OBERON for PC on an MS-DOS base

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Computersysteme

Andreas R. Disteli

OBERON for PC on an MS-DOS Base

November 1993

203

ETH Zürich
Departement Informatik
Institut für Computersysteme
Prof. Dr. J. Gutknecht

Author's address:

A.R. Disteli, Institut für Computersysteme, ETH Zentrum, CH-8092 Zurich, Switzerland.
e-mail: disteli@inf.ethz.ch

Table of contents

1. Introduction

Oberon is a complete operating system and language developed at the Institute for Computer Systems. The whole project was targeted towards one specific type of machine, namely for the personal workstation Ceres. The Oberon system and the language found great attention outside ETH, but was not available on commercial machines. This was a big handicap for any kind of propagation. Some years ago several members of the institute started porting Oberon to commercial hardware platforms like DECstation, SPARC, RS/6000 and MacIntosh–II. This was a good chance to spread both the system and language Oberon over a large number of new users. But most of the people merely have access to simple MS–DOS based personal computers. Until now, Oberon was therefore not available on the most frequently used computers all over the world. With a port for DOS–machines a huge range of new potential users can be opened. For teaching purposes it would be interesting to use Oberon as well, because it proved to be very adequate as a language and system for students. Another point to consider is that PC machines became more and more inexpensive in the last years, which makes it possible for nearly everyone to own a computer for working at home. For all these reasons, we decided in summer 1991 to start the DOS–Oberon project, targeted at the Oberon system and language in combination with the most popular computer in the world.

The port had been structured into several parts. First of all, the compiler had to be ported, though compiler backends existed for all of the above mentioned machines, none of them generated code for the Intel 80x86 family. So, the complete code generation part of the compiler had to be rewritten.

Language

It was a personal decision to code the whole system in the language Oberon itself, except of some special modules which need to be written in assembler for performance reasons. Therefore, an assembler generating Intel code and running under Oberon had to be implemented. Only one module was written under native DOS: The meta loader (loading the actual module loader) was implemented in Microsoft assembler. The whole system was cross developed on a Ceres workstation running native Oberon. In particular, all the modules were crosscompiled for the bootstrap.

Storage

In a second step, a runtime environment had to be designed, including storage layout and memory management, as well as a

module loading mechanism. Many problems came from the fact that the processor needs to operate in different and incompatible modes (see Chapter 3, Memory Model).

Display

Another severe problem was the display system with its different standards and resolutions. The normal VGA resolution is standard on practically all machines. However, it soon turned out to be too small in size. In addition, programming the VGA card is unwieldy because of the many registers which must be set for each drawing operation. This slows down the performance of the display but it works on all machines equipped with a VGA card. On the other hand, a Super-VGA card offers a bigger resolution, but should be programmed separately when a fast display is required, because each card has its own extensions (See Chapter 7, The Display).

Restrictions

We intentionally restricted our implementation to 80386 or higher processors, because Oberon is a full 32-bit application and older processors like 80286 provide only support for 16-bit applications. Therefore, we decided not to support processors below the 80386. Also, our implementation does not allow specific drivers like expanded memory managers to be installed. The reason for this can be found in the need to monopolize protected mode unless special software (so-called extenders) are used (See Chapter 2: The Intel Environment). It was our goal to use as few as possible of externally developed software packages like drivers, special managers etc. For compatibility reasons we used a standard mouse driver (Logitech) and the HIMEM.SYS driver for the extended memory (above 1 Megabyte).

2. The Intel Environment

8088 processor

In 1981, IBM announced the first PC with an Intel 8088 processor. This is nothing special when seen by today's eyes but at that time it was a big jump from a 8–bit to a 16–bit CPU. The new processor was able to access up to 1 MB physical address space, however only 640 KB were accessible to normal programs, while the rest was reserved for video buffers and BIOS (Fig. 1). Also, most of the new drivers, controllers, etc. were allocated in this reserved area.

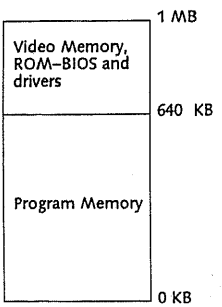

Fig 1: Memory map of a PC with 8088

80286 processor

In 1982, Intel brought the new 80286 processor, a successor of the 8086 and 8088 (Fig. 2). It supported a bigger physical address range of 16 MB and, most importantly, was prepared for multitasking. As a consequence, a method had to be invented to save one program's code and data area from access by other programs. This method was called *Protected Mode (PM)* in contrast to the old so–called *Real Mode (RM)*.

Real mode

In RM, every address consists of a segment part and an offset part. A segment can be seen as a piece of memory with a base address and a limit. In the case of a RM segment, the base address is aligned to a 16 byte boundary and the limit is 64 KB. Due to this alignement, the base address can be divided by 16 and is still unique. This technique saves 4 bits and the base address, although it addresses memory up to 1 MB, can be stored in a 16 bit location. A physical address is therefore computed by

address := segment * 16 + offset.

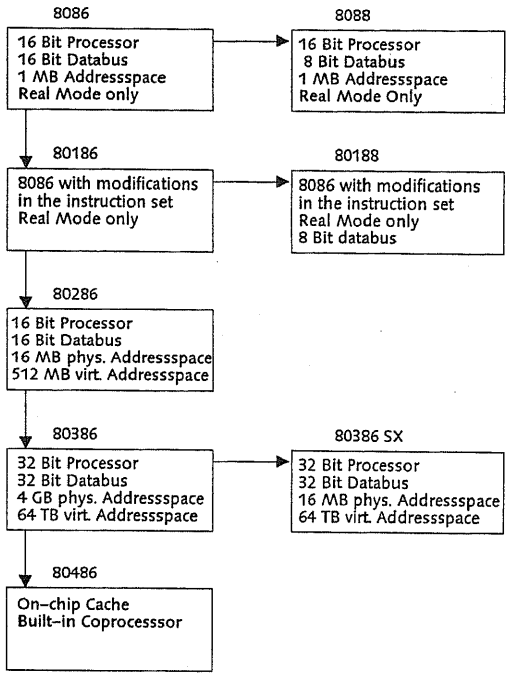

Fig. 2: Overview of the Intel processor family

Protected mode

With segment and offset, both 16 bit in size, addresses up to 1 MB could be generated. In the 80286 PM, the segment part of the address is no longer a simple address, but a pointer into a descriptor table containing a series of 24-bit addresses. Adding the 16-bit offset, the programmer can thus access up to 16 MB of memory. Of course, one and the same segment:offset pair can represent different physical addresses, depending on the contents of the descriptor table. It is therefore often called *virtual address* (Fig. 3).
At address generation time, the segment part of each address must be contained in one of the segment registers called SS, CS, DS or ES (see table 1 for explanations). The offset is either located in memory or in one of the processor registers BX, BP, SI or DI. The other registers, AX, CX, DX and SP can not be used for address calculation. These registers are dedicated and, as such, are the cause of a lot of problems. General use of all registers would be desirable.

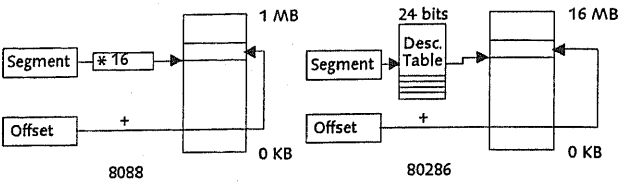

Fig. 3: Address generation

Segment registers:			
CS =	Code segment	ES =	Extra segment
DS =	Data segment	SS =	Stack segment
Processor registers:			
AX =	Accumulator	SI =	Source index
BX =	Base register	DI =	Destination index
CX =	Count register	BP =	Framepointer register
DX =	Data register	SP =	Stack pointer

Table 1: The registers of the 80286

Descriptor tables

The 80286 PM stands for coexisting tasks running under a safe operating system. Each task has its own local address range and it can also access the global address range of the operating system. This allows to prevent one task from overwriting data by other tasks, while accessing common data is still possible. But this implies the need for a *global descriptor table (GDT)* for the system and a *local descriptor table (LDT)* for each task. Furthermore each task runs on a certain *privilege level*, which can be used for granting different access rights. Each so–called *selector*, looks as follows (Fig. 4):

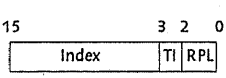

Fig. 4: Selector

Bits 0 and 1 indicate the *requested privilege level (RPL)* for this segment, bit 2 specifies the active table GDT (TI = 0) or the LDT (TI = 1). Bits 3 to 15 are used as an index number into the descriptor table. Each descriptor contains information about the described segment. It includes base address, privilege level, size and type of the segment

(Fig. 5). The base address is 24 bits in size. So, a segment can start anywhere within the possible 16 MB address range. The limit field gives the size of this segment in bytes. This restricts the size of a segment to 64 KB. In other words, the PM of the 80286 forces the same segmentation restrictions as the RM, but it can address a larger memory. For detailed information about this topic we refer to [Dun90].

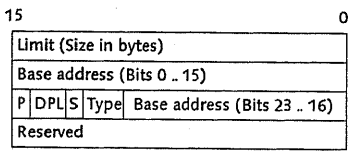

Fig. 5: 80286 segment descriptor

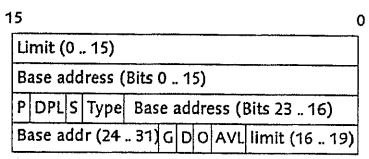

Fig. 6: 80386 segment descriptor

Interrupts

Another important point is interrupt handling. In RM, the interrupt table begins at physical address 0. It contains a 4-byte-vector for each interrupt channel. Interrupt channels are numbered from 0H to 0FFH. Thus, the first 1024 bytes are reserved for interrupt vectors. In PM, the *interrupt descriptor table (IDT)* can be freely located and it also contains 8-byte-descriptors like LDT and GDT. Another nice feature of the 80286 is the following: Intel thought that the old RM would gradually disappear and they provided only a mechanism to switch from RM (which is the initial state after a boot) to PM. However, there is no possibility to switch the processor back from PM to RM, except by a reset of the processor!

80386 processor

In 1985, the 80386 became available. This is a genuine 32-bit processor with a 32-bit hardware bus. In comparison with the old 80286, there are new features which make the processor more usable. E. g., there are new addressing modes (scaled indexing) and all registers can be used for address calculation. Unfortunately, there are still some special instructions that need dedicated registers, as

for example writing to or reading from a port, the divide instruction, etc. The segment descriptor has also been changed (Fig. 6). It is now possible to create a segment that is larger than 64 KB in size. Base addresses are extended to 32 bits, the segment limit to 20 bits and an additional bit in the extended part of the descriptor indicates, whether the limit is in terms of bytes (G = 0) or in terms of 4 KB pages (G = 1). With this, a segment size of 4 GB is now possible, while programs, written for older version of the processor, are still executable (backward compatibility). [Dun90]

Shadow registers

Also, switching back from PM to RM is now possible without a reset operation. But it is not done merely with setting the processor back to real mode. All the segment registers must have a legal RM value afterwards. One could think that this would be done by simply assigning an old RM selector value to the corresponding segment register. However, when in PM, the information about the segment is hold in a shadow register that is not accessible to the programmer. Back in RM, the shadow register seems still to be checked, although all sizes and limits are given for RM. Consequently, before switching back from PM to RM, the selectors must be set to a descriptor with valid values for RM (limit, granularity, ...), otherwise a fault occurs.

Virtual 86 mode

For the so-called V86 mode (not discussed here), see [Dun90]. The virtual 86 mode allows several RM programs running simultaneously. The address is computed like in RM, not using GDT and LDT. So, each RM program has its normal 1 MB address space like on a 8086, although running in protected mode.

2.1 A memory model for Oberon

A memory model had to be found, that allows easy and fast memory access and that was not too complicated for the code generation of the compiler.
Considering the explanations above, there are several possibilities to choose from:

1. Use real mode only
   + : no mode switches necessary
   – : 64 KB segment restriction
   Only 640 KB of conventional memory
   Works with older versions of the 80x86
   compiler/loader must support segmentation

dedicated registers for calculations

2. Use 80286 protected mode
   + : 16 MB of memory accessible
   – : 64 KB segment restriction
   compiler/loader must support segmentation
   dedicated registers for calculations
   mode switches necessary

3. Use 80386 protected mode, 1 big code/data segment
   + : 4 GB of memory accessible
   general purpose registers
   compiler/loader does not have to support segmentation
   linear addresses
   No segment register changes while in PM
   – : mode switches necessary

4. Use 80386 protected mode, separate code and data segments
   + : 4 GB of memory accessible
   general purpose registers
   linear addresses
   – : mode switches necessary
   segment register may change while in PM
   special handling of data and code addresses

The easiest and most advantageous model seems to be model 3 with code and data in the same segment. Model 1 and 2 have too many restrictions and model 4 needs special handling in some situations (E. g., The loader must load code into the code segment and therefore it needs write access for this segment. Because the code segement is normally read–only, the settings of such a segment must be changed for the loading of a module).

2.2 Switch between protected mode and real mode

The code for the switch between real mode and protected mode must be placed in a 16–bit segment below the 640 kB limit, so that it can be executed in both real mode and protected mode. The following procedures show the individual steps for these mode changes. They will be detailed in chapter 5 discussing the Loader and Kernel.

```
PROCEDURE SwitchToProtectedMode;
BEGIN
    Save all segment registers;
    Disable interrupts;
    Load protected mode IDT;
    Get protected mode stack pointer;
    Set protection bit in CR0;
    Flush prefetch queue;
    Execute a FAR-jump to set the CS register;
    Set PM selectors;
    Enable interrupts;
END SwitchToProtectedMode;

PROCEDURE SwitchToRealMode;
BEGIN
    Save all segment registers;
    Disable interrupts;
    Load real mode IDT;
    Get real mode stack pointer;
    Clear protection bit in CR0;
    Flush prefetch queue;
    Execute a FAR-jump to set the CS register;
    Set RM selectors;
    Enable interrupts;
END SwitchToRealMode;
```

2.3 Calls to the Operating System and Traps

MS-DOS is a single-tasking and single-user operating system. As a consequence, when an application is running, the operating system has no control anymore and it is only invoked via a trap or an interrupt. These events can be separated into 3 groups:

- System calls
- Synchronous interrupts
- Asynchronous interrupts

Each system-routine is called via an interrupt that is activated by the application program (software interrupt). The system provides a big library of intrinsic functions. Also, installable drivers offer their functions usually through a system call. Typically, the parameters for a system call are passed in registers. For example, writing a single character to the screen is done by the following routine:

```
MOV AL, 02h          ; select function
MOV DL, char         ; character to be displayed.
                     ; Parameters stored in the registers
INT 21h              ; system call number. Releases an interrupt
```

Synchronous interrupts are exceptions. They are caused by execution faults like 'division by zero' or 'invalid opcode' etc. In contrast, asynchronous interrupts are triggered by hardware devices, e. g. 'timer tick'.
System calls must be executed in RM to guarantee correct functionning, while interrupts can be handled in PM as well, if there exists a corresponding entry in the PM IDT. Otherwise, the interrupt must be rerouted down to RM. This is the task of the so–called *extender*.

Extender

The extender must take care for the correct initialisation of the GDT, LDT and IDT, assuming that we start in RM. Furthermore, it must handle interrupts correctly, e.g. handling them in PM if possible and/or switch the processor down to RM, simulating the interrupt again after setting up the parameters and switching back to PM when the interrupt handling is finished. To summarize, for each interrupt in PM there must be a handler installed of the form:

```
SwitchToRealMode;
Call interrupt again;
SwitchToProtectedMode;
```

This procedure can be illustrated by the following block diagram (Fig. 7):

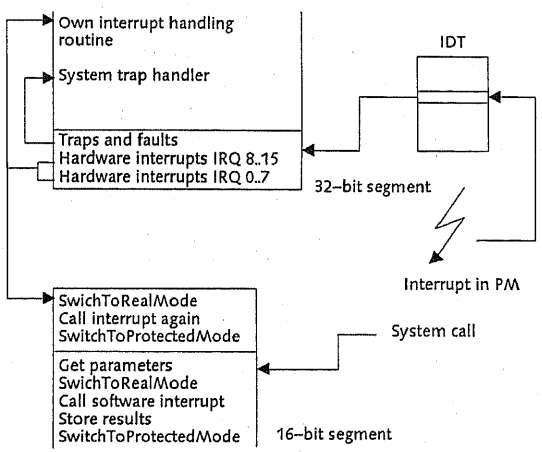

Fig. 7: Extender structure

2.4 The DOS File System

DOS offers two different kinds of service for file management and directory management [Dun90, Bro91]. The first kind is the access via *file control blocks (FCB).* It shouldn't be used anymore. The first DOS versions supported FCBs because the CP/M operating system worked the same way. FCB functions are still supported but *file handles* replaced them in later versions. They are simpler and more flexibel than the FCBs. A similar method is used under UNIX.
The following list gives an overview of the file operations used. For a complete reference, see [Bro91].

File functions used under Oberon:

Get Default Drive (Int 21H, Function 19H):
Gets the current default drive.

Create Subdirectory (Int 21H, Function 39H):
Creates a new subdirectory under the given name.

Remove Subdirectory (Int 21H, Function 3AH):
Removes a subdirectory.

Change Directory (Int 21H, Function 3BH):
Sets the current directory to the one specified.

Open New File (Int 21H, Function 3CH):
Opens a new file or truncates an already existing file under this name.

Open Old File (Int 21H, Function 3DH):
Opens an already existing file.

Close File (Int 21H, Function 3EH):
Flush buffers, update directory information and invalidate file handle.

Read File (Int 21H, Function 3FH):
Read from a file and put data into a buffer.

Write File (Int 21H, Function 40H):
Write data from a buffer into a file.

Delete File (Int 21H, Function 41H):
Delete specified file.

Set Position (Int 21H, Function 42H):
Set position to a specified location in the file.

Dublicate Handle (Int 21H, Function 45H):
When a file must be updated on the disk but should stay open, it is possible to dublicate the handle of this file. The file can then be closed with this new handle. The old handle still gives access to the open file.

Get Current Directory (Int 21H, Function 47H):
Determine current directory.

Get First File (Int 21H, Function 4EH):
Takes a name pattern and looks for matching filenames. Wildcards (*) can be used.

Get Next File (Int 21H, Function 4FH):
Gets the next file matching the name pattern which was specified with Get First File. This function must be called first.

Rename File (Int 21H, Function 56H):
Renames a file from old to new. If old and new are on the same logical volume, rename acts like a move.

Get Date and Time (Int 21H, Function 57H):
Reads date and time of the specified file.

Set Handle Count (Int 21H, Function 67H):
Sets the number of files available to the calling program.

Rider concept

The Oberon file system is much more flexible. Oberon introduces the *rider* concept. A rider is an access mechanism that is associated with the file and denotes a position within this file. Reading and writing is done under control of this rider. More than one rider can be associated with one and the same file at the same time. In conventional models, this corresponds to multiple opened files. See [Wir92] for more details about the Oberon file system.
DOS doesn't know the concept of riders. Although it is possible to have two different handles pointing to the same file, they are rigidly coupled: Changing the position via one handle also changes the position information of the other handle. The rider philosophy is completely missing in DOS.

File names

Another unsatisfactory fact are the short file names. DOS only allows 8 characters plus 3 characters extension, separated by a point, as a filename. Oberon works with full 32 character file names which can use more than one point as a separator. So, a solution had to be found that maps the Oberon names onto DOS names.
Chapter 9 describes the implementation of the Oberon file system under DOS.

2.5 BIOS Calls

The BIOS (Basic In–Out System) contains a collection of routines, providing an interface to important components of the system like screen, keyboard, disk controller, mouse etc. It contains also test

routines (used while booting) and a bootstrap program that loads the operating system from disk [Dun90].
The following groups of BIOS calls are used. For a complete overview of all available calls refer to [Bro91].

Interrupt 10H, Video

Function:	Purpose:
0H	Set video mode
2H	Set cursor position
3H	Get cursor position
9H	Write character

Interrupt 11H, Environment

Function:	Purpose:
no number	Get equipment list

Interrupt 13H, Low-level disk I/O

Function:	Purpose:
0H	Reset drive
2H	Read sector
3H	Put sector
5H	Format track

Interrupt 14H, Serial port

Function:	Purpose:
0H	Initialize port
1H	Write character to port
2H	Read character from port
3H	Get port status

Interrupt 16H, Keyboard

Function:	Purpose:
0H	Get keystroke
2H	Get shift flags
11H	Check for enhanced keystroke

Interrupt 17H, Parallel port

Function:	Purpose:
0H	Write character to port
1H	Initialize port
2H	Get port status

Interrupt 1AH, Timer

Function:	Purpose:
0H	Get system time (Ticks since midnight)
2H	Get real time
3H	Set real time
4H	Get real date
5H	Set real date

Interrupt 33H, Mouse

Function:	Purpose:
0H	Reset driver
3H	Get position and buttons
4H	Set position
7H	Set horizontal limits
8H	Set vertical limits
0FH	Mickey to pixel ratio

3. Memory Model

Low and high memory At this point, we want to declare two expressions, between which must be clearly distinguished. First, there is the so-called *low* or *conventional* memory. It denotes the first MB of physical memory and is a relict of the older versions of the processor which could only address 1 MB of memory. Secondly, there is the *high* or *extended* memory, which denotes all physical memory above 1 MB.

Segment model As discussed in chapter 2, there are several reasons for using a one-segment model as address space for the Oberon system. Most importantly, it is simple to implement and fast. For the definition of the exact layout model, the following facts had to be considered:

1. The video buffer must be contained in the same segment as the code and the data. So, no segment changes are necessary what allows fast access of the display memory.
2. The heap for dynamic memory allocation should be a contiguous piece of memory.
3. A stack for procedure activation must be allocated.
4. A piece of conventional memory for allocating low-level modules is needed. Their purpose is loading the Kernel and passing control to it and to its own memory management. Before the Kernel is loaded and initialized, memory is allocated under the control of the DOS operating system. It's memory allocation routines only manage conventional memory which implies the need of this low memory part. Otherwise, a save deallocation of the used memory is not guaranteed.
5. The entire physical memory should be addressable by Oberon. BIOS and DOS-calls often use buffer addresses to get or return results. These buffers are allocated in the conventional memory, so they should also be addressable under Oberon to access the data.

DOS boot sequence The low memory consists of several parts. After the boot sequence, the first KB contains all 256 interrupt vectors. The next 512 bytes are used for ROM BIOS tables. Then the resident device drivers are allocated, as there are the MS-DOS BIOS (IO.SYS) and the MS-DOS Kernel (MSDOS.SYS), followed by the MS-DOS tables, disk buffers, file control blocks and possible installable device drivers. Finally, there is the resident part of the MS-DOS shell called COMMAND.COM. The resident part is used for terminating programs which where started by COMMAND.COM. Below the 640 KB boundary, there is the transient part of the shell. It gets the commands from the keyboard or from a batch file and executes

them. After each call, the transient part is checked for consistency and possibly reloaded by the resident part. The free RAM between the resident and transient part of the shell is available for programs. Above 640 KB there are the ROM–BIOS and other ROM and RAM, i. e. the display memory. The following pictures (Fig. 1 + 2) show the structure of the low memory. For a complete description of the boot sequence, we refer to [Dun90a].

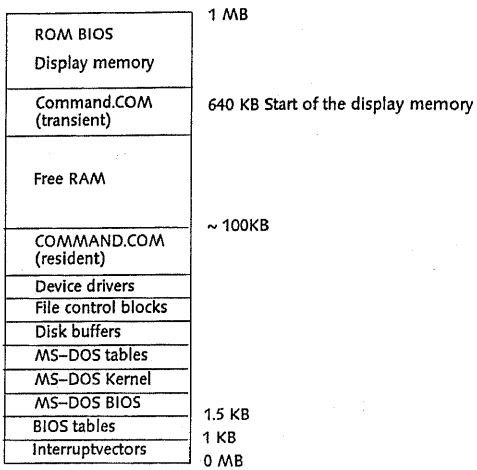

Fig 1: Memory structure after boot [Dun90a]

Oberon boot sequence Before the Oberon memory management gets control, modules *Loader.Obj*, *Modules.Obj* and *Kernel.Obj* and their descriptors are allocated below 1 MB. *Loader.Obj* provides loading modules into memory and relocating variable addresses and procedure calls. *Modules.Obj* is the interface between *Loader.Obj* and the Oberon system. Loading a module is handled by *Modules.Obj* which calls the routines of *Loader.Obj*. *Kernel.Obj* manages memory allocation and system–calls. For more information, see chapter 5: The Loader and the Kernel.

As soon as *Kernel.Obj* is initialized by *Loader.Obj*, it has all the information about the extended memory (start address, size). Then, all requests for memory lead to an allocation in extended memory. The minimal need for extended memory is 2 MB. Although for the basic system 2MB showed to be enough, we recommend 4 MB or more when working with memory intensive applications.

The Oberon garbage collector works only in the heap part of the high memory. There are two reasons for this: First, a contiguous piece of

memory is assumed for garbage collection. This means in particular that the heap cannot be allocated below the video buffer area. The second reason is the fact, that the low memory area is an ideal place to allocate the stack.

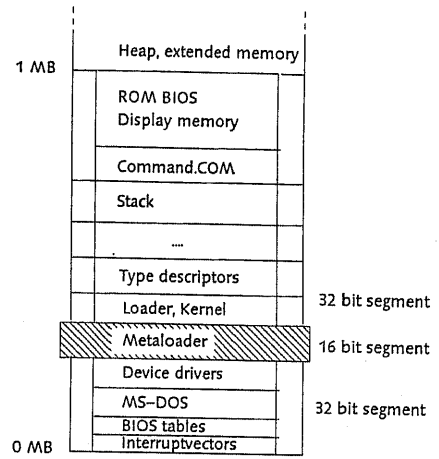

Fig. 2: Memory layout for Oberon

Memory layout

As can be seen in fig. 2, all physical memory is covered by the 32-bit segment. This means that all addresses can be accessed by Oberon. The 16-bit segment (hatched) of the Metaloader is somewhere in memory, depending on the memory structure at Oberon boot time. Note that this small segment can be accessed by Oberon because it is completely embedded in the 32-bit segment. This meta-loader is executable under DOS. It loads the actual module loader and handles system-calls, e. g. takes the function of a so-called *extender*. See also chapter 5: The Loader and the Kernel.

The disadvantage of this model is the amount of low memory that is not used neither by the memory management of Oberon nor by DOS. However, handling this memory especially under Oberon would be too complicated. This memory can be used for additional driver modules that need to be in low memory because they use buffer addresses. In fact, the Oberon compiler provides some support for loading modules in low memory. See chapters 4 and 5 for more details.

Fig. 3 shows the run-time model with the different selectors and registers. Code and data segment of the Oberon system both start at

the same address. This makes it easier to load and initialize the modules because the code segment where the module is loaded into is normally write-protected. The two segments ES and FS point to the same position but they are not used except ES for some special system calls. GS is set to the start of the video buffer. Thus, the display buffer can be accessed without adding an offset from segment start to video buffer start. The two registers ESP (Extended stack pointer) and EBP (Extended base pointer or frame pointer, resp.) point into the stack segment. They are not used for other purposes. EIP (Extended instruction pointer) shows the current position of execution.

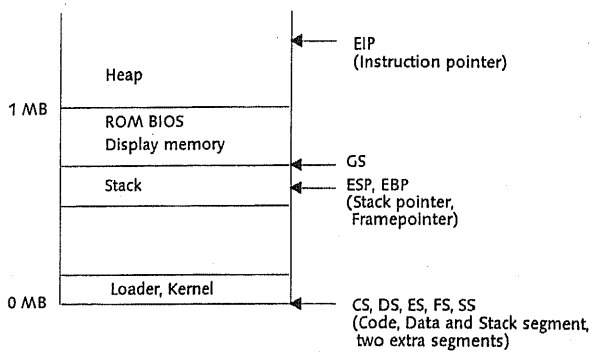

Fig. 3: Run-time Model

4. The Compiler

The compiler was the first program that had to be ported to the DOS platform. Without a compiler, there was no possibility to translate Oberon programs into Intel object code. We decided to take the standard one–pass Oberon compiler [Wir92] as a basis and to adapt the Intel backend parts. The original backend produced code for the National semiconductor NS32000. This microprocessor has several advantages over the Intel chip. The following overview compares the two processors.

4.1 Processor architecture

Intel registers

The 80386 offers 8 32–bit registers, 2 of them are already used for stack management. Each register gives access to either bits 0–15 (16 bit access, compatibility to older processors) or 0–31 (extended 32 bit access). Bits 0–7 and bits 8–15 of EAX, EBX, ECX and EDX can also be used separately.

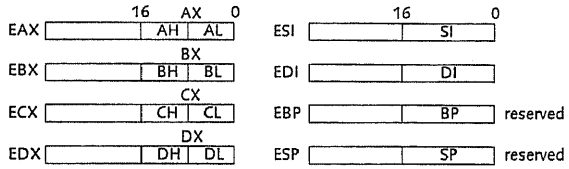

Fig. 1: Intel registers

NS 32000 registers

The NS 32000 has eight general purpose register, each of them can be accessed doublewordwise, wordwise and bytewise.

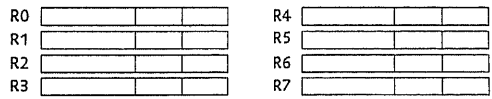

Fig. 2: NS registers

Addressing modes

The calculation of the effective address is staightforward:

*Eff. Address := Base + Index * Scale + Displacement*

Base, Index: General purpose register
Scale: Scalefactor 1, 2, 4 or 8
Displacement: 8–bit or 32–bit displacement wich can be added to the address

Any combination of these three items are possible. This allows the following addressing modes:

mode		Example
Immediate	value	87654321h
Register	Reg	EAX
Register indirect	[Reg + disp]	[EAX], [EAX + 1234h]
Register scaled	[Reg + Ind*scale]	[EAX + EBX*2]
	[Reg + Ind*scale + disp]	[EAX + EBX*4 + 8]
	[Ind*scale + disp]	[EBX*1 + 12345678h]
	[Ind*scale]	[EBX*8]
Absolute	[disp]	[12345678h]

If no segment register is specified, DS is taken by default. All other segment registers must be specified at the beginning.
For example: DS:[EAX + 4] is the same as [EAX + 4]
FS:[EBX + ECX*2]

The NS32000 offers a superset of these addressing modes [NS83]:

mode		Example
Immediate	value	87654321h
Register	Reg	R0
Register indirect	disp(Reg)	0(R0), 1234h(R0)
Memory relative	disp1(disp2(Reg))	8(–4(SP)), 8(–4(FP))
Absolute	@disp	@12345678h
External	EXT(disp1) + disp2	
Top of Stack	TOS	
Memory space	* + disp, disp(FP)	* + 1234h, 1234h(FP)
Scaled	basemode[Rn:i]	0(R0)[R1:B], @12345678h[R1:D], R0[R1:Q]

Disp can be a 7–, 14– or 30–bit signed value.

Dedicated registers

In the 80386 the double indirection and the possibility of memory–to–memory moves are missing. In addition, the 80386 needs dedicated registers for some operations. Although the register set is called general purpose, in reality there are still instructions using special registers only (See below).

Function	Mnemonic	Dedicated registers
Port IO	IN AL, DX	EAX, EDX
	OUT DX, AL	EAX, EDX
Convert byte to word	CBW	EAX
Convert word to double	CWDE	EAX
Compare string	CMPSB, CMPSW, CMPSD	ESI, EDI
Unsigned divide	IDIV EAX, reg/mem	EAX, EDX
(EDX:EAX is the dividend, quotient is stored in EAX, remainder in EDX)		
Move string	MOVSB, MOVSW, MOVSD	ESI, EDI
Repeat	REP	ECX
Rotate, Shift	ROL, ROR, SHL, SHR, SAL, SAR	ECX
Shift double	SHLD, SHRD	ECX
Store String	STOSB, STOSW, STOSD	ESI, EDI

Only the instructions used by the compiler are listed here. For a complete overview, see [Smi87].

Register management

Intel's register philosophy gave a lot of problems in register management. The compiler can very easily get into the situation of needing a dedicated register that is already used. A simple way to prevent such register shortage is to make available the frequently used dedicated registers later than the others.
The chosen order of preference in the compiler is: EAX, EBX, EDX, EDI, ESI, ECX.
Although this works in most cases, it does not exclude conflicts in all cases. Conflicts still can occur. In case of conflict, i. e. if an already occupied register is needed, the compiler temporarily frees the desired register by copying its contents to a arbitary free register or, if all registers are occupied, to the stack. This technique is used e. g. for the DIV and MOD instructions. Note that push–and–pop must take place in a context where we can make sure that the value we get back as a result of pop is the same as the one we pushed before.
In the original compiler, the registers are freed at the end of a statement or a conditional expression only. This is possible because the NS32000 has more registers, memory–to–memory moves and double indirection. There is no need to free registers before.

However, the Intel compiler must free registers as soon as possible, otherwise it will soon run out of registers. Because all the above mentioned features are missing at Intel, additional techniques must be found. They are:

– A memory–to–memory move is done via a memory–to–register move and a register–to–memory move. Obviously, the temporary register can be freed again.
– If a expression consists of several factors and terms, the result of each part is placed in a register. Parts that are not used anymore, return their registers.
– Index registers can be freed immediately after use.
– In case of indirections, the same register can be multiply used.
For example:

```
TYPE Ptr = POINTER TO P;
TYPE P = RECORD next: Ptr; val: INTEGER END;
VAR a: Ptr;
BEGIN a.next.next.next.next.val := 10 END
```

is translated into the following NS32000 code

```
MOVD 0(a(SB)), R7          a.next
MOVD 0(R7), R6             a.next.next
MOVD 0(R6), R5             a.next.next.next
MOVD 0(R5), R4             a.next.next.next.next
MOVXBW 10, 4(R4)           a.next.next.next.next.x := 10
```

or into the following Intel code

```
MOV EAX, DWord [a]         a
MOV EAX, DWord [EAX]       a.next
MOV EBX, DWord [EAX]       a.next.next
MOV EAX, DWord [EBX]       a.next.next.next
MOV EBX, DWord [EAX]       a.next.next.next.next
MOV Word 4[EBX], 10        a.next.next.next.next.x := 10
```

In the first example, a new register is taken with every indirection. The second example reuses the freed registers and thus can compile more complex expressions. The loading of the first part (a.next) can be delayed because the compiler generates an *Ind Item* [Wir92]. With NS32000 code the loading of such an item results in one instruction. With Intel, two instructions are necessary (Memory–to–Register and Register–indirect).

Floating–point

The floating–point unit of Intel (80387 or included in 80486 DX) features a stack architecture in comparison with the register architecture (F0 – F8) of the NS32000. Practically all instructions operate on the top–of–stack element.

Relocation

In the original compiler, constants and global variables are all referenced by the static base (SB). The Intel hasn't got such a register. All places where global variables and constants are used must be relocated with the absolute address at load time. In order to locate relocations, a fixup chain is generated while compiling. Also for external procedure calls, a fixup chain is built up at compile time which is to be relocated at load time. This leads to a separate link section in each object file.

NIL–checks

Because there are no 'save' areas in memory which are read and write protected, separate NIL–checks need to be optionally handled by software. Each time a pointer is dereferenced the compiler generates code for testing whether the address is NIL (= 0) or not. When a program doesn't have dereferences or when the program is verified, so that NIL–dereferences can't occur (still assuming that all other programs are verified and are working correctly, too), this option can be switched off.
Note that Intel offers four special so–called debug registers that each can control a single address. When one of these specified addresses, which are contained in those registers, is referenced, the processor generates a trap. These four debug registers could be used for this purpose, but are not sufficient, because only 16 bytes can be checked this way. A NIL–access is typically something of the form MOV Reg, offset(NIL), where offset can be bigger than 16. It is recommended to compile a save program without NIL–checks with regard to performance and code size. A program working intensively with pointers can get 20 % smaller in size, when compiling without NIL–checks.

Other available options of the compiler are:

- /n NIL–checks off
- /x Index–checks off
- /t Type–checks off
- /o Overflow–checks off
- /s Allow new symbol file
- /i Information about unused and uninitialized variables

4.2 Specific Features

Some processor-specific additions have been made to the Intel compiler. They are all implemented in the SYSTEM module.

SYSTEM.GETREG(reg, value)
SYSTEM.PUTREG(reg, value)
reg: 0 = EAX, 1 = ECX, 2 = EDX, 3 = EBX, 4 = ESP, 5 = EBP, 6 = ESI, 7 = EDI
value: integer or character
These procedures get a value from or put a value into the specified register. The used register size (1 byte, 2 bytes or 4 bytes) is determined by the size of the parameter value.

SYSTEM.PORTIN(port, value)
SYSTEM.PORTOUT(port, value)
port: Port address
value: integer or character
These procedures read a value from or write a value to the specified port. The used register size (1 byte, 2 bytes or 4 bytes) is determined by the size of the parameter value.

SYSTEM.CLI()
Clears the interrupt flag and disables incoming interrupts.

SYSTEM.STI()
Sets the interrupt flag and enables incoming interrupts.

FOR–Statement

Also included in the language is the FOR–Statement.

FORStatement =
FOR *i* := *beg* TO *end* [BY *step*] DO *statement sequence* END.

Beg, *end* and *step* are evaluated at the beginning. They are treated as constants. The *step* variable is added to *i* after each execution of *statement sequence*. If no *step* is declared, it is assumed to be 1. The FOR–statement is equivalent to one of the following sequences:

```
(* Step > 0 *)
i := beg;
WHILE i <= end DO
 statement sequence;
 INC(i, step)
END
```

resp.

```
(* Step < 0 *)
i := beg;
WHILE i <= end DO
 statement sequence;
 DEC(i, step)
END
```

4.3 The Object File Format

The compiler translates source files into so-called object files. They contain besides the coded program information about different sizes used for loading, imported modules, exported procedures, types and variables, as well as fixup chains and type structures. The following notation is used: When the size can be specified, it is noted after a colon.
E. g.: refpos:4 means that refpos is 4 bytes in size.

```
Object File =
    Header Entries Commands Pointers Procvariables Imports
    Links Fixups Code Types Reference.

Header =
    objmark refpos:4 nofentries:2 nofcmds:2 nofpointers:2
    nofimports:2 noflinks:2 noftypdscs:2 datasize:4
    constantsize:2 codesize:4 key:4 name.

Entries = entrytag {entries:4}.
Commands = commandtag {name entry:4}.
Pointers = pointertag {pointer:4} {typedesc:4}.
Procvariables = procvartag.
Imports = importtag {key:4 name}.
Links = linktag {links:4}.
Fixups = fixuptag.
Code = codetag {constants} {code}.
Types =
    typetag {tdsize:2 tdadr:4 recsize:4 nofptrs:2 noftbp:2 {ptrs:4}
    {typeboundprocs:4}}.

Reference =
    reftag {objmark adr:2 name {mode:1 form:1 adr:4 name}}.

name = {byte} 0X.
constants = {byte}.
code = {byte}.

objmark = 0F8X.
entrytag = 81X.
commandtag = 82X.
pointertag = 83X.
procvartag = 84X.
```

importtag = 85X.
linktag = 86X.
fixuptag = 87X.
codetag = 88X.
typetag = 89X.
reftag = 8AX.

4.4 Program Size

The following tables show the differences in size of the original NS–compiler and the new Intel compiler. The Intel compiler was expected to be bigger in size, because of the non–orthogonal instruction set of the processor and the register management.

DOC stands for *D*OS *O*beron *C*ompiler, OC for the NS *O*beron *C*ompiler.

DOS Oberon sizes

Module	Function	Size
DOCS.Mod	Scanner	331 lines
DOCT.Mod	Table handler	608 lines
DOCC.Mod	Code generator	624 lines
DOCE.Mod	Expressions	1345 lines
DOCH.Mod	High level code generator	672 lines
Compiler.Mod	Parser	1024 lines
Total		4594 lines

NS Oberon Sizes

Module	Function	Size
OCS.Mod	Scanner	309 lines
OCT.Mod	Table handler	536 lines
OCC.Mod	Code generator	561 lines
OCE.Mod	Expressions	916 lines
OCH.Mod	High level code generator	506 lines
Compiler.Mod	Parser	920 lines
Total		3748 lines

5. The Loader and the Kernel

5.1 The Metaloader

The metaloader is a bootstrap and interface module. It prepares the necessary memory structure, loads the actual Oberon module loader and passes control to it for loading *Modules.Obj*. In addition, it takes the functions of a DOS-extender. DOS and BIOS calls are passed to the extender part of the mataloader for further handling. The order of the different actions taken by the metaloader is shown below:

1. Get environment variables:
   – The maximum size of the Oberon heap can be specified with SET OBERONMEM = xxxx (KB).
   If no size is given, all available memory is taken.
   – A path to the Oberon directory can be specified with SET OBERON = drive:\path.
   The specified directory is taken as the main directory for the Oberon system (See chapter 9: The Filesystem). The default is the current directory.
   Both environment variables can be set via the AUTOEXEC.BAT file or via the shell.
2. Set correct floppy disk parameters.
3. Build the protected mode interrupt descriptor table (IDT).
   With each interrupt in PM an interrupt vector *selector:offset* is associated.
4. Check for HIMEM.SYS driver and allocate a contiguous memory block of the desired size. This memory is later managed by the kernel.
   HIMEM.SYS is a driver that implements the eXtended Memory Specification (XMS). It provides functions for allocating, changing and freeing extended memory blocks. See also [Dun90]
5. Read the header of the loader object file.
   The header information is needed for allocating a memory block for the code and the data and relocating global variables and procedure addresses.
6. Get low memory for the Oberon module loader, hardware interrupt handling and for the stack. This memory is fully contained in the 32-bit segment and accessible from PM (see chapter 3: Memory Model).

Copy the code for the hardware interrupt handling into the 32-bit segment. Some hardware interrupts (Division by zero, overflow, invalid opcode, segment violation) invoke the Oberon traphandler and are not routed down to RM. In such cases, the interrupt handler pushes a trap number, depending on the occurred trap, onto the stack and calls the Oberon trap handler.

7. Build the global descriptor table (GDT).

```
entry 0 --> Null. Accessing memory with this handle causes a trap
entry 1 ---> Code16 descriptor
  Set 64 KB limit.
  Set base address. This address is found in the current code segment (CS).
  Set appropriate attributes for this segment.
entry 2 ---> Data16 descriptor
  same as above, but this segment must be readable and writeable.
entry 3 ---> Code32 descriptor
  Set limit to specified size (OBERONMEM) or to default.
  Set base address. This address is 0.
  Set appropriate attributes for this segment.
entry 4 ---> Data32 descriptor
  same as above, but this segment must be readable and writeable.
```

8. Read Loader.Obj (the Oberon module loader) and pass important addresses from the metaloader to this loader. Scan through all links and do the fixups.
9. Load the global descriptor table (GDT) and the interrupt descriptor table (IDT), switch to PM and load the stack pointer. Jump to the beginning of the Oberon module loader.

At this point, the Oberon modules take control. The metaloader is only used for system calls and interrupt handling. See chapter 2: The Intel Environment, for the switch mechanism from protected mode to real mode and vice versa.

5.2 The Module Loader

The module loader is responsible for loading and freeing Oberon modules. It has to allocate memory for the module descriptor, the code/data part of the module with additional information and for the type descriptors. It also handles imports and relocates global variables and external procedure calls. When the loader itself is loaded and started, it bootstraps with loading module Modules.Obj (Interface to the loader) which then loads module Oberon.Obj (Contains the main loop of the Oberon system). Before the kernel is loaded, the memory is managed by DOS. This means that all

memory which is needed for loading Modules.Obj and Kernel.Obj (Kernel.Obj is imported by Modules.Obj) is allocated in the first megabyte of memory. For this purpose of the memory a special procedure was implemented:

```
PROCEDURE DOSNew (VAR adr: LONGINT; size: LONGINT);
BEGIN
  Get Memory from DOS. Size must be aligned to 32 bytes boundary.
  Calculate address.
  Build descriptor as in the Oberon NEW-procedure.
  Initialize memory with 0.
END DOSNew;
```

Memory allocation

Memory blocks allocated with this procedure have the same structure as blocks that are allocated by the Oberon memory management. The interrupt number 'MyInt' is a special dummy number not used by other calls. In addition to normal DOS memory allocation the block is registered in a table and freed when the user quits the Oberon system. At the end of an Oberon session, all low memory as well as the extended memory is returned to the system. As soon as the Oberon kernel is loaded and initialized, the memory management is done by the kernel and the above mentioned procedure is not used anymore, except for modules that must be loaded in low memory. This could be driver modules that work with buffer addresses, e. g. CD-ROM drivers. A special tag in the object file indicates whether a module can be loaded normally in extended memory or wether it must be loaded in low memory. Usually, driver modules require low memory.
Apart from this exception, only the high memory is used. We notice again that the garbage collector doesn't take care of the low memory. Any module in low memory can be freed as all other modules, but the garbage collector doesn't collect low memory. However, as low modules are permanent anyway (kernel, loader and drivers) no memory needs to be collected.

Filenames

Module Files provides a translation table for Oberon 32-character filenames. Before this table is initialized, all module names should fit to the 8+3 character DOS names. This concerns modules Kernel.Obj, DOS.Obj, Files.Obj, FileDir.Obj and Modules.Obj.

All modules that are loaded afterwards are not bound any more to the 8+3 character name restriction.
Note again that only Kernel.Obj and Module.Obj are allocated in the low memory and that all the other modules are in the extended memory.
The following figure shows how the descriptors and the code/data

part are organized in detail (Fig. 1).

Relocation

When all module descriptors and type descriptors are allocated and all importing is completed, the loader begins relocating the global variables and external procedure calls. Remember that such items are chained together in fixup chains. A fixup entry consists of an address part (16 bit) and the offset in the code part of the module. In case of a variable, the effective address of the static base (SB) is added to this offset and gives the absolute address of the variable. The address part links to the next fixup in this module. This mechanism gives a restriction to a maximal code size of 64 KB (16 Bit unsigned) per module. In case of an external procedure, the address can be found via import list (module) and entry table (procedure). The fixup technique is the same as for the variables.

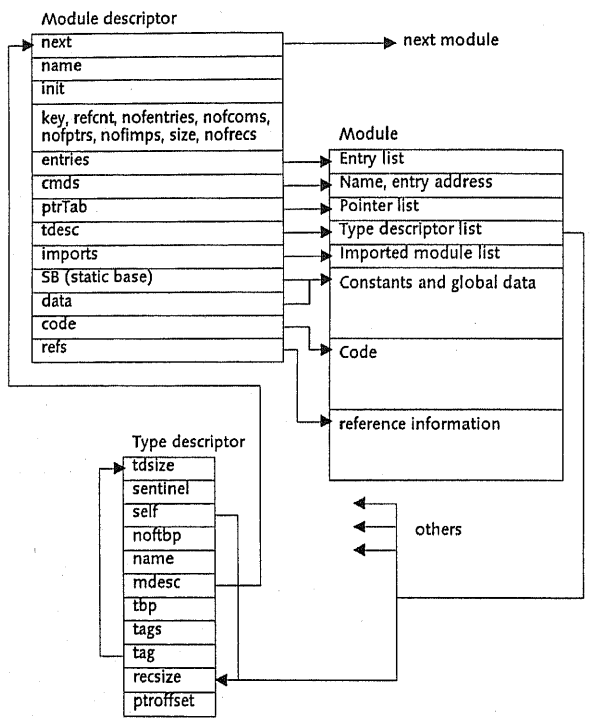

Fig 1: Module Descriptors

The loader is the first module loaded into the Oberon segment (See Chapter 3: Memory Model). Some data of the metaloader are

needed in the Oberon environment. As a side effect, these data are installed by the metaloader when loading the module loader (mentioned above). A similar relation holds between loader and kernel. All loaded modules are inserted in a module chain. This allows access to each module's code and data, as all information about a module is hold in the module descriptor (Fig. 1). The kernel is the first module in this chain. The loader, not included in this chain, can't be accessed by the Oberon runtime system as it is completely isolated. There is no link between an Oberon module and the loader. So, when the kernel is loaded, some important data are copied to the kernel by the loader.
These data comprise for example the address of the DOShandler, an area of data transfer between Oberon and DOS, heap address, video RAM address and different buffer addresses. (See below for the initialisation procedure)

```
PROCEDURE InitKernel (m: Module);
VAR data: LONGINT;
   handler: Proc; load: Proc1; getproc: Proc3; free: Proc4;
BEGIN
   data := m.SB;
   m.refcnt := 1;   (* Kernel can't be unloaded *)
   handler := DOSHandler; load := Load; getproc := GetProc; free := Free;
   SYSTEM.PUT(data-4, SYSTEM.VAL(LONGINT, handler)); (* DOSHandler *)
   SYSTEM.PUT(data-8, SYSTEM.ADR(Reg)); (* Parameterblock *)
   SYSTEM.PUT(data-16, SYSTEM.ADR(Transfer[0])); (* Data transfer area *)
   SYSTEM.PUT(data-20, heapAdr);
   SYSTEM.PUT(data-24, heapSize);
   SYSTEM.PUT(data-28, StackOrg);   (* Stackorigin *)
   SYSTEM.PUT(data-32, IntTransfer); (* Address of int transfer buffer *)
   SYSTEM.PUT(data-36, SYSTEM.ADR(modules)); (* Module list anchor *)
   SYSTEM.PUT(data-40, SYSTEM.VAL(LONGINT, getProc));
   SYSTEM.PUT(data-44, SYSTEM.VAL(LONGINT, load));
   SYSTEM.PUT(data-48, SYSTEM.VAL(LONGINT, free)); (* loader procs *)
   SYSTEM.PUT(data-52, SYSTEM.ADR(main));
   SYSTEM.PUT(data-56, SYSTEM.ADR(File1));
   SYSTEM.PUT(data-60, SYSTEM.ADR(File2));
   SYSTEM.PUT(data-64, Display);
   SYSTEM.PUT(data-68, BufBeg);
   TableRoot := data-72;
   body := SYSTEM.VAL(Proc, m.entries[0]); body; m.init := TRUE;
   SYSTEM.GET(data-76, Routine[0]); (* NEW *)
   SYSTEM.GET(data-80, Routine[1]); (* SYSTEM.NEW *)
   SysNew := SYSTEM.VAL(sysnew, Routine[1]);
   New := SYSTEM.VAL(new, Routine[0]);
```

```
  SYSTEM.PUT (data-102, DS);
  SYSTEM.PUT (data-104, ES);
  KernelLoaded := TRUE;
END InitKernel;
```

5.3 The Kernel

The kernel was originally implemented for the DEC station. It provides procedures for allocating and collecting memory and some trap handling utilities. Only the memory section above 1 megabyte is handled by the kernel. The section below is used by the loader only and is managed purely by DOS. The reason for this is the easier management of one big memory block instead of two. Otherwise, the range from 0A0000H up to 0FFFFFH (Video and part of BIOS) would be right between the two memory blocks. As mentioned above, important data are copied from the metaloader or the loader up to the kernel. These data are listed below.

```
TYPE
  REGISTER = RECORD
               AX, CX, DX, BX: INTEGER;
               SP, BP, SI, DI: INTEGER;     (* SP, BP: read only *)
               Flags: INTEGER;              (* read only *)
               CS, SS: LONGINT;             (* read only *)
               DS, ES: LONGINT
             END;
  Buffer = ARRAY 4096 OF CHAR;

  Buf = POINTER TO Buffer;
  Register = POINTER TO REGISTER;

  (* the declaration order of the following variables is known by the loader! *)
  (* The loader initializes all variables except Reg and Transfer *)

VAR
  INT: PROCEDURE;
  Reg: Register;
  Transfer: Buf;
  heapAdr: LONGINT;
  heapSize: LONGINT;
  StackOrg: LONGINT;
  IntTransfer: LONGINT;
  modules: LONGINT;
  Load: PROCEDURE (mod: ARRAY OF CHAR; VAR m: Module;
                                       VAR res: INTEGER);
  GetProc: PROCEDURE (cmd: ARRAY OF CHAR; m: Module;
                                       VARcommand: Proc; VAR res: INTEGER);
  Free: PROCEDURE (mod: ARRAY OF CHAR; all: BOOLEAN;
                                       VAR res: INTEGER);
  Main: LONGINT;                       (* Address of Mainpath *)
  File1: LONGINT;
  File2: LONGINT;                      (* Address of filenames *)
```

```
  Display: LONGINT;                (* Address of Display *)
  dx: LONGINT;
  TableRoot: LONGINT;
  new: PROCEDURE(tag: Tag): LONGINT;
    (* accessed by the loader to get the address for NEW *)
  systemNew: PROCEDURE(size: LONGINT): ADDRESS;
    (* accessed by the loader to get the address for SYSTEM.NEW *)
```

Interrupt procedure

The Kernel also provides a mechanism for DOS and BIOS calls (see chapter 2: The Intel Environment) and for installing interrupt procedures. The following example shows how such a procedure is installed:

```
PROCEDURE +Int;                  (* Interrupt procedure *)
VAR ...
BEGIN
  SYSTEM.CLI();                  (* disable interrupts *)
  ...                            (* code *)
  SYSTEM.STI();                  (* enable interrupts *)
  SYSTEM.PORTOUT(20H, 20H)       (* EOI, end of interrupt *)
END Int;

PROCEDURE Install*(port: INTEGER);
BEGIN
  IF port = COM1 THEN Kernel.InstallIP(Int, 4)
    (* Install procedure Int. The second parameter in InstallIP denotes
       into which hardware interrupt the procedure should be hooked.
       Inthis case into COM1 interrupt *)
  ELSIF port = COM2 THEN Kernel.InstallIP(Int, 3)
    (* Hook into COM2 interrupt *)
  (* ELSIF ... *)
  ELSE HALT(88) END;
END Start;
```

6. The DOS Module

Module DOS is the interface between the MS-Disk Operating System and Oberon. In particular, it provides an interface to all DOS and BIOS calls that are needed by the Oberon system. Although such calls could be made by any module in situ, they are concentrated in one place. This gives a clear interface to DOS low-level routines.

Parameter passing

The parameters of a system call are normally passed to the operating system in registers. In some cases, e. g. where a string is needed, a memory buffer is used. In such cases, the parameter register contains the address of the data buffer. The actual call is implemented as a software interrupt. The following assembler statements give an example:

```
(* Get system clock *)
MOV AH, 2H   (* Set function number *)
INT 1AH      (* Generate interrupt 1AH *)
```

Results are also passed in registers. Moreover, some calls raise the carry bit, if the operation was not successful, others pass the 'successfully done' information in a register, typically EAX.
In module Kernel, there exists a global variable *Reg* providing the address of the parameter block and a buffer called *Transfer* supporting data transfer between Oberon and DOS. System calls must be executed in real mode and can't therefore access the memory above one MB. All data transfer is therefore done via the transfer buffer. Instead of calling an interrupt directly, a procedure is called that switches back to real mode, copies the parameters from *Reg* to the registers, invokes the corresponding interrupt, writes back the results into *Reg* and switches back to protected mode. The following procedure shows this:

```
PROCEDURE SystemCall;
BEGIN
  Store the parameters for the system call into the register variable;
  Fill transfer buffer, if needed;
  Switch to real mode;
  Copy the contents of the register variable into the real registers;
  (* The transfer buffer is already in real mode, nothing needs to be done here *)
  Invoke specified interrupt;
  Copy results back into the register variable;
  Switch to protected mode;
END SystemCall;
```

This shows the definition of the parameter block and transfer buffer:

```
TYPE
  REGISTER = RECORD
               AX, CX, DX, BX: INTEGER;
               SP, BP, SI, DI: INTEGER;          (* SP, BP: read only *)
               Flags: INTEGER;                   (* read only *)
               CS, SS: LONGINT;                  (* read only *)
               DS, ES: LONGINT
             END;

  (* Results in field Flags
     Bit 0: Carry CF          Bit 4: Auxiliary AF
     Bit 1: unused            Bit 5: Overflow OF
     Bit 2: Parity PF         Bit 6: Zero ZF
     Bit 3: unused            Bit 7: SignSF *)

  Buffer = ARRAY 4096 OF CHAR;
  Buf = POINTER TO Buffer;
  Register = POINTER TO REGISTER;


VAR
  INT: PROCEDURE;
  Reg: Register;
  Transfer: Buf;
```

System calls

A call to the operating system is done as shown below:

```
PROCEDURE WaitForKeyboard*; (* Call without buffer *)
BEGIN
  Kernel.Reg.AX := 700H;
  Kernel.INT(Kernel.Reg, 21H);
END WaitForKeyboard;


PROCEDURE DeleteFile*(name: ARRAY OF CHAR); (* Call with buffer *)
VAR i: INTEGER;
BEGIN i := 0;
  Kernel.Reg.AX := 4100H;
  Kernel.Reg.DS := Kernel.DS; (* DS:DX is the start of the transfer buffer *)
  Kernel.Reg.DX := Kernel.DX;
  REPEAT Kernel.Transfer[i] := name[i]; INC(i) UNTIL name[i] = 0X;
  Kernel.Transfer[i] := 0X;
  Kernel.INT(Kernel.Reg, 21H)
  Done := ~SYSTEM.BIT(SYSTEM.ADR(Kernel.Reg.Flags), 0);
END DeleteFile;
```

System parameters

Module DOS also contains some system parameters.

```
VAR
  Done,                          (* Last call was successful *)
  CoAvail: BOOLEAN;              (* Coprocessor available *)
  DispHeight, DispWidth: INTEGER; (* Screen resolution *)
  tag: ARRAY 3 OF CHAR;          (* Tag of the Oberon files *)
```

Available procedures Additionally, the following service procedures are available:

```
Output to the screen and wait:
   PROCEDURE WriteChar(ch: CHAR);
   PROCEDURE WriteString(s: ARRAY OF CHAR);
   PROCEDURE WriteInt(i: LONGINT);
   PROCEDURE WriteLn;
   PROCEDURE WriteHex(i: SYSTEM.BYTE);
   PROCEDURE Wait;

Serial port:
   PROCEDURE InitSerialPort(port, wlength, stopbits, parity, baud: INTEGER);
   PROCEDURE RecCharSer(port: INTEGER; VAR ch: CHAR);
   PROCEDURE SendCharSer(port: INTEGER; ch: CHAR);
   PROCEDURE GetStatusSer(port: INTEGER; VAR status: INTEGER);

Parallel port:
   PROCEDURE InitParallelPort(port: INTEGER);
   PROCEDURE PrintChar(port: INTEGER; VAR ch: CHAR);
   PROCEDURE GetStatusPar(port: INTEGER; VAR status: INTEGER);

Floppy drive:
   PROCEDURE Format;
   PROCEDURE ResetDrive;
   PROCEDURE GetSector(sec, cyl, head: INTEGER);
   PROCEDURE PutSector(sec, cyl, head: INTEGER);

Timer:
   PROCEDURE GetClock(VAR time, date: LONGINT);
   PROCEDURE SetClock(time, date: LONGINT);
   PROCEDURE GetTicks(): LONGINT;

Keyboard:
   PROCEDURE KBAvail(): BOOLEAN;
   PROCEDURE GetChar(VAR ch: CHAR; VAR ext: BOOLEAN);
   PROCEDURE ControlKeys(VAR Keys: INTEGER);

Display:
   PROCEDURE InitDisplay(mode: INTEGER);
   PROCEDURE GetColor(col: INTEGER; VAR red, green, blue: INTEGER);
                              (* implemented in the driver *)
   PROCEDURE SetColor(col, red, green, blue: INTEGER);
                              (* implemented in the driver *)

Mouse:
   PROCEDURE InitMouse;
   PROCEDURE SetMouse(x, y: INTEGER);
   PROCEDURE GetMouseInfo(VAR x, y, buttons: INTEGER);

Files:
   PROCEDURE Open(VAR name: ARRAY OF CHAR; new: BOOLEAN;
                  VAR handle: LONGINT; accessmode: INTEGER);
   PROCEDURE Close(handle: LONGINT);
   PROCEDURE Delete(name: ARRAY OF CHAR);
   PROCEDURE Rename(old, new: ARRAY OF CHAR);
   PROCEDURE Length(VAR len: LONGINT; handle: LONGINT);
   PROCEDURE DirOpt(dos: ARRAY OF CHAR; VAR time, date, size: LONGINT);
   PROCEDURE GetDateTime(VAR date, time: LONGINT; handle: LONGINT);
   PROCEDURE SetPos(pos, handle: LONGINT);
```

```
    PROCEDURE DoubleHandle(handle: LONGINT): LONGINT;
    PROCEDURE Write(size, adr, handle: LONGINT);
    PROCEDURE Read(size, adr, handle: LONGINT; VAR read: LONGINT);
    PROCEDURE GetID(VAR ID: ARRAY OF CHAR; handle: LONGINT);
    PROCEDURE SetID(ID: ARRAY OF CHAR; handle: LONGINT);
    PROCEDURE GetTag(VAR tag: ARRAY OF CHAR; handle: LONGINT);
    PROCEDURE SetFileCount(no: INTEGER);
    (* for future use *)
    PROCEDURE LockFileAccess(handle, Offset, Length: LONGINT;
                                shareable: BOOLEAN): BOOLEAN;
    PROCEDURE UnlockFileAccess(handle, Offset, Length: LONGINT;
                                shareable: BOOLEAN): BOOLEAN;
    PROCEDURE IsHandleRemote(handle: LONGINT): BOOLEAN;
    PROCEDURE BufferingRecommended(AccessMode: INTEGER;
                                FileRemote: BOOLEAN;
                                VAR BufferedRead, BufferedWrite: BOOLEAN);

File directory:
    PROCEDURE GetFirstFile(VAR name: ARRAY OF CHAR);
    PROCEDURE GetNextFile(VAR name: ARRAY OF CHAR; VAR end: BOOLEAN);
    PROCEDURE MakeDir(name: ARRAY OF CHAR);
    PROCEDURE RemoveDir(name: ARRAY OF CHAR);
    PROCEDURE ChangeDir(name: ARRAY OF CHAR);
    PROCEDURE GetDir(drv: ARRAY OF CHAR; VAR name: ARRAY OF CHAR);
    PROCEDURE GetDrive(VAR drv: ARRAY OF CHAR);

Miscellaneous:
    PROCEDURE AllocTermProc(proc: Proc);
    (* Procedures which are called before the system quits,
                                e. g. close all temporary files *)
    PROCEDURE Quit; (* Quits the system *)
```

7. The Display

The display driver was one of the most time critical parts of the whole implementaton. It must be fast to give a good responsiveness of the system and it should also be able to work with a reasonable resolution. While there is a big choice of different interface cards and standards, we had to restrict our implementation to just a few of them. Some of the standards are SVGA, VGA, EGA, MCGA, MDA, Hercules, ...For more information about interface cards and standards, see also [Fer91].

The original Oberon system operates in a monochrome graphic mode with 1024 * 800 pixels. The aim was a resolution that optimally matches with our Ceres machines. Also, we wanted to have a standard that is as wide–spread as possible.

Monochrome display

In a first step, a display driver was written for a VGA card with just 640*480 pixels and a monochrom display. For efficiency reasons, this module is coded in assembler and BIOS is bypassed, i. e. the video RAM is accessed directly.

Color display

The next step was a color display. The selected card offers 16 colors with VGA resolution, the same number of colors as our Ceres machines have. This implementation is still maintained together with all the other parts of the port [Ott92].

ET4000 card

Later we developed a driver for the ET4000 SVGA card with a resolution of 1024*768 and 256 colors because ET4000 is well known and fast. We also evaluated the Trident 8900. However, while implementing, it turned out that the ET4000 with two bank registers is faster than the Trident with only one bank register (especially for Copy Block). For precise information about both interface cards, see [Fer91].

But still the display was slow in comparison with our original Ceres machine.

S3 card

We had now nearly the same display resolution with even more colors though, but we wanted a faster card. One possibility was to use an accelerator card. In fact, we found the ideal solution in the S3 chip set. It is not very expensive and is high performance thanks to its own graphic processor.

Our development machine features a S3C805 local bus version of this chip, running at a resolution of 1024 * 768 pixels and 256 colors. It is approximately 4–5 times faster than the Ceres color display at a resolution of 1024 * 800 and 16 colors.

The S3 chip knows these 4 operations:
- Draw line
- Fill rectangle
- Bit block transfer
- NOP, resp. set parameters

Perhaps surprisingly, this is just enough for our needs. Namely, all Oberon display procedures can be mapped to one of the above mentioned commands:

– CopyPattern	—>	Draw line (textured)
– CopyBlock	—>	Bit block transfer
– ReplConst	—>	Fill rectangle
– ReplPattern	—>	Draw line (textured)
– FillPattern	—>	Draw line (textured)
– Dot	—>	Draw line

In addition to Oberon's basic display operations, the S3–card even implements an operation Line to draw a general line.

Clipping

Also, the S3–processor implements clipping in hardware. The chip allows setting up a rectangular area that specifies the current boundaries. Subsequent operations optionally draw inside or outside the clipping area.

Pattern expansion

Working now with 256 colors, we use 8 bits per pixel. However, all Oberon fonts are stored as a black/white pattern with 1 bit per pixel. Fortunately, the S3 chip offers a possibility to specify a background and a foreground color and an associated automatic expansion mode. 0's in a pattern are mapped to background color, 1's to foreground color. The combination of the two colors are specified in a special fg/bg mix register. For details, see below.

Bg/Fg Mix Register (bits 0 – 3)

Value	Meaning	Used for
0	inverted screen	
1	all bits = 0	replace mode (Background)
2	all bits = 1	
3	no change	invert and paint mode (Background)
4	inverted color	
5	screen XOR color	invert mode (Foreground)
6	inverted screen XOR color	
7	color	paint and replace mode (Foreground)
8	inverted screen OR inverted color	
9	screen OR inverted color	
10	inverted screen OR color	

11	screen OR color
12	screen AND color
13	inverted screen AND color
14	screen AND inverted color
15	inverted screen AND inverted color

Parameter passing

The parameters are passed to the interface card registers via I/O ports. In principle, this makes programming very simple. There is only one fact to care for. The card manages a FIFO queue for instructions. It depends on the type of card whether 8 or 16 stack entries are available. Before any value can be written to a card register, it must be guaranteed that there is a free queue entry to hold them. The easiest and also recommended way is a preliminary check for empty queue. If there are no entries available, the program should wait until the queue is empty, otherwise an overflow interrupt is generated. For detailed information, see [S3 92].

The following code sequences show how compact the raster operations can be implemented when using the S3 card:

```
PROCEDURE InitDisplay;   (* Sets the display into S3 mode *)
BEGIN
  Kernel.Reg.AX := 4F02H;
  Kernel.Reg.BX := 205H;   (* S3 1024 x 768 x 256 (VESA) *)
  Kernel.INT(Kernel.Reg, 10H);
END InitDisplay;

PROCEDURE WaitFIFOempty;   (* Waits, till the FIFO stack is emtpy *)
BEGIN
  (*$ DInline.Assemble
    MOV DX, $9AE8
lab1  IN AX, DX
    AND AX, $200
    JNZ lab1
  END *)
END WaitFIFOempty;

PROCEDURE Dot(col, x, y, mode: INTEGER);
BEGIN
  y := Height-y;
  WaitFIFOempty;
  IF mode = invert THEN
      SYSTEM.PORTOUT(FGmix, LONG(25H))  (* Set Fg mix register *)
  ELSE
      SYSTEM.PORTOUT(FGmix, LONG(27H))
  END;
  SYSTEM.PORTOUT(FGcol, col);                    (* Set Fg color *)
  SYSTEM.PORTOUT(MFcont, SHORT(pixctrl1));(* Function control *)
  SYSTEM.PORTOUT(curX, x);                       (* X coordinate *)
  SYSTEM.PORTOUT(curY, y);                       (* Y coordinate *)
  SYSTEM.PORTOUT(cmdReg, 121BH);                 (* Command *)
  SYSTEM.PORTOUT(shortStroke, LONG(10H)); (* Draw dot *)
END Dot;
```

```
PROCEDURE CopyBlock*(SX, SY, W, H, DX, DY, mode: INTEGER);
VAR xpos, ypos: INTEGER;
BEGIN
   IF (W <= 0) OR (H <= 0) THEN RETURN END;
   xpos := 0; ypos := 0;
   IF SY < DY THEN INC(SY, H-1); INC(DY, H-1); ypos := 128 END;
   IF SX < DX THEN INC(SX, W-1); INC(DX, W-1) ELSE xpos := 32 END;
   SY := Height - SY; DY := Height - DY;
   DEC(W); DEC(H);
   WaitFIFOempty;
   IF mode = invert THEN SYSTEM.PORTOUT(FGmix, LONG(65H))
   ELSE SYSTEM.PORTOUT(FGmix, LONG(67H)) END; (* Fg mix register *)
   SYSTEM.PORTOUT(MFcont, SHORT(pixctrl1));    (* Function control *)
   SYSTEM.PORTOUT(curX, SX);                   (* Source X *)
   SYSTEM.PORTOUT(curY, SY);                   (* Source Y *)
   SYSTEM.PORTOUT(diaStep, DX);                (* Desination X *)
   SYSTEM.PORTOUT(axStep, DY);                 (* Desination Y *)
   SYSTEM.PORTOUT(majAxis, W);                 (* Width *)
   SYSTEM.PORTOUT(MFcont, H);                  (* Height *)
   SYSTEM.PORTOUT(cmdReg, SHORT(copycmd) + xpos + ypos);
                                               (* Command *)
END CopyBlock;
```

8. Floating Point Arithmetic

Not all of the Intel processors have a floating point unit (FPU) on chip. Some (80386SX, 80486SX) need a special chip (80x87) containing such an FPU. Although the basic Oberon system doesn't do REAL computations, it is desirable to have floating point operations at disposal. For this reason, an emulator for the 80x87 was written. It is based on the MIPS R2010 floating point emulator. The non-portable parts (code procedures with register conventions) have been adapted for the DOS implementation.
When a floating point instruction should be executed and there is no FPU installed, the system traps. The idea is now to hook into this interrupt and call the FPU emulator as interrupt handling procedure. After handling the interrupt, the program continues execution normally. Note that the emulation also includes the FPU stack to be updated.

Emulated instructions

The emulator merely knows the instructions ADD, SUB, DIV and MULT plus ABS and NEG. All other instructions (e. g. trigonometric functions) must be reduced to this base. (See also below: Math modules).

Address decoding

So, the emulator is invoked upon trap 7 (no coprocessor). The task of the floating point emulator now consists of: (1) decode the address and (2) interprete the function. Interpreting the function was already implemented, but decoding the address had to be redone. For that purpose, we used a trick. Assume that the needed REAL number is at a location off[Reg1+Reg2*4]. Then the absolute address can be computed with LEA Reg, off[Reg1+Reg2*4]. So, the whole decoding is done by the processor's own decoding facility. All we need do is placing an LEA (Load effective address) in front of the address part of the floating point instruction and executing this instruction. This way of implementing selfmodifying code turned out to be very useful, elegant and even safe!
The following code shows how it is done in detail:

```
PROCEDURE *Adr;      (* Empty procedure; lets space for 10 bytes *)
BEGIN HALT(32) END Adr;

PROCEDURE GetAdr(start, len: LONGINT);
VAR Padr: LONGINT; nop: INTEGER; code: SHORTINT; P: Proc;
BEGIN
  P := Adr;
  Padr := S.VAL(LONGINT, P);                    (* Procedure Address *)
  nop := 9090H;                                 (* 90H, 90H *)
  S.PUT(Padr+1, nop); S.PUT(Padr+3, nop);       (* Init with NOP *)
  S.PUT(Padr+5, nop); S.PUT(Padr+7, nop);
```

```
  S.PUT(Padr+9, nop);
  IF len # 0 THEN S.MOVE(start+1, Padr+3, len) END; (* Put Code *)
  S.GET(start, code);
  S.PUT(Padr, 90X); S.PUT(Padr+1, 8DX);          (* Put NOP, LEA *)
  S.PUT(Padr+2, code MOD 8 + code DIV 64*64); (* Put dest to EAX *)
  S.PUTREG(0, EAX); S.PUTREG(1, ECX);            (* Set registers *)
  S.PUTREG(2, EDX); S.PUTREG(3, EBX);
  S.PUTREG(6, ESI); S.PUTREG(7, EDI);
  S.GETREG(5, oldebp); S.PUTREG(5, EBP);         (* Set old EBP *)
  Adr; S.GETREG(0, adr);                         (* Get Address *)
  S.PUTREG(5, oldebp);                           (* Reset EBP *)
END GetAdr;
```

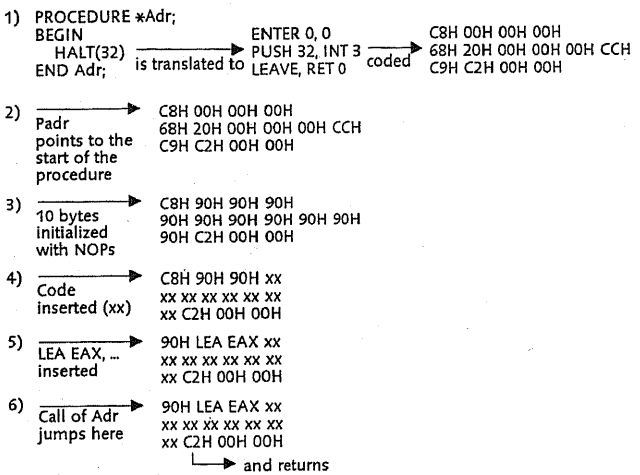

Fig. 1: Process of code patch

All registers have to be saved at the beginning of the interrupt handling procedure and must be restored again before the address can be computed. The frame pointer (EBP) is saved before calling *Adr* and restored afterwards.

8.1 Math modules

As mentioned before, the emulator doesn't know other floating point operations than ADD, SUB, MULT and DIV. The two math libraries, Math.Obj and MathL.Obj had to be adapted and optimized for emulation, because the 80387 features instructions for SQRT,

logarithmic and trigonometric functions and many more. The existence of a math coprocessor is checked at startup time and the flag DOS.CoAvail is set appropriately. In the math modules, two variants are implemented, one for the coprocessor and the other for the emulator. (See below)

Example for the Sin(x) function:

```
PROCEDURE -Sin(x: REAL):REAL
   0C8H, 0, 0, 0,          (* ENTER 0, 0 *)
   0D9H, 45H, 08H,         (* FLD ST(0), 8[EBP] *)
   0D9H, 0FEH,             (* FSIN ST(0) *)
   0DDH, 0D9H,             (* FSTP ST(1) *)
   0C9H, 0C2H, 4, 0;       (* LEAVE, RET 4 *)

PROCEDURE sin*(x: REAL): REAL;
VAR n: LONGINT; y, yy, f: REAL;
BEGIN
   IF DOS.CoAvail THEN  (* Coprocessor exists, use float instructions *)
      IF x < 0.0 THEN RETURN -Sin(-x) ELSE RETURN Sin(x) END
   ELSE  (* No coprocessor available, emulate function *)
      y := c31*x; n := ENTIER(y + 0.5);  (*c31 = 2/pi*)
      y := 2*(y-n); yy := y*y;
      IF ~ODD(n) THEN f := ((p33*yy + p32)*yy + p31) / (p30 + yy) * y
      ELSE f := ((q33*yy + q32)*yy + q31) / (q31 + yy) END;
      IF ODD(n DIV 2) THEN f := -f END;
      RETURN f
   END
END sin;
```

c31, p33, p32, p31, p30, q33, q32, q31 are REAL contants used for emulation.

This emulation still leaves room for improvement in precision, but it works fine for normal requirements.
The other procedures in the math module are implemented in a similar way.

Definition of Math.Obj:

```
DEFINITION Math;
   CONST e = 2.7182817E+00;
   VAR pi: REAL;

   PROCEDURE arctan (x: REAL): REAL;
   PROCEDURE cos (x: REAL): REAL;
   PROCEDURE exp (x: REAL): REAL;
   PROCEDURE ln (x: REAL): REAL;
   PROCEDURE sin (x: REAL): REAL;
   PROCEDURE sqrt (x: REAL): REAL;
END Math.
```

Definition of MathL.Obj:

```
DEFINITION MathL;
   CONST e = 2.71828182845905D+000;
   VAR pi: LONGREAL;

   PROCEDURE arctan (x: LONGREAL): LONGREAL;
   PROCEDURE cos (x: LONGREAL): LONGREAL;
   PROCEDURE exp (x: LONGREAL): LONGREAL;
   PROCEDURE ln (x: LONGREAL): LONGREAL;
   PROCEDURE sin (x: LONGREAL): LONGREAL;
   PROCEDURE sqrt (x: LONGREAL): LONGREAL;
END MathL.
```

There are many more functions provided by the coprocessor, but they are not used to implement the standard Oberon floating point interface.

9. The File System

As can be seen in [Wir92], the Oberon file system is very flexibel and offers new perspectives in comparison to conventional file systems and in particular to the DOS file system. Although there are no subdirectories under Oberon, the 32 character filenames offer enough flexibility to create groups of files in the same directory, i. e. files with the same prefix or suffix belong to the same group. DOS does not allow filenames longer than 8+3 characters. Also files can't be multiply opened with independent read and write pointers. DOS files with different handles necessarily refer to the same read/write position.
The base for the DOS file system was the implementation for the UNIX implementation.

Oberon directory

The lack of long file names makes necessary two separate directories. The first one is the normal directory provided by the DOS operating system, the second one is an Oberon directory mapping Oberon file names to DOS file names. The Oberon directory is implemented as a special file called *FILENAME.TEX*. It is read into memory at startup time. Its memory representation is a simple linear list in alphabetical order with a sentinel (Fig. 1).

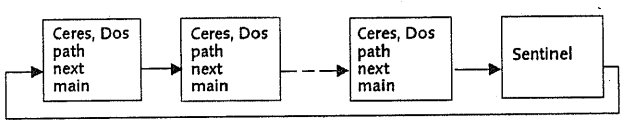

Fig. 1: Oberon Directory

Each entry in the Oberon directory looks as follows:

```
NodePtr = POINTER TO Node;
Node = RECORD
   Ceres, Dos: FileName;
                        (* 32 character name, corresponding 8+3 DOS name *)
   path: Path;          (* Subdirectory in which the file is located *)
   next: NodePtr;       (* Pointer to next node *)
   main: BOOLEAN;       (* Main or current directory *)
END;
```

Translation table

When the user quits Oberon, the translation table is written back to disk. If unforeseen circumstances hinder a correct shutdown of the Oberon system, the Oberon directory can not possibly be written to

disk. If at system startup time no *FILENAME.TEX* is found, then all files in the Oberon DOS directory are opened and individually checked. The first two bytes contain the Oberon tag. The following 32 bytes give the Oberon name and the corresponding Oberon/DOS pair can be inserted into the directory. If there is no tag, the Oberon name by definition is the same as the DOS name. A similar algorithm is used if a file was added or deleted under control of DOS in the Oberon directory. If there are two different files with the same Oberon name, then the older one is deleted. Thus, the translation table is guaranteed to be consistent at any time.

File handle

Each file is internally represented as a pointer to a file handle, i. e. to a complete set of information about the file.

```
File = POINTER TO Handle;
Handle = RECORD
   origName,              (* Original 32 character name *)
   workName,              (* Current name under DOS *)
   registerName           (* Registered DOS name, empty when new file *)
      : FileName;
   path: FileDir.Path;    (* Subdirectory in which the file is located *)
   fd,                    (* File handle number *)
   len,                   (* File length *)
   pos: LONGINT;          (* Current file position *)
   bufs                   (* Files up to 16 KB are held in memory *)
      : ARRAY nofbufs OF Buffer;
   swapper,               (* Indicates which buffer will be swapped out next *)
   state: INTEGER;        (* State: create (file is in buffers), open and close *)
   dospath: BOOLEAN;      (* File is in the current/main directory or in any
                             subdirectory *)
   offset: SHORTINT;      (* Oberon files start at offset 34 *)
END;
```

Files.New

Each time a new file is created by *Files.New(name)*, a new file handle is allocated, the tag and the name are inserted into the first buffer and the file name is copied into the *origName* field. No disk access is done at this point. Every file has 4 memory buffers of 4 KB size each. Small files can thus be kept completely in memory, which guarantees fast access. If all buffer space is used, a temporary file is created under DOS named *1.xxx*, where *xxx* denotes a consecutive number. The record field *origName* still keeps the original Oberon filename. Although a file is created on disk now, it is not inserted in the Oberon directory yet. This and the renaming to *origName* (as far as the 8+3 character limit allows) is first done when the file is registered by *Files.Register(f)*.

Other files with the same Oberon name can be opened at the same time. Then, temporary files with different names (1.yyy, see above) are created, but all files have the same *origName*.

All temporary files that have been created but not renamed, i. e. have

not been registered under Oberon, are deleted at the next system startup time.

Files.Old

If a file is opened with *Files.Old(name)*, the correponding DOS name is looked up in the mapping table and the file is accessed by DOS under its DOS name. Otherwise, if the name is not in the Oberon directory, NIL will be returned. Every opened file is inserted in a *cache*, so that a second call of *Files.Old(name)* will find the already opened file in the cache and return the same file handle.

```
CONST cacheSize = 64;
VAR cache: ARRAY cacheSize OF LONGINT (* = File *);
```

Files.Register

Files.Register(file) flushes all file buffers and inserts the Oberon/DOS name pair into the Oberon directory. If the file was already created earlier under a temporary name (e. g. *1.xxx*), it must be closed, renamed to the corresponding DOS name and reopened with now the DOS name. This is imposed by DOS that doesn't allow renaming of open files. A small file which can be held in buffers may be created directly under the correct DOS name. Since no temporary file was created, renaming is not necessary. All registered files are also inserted in the cache, which makes them available for future access.

DOS files

It is now possible to access DOS files in all other DOS directories, even on a floppy disk. However, to this purpose, the scanner must allow the colon ":" and the backslash "\" as valid characters in names. If a file within a directory path is to be opened, e. g. *a:\mydir\test.mod*, the file module checks the file name for "\" in the name, which indicate a DOS path. The file *test.mod* is then searched in the directory *a:\mydir* instead of the current directory. Similarly, if a file is stored and its name contains any "\", the given path is taken instead of the default *current* path. The so stored file doesn't contain the tag and the Oberon name and it is not inserted in any mapping table. This makes it possible to store files directly onto a DOS disk from within Oberon. This is a technique to access files external to the Oberon system and it could replace the *System.CopyToDOS* and *System.CopyFromDOS* export and import commands.

Subdirectories

Another way to put files into different directories is to use the *subdirectory facility*. To that purpose, module System offers four new commands:

```
System.MakeDir ~
        Creates a new subdirectory,
        e. g. System.MakeDirectory C:\WORK\NEWDIR
```

```
System.RemoveDir ~
        Removes specified subdirectory,
        e. g. System.RemoveDirectory C:\WORK\NEWDIR
        The subdirectory must be emptied before removing it.

System.CurrentDir
        Shows the current directory

System.ChangeDir ~
        Sets the current directory to the one specified,
        e. g. System.ChangeDirectory C:\WORK\SOURCES~
```

In fact, Oberon maintains two directories called *main* and *current*. After booting the system both directories are set to the actual current directory under DOS. If an environment variable for Oberon was declared in the AUTOEXEC.BAT file, it will be taken as the current directory. All files will then be located in the *current* directory which can be changed with the System.ChangeDir command. A desired file is first looked up in the *current* directory and then in the *main* directory. Files in *main* cannot be deleted. Thus, *main* is predestined to hold a stable version while new implementations are allocated in a special subdirectory. Transfers between *current* and *main* can be done with *System.CopyFiles*.

```
System.CopyFiles C:\WORK\TESTS\NEW.OBJ => New.Obj~
(* The current directory is set to C:\WORK *) or
System.CopyFiles New.Obj => C:\WORK\NEW.OBJ~
(* The current directory is set to C:\WORK\TESTS *)
```

If in these commands the destination directory is either *current* or *main*, the Oberon DOS directory is updated, otherwise it is assumed that the destination is an ordinary DOS directory without a mapping table.

Each change of *current* forces the mapping table to be backed up onto disk, so each subdirectory ever accessed by Oberon contains a file called FILENAME.TEX. This enables fast access when the directory is used again later.

References

[Bro91] Ralf Brown, Jim Kyle: PC Interrupts
A programmer's reference to BIOS, DOS and third–party calls
(c) 1991 Addison Wesley Publishing Company, Inc.
ISBN 0–201–57797–6

[Dun89] Ray Duncan: Programmierleitfaden für MS–DOS Funktionen
übersetzt von Peter Riswick
Redmond, Washington: Microsoft Press;
(c) 1989 Vieweg, Braunschweig
ISBN 3–528–04650–3

[Dun90] Ray Duncan: DOS ohne Schranken
Orig.: Extending DOS, Programming MS–DOS for the 1990s
(c) 1990 Addison–Wesley (Deutschland) GmbH
ISBN 3–89319–300–6

[Dun90a] Ray Duncan: Programmierhandbuch MS–DOS
Orig.: Programming in the MS–DOS Environment
2nd Section of the MS–DOS Encyclopedia
(c) 1990 Friedr. Vieweg & Sohn Verlagsgesellschaft mbH Braunschweig
ISBN 3–528–04631–7

[Els88] Jürgen Elsing
MS–DOS Assembler Programmierung: praktische Anwendungen
von DOS–Aufrufen in Maschinensprache
(c) 1988 IWT Verlag GmbH, Vaterstetten bei München
ISBN 3–88322–198–8

[Fer91] Richard F. Ferraro
Programmer's Guide to the EGA and VGA Cards
Second Edition, March 1991
(c) 1990 by Richard F. Ferraro
Addison–Wesley
ISBN 0–201–57025–4

[Mue90] John Mueller and Wallace Wang
Microsoft Macro Assembler 5.1
Programming in the 80386 Environment
(c) 1990 Windcrest books, Division of TAB BOOKS Inc.
ISBN 0–8306–3179–8

[NS83] NS16000 Instruction Set Reference Manual
(c) 1983 National Semiconductor Corporation
California
Order No. 420010099–001

[Not92] Semesterarbeit
Hintergrundbitmaps für DOS–Oberon
Thomas Notter, Sommersemester 1992

[Ott92] Semesterarbeit
VGA Rasteroperationen für DOS–Oberon,
Hans–Werner Ott, Sommersemester 1992

[Pet91] Semesterarbeit
i386–Assembler für Oberon,
Harald E. Peter, Wintersemester 1991/92

[Pho89] Phoenix Technical Reference Series
System BIOS for IBM PC/XT/AT Computers and Compatibels
(c) 1989 Phoenix Technologies, Ltd.
ISBN 0–201–51806–6

[Smi87] Bud E. Smith, Mark T. Johnson: Programming the Intel 80386
(c) 1987 Scott, Foresman and Company, Glenview, Illinois
ISBN 0–673–18568–0

[S3 92] 86C801/86C805 GUI Accelerator
August 1992
S3 Incorporated
2880 San Tomas Expressway
Santa Clara, CA 95051–0981

[Wir92] Niklaus Wirth, Jürg Gutknecht: Project Oberon
The Design of an Operating System and Compiler
(c) 1992 ACM Press
ISBN 0–201–54428–8

Acknowledgements

A lot of people were involved in this project, either by testing the system and reporting problems or by giving useful hints.

First of all, I would like to thank J. Gutknecht for his guidance and suggestion during the whole project and for proof–reading this report.

Further, my thanks go to:
- N. Wirth for providing the source code of the compiler.
- J. Templ for providing the source code of the file system of SPARC–Oberon.
- R. Crelier for providing the source code of the Kernel and the floating point emulator of DEC–Oberon.
- M. Brandis for all the useful hints during the implementation of the extender.
- H. Peter for the assembler (semester work)
- H.W. Ott for the color display (semester work)
- T. Notter for the bitmaps (semester work)
- K. Rege for his help during the implementation of the DOS–Oberon file system.
- R. Sommerer for his valuable suggestions during the whole project.
- U. Hiestand for the LEDA editor with which this report was written.

Zurich, October 1993

Gelbe Berichte des Departements Informatik

183	W.B. Teeuw, Ch. Rich, M.H. Scholl, H.M. Blanken	An Evaluation of Physical Disk I/Os for Complex Object Processing
184	L. Knecht, G.H. Gonnet	Alignment of Nucleotide with Peptide Sequences
185	T. Roos, P. Widmayer	Computing the Minimum of the k-Level of an Arrangement with Applications
186	E. Margulis	Using *nP*-based Analysis in Information Retrieval
187	D. Gruntz	Limit Computation in Computer Algebra
188	S. Mentzer	Analyse von Methoden und Werkzeugen zur Entwicklung grosser Datenbank-Anwendungs-Systeme
189	S.J. Leon	Maximizing Bilinear Forms Subject to Linear Constraints
190	H.-J. Schek, G. Weikum, H. Ye	Towards a Unified Theory of Concurrency and Recovery
191	M. Böhlen, R. Marti	A Temporal Extension of the Deductive Database System ProQuel
192	R.H. Güting	Second-order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization
193	M.H. Scholl, Ch. Laasch, Ch. Rich, H.-J. Schek, M. Tresch	The COCOON Object Model
194	H.P. Frei, D. Stieger	A Semantic Link Model for Hypertext Retrieval
195	C. Laasch, Ch. Rich, H.-J. Schek, M.H. Scholl, S. Dessloch, T. Härder, F.-J. Leick, N.M. Mattos	COCOON and KRISYS - A Survey and Comparison
196	R. Gross, R. Marti	Intensional Answers in Generalized Deductive Databases
197	K.M. Chandy, B.A. Sanders	Conjunctive Predicate Transformers for Reasoning about Concurrent Computation
198	N. Wirth, S. Ludwig	An Extension-Board with an FPGA for Experimental Circuit Design CL - An Editor for the CLi6000 Field Programmable Gate Array and its Implementation CL-Editor User Manual
199	R. Vingralek, Y. Breitbart, H.-J. Schek, G. Weikum	Concurrency Control Protocols Guaranteeing Atomicity and Serializability
200	M. Bronstein, M. Petkovšek	On Ore Rings, Linear Operators and Factorisation
201	U. Maurer	Fast Generation of Prime Numbers and Secure Public-Key Cryptographic Parameters
202	B. Hösli	Robust Logic and Structural Properties of the Sequent Calculus