END ExceptionHandler;
```

In the example, the globally declared variable *father* represents the caller of the running program's coroutine and the variable *son* is the most recently suspended coroutine, i.e. the coroutine which caused the most recent program termination. *cpuIV* is the interrupt vector entry which is used when a trap occurs. Procedure *Call* saves its father in variable *grandfather* and transfers the control from its own coroutine to the coroutine passed over to it as a parameter. The started coroutine is the working stack of the running program and the suspended coroutine (saved in variable *father*) is the working stack of the caller of the running program. The running program terminates execution with a trap. The trap handler is invoked, copies the suspended coroutine into the globally declared variable *son*, replaces the suspended coroutine by the coroutine of the caller's program (variable *father*), and

transfers control to it, i.e. to routine *Call*, where the trapped program was activated by a coroutine transfer. Here, the trapped coroutine (*son*) is copied to the parameter *p* and the variable *father* is reset to its value before the program activation. From the value of the coroutine variable returned by procedure *Call*, the reason for the termination of the program may be derived. The routines in the module execute on priority 15, i.e. with all interrupts disabled.

The shown local module *ExceptionHandler* could have been programmed in module *Program* because it imports only objects provided by Modula-2. The reason for handling exceptions (traps) in module *Monitor*, the hierarchically lowest module in the operating system, is that the actual implementation provides several additional features:

The execution of the running program may also be terminated by the clock process. If the user of the system types CRTL-C, the keyboard scanner (i.e. the clock process) tries to terminate the execution of the running program. The termination of the running program is delayed until the running program does not execute a routine in the resident operating system. This condition is tested once for each clock interrupt, i.e. once every 20 msec.

The management of program levels and of termination procedures as well as the activation of the termination procedure in the opposite order of their announcement after the termination of a program is implemented in module *Monitor*. These features must be provided at a low level in the system in order to enable other modules in the system to use them.

Several perhaps less important features like the automatic writing of memory dumps to the disk and the killing of the system if a trap occurs during the execution of a routine in the resident system cause additional complexity for the exception handler. It should, however, be stated that the programming of even this delicate program killing mechanism is done entirely in Modula-2. Code-procedures are only used for the control of the disk drive in order to make memory dumps after program crashes and for invoking a trap (procedure *Terminate*).

6 Implementation of Files on Disk

A brief overview of the file system interface is given in section 6.1. The standard file storage medium for Lilith are disk cartridges for Honeywell Bull D120/D140 disk drives. The orgranization of files on such cartridges by module *DiskSystem* is explained in 6.2. Data security aspects are commented upon in 6.3. The performance of a computer system depends heavily on the performance of its file system. The average data throughput to and from disk files has been increased by allocating sequential disk sectors to files and by buffering algorithms mainly supporting sequential accesses to files. This is discussed in 6.4.

6.1 Files in Medos-2 (Overview)

Files are used for three main purposes, namely for long-term storage of information (on permanent, named files), for communication among programs, and for secondary storage of information (on temporary, unnamed files).

Files provided by Medos-2 are uniquely identified byte-sequences. Each file is stored on a certain medium and may have a textual filename. The interface to files is provided by module *FileSystem*. It exports the type *File* and operations on variables of this type for creating, opening, naming, writing, reading, positioning, and closing files in a simple and convenient way. Please refer to Appendix 1 for more details.

The provided interface is so complete that the skilled programmer may access files randomly, modify the information in files, implement specialized I/O routines or packages, control the lifetime and protection of a file, and even force the buffered information to be stored on the corresponding file's storage medium. These latter facilities have proved to be valuable for the implementation of "operating system independent" I/O packages like module *InOut* [Han82] and *CompFile* [Gei83]. A relational database system [KMP83, RRU82] and a file system providing hierarchically organized files [Sug82], and even Medos-2's file system itself (see next section) use these "low-level" facilities heavily.

The most unusual feature provided by module *FileSystem* is, however, that any program (executed in unshared mode, see section 4.2) may declare that it provides a physical implementation of files (i.e. an access-path to files) on a certain uniquely identified medium. Thus, there is a sharp distinction between the abstract definition of a file provided by module *FileSystem* and the implementation of files on a certain (type of) medium provided by a resident or non-resident module.

The declaration of a certain implementation of files is made by a call of procedure *CreateMedium*, which announces two procedures to module *FileSystem* together with the supported medium's identification. Whenever an operation has to be performed on a file and cannot be handled locally by module *FileSystem*, one of the two routines announced for the corresponding medium is called. One of the announced routines performs all operations on filename directories, i.e. insert, delete, lookup, and rename filename of a certain file, the other announced routine performs all other defined operations on files, e.g. create, open, and close a file.

Although a client of module *FileSystem* may directly invoke the two routines which

provide access to files on a certain medium (procedure *FileCommand* and *DirectoryCommand*) and may also, for example, access the provided file buffer, only few implementation dependent details are visible to clients of module *FileSystem*. Files may therefore be implemented on a large class of media or devices, e.g. a teletype, paper tape, magnetic tape, disk, main memory, and they may even be accessed remotely. Pseudomedia or -devices, like printer spoolers, may also be provided. Several modules have been developed which provide implementations of files on a certain medium, i.e. provide implementations of the two routines which must be anounced to module *FileSystem*.

Module *DiskSystem* implements files on cartridges for the Honeywell Bull D120/D140 disk drives. This module is included in the resident part of Medos-2 because users normally have their files on cartridges for this type of disk drives and because at least one file medium must be accessible by the resident system (in order to be able to load programs).

Other (types of) media are supported by non-resident modules. Main memory resident files are, for example, used for inter-pass files by the Modula-2 compiler. Such files are provided by module *MemoryFiles*. Module *TerminalFiles* makes the keyboard (standard input) and the display (standard output) accessible as files. The Magnet local area network [Hop83] makes files accessible as ordinary files over a network. A printer spooler has also been implemented this way.

It should be remarked that all such "implementations" of files import module *FileSystem* and not vice-versa as one might expect. Module *FileSystem* is therefore one of the hierarchically lowest placed modules in Medos-2. The hierarchy of resident modules is shown in Figure 4.1.

Internal File Identifications and Filenames

All files supported by Medos-2 have a unique identification, the so-called *internal file identification* and might also have an external (symbolic) *filename*. Both the internal file identification and the filename 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 the internal file identification are called the *internal medium identification* and the *local file identification*. The corresponding two parts of a filename are called the *mediumname* and the *local filename*.

The uniqueness of an internal file identification is guaranteed if there exists no pair of media with the same internal medium identification, and a given local file identification is used only once for a file on a certain medium.

In the present version of Medos-2 (version 4.2), most file implementations do not support a unique internal medium identification. This has not turned out to be critical as long as only one medium of a certain type can be accessed at the same time from one machine and internal file identifications are only stored "permanently" on the medium upon which the identified file is stored. The introduction of a local area network, through which files may be accessed remotely, will enforce a more serious handling of the internal medium identification in future versions of the system (e.g. by introducing a medium label on each removable medium).

6.2 The Organization of Disk-Files

Module *DiskSystem* implements files with disk cartridges for Honeywell Bull D120/D140 disk drives. (See also Appendix 1.3.) The disk driver is provided by a second module called *D140Disk*. The design of these modules compromises between the following main requirements for the implementation of files:

- fast access, in particular if strictly sequential,
- robustness against hard- and software failures,
- recoverability from inavoidable errors,
- accomodation of a large number of (mostly short) files, and
- economical usage of disk and main memory storage space.

The implementation of files in module *DiskSystem* is split up into two principal parts represented by two local modules in module *DiskSystem*: Uniquely identified, temporary or permanent, nameless files are provided by module *VirtualDisk*; naming of files is supported by module *Names*. The following two sections describe roughly how files and filenames are provided by the two modules.

Files

All files on a cartridge for the Honeywell Bull D120/D140 disk drive are described by the so-called *file directory*. The file directory is a file. It has a fixed length (196'608 Byte) and is allocated at a fixed location on the disk. Each of the file directory's 768 entries (file descriptors) may describe one file by its local file identification, length, disk allocation, lifetime, protection, number of last modification, and date of creation and last modification. Only one file descriptor is stored per disk sector. Figure 6.1 shows the description of a file on a disk.

In order to avoid compaction of files (which is time consuming and prone to failures), so-called *pages* are allocated to a file on demand and addressed indirectly through a pagemap. A page contains eight 256-Byte sectors. The 8 sectors of a page are allocated equally spaced in a track on the disk (interleaving factor = 12). The pagemap is part of the file descriptor and has space reserved for 96 page-pointers. This number limits the maximum size (length) of a file to 96 * 2048 Byte. (A newer version of module *DiskSystem* removes this limitation at a cost of about 1 kword main memory for program code [Ruc82].)

The free pages on a disk are the pages which are not allocated to any file on the disk. In order to speed up the allocation of a free page to a file, module *VirtualDisk* maintains a memory resident *page allocation table* of 392 word (one per cylinder). Each bit in a word indicates whether or not its corresponding page is allocated to some file. The memory resident allocation table makes it feasible to give a file a good allocation: Whenever a free page is allocated for a file, the free page closest to the page allocated at the end of the file is searched and allocated to the file. This strategy minimizes the overhead for seeks if a file is read or written sequentially.

Free entries in the file directory are described by a *file number allocation table* of 768 / 16 = 48 word. Each bit in this table indicates whether or not the corresponding file descriptor in the file directory describes a file. This table is used as an aid whenever

a file is created. If it turns out that a file descriptor is not free although it was marked free in the file number allocation table another number of a "free" file descriptor is searched. This is tried until either a free file descriptor is found or the file directory is (believed to be) full.

Upon opening the *DiskSystem* for accesses to a certain disk (e.g. when the machine is booted), the entire file directory is scanned and the page and file number allocation tables for the disk are computed. Inconsistencies found during this initial scan of the file directories are either written out on the screen and cause the system to reject the disk or are automatically removed. For example, a temporary file (if found) is removed without comment.

The uniqueness of a file's local file identification is guaranteed by generating a new local file identification whenever a file is created on the medium. This is done in the following way: The local file identification consists of two parts, namely the number of the file's entry in the file directory (0 to 767) and a so-called version number (0 to 32'767). The version number is always stored in the entry and is incremented by one (modulo 32'768) whenever the entry is allocated for a new file. This algorithm works if it is assumed that the time until the same local file identification is generated again is so long that all occurrences of a certain local file identification are deleted before it is generated the next time.

An opened file is described by a so-called *virtual disk descriptor* in main memory. The virtual disk descriptor contains the most often used information from the corresponding file descriptor, the level number of the owning (activated) program, and information describing the accesses to the file i.e. the file's local file identification, size, protection, permanency, 12 of its page-pointers, the owning program's level number, current state, current position, buffer, flags indicating information changed in the virtual disk descriptor, and statistics over the last 16 disk accesses for the file. A file may only be opened once at a time if the file is provided by module *DiskSystem*. This restriction makes it possible to include information describing the accesses to a file (i.e. current position, current state, etc.) in the descriptor describing a file (with information like local file identification, size of the file, its disk allocation, etc.). 16 virtual disk descriptors are allocated permanently. This limits the maximum number of concurrently opened files to 16. Two of these files are used by module *DiskSystem* itself for accessing the file directory file and the name directory file (see next section). A virtual disk descriptor is shown in Figure 6.2. This figure shows also a variable of type *File* and a buffer bound to this file variable.

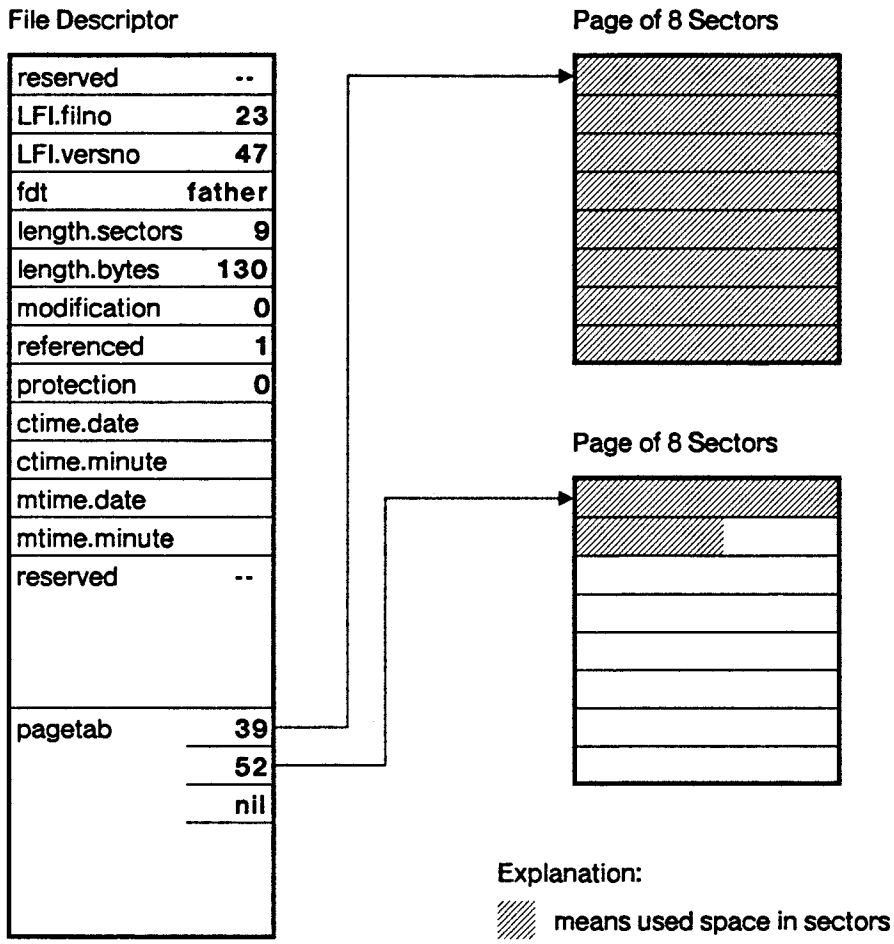
Buffers are allocated on demand from a pool of 16 buffers. Each buffer is described by a *buffer descriptor* which also contains the corresponding buffer space (256 Byte). Other information in the buffer descriptor is a pointer to the virtual disk descriptor for which the buffer is allocated, the number of the buffered sector relative to the beginning of the corresponding file, a page-pointer pointing to the disk page within which the buffered sector resides, the last response from the disk driver, and a *clock* used by the buffer management to distribute buffers among opened files. The buffering technique used is described in Chapter 6.4.

The allocating of a fixed number of virtual disk descriptors and buffer descriptors is

due to the fact that the system provides no general scheme for the dynamic allocation of variables (in the heap). (See also Chapter 5.1.) Robustness considerations also speak against a dynamic allocation of these descriptors. The number of buffer descriptors is chosen to be at least equal to the number of virtual disk descriptors. Deadlock situations are prevented this way. Each opened file may lock one buffer descriptor to the corresponding virtual disk descriptor because a variable of type *File* may have pointers to the corresponding buffer.

Filenames

Local filenames of permanent files are typically stored in the so-called *name directory*. The name directory also has a fixed length and location on the disk. It contains 768 *name descriptors*, one for each possible file on the disk. Its i'th entry is used for the name descriptor of the i'th file in the file directory. Thus, it is very simple to find the name entry to a certain file, given the file's local file identification. A name descriptor of a named file contains the file's local filename, i.e. an identifier or a sequence of identifiers separated by dots, and the file's local file identification. (See Figure 6.1.) The procedure *Lookup(f, <filename>, new)* is used to search the local filename in the name directory. The corresponding file is opened with the aid of its local file identification, if the name is found in the name directory.



Name Descriptor

localfilename 'Example.MOD'	
nk	1
LFI.filno	23
LFI.versno	47
reserved	--

Figure 6.1 File Description on Disk

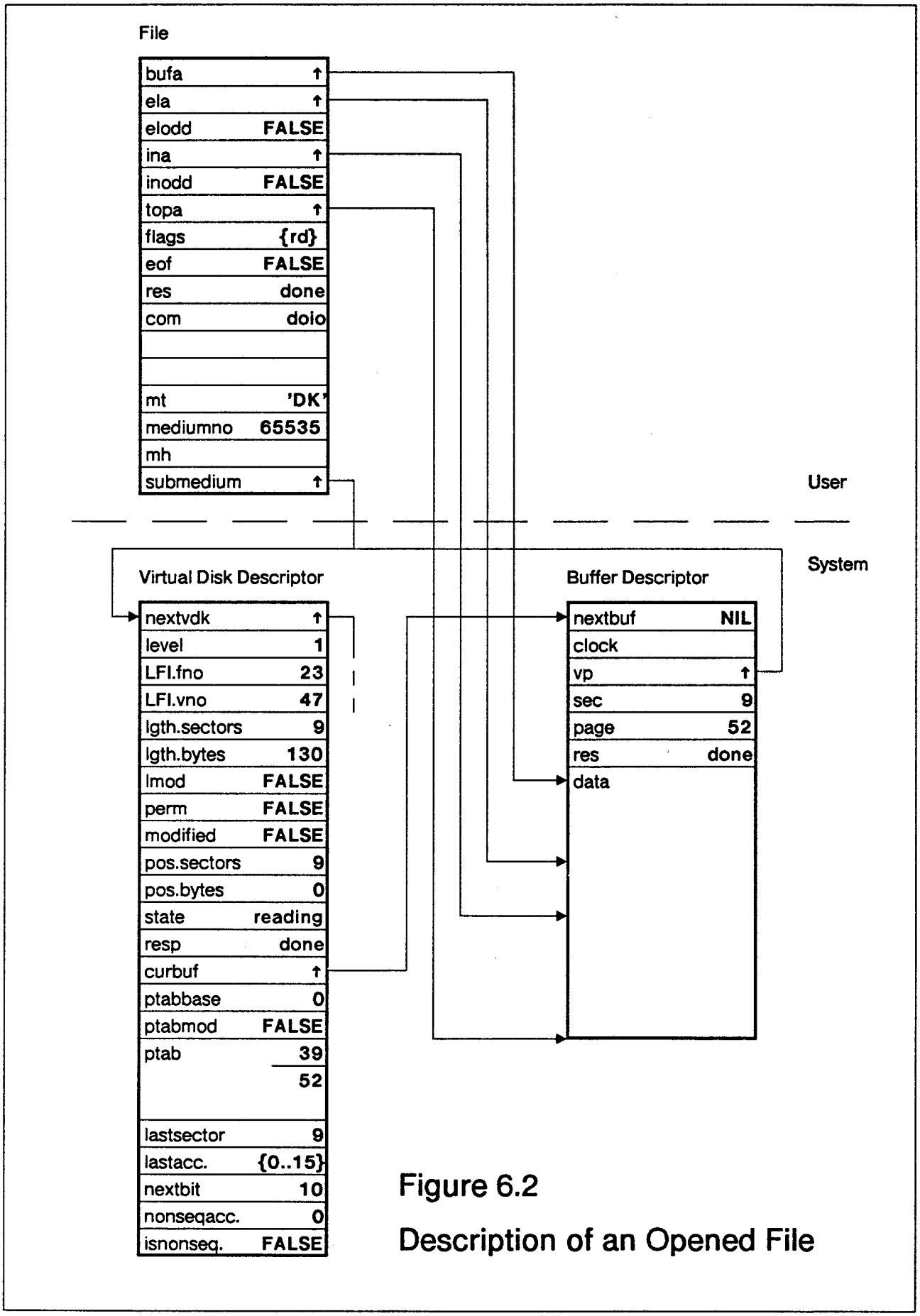


Figure 6.2

Description of an Opened File

6.3 Data Integrity Aspects

It is well known (but often forgotten) that no real computer system can guarantee an absolute integrity of stored data. This is especially true for low-priced, single-user computer systems. On the other hand the integrity of long-term storage is of paramount importance to the user. During the design of module *DiskSystem*, a great deal of attention has therefore been paid to its robustness. Resistance against corrupted information and against misuse as well as recoverability after inevitable program crashes was considered in its design. Most techniques for improving the robustness of a file system, however, also cost in terms of either more expensive hardware components or larger and slower software. Thus, an adequate compromise must be found between the desired file system's robustness and the just acceptable overhead of obtaining it. It can easily be imagined that the chosen compromise depends much on the intended application area of the computer system. Data integrity aspects are much more important for computer systems used for the implementation of large data bases which may be updated interactively, than for computer systems which are "merely" used for document preparation and program development. The difference between the two cases lies mainly in the value of the stored information and in the easiness by which corrupted files (data bases) can be reconstructed from backup (or redundant) information. Even if it is taken into account that most file systems designed for smaller computer systems are not intended to support large data base applications, their reliablilty is definitely too low. For example, in an ill-designed system, corrupted data in a single sector can create such confusion that the valid data on the rest of the disk is practically inaccessible.

Both, erroneous hardware and software may cause corruption of stored information. Several techniques exist to reduce the probability of lost data. For the intended application of the personal computer Lilith, it doesn't seem to be worthwhile to always store files on two different media (mirrored disks [Katzm, Hoe82]). This approach leads to too expensive hardware and is highly inconvenient for users of the machine. Even the approach to generally store the information twice on the same medium (disk) is doubtful because of the inferior utilization of the storage medium and because of the lower attainable maximal write rate. Several systems provide therefore a "secure" storage option which enables a user to specify that a file should be stored twice (e.g. mirrored files, stable storage [SMI80]). This latter approach seems, however, to be so complex that, to my knowledge, no operating system implemented for a personal computer provides this option. Medos-2 is no exception to this. The user of Lilith must from time to time explicitly make back-up copies of important files. If this method is inadequate, the application program (e.g. a data base system) must be written such that the desired degree of data integrity is provided for the specific application. A highly reliable file server accessible via a local area network is an obvious and preferreable alternative.

The methods mentioned above to improve the data integrity in long-term storage deal mainly with erroneous hardware components. Erroneous software is, however, as dangerous as marginally performing hardware. Aside from programming errors in the file system software, two main groups of problems may cause the file system to affect data integrity: Corruption of the operating system by hardware or user

software, and unexpected or enforced system crashes.

File systems generally keep state information about accessible files in main memory. If the information stored in main memory is faulty, the file system may be caused to corrupt information stored on an accessable disk and thereby to enlarge the number of faults. Lilith and many other personal computers provide no absolute protection mechanisms. Any erroneous program may for example overwrite the file system's code and data. To reduce the probability of an overwritten operating system, little can be done but to improve the encapsulation mechanisms (introduce "firewalls"). This can be achieved both by software (more "safe" programming languages, code interpretation) and by modifying the machine architecture (instruction set, hardware). Another independent approach is to tackle the problem of existing faulty information. This problem must be solved anyway in a reliable system, as any computer system may corrupt information in main memory. By introducing redundant information in stored data, inconsistencies may hopefully be detected by simple checks (assertions). The detection of faults enables the file system to prevent the generation of new faults and thereby to keep a local fault local or even to correct the fault. Module *DiskSystem* makes heavy use of this technique. A third method to reduce the probability of information destruction in secondary storage is to make the disk controller sufficiently intelligent that it can itself check the validity of disk operations. Such validity checks can be made nearly uncircumventable. The Alto computer [TML79] minimizes damages caused by software failures in this way. The main drawbacks of this approach are that a standard disk drive and/or disk controller cannot be used and that the validity checks make the allocation and deallocation of disk sectors (pages) of a file awkward. The controller must typically read and check the addressed sector's header before the header may be changed after a further disk rotation (overhead!).

The second group of problems which may cause inconsistencies on a disk concerns the interruption of processing at improper moments. Hardware and software errors as well as users pressing the *boot-button* or switching off the power may cause the processor to stop proper execution. Beside hardware problems, which may make sectors detectably unreadable, the required atomicity property of file operations may also be affected by an unexpected or enforced system crash. Module *DiskSystem* provides high reliability in such cases (failure atomicity) by merely *pasive*, atomic operations on directories (i.e. atomic oprerations without additional data structures or data accesses) [SMB79].

Data Integrity Aspects of the Chosen Data Structures on Disk

In order to control the global damage that could result from local errors on the disk, the information must be stored such that an error on the disk affects as few files as possible. Within affected files, an error should corrupt as few Byte as possible. This ordering of importance is essential for the user of the file system: Files and Byte are the logical units of information which the file system *stores* for the user and which may be affected by errors. It is, however, better to loose, for example, 256 Byte in one file than to loose one Byte each in 256 files. This is because one damaged Byte in a file may have the consequence that the whole file is useless for its user. One

Byte lost in each of 256 files will in most cases have more serious (i.e. less local) consequences for the user of the disk than 256 Byte lost in one file.

Most hardware mechanisms which may cause data corruption on a disk affect the data stored in one or a few sectors. With two exceptions, module *DiskSystem* stores in each disk sector information of (or about) a single file only.

The first exception is the name directory in which the names of eight files may be stored in a single sector. The loss of a file's textual name does, however, not mean that the file itself is lost. The old or a new name can later be assigned to a nameless permanent file.

The second exception concerns the directory file. A file is described by a file descriptor stored in one sector of the directory file. This conforms to the desired property of describing merely one file per disk sector. The exception is that the directory file itself is also described as a file in one sector. If the information in the sector describing the directory file is corrupted and no precautions are taken, all files stored on the disk may be lost. This problem is solved by allocating the directory file as a contiguous file with a fixed length at a fixed location on the disk. The file descriptor of the directory file can therefore be considered as a constant. It can be reconfigured at any time, and it is only seldom inspected and never changed by ordinary software.

It follows from the above explanations that *a corrupted sector affects at most one file seriously*.

The atomicy property of file system operations is achieved by merely *passive* means: The data structures of the two directories are defined such that the state of a file, with one minor execption, always can be changed from one externally consistent state to the next by overwriting only *one* sector. The consequence is that, apart from the exception, the information describing files on the disk is always in a consistent state. Software crashes (or machine stops) therefore causes no serious recovery problems. If a disk write operation fails for some reason (boot-button, power failure, etc.), the addressed sector will probably be unreadable. One such damaged sector generally affects at most one file as was explained above. Two minor consistency problems may arise if a crash (or stop) occurs at an unexpected moment:

Opened temporary files remain described in the file directory after the crash, although it is defined in module *FileSystem* that a temporary file is removed when its owning program terminates execution. This problem is solved by removing temporary files in the file directory when the disk is mounted the next time, i.e. during the initialization of the page allocation table in module *DiskSystem*.

The second problem - the minor exception mentioned above - concerns the name directory. Renaming a named file causes no problems as this is done with only one write operation. The insertion of the local filename of a nameless (and temporary) file into the name directory and the removal of the local filename of a permanent file and thereby converting the file to a temporary file cause problems because the file's local filename is stored in the name directory file and the boolean flag indicating a files permanency is stored in the file directory file. The corresponding file descriptor and name descriptor cannot be updated by merely one disk write operation. The

required atomic property of the insertion and removal operations is solved in the following way: When a nameless file is given a local filename, its name is first inserted into the name directory whereafter the file's permanency is changed to be permanent. If the system crashes (or stops) after the insertion of the file's name but before it is set permanent, the result will be that the (still temporary) file is removed the next time the disk is mounted. The name directory will hereafter contain an entry describing the local filename of a not existent file. This local filename is removed the first time a file with the corresponding name is looked up (but cannot be opened) or when a file described by the corresponding entry in the file directry is given a name. The removal of a file's local filename is handled similarly between the two operations, but in the opposite order. The file's state is first changed to be temporary whereafter it's name is removed in the name directory. The recovery from a crash is identical to that for a name insertion.

Detection of Corrupted Information by Module DiskSystem

Files on the Honewell Bull D120/140 disk drive are described by two directories on the disk cartridge (see Chapter 6.2). A mounted disk and the files stored on the mounted disk are also described by data structures local to module *DiskSystem*. Both the information on the (mounted) disk as well as the information kept locally to module *DiskSystem* may be erroneous. In order to guarantee the integrity of the information stored on the disk, it is essential that corrupted information about stored files be detectable because the use of such erroneous information may cause a disaster [Lam81]. Module *DiskSystem* mainly improves its robustness to misuse, corrupted information, and hardware failures by checking implicit or explicitly stored redundancies at many places in the program. Examples of such redundancy checks are listed below:

Parameter-checking: All provided routines thightly check the legality of a call and the validity of the arguments. For example, all opened files handled by module *DiskSystem* are described by a virtual disk descriptor (see 6.2) within module *DiskSystem*. A variable of type *File* passed as a parameter to procedure *FileCommand* is checked against the information stored in the corresponding virtual disk descriptor before the operation is performed.

Page-Pointers: A page-pointer (see 6.2) describes the address of a disk page, i.e. of eight sectors on a disk. The actual value stored in a page-pointer is the page-number * 13, where the page-number is the disk address of the first sector of a page divided by eight (0 <= page-number < 4704). Whenever a page-pointer is interpreted, it is checked that it is divisble by 13 (without remainder). All single-bit errors in a page-pointer and generally 12 out of 13 arbitrarily modified page-pointers are thus detected.

Page Allocation Table: The page allocation table (see 6.2) describes the free pages of the mounted disk. The table is introduced for efficiency reasons. It allows both a fast *and a good* allocation of free pages to a file. The use of faulty information in the page allocation table is disasterous, if the fault causes a page to be allocated to several files (so-called crossing files). Module *DiskSystem* (and the hardware) provide no lower level checks preventing the allocation of a sector (or page) to two

or more files as does, for example, the Alto computer [TML79]. The information stored in the page allocation table is treated as the absolute truth about free pages on the mounted disk. In order to detect damages caused by memory errors and erroneous programs (overwritten page allocation table), each word of the page allocation table is checked like page-pointers: One word of the page allocation table describes the allocation of merely 12 pages. This can be done with 12 bit. In stead of storing the 12 bit directly in a word of the page allocation table, the 12 bit are treated as a number ranging from 0 to 4095 and are multiplied by 13. Before an entry in the page allocation table is used, its divisibility by 13 is checked, and it is converted to a set of 12 bit. This check detects all single-bit errors and generally 12 out of 13 arbitrary modifications of a single entry of the page allocation table. Furthermore, a free page is represented by a zero-bit. A word set to zero by erroneous software would therefore indicate 12 allocated pages, which is not critical. A page marked free in the page allocation table must be free in order to prevent a disaster. The opposite is not necessary: A page may be marked allocated in the page allocation table although it is not allocated to any file. A disk merely looks full before all pages are truly allocated.

File Descriptors: The number of a file's file descriptor in the file directory file (see 6.2) is stored in the file descriptor itself. Most read or write errors to the file directory file are detectable by checking the correctness of the file number and the divisibility of the page-pointers in the descriptor.

6.4 File Buffering

More than a decade ago, Gene Amdahl [Amd70] speculated that the I/O bandwidth required by a computer system is proportional to its instruction execution rate. Although the required I/O bandwidth also depends heavily on other factors (e.g. the actual application of the computer), it is obvious today that a floppy disk drive used as the main secondary storage device for an otherwise high-performing work station like Lilith, PERQ, or Alto would downgrade the overall performance of the work station unnecessarily. Beside using a better performing hard disk drive, buffering together with a good allocation of sectors to files are the main techniques for achieving higher I/O performance to secondary storage in Medos-2. Section 6.4.1 briefly presents the file buffering concept in Medos-2. Section 6.4.2 gives an estimate of how buffers should be distributed among opened files. Read-ahead for sequentially and randomly accessed files is discussed in 6.4.3, and the organisation of the buffer pool is presented in 6.4.4. Section 6.4.5 shows the results of some performance measurements.

6.4.1 The File Buffering Concept in Medos-2

All I/O appears completely synchronous in Medos-2's file system. Necessary buffering is performed by the file implementations, i.e. by the modules implementing files on supported media or providing an access path to supported media. For the normal programmer, the buffering therefore seems to be provided by the file system. The size of the buffers, the number of buffers, and the buffering method (read-ahead, write-behind, LRU-replacement, etc.) may freely be chosen by each file

implementation and is, with the exception of the buffer size, invisible to file system clients. The exception being four address fields in the client's file variable which describe the buffer containing the Byte at the file's current position. Please refer to section 6.1 and to module *FileSystem* in Appendix 1 for more details.

Although many programmers and especially implementors of database management systems [Sto81] argue against operating systems providing file buffering for secondary storage devices, many operating systems (including Medos-2) provide system buffering of files (e.g. DEMOS [Pow77], UNIX [RT78, and Pilot [Red8o]). The main reasons for this are first, that I/O to secondary storage often turns out to be *the bottleneck* of a computer system, limiting its throughput and its performance and, second, that experience (of operating system designers) indicates an intolerably large gap between what implementors of I/O-packages (run-time systems, database management systems, etc.) *claim can be optimized* by them (knowing about actual applications) and what *is actually optimized* in typical (i.e. most often used) I/O-packages.

Besides the possibility to provide a better *overall* system performance by system buffering, mainly the following arguments speak in favour of system buffering in Medos-2:

Files are used within Medos-2 itself. The resident loader reads object-code files and module *DiskSystem* accesses both directory files (see 6.2) and ordinary files. System buffering saves both code and buffer space otherways used if no system buffering were provided. For example, an operation on a disk-file might cause an access to the file directory file. This simply invokes a possibly recursive call of a file system routine.

The local management of buffers by each file implementation (e.g. module *DiskSystem*) makes it possible to tailor the buffering mechanism to the characteristics of the concrete secondary storage device and the dominant mode of accessing files on it.

Typical arguments from database management system implementors against system buffering, like high overhead for system calls, marginally performing buffer management for sequential accesses to files, or no possibility to flush the contents of system buffers to the secondary storage medium [Sto81] are hardlto the secondary storage medium [Sto81] are hardly applicable to Medos-2. Medos-2 has no additional system call overhead (parameter checking should also be made by user-written I/O-routines), the routines defined in the file system interface make it possiand the buffer manager in module *DiskSystem* recognizes sequential accesses to a file which causes a changed read-ahead, toss-immediatly strategy to be used for the file's buffers (see section 6.4.2). Medos-2 was, however, not designed to especially support substantial database applications.

6.4.2 Distribution of Buffers to Disk Files

Currently, the buffer manager in module *DiskSystem* has 16 permanently allocated buffers, each with space for one sector (256 Byte). The number of buffers is fixed because no dynamic allocation of variables is available to module *DiskSystem* (see

Chapter 5.1). The number of buffers must be larger or equal to the maximum number of simultaneously open disk files because each opened file may lock one buffer. The number must be as small as possible (acceptable) in order not to waste memory space. Experience shows, however, that programs typically uses only a few files at the same time (zero, one or two, seldom more). As disks for the Honeywell Bull D120 disk drive are the main secondary storage medium for Lilith, and as it turned out that several programs (at least during shorter periods) are I/O-bound (e.g. the Modula-2 compiler, the editor, the debugger and others), it is advantageous to improve the I/O throughput to the disk by better buffering techniques.

In Medos-2, most file accesses are sequential, and only a relatively small buffer pool is available (16 * 256 Byte = 4096 Byte). These facts indicate that it is not worthwhile to try to reduce the number of disk accesses by using the few buffers as a simple disk cache, managed by a least recently used (LRU) or second chance (SCH) replacement algorithm.

If the number of disk accesses cannot be significantly reduced, the only other method to improve the I/O bandwidth to a single given disk is to reduce the average time needed for a disk access. From inspection of the different timings of the Honeywell Bull D120 disk drive, it can be seen that the average time needed for an arbitrary head movement is about eight times longer than the average arbitrary rotational delay (latency).

- average of arbitrary head movement (seek) 65 ms
- average of arbitrary rotational delay (latency) 8.33 ms
- sector transfer time (to interface buffer) .333 ms

The average time needed for a disk access can therefore be reduced most effectively by reducing the number of head movements or by reducing the distance of an average head movement. This can generally be achieved by ordering (scheduling) requests for disk accesses such that the total time needed for head movements is minimized. More "general" disk scheduling strategies, such as the scan algorithm [Hoa74], are, however, not applicable, because there is no unrestricted competition among concurrent processes accessing files in Medos-2 (or in other typical single-user environments).

In module *DiskSystem*, the problem is tackled by a good allocation of sectors to files and by read-ahead (prefetching):

Sectors allocated for one file are allocated almost contiguously on the disk. The eight sectors of a page (the allocation unit) are allocated equally spaced around a track (interleaving factor = 12, i.e. 1/4 disk rotation from one logical sector to the next). When a free page is appended to a file, the page is, whenever possible, allocated on the same track as the page allocated at the end of the file. If there is no free page in the same track, a free page in the same cylinder is searched. (Unfortunately, it costs a seek-operation to change the track in the same cylinder on the used drives.) If still no free page is found, the cylinder with free pages nearest to the cylinder of the page at the end of the file is searched and a page is allocated from that cylinder. The almost contiguous allocation of sectors to files obviously minimizes the time needed for head movements, if only one file is sequentially accessed. This is a frequent case (e.g. searching a local filename in the name

directory file, loading programs from object-code files). If more than one file is sequentially accessed, reading-ahead respectively writing-behind on a file would decrease the average number of head movements and/or their average length, provided that sectors of a file are allocated close to each other. This technique requires, however, more than one buffer for one or several of the sequentially accessed files.

To answer the question of how to distribute buffers to files, the following three methods were considered. An *active file* is a file which is either read, written, or modified. The *activity* of a file means the number of sectors read or written for the file in a certain period:

a) Distribute the available buffers equally among active files.
b) Distribute the available buffers proportional to the activity of active files.
c) Distribute the available buffers proportional to the squareroot of the activity of active files.

The following idealized assumptions are made:

1) All necessary seeks need equally much time.
2) Rotational delays and data transfer times can be ignored.
3) Files are accessed sequentially.
4) The cost of reading or writing all buffers of a file is one seek operation.

Denotations

Squareroot function:	sqrt(<Expression>)
Square function:	sqr(<Expression>)
Sum function:	sum(<Expression> | <Range>)
Number of active files:	F
Number of buffers allocated to file i:	Bi
Total number of buffers:	B = sum(Bi | i = 1 .. F)
Activity of file i:	Ai
Total activity:	A = sum(Ai | i = 1 .. F)
Seek frequency for file i:	Si = Ai / Bi
Total seek frequency:	S = sum(Si | i = 1 .. F)

It follows for the total seek frequencies *Sa*, *Sb*, and *Sc* of the three considered strategies:

Strategy a: Bi = B / F

Sa = sum(Ai / Bi | i = 1 .. F) = F * A / B

Strategy b: Bi = B * Ai / A

Sb = sum(Ai / Bi | i = 1 .. F) = sum(A / B | i = 1 .. F) = F * A / B = Sa

Strategy c: Bi = sqrt(Ai) * B / sum(sqrt(Aj) | j = 1 .. F)

Sc = sum(Ai / Bi | i = 1 .. F) = sqr(sum(sqrt(Ai) | i = 1 .. F)) / N

The probability of two (sequences of) disk accesses next to each other being for the same file and need therefore no seek beetween them should also be taken into account. This probability is estimated to be *Si/S* for file *i*. From this estimate it follows that the frequency of *real* seeks *Sri* for file *i* and the total frequency of real seeks *Sr* are:

Sri = Si - sqr(Si) / S
Sr = sum(Sri | i = 1 .. F) or
Sr = S - sum(sqr(Si) | i = 1 .. F) / S

It follows for the three total real seek frequencies *Sra, Srb,* and *Src*:

Sra = (1 - sum(sqr(Ai) | i = 1 .. F) / sqr(A)) * F * A / B
Srb = (1 - 1 / F) * F * A / B = (F - 1) * A / B
Src = (sqr(sum(sqrt(Aj) | j = 1 .. F)) / (A * F) - 1 / F) * F * A / B

Discussion

If *Ai* = A/F for all active files i, it follows that *Sa = Sb = Sc* and *Sra = Srb = Src*.

From the iequalities

1 / F <= sum (sqr(Ai) | i = 1 .. F) / sqr(A) <= 1 and
1 <= sqr(sum(sqrt(Ai) | i = 1 .. F)) / A <= F

it follows that *Sra <= Srb* and *Src <= Srb*, i.e. the distribution of buffers proportionally to the files' activities is not optimal compared to the two other allocation strategies. If *F = 2*, it can easily be shown that *Sra <= Src*, i.e. if two files are active, it is better to distribute the available buffers equally among the active files.

Motivation for Strategy c

The total seek frequency is minimized by setting the following derivative to zero:

dS / dBi = - Ai / sqr(Bi) - sum((Aj / sqr(Bj)) * (dBj / dBi) | i <> j)

Assumption: Only *Bi* and *Bk* are changed. It follows that

Bk = B - Bi - sum(Bj | j # i, k)
dBk / dBi = -1
dS / dBi = - Ai / sqr(Bi) + Ak / sqr(dBk)

Strategy c follows by setting *dS / dBi* = 0, i.e. strategy c minimizes the total seek frequency:

Bi / Bk = sqrt(Ai) / sqrt(Ak)

It has be shown that strategy a) and c) under the given assumptions cause fewer head movements than strategy b). This is interesting because simple intuition

favours strategy b). Commonly used LRU-like buffer replacement strategies tend to distribute buffers proportional to file activity, i.e. tend also to use strategy b).

Buffer distribution strategies a) and c) are difficult to compare, but for reasons given below, it was decided to distribute available buffers *equally* among active files, i.e. to prefer strategy a):

- Strategy c) is awkward to implement.
- It can be shown that strategy a) performs better than c) if merely two files are accessed.
- All three strategies distribute buffers the same way if the activities of all active files are equal.
- The number of buffers is small (16). Each active file locks one buffer and it makes sense to reserve at most 8 buffers for a file. It follows that both strategies tend to distribute buffers similarly if more than two files are active.

6.4.3 Read-Ahead for Sequentially and Randomly Accessed Files

Mainly for reasons of (logical) complexity (memory space!) and reliability, it was decided not to employ a write-behind buffering strategy. Write-behind buffering is also less important than read-ahead because only a minor fraction of all disk accesses are write operations. On the other hand, it was decided the refine the read-ahead strategy by distinguishing between sequentially and randomly accessed files.

In the current version of module *DiskSystem*, the distinction between the two access modes is based on a statistic over the most recent 16 disk accesses for each file. From the number of non-sequential disk accesses among those 16 accesses a rough (rapidly adapting) figure is derived for the number of sequentially accessed sectors on each active file, the so-called *sequence-length*. If the sequence-length of a file is relatively small compared to the number of buffers allocated to it, the file is considered to be randomly accessed, otherwise sequentially.

The buffer management (see next section) and the read-ahead are treated differently for the two access modes:

For randomly accessed files, at most sequence-length sectors are read ahead. Buffers containing information from such files are handled by a LRU-similar strategy (generalized clock algorithm, see next section).

For sequentially accessed files, more sectors may be read ahead as buffers are reserved for it. Whenever a buffer has been used by the file, it is set free (toss-immediately strategy). The number of read ahead sectors is chosen such that the *average* number of buffers filled with information for the file is (roughly) equal to the number of buffers reserved for it. The derivation of the number of sectors to read ahead is given below:

Denotations:

Number of available buffers (for one sector):	B
Number of active (sequentially accessed) files:	F
Number of sectors to read ahead:	L

After the last *L* read in sectors have been processed, the next *L* sectors are read in from the disk. A processed buffer is freed immediately. An active file has always one buffer locked to it. From this the average number of filled buffers can be estimated to be *(L + 1) / 2* buffers. The value for *L* can therefore be chosen higher than *B / F*. *L* should be chosen such that at the moment when sectors must be read again for a file, just *L* buffers are free. The file for which the sectors must be read occupies no buffers, all other *F - 1* files an average of *(L + 1) / 2*. It follows from this

$$L = B - (F - 1) * (L + 1) / 2$$

or

$$L = 2 * (B + 1) / (F + 1) - 1$$

With this choice of *L* in place of *B / F* the number of buffers is virtually increased by the factor

$$(2 * (B + 1) / (F + 1) - 1) / (B / F)!$$

For $B = 16$ and $2 <= F <= 10$ the virtual increase in the number of buffers is in the range 29% - 45%.

6.4.4 The Organization of the Buffer Pool

Module *DiskSystem* includes a relatively small pool of buffers which are managed by the so-called *buffer manager*. Besides the memory space needed for the buffering of one disk sector (256 Byte) each buffer also contains some information needed for the administration of the buffer. The data structure of a buffer is called a *buffer descriptor*, one of which is shown in Figure 6.2. A buffer is either unused or it used for the buffering of a disk sector allocated to a certain (opened) file. In module *DiskSystem*, an opened file is represented by its virtual disk descriptor. Please refer to section 6.2. In the buffer, the buffered sector is identified by a pointer to the virtual disk descriptor of the file, to which the buffered sector is allocated, and by its number relative to the beginning of the file. An opened file may lock at most one buffer at the same time. The remaining, unlocked buffers are managed by a *generalized clock* buffer replacement algorithm assigning dynamically calculated weights [Smi78, EH82]. The generalized clock algorithm was chosen mainly because it is simple to implement and enables different buffer replacement strategies to be provided at the same time. The buffer manager uses the generalized clock algorithem in the following way:

With each buffer a count-field is associated, the so-called *clock*. The buffers are arranged in a circular list. A globally declared pointer, called *last*, points to the most recently referenced buffer. Whenever a buffer is unlocked (from an opened file), it is inserted into the circular list, the buffer's clock is set to *i*, and *last* is set to point to it.

i is a parameter to the unlock-routine.

When a buffer for a certain (logical) sector is locked to a file, the sector is first searched among the linked buffers. If a buffer storing the searched for sector is found, it is removed from the circular list, the pointer to the virtual disk descriptor and the sector number are stored in the buffer, and the buffer is linked to the corresponding virtual disk descriptor. Otherwise, the *last*-pointer circles around the circular list. If the clock of a pointed to buffer is zero, then the buffer is removed from the circular list and locked to the file. Otherwise, the clock is decremented by one, the *last*-pointer advanced to the next buffer, and the process repeated. The disk-address of a buffered sector is represented by a page-pointer, also stored in the buffer. If the page-pointer of a locked buffer does not correspond to a valid disk-address, an unused buffer was locked for the file. If that is the case and if the corresponding sector on the disk contains information belonging to the file, the sector must be read from the disk before the buffer may actually be referenced. Just before a buffer is unlocked, its contents are written to the disk if the file is written or modified, i.e. a write-through algorithm is used for writing back modified buffers.

By chosing different values for *i* when a buffer is unlocked, different replacement strategies can be achieved. For example, the FIFO strategy is obtained if the clock is always set to zero. In this case the clock field is not needed as its value is always zero. The second chance (SCH) strategie is obtained if the clock is set to one whenever a buffer is unlocked. The buffer manager distinguishes between three different cases when a buffer is unlocked: unlocking a buffer reserved for a sequentially accessed file or for a file in write-mode, unlocking a read ahead buffer, and unlocking a buffer referenced by a randomly accessed file.

When a buffer which has been processed by a sequentially accessed file or by a file upon which information is written, is unlocked, its clock is set to zero in order to indicate that the buffer should be considered free (toss-immediately strategy).

The remaining number of sectors to read ahead for a file is indicated by a variable *remaining sectors*. Whenever a sector is read from the disk its value is decremented by one. The clock of a read ahead buffer is set to *number of active files + remaining sectors*. (An *active file* is a file which is read, written, or modified.) This way, the first read ahead sector gets the highest clock value and the clock of the last read ahead buffer gets the value *number of active files + 1*. The intention of this setting of the clock is to indicate that read ahead sectors are very likely to be referenced in the near future. If, however, one buffer storing a read ahead sector must be dropped, the buffer most recently read ahead should be taken.

When a buffer of a randomly accessed file is unlocked, its clock is set to *number of randomly accessed files DIV 2 + 1*. This setting of the clock intends to provide a LRU-like buffer replacement strategy for buffers storing sectors of randomly accessed files and to reserve about the same number of buffers for a randomly accessed file as are on the average "reserved" for a sequentially accessed file (i.e. *(L + 1) / 2*, see section 6.4.3). The buffer manager includes two essential modifications to the generalized clock algorithm for handling unlocked buffers mentioned above, namely a possibility to *clear* buffers and a possibility to *conditionally lock* a buffer for a file:

Beside the possibility to lock and unlock a buffer storing a certain sector of a file, a third routine is provided to *clear* buffers storing sectors whose sector numbers are greater than or equal to a given value. This routine is called whenever an opened file is closed or its length shortened.

A parameter to the lock routine indicates whether a buffer must be locked to the opened file or only if a *free* buffer exists in the buffer pool. A buffer is considered free if its *clock* is zero. Only free buffers are used to buffer read ahead sectors.

6.4.5 Performance Measurements

The value of a nearly contiguous allocation of sectors to files and of reading ahead sectors from sequentially accesssed files can be derived from Figure 6.3 and Figure 6.4. The time needed to read a 131'072 Byte file twice and the time needed to copy the same file on a system without read-ahead is shown in Figure 6.3. The time needed to copy a 131'072 Byte file is shown, measured on a system with and without read-ahead on disk-files. The following program-piece was used to copy a file:

```
.
.
REPEAT
  ReadByteArea(infile, buffer, buffersize, gotbytes);
  WriteByteArea(outfile, buffer, gotbytes);
UNTIL infile.eof;
.
.
```

A similar program-piece was used to read a file twice. The copy time and read time were measured for series of different buffer sizes, ranging from 2 Byte to 32'768 Byte and the size of the next larger buffer being twice the size of the previous buffer. The differences in the time needed for copying a file by a system not including read-ahead and for reading the source file twice by the same system is mainly due to the reduced time needed for seeks when only one file is accessed and to the nearly contiguous allocation of files on disks. How the read-ahead on sequentially read files reduces the copying time is illustrated by Figure 6.4. The increasing copying time towards smaller buffer sizes (< 8 Byte) is due to the execution time for the routines *ReadByteArea* and *WriteByteArea*. These routines need an atypically high initialization time before the first Byte is transferred. The decrease of the copying time for buffer sizes between 256 Byte and 2048 Byte on the system with read-ahead illustrates nicely the combined effect of system provided read-ahead (of at most 2048 Byte = 8 sectors) and a comparable large user-program provided buffer. The higher copying time of the system with read-ahead when larger buffers are used (> 2048 Byte) is caused by lost rotations. The eight read ahead buffers cannot be copied fast enough from the system buffers to the user-provided buffer. Figure 6.5 shows the relative copying time of the two systems. This figure illustrates clearly that the read-ahead strategy only helps when relatively small data elements are transferred at a time. This is, however, the typical case for most programs using Medos-2.

How the execution time of typical programs may be influenced by the read-ahead buffering strategy is shown by three examples: the compiler, the editor, and the debugger.

Column 1 in the following table lists the execution time of the compiler passes on a system without read-ahead of disk files and with all inter-pass files stored on the disk. The previous version of module *DiskSystem* was compiled (about 1700 source lines). Columns 2, 3, 4, and 5 lists relative improvements in percent to column 1.

Column 2: Reading ahead at most 4 sectors.
Column 3: Reading ahead at most 8 sectors.
Column 4: Memory resident inter-pass files.
Column 5: Memory resident inter-pass files and reading ahead at most 8 sectors.

	1	2	3	4	5
	s	%	%	%	%
Pass 1	36	8.5	17.8	30.6	30.6
Pass 2	24	31.0	35.4	62.5	62.5
Pass 3	26	28.0	32.0	28.8	28.8
Pass 4	15	23.0	23.0	46.7	46.7
Lister	40	20.0	27.5	0.0	27.5
Total	140	20.5	27.7	28.6	30.0

Columns 4 and 5 only differ when program *Lister* is executed. If the compiler uses memory resident files, the *Lister* is the only program in the compiler accessing more than one disk file at the same time! From the given figures, it can be derived that for the compiler the improved buffering technique for disk files yields a similar improvement in the compilation time as do memory resident inter-pass files (columns 3 and 4).

At the end of an edit session the edited file is constructed by copying text fragments from two or more files together. The time needed to terminate an edit session is reduced 30 % - 50 % by the modified system. For example when editing module *DiskSystem* the termination time is reduced from around 48 sec to 27 sec.

The *debug* program inspects a file containing a dump of the main memory. The dump file is randomly accessed by the debugger. The time needed by the debugger to set up its internal tables is reduced by around 50 % by the modified system.

No serious study has been made of the efficiency of Medos-2's disk I/O compared to other systems' disk I/O. A simple experiment suggests however that Medos-2's efficiency is comparable to that of three other systems [Rit78]: UNIX on PDP-11/70, DEC's IAS for PDP-11, and Honeywell's GCOS TSS for the H6070. The experiment consisted of timing a program that copied a file that, on the the PDP-11, contained 480 sectors with 512 Byte per sector. The file on the H6070 had the same number of Byte, but there were 1280 Byte per sector. The file copied on Medos-2 contained 512 sectors with 256 Byte per sector. With otherwise idle machines, the real times to copy the files were:

System	s	Byte	Byte/s	ms/sector	remarks
Medos-2	9	131072	29100	8.8	typical
Medos-2	6.28	131072	41700	6.14	best
UNIX	21	245760	23400	21.9	
IAS	19	245760	25900	19.8	
GCOS	9	245760	54600	23.4	

The effective transfer rates of Medos-2 compare well with the other three systems. If the transfer rate is measured in transferred sectors per second, Medos-2's disk I/O turns out to be significantly better. This is remarkable because the average seek time of the disk drive used by Medos-2 is atypically long (65 ms). The good result is mainly due to Medos-2's better allocation of sectors to files compared to the other systems. The best average transfer time being 6.14 ms/sector should be compared to the minimum possible time between two transfers being 4.17 ms with the current sector interleaving. From the given figures it can be derived that the average contribution of seeks, rotational latency, and directory inspections to the transfer time of a sector is only 2 ms to 5 ms during a file copy in Medos-2! The variation in the times needed to copy a file is caused by differences in the allocation of sectors to files.

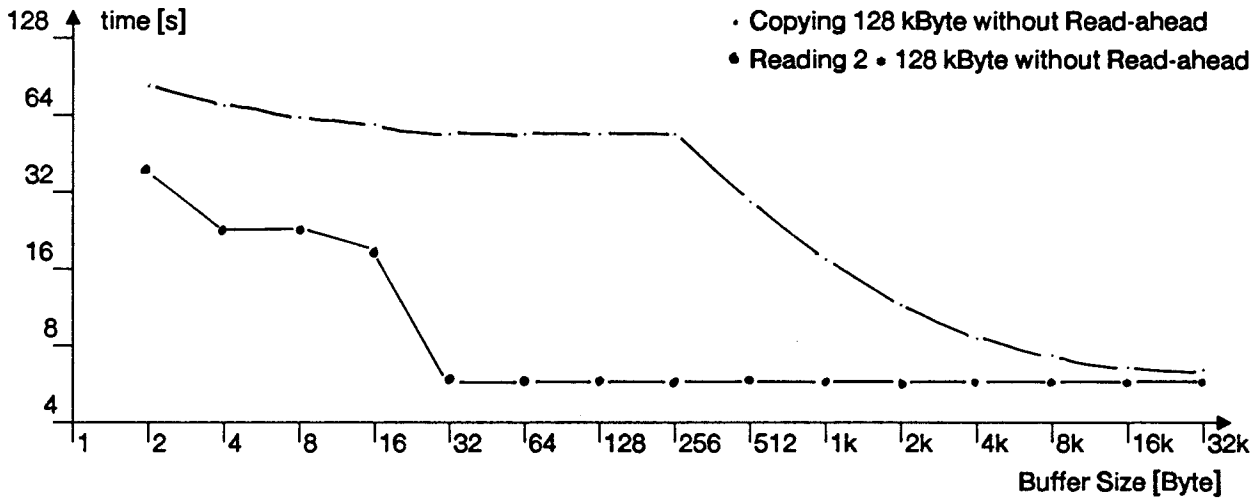


Figure 6.3 Copy- and Read-times

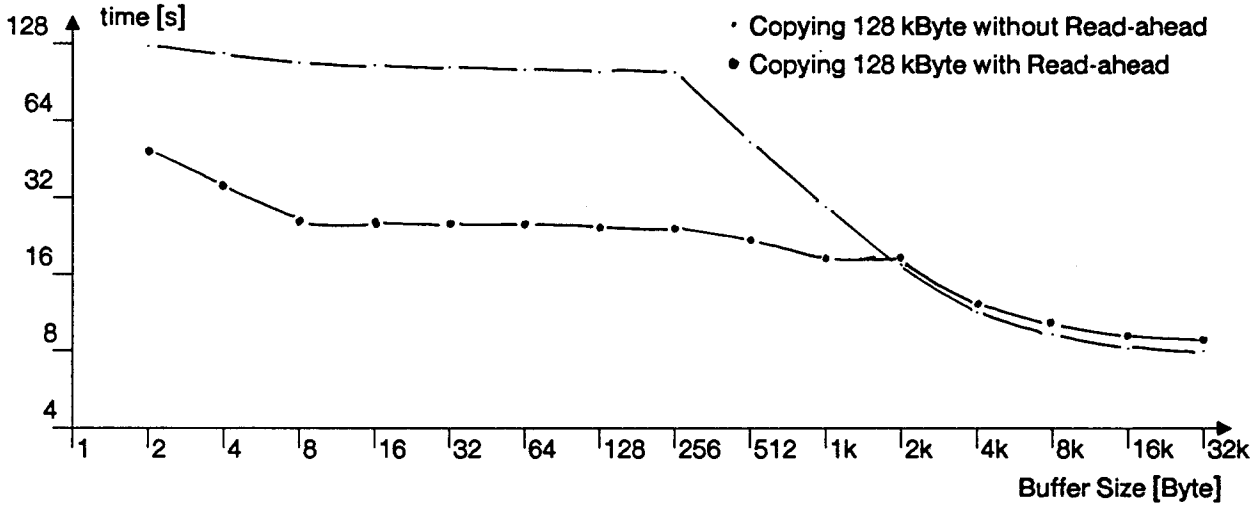


Figure 6.4 Copy-times

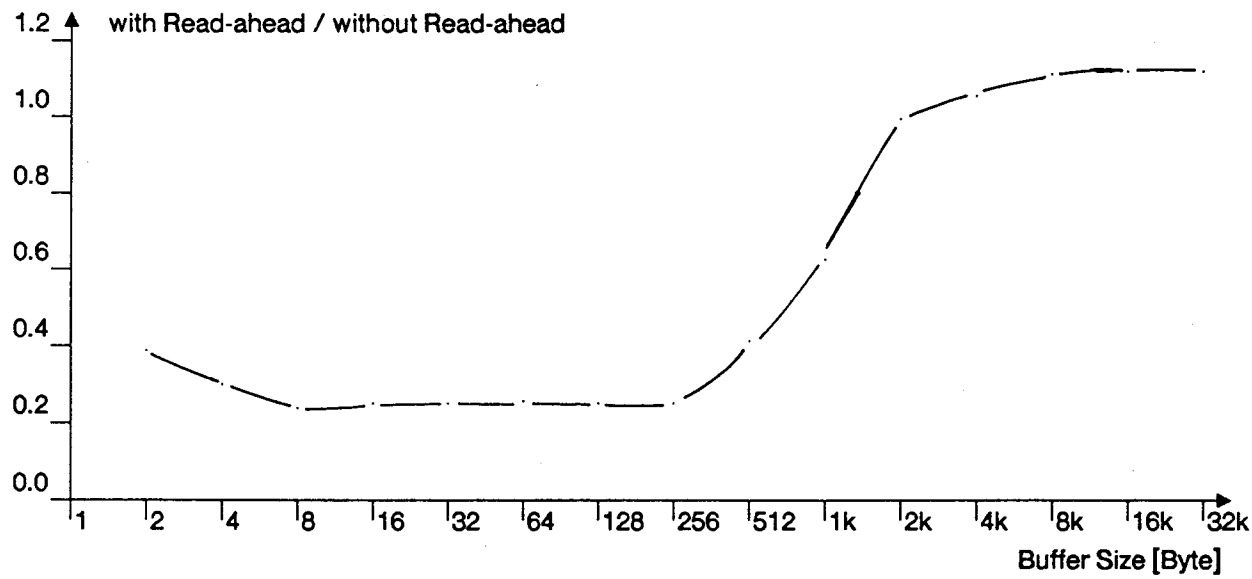


Figure 6.5 Quotient of Copy-times

7 Conclusions

In section 7.1 and 7.2 the more important advantages and disadvantages of Medos-2 are discussed. Evaluation of Modula-2 as a systems implementation language and of Lilith in the environment of personal computing are given in section 7.3 and 7.4. Section 7.5 contains some final remarks and perspectives for the future development of Medos-2.

7.1 System Advantages

Among the more attractive attributes of Medos-2 are the following:

Full support of Modula-2. All program interfaces are presented as Modula-2 definition modules. The Modula-2 compiler and Medos-2's linking-loader check the syntactically correct use of module interfaces at compile-time and load-time.

Simple structure. The resident part of Medos-2 consists of 15 modules. Most programmers need to be familiar with only a few of these modules, namely *Terminal* and *FileSystem*. These system-provided modules are used like other separate modules such that programmers need distinguish between system modules and library modules. System interfaces are documented by short but nevertheless quite complete descriptions.

Moderate size. A resident operating system of less than 15'000 word is resonably small, especially when considering that file buffers and a default-font are included in this memory space.

Efficiency. Both program execution times and average I/O transfer rates compare favourably with other mini-computer systems of today [Jac82], (section 6.4.5).

Reliability. Medos-2 has turned out to be quite reliable for its users. Only seldom have programming errors in the system caused troubles for its users. Its robustness against erroneous user-programs is also quite high, considering that no hardware supported protection mechanisms are used. The remarkably low frequency of system crashes are presumably due to the use of a high level programming language supporting modularization of programs and to its compiler's inclusion of runtime tests in the generated code (e.g. index-checks).

Openness. Medos-2 is open in many respects: Programmers may for example directly access devices, provide their own implementation of files, activate programs like procedures, provide their own storage-allocator. No uncircumventable protection mechanism hinders the user from getting as much as wanted out of Lilith.

7.2 System Disadvantages

Beside the many reasonably sound principles applied during the construction of Medos-2, several adhoc solutions to problems were less than ideal:

Memory management. The management of the heap by module *Program*, the hierarchically second highest module in the resident system, was not a good idea. As the system was designed, it was thought that a better memory utilization could be

achieved this way (see section 5.1). Whether a better memory utilization is obtainable by letting the module *Program* perform memory allocation is an open question without having other implementations for comparison. Now, Lilith's memory is twice as large as it was when Lilith was conceived, and memory space is no longer as critical as it was earlier. The consequence of allocating heap-space in module *Program* is, however, that nearly no resident module can (and actually none does) allocate memory space on demand. The task of managing the "reverse" memory stack like a heap was left over to a nonresident module (e.g. module *Storage*). The idea behind this decision was to make it possible for programmes to implement their own heap management strategy. Having a user-provided module to allocate heap-space excludes, however, in most cases the use of many library modules which may also allocate heap-space, this time by use of the library module *Storage*. Module *Storage* and the user-provided heap-allocator are in most casese unable to coexist.

Processes. The total exclusion of processes from Medos-2 is a too restrictive strategy. Currently, module *Monitor* includes a simple process scheduler in its implementation part. It would have been a simple matter to provide a process scheduler supporting only operating system internal processes. The main reason for not explicitly supporting processes in Medos-2 was, besides memory space considerations, that it was not understood how the system could recover from a crashed program executed by more than one process by *simple* means. Operating system provided processes get more important when Medos-2 is used for programming network servers.

Adaptability. Especially module *DiskSystem* has been tailored closely to the available disk drive in order both to save memory space and to get high transfer rates. As a consequence, it may turn out to be difficult to adapt this module to other types of disk drives storing similar amounts of information.

File directories. The system includes no general interface allowing the inspection (and manipulation) of filename directories. As long as the system was only used as a stand-alone machine, this deficiency was not manifest. From the moment when files stored on other machines could be accessed through the local area network [Hop83] the importance of such an interface became clear for most users of Lilith.

7.3 Evaluation of Modula-2 as a Systems Implementation Language

The value of using a high-level language like Modula-2 for the implementation of an operating system can almost not be overestimated. Programs written in assembly are difficult to understand and modify. In contrast, programs written in a high level language are much easier to understand, repair, adapt, and extend. The portability of a program will generally also be much better if it is written in a high-level language. All these beneficial effects of high-level languages apply to Modula-2.

The concept of separate compilation turned out to be vital for Medos-2. Separate compilation (together with the linking-loader of Medos-2) makes it possible to easily change the implementation of a system resident module and to add new modules to the system without having to recompile or relink all user-written programs. The

checking of the use of module interfaces at compile-time is of course also very valuable, because most trivial errors in user-programs are found in this way before a program is executed.

The portability of the system was unplanned. Most modules of Medos-2 are, however, portable enough that they may be compiled and executed on a PDP-11 without making any changes. For example module *FileSystem* and module *DiskSystem* were tested and debugged on a PDP-11 before the same modules were cross-compiled for the first execution on Lilith.

The value of the special features provided for low-level programming in Modula-2 depends of course on the actual application. In the implementation of Medos-2, all the provided low-level facilities have been used, although very rarely. For example, type converters are only used at about half a dozen locations in the resident system (corresponding to about 6000 source lines).

From an operating system designer's point of view, a much better support of abstract data types and instances of such types is desirable than is actually provided by the so-called *opaque types*. As things stand now, it is very difficult (and tricky) or even impossible to determine the exact lifetime of, for example, an opened file. Medos-2 defines the owner of an opened file to be an activated program. This might cause an opened file to exist either shorter or longer than actually desired.

7.4 Evaluation of Lilith's Architecture and Hardware

The M-code architecture of Lilith turned out to perform well: the code is dense and the execution of separately compiled Modula-2 programs is supported almost perfectly. The execution speed of programs written in a high-level language is comparable or better than on most current mini-computers.

Some few concepts of the machine cause, however, problems for the operating system designer:

The not uniformly addressable memory makes it difficult to make good use of the existing physical memory. It should, however, be remembered that the addition of 65536 word of memory was an afterthought, performed primarly to accomodate the bitmap and fonts.

Lilith has no (non trivial) mapping from virtual addresses to physical addresses. As a consequence the operating system has no efficient way to combine several free segments in memory to one larger segment. Several unrelated computations cannot be supported in a reliable way under such circumstances. This was not one of the initial goals of the system. More severe is the restriction that the operating system itself (or generally all activated programs apart from the currently running program) can generally not obtain memory space on demand, even if enough free memory is available, without making the system unreliable. The quite restrictive handling of the heap in Medos-2 is essentially due to this fact. (Only the currently running program may allocate space in the heap.)

Nearly every designer of a device interface felt free to choose concepts for controlling the interface. The result is that connected devices cannot even be

turned off in a standard way. As long as devices are controlled by resident drivers this fact may be tolerable. The desired openness of the system requires, however, that devices are controlled by nonresident, user-provided drivers. If such a program crashes, the device controlled by it is generally not turned off, and the operating system has no possibility to do it for the crashed program. Device controllers writing into memory by direct memory accesses are especially dangerous in this respect, because the memory space used by the next following program may be corrupted by the wildly running device controller.

Several M-code instructions (essentially memory move instructions) are implemented uninterruptably and have execution times as high as 50 ms. For example, the scrolling of the display may cause the execution of such an instruction. Such implementations of instructions generally reduce the computer's ability to respond to real-time events.

7.5 Perspectives

From the beginning, Lilith was intended to be a purely personal computer mainly constructed to support the development and testing of programs as well as the preparation of documents. The sucess of Medos-2 (if it has any) is mainly due to its simple, transparent interface for use. Nearly from the start, it was possible to maintain the system and all applications on the system itself. The heavy use of the system and the basic application programs like the compiler, the editor, the debugger, and the file utilities caused the designers to become aware of deficiencies and motivated them to correct them quickly.

A number of facilities provided in other systems are not present in Medos-2. Many of these things would be useful, or even vital to some applications. Good arguments exist for including new extensions in a system, especially if one can become convinced that a proposed extension is not merely a narrowly conceived, isolated "feature" that will not intergrate well with the rest of the system. For example, it is necessary to realize that the limited address space of Lilith imposes severe constraints on the size of the resident system.

Under the present view, it would be generally desirable if Medos-2 were expanded to perform better when several computers are connected by a local area network. Both network servers and ordinary work-stations should be adequately supported by Medos-2. Concurrent processes, at least, must be provided by the operating system for this purpose. A more general management of the heap is also desirable. The support of file directories on any medium and, for the standard disk drive, of files as large as the storage medium are toppics for improvements.

Seite Leer /
Blank leaf

Appendix 1 Descriptions of the Modules in Medos-2

Appendix 1.1 Module CardinalIO

Module *CardinalIO* helps in reading octal numbers from the keyboard and writing octal numbers on the display. The module is used for writing out error messages and for debugging purposes within Medos-2.

```
DEFINITION MODULE CardinalIO; (* Medos-2  Sv.E. Knudsen  25.09.80 *)

  EXPORT QUALIFIED ReadOct, WriteOct;

  PROCEDURE ReadOct(VAR c: CARDINAL);
  PROCEDURE WriteOct(c: CARDINAL);

END CardinalIO.
```

Explanations

ReadOct(c)

Procedure *ReadOct* reads in octally an cardinal number *c*. The terminating character (ESC, EOL and space) is not read in.

WriteOct(c)

Procedure *WriteOct* writes out octally the cardinal number *c*. The format of the written number is one space followed by six octal digits.

Appendix 1.2 Module DefaultFont

Module *DefaultFont* contains the so-called default font, the font used by module *DisplayDriver* for writing text on the screen.

Note: This module should only be used by module *DisplayDriver*.

```
DEFINITION MODULE DefaultFont;   (* Medos-2  W. Winiger  4.10.80 *)

  FROM SYSTEM IMPORT ADDRESS;

  EXPORT QUALIFIED defaultFont;

  VAR defaultFont: ADDRESS;

END DefaultFont.
```

Explanations

The variable *defaultFont* is a the address of the default font divided by four.

Appendix 1.3 Module DiskSystem

1 Introduction

Module *DiskSystem* stores files on 10 MByte cartridges for the Honeywell Bull D120/D140 disk drive. The definition module is shown in section 2. The provided routines are explained in section 3. The characteristics and restrictions of the current implementation are given in section 4. Nine files are preallocated. These files are listed in section 5. All error messages are explained in section 6.

2 Definition Module DiskSystem

```
DEFINITION MODULE DiskSystem;  (* Medos-2  Sv.E. Knudsen 25.09.80 *)

  FROM FileSystem IMPORT File, Response;

  EXPORT QUALIFIED
    InitVolume, OpenVolume, CloseVolume,
    FileDesc, FDKind, nofile, father, son,
    FileNumber, Position, Minute, Page,
    FreePages,
    ReadFileDesc, WriteFileDesc,
    Name, ExternalName, NameKind, free, fname,
    ReadName, WriteName;

  CONST
    nofile = 0; father = 1; son = 2;
    maxfiller = 27;
    modifyprot = 100000B;
    sons = 16;
    pagetablength = 96;

    free = 0; fname = 1;
    enlength = 24;

  TYPE
    FileNumber     = CARDINAL;
    FDKind         = CARDINAL;          (* nofile, father, son *)
    Position       = RECORD
                       block: CARDINAL; (* whole sectors *)
                       byte: CARDINAL;
                     END;
    Minute         = RECORD
                       day: CARDINAL;   (* coded as Time in Monitor *)
                       minute: CARDINAL;
                     END;
    Page           = CARDINAL;
```

```
  FileDesc      = RECORD
                    reserved: CARDINAL;
                    filno: FileNumber;
                    versno: CARDINAL;
                    CASE fdt: FDKind OF
                      nofile:
                        filler: ARRAY [0..maxfiller] OF CARDINAL |
                      father:
                        length: Position;
                        modification: CARDINAL;
                        referenced: CARDINAL; (* ref'ed if <> 0 *)
                        protection: CARDINAL; (* prot'ed if <> 0 *)
                        ctime: Minute; (* creation time *)
                        mtime: Minute; (* last modification *)
                        fres: ARRAY [0..3] OF CARDINAL;
                        sontab: ARRAY [1..sons-1] OF FileNumber; |
                      son:
                        fatherfile: FileNumber;
                        fathervers: CARDINAL;
                        sonno: CARDINAL;
                    END;
                    pagetab: ARRAY [0..pagetablength-1] OF Page;
                  END; (* FileDesc *)

  NameKind      = CARDINAL; (* free, fname *)
  ExternalName  = ARRAY [0..enlength-1] OF CHAR;
  Name          = RECORD
                    en: ExternalName;
                    CASE nk: NameKind OF
                      free:  |
                      fname:
                        fno: CARDINAL; (* file number *)
                        vno: CARDINAL; (* version number *)
                        fres: CARDINAL;(* reserved (set to zero) *)
                    END
                  END;

PROCEDURE InitVolume(VAR r: Response);
PROCEDURE OpenVolume(VAR r: Response);
PROCEDURE CloseVolume(VAR r: Response);

PROCEDURE FreePages(): CARDINAL;

PROCEDURE ReadFileDesc(no: FileNumber; VAR fd: FileDesc;
                       VAR r: Response);
PROCEDURE WriteFileDesc(no: FileNumber; VAR fd: FileDesc;
                        VAR r: Response);
```

```
  PROCEDURE ReadName(no: FileNumber; VAR n: Name; VAR r: Response);
  PROCEDURE WriteName(no: FileNumber; n: Name; VAR r: Response);

END DiskSystem.
```

3 Explanations

InitVolume(res)

Procedure *InitVolume* initializes the mounted disk, i.e. the nine preallocated system files are initialized on the disk (see section 5). *Warning*: The initialization of a disk destroys all files previously stored on it. A disk can only be initialized, if it is not opened for ordinary file accesses.

res = done	if medium is initialized,
res = notdone	if module *DiskSystem* is opened for file accesses,
res = ...	if some other error occured.

OpenVolume(res)

Procedure *OpenVolume* makes module *DiskSystem* ready for accesses to the mounted disk.

res = done	if medium is opened,
res = notdone	if module *DiskSystem* is opened for file accesses,
res = ...	if some other error occured.

CloseVolume(res)

Procedure *CloseVolume* closes module *DiskSystem* for accesses to files. *CloseVolume* can only be called from the program level from which it has been opened previously. Module *DiskSystem* cannot be closed, if any file beside the directory files are opened.

res = done	if medium is closed,
res = notdone	if module *DiskSystem* could not be closed.

FreePages(): CARDINAL

Function *FreePages* returns the number of still free pages on the mounted disk.

ReadFileDesc(filenr, filedesc, res)

Procedure *ReadFileDesc* reads the file descriptor of file *filenr* and returns it in variable *filedesc*. *res* indicates the success of the operation.

res = done	if the file descriptor has been read,
res = ...	if some error occured.

WriteFileDesc(filenr, filedesc, res)

Procedure *WriteFileDesc* writes *filedesc* as the file descriptor of file *filenr*. *res* indicates the success of the operation.

res = done	if the file descriptor has been written,
res = ...	if some error occured.

ReadName(filenr, name, res)

Procedure *ReadName* reads the name descriptor of file ilenr and returns it in variable *name*. *res* indicates the success of the operation.

res = done	if the name descriptor has been read,
res = ...	if some error occured.

WriteName(filenr, name, res);

Procedure *WriteName* writes the name descriptor *name*of file ilenr. *res* indicates the success of the operation.

res = done	if the name descriptor has been written,
res = ...	if some error occured.

4 Main Characteristics and Restrictions

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

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

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

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

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

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

65535).

5 System Files

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

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

6 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:

- `DiskSystem.` *procedure name* `:` *error indicating text*

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

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

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

- `DiskSystem.PutBuf: bad page: pageno =` *page number* `fno =` *file number*

Page indicates a disk address which is allocated to a "system file", but the file is not

a "system file", or the page indicates a disk address for normal files, but the file is a "system file".

- `DiskSystem.GetBuf: bad buffering while reading ahead`

The disk address of a certain allocated sector was not found.

- `DiskSystem.FileCommand: bad directory entry: fno =` *file number* `read`

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`
  `file name =` *local file name*
  `found fno =` *file number*`, expected fno =` *file number*

An inconsistency in the name directory was detected.

Warning

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

Appendix 1.4 Module DisplayDriver

Module *DisplayDriver* is the lowest level interface to the raster scan display. It actually provides an interface to four different topics, namely

1) the display hardware,
2) the default font,
3) the default bitmap and
4) the standard write procedure.

Note: This module should only be used within Medos-2 and library modules.

```
DEFINITION MODULE DisplayDriver; (* Medos-2 V4  N. Wirth 10.6.82 *)

  EXPORT QUALIFIED
    BMDescriptor, ScreenWidth, ScreenHeight, Show, BuildBMD,
    DFF, CharWidth, LineHeight,
    BMD, BMF, MapHeight, ChangeBitmap, Write;

  TYPE BMDescriptor = RECORD f,w,h,z: CARDINAL END ;

  PROCEDURE ScreenWidth(): CARDINAL;
  PROCEDURE ScreenHeight(): CARDINAL;
  PROCEDURE Show(VAR bmd: BMDescriptor; on: BOOLEAN);

  PROCEDURE BuildBMD(fp, width, height: CARDINAL;
                     VAR bmd: BMDescriptor);

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

  VAR BMD: BMDescriptor;
  PROCEDURE BMF(): CARDINAL;
  PROCEDURE MapHeight(): CARDINAL;)

  PROCEDURE ChangeBitmap(height: CARDINAL; VAR done: BOOLEAN);

  PROCEDURE Write(ch: CHAR);

END DisplayDriver.
```

Explanations

ScreenWidth(): CARDINAL

Function *ScreenWidth* returns width of the connected *display* in number of dots.

ScreenHeight(): CARDINAL

Function *ScreenHeight* returns the height of the connected *display* in number of (dot-) lines.

Show(bmd, on)

Procedure *Show* initializes the display interface such, that it would display the bitmap described by the bitmap descriptor *bmd*. If the boolean *on* is TRUE, the bitmap is actually shown on the display.

BuildBMD(fp, width, height, bmd)

Procedure *BuildBMD* initializes the bitmap descriptor *bmd* such, that it describes a bitmap in frame *fp* with *height* dot lines each with *height* lines.

DFF(): CARDINAL

Function *DFF* returns the frame pointer pointing to the default font, i.e. its address divided by four.

CharWidth(): CARDINAL

Function *CharWidth* returns the dot-width of the characters in the default font.

LineHeight(): CARDINAL

Function *LineHeight* returns the dot-height of a text line written with the default font.

BMD

Variable *BMD* is the bitmap descriptor of the default bitmap.

BMF(): CARDINAL

Function *BMF* returns the frame pointer pointing to the default bitmap, i.e. its address divided by four.

MapHeight(): CARDINAL

Function *Mapeight* returns the dot-height of the default bitmap.

ChangeBitmap(height, done)

Procedure *ChangeBitmap* changes the height of the default bitmap to *height* dots. If the change was performed, *done* is set TRUE.

Write(ch)

Procedure *Write* writes the character *ch* on the default bitmap with the default at the current write position. The current write position is changed by calls to procedure *Write* and *ChangeBitmap*. 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

Implementation Restrictions

1) *Show* supports only two different bitmaps, namely the default bitmap and one user-defined bitmap.

2) *BuildBMD*: *height* may not be 0 or 1; *width* should be divisible by 16.

3) *ChangeBitmap*: *height* must be even.

Error Handling

HALT is called if *BuildBMD* is called with *height* less than 2. No message is written out.

Appendix 1.5 Module D140Disk

Module *D140Disk* is a driver for the Honeywell-Bull D120/D140 disk drive. The driver is mainly intended to be used from module *DiskSystem*, the module implementing files on this type of media.

```
DEFINITION MODULE D140Disk;  (* Medos-2 V4  Sv.E. Knudsen 27.05.82 *)

  FROM SYSTEM IMPORT WORD;
  FROM FileSystem IMPORT Response;

  EXPORT QUALIFIED
    drives, tracks, sectors, sectorsize,
    DiskReset, DiskStatus, DiskRead, DiskWrite;

  CONST
    drives     =   2;
    tracks     = 784;
    sectors    =  48;
    sectorsize = 128;

  PROCEDURE DiskReset;
  PROCEDURE DiskStatus(): Response;
  PROCEDURE DiskRead(drive, diskadr: CARDINAL;
                     VAR buffer: ARRAY OF WORD; VAR res: Response);
  PROCEDURE DiskWrite(drive, diskadr: CARDINAL;
                     VAR buffer: ARRAY OF WORD; VAR res: Response);

END D140Disk.
```

Explanations

DiskReset

Procedure *DiskReset* resets the disk interface and the connected disk drive. The heads are hereby positioned over the tracks in cylinder zero.

DiskStatus()

Function *DiskStatus* returns the current status of the drive. The possible results (of type *Response*) are listed below.

DiskRead(drive, diskaddr, buffer, res)

Procedure *DiskRead* reads sector *diskaddr* from the disk in drive *drive* into the buffer *buffer*. The buffer must be 128 word long. The result parameter *res* is *done* if the operation was executed normally. Other possible values of *res* are listed below.

DiskWrite(drive, diskadr, buffer, res)

Procedure *DiskWrite* writes the buffer *buffer* to sector *diskaddr* on the disk in drive

drive. The buffer must be 128 word long. The result parameter *res* is *done* if the operation was executed normally. Other possible values of *res* are listed below.

Possible values for parameter *drive*:

The *D120* disk drive ignores the parameter *drive*.

The *D140* disk drive has a fixed disk and a slot for a removable disk. Drive zero is the drive for the removable disk pack, drive one is the drive with the fixed disk.

Possible values of function *DiskStatus* and parameter *res*:

done	the drive is ready
notdone	the drive is not ready
harderror	invalid sector accessed
hardprotected	the accessed pack is write-protected (DiskWrite only)
hardparityerror	a CRC-error was detected (DiskRead only)
timeout	the addressed sector was not found)
harderror	another error type detected by the drive
softerror	disk address >= 37632

Implementation Notes

The disk address of a sector [0..37631] is mapped into a physical disk address (cylinder, surface, sector) by an algorithm similar to the following one:

```
cylinder := diskadr DIV ( 2 * sectors );   (* 2 surfaces *)
surface  := diskadr DIV sectors MOD 2;
sector   := diskadr MOD sectors;
IF cylinder < 15 THEN
  sector := 3 * sector      (* boot and dump cylinders    *)
ELSE sector := 12 * sector  (* cylinders for normal files *)
END;
sector := sector MOD sectors + sector DIV sectors
```

Error Handling

Normally all detected errors are handled by assigning an error indicating *Response* to the parameter *res*. Whenever a more serious error is detected, one of the following error messages is written out.

```
- D140Disk:  soft timeout in wait
```

A disk operation was timed out by software. This error occurs mainly, if the disk is switched off while a disk operation is processed.

```
- D140Disk.DiskRead: response
-          diskadr = disk address, statusbits = statusbits
```

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

```
- D140Disk.DiskWrite: response
```

- `diskadr = ` *disk address*`, statusbits = ` *statusbits*

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

In the explanations of the messages, the following terms are used for inserted values:

response	string	Text describing the *response*
statusbits	octal number (177400 .. 177777B)	Status from disk interface
disk address	octal number (0 .. 111377B)	Disk address of a sector

The least significant 8 status bits have the following meanings when included:

001B	invalid sector encountered
002B	drive fault
004B	drive ready for seek operation
010B	write operation tried, but pack is write protected
020B	time out
040B	CRC error
100B	data transfer completed
200B	drive ready for a seek, read or write operation

Apendix 1.6 Module FileMessage

Module *FileMessage* helps writing out error messages in the file system.

```
DEFINITION MODULE FileMessage; (* Medos-2  Sv.E. Knudsen  4.10.80 *)

  FROM FileSystem IMPORT Response;

  EXPORT QUALIFIED WriteResponse;

  PROCEDURE WriteResponse(r: Response);

END FileMessage.
```

Explanations

WriteResponse(r)

Procedure *WriteResponse* writes a text corresponding to the response *res* on the terminal.

Error Handling

If the argument *res* to *WriteResponse* is no valid response, the ordinal value of the argument is written out octally.

Appendix 1.7 Module FileSystem

1 Introduction

A (Medos-2) file is a byte-sequence 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 section 2. The explanations needed for simple usage of sequential (text or binary) files are given in section 3. More demanding users of files should also consult section 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 section 5.

2 Definition Module FileSystem

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

  FROM SYSTEM IMPORT ADDRESS, WORD;

  EXPORT QUALIFIED
    File, Response,
    Create, Close, Lookup, Rename,
    ReadWord, WriteWord, ReadChar, WriteChar,
    Reset, Again, SetPos, GetPos, Length,
    Command, MediumType, FileCommand, DirectoryCommand,
    Flag, FlagSet,
    SetRead, SetWrite, SetModify, SetOpen, Doio,
    FileProc, DirectoryProc, CreateMedium, RemoveMedium;


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

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

    Command        = (create, open, close, lookup, rename,
                      setread, setwrite, setmodify, setopen,
                      doio,
                      setpos, getpos, length,
                      setprotect, getprotect,
```

```
                    setpermanent, getpermanent,
                    getinternal);

  File          = RECORD
                    bufa: ADDRESS;
                    ela: ADDRESS; elodd: BOOLEAN;
                    ina: ADDRESS; inodd: BOOLEAN;
                    topa: ADDRESS;
                    flags: FlagSet;
                    eof: BOOLEAN;
                    res: Response;
                    CASE com: Command OF
                      create, open, getinternal:
                        fileno, versionno: CARDINAL
                    | lookup: new: BOOLEAN
                    | setpos, getpos, length:
                        highpos, lowpos: CARDINAL
                    | setprotect, getprotect: wrprotect: BOOLEAN
                    | setpermanent, getpermanent: on: BOOLEAN
                    END;
                    mt: MediumType; mediumno: CARDINAL;
                    mh: MediumHint;
                    submedium: ADDRESS;
                  END;


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

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

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

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

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

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

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


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

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

END FileSystem.
```

3 Simple Use of Files

3.1 Opening, Closing, and Renaming of Files

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

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

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

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

DK.SYS.directory.OBJ

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

Create(f, mediumname)

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

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

Close(f)

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

Lookup(f, filename, new)

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

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

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

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

Rename(f, filename)

Procedure *Rename* changes the name of file f to filename. If filename is empty or contains only the medium name, f is changed to a temporary and nameless file. If filename contains a local name, the actual file will be permanent after a successful call of *Rename*. After the call

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

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 word (binary file), although this is *not* enforced by the file system. The first called routine causing any input or output on a file (i.e. ReadChar, WriteChar, ReadWord, WriteWord) determines whether the file is to be considered as a text or a binary file.

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

Reset(f)

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

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

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

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

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

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

If f.eof is TRUE:

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

Again(f)

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

Implementation Note

The current versions of the routines *ReadWord* and *WriteWord* do not support reading and writing of words at odd positions (for more information on *current position*, see 3.3).

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 on a file, and the position of the element is the number of the (first) Byte holding the element. By aid of the procedures *GetPos*, *Length* and *SetPos* it is possible to get the current position of a file, the position just behind the last element in the file, and to change explicitly the current position of a file.

SetPos(f, highpos, lowpos)

A call to procedure *SetPos* sets the current position of file f to *highpos * 2**16 + lowpos*. The new position must be less or equal the length of the file. If the last

operation before the call of *SetPos* was a write operation (i.e. if file f is in the writing state), the file is cut at its new current position, and the elements from current position to the end of the file are lost.

GetPos(f, highpos, lowpos)

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

Length(f, highpos, lowpos)

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

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
```

4 Advanced Use of Files

4.1 The Procedures FileCommand and DirectoryCommand

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

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

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

The commands *lookup* and *rename* must be executed by *DirectoryCommand*, other commands may be executed either by *FileCommand* or by *DirectoryCommand*. Unless the command is *lookup* or *rename*, a call to *DirectoryCommand* will be converted by the file system to a call to *FileCommand*. This facility is only useful for the commands *create* and *open* (see also 4.2).

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

create	create a new temporary (and nameless) file	(3.1)
open	open an existing file by *IFI*	(4.2)
close	close a file	(3.1)
lookup	look up (or create) a file by file name	(3.1)
rename	rename a file	(3.1)
setread	set a file into state *reading*	(4.5)
setwrite	set a file into state *writing*	(4.5)

setmodify	set a file into state *modifying*	(4.5)
setopen	set a file into state *opened*	(4.5)
doio	get next buffer	(4.5)
setpos	change the *current position* of the file	(3.3)
getpos	get the *current position* of the file	(3.3)
length	get the *length* of the file	(3.3)
setprotect	change the *protection* of the file	(4.4)
getprotect	get the current *protection* of the file	(4.4)
setpermanent	change the *permanency* of the file	(4.3)
getpermanent	get the *permanency* of the file	(4.3)
getinternal	get the *LFI* of the file	(4.2)

After the execution of a command, field *res* of the file reflects the success of the operation. Other fields of the file variable might, however, contain additional return values, depending on the executed command and the *state* of the file (see 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 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 4.5. The field *flags* enables a simple (and therefore efficient) test of the state of the file, whenever it is accessed. The "flag" *ag* is set by routine *Again* and cleared by read routines.

4.2 Internal File Identification and External File Name

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

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

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

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

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

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

The identification of a file by a user selected or computed name (a string) is however both commonly accepted and convenient. The syntax of a *file name* is given in 3.1. The routines *Create, Lookup, Rename* and *DirectoryCommand* all have a parameter specifying the file name.

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

The *local file name* part of the file name will be handled by the routine implementing *DirectoryCommand* for the medium given by the IMI (see also 5.).

Implementation Notes

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

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

When a *medium name* is "converted" to an *internal medium identification*, the letter(s) is (are) copied to the *MediumType* part (field *mt*), and the digits are considered as a decimal number whose value is assigned to the *medium number* part (field *mediumno*). If the medium name contains no digits, medium number 65'535 (= 177'777B) is assumed.

```
"DK"        =>    ( "DK", 65535)
```

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

4.3 Permanency of Files

As explained in 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 *)
```

4.4 Protection of Files

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

Protect File

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

Unprotect File

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

Get File Protection

```
f.com := getprotect;
FileCommand(f);
(* f.wrprotect = TRUE if and only if f is protected *)
```

4.5 Reading, Writing, and Modifying Files

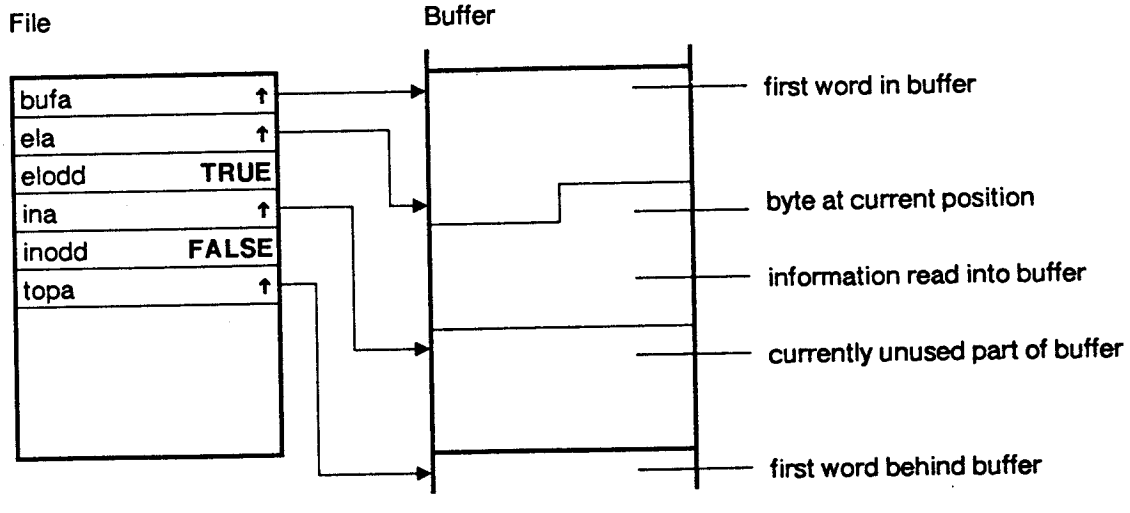
A file can be in one of four possible *I/O states* (or simply, *states*), namely in state *opened*, *reading*, *writing* or *modifying*. Just after a file has been connected (e.g. by a call to procedure *Create*), a file is in state opened, and its current position is zero. The state of a file can only 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 word of the buffer
topa	address of the first word behind the buffer
ina	address of the first not (completely) read in word 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
```

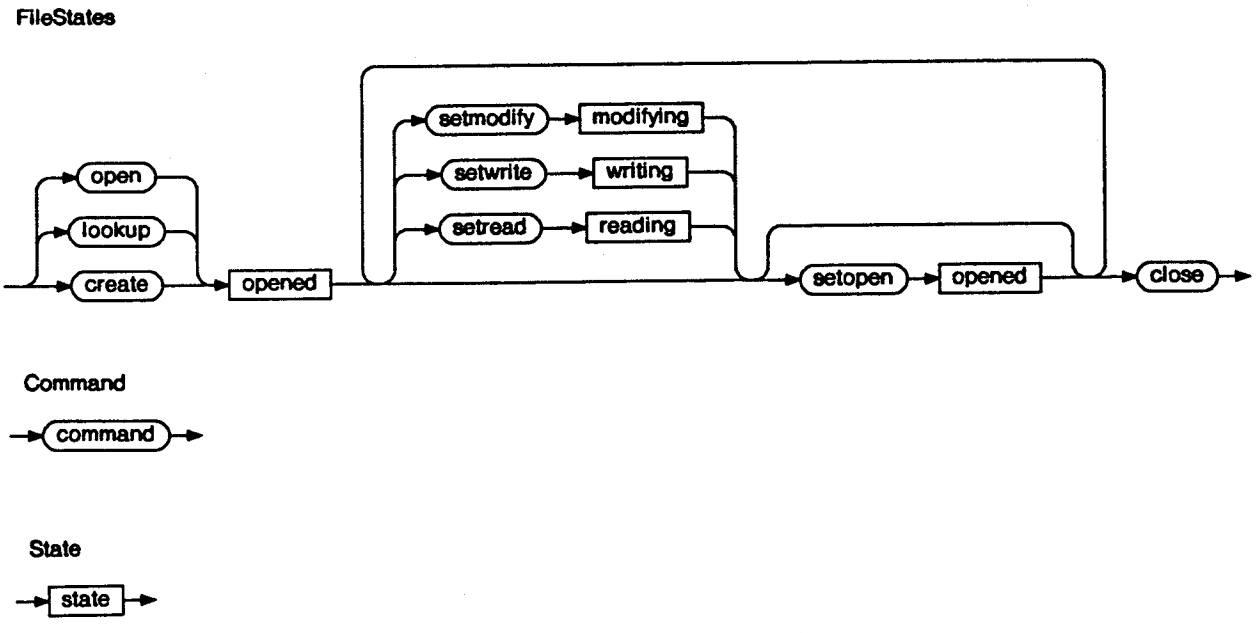
The fields *bufa, ina, inodd*, and *topa* are *read-only*, as they contain information which must never be changed by any user of a file.

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 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 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 *op*

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.

4.6 Examples

Procedure Reset(f)

```
PROCEDURE Reset(VAR f: File);
BEGIN
```

```
  SetOpen(f);
  SetPos(f, 0, 0);
END Reset;
```

Write File f

```
(* assume that file f is correctly positioned *)
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 correctly positioned *)
SetRead(f);
WHILE NOT f.eof DO
  WHILE ela < ina DO
    "use ela↑";
    INC(ela);
  END;
  Doio(f);
END;
SetOpen(f);
IF f.res <> done THEN
  (* Some read error occured *)
END;
```

Procedure WriteChar

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

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

BEGIN
  WITH f DO
    LOOP
      IF flags * FlagSet{wr,bytemode,er} <> FlagSet{wr,bytemode} THEN
        IF er IN flags THEN RETURN END;
        IF NOT (wr IN flags) THEN
          IF rd IN flags THEN
```

```
            (* Forbid to change directly from reading to writing! *)
            res := callerror; eof := TRUE;
            flags := flags + FlagSet{er, ef}
          ELSE SetWrite(f)
          END
        END;
        INCL(flags, bytemode)
      ELSIF ela >= topa THEN Doio(f)
      ELSIF elodd THEN
        SXB(ela, TRUE, ch);
        INC(ela, TSIZE(WORD)); elodd := FALSE;
        RETURN
      ELSE
        SXB(ela, FALSE, ch);
        elodd := TRUE;
        RETURN
      END
    END
  END
END WriteChar;
```

5 Implementation of Files

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

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

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

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

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

RemoveMedium(mediumtype, mediumnumber, done)

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

Implementation Note

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

Appendix 1.8 Module Frames

Module *Frames* serves the allocation and deallocation of main memory segments (*frames*) referenced by so-called *frame pointers*. The allocated frames reside at a fixed location and have life times like segments allocated from the normal heap. In contrast to heap segments which are referenced by (Modula-2) pointers, frames are accessed by a few M-code instructions only. Frames may, however, reside in the whole main memory and not just in its normally addressable part.

```
DEFINITION MODULE Frames;      (* Medos-2 V4  Sv.E. Knudsen  4.6.82 *)

  EXPORT QUALIFIED
    FramePointer, nil,
    Allocate, Deallocate, ChangeSize, Size;

  TYPE FramePointer = CARDINAL;

  CONST nil = 177777B;

  PROCEDURE Allocate(VAR fp: FramePointer; size: CARDINAL);
  PROCEDURE Deallocate(VAR fp: FramePointer);
  PROCEDURE ChangeSize(fp: FramePointer; newsize: CARDINAL; VAR done:
  PROCEDURE Size(fp: FramePointer): CARDINAL;

END Frames.
```

Explanations

Allocate(fp, size)

Procedure *Allocate* allocates a frame of the given size and assigns its address divided by 4 to *fp*. If the space is not available, *fp* gets the value *nil*.

Deallocate(fp)

Procedure *Deallocate* frees the given frame.

ChangeSize(fp, newsize, done)

Procedure *ChangeSize* changes the size of the given frame to the indicated size. *done* is set TRUE if the change was done.

Size(fp): CARDINAL

Function *Size* returns the allocated size of the given frame.

Error Handling

A detected error causes HALT to be called. No message is written out. If an error is detected during the clean-up after the execution of a program, Medos-2 might be forced to commit suicide.

Appendix 1.9 Module Monitor

Module *Monitor* enables

1) to set and get the internal clock,
2) to read characters from the keyboard,
3) to activate (call) and terminate execution of coroutines like procedures, and
4) to recover from crashes by so-called *termination procedures.*

Module *Monitor* hides for most programmers the existence of (driver-) processes within Medos-2.

Note: This module should only be used within Medos-2 and library modules.

```
DEFINITION MODULE Monitor;    (* Medos-2  Sv.E. Knudsen  1.6.81 *)

  FROM SYSTEM IMPORT PROCESS;

  EXPORT QUALIFIED
    Call, Terminate, Status,
    CurrentLevel, SharedLevel,
    TermProcedure,
    SetTime, GetTime, Time,
    Read;

  TYPE
    Status = (normal, instructionerr, priorityerr,
              spaceerr, rangeerr, addressoverflow,
              realoverflow, cardinaloverflow,
              integeroverflow, functionerr,
              halted, asserted, warned, stopped,
              callerr);

    Time = RECORD
             day: CARDINAL; (* ((year-1900)*20B + month)*40B + day *)
             minute: CARDINAL; (* hour*60 + minute *)
             millisecond: CARDINAL; (* second*1000 + msecond *)
           END;


  PROCEDURE Call(VAR p: PROCESS; shared: BOOLEAN; VAR st: Status);
  PROCEDURE Terminate(st: Status);

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

  PROCEDURE TermProcedure(term: PROC);

  PROCEDURE SetTime(t: Time);
```

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

  PROCEDURE Read(VAR ch: CHAR);

END Monitor.
```

Explanations

Call(process, shared, status)

Procedure *Call* executes the coroutine *process* like a parameterless procedure. *Current level* is incremented by one during the execution of the coroutine. If the parameter *shared* is true, *shared level* of the called coroutine remains unchanged otherwise it is set equal to the called coroutine's current level. After the execution of the coroutine, status inditates the termination cause.

Terminate(status)

The currently running program level can be terminated by a call of procedure *Terminate*. The parameter *status* indicates the termination cause, and will be handled over as status to the resumed program. (Please refer to the description of procedure *Call*.)

CurrentLevel(): CARDINAL

Function *CurrentLevel* returns the level number of the currently executed program.

SharedLevel(): CARDINAL

Function *SharedLevel* returns the level number of the lowest program sharing resources with the currently executed program.

TermProcedure(termproc)

A call to procedure *TermProcedure* causes that the procedure given as parameter 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.

SetTime(time), GetTime(time)

The internally maintained clock can be set or read by calls of the procedures *SetTime* and *GetTime*. As long, as the time has not been set, the field *day* has the value zero.

Read(ch)

Procedure *Read* gets the next character from the keyboard. If no character has been typed, 0C is returned. Character 36C is the last character of a line (eol character).

Appendix 1.10 Module Program

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 2. The program concept and explanations needed for the activation of a program are given in chapter 3. The *heap* and two routines handling the heap are explained in chapter 4. Possible error messages are listed in 5. The object file format may be inspected in 6.

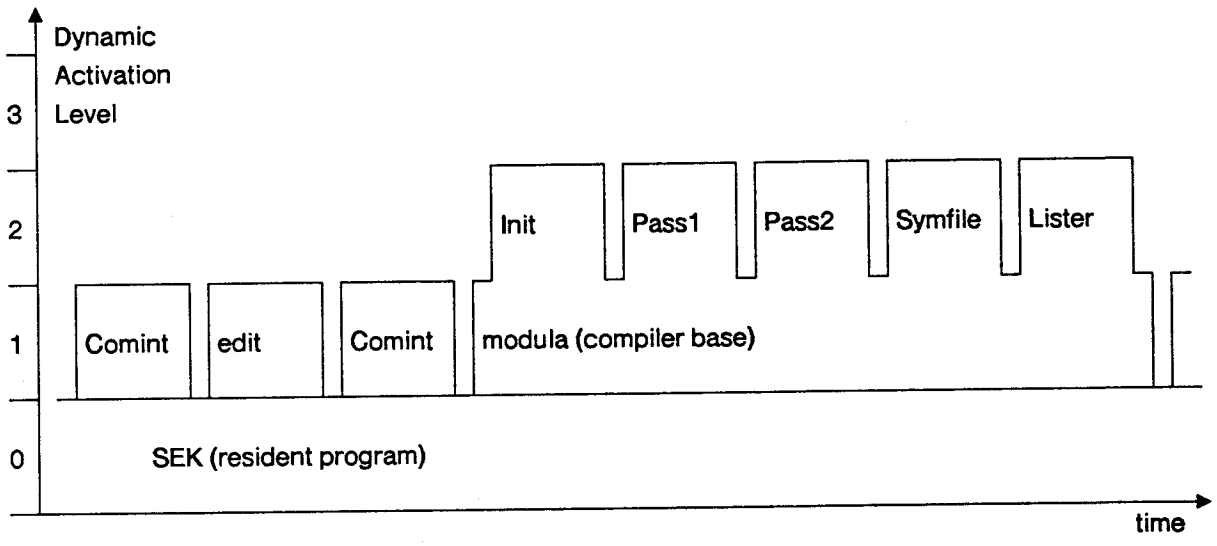
2 Definition Module Program

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

  FROM SYSTEM IMPORT ADDRESS;

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

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

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

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

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

END Program.
```

3 Execution of Programs

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

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



The figure illustrates, how programs may be activated. At 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* (a string).

The calling program is also resumed, if a program terminates by a crash (*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 any one time, i.e. there are no instances, no recursion (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, with 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 character
Identifier   = Letter { Letter | Digit } .
```

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

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

Missing object code to imported modules is searched for like programs. The (first 16 characters of the) module name is converted to a file name by inserting *DK.* at the beginning of the module name and appending the extension *.OBJ* to it. If the file is not found, a second search is made after the prefix *DK.* has been replaced by the prefix *DK.LIB.*. If the object code file is not yet found, the object code file for another missing module is searched. This is tried once for all imported and still not loaded modules.

Program name	`directory`
First searched file	`DK.directory.OBJ`
Second searched file	`DK.SYS.directory.OBJ`
Module name	`Storage`
First searched file	`DK.Storage.OBJ`
Second searched file	`DK.LIB.Storage.OBJ`

Call(programname, shared, status)

Procedure *Call* loads and starts the execution of program *programname*. If *shared* is

TRUE, the called program shares (sharable) resources with the calling program. The *status* indicates if a program was executed successfully.

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

Terminate(status)

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

CurrentLevel(): CARDINAL

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

SharedLevel(): CARDINAL

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

MainProcess(): BOOLEAN

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

Implementation Notes

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

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

Only up to 96 modules may be loaded at any time. The resident part of Medos-2 consists of 15 modules.

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

The maximum number of active programs is 16.

Related Program

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

Example: Command Interpreter

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

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

  CONST
    programnamelength = 16;

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

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

4 Heap

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

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

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

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

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

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

AllocateHeap(quantum): ADDRESS

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

DeallocateHeap(quantum): ADDRESS

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

Implementation Note

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

Related Module

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

Example: Procedures ALLOCATE and DEALLOCATE

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

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

5 Error Handling

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

- Program.Call: *error indicating text*

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

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

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

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

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

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

The listed modules were not found.

Appendix 1.11 Module SEK

Module *SEK (Sequential 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;    (* Medos-2 V4  Sv.E. Knudsen  27.05.82 *)

  FROM Program IMPORT Status;

  EXPORT QUALIFIED
    CallComint,
    PreviousStatus,
    NextProgram, SetParameter, GetParameter,
    Login, LeaveLogin,
    TestDK;

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

  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 actvated by module SEK may by a call to procedure *NextProgram* define,

which program should be executed after its own termination. If a program make 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 initialization 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 Honeywell-Bull D120/D140 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*.

Error Handling

Whenever module *SEK* initializes the system, it writes the version number of the system followed by a point or a slash. If the slash is written, this indicates that the initialization wasn't successful. Some informative messages might be written out just before the slash.

If the disk drive has been in an 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.

Appendix 1.12 Module System

Module *System* is the runtime system needed to execute Modula-2 programs on Lilith. It provides an implementation of procedure NEWPROCESS in the predefined module SYSTEM. Several variables defined at fixed locations by the firmware are also declared by the module. The communication from the linker generating the resident part of Medos-2 to module *System* and to the resident loader, as well as from the resident operating system to a debugger is also handled by variables declared in this module.

Note: Module *System* should *never* be imported.

```
DEFINITION MODULE System;    (* Modula-2 Compiler  Ch. Jacobi,
                                Medos-2 V4  Sv.E. Knudsen  26.1.82 *)

  FROM SYSTEM IMPORT PROCESS, ADDRESS;

  EXPORT QUALIFIED
      EndProcess, NewProcess,
      dataFrameLength,
      ProcedureMark, ProcessDescriptor, ProcessPointer, Vector,
      deviceMask, pRegister, savePRegister,
      interruptVectors, dataFrameTable,
      userProgram, codeKey, loadedModules,
      prevLoadedModules, prevProcess;

  CONST dataFrameLength = 128;

  TYPE
    ProcedureMark =
      RECORD
        g:   ADDRESS;
        l:   ADDRESS;
        pc:  CARDINAL;
        msk: BITSET
      END;

    ProcessDescriptor =
      RECORD
        mark: ProcedureMark;
        s: ADDRESS;
        h: ADDRESS;
        errCode:  CARDINAL;
        trapMask: BITSET
      END;

    ProcessPointer = POINTER TO ProcessDescriptor;
```

```
    Vector =
      RECORD
        CASE CARDINAL OF
          0: driver, interrupted: PROCESS |
          1: driverPtr, interruptedPtr: ProcessPointer
        END
      END;

  PROCEDURE EndProcess;
  PROCEDURE NewProcess(P: PROC; a: ADDRESS; s: CARDINAL; VAR P1: PROC

 (* firmware locations *)

  VAR deviceMask:        BITSET; (* process scheduler to firmware *)
      pRegister:         ProcessPointer; (*abs. linker to firmware*)
      savePRegister:     ProcessPointer; (* firmware to debugger *)
      interruptVectors:  ARRAY [7..15] OF Vector;
      dataFrameTable:    ARRAY [0..dataFrameLength-1] OF ADDRESS;

  (* reserved for operative system, post mortem dump *)

  VAR userProgram:       PROC;            (* abs. linker to System *)
      codeKey:           CARDINAL;        (* abs. linker to loader *)
      loadedModules:     CARDINAL;        (* abs. linker to loader *)
      prevLoadedModules: CARDINAL;        (* loader to debugger *)
      prevProcess:       ProcessPointer;  (* loader to debugger *)
      residentModules:   CARDINAL;        (* modules in Medos-2 *)

END System.
```

Explanations

Module *System* is given module number zero and linked to be loaded at memory location zero by the so-called absolute linker (i.e. the linker generating the *boot file*). The compiler compiles a call of SYSTEM.NEWPROCESS into the code sequence

call external module 0 procedure 2 (CX 0 2).

The procedure number 2 is given by the compiler because procedure *NewProcess* is declared as the second procedure in the definition module.

The variables declared in module *System* should simply be considered as space-holders. One exception is variable *userProgram*. This procedure variable is initialized to be the initialization part of the resident system. After the booting of a program, the initialisation code of module *System* is executed. By a call of procedure *UserProgram*, the main module of resident system is activated (normally module *SEK*).

Appendix 1.13 Module 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;   (* Medos-2  Sv.E. Knudsen  1.6.81 *)

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

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

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

END Terminal.
```

Explanations

Read(ch)

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

BusyRead(ch)

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

ReadAgain

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

Write(ch)

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

eol	36C	Sets the writing position at the beginning of the next line
CR	15C	Sets the writing position at the beginning of the current line
LF	12C	Sets the writing position to the same column in the next line
FF	14C	Clears the screen and sets the writing position into its upper left corner

BS 10C Sets the writing position one character backward
DEL177C Sets the writing position one character backward and erases the chara

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)*.

Appendix 1.14 Module 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; (* Medos-2 V4 Sv.E.Knudsen 7.6.82 *)

  EXPORT QUALIFIED
    ReadProcedure, AssignRead, Read,
    WriteProcedure, AssignWrite, Write;

  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 parameter *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 Restriction

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

Appendix 1.15 Module UserIdentification

The module *UserIdentification* serves the identification of (not necessarly) 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 UserIdentification; (* Medos-2 V4  SEK 7.10.1982 *)

  EXPORT QUALIFIED
    User, GetUser, SetUser, ResetUser;

  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 UserIdentification.
```

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)

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. (See next page!)

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 =       0      no user
    1  <= 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
  pw1 := 0; pw2 := 0;
  c := 0;
  WHILE (c <= HIGH(password)) AND (password[c] <> 0C) DO
    h := pw2; pw2 := pw1;
    pw1 := (h MOD 509 + 1) * 127 + ORD(password[c]);
    INC(c)
  END
END ConvertPassword
```

Restriction

Module *UserIdentification* supports "only" 8 program levels. If procedure *SetUser* or procedure *ResetUser* is called from programs with level number >= 8, HALT is called. No message will be displayed.

Appendix 2 Format of Object Code Files

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

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

The load file is a sequence of word, with *FrameType* and *Number* each represented in one word.

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

```
ObjectFile    = Module { Module } .
Module        = [ VersionFrame ] HeaderFrame [ ImportFrame ]
                  { ModuleCode | DataFrame }.
VersionFrame  = VERSION FrameSize VersionNumber.
FrameSize     = Number.
VersionNumber = Number.
HeaderFrame   = MODULE FrameSize ModuleName DataSize [ CodeSize Flags ]
ModuleName    = ModuleIdent ModuleKey.
ModuleIdent   = Letter { Letter | Digit } { "0C" }.  /exactly 16 characters/
ModuleKey     = Number Number Number.
DataSize      = Number.  /in word/
CodeSize      = Number.  /in word/
Flags         = Number.
ImportFrame   = IMPORT FrameSize {ModuleName}.
ModuleCode    = CodeFrame [ FixupFrame ].
CodeFrame     = CODETEXT FrameSize WordOffset { CodeWord }.
WordOffset    = Number.   /in word from the beginning of the module/
CodeWord      = Number.
FixupFrame    = FIXUP FrameSize {ByteOffset}.
ByteOffset    = Number.   /in Byte from the beginning of the module/
DataFrame     = DATATEXT FrameSize WordOffset { DataWord }.
DataWord      = Number.
VERSION       = "200B".
MODULE        = "201B".
IMPORT        = "202B".
CODETEXT      = "203B".
DATATEXT      = "204B".
FIXUP         = "205B"
```

Currently the *VersionNumber* is equal to 3.
Currently the *Flags* are set to 0.

The *ByteOffsets* in *FixupFrame* point to bytes in the code containing *local* module numbers. The local module numbers must be replaced by the *actual* numbers of the

corresponding modules. Local module number 0 stands for the module itself, local module number i (i > 0) stands for the i'th module in the *ImportFrame*.

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

References

[Abe79] R. B. Abel, A Diske Cashe, EURO IFIP 79, North-Holland Publishing Company, 1979, pp. 575-580.

[Amd70] G. M. Amdahl, Storage and I/O Parameters and Systems Potential, Proceedings of the IEEE Computer Group Conference, Washington D. C, June 1970.

[BBF82] G. Beretta, H. Burkhardt, P. Fink, J. Nievergelt, J. Stelovsky, H. Sugaya, A. Ventura, J.Weydert, XS-1: An Integrated Interactive System and its Kernel, Proc. 6th International Conference on Software Engineering, Tokyo, September 1982, pp. 340-349.

[BH73] P. Brinch Hansen, Operating System Principles, Prentice-Hall, Englewood Cliffs, N.J., 1973, cf. p. 1.

[BH75] P. Brinch Hansen, The Programming Language Concurrent Pascal, IEEE Transaction on Software Engineering 1, 2, June 1975, pp. 199-207.

[BH77] P. Brinch Hansen, The Architecture of Concurrent Programs, Printice-Hall, Inc, Englewood Cliffs, N. J., 1977.

[BH81] P. Brinch Hansen, Edison--A Multiprocessor Language, Software - Practice and Experience, April 1981, pp 325-362.

[Den82] P. J. Denning, Are Operating Systems Obsolete? CACM, Vol 25 Nr 4, April 1982, pp. 225-227.

[DMN68] O.-J. Dahl, B. Myhrhaung, K. Nygaard, The Simula 67 Common Base Language, Norwegian Computing Center, Oslo, 1968.

[EH82] W. Effelsberg, T. Haerder, Principles of Database Buffer Management, Interner Bericht, Fachbereich Informatik, Universität Kaiserslautern, April 1982.

[Gei83] L. Geissmann, Separate Compilation in Modula-2 and the Structure of the Modula-2 Compiler on the Personal Computer Lilith, Diss. 7286, ETH Zürich, 1983.

[Han82] L. Geissmann, J. Hoppe, C. Jacobi, S. E. Knudsen, W. Winiger, N. Wirth, Lilith Handbook, Institut für Informatik, ETH, Zürich. October, 1982.

[Hoa74] C. A. R. Hoare. Monitors: an Operating System Structuring Concept, CACM, Vol 17 Nr 10, October 1974, pp. 549-557.

[Hoe82] J. Hoeg. DAMOS System Specification, CSS/006/PSP/0044, Christian Rovsing A/S, Ballerup, Denmark, 1982.

[Hop83] J. Hoppe, Magnet: A Local Network for Lilith Computers, Institut für Informatik, ETH, Zürich, A planned report.

[Ich80] J. D. Ichbia, et al, Reference Manual for the Ada Programming Language, U. S. Department of Defence, July 1980.

[Jac82] Ch. Jacobi, Code Generation and the Lilith Architecture, Diss. 7195, ETH Zuerich, 1982.

[Jon78] A. K. Jones, The Object Model: A Conceptual Tool for Structuring Software, Operating Systems, An Advanced Course, Lecture Notes in Computer Science 60, Springer-Verlag, Berlin Heidelberg 1978, pp. 7-16

[Jon79] A. K. Jones, et al, StarOS, a Multiprocessor Operating System for the Support of Task Forces, Proc. of the Seventh Symposium on Operating System Principles, ACM SIGOPS, December 1979, pp. 117-127.

[JW75] K. Jensen, N. Wirth, Pascal User Manual and Report, Springer-Verlag, New York, 1977.

[Kah81] K. C. Kahn, et al, iMAX: A Multiprocessor Operating System for an Object-Based Computer, Proc. of the Eight Symposium on Operating System Principles, ACM SIGOPS, Vol 15 Nr 5, December 1981, pp. 127-

[Kat78] J. A. Katzmann, A Fault-Tolerant Computing System, Eleventh Hawaii International Conference on System Science, January 1978, pp. 85-102.

[KMP83] J. Koch, M. Mall, P. Puffaken, M. Reimer, VJ. W. Schmidt, C. A. Zehnder, Modula/R Report Lilith Version, Institut für Informatik, ETH, Zürich, Feburary 1983.

[Lam74] B. W. Lampson, An Open Operating System for a Single-User Machine, Lecture Notes in Computer Science 16, Springer-Verlag, Berlin Heidelberg, 1974, pp. 208-217.

[Lam81] B. W. Lampson, Atomic Transactions, In G. Goos and J. Hartmanis (editors), Distributed Systems - Architecture and Implementation: An Advanced Course, Springer-Verlag, Berlin Heidelberg, 1981.

[LSAS77] B. Liskov, A. Snyder, R. Atkinson, C. Schaffert, Abstraction Mechanisms in CLU, CACM, Vol 20 Nr 8, August 1977, pp. 564-576.

[MMS78] J. G. Mitchell, W. Maybury, and R. Sweet, Mesa Language Manual, Report CSL-78-1, Xerox PARC, Palo Alto, California, 1978.

[Nae79] H. H. Naegeli, Programmieren mit PORTAL,
Landis und Gyr, Zug, Switzerland, Juli 1979.

[Par72] D. Parnas, A Technique for Software Module Specifications with
Examples, CACM, Vol 15 Nr 5, May 1972, pp. 330-336.

[Pow77] M. L. Powell, The DEMOS File System,
Los Alamos Scientific Laboratory, Los Alamos, New Mexico, 1977.

[RRU82] J. Rebsamen, M. Reimer, P. Ursprung, C. A. Zehnder,
LIDAS - A Database System for the Personal Computer Lilith,
Report 50, Institut für Informatik, ETH, Zürich, June 1982.

[Red80] D. D. Redell, et al, Pilot: An operating system for a personal computer,
CACM, Vol 23 Nr 4, Febuary 1980, pp. 81-92.

[Rit78] D. M. Ritchie, A Retrospective,
The Bell System Technical Journal, Vol 57 Nr 6, Part 2,
Whippany, N.J., 1978, pp. 1947-1969.

[RT78] D. M. Ritchie, K. Thompson, The UNIX Time-Sharing System,
The Bell System Technical Journal, Vol 57 Nr 6, Part 2,
Whippany, N.J., 1978, pp. 1905-1927.

[Ruc82] H. Ruckstuhl, Erweiterung des Dateisystems der Lilith,
Diplomarbeit, Institut für Informatik, ETH, 1982.

[SMB79] D. Swinehart, G. McDaniel, D. Boggs, WFS: A Simple Shared File
System for a Distributed Environment, Proceedings of the 7th
Symposium on Operating Systems Principles, ACM, Dec. 1979.

[Smi78] A. J. Smith, Sequentiality and Prefetching in Database Systems,
ACM Transactions on Database Systems, Vol 3 Nr 3, September 1978,
pp. 223-247.

[SMI80] H. E. Sturgis, J. G. Mitchell, and J. Israel,
Issues in the Design and Use of a Distributed File System,
Op. Sys. Rev., Vol 14 Nr 3, July 1980, pp. 55-59.

[Sto81] M. Stonebraker, Operating System Support for Database Management,
CACM, Vol 24 Nr 7, July 1981, pp. 412-418.

[Sug82] H. Sugaya, Tree File: A Data Organisation for Interactive Programs,
Diss. 6944, ETH Zürich, 1982.

[TB74] D. C. Tsichritzis, P. A. Bernstein, Operating Systems,
Academic Press, New York, 1974, cf. p. 8.

[TML79] C. T. Tacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, D. B. Boggs, Alto: A Personal Computer, Xerox, Palo Alto Research Center, Palo Alto, California, 1979.

[Wir71] N. Wirth, The programming language PASCAL, Acta Informatica 1, 1971, pp. 35-63.

[Wir77] N. Wirth, Modula: a language for modular multiprogramming, Software - Practice and Experience, January 1977, pp. 3-35.

[Wir81] N. Wirth, The Personal Computer Lilith, A.I. Wassermann Ed., Software Development Environments, IEEE Computer Society Press, 1981.

[Wir82] N. Wirth, Programming in Modula-2, Springer-Verlag, Berlin Heidelberg, 1982, pp. 139-170.

[WN79] M. V. Wilkes, R. M. Needham, The Cambridge CAP Computer and its Operating System, North Holland, New York, 1979.

Curriculum Vitae

I was born on the 19th of October 1947 in Fredriksberg, Denmark. From 1954 to 1959 I attended the primary school in Vallensbaek, and from 1959 to 1962 the "realskole" in Roskilde. From 1962 to 1963 I attended school in Buchs/SG, Switzerland. In 1963 I entered the "gymnasium" in Vaduz, Liechtenstein where I obtained the "Matura" in 1970.

From spring to autumn 1970 I worked with ultra-high vacuum, mass-spectrometry, and sputtering at BALZERS AG in Balzers, Liechtenstein.

In the fall of 1970 I began my studies in physics at the Swiss Federal Institute of Technology (ETH) in Zürich and finished in the fall 1975 with the diploma "Dipl. Phys. ETH". As a diploma-thesis I measured the direction and energy dependency of lowenergy, photo-emitted electrons from solid and liquid mercury.

Since January 1976 I have worked as an assisatent at the "Institute für Informatik" of ETH Zürich in the research group of Prof. N. Wirth on the CDC-Pascal and the Lilith projects.