Dieter R. Pawelczak

Protected Mode Programming with Pass32

The Pass32 User Manual & Reference Guide

Dieter R. Pawelczak

Fasanenweg 41,
D-85540 Haar,
Germany

For my dear wife Alexandra

About

I started with Pass32 in late 1995, simple because there was no good tool to create protected mode applications for my Pro32 Dos Extender. Of course, there were many other assemblers, but there was none, that could generate a complete application in one run.

Pass32 should be similar to Turbo Pascal, and it should do all in one pass - that’s were the name is from. Pass32 has grown from its early time and is now a tool, that not only allows to generate protected mode applications with Pro32 - today, it can assembly dos extenders by its own. Since Version 2.5, Pass32 supports as well special registers like CR0 and can compile different code segment attributes at once: USE16 and USE32.

The features of Pass32 in brief:

- Pass32 supports modular programming
- Pass32 can generate DOS TINY and 32 bit protected mode applications
- The Dos Extender is directly linked to the program binary
- Pass32 combines assember, debugger and linker
- Pass32 can integrate a debugger in your application
- The assembler directives are simple and powerful
- Macros can create very complex code with an easy syntax
- Pass32 is fast, it compiles about 1000 lines on a DX4-100
- Pass32 is a real mode programm, nevertheless the source code size is not limited in size, the only limit is the number of symbols, procedures and macros
- Pass32 comes with a large run-time-library for standard I/O, file handling, joystick and graphic functions
- It includes a graphic library for Vesa1.2 and Vesa2.0 graphic adaptors.

Contents

Introduction

Welcome to the world of PASS32 Assembler. PASS32 was created for easy protected mode programming. The idea for the assembler was on the one hand to create a programming tool for the Pro32 Dos Extender. On the other hand, I wanted to create my own assembler. I don't like complicated linking and code with public or extern variables. I am used to write code straight forward and that's the idea of this assembler. The assembler does not create object code, but standalone executable code. You can, of course, link binaries into the code or include from other source files.

The assembler is a combination of Assembler and Linker. For protected mode programming the Dos Extender Pro32 is linked to the program.

The Assembler supports 5 memory models:

- TINY for regular 16 bit DOS .COM files
- FLAT for 32 bit protected mode files (fully compatible with Pro32)
- DLL for a PASS32 Version of 32 bit Dynamic Link Library
- OVL for a PASS32 Version of 32 bit overlay code/driver code
- WIN32 for Win32 applications[1]
  - CONSOLE for Win32 console applications
  - GUI for standard Win32 GUI applications
  - DLL for Win32 dynamic link libraries

For a more comfortable program development the assembler has a build-in debug function, which allows to set break points, to trace through the code and to view the source code.

This book explains in general the methods of protected mode programming. It shall not replace a good assembler tutorial, although it provides enough information for beginners to write their own assembler programs. The book concentrates mainly on protected mode programming and touches operating system programming. The first chapter provides an introduction in assembler programming and a set of rules for beginners. Chapter 2 explains protected mode basics. In chapter 3 the Dos Protected Mode Interface (DPMI) and the programming with Dos Extenders is explained. Chapter 4 concentrates of FPU, co-processor programming. In Chapter 5, the book deals with DLL programming. Chapter 6 describes enhanced assembler programming methods (macros, types). Chapter 7 introduces direct hardware access in protected mode. Chapter 8 provides a summary of all Pass32 directives.

The disk attached to the book contains the complete Pass32 developper environment, including the Pro32 debugger. Uncompressed, the files are stored in several subdirectories:

1.The Win32 PE-Format is supported in an alpha Version

- `\BIN`: binaries like **`PASS32.EXE`**, **`PRO32.EXE`**, **`PROSET.EXE`**
- `\INC`: include files, assembler modules for demo files
- `\DOC`: documentation **`PASS32.DOC`**, **`PASS32.TXT`**, **`PRO32.DOC`** etc
- `\DEMO`: Pass32 demo files
- `\EXAMPLES`: Example files introduces in this book
- `\DISS32`: source files for the 32 bit disassembler
- `\GRAPH`: example files for VESA graphics driver
- `\PRO`: example files of the Pro32 Dos Extender
- `\WIN32`: source files for the win32 programming

You should add the `\BIN` directory to your path!

Assembler instructions and Pass32 directives are marked with courier fonts like `.DATA`, `.CODE`, etc. Arguments for the Assembler and file names are marked with bold courier fonts like **`Pass32 demo -a -t`**

1. First Steps In Assembler

1.1 First Example

I want to start as all assembler, C or Pascal manuals start - with a simple *Hello-World* example. This will be the only real mode example, but I think you can create any real mode program, if you understand this example: (**`Hello1.ASM`**)

```
.MODEL TINY
.DATA
  HelloMesg db 'Hello,World',10 ,13,'$'
.CODE
START:
  mov dx,OFFSET HelloMesg ; offset of the text string
  mov ah,9                ; print string function number
  int 21h                 ; Dos call
  mov ah,4ch              ; terminate function number
  int 21h                 ; Dos call
END START ; marks the en try procedure of the program
END
```

Let us go line by line again through the code and let me explain what is behind these commands. Note, that all directives and assembler instructions are case-insensitive.

```
.MODEL TINY
```

This is the model definition. Pass32 supports five model types: `TINY`, `FLAT`, `DLL`, `OVL`, `WIN32` and three `WIN32` model sub-types: `GUI`, `CONSOLE` and `DLL`. The model definition therefore defines, for what target system the application is compiled, e.g. DOS (`TINY`) or Windows (`WIN32`). Additionally, the model implicitly defines, if the target is compiled as a real or protected mode application: All models are compiled for 32 bit protected mode, except the `TINY` model, which generates a 16 bit real mode application.

```
.DATA
  HelloMesg DB'Hello,World',10, 13,'$'
```

The directive `.DATA` marks the beginning of the data segment. Note, that almost all assembler directives can only be used inside a segment. Inside the `.DATA` segment, we can define data, that is initialized, i.e., that uses pre-defined values. In our example, we define a simple text string. The directive `DB` defines a data storages, that allocate one byte; `DB` stands for ***D**ata **B**yte*. The numbers following the text string define LineFeed and CarriageReturn to place the cursor in the next line. The '$' character is the old DOS string termination symbol. The DOS print function prints the string character by character until it reaches a '$' character.

```
.CODE
```

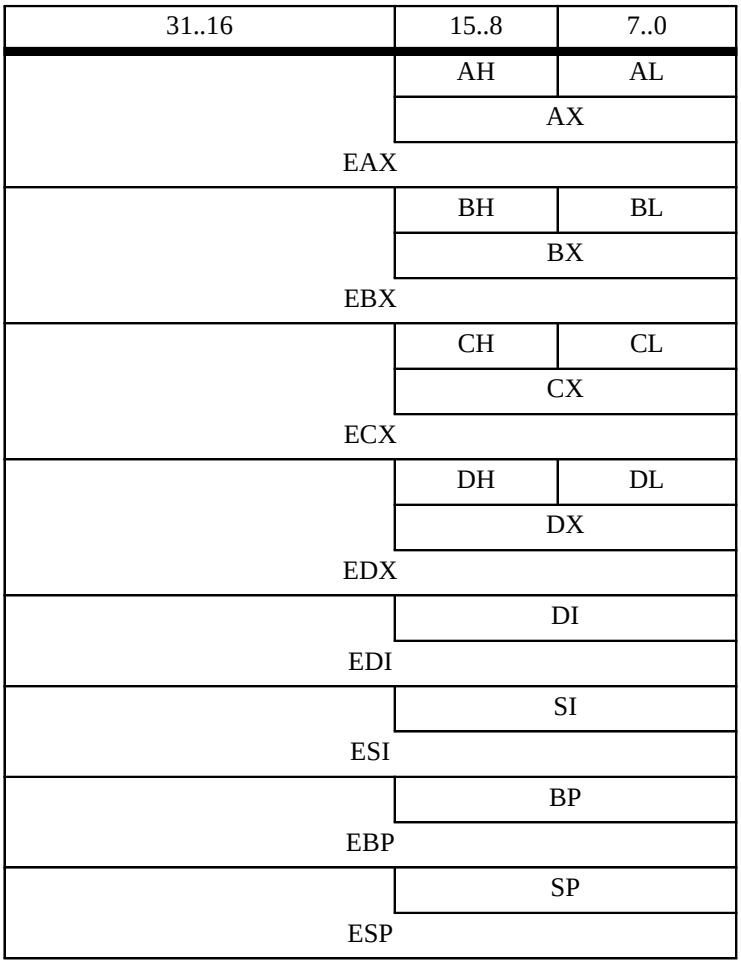
The `.CODE` directive marks the beginning of the code segment. The code segment contains all assembler instructions, that may be executed during the program execution. Although you can define data constants inside the code segment as well, you should separate data and code definitions as it makes your code easier to read and maintain.

```
START:
```

This is a label definition. A label does not produce code. It marks a point inside your code, that can be addressed through its label name. We named this label START, as it marks the beginning of our program. Note, that labels can have any name. The name START does not refer directly to the program's entry point! A label is defined by a name and a colon. In Pass32 names are case-insensitive, they are at least one and maximal 127 characters long. A label can consist of the following characters: 0..9, a..z, A..Z, '_', '#', '@', '.'.

```
mov dx, OFFSET HelloMesg ; offset of the text string
```

This is the first assembler instruction in our example. It loads the register `DX` with the offset address of the text string `HelloMesg`. In high-level programming languages, we work with variables. We can define variables in Assembler as well, e.g. the text string `HelloMesg` is a string variable. The ***C**entral **P**rocessing **U**nit* (CPU) of our PC can not directly work with variables. All calculations are done with internal data storages called the processor registers. In the case of an i486 or Pentium II processor, the processor provides 8 general 32-bit registers:



Tab. 1.1 *The processor's registers*

These registers are separated in one 16-bit and two 8-bit registers, but be careful, these subsections refer to the same register! The registers can be used for calculations and data processing. The `ESP` register is a special register, as it refers to the stack. The stack is a pile of numbers, which are always placed on top and taken out from top again. The stack is used, when the processor should store something for a short period. When something is pushed on the stack (`push` is the actual processor instruction, e.g. `push eax`), `esp` is decremented by 4 and the value is stored at the address, that `esp` points to. If something is popped from the stack (`pop` is the actual processor instruction, e.g. `pop eax`), the processor first reads the value at the address `esp` points to and then increments `esp` by 4.

The processor has other special registers: The flag register and the program counter. The flag register stores the status of the last assembler instruction, for example, if a comparisson was equal, if a substraction produced a number below zero, if an addition overflew the 32 bit number range, etc. The processor offers special instructions, which check these flags. These instructions are called conditional branches, as they change the program flow according to the status of the flags. The program counter, `EIP` (***E**xtended **I**nstruction **C**ounter*) is similar to the

stack register: The value at the address `EIP` currently points to, is read; the instruction is analyzed and after the execution of the instruction, `EIP` is incremented, so that it now points to the next instruction. We can not load or store `EIP` directly. The processor offers call, jump and return instructions, which will modify `EIP`.

As the x86 processor family is a CISC (***C**omplex **I**nstruction **S**et **C**PU*) processor family, the CPUs provide a complex instruction set: all general registers (`AL..DH`, `AX..SP`, `EAX..ESP`) can be added, multiplied, shifted, compared subtracted and divided. A typical assembler instruction has the following layout:

Action - command	Target Register	Operand
e.g. `mov, add, sub, imul, idiv, cmp`	e.g. `eax, bx, cl, dh, si, esi, ebp, sp`	e.g. `100, offset HelloMesg`

We now understand what `mov dx, offset HelloMesg` does: it simply loads a value into the `DX` register.

```
mov ah,9 ; print string function number
int 21h ; Dos c all
```

I skipped one line, and come to the `int` instruction. This instruction invokes an interrupt. The x86 CPUs provide 256 interrupts and `int 21h` invokes the interrupt number `21h`. The suffix 'h' stands for the hexadecimal notation. Assembler programmers typically write numbers in the hex-notation. There is a simple reason: A processor can only handle bits. A 32 bit number is presented internally by 32 bits, i.e. 32 D-flip flops with possible states '0' and '1'. In many cases, assembler programs need to modify specific bits. When we describe every 32 bit number in bits, we had to write a lot of zeros and ones, which wouldn't be much effective.

There is a more elegant solution: the hex notation. A hex digit alway presents 4 bits. As 4 bits allow 16 permutations, we need 16 symbols to present four bits. Theses symbols are 0..9 and A..F. I'm afraid, you can't count these with your fingers, but it is a really nice method, because an 8 bit number - one byte - is presented by to hex digits: 00-FF, a 16 bit number by 4 hex digits, a 32 bit number by 8 hex digits and so on. You can tell in a glance from a hex digit, which bits are set and which are not. You can't do that with decimal numbers - simple example:

binary	0011	1101	1001	0100	0110	0010	0101	1100
hexadecimal	3	d	9	4	6	2	5	c
decimal	1,033,134,684							

Tab. 1.2 *Hexadecimal notation*

As hexadecimal numbers can start with a letter, we have to distinguish them from lables or variable identifiers. For this reason, any number must start with a digit 0..9. The hexadecimal value A0 therefore must be written as 0A0H. Pass32 supports other notations as well:

Suffix	Notation	Examples
b	binary	0101100b, 1001000110001101b
- (default)	decimal	1024, 65535, 1103847511
h	hexadecimal	0efh, 16h, 0affeh
o	octal[a]	077, 123, 77551

a. The base for this notation is 8, an octal number consists of digits between 0..7

Tab. 1.3 *Number Notation with Pass32*

Let us come back to the `int` instruction. There are basically two kind of interrupts: SW and HW interrupts. Software interrupts are called via the `int` instruction. The processor fetches the address for the interrupt service routine from an internal table, called the ***I**nterrupt **D**escriptor **T**able* (IDT). The processor executes the interrupt and continues the execution of the main program, after the interrupt has finished. HW interrupts are processed identically, the only difference is the origin of the interrupt: a HW interrupt is invoked by a HW event, e.g. a key stroke, an overflow of the internal timer, etc. The x86 processors do not distinguish between these interrupts. So it may be confusing, that `int 21h` invokes a DOS function and `int 9h` invokes the keyboard handler.

All functions of the DOS operating system and the DPMI (***D**os **P**rotected **M**ode **I**nterface*) are called via interrupts. To execute a DOS functions, you load the function number in `AH` and execute `int 21h`. Additional parameters may be passed via other registers. Tab. 1.4 shows some standard dos functions.

Function	Parameter	Description
`AH`=2	`DL`=character	Print character
`AH`=7	`AL`=character	Wait for keystroke, result in AL
`AH`=9	`DX`=offset	Print string, offset to String in DX, string must end with the '$' Symbol.
`AH`=4ch	`AL`=return code	Terminate the program.

Tab. 1.4 *Some DOS functions*

```
mov ah,4ch ; terminate function number
int 21h ; Dos call
```

We already learned about the `int 21h` functions. Note, that we always have to terminate a program correctly. Under DOS/DPMI we must use the Function 4Ch of `int 21h`. Under Win32, we must use the Kernel Function `ExitProcess`.

```
END START ; marks the entry procedure of the program
END
```

The directive END defines the program entry point. We use that directive in combination with a label or a procedure. In our case, the program will be executed from the label START. The directive END also defines the end of the assembler source.

We assemble the program with **PASS32 HELLO1 -t**. The **-t** option is used in combination with the TINY model. Actually the **-t** option is a linker option and tells PASS32 to create a **.COM** file. The output of the assembler tells you about a correct assembling:

```
Pass32-Assembler (c) 1996 by Dieter Pawelczak
Assembling:HELLO1.ASM
Pass: 1
Pass: 2
Linking
Total Source Lines:  13 Total Code Bytes :  12
Total Data Bytes :   14 Total Bytes :       26
Total instructions:   5 Total Time :        0.27
Output File :HELLO1.COM
```

If you run **Hello1** you will get the output „Hello,World“. Although it was a very tiny program with roughly more than ten lines, we already learned a lot about assembler programming. I want to show you in advanced, how the same example could look in protected mode (**HELLO2.ASM**):

```
.MODEL FLAT
.DATA
HelloMesg db 'H ello,World',0
.CODE
  mov edi,OFFSET HelloMesg ; offset of the text string
  call systemwriteLn ;  call a protected mode library function
  mov ah,4ch ; DPMI terminate function
  int 21h ; call DPMI  function
.include system.inc ; include SYSTEM.INC (contains systemwriteLn)
END
```

You assemble the demo with **PASS32 HELLO2**. If you run **Hello2**, you will get the same output, but now from protected mode!

Let's take a closer look at that example: The first line .MODEL FLAT defines the FLAT memory model. This is the standard memory model for any 32 bit protected mode program: Data and Code are in the same segment. This segment can be up to 4GByte in size. The Pass32 assembler automatically links the Pro32 Dos Extender to the program, when you choose the FLAT model.

As like in real mode, we define our text string after the `.DATA` directive. With the `.CODE` directive, we tell the assembler, that the folowing commands/instructions refer to code segment. All instructions are compiled as 32 bit instructions due to the `FLAT` model. This is very important, as the attempt to run 16 bit code in 32 bit protected mode will lead to an exception, moreover, if you run 32 bit code under real mode, the processor stops execution and usually resets.

We don't use an interrupt function to print the message, as Pass32 provides a lot of library functions. And we could not use the `int 21h` instruction directly to print the string, unless we enable the extended dos support: DOS is a 16 bit operating system and can not access memory above 1MB - above the address 100000H. A 32 bit protected mode program usually runs above the 1MB barrier, as the extended memory starts at that address. So it would not make sense to force DOS to print a string, which is addresses above 100000H.

The `.INCLUDE` directive tells the assembler to include another assembler file at the current cursor position. The assembler file may contain data, functions, procedures and may even include again another module. Any module is only included once, too avoid duplicate data or code definitions. If you do not add an extension to your file name, the file name is extended with **`.ASM`**. The Assembler searches the following directories for the module:

- The current directory, from which Pass32 has been called
- The subdirectory **`INC`** of the Pass32 directory
- The parallel directory **`\INC`** of the Pass32 directory

Usually the Assembler is located in the **`\BIN`** directory, include files are located in the parallel directory **`\INC`**.

Pass32 is more than a simple Assembler. It allows intelligent linking and code optimization. Intelligent linking means, that only code is linked into the application, that is actually used. Code optimization means, that useless instructions are removed, e.g. mov eax,eax, and that some instructions are replaced by faster ones, e.g. `mov eax,0` by `xor eax,eax` or `add eax,00000010h` by `add eax,10h`. You should try the following assembler function: **`PASS32 HELLO2 -o.`** As we see the assembler uses a third pass to optimize the code.

As we can see from these examples, there's no big difference between the DOS **`.COM`** format and the Pro32 FLAT memory model. You could say, the Pro32 FLAT memory model is a huge **`.COM`** format, with 32 bit offsets instead of 16 bit. When you start writing protected mode programs, you should think of this model and you can't go wrong!

As I mentioned above, we can not use all DOS functions directly. We have to enable extended DOS support first. Pass32 offers the module **DOSX.INC**, which allows to execute extended DOS functions[1]. The third example is **HELLO3.ASM** - it uses the extended DOS function 9h to display the message:

```
.UCU .NM
.INCLUDE DOSX.INC ; include extended DOS library ...
.DATA
  mesg db 'Hello, World - with extended DOS!',13,10,'$'
.CODE
START:
  mov edx,OFFSET mesg ; offset to text string
  mov ah,9h ; extended  dos function string to standard output
  int 21h ; dos call
  mov ax,4c00h
  int 21h ; terminate
END START
END
```

A list of all extended DOS function can be found in the appendix. Please note, that extended dos functions always need to copy the operands into real mode memory before the execution and copy back into the extended memory area after the execution. This makes the execution much slower. A much faster way is the usage of zero selectors in PM, which can directly access the real mode area.

1.2 Beginner's Rules

The following section defines some programming rules for assembler programming. It should help beginners with coding in assembler or converting existing code into assembler. Some basic pitfalls are also described.

1.2.1 Assignment and expressions

An assignment is one of the basic machine instructions: the `mov` instruction:

1.Pro32 does not support extended DOS functions directly. Pro32 GOLD has an integrated plug-in to enable 32 bit DOS support. Some other DOS extenders have extended DOS functions already included. DOSX.INC is generic and DPMI compatible and works with other DOS extenders as well.

Pascal	C	Assmbler
A:=0;	A=0;	mov A,0
A:=A+1;	A++;	inc A
A:=A+B;	A=A+B;	add A,B

A:=B*6	A=B*6;	mov eax,B
		imul eax,6
		mov A,eax

A:=B*C+D	A=B*C+D	mov eax,B
		mov ebx,C
		imul ebx
		add eax,D
		mov A,eax

H:=X;X:=Y;
Y:=H;
	H=X;X=Y;Y=H;	mov eax,X
		xchg Y,eax
		mov X,eax

A:=B div
15;
	A=B/15;	xor edx,edx[a]
		mov eax,B
		mov ebx,15
		idiv ebx
		mov A,eax

a. The (integer) division always assumes a larger dividend as the divisor. Therefore, a idiv ebx instruction expects EDX:EAX (a 64 bit value) as dividend. A idiv bx instruction would require DX:AX (a 32 bit value) as dividens and an idiv bl instruction would take AX (a 16 bit value) as the dividend. Note, that the div/idiv instruction creates a division by zero exception, either if the divisior is zero, or the divident is zero, or the division result overflows the result register (EAX,AX,AL), e.g. if you divide DX:AX=100000 by 1, the result does not fit into AX! The division always provides two results: the division result and the remainder. The remainder is stored into EDX, DX or AH!

A common pitfall is the division. First it may effect two register, e.g. EDX, EAX. And then, the division result may not fit into the result register and cause a division by zero exception. Therefore, to be on the save side always use the following code for divisions.

```
PUSH EDX ; EDX is changes by the DIV instruction!
PUSH EBX ; EBX will take  the divisor value
XOR EDX, EDX ; not a 64  bit division!
MOV EBX, divisor
MOV EAX, dividend
DIV EBX
POP EBX
POP EDX
; result in EAX
```

Note, that an 8 bit division, e.g. div bl, will store the result in AL, the remainder is stored in AH. For an 8 bit division, use the following code:

```
PUSH EBX ; EBX will take the divisor value
MOV BL, divisor
XOR EAX,EAX
MOV AX, dividen d
DIV BL
POP EBX
XOR AH,AH
; result in EAX
```

A similar rule applies to the MUL instruction: The result of a 16 bit multiplication is 32 bit, the result of a 32 bit multiplication is 64 bit. Again we have to take care that DX, EDX are changed by the MUL instruction together with EAX, AX. The MUL instruction takes EAX as the first multiplicator and any other general register as the other multiplicator. The result is always stored in EAX:EDX (if the multiplicator was 32 bit, e.g. EBX, EAX), AX:DX (if the multplicator was 16 bit, e.g. BX, AX) or AX (if the multiplicator was 8 bit, e.g. BL, AL). Since the i386 we have an immediate IMUL instruction, i.e. you can multiply any general register with an 8 bit value directly, e.g. IMUL EBX, 10. This variant of the MUL/IMUL instruction does not affect the EDX register and the result is directly stored in EBX. For most multiplications it is easier to use this immediate IMUL instead of the MUL/IMUL instruction.

Fast multiplications and divisions can be gained by shifting the registers contents to the left or to the right. A shift to the right by n bits is an unsigned division by 2^n (n>0). A shift to the left by n bits is an unsigned multiplication by 2^n (n>0), e.g.:

```
shl eax,2 ; equals eax*4
shl ecx,5 ; equ als ecx*32
shr eax,1 ; equ als eax/2
```

1.2.2 IF, THEN, ELSE, CASE

These highlevel language structs are not directly supported. The use of conditional jump instruction easily tranfsers them into assembler:

Pascal	C	Assmbler

IF A=0 THEN
  A:=5
	if (A==0) A=5;	cmp A,0
		jne endif
		  mov A,5
		endif:

IF A=0 THEN
  A:=5 ELSE
  A:=A-1;
	if (A==0)
	  A=5; ELSE
	  A--;
		cmp A,0
		jne else
		  mov A,5
		  jmp endif
		else:
		  dec A
		endif:

IF
(A=0)AND(B=0)
  THEN C:=0;
	if ((A==0)&&(B==0))
	  C=0;
		cmp A,0
		jne endif
		cmp B,0
		jne endif
		mov C,0
		endif:

IF
(A=0)OR(B=0)
  THEN C:=0;
	if ((A==0)||(B=0))
	  C=0;
		cmp A,0
		je then
		cmp B,0
		je then
		jmp endif:

CASE A OF
0: B:=0;
1,2,3: B:=1;
4,5,6: B:=2;
ELSE
B:=3;
END
	switch(A)
	{
	case 0: B=0;
	break;
	case 1:
	case 2:
	case 3: B=1;
	break
	case 4:
	case 5:
	case 6: B=2;
	break
	default: B=3;
	break;
	}
		cmp A,0
		jne N0[a]
		mov B,0
		jmp NX
		N0: cmp A,3
		ja N3[b]
		mov B,1
		jmp NX
		N3: cmp A,6
		ja N6
		mov B,2
		jmp NX
		N6:
		mov B,3
		NX:[c]

a. JNE performs a jump incase the comparison was not equal. JE performs a jump incase the comparison was equal.
b. JA performs a jump incase the first operand is above the second operand. JB performs a jump in case the first operand is below the second operand.
c. Note, that the code for the switch instruction needs as many lines as the equivalent in C.

A source for errors are the relative jump instructions: Either the instruction is negated, or the signed and unsigned integer comparisons are mixed up. Take care, the simplest way to test, if a certain value is reached, is done by using the `JNE` instruction and putting the necessary actions between the `JNE` instruction and the target label. This is the most optimized translation of an IF instruction:

```
IF EAX=0 THEN do_action
```

refers to

```
cmp eax,0
jne not0
; EAX = 0
; do the action
not0:
```

This does not refer to an IF, ELSE construct - compare with the table above. A simple rule for labeling such constructs is to use labels like `N0`, 'Not 0'. If you need ELSE or a complete case, use `NX` as the exit label.

Hardware addresses are typical unsigned integer values. You should use `JA` (jump if Above), `JB` (jump if below) to determine differences. Numbers are typically signed integer values, i.e. you should use `JG` (jump if greater) or `JL` (jump if less), when you compare two values.

1.2.3 Loops

The processor provides directly a loop instruction. Unfortunately, the `loop` instruction provides only one construct, namely the repeat until construct. Due to those restrictions and due to the fact, that the loop construct is restricted to 127 bytes offset only, it is recommended not to use the `loop` instruction. It should be statet, that the `loop` instruction is slower than an equivalent assembler construct on a pentium processor.

Pascal	C	Assmbler
REPEAT ECX:=ECX-1; UNTIL ECX=0;	do { ECX--; } while(ECX!=0)	L0: loopd L0

REPEAT
A:=A+10;
UNTIL A>100;
	do
	{
	A=A+10;
	} while(A<=100)
		L0:
		ADD A,10
		CMP A,100
		JBE L0

WHILE (A<100)
do A:=A+10;
	while (A<100)
	{
	A=A+10;
	}
		L0: cmp A,100
		jae L1
		add A,10
		jmp L0
		L1:

FOR I:=0 TO 99
DO
A[i]:=0;
	for (I=0;I<100;I++)
	A[i]=0;
		mov I,0
		L0: cmp I,99
		ja L1
		mov eax,I
		mov [A+eax],0
		inc I
		jmp L0
		L1:

A typical pitfal is the loop instruction: Imagine the following code:

```
mov eax,0
mov ecx,1000
L1:
mov dword ptr Screen+4*ecx,0
loop L1
```

The result is, that the first 4 bytes of the `Screen` array are not initialized with 0, because the `loop` instruction repeats only until `ECX` is zero. The correct solution of the problem would be:

```
mov eax,0
mov ecx,1000
L1:
mov dword ptr Screen+4*ecx,0
dec ecx
jns L1
```

The loop is repeated until `ECX` reaches -1, i.e. the case `ECX=0` is also processed inside the loop. Note, that if the loop exceeds 64K, i.e. if `ECX` is above 65535, you shall use `loopd`. The `loop` menmonic refers to `CX` only.

2. Protected Mode Programming Basics

This Chapter introduces protected mode basics. It describes the differences between real mode and protected mode. In tiny steps all necessary action will be taken to switch the processor into 32 bit protected mode.

2.1 Protected Mode versus Real Mode

When the PC is switched on, the processor starts in real mode. In real mode, all CPUs of the x86 family, including latest 686 processors, are compatible with the obsolete 16 bit 8086/8088 CPU. The CPU can not address more than 1 MB. As the address range from 0A0000H to 0FFFFFH Is usually used by the BIOS, actually only 640KByte memory can be used for applications.

In protected mode, the processor can address the whole address space of the processor. This comprises up to 64TByte in combination with virtual memory management. Theoretically, each application can have a virtual 4G address space[1]. Additionally to the memory management and protection means, the CPU provides methods for multitasking under protected mode.

It should be noted at this point, that a lot of processor extensions, especially the protection means are also valid for real mode. Indeed, the processor creates an invalid opcode exception, when the CPU reads an instruction, that it can't interpret. Unfortunately, these exceptions are not handled by the real mode operating system, e.g. DOS. Therefore we assume, that the processor crashes in real mode in case of an illegal operation, but atcually, the processor invoks an exception handler, which is not provided by the operating system!

It is very easy to switch the processor into proteced mode. The processor uses one flag in an internal register, which defines the operating mode. This is the PE-Flag (***P**rotected mode **E**nable*: bit 0) in the CR0 register. To enable protected mode, the bit must be set to 1:

```
mov eax,cr0
bts eax,0 ; sets bit 0
mov cr0,eax
```

To switch back to real mode, the bit must be cleared:

```
mov eax,cr0
btr eax,0 ; resets bit 0
mov cr0,eax
```

Unfortunately, this bit does not initialize the protected mode. A lot more needs to be done before we can actually switch into protected mode:

- interrupts must be re-directed to protected mode service routines, or all interrupts must stay disabled,
- the ***I**nterrupt **D**escriptor **T**able* (IDT) must be created,

1.Under Win32, each application gets a virtual 4G address space

- the ***G**lobal **D**escriptor **T**able* (GDT) needs to be established,
- Code, Data and Stack descriptors must be defined.

2.2 Addressing in Real Mode

To understand the processor in real mode, we'll have a look on how the processor addresses memory in real mode. A real mode address is 20 bits long. The address is calculated from the segment register and the offset. The segment registers are `CS`, `DS`, `ES`, `FS`, `GS` and `SS`. `CS` stands for Code Segment. The processor automatically uses this segment register when reading an instruction from the instruction pointer. `DS` is used as the default data segment, i.e. any memory access is per default related to `DS`. Access to other segments needs a segment override, i.e. the segment register must be specified. `SS` is automatically used for stack access. So a `push` or `pop` instructions refers to `SS:ESP`. The segment registers are 16 bit registers in real and in protected mode. In real mode, there is no virtual addressing, therefore any linear address refers to the absolute physical address in memory. The address is calculated multiplying the segment register value by 16 and adding the offset. In hex notation, this may for example look like this:

```
                CS:  1004h
                IP:   0100h
  Physical Address:    10140h
```

As you can see, the multiplication by factor 16 is identically with shifting the segment register value 4 bits to the left. In binary notation, the same example would be:

```
                CS:  0001.0000.0000.0100
                IP:       .0000.0001.0000.0000
  Physical Address:    0001.0000.0001.0100.0000
```

So another way to explain the real mode address calculation would be to say, the segment register presents the upper 16 bit of the address, thus bit 19 to 4. The 16 bit offset address, is added to the bits 15 to 0. The result is a 20 bit address. A funny thing is, that an i386 actually has 32 address lines. What happens, if the address calculation exceeds the 20 bit? This happens, when the segment register plus the offset produces an overflow:

```
                CS:  1111.1000.0000.0000
                IP:       .1000.0000.0000.0000
  Physical Address:    0000.0000.0000.0000.0000    on a real 8086
  Physical Address:  1.0000.0000.0000.0000.0000    on a 80286f
```

The 80286 and new processors calculate the address correct, i.e. bit 20 is set. Therefore we can access actually 1 MB plus 65520 bytes. This memory area from 010000h to 01FFEF is called ***H**igh **M**emory **A**rea* (HMA). DOS can load drivers to this memory. Note, that this function of the CPU is disabled per default by most mother boards. The address line 20 is forced to low level, to simulate the behaviour of the original 8086 processor. Typically, the XMS (*e**X**tended **M**emory **S**pecification*) memory driver, e.g. **HIMEM.SY**S, controls the A20 line. A correct 32 bit address calculation needs an enabled A20 address line, because otherwise an address 010000h is reflected to 000000h!

2.3 Protected Mode Address Calculation

In protected mode, the address calculation is completely different. The 80286 provided a 16 bit protected mode. For reasons of compatibility, the 80386 and newer processors support this addressing as well. The 16 bit protected mode provides as like the real mode only 64K segments. In general, we concentrate on 32 bit programming. Nevertheless, for the mode switch, we still need this 16 bit mode. In protected mode, our segment registers have a complete different meaning: They hold a 16 bit value, an index to a segment descriptor. As the value itself has nothing in common with the segment address, the registers CS, DS, ES, FS, GS and SS are called selector registers - they select a segment descriptor out of a list. In protected mode, there are two tables, which hold the segment descriptors: The global and the local decsriptor table (GDT, LDT). The GDT holds the system descriptors, the LDT application related descriptors. The selector registers now contain an index to the descriptor and a flag, which tells the CPU from which table the descriptor is used.

The selector has the following format:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Index													0	0	G[a]

a. G=0: GDT, G=1: LDT

Tab. 2.1 *Selector Contents*

The following example explains the protected mode address calculation in an abstract way:

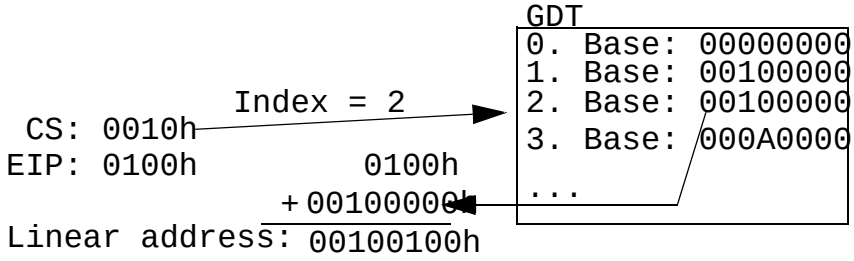


Fig. 2.1

The CPU knows from bit 0, that the selector is part of the global descriptor table. It reads the 32 bit base address from the corresponding entry in the GDT and adds the 32 bit offset. The result is a 32 bit linear address. Note, that the linear address of the CPU does not need to be identical with the physical address. The CPU offers another step in the address calculation, before the actual address lines are accessed: the address management divides the whole memory in 4K pages. A nested list of these pages define the actual address in memory. These 4K pages can be swapped to disk. Therefore, the CPU can offer more memory, that actually is available. In our examples, we ignore this step and assume, that the linear address is equal to the physical address[1].

2.4 Descriptor and Global Descriptor Table

The descriptor holds information about the memory segment. As we learned in the previous chapter, the 32 bit base address is defined in the descriptor. The name protected mode already implies memory protection: The descriptor defines also, the size of the memory segment, if the memory is read only, if it is a data or a code segment. The contents of a descriptor are shown in Tab. 2.2:

Bits / offset	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Descriptor Limit 15..0															
2	Descriptor Base 15..0															
4	1	DPL		1	CD	EC	RW	A	Descriptor Base 23..16							
6	Descriptor Base 31..24								G	BD	0	AVL	Limit 19..16?			

Tab. 2.2 *Descriptor contents*

G: Limit Granularity: 0: byte granular (Limit=Limit field)
1: page granular (Limit=4096*Limit field)

B/D: Segment Attribute Size: 0: 16 bit = use16
1: 32 bit = use32 (default)

AVL: Available Flag: (unused)

DPL: Privileg Level 00: kernel level
01: device driver level

1.Note, that this assumption can lead to a fatal error, when hardware is related: Assuming, you provide a disk buffer at the virtual address 00100000h. The disk drive is going to write the contents of this buffer to your harddisk. Unfortunately, your OS provides virtual address management. Your virtual address 00100000h is actually mapped to 00010000h and the physical memory 00100000h is not even present. The result is, that the disk drive writes garbage to disk and it takes very long to find this bug!

	10: operating system level
	11: user application level
C/D: Segment Type	0: data
	1: code
E/C: Expand/Conforming	0: data=expand-up code=non-conforming (default)
	1: data=expand-downcode=conforming
R/W: Read / Write	0: data=readcode=non-readable
	1: data=read/writecode=readable
A: Access	0: not accessed,
	1: accessed

The descriptor limit takes only 20 bit, which refers to an address range of 16 MByte. The granularity flag (G) defines, whether the limit value is shifted again 12 bit. The resulting address rangs is 4 GByte. The idea behind is, that when virtual memory managment is used, the whole memory is divided into 4 K pages. It wouldn't make any sense to allocate 4.5 K memory for an descriptor, as the memory management would become very ineffective. Therefore in combination with virtual address management, it is very useful to allocate memory only in 4 K steps.

The descriptor base takes the complete 32 bit and defines the virtual basis address of the segment. Note, that if no virtual address managment is used, the virtual basis address is equal to the physical basis address.

We have two flag fields. The first flag field at offset 5 holds standard flags for protection means. These flags describe, if the segment is a code or data segment (C/D). A data segment can either be read and writeable, a code segment can only be readable or not readable (R/W). The expand flag defines, whether the segment is expanding upwards (typically heap memory, which starts at a fixed address and dyanimcally grows) or downwards (typically stack memory) (E/C). DPL defines the protection level. DPL=00 is the highest privileged level. It is comparable with real mode, as a code segment of this privileg level can access every thing: changing the IDT, GDT, switching back from protected mode to real mode, disabling interrupts, etc. In lower privileg levels, instructions, which endanger the system stability invoke an general protection fault exception[1].

The second flag field is not available in 16 bit protected mode. It provides one essential flag for 32 bit programming: The Big or USE32 flag (B/D). If it is set to 1, a code segment handled as a 32 bit code segment, e.g. the processor reads per default 32 bit instructions[2], a data segment can exceed 64K. The second flag also holds the granularity bit, that allows to access the whole 4G address space.

1.Although the processor provides clear protection means, no commercial OS is really based on them.

2.A 16- and 32-bit instruction is distinguished by the register and address prefix 66h/67h. This prefix is used, when the registers or address modes differ from the default segment type: If ecx is used in a USE16 segment, the register prefix 66h is used. If the register cx is used in a USE32 segment, the register prefix 66h is used. If a USE16 segment holds a 32 bit addressing, e.g. [edi], the address prefix 67h is used. If a USE32 segment holds a 16 bit addressing, e.g. [di], the address prefix 67h is used as well.

The GDT register of the CPU is a six byte large buffer, that contains two values: A 16 bit value holding the number of maximum entries and the 32 bit linear base address. Note, that the index field in the selector has only 12 bits. The maximum number of descriptors is therefore 4096. The definition of the GDT could look like this:

```
.CONST
.ALIGN 8
  MAX_GDT_ENTRIES .EQU 32
.BLOCK
  GDTRECORD DW  MAX_GDT_ENTRIES*8
            DD  OFFSET GDT
.NOBLOCK
  GDT       DD  256 DUP(0)
```

The following sample function will create and store descriptors in the global descriptor table:

```
PROC Create_Descriptor ; EAX:Basis; EDX:Limit; ECX:Access_Rights
  push eax
  push ecx
; Test the limi t, if G-bit has to be set
  push edx
  mov eax,edx
  shr eax,24
  cmp eax,0
  je short L0
; Shift limits 12 bits and set G-bit...
  pop edx
  pop ecx
  bts ecx,15   ; set  G-bit
  bts ecx,14   ; set  D-bit
  push ecx
  shr edx,12   ; shif t limit
  push edx
L0:
; The GDT is predefined with zero contents.
; We search for  the first free table entry...
  mov bx,offset GDT
  xor esi,esi
L1:
  add si,8             ; first descriptor is per default zero
  cmp si, MAX_GDT_ENTRIES*8
  jae short X           ; GDT full
  mov eax,[bx+si]
  cmp eax,0
  jne short L1
```

```
    mov eax,[bx+si+4]
    cmp eax,0
    jne short L1
; Got free entry...
    add bx,si
    pop edx
    pop ecx
    pop eax
    mov [bx],dx ; limits - lower  16 bits
    mov [bx+2],ax ; basis - lowe r 16 bits
    mov [bx+5],cl ; Flags, lower  8 bits
    shr eax,16
    mov [bx+4],al ; Basis Bits 1 6..23
    mov [bx+7],ah ; Basis Bits 24..31
    shr edx,16
    and dl,0f0h
    or ch,dl
    mov [bx+6],ch ; Flags Bits 8 ..15
    xor eax,eax
    mov ax,si     ; Selextor in   SI - return in AX
    ret
X:                  ;     Error!
    xor eax,eax   ; return Selec tor 0
    stc
    ret
ENDP Create_descriptor
```

Note, that the first entry in the GDT is the so-called NULL descriptor. It has the selctor value 0. A selector register can be load with this selector. If you read or write using this selector, a general protection fault exception will be invoked.

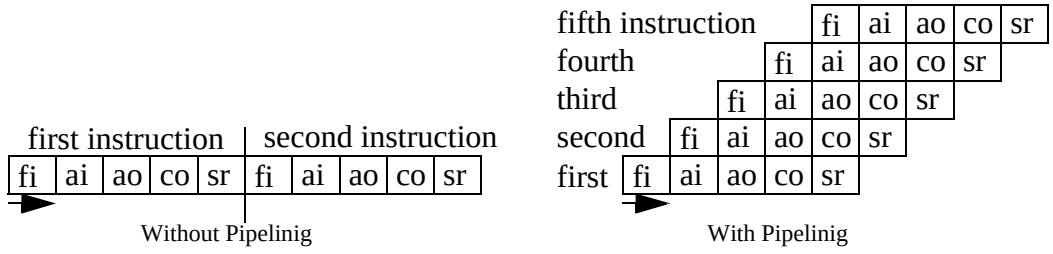
The following code will create all necessary descriptors for a simply protected mode application:

```
    xor eax,eax
    mov ax,cs      ; Code Segme  nt
    shl eax,4       ; Basis = CS shifted by bits to the left
    mov edx,0ffffh ; Limit = 64K
    mov cx,0009ah  ; 16 bit code  segment
    call Create_Descriptor
    mov SelCSeg,ax ; store selec tor value
    xor eax,eax
    mov ax,ss      ; Stack Segm  ent
    mov Real_ss,ax ; store real mode stack value
    shl eax,4
    mov edx,0ffffh
```

```
mov cx,00092h  ; 16 bit stack segment
call Create_Descriptor
mov SelSSeg,ax
mov ax,ds      ; D  ata Segment
shl eax,4      ; B  asis = same as DS
mov edx,-1     ; Li  mit = 4G
mov cx,00092h  ;  3 2 bit Data segment
call Create_Descriptor
mov SelDSeg,ax
```

Note, that we need a 16 bit code segment for the mode switch: when the processor has switched into protected mode, the USE bit of the code segment does not change, therefore it assumes a 16 bit code segment. The same happens, when the processor switches from a 32 bit code segment into real mode: The processor would again assume a 32 bit(!) real mode segment, which is not a valid processor configuration. Therefore, we will need a 16 bit code segment, which holds the necessary code for the mode switch from and to protected mode.

The i386 and newer processors use CPU pipelines to speed up the execution of commands. The idea behind a pipeline is, that the CPU can do several (for every instruction) necessary steps in parallel: it can for instance fetch a new instruction from memory and perform an arithmetic operation in the same time. A sample pipeline could consist for instance out of five steps: fetch instruction (fi), analyse instruction (ai), arithmetic operation (ao), calculate offset (co), store result (sr). The following figure demonstrated such a pipeline. It is easy to understand, that such a pipeline would increase the CPU speed (in the optimum) by 5, as with every clock a new instruction starts and another instruction ends:



An i386 processor uses pipelinig. Imagine, what happens, when one of the instructions inside the pipeline switches the CPU mode. This will happen after the processor has analysed the instruction, therefore in stage 2 or 3. The rest of the pipeline is already filled with the new instructions, so the processor has to treat some instructions different in the pipeline: perform the last operations of real mode instructions and already fetch and execute protected mode instructions. Note, that newer processors use a much deeper pipeline with 16 and more stages. As the processor can only perfrom either real mode or protected mode instructions, we have to flush the pipeline, when we switch to protected mode. This is done by a jump instruction. As all instructions, which follow a jump instruction have to be discarded in the pipeline.

As we learned, that switching the PE bit does not initialize any descriptor, we even use a FAR jump instruction. This FAR jump instruction will read a new CS value and flush the pipeline. Note, that prior to the FAR jump instruction, the processor still uses the old (real mode) CS segment register to fetch the instructions. This is a somehow funny behaviour, because the CPU uses a real mode segment register in protected mode! This is due to the fact, that the mode switch itself does not change any segement / selector register values. Additionally, the processor does not actually use the segment or selector values directly: when a segment / selector register is load, the processor uses internal (hidden) registers, which are similar to descriptors: these registers define, where the segment starts, who is allowed to read / write, if its 32 or 16 bit, etc. These hidden registers are untouched by the CPU during the mode switch and changed only, when the selector register is load with a new value. Some older dos extenders, e.g. 16 bit dos extenders do not reload all segment registers, when they switch back to real mode, e.g. FS, GS. Another application accessing these (protected mode selectors) in real mode, will then cause an exception, which typically results in a crash of the real mode application.

A simple switch into protected mode could look like this:

```
    xor eax,eax
    mov ax, cs
    shl eax, 4
    add dword ptr [GDTRecord+2],  EAX ; set linear address of GDT
    mov eax,cr0
    bts   EAX, 0                      ; test and set BIT 0
    jc X                               ; Error - already in PMode!
    cli                               ; No IDT, therefore disable interrupts!
    lgdt GDTRECORD
    mov cr0,eax                       ; activate PMode
.CONST
    db       0EAh                      ; FAR-JMP, to flush CPU-Pipeline and
    dw       Offset PM                 ; to load CS
    SelCSEG dw 0
.CODE

PM:
    mov ax,cs:Sel_SSeg ; load stack selector
    mov ss,ax
    mov ax,cs:Sel_DSeg ; load Da ta selector
    mov ds,ax
```

Have a look at the example files **PMODE1.ASM** and **PMODE2.ASM**. The first example switches into 16 bit protected mode, the second example switches into 32 bit protected mode.

2.5 Protected Mode Interrupts and Exceptions

2.5.1 Interrupts

In real mode, the interrupt descriptor table IDT is found at the first 1024 bytes in memory, i.e. at offset 0000000h. All 256 interrupts are described by a FAR pointer in the table. As a FAR pointer in 16K real mode requires 4 bytes (16 bit segment + 16 bit offset), the whole IDT takes 1Kbyte in real mode. The pointer to interrupt 21h, for example is found at address: 21h*4 = 00000084h.

In protected mode, the IDT contains a list of ***I**nterrupt **D**escriptors* (ID), which occupy 8 bytes. The whole IDT with 256 interrupts, therefore requires 2KByte.

Bits / offset	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Interrupt Service Routine Offset (EIP) 15..0															
2	Interrupt Service Routine Descriptor (CS)															
4	1	DPL		0	Type				unused = 0							
6	Interrupt Service Routine Offset (EIP) 31..16															

Tab. 2.3 *Descriptor contents*

In general, the concept is similar to real mode: The IDT contains for each interrupt a descriptor, with an Selector:Offset to the ***I**nterrupt **S**ervice **R**outine* (ISR).

The protected mode defines different interrupt types:

- Interrupt Gate (TYPE = 0eh): The ISR is called with interrupts disabled,
- Trap Gates (TYPE = 0fh): The ISR is called with interrupts enabled.

The IDT provides for each of the 256 interrupts an interrupt descriptor. To load the IDT, the instruction LIDT is used. It uses the same format as the LGDT instruction:

```
.ALIGN 8
.CONST
  MAX_IDT_ENTRIES .EQU 255
.BLOCK
  IDTRECORD DW   MAX_IDT_ENTRIES*8
            DD   OFFSET IDT
.NOBLOCK
  IDT       DD   MAX_IDT_ENTRIES*8 DUP(0)
.CODE
```

```
LIDT IDTRECORD
```

Note, that you can reduce the number of interrupts in the `IDTRECORD`. The maximum is limited to 256.

2.5.2 Exceptions

The protected mode defines exceptions. An exception originates from the CPU, i.e. an exception is an interrupt, which is caused by the CPU due to protection means. A typical exepction is a division by zero: The CPU recognizes such an error in an application and interrupts the application with an exception. The main difference between exeption and interrupt is, that some exception put an error code on the stack. A simple return instruction from the ISR would not remove that error code from the stack. Therefore we need especial exception handlers for processor exceptions. Unfortunately, the processor exceptions are assigned to fixed interrupt vectors.

If we take a closer look at Tab. 2.4, we will see, that the reserved exception interrupt vectors are already used in a standard PC. For example `int 08h` is used by the hardware clock and also points to the double fault exception. Interrupt 010h is used by the VGA board, but also used as co-processor exception.

Everytime, an exception handler is called, it needs to check first, whether the cause is actually an exception, a hardware interrupt, or a software interrupt, i.e. `int` instruction.

Exception	Description	Remark
0 / 00h	Division by zero	No error code
1 / 01h	Debug Trap	No error code
3 / 03h	Break Point Interrupt (`int 03`)	No error code
4 / 04h	Overflow (`into`)	No error code
5 / 05h	Bound error	No error code
6 / 06h	invalid opcode	No error code
7 / 07h	no FPU extension available	No error code
8 / 08h	double fault	error code 0000h
10 / 0ah	invalid task state segment	error code = selector
11 / 0bh	segment not present	error code
12 / 0ch	stack error	error code
13 / 0dh	general protection fault	error code
14 / 0eh	page error	error code
16 / 10h	FPU exception	No error code
17 / 11h	alignment check	error code 0000h

Tab. 2.4 *Exceptions*

3. Dos Protected Mode Interface (DPMI)

In Chapter 2 we learned how to switch the processor into protected mode. Unfortunately this code runs on the highest privileged level. If you run such an application under Windows for instance, it will generate a general protection fault. Therefore we need other means to switch the processor into protected mode. Windows, Intel and other leading computer manufacturers came together and defined the DPMI standard: The ***D**os **P**rotected **M**ode **I**nterface*.

The DPMI provides to services to switch the processor into protected mode, to create and manage descriptors and to access real mode code like BIOS interrupts, Dos calls, etc.

DPMI is supported by Windows and other memory managers like QUEMM, 386MAX, etc. Unfortunately, the default memory manager EMM386 does not support DPMI. EMM386 supports only VCPI (***V**irtual **C**ontrol **P**rogram **I**nterface*). VCPI is an older (obsolete) standard, that was used by Windows itself to switch into protected mode. VCPI supports only the mode switching feature, but not any further services.

3.1 Detect DPMI

Before we can use DPMI services, we have to test, if DPMI is available. The DOS operating system provides a multiplex interrupt service to detect the presents of device drivers, memory managers etc. We can invoke the interrupt 2fh with the function code 1687h in AX to determine, whether the operating system provides a DPMI server:

Function call: INT 2fh

```
AX      = 1687h
```

Results, if successful

```
AX = 0
BX = 0000000000000000mb m =  1: 32 bit DPMI supported
CL = processor (80x86)
DX = DPMI Version, DH = majo r, DL = minor version number
SI = number of memory paragr aphs necessary for the mode switch
ES:DI = 16-bit real mode far  call address to DPMI enable procedure
```

If not successful:

```
AX != 0
```

Example:

```
 mov AX,1687h
 int 2fh
```

```
cmp AX,0
jne DPMI_ERROR                    ; No DPMI available
...
```

The interrupt call returns the entry point of a DPMI function to switch into protected mode and some useful information in the registers. If bit 0 of BX is set, the DPMI provides 32 bit.[1]

If there is no DPMI, we can not make use of the DPMI services. We could now try again to switch into protected mode by hand, but the conclusion, that if no DPMI is active, we can switch the processor into protected mode by using priviledged instructions is not true. So although DPMI provides all necessary services to code in protected mode, we still need a tool, that gives us a guarantee, that the services are available. This is the basic idea of a Dos Extender. A Dos Extender is a small program, that tests, if DPMI is available and if not emulates DPMI. Now if we link the Dos Extender to our program, we can use DPMI service, because the Dos Extender makes sure, that DPMI is available[2]. Most Dos Extender support as well VCPI, so a protected mode application loaded by a dos extender can basically run on any system configuration, i.e. plain DOS without memory manager, plain DOS with memory manager, Windows DOS Box, Linux Dos emulation with DPMI, etc.

3.2 Mode switch with DPMI

DPMI is a one way alley, i.e. you can switch the processor into protected mode, but you can't switch it back[3]. This may be a bit confusing in the beginning, but as DPMI allows to invoke 16 bit real mode code, there is no reason for an application to switch between real and protected mode. If the application terminates, the processor is switched back into real mode.

A nice feature is, that DPMI restores the state of the real mode system after the application has terminated. This makes DPMI programming much easier and more stable than real mode programming. Due to the processor protections, a DPMI program can hardly crash the system.

The DPMI API, the service functions are avaiable through the interrupt 31h. The DPMI function should be passed in AX. DPMI service functions are only available in protected mode. Before we request a DPMI service, we therefore need to switch the processor into protected mode.

1.Note, that there is no way to determine in advanced, if the DPMI host provides 16 bit. If bit 1 is set, the DPMI host provides 32 bit, but not necessarely 16 bit as well.

2.There are some different understanding of dos extenders: A dos extender can also mean, that it allows to perform Dos operations (i.e. int 21h instructions) in protected mode applications, e.g. extended dos functions. So a dos extender does not always provide DPMI services. Some dos extender even expect DPMI to be available (e.g. newer GO32 versions). In this case, the user has to load a DPMI application prior to the protected mode program. Through out the book, a dos extender means a DPMI service program and application loader like Pro32 / WDosX / Dos32, etc.

3.DPMI actually provides raw mode switching features, i.e. you can switch the processor into protected mode and back into real mode. These services are not recommended and unusual for DPMI programming. If you use these services, you must make sure, that your program terminated from protected mode - otherwise not all resources are freed by the DPMI host.

The function 1687h of interrupt 2fh returns a far procedure in ES:DI, that marks the entry point of a DPMI program. If we call this function, the DPMI host performs the following actions:

- switch the processor into protected mode
- creates a 16 bit code descriptor for CS, which has the same basis as CS of the caller (real mode CS) and limit 64K
- creates a 16 bit data descriptor for DS, which has the same basis as DS of the caller (real mode DS) and limit 64K
- creates a 16 bit stack descriptor for SS, which has the same basis as SS of the caller (real mode DS) and limit 64K[4], SP is in general not changed
- creates a selector for the program environment, which is found in the PSP at 2ch[5]
- maps all hardware interrupts to protected mode or to (virtual) real mode handlers

Note, that we need to provide a free memory segment to the function in ES. The amount of necessary paragraphs has been return by int 2f, 1687h in the SI register. If SI was zero, we can ignore the value in ES. Additionally, we have to specify, if we want to run a 16 bit or 32 bit DPMI application. The least significant bit of AX (bit 0) defines the DPMI mode: If set, the application is 32 bit[6]. The complete interface of the DPMI entry points is given as follows:

Function call: call far [es:di - according to int 2fh, AX=1687h result]

```
AX = 000000000000000mb m = 1: 32 bit DPMI, m = 0: 16 bit DPMI
ES = Free Memory, as request ed
```

Results, if successful

```
carry flag clear
CS: 16 bit code selector
DS: 16 bit data selector
ES: 16 bit program environme nt selector
SS: 16 bit stack selector
FS, GS: 0
```

If not successful:

```
carry flag set, program still in real mode
```

Example:

```
mov AX,1687h
int 2fh
cmp AX,0
```

4.Note, that the stack contents are not changed, i.e. if you push a real mode segment on the stack priot to the mode switch and pop the value afterwards, it is invalid and will force an exception if you load a selector register with it!

5.Check the **getenv** example provided with Pro32.

6.This book does not describe the obsolete 16 bit DPMI mode, only the 32 bit DPMI mode will be explained. See the DPMI specification for the differences.

```
      jne DPMI_ERROR                    ; No DPMI available
      mov DPMI_ENTRY,DI
      mov DPMI_ENTRY+2,ES
      cmp SI,0                  ; host does not request memory
      je NoMem
      mov AX,4800h              ; dos allocate memory
      mov BX, SI                ; number of paragraphs from SI
      int 21h                   ; allocate memory
      jc MEM_ERROR
      mov es,ax
NoMem:
      mov ax,1                  ; 32 bit protected mode!
      call far ptr DPMI_ENTRY
      JC DPMI_ERROR
```

You should have a look at the example file **DPMITST.ASM**, which prints all information provided by function 1687h, interrupt 2fh and switches the processor into protected mode. You should also try **CHECK32.ASM**, which is a tiny protected mode application, which prints some more information about the DPMI host, e.g. available memory, etc (Pro32 Example files).

3.3 Dos Extender

A dos extender makes life again much easier. First, the dos extender handles all DPMI functions to switch into protected mode and provides a loader, which will load our application into the extended memory space. Our application is not restricted to the 640 K Dos memory anymore. Even, if a system does not provide DPMI, a dos extender will switch the processor into protected mode and provide a DPMI API. A dos extender is typically linked at the beginning of the program. When the dos extender is executed, it first checks, if DPMI is available. If it is, the dos extender uses the DPMI service to switch into protected mode and then loads the program. If there is no DPMI, the dos extender typically checks VCPI. If VCPI is available, it will use VCPI to switch into protected mode and emulate the DPMI API. If no VCPI is available, the dos extender will test, if **HIMEM.SYS** is available. **HIMEM.SYS** does not allow to switch into protected mode, but it manages the extended memory. So if there is neither DPMI nor VCPI, it uses **HIMEM.SYS** to allocate XMS memory. If there is no DPMI, nor VCPI, nor XMS, then the dos extender will try to use the bios interrupt 15h, function `AX=8800h` to allocate XMS memory. If the bios interrupt is not available, the dos extender can now try to find free memory and the upper memory limit by its own or abort with the message, that no extended memory is available.

We refer to the Pro32 Dos Extender through out the book, as it is distributed together with Pass32. A Pass32 / Pro32 program uses the flat memory model, i.e. the program uses a single huge segment for code and data. The layout of the segment is described in Tab. 1 (compare with chapter 10.1).

Directive	Offset	Description
	00000000-000000ff	PSP - holds command line, environment settings
.code	00000100-xxxxxxxx	Main Program, the entry point is 0100h
.data	xxxxxxxx-xxxxxxxx	Initialised Data
.data?	xxxxxxxx-xxxxxxxx	Uninitialised Data
.data?	xxxxxxxx-ffffffff	Heap
	00000-xxxxx	Extra Stack segment

Tab. 3.1 *Typical Pass32 / Pro32 protected mode program*

The actual binary format is very similar to the DOS .COM format. The (obsolete) DOS PSP contains additional information of the DPMI host.

At the program start, we find the segment / selector registers load with the following selector values:

CS	32 bit code selector
DS	32 bit data selector
ES	16 bit video selector
FS	32 bit zero selector
GS	32 bit zero selector
SS	32 bit stack selector

Tab. 3.2 *Selector Register values at program start*

We can access the video memory for example with

```
.CODE
   mov es:[0],'A'            ; write an 'A' to top of the screen
```

We can access the bios data area with

```
.CODE
   mov ax,FS:[41Ah]        ; read the address of the Keyboard buffer
   mov KeyboardBuffer,ax   ; this would be in real mode 0040:001A
```

The FS and GS hold a so-called zero selector. They reference a descriptor with a base address of zero and limit of 4G. Basically you can address any memory of the machine via those selectors[7]. This descriptor is especially useful, when accessing real mode data or video data.

7.There are, of course limits: Under Windows and many other DPMI hosts, virtual memory management is enabled. Virtual memory management allows detailed memory protection: Each 4k memory page can be read or writeable according to the priviledged level. As a DPMI application usually runs in the lowest privileged level, a general protection fault is invoked, when protected memory areas are accessed.

We can access the video screen via FS, for instance by

```
.CODE
    mov FS:[0b8000h],'A'
```

DS and CS have the same basis address. Note, that just like in the TINY model, code and data are in the same segment. We cannot write into the code segment with CS - this leads to a general exception. But we can read from CS and write via DS.

The DS descriptor is the default descriptor. Every memory access without a segment definition refers to the DS segment:

```
    mov [25h],AX          ; is same as mov DS:[25h],AX
```

The stack segment is placed in another memory location, so that stack code and data never collide.

In case our DS or ES register will be destroyed, Pro32 offers a constant data area at the begin of our code segment, the so-called PSP. The first bytes of our code segment contain the following data:

00-01	DS - data selector
02-03	ES - video selector
04-05	FS, GS - zero selector
06-07	Real Mode File Buffer Selector
08-09	Real Mode File Buffer Segent
0A-0D	Actual allocated XMS Memory
0E	Flag, if windows has been detected[a]
0F	Flag, if other DPMI host is active[a]
2C-2F	selector to DOS environment
80-FF	command line with arguments

a. available with Pro32 Version 1.47 and newer versions.

Tab. 3.3 *The PSP of a Pro32 application*

You can access each of these data via CS:

```
.CODE
    mov ax,CS:[2]
    mov es,ax             ;restore es value
    mov ax,CS:[4]
    mov fs,ax             ;restore fs value
    mov gs,ax             ;restore gs value
    mov ax,CS:[0]
    mov ds,ax             ;restore ds value
    cmp byte ptr cs:[0eh],1
    je Windows
```

```
mov eax,cs:[0AH]
mov eax,MEMSIZE
```

The predefined variable identifier MEMSIZE points to CS:[0AH]. The last two examples are equivalent!

The real mode file buffer selector is a 32 KByte[8] buffer placed in real mode memory. You can use this buffer for DOS, BIOS or other real mode functions[9]. The real mode file buffer segment is the correspondening real mode segment value. Note, that you must use this value for real mode functions - in protected mode you must use the selector value!

At `CS:80h` and the following bytes you'll find a copy of the parameter line. Pass32 provides some library functions to analyze the parameter line, i.e. to separate the parameters.

In the opposite of other dos extenders, the code is not relocated: the code starts always at a virtual offset 00000100h. Every memory access inside this code segment is fixed to an absolute address inside this segment. Other dos extenders, e.g. DOS4GW, use also the flat memory model. Instead of providing a virtual segment, they provide a single flat segment starting at address 0. The code must be relocated by the loader, as the code can be load to any address starting from offset zero. In this case, a read from `cs:[B8000]` reads directly from the screen and a write to `ds:[A0000]` write to the video memory.

3.4 Using DPMI functions

In appendix B.3, you can get a list of all DPMI functions. The core of these DPMI functions (the most common used functions) are explained in the following example: a simple graphic module. Note, that the Pro32 dos extender does not provide all available DPMI functions. For instance the raw mode switching service is not available.

The graphic module has the same format as any assembler source. We name the module **`GRAPH.INC`** to demonstrate the difference between program and a module source. A program can include the module with the following directive:

```
.INCLUDE graph.inc
```

I want to start with the `InitGraph` Procedure. This procedure should initialize the graphic mode, set up a new descriptor for the graphic memory, install a new (user) defined graphic palette and get the address for the internal character ROM. As we assume a real mode graphic bios, you certainly see that we need a lot of real mode procedures to get the job one.

8.Why 32K is one of the FAQs. The answer is simple, 16K is too less and 64K would be too much. The purpose of the dos extender was to use a minimum of DOS memory. Pro32 for optimizes the DOS memory usage, i.e. when Pro32 has switched into protected mode, every code, that is not longer necessary will be freed to safe memory.

9.Use this buffer with care, as extended dos functions my use the same buffer.

To use the DPMI we can include the **DPMI.INC** file. The file contains a data definition field for the communication with real mode. This field is a 52 byte long field, where real mode registers, segment register, real mode flags etc are stored. These register storage are simply called `intedi`, `inteax`, `intes` etc, because they usually are used with a real mode `int` instruction.

Let us first include the graphic palette into the graph library. The palette is stored in the file graph.pal. We can include this binary file with the `.LOADBIN` directive. The `.LOADBIN` is similar to the `.INCLUDE` directive, it includes a binary file at the current offset in the code segment. As we want to know the offset of our palette, we define a label, before we include the palette (**GRAPH.INC**):

```
  .CODE
.PUBLIC colorpalette: ; declare label as public
.loadbin graph.pal ; load VGA Palette into program file
```

We can now address the palette via the offset: `OFFSET colorpalette`[10]. When we call the real mode bios to use our graphic palette, we must copy the palette first to real mode. This is always the problem when using real mode procedures. We should therefore try to use as less as possible real mode functions! To copy our palette to a real mode area, we use the File Buffer Area (a 32 KByte free data area in real mode)(**GRAPH.INC**):

```
Initgraph PROC NEAR ; Copy Colorpallete Into DOS Memory
   mov ax,[6] ; Real M ode File Buffer Selector
   mov es,ax
   mov edi,offset color palette ; access the colorpalette
   mov ecx,84 ; number  of entries / 4
INIT@PALLOOP:
   mov eax,[edi+ecx*4]  ; make full use of 32 bit register and memory
   mov es:[ecx*4],eax ;  copy to real
   loop INIT@PALLOOP
   mov eax,[edi] ; copy  the first 4 bytes as well
   xor edi,edi
   mov es:[edi],eax
```

After we initialized the graphic mode, we can install our own palette (**GRAPH.INC**):

```
   mov ax,13h
   int 10h ; init 320x 200x256 Color Mode
```

10.Note, that labels are declared as local inside procedures. To access a label global, you either declare the label as public or declare the label as external identifier - see 8.2.2 Defining Labels on page 81

To install the palette, we use the real mode function AX=1012h of the int 10h. But how can we use a real mode segment in protected mode? We can't! We have to use the real mode register structure as defined in **DPMI.INC**. The real mode register structure will contain all register values to perform a real mode function call. After the call, the resulting registers are stored in the structure. DPMI offers two kind of real mode functions: interrupts and FAR procedures. These are the DPMI service functions 0300h and 0301 (see appendix B3.20 / B3.21)

Function call: INT 31h

```
AX      = 0300h / 0301h
BX     = interrupt number (  BH must be 0) (ignored for 0301h)
CX     = number of words to copy from the protected mode stack to the
         real mode stack
ES:EDI = selector:offset of real mode register transfer data structure
```

Results[11]:

```
ES:EDI = selector offset of modified real mode register transfer data
         structure
```

We can pass valus via the stack. If there are no values passed, please make sure, that CX is zero! The DPMI service function expects a far pointer in ES:EDI to the real mode register structure (**GRAPH.INC**):

```
mov ax,ds ;SetPalette
mov es,ax ; ES = DS !!!
mov edi,offset intedi ; EDI  = OFFSET of Data Field
mov inteax,1012h ; BIOS func tion AX=1012h
mov intebx,0 ; BX = first pa  lette register
mov intecx,112 ; CX = 112 co  lors total
mov intedx,0 ; DX = Offset o  f the palette
mov ax,[8] ; Real Mode Segme nt To File Buffer
mov intes,ax ; ES = Real Mod  e Segment of the palette
mov ax,300h ; DPMI Function  0300h: Call Real Mode Int
xor cx,cx ; No parameters on  the PM Stack
mov bx,10h ; Interrupt Numbe r, BH must be 0
int 31h ; call DPMI function
```

It is a few instruction longer and a bit more confusing, but it works: The DPMI calls int 10h in real mode with the expected parameters. We can read the return parameters as well from this data field. Our next problem is to get the address of the video character ROM. We need as return parameters the ES register (Segment) and the BP (OFFSET) register. Again, we have to use the data field and the DPMI function 0300h (**GRAPH.INC**):

11.Make sure, that the execution of the real mode function does not effect the stability of the system. There is no exception handling in real mode.

```
    mov ax,ds
    mov es,ax
    mov edi,offset intedi ; make sure ES:EDI points to our structure
    mov inteax,1130h ; Get Offset of BIOS CHAR ROM
    mov intebx,300h
    mov ax,300h ; DPMI  Function 0300h: Call Real Mode Int
    xor cx,cx ; No para meters on the PM Stack
    mov bx,10h
    int 31h ; call real  mode int 10h Function 1130h
    xor eax,eax
    xor ebx,ebx
    mov ax,intes ; ax =  real mode ES
    mov bx,word ptr intebp ; bx = real mode BP (intebp = EBP = dword!)
    shl eax,4
    add eax,ebx ; calculate linear address (Segment+Offset)
    mov RomFont,eax ; save address of ROM character set
```

We calculate the address of the character ROM from the real mode segment value and the offset. We assume, that the physical address is equal with the linear address. The InitGraph procedure is now nearly finished. Our last thing to do is to create a selector to access the graphic memory(**GRAPH.INC**):

```
    mov ax,2 ; create real mode selector
    mov bx,0a000h ; for  graphic screen
    int 31h
    mov GSEL,AX ; store  selector
    ret
ENDP InitGraph
```

The DPMI function 0002h creates a selector from a real mode segement register value. This function is very useful to translate segment value into protected mode descriptors / selectors.

Before we test our graphic module, I want to take a short look on the `PutPixel` function. This function is now totally in protected mode (**GRAPH.INC**):

```
PROC PutPixel ; ECX: X EDX: Y BL : Color
    push edx ; save edx
    mov ax,gsel
    lea edx,[edx*4+edx]  ; edx:=edx*5
    mov es,ax
    shl edx,6 ; edx:=edx*64 <= edx*64*5 = edx * 320
    mov es:[edx+ecx],bl  ; plot ( edx*320+ecx )
    pop edx
    ret
ENDP Putpixel
```

We make, of course, profit of the fast 32 bit addressing. In combination with the `lea` instruction (***L**oad **E**ffective **A**ddress*), we can multiply very fast. The `lea` instruction is used to load address values. Whereas `mov` addresses the memory at the given address, `lea` calculates the address and returns the address in the destination operand:

```
lea edx,[edx*4+edx] ; edx:=edx*5
```

will multiply `EDX` by 5 and store the result in `EDX`. This is much faster as the `mul` instruction as it is processed in a single cycle (586).

Let us test our graphic module **`GRAPH.INC`** with a simple test program (**`TESTPAL.ASM`**):

```
.MODEL FLAT
.INCLUDE GRAPH.INC
.CODE
START:
    call initgraph
    mov edx,0
    @Loop:
    mov ecx,0
    @LineLoop:
    mov ebx,edx
    shr ebx,1
    add ebx,ecx
    shr ebx,2
    call putpixel
    inc ecx
    cmp ecx,320
    jb @LineLoop
    inc edx
    cmp edx,200
    jb @Loop
    mov ax,4c00h
    int 21h
END START
END
```

In general, the dos extender does the complete setup, mode switch and provides us with all necessary descriptors. Basically we don’t need any DPMI calls except for real mode calls and interrupt settings. Chapter 7 describes direct harware access using DPMI services.

4. Co-Processor programming

In some cases integer values, as they can be presented by the processor's registers, are not able to solve a mathematical problem. We need as well floating point operations. With the 80486 DX processors the FPU is integrated in the processor. Therefore Pass32 treats FPU instructions equal to CPU instructions.

4.1 Definition of floating point numbers

The FPU provides 8 register which use 10 bytes to store floating point numbers. There are basically three ways to present numbers inside a computer: integer numbers, fixed point and floating point numbers. A fixed point number is similar to an integer number, it's valus is simply shifted by one, two or more decimal stellen. The integer number 100 could also present 1.00 or 10.0 or 0.100. Fixed point numbers are used for money calculations, etc. A fixed point number has the same problem as integer numbers have: the limit of the range. If we take a 16 bit fixed point number with two digits after the point, we have a range from -327.68 to +327.67. This is not much, if we think of money for instance. A floating point number consists of two values: a mantisse and an exponent. The exponent presents a binary exponent, e.g. multiplication factor: the $\texttt{mantisse} * 2^{\texttt{exponent}}$ gives the actual value of the number. The mantisse is always normized as a real number between 0 and 0.99999... A real number in this case is defined as a binary number which is defined as:

$$x = b_0 * 1/2 + b_1 * 1/4 + b_2 * 1/8 + b_3 * 1/16 + b_4 * 1/32 \ldots$$

Now we have two values, which define the number range. The first value, the mantisse defines how many valid decimal digits our number provides - you can imagine, that a mantisse of 7 bit gives a maximum resolution of a 1/256, which is 0.00390625, i.e. you have a maximum of two valid digits. Now the exponent defines the range of the number. It does not enlarge the resolution, but imagine an exponent of 10 bit: your number can take values from 2^{-512} to 2^{+511}; this results in a range from about 10^{-154} to 10^{+153}. The IEEE has normed a set of floating point numbers and defined the values for mantisse and exponents. The FPU acts according to these standards and provides three kind of floating point numbers: `single` (32 bit), `double` (64 bit) and `temporary`[1] (80 bit):

1.the 80 bit temporary number is actually not according to the standard. It is used internally for the calculations.

Single	1.5E-45	3.4E+38	7-8 digits
double	5.0E-324	1.7E+308	15-16 digits
temp.	3.4E-4932	1.1E+4932	19-20 digits

Tab. 4.1 *Range of float numbers*

Floating point numbers provide a huge range, but they also have one big disadvantage: The actually provide only as many valid digits as the mantisse provides bits, so it is a valid operation to subtract two floating point numbers with totally different exponents. The problem is, the result is not correct! Therefore the FPU internally always calculates with the temporary format, to avoid such errors. Nevertheless, as soon as the number values differs more than 10^{20}, the FPU can't work with them.

4.2 The FPU internals

The FPU is stack based, i.e. similar to the processor stack, the FPU provides a stack to store its operands. The top of the stack is similar to the ackumulator of the processor: the FPU always uses the top of the stack as one of the operands and/or as result register. The FPU provides 8 registers, i.e. the stack is 8*10 bytes deep. The stack pointer arithmetic is *modulo* 8, i.e. when the stack pointer points to the 9th element, it points again to the stack top.

The FPU provides a 16 bit status register.

.

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	B	C3	ST			C2	C1	C0	IR	-	PE	UE	OE	ZE	DE	IE

Tab. 4.2 *FPU Status Register*

B: Busy is set, if the FPU is currently calculating a numerical expression. Note, that the FPU operations take longer than standard processor instructions.

ST: Stack pointer. The value 0-7 defines the top of the stack.

Execption Flags: PE: resolution error, UE underflow, OE overflow, ZE, division by zero, DE operand not normalized, IE invalid operation[1].

IR: interrupt request - set in combination with one of the exception flags.

C3,C2,C1,C0: status bits of the stack top. In C3, C0, you'll find the result of a comparison:

1.The FPU can invoke an exception (int 10h). The execption handler should analyze the cause of the exception by examine the status word.

fcom St1	C3	C0
ST>ST1	0	0
ST<ST1	0	1
ST=ST1	1	0

Tab. 4.3 *Comparison of Floating Point Numbers*

Since the 80486, the processor has direct access to the FPU, so you can for instance directly load the status word into the AX register with FSTSW AX will store the status word into AX. Luckily, the status bits are equivalent with the carry and zero flag of the processor status, so a SAHF instruction will load the FPU status into the CPU flags and you can directly use the conditional jump instructions to compare two float numbers, e.g.:

```
fcom ST1
fstsw AX
sahf
jg is_greater
```

The FPU provides a 16 bit control register:

.

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	-	-	-	IC	RC		PC		IEM	-	PM	UM	OM	ZM	DM	IM

Tab. 4.4 *FPU Status Register*

The exception mask flags: PM resolution error, UM underflow error, OE overflow error, ZM division by zero, DM operand not normalized and IM invalid operation. If a bit is set, the exception will not invoke an interrupt request.

IEM: If the interrupt enable mask is set, the FPU will not trigger an interrupt.

RC: The round control defines how the FPU rounds results: RC = 00 rounds to the next (possible) value, RC = 01 rounds downwards, e.g. to minus infinite, RC = 10 rounds upwards, e.g. to (positive) infinite, RC = 11 rounds towards zero. RC = 00 is the standard and is most exact.

PC: defines how the FPU should internally round values: PC = 00 rounds to temporary (80 bit numbers - standard), PC = 01 rounds to single, PC = 10 rounds to double numbers.

IC: The infinite control defines how the processor treats infinity: If IC = 0 positive and negative infinity is equal (all real numbers lie upon a circle), if IC = 1, the FPU provides positive and negative infinity (all real number lie upon a line).

4.3 Draw Circle Function with the FPU

We want to add a function to our graphics module, which is able to draw circles. We use the simple mathematical expression to create the circle:

$$y = M_y + R_y \cdot \cos(\varphi) \qquad x = M_x + R_x \cdot \sin(\varphi)$$

The first thing we do in our procedure is to define data. Data which is defined within the `PROC` and `ENDP` of a procedure can be optimized with the `.SMART` option (**`GRAPH.INC`**):

```
PROC Circle ; CX:X DX:Y SI:RadiusX DI:RadiusY BL:Color
.DATA
   CircleR1 dw 0
   CircleR2 dw 0
   CircleMX dw 0
   CircleMY dw 0
   CircleX dw 0
   CircleY dw 0
   CircleStart re 0
   CircleResolution re  0.02
```

As you can see we define also two floating point constants: `CircleStart` and `Circle-Solution`. The `CircleSolution` is the increment for φ, `CircleStart` represents φ.

The registers are called `ST(0)` = `ST` (stack top) to `ST(7)`. As the FPU is a stack oriented processor, we therefore call the FPU registers also *stack*. We can store results and constants on the stack (using the stack is faster than using memory references!).

The first thing we do is to store our parameters in variable identifiers. Then we use directly the FPU to set φ to zero(**`GRAPH.INC`**):

```
.CODE
   mov CircleR1,si
   mov CircleR2,di
   mov CircleMX,cx
   mov CircleMY,dx
   fldz ; load stack t op with zero
   fstp CircleStart ;  store zero to CircleStart and remove it from the
                       stack
   fld CircleResolution
```

With the last instruction we store `CircleSolution` on the stack. We keep this floating point constant on the stack as long as we calculate with it. Note, that we load the constant at first and that we load this constant only once, because when it is on the stack, we can use the stack directly. The first register (stack top) contains now 0.02.

We initialize `CX` with 2*φ/0.02 = 314; `CX` is again our loop register (**`GRAPH.INC`**):

```
    mov cx,314 ; 'Pi'
CircleLoop:
    push cx ; Save cx
    fild CircleR2 ; ST(2)
    fild CircleMY ; ST(1)
    fld CircleStart ; ST(0)
```

We load the integer variables R_y (radius) and M_Y (center position y) with the `FILD` (***FPU Integer LoaD***) instruction. At last we load our φ. As a comment I added the current register location of the FPU. Note that `ST(3)` is our 0.02 constant. We want to calculate y. This is done by the following sequence (**`GRAPH.INC`**):

```
    fcos ; calculate the cosine of φ = ST(0) ; result in ST(0)
    fmul st,st(2) ; multiply ST( 0) with R2
    fadd st,st(1) ; add MY to ST (0)
    fistp CircleY ; store the re sult as integer value in CircleY
                  ;and remove it f rom the stack
    fcompp ; (compare and) remov e MY and R2
    mov dx,CircleY ; store the r esult in DX
```

Calculating the X coordinate is done in the same way (**`GRAPH.INC`**):

```
    fild CircleR1 ; ST(2)
    fild CircleMX ; ST(1)
    fld CircleStart ; ST(0)
    fsin fmul st,st(2)
    fadd st,st(1)
    fistp CircleX
    fcompp ; ST(0) is now 0.02
    mov cx,CircleX ; store the r esult in CX
    fld CircleStart
    fadd st,st(1) ; + CircleReso lution
    fstp CircleStart ; store and  remove φ
    call putpixel
    pop cx
    loop CircleLoop
    fcompp ; remove CircleResolut ion
    ret
```

```
ENDP Circle
```

Adding to the `CircleSolution` the 0.02 factor, we don't need to load the constant from memory, because the constant is still on the FPU stack! To remove values from the stack we can either pop the values, using `fstp` (***F**PU **ST**ore and **P**op*) or using the fcomp (compare and pop) instruction to remove the values from the stack.

We will test the circle procedure with `CIRCLE.ASM`. The demo draws randomly circles in different colors on the screen. For the random generation, we use the function `SystemGetRandom` from **`SYSTEM.INC`**. The function creates random numbers in the range of 1..65535, according to the value of `EAX`.

5. Writing A DLL Library

In the mean time, DLL has become a common programming methods. Although the names are different, all modern operating systems provide kind of DLLs. DLL stands for dynamic link library, i.e. the library is linked at run time and not during the assembler / linker process. There are several advantages: first, several application can share the same library, second, the library can be adapted to the hardware without changing the code of the application and third, the DLL can be load and removed from memory according to its dynamical needs and memory requirements.

A DLL code is typically load only once into memory. Every application shares the same code. Every application reserves its own data area for the DLL, so that the DLL can hold different data for different applications.

The Pass32 Assembler provides a similar mechanism for protected mode programs. The assembler can create pure binary with an interface header. The interface header can be included in an application, so that the program is linked as if it uses a standard module. During the run time, the DLL is load and used by the application.

This Chapter describes the basic means of DLL programming. Our gaol is to create a simple graphic DLL. But before we start, we will first have a look on a simple example.

5.1 The First DLL

Our first DLL should simply demonstrate the method of DLL programming. The DLL consists of four public procedures, which simply print a message on the screen, when the are called. This is the interface of our first DLL (**`TESTDLL.ASM`**):

```
.MODEL DLL
.INTERFACEb


PROC TestDLLMain OFFSET DLLMain
PROC TestDLLProc1 OFFSET DLLProc1
PROC TestDLLProc2 OFFSET DLLProc2
PROC TestDLLProc3 OFFSET DLLProc3
```

The interface part ends with a `.DATA`, `.DATA?`, `.CODE` or a `.CONST` directive. The only elements of the interface are procedure definitions and their corresponding procedure offsets. The interface is included in the main program. When a DLL is load, the interface is load from the DLL. You can now call any procedure defined in the interface from the main program. If

you need more memory to install another DLL you can free the DLL until you want to use one of its procedures again. We use the `.FAR`[1] directive so that we can call far procedure with a forward reference. The order of the DLL procedures is free. Note that you must declare public procedure as FAR (**`TESTDLL.ASM`**):

```
.CODE
.FAR
PROC DLLMain FAR
   push ds
   mov ds, word ptr cs :[0] ;load DS selector
   mov edi,offset DLLMe sg
   mov bh,14
   mov TextColor,bh
   call systemwriteLn
   pop ds
   ret
ENDP DLLMain
```

As you see, the `DS` selector is saved and restored in the procedure. You should not forget, that when calling the DLL form the main program, the DS selector usually points to the segment of our main program. If we want to access DLL data, we must get the 'DLL - DS' from the DLL interface[2]. As you can see from 8.5 The DLL model on page 95, the DLL - DS is stored at `CS:[0]`.

A DLL typically has no heap memory. If you want heap memory, you must define a memory value with the `.MEM` directive.

Now we want to test the DLL. Look at the short demo program (**`DLLTEST.ASM`**)

```
.MODEL FLAT
.include TESTDLL.ASM
   DLL_ERROR .EQU 0
.DATA
   DLLname db 'TESTDLL. DLL',0
   ErrorMesg db 'ERROR: TSTDLL.DLL not found!',0
   ErrorMesg2 db 'ERROR : Too less memory available!',0
```

As you can see, we simply include the DLL. This does not mean of course, that the whole DLL code is included - this would make no sense; only the interface part of the DLL is included. In our example, the interface defines 5 variable identifier:

1.As Pass32 uses the TINY or FLAT memory model, Pass32 assumes per default near subroutine calls, i.e. a call uses a 16 bit (USE16) or 32 bit (USE32) offset. As a DLL is load into a different segement, its functions have to be called via a 16:16 (USE16) or 16:32 (USE32) bit pointer. The `.FAR` directive tells the assembler to assume far subroutine calls as default.

2.You can easily access data from the main program by not loading DS, but using ES to access DLL data.

```
TESTDLL DW ?
TestDLLMain DF ?
TestDLLProc1 DF ?
TestDLLProc2 DF ?
TestDLLProc3 DF ?
```

The first identifier is not part of the interface, the first identifier is simply a name for the interface. This name is taken from the source name: **TESTDLL.ASM**. You'll need this identifier for the `LoadDLL`, `InitDLL` and `FreeDLL` function. This identifier is a kind of identification for the DLL. This identification needn't be equal with the file name 'TESTDLL.DLL'! When the interface part of two different DLLs is equal, you can load both DLLs in the same interface. This might be necessary, when you want to address different graphic adapters, or different sound boards. You write for each configuration a different DLL, but all with the same interface part. This means: the public procedure identifiers must be equal and the order of the interface. You can now include one interface in your main program, and according to your hardware configuration at run time, you load only the DLL required. Automatically your program is configured correctly for the available hardware.

To load the DLL we can use this simple sequence (**DLLTEST.ASM**):

```
.CODE
   mov esi,offset TESTDLL ; OFFSET TO DATA BUFFER TESTDLL
   mov edi,offset DLLname ; OFF SET TO Filename
   call InitDLL
   call loadDLL
   jc dllnotfound
```

The functions `LoadDLL`, `InitDLL` and `FreeDLL` are part of the **DLLSYS.INC** file. This file is automatically added to your source file, when you include any DLL source file. You use the DLL 'id' and the offset of the filename as parameters for the `loadDLL` function. The `InitDLL` function simply needs the DLL 'id'. You should initialize all DLL at the beginning of your program to avoid calls to nowhere = exceptions! When you free a DLL, the DLL is automatically initialized. Any call is useless, but does not harm the system.

The `loadDLL` function first searches the current directory, if the DLL is not found, it scans the whole system path set by the $PATH environmental variable. Note, that the function may load an older version of the DLL, as it maybe found via the $PATH variable. It sets the carry flag if any error occurs and returns an error number in the AX register:

```
AX=1 Memory Error AX=0 Load Error
```

If you don't want to handle the error messages by yourself, you can set a label:

```
DLL_ERROR .EQU 1
```

The `loadDLL` function then automatically aborts if an error occurs. After a successful loading of our DLL we can use the DLL functions as if they are part of our program (**DLLTEST.ASM**):

```
call TestDLLMain
mov edi,offset Mesg1
mov bh,13
mov TextColor,bh
call systemwriteLn
call systemgetkey
call TestDLLProc1
call TestDLLProc2
mov edi,offset Mesg1
call systemwriteLn
call systemgetkey
call TestDLLProc3 . . .
```

You must assemble both files: **TESTDLL.ASM** and **DLLTEST.ASM**. To create a DLL you must run PASS32 with the **/DLL** option:

```
PASS32 TESTDLL /DLL
PASS32 DLLTEST
```

You can alter the DLL and restart **DLLTEST**. You will see the effect! You can use the internal debugger for DLLs. Why don't you try:

```
PASS32 TESTDLL /D /DLL
```

and run **DLLTEST** again?

5.2 A Graphic DLL

The first example should gave us enough information to create a more complex DLL. The fine thing about a DLL is, that you can easily create a DLL from an assembler module. You can even create a DLL from a complete assembler program. You simply add an interface to your file! This is the interface of our graphics DLL (**GRAPHDLL.ASM**):

```
.MODEL DLL
.INTERFACE

PROC InitGraph OFFSET Initgraph
PROC PutPixel OFFSET PutPixel
```

```
PROC GetPixel OFFSET GetPixel
PROC OutChar OF FSET OutChar
PROC OutTextXY  OFFSET OutTextXY
PROC Circle OFF SET Circle
PROC Line OFFSE T Line
PROC Rectangle  OFFSET Rectangle ...
```

We must add the FAR directive in every public procedure definition and - be careful, we must add loading and restoring the data descriptor if we want to access DLL data!

Look at the example **CIRCLE2.ASM**. At the beginning we have the same code sequence as in our **DLLTEST** demo. We load the DLL and check for an error, if no error occurs, we can use the graphic routines just as they had always been part of our program. For clear programming you should write a short module, which does the whole including and testing.

You should have a look at the **GRAPHIC.INC** module. The module offers a lot of graphic functions and addresses the graphic screen via different graphic DLLs: VGA, XVGA, VESA1.2 and VESA2.0 drivers.

5.3 A simple Windows DLL

Although this book does not cover Win32 programming, it should be stated, that Pass32 can also create Win32 DLLs. The difference of a Win32 DLL is, that it shares the same code segment with the main program. A DLL function therefore can not be declared as far. When Windows loads an application, it creates a virtual 4G address space for this application. Every DLL, that is required by the application is mapped into the virtual address space. Every application is (in principle) separated from the other applications and should not be able to harm other applications. When we create a DLL with Pass32, we use the same interface definition as for a Pass32 DLL. Whereas the Pass32 DLL uses a 256 byte long interface header, Pass32 creates an export table for the Win32 DLL according to this interface header. Note, that a Win32 DLL needs an intializer, which is basically the first function of the DLL. The init function needs to return zero to tell the windows desktop system, that the loading was successful.

5.4 A short chapter on OVL writing

Pass32 provides two additional linker forms: pure binaries and overlays. Pure binaries can be used for embedded system programming or microcontroller programming, which are based on an x86 core. Pure binaries can use the .ORG directive, to define the offset address of the program code. The .ORG directive must be used prior to the first .CODE or .CONST definition. Another linker option is the export as overlay. An overlay takes the same 256 byte interface

as a DLL, with the difference, that the overlay is load into the same segment. Therefore an overlay must be assembled at a different start address than the main program. A DLL is load in a totally new segment. When you free the DLL, all memory and descriptors will be freed as well. An overlay is load directly into your code segment. This has one advantage, and a lot of disadvantages: The advantage is, that overlay and main program share the same heap memory. It is easy to handle data with an overlay - DS and CS must not change, you can use a 32 bit offset to address data instead of an 48 bit pointer. But overlays are limited by the program heap memory, and you must define the location of the overlay while you are coding your program. You can compare the overlays with Windows' Drivers. You can write special hardware driver as overlays, and load the specific driver at run time. Especially for drivers the memory sharing with the main program is useful.

The only difference between an OVL source and a DLL source is the `.ORG` directive in the interface part. The simple overlay example has the following interface part (**`TESTOVL.ASM`**):

```
.MODEL OVL
.INTERFACE
.ORG 50000h

PROC TestOVLMain OFFSET OVLMain
PROC TestOVLProc1 OFFSET OVLProc1
PROC TestOVLProc2 OFFSET OVLProc2
PROC TestOVLProc3 OFFSET OVLProc3
```

The `.ORG` directive sets the code offset. Without this directive, the overlay would get an offset of 00000100h - the overlay would overwrite the main program, when it is load!

You must make sure, that the offset for the overlay is unused memory heap of your program!

The usage of an overlay module is again similar to the usage of a DLL module: You include the interface part of the OVL file with `.INCLUDE`. When an OVL module is included, the file **`OVLSYS.INC`** is automatically appended to your source. This module offers two main functions: `InitOVL` and `LoadOVL`. You should run the `InitOVL` function at the beginning. `LoadOVL` loads the overlay directly into the code segment, at the specified .ORG address! This is the sequence to load an overlay:

```
.CODE
   mov esi,offset TESTOVL ; OFFSET TO DATA BUFFER TESTOVL
   mov edi,offset OVLname ; OFFSET TO Filename
   call InitOVL
   call loadOVL
   jc OVLnotfound
```

The `loadOVL` functions sets the carry flag, if the overlay is not found. When the overlay is load, you can treat any overlay function as part of the program.

5.5 The binary format

In general, Pass32 is used to create direct executables, i.e. applications with the extension .EXE or .COM. Sometimes, it is required to get a plain binary file. For example, when x86 Software for embedded systems is written for ROM usage. With the -f option, Pass32 can create a binary image. In combination with the `.ORG` directive, the start offset can be defined. Note, that Pass32 supports only one segment, i.e. a flat segment.

6. Macro Power

This Chapter describes programming means of the Pass32 assembler, which are generally mode independent: Macros. A macro is a collection of instructions, directives and assembler commands, which can be combined in one macro. The ’power’of a macro is, that you can once define a complex set of instructions and then use them easily in your code. What might look like pascal or basic can be correct assembler code:

```
.CODE
   Writeln(’Hello, World’);
   Exit(0);
END
```

`Writeln` and `Exit` are macros. First, we have a look at the macro exit. This is a common macro type. The macro simple stands for a set of assembler instructions. Instead of writing an exit sequence, or calling an exit procedure, we can simply use the exit macro. But as you see, a macro can do more than inserting assembler instructions. A macro can use Pass32 directives as well. So we can define data storages with macros, use conditional assembly etc. And exactly this is, how we can create a syntax like Writeln(’Hello, World’).

Let’s have a look at the example file **`MACRO.ASM`**. In the main program, we have a basic like syntax:

```
.CODE
START:
   Print(’********** HELLO! *************’)
   print(’This is a simple macr o example!’)
   print(’******************************’)
   PrintError(’EOF Demo reached. ’)
END START
END
```

`Print` and `PrintError` are macros. Now, lets have a look at the macro `Print`:

```
.MACRO Print(Message)
   Create_Message(.LOCAL MESG,Message)
   Writeln(offset .LOCAL MESG)
ENDMACRO
```

Obviously, the macro uses again other macros. We don't learn much from this macro. But we can see, that we can pass the parameter to another macro. Obviously `Create_Message` is a macro, which generates a data buffer with the contents of our message. So let's look at the macro `Create_message`:

```
.MACRO Create_Message(name,string)
.DATA
   name db string,0 ;  create string message
.CODE
   ENDMACRO
```

This macro produces data and not code. The parameter string contains our message. But what is the parameter name for? Quite easy, to access a data storage, we need an identifier, a name for the data storage. The name comes from the macro print: As we see, `Create_Message` is called with the parameter `.LOCAL MESG`. The name of the data storage is `MESG`. As we want to use the macro print more than once, the identifier name must be unambigous. This is done by the `.LOCAL` directive. Inside a macro the `.LOCAL` directive simply extends the identifier name by a hex number, so instead of `MESG`, our parameter is something like: `@HHHHHHHHMESG`, where `HHHHHHHH` stands for an individual hex number.

The other macro used by `Print` is `WriteLn`:

```
.MACRO WRITELN(stringoffs)
   WRITE(stringoffs)
   mov dl,10
   mov ah,2
   int 21h
   mov dl,13 ; do Carr iage Return
   mov ah,2
   int 21h
ENDMACRO
```

The parameter `stringoffs` is passed to the macro `Write`. The rest of the macro simply prints #10,#13 via standard output. As we can see from the `Print` macro, the parameter `stringoffs` is the same as the first parameter to `Create_Message`. We first create a data storage and then call a function to display this data storage. The macro `Write` is simply a function to display the string via DOS:

```
.MACRO WRITE(stringoffs)
   mov edi,stringoffs
.LOCAL @start:
   mov dl,[edi]
   cmp dl,0
   je short .LOCAL @end
   mov ah,2
```

```
    int 21h ; display string char by char
    inc edi
    jmp .LOCAL @start
.LOCAL @end:
ENDMACRO
```

Interesting is the use of `.LOCAL` inside of this macro. We need `.LOCAL` to define unambigious label names. Without `.LOCAL`, the second use of this macro would lead to the error: 'duplicate label', because `@start` would have been defined already. With `.LOCAL` we create macro individual labels. Again, the labelname is extended by an individual hex number.

Now, what happens when we call the macro `print` like this: `print('Hello, World')`?

```
print('Hello World')
| ======> Create_Message(.LOCAL MESG,'Hello World')
        | ======> .DATA
                  @00000001MESG DB 'Hello World',0
                  .CODE
| ======> WriteLn(offset .LOCAL MESG)
        | ======> Write(offset .LOCAL MESG)
                  | ======>mov edi, @00000001MESG
                           @00000002@start:
                           mov dl,[edi]
                           cmp dl,0
                           je short @00000002@end
                           mov ah,2
                           int 21h ; display string char by c  har
                           inc edi
                           jmp @00000002@start
                           @00000002@end:
        mov dl,10
        mov ah,2
        int 21h
        mov dl,13 ; do Carriage Return
        mov ah,2
        int 21h
```

With this knowledge, we can write totally different assembler code. And exactly that's the reason why Pass32 comes with a macro library **`SYSTEM.MAC`**, which presents standard I/O functions as macros (an extension of **`SYSTEM.INC)`**. Instead of a call to the function `SystemWriteln`, you can simply use the Macro `WriteLn(String)`. Look at the very short demo file **`MACRO2.ASM`**:

```
.include system.mac
.CODE
start:
   call systemclrscr
   print(15,6,'******************** MACRO 2 ********************')
   print(15,7,'*                                              *')
   print(15,8,'*                                              *')
   print(15,9,'*                                              *')
   print(15,10,'******************** MACRO 2 ********************')
   color(14,0)
   print(31,8,'Simple Macro Demo')
   color(14,7)
   print(0,0,' Macro2.ASM ');
   print(0,24,' Press a ny key to continue ');
   call systemgetkey
   color(7,0)
   print(0,24,' ');
   gotoxy(1,12);
   exit(0)
END start
END
```

As you can see, it is a combination of macros and function calls. We learned quite a lot advantages of macros. But I should mention the disadvantage, too. A macro is not a function, the usage of a macro is not a call instruction, but the insertion of the macro instructions. If you for example convert the whole SystemWriteln function into a macro, your program will get quite long, if you often call this macro. So you have to decide between a fast execution (without overhead like call and ret) and a smaller code. You should therefore use macros either to simplify your source code, for example `Writeln('Hello, World')`, to gain speed by executing a 'fast' function call, or to combine a small number of assembler instructions, which is used quite often, for example `exit(0)`.

7. Access to Hardware from Protected Mode

The DPMI interface allows to use all BIOS and real mode resources from protected mode. Unfrotunately, the DPMI is not designed to emulate this resources from protected mode. Therefore, a call to a mouse driver function e.g., is actually a call to real mode. Even every time, the mouse moves, the real mode mouse handler is invoked. Another example is VESA graphics. When we use VESA graphics according to Version 1.2, we have to invoke real mode functions to move the graphic window - although we can address the whole address space, we are restriced to a 64K graphic page. The Vesa standard V2.0 solved this problem: From 32 bit protected mode it is now possible to access the complete video ram.

7.1 Protected Mode Mouse Driver/Handler

This chapter describes the implementation of a protected mode mouse driver. We can of course use the real mode driver `int 33h` to access the mouse, but this would mean, that our processor is running mainly in real mode. We want to access the mouse driver directly. This means we install our own mouse driver, which is totally written in protected mode. For the ininitialisation and the mouse detection, it still needs a real mode mouse driver. As a PS2 mouse would additionaly require a new keyboard handler, the demo supports only a serial mouse on COM1 or COM2.

The real mode mouse driver is a TSR program. It installs according to the mouse port COM1 or COM2 (or PS/2 mouse), an interrupt service routine. This interrupt routine is called every time, the serial mouse sends some information over the port. In words, every time we move the mouse or press a button, the real mode procedure is called. If we write our own interrupt service routine, we would get two effects at once: The processor mustn't switch to real mode and - every mouse information is directly available - we do not even need to call the `int 33h`! With Pro32 we can install a hardware interrupt service routines directly with the DPMI Function 0205h. When an HW interrupt occurs, our installed procedure is called; when an exception occurs, the correspondending exception handler routine will be called. As long as we do not create our own exception handling, Pro32 does the exception handling. For more information about exception handling, you can look at the Pro32 demo file **`DEMOEXC.ASM`** - this example detects a division by zero with a user defined exception handler...

The following code installs our own mouse interrupt service routine with function 0205h:

```
PROC InitMouseCom1 NEAR
    mov cx,cs
    mov edx,OFFSET Com1Mouse
    mov ax,0205h ; set pm
    int mov bx,0ch ; int 0ch
```

```
    int 31h ; set new interrupt
    ret
ENDP InitMouseCom1
```

We don't need to save the old interrupt state, because all protected mode interrupts are invalid, when we return to real mode. As the mouse interrupt is an hardware interrupt, we need to send an EOI (End OF Interrupt) to the interrupt controller at the end of our routine:

```
PROC Com1Mouse
    push eax ....
    mov al,20h
    out 20h,al
    pop eax
    iret
ENDP Com1Mouse
```

The next problem is the communication with the mouse. We assume that a real mode mouse driver had been load already, so we don't need to initialize the mouse, the port, the interrupt, etc.

A MS compatible mouse sends three bytes for a movement, or a button click. The first byte has information of the move directions and the button, we realize the first byte, because the 6th bit is always set. Our service routine does not wait for all three bytes - we receive a single byte and wait until we have got all three bytes. Here is a list of information in these bytes:

1st byte	2nd byte	3rd byte
0 1 L R Y Y X X	0 0 X X X X X X	0 0 Y Y Y Y Y Y

Tab. 7.1 *The Serial Mouse Protocol*

L stands for the left mouse button and R stands for the right mouse button (1 means pressed). X describes the X-increment between the last calls and Y the Y-increment. X and Y are signed 8 bit numbers. The first byte is indicated by '01'.

After we received the third byte, we can analyse the information, and store the information in global data:

```
.PUBLIC .DATA
    MSX DW 100 ; mouse x position
    MSY DW 100 ; mouse  y position
    MSMAXX DW 640 ; max mouse x position
    MSMAXY DW 400 ; max  mouse y position
    MSLEFT DB 0 ; left  mouse button
    MSRIGHT DB 0 ; righ t mouse button
```

When our service interrupt is installed, we can access any mouse data through these identifiers. We add to our service routine a dummy procedure which should draw the mouse. In our main program we can define a procedure which draws the mouse on screen, depending on the video mode of the program, We replace the dummy procedure in our mouse driver - and our mouse support is perfect!

The example files **MSDEMO.ASM** and **MSDEMO2.ASM** test our protected mode mouse driver. **MSDEMO2.ASM** uses the **GRAPHDLL.DLL** we created in 5. Writing A DLL Library on page 53, to display graphics.

7.2 Vesa 2.0 graphic driver

The VESA 2.0 extensions are mainly concerning the protected mode support for graphic adaptors. In the mean time, modern operating systems do not use real mode code to access the video screen. This allows on the one hand to access segments above 1MB and on the other hand, segments are not restricted to be 64K in size. VESA 1.2 solved this problem by splitting the video memory in 64K pages, which are mapped to the 64K segment at A0000h. With VESA 2.0, this window management is obsolete, because the graphic adaptor can map the whole video memory into the 4G address space. If VESA 2.0 is installed, we simply need the physical address of the video memory, use the VESA function to enable the graphic mode. Once the graphic mode is set up, we can access thw whole screen via a video selector, or the zero selector for instance.

A common pitfal is, that the physical address is usually not equal with the linear address, due to the virtual management[1]. DPMI provides a function call to determine the linear address of a physical address. This function is required, if we want to access the VESA graphic in a Windows DOS emulation:

Function call: INT 31h

```
AX      = 0800h
BX:CX   = physical address in memory
SI:DI   = size of region in bytes
```

Results, if successful:

```
carry flag clear
BX:CX  = linear address
```

The following steps are required to access the video ram of a VESA2.0 adaptor: First, we have to learn the actual physical address. Then we map the original physical address to a linear address. Now we can address the video ram through the linear address. The following code example will calculate the linear address of the video ram for a VESA2.0 graphic card:

1.Pro32 disables virtual mangement when possible, because it slows down the processor’s speed.

```
MOV InteAX,4F01h
MOV InteCX,101h
MOV AX,CS:[8]      ; Pro32 real mod e buffer segment value
MOV IntES,AX
MOV InteDI,0       ; Offset 0
MOV AX,DS
MOV ES,AX
MOV AX,300h
MOV BX,10h
MOV CX,0
MOV EDI,Offset InteD I
INT 31h            ; call VESA function: Get Mode Info
MOV FS, word ptr cs :[6] ; Pro32 real mode buffer selector
MOV CX,FS:[40]
MOV BX,FS:[42]
CMP dword ptr fs:[40],0 ; linear frame buffer address
je XX                   ; if zero -> Error!
test byte ptr fs:[0],128; support for VBE 2.0?
je XX                   ; no -> Error!
MOV SI, SIZEX*SIZEY/65536; graphic screen size
MOV DI, SIZEX*SIZEY% 65536
MOV AX,0800h  ; get linear address
int 31h
MOV word ptr FrameBu fferBase,CX
MOV word ptr FrameBufferBase+2,BX
```

8. The Pass32 Assembler

This Chapter gives an overview on the features of the Pass32 Assembler. It describes all assembler directives and the general layout of assembler programs.

8.1 Defining Code, Data and Memory Model

Pass32 supports two main memory models: The Dos compatible TINY model which is used for **.COM** executables and the Dos Extender compatible FLAT memory model. The TINY model is restricted to a 64 Kbyte segment for code, data and stack. The FLAT memory model can use up to 4 Gigabytes code and data. The creation of **.COM** files is useful, if you like to create DOS oriented code (file transfer, file filters etc). The flat memory model is used for working with a great amount of data, for example for a graphic oriented system. It is the decision between real and protected mode, between 16 and 32 bit!

You define the memory model with the `.MODEL` directive. You should always use this directive! This directive defines the segement type attribute (USE16 or USE32).

8.1.1 Defining the TINY model:

Total Segment Size 64K for code, stack and data. The memory model is defined by writing:
.MODEL TINY
Tab. 8.1 shows the typical layout of a `.COM` program.

Directive	Offset	Description
	0000-00ff	PSP - holds command line, environment settings
.code	0100-xxxx	Main Program, the entry point is 0100h
.data	xxxx-xxxx	Initialised Data
.data?	xxxx-xxxx	Uninitialised Data
	xxxx-ffff	Reserved for Stack and Heap

Tab. 8.1 *The Tiny Model*

The complete program cannot exceed 64K. The shaded sections in the table are actually part of the program binary. The other parts are created during the run time by the operating system (PSP, STACK) or by the program itself (`.DATA?`). The stack starts from 0ffffh downwards. You must make sure, that your program has enough stack size. A good idea is to allocate an extra 64K for the stack. In this case you don't have to control the stack usage and your code/data size (**NEWSTACK.ASM**):

```
.MODEL TINY
```

```
.CODE
.DATA MemError DB 'Not enough real mode memory available',13,10,'$'0
START:
     MOV AH,48h
     MOV BX,65536/16    ; amount of memo ry = BX*16
     INT 21H
     JC NoMem           ; error - not en ough memory
     CLI                ; no interrupts now
     MOV SS,AX
     MOV SP,0fffeh      ; word aligned s tack
     STI
     ...                ; start of the original program
NoMem:                  ; print error Message with DosPrint
     MOV DX, offset MemEr ror
     MOV AH,9
     INT 21H
     MOV AX,4C03h
     INT 21h
```

Example 3: *Defining a new stack Segment*

8.1.2 Defining the FLAT model:

Total segment size 4G[1] for code, stack[2] and data.

Directive	Offset	Description
	00000000-000000ff	PSP - holds command line, environment settings
.code	00000100-xxxxxxxx	Main Program, the entry point is 0100h
.data	xxxxxxxx-xxxxxxxx	Initialised Data
.data?	xxxxxxxx-xxxxxxxx	Uninitialised Data
.data?	xxxxxxxx-ffffffff	Heap
	00000-xxxxx	Extra Stack segment

Tab. 8.2 *The Flat Model*[3]

1.The 4G address space can be addressed by these applications. There is usually a limit of 64MB by the dos extender or even smaller limits due to the operating system. Pro32 and Pass32 generally support programs up to 4G length.

2.The Stack segment is usually different from the CS/DS segment when using a dos extender, but some dos extender define SS=DS.

3.This memory model is provided by the Pro32 debugger. The stack segment is an extra segment, which is resident in the real mode memory area (usually locked under Windows/WinNt)

Tab. 8.2 shows the layout of a typical flat memory model, as it is provided by the Pro32 Dos Extender. Other dos extender might start directly at the offset address 000000000h (e.g. WDOSX). Again other dos extenders might start at a virtual address, e.g. 01000000h, where the low 1 MB address range presents exactly the real mode address range. These dos extenders are supported as well by Pass32, by setting the program start offset, e.g:

```
.MODEL FLAT
.ORG 1000000h
```

8.1.3 Data definitions

As you can see from the tables, there are different data types:

- predefined data - at the end of the **.EXE** / **.COM** file
- undefined data - value is unknown
- constant data - part of the code

You define the beginning of data with the .DATA, .DATA? or .CONST directive:

.DATA	The following definitions are part of the „data segment“. They allocate and initialize a data storage at the end of the program code
.DATA?	The following definitions are no part of the program. A data storage is allocated, but of course not initialized.
.CONST	The following definitions will be placed directly into the code segment. A .CONST directive at the begin of an assembler source forces the first bytes of the code to be data storage and not to be instructions!

Tab. 8.3 *Data Segment Definitions*

The .DATA and the .CONST directive differ in the location of the data storage and in the usage of the data. The syntax is for both directives the same. You can place constant variable identifiers directly into the code segment:

```
     H1 DW ?
PROC Test
     mov H1,AX
     mov BX,H2 ...
     ret
     H2 DW ?
ENDP Test
```

A constant variable is usually accessed via the CS selector; `mov H1,AX` therefore will become `mov CS:H1,AX`. To avoid a general exception (a write memory access with CS) you *must* use a segment override, for example DS, to alter constant data. In a clear source code you should use constant data as constants and not alter them!

A data definition is done by defining an identifier (variable name) a storage size directive (size of variable type) and the data value. Pass32 offers the following storage size directives:

DB	1 byte	allocates one byte storage(8 bit)
DW	2 bytes	allocates one word storage(16 bit)
DD	4 bytes	allocates a double-word storage (32 bit)
DF, DP	6 bytes	allocates a 32-bit far pointer
DQ	8 bytes	allocates a quad-word storage
DT	10 bytes	allocates a ten bytes storage

Tab. 8.4 *Data storage directives*

For floating point constants there are four more directives:

RS	4 bytes	allocates storage for single floating point number
RD	8 bytes	allocates storage for double floating point number
RC, RE	10 bytes	allocates storage for a comp (extended) float number

Tab. 8.5 *Data storage directive for float numbers*

Any directive/label/procedure name is treated as not-case-sensitive.

8.1.4 Data Expressions

To initialize data you use the following syntax:

```
[NAME] DB | DW | DD | DF | DP | DQ | DT expression {,expression}
```

To initialize a floating point number

```
[NAME] RS | RD | RE | RC float-constant {,float-constant}
```

An expression might consist of the following numbers:

- hexadecimal : denoted with an h following the number and must begin with one of the digits 0 - 9, e.g. 1234h. 0fffh, 4a7h
- octal : denoted with an o suffix, eg. 0123o
- binary : denoted with a b at the end of the number, eg. 010100b
- decimal : this is the standard notation, eg. 65535

- characters : a character is presented by its ascii code, eg. 'A' = 65 several characters are stored as they are: 'YOU' = 59h,4fh,55h[1]

and the following operations:

- `( )` : marks an expression for priority evaluation
- `[ ]` : marks an expression as a memory reference
- (unary) `-` : changes the sign of the expression
- `NOT` (unary): logical NOT (inversion) of the expression
- `AND` : logical AND of two expressions
- `OR` : logical OR of two expressions
- `XOR` : logical XOR of two expressions
- `*` : multiplies two integer expressions
- `/` : divides two integer expressions
- `%` : gives the remainder of an integer division (modulo)
- `-` : subtracts two integer expressions
- `+` : adds two integer expressions
- `SIZE` : size of a data type (DB = 1, DW = 2, ...)
- `BYTE PTR` : to address a single byte memory location
- `WORD PTR` : to address a word memory location
- `DWORD PTR` : to address a double word memory location
- `FWORD PTR` : to address a 32-bit far pointer location
- `QWORD PTR` : to address a quad word memory location
- `TBYTE PTR` : to address a ten byte memory location

Inside a `.CONST` data expression, you can use the directive:

- `OFFSET` : to get the address of a label / procedure / identifier[2]

Some examples of data definitions:

```
.DATA
    ByteVar   DB 01000b OR 01b OR 100001b
    ByteRow   DB 0,1,2,3,4,5,6,7,8,9,10
    WordVar   DW 0EF7Ah AND 01110011011b
    WordRow   DW 10+2*20,20+2*20,30+2*20,48*8,48*16,128%7
    DWordVar  DD 0000100h+SIZE ByteRow*11+SIZE ByteVar
    BCDNumber DT ?
    mesg      DB 'Hello, World!',0
    unicode   DW 'Hello, World!',0
```

1.Using characters with word constants (DW), Pass32 creates UNICODE conform characters (16 bit)
2.Note, that you can't use offset inside .DATA. This is due to the fact, that .DATA is already parsed in the first pass, when the offset addresses are not yet known.

The question mark is a special expression, the value of the expression is unknown and unimportant! You *can not* use the question mark with an operation: 10*?,? or 01b (wrong!!!!) The question mark is the only allowed expression for undefined data:

```
.DATA?
    IntResult DD ?
    ResultVec DD ?,?,?
    FloatRes  RD ?
    BCDResult DT ?
```

A floating point constant can be defined with RS, RD, RE or RC: RS is the identifier for a single floating point number (4 bytes). The constant has the following format:

```
[-] digits [.digits] [e +|- digits]
```

The range for the floating point numbers are:

RS	1.5E-45	3.4E+38	7-8 digits
RD	5.0E-324	1.7E+308	15-16 digits
RC, RE	3.4E-4932	1.1E+4932	19-20 digits

Tab. 8.6 *Range of float numbers*

Some examples for floating point constants:

```
.DATA
    Single        RS 1.5
    SingleRow     RS 3.7E-8, 19.2145, 18.125E+12, 29.111, 77.99
    Double        RD 1.999E+200
    DoubleRow     RD 1.77125, 1.998192, 0.25E-38, 0.125E+128
    Comp          RC -1.999e-500
    CompRow       RE 0.999999999E-22,33344556E+88,9123456E-88
```

The DUP duplicate directive creates an array of a data type[1]:

```
[NAME] DB | DW | DD | DF | DP | DQ | DT field-count DUP ( expression )
[NAME] RS |  RD | RE | RC field-count DUP (floating-point-constant)
```

The expression or floating-point-constant is duplicated field-count times. Examples:

1.Note, there's no recursive usage of DUP, instead of DB 256 DUP(1024 DUP(?)) write DB 256*1024 DUP(?)!

```
 .DATA
    SingleArray   RS 1000 DUP(0.1)  ; 1000 times the constant 0.1
    ByteField     DB 1024 DUP(0) ;  1k bytes with value 0
 .DATA?
    VideoBuffer   DB 256*1024 DUP(? ) ; 256K Video Buffer
```

Basically, data identifiers are treated as local inside a module. This means, you can access only data identifiers, which are defined in the same module; and, in different modules, you can use the same names for data identifiers...

If you want to access data defined inside another module, you have to make export this variable. There are basically two ways of exporting data: You can use the `.PUBLIC` directive inside this module:

```
.DATA
     privateid DB 0
     .PUBLIC publicid DB 1
```

The identifier `publicid` can now be accessed from other modules. You can use `.PUBLIC` before the `.DATA` directive to make all data identifiers public:

```
.PUBLIC .DATA
     ispublic DB 0
     ispublic2 DB 1
```

The first method needs a change to the module file. This might not always be useful as the identifiers made public will always be exported (if they are needed or not). You can force a module to export a data identifier (which is not made public) with the `.EXTERN` directive:

```
MODULEA.INC:
 .DATA
privateid DB 0
     .PUBLIC publicid DB 1

MODULEB.INC:
.INCLUDE MODULEA.INC
.EXTERN privateid ; now publicid and privateid can be accessed!
```

For the .EXTERN directive, the identifier must be known, this means, the module must be included first[1]. .EXTERN is ignored, if the data identifier is already made public. .EXTERN and .PUBLIC can also be used with lables and procedures.

The .ALIGN directive forces the assembler to align data or code, depending in which segment the directive is used[2]. With .ALIGN you can enable alignment inside a data segment:

Every data identifier begins at an offset divisible by 2 (TINY) or 4 (FLAT) according to the memory model. If you need a complete block of data, you can use the .BLOCK and .NOBLOCK directive. Between .BLOCK and .NOBLOCK each data identifier follows the other without a gap. Therefore you have data alignment for the first item, but not certain for the other elements. The .NOALIGN directive disables data alignment. You can specify the data alignment by a cardinal number of 2,4 or 8 irrespective of the memory model.

Example:

```
.MODEL FLAT
.DATA .ALIGN            ; same as .ALIGN 4
     AlignedByte        db ?
     AlignedWord        dw 0
     ALignedDWord       dd 0123456h

.BLOCK
     TextStrings        db 'INPUT ','OUTP UT','LIST ','RUN '
                        db 'NEW ','OPEN  ','CLOSE ','CLEAR '
     TextStringEnd      db 0
     TextStringElements db 8
.NOBLOCK
     AlignedBytes db 0,0,1,1,2,2
```

I

f you use .ALIGN inside the code segment, Pass32 will insert as many nop instructions until the next instruction is aligned[3].

8.1.5 Predefined Data Identifiers

Pass32 defines one global variable identifier for the TINY and two global variable identifier for the FLAT memory model:

1.Note, the .EXTERN directive actually copies the information of the private module data identifier into the public data area. For this reason, .EXTERN allocates the identifier twice and needs more memory than the use of .PUBLIC.

2.Note, that Pass32 V2.0f ignores an .ALIGN directive outside any segment (for example at the beginning)

3.Note, .ALIGN doesn't have an effect on the code segment generally, like it has in the data segment.

a) TINY

- LASTDATA: OFFSET of the first free data address of the program heap points to the last .DATA? identifier + its size.

b) FLAT

- LASTDATA: OFFSET of the first free data address of the program heap points to the last .DATA? identifier + its size.
- MEMSIZE points to CS:[0Ah] of the protected mode PSP = amount of allocated memory for the program segment (Pro32 Dos Extender[1]).

If you name any variable identifier with one of these names, you will get a warning message.

8.1.6 Usage of Data Identifiers

Any variable or constant identifier replaces the actual address. You can use the variable instead of its address:

```
mov ax,WordValue ; would be the same as:
mov ax,[OFFSET WordValue] ;
```

The OFFSET identifier returns the address of a label, a procedure or a variable. If you want to address different elements of a variable identifier, you can use the PTR identifier. For example, if you want to make a far call via a variable:

```
.DATA
     FarPtrVar DF ?
.CODE
     mov ax,cs
     mov edi, OFFSET FarPtrEntry
     mov WORD PTR FarPtrVar+4,AX  ; AX and FarPtrVar different types!
     mov DWORD PTR FarPtrVar,EDI  ; EDI and FarPtrVar different types!
     call FarPtrVar ; FarPtrVar i s expected type!
FarPtrEntry:
```

If you try to load or store with a different type of variable, you will get an error message:

```
mov ax,ByteVar ; illegal types!
```

You must use the PTR identifier if you want to avoid an error:

```
mov ax,BYTE PTR ByteVar
```

1.When other Dos Extenders are used, you must ignore MEMSIZE, as the contents are invalid. Nevertheless, LASTDATA points to the beginning of the free heap and can be used with any Dos Extender

Pay attention, if you use aligned data, you can not be sure what the other bytes of a byte or word identifier contain! You should not use the knowledge that an aligned data storage uses a minimum of 2,4,8 or 16 bytes to store other values than the defined one! For example it would be terrible programming if you'd store a double word value into AlignedByte, because you know that four bytes had been reserved! The `SIZE` identifier will return an amount of 4 for the aligned byte, word or dword identifier and 8 for a 32 bit far pointer and a quad word!

You can define data anywhere in your source code:

```
PROC Print
.DATA
     PrintX DW ?
     PrintY DW ?
.CODE
     mov PrintX,Cx
     mov PrintY,Cy


     cmp ax,0
     je AxIsZero
.Data
     AXZero db 0
.CODE
     mov AxZero,1
AxIsZero:
```

Note that memory for the data identifier is allocated at the end of the program code. As constant data defined with the `.CONST` directive is allocated directly in the code segment, you can not define data with the `.CONST` directive anywhere in the code!!!

wrong:

```
.CODE
     PROC Dummy
.CONST
     Dummy1 DD ?
     Dummy2 DD ?
.CODE
     xor eax,eax
     mov Dummy1,eax
     mov Dummy2,eax ...
```

When the procedure Dummy is called, the first 8 bytes contain undefined data - the behaviour of the procedure is unknown... You can place `.CONST` definitions at the end of a procedure (after the `RET` instruction) or between two procedures. Or you can use the `.CONST` directive, to create special instructions, for example:

```
PROC NewInt8
     call Newinthandler
.CONST
     DB 0EAh ; JUMP
     OLD8OFFS DD ? ; FAR JUMP TO  OLD INTERRUPT HANDLER
     OLD8SELDW ?
.CODE
ENDP NewInt8

PROC InitNew8
     mov ax,204h
     mov bl,8
     int 31h ; get interrupt addr ess
     mov DS:OLD8OFFS,edx ; store  old interrupt vector
     mov DS:OLD8SEL,cx ; .CONST d ata needs DS Prefix!
...
ENDP InitNew8
```

Usually it makes no difference whether a variable is declared before or after its usage. Immediate memory access, `eg. mov [0],0100h`, `cmp ByteVar,0` etc must use an argument override if the variable is not defined:

wrong:

```
     mov AxIsZero,0
.DATA
     AxIsZero db ?
```

AxIsZero is not defined when assembling the immediate mov instruction. The assembler therefore doesn't know what type AxIsZero will be. You need an argument override:

right:

```
     mov byte ptr AxIsZero,0
.DATA
     AxIsZero db ?
```

or

```
.DATA AxIsZero db ? ; AxIsZero now defined
.CODE
     mov AxIsZero,0
```

Basically you should define data before using it. This makes reading of the code easier.

8.2 Addressing Data, Defining Labels and Procedures

8.2.1 Addressing Memory

The Pass32 supports the 32 bit address modes of the 80386 and newer processors. You should use square brackets to address with index or base registers. In both models 16 and 32 bit addressing is allowed. In the TINY model you must make sure, that the extended registers do not override a segment, they should not address offsets above 0ffffh! Expressions in the code segment are basically identical to those in the data segment (see 8.1.4 Data Expressions on page 72), but they are extended by the register addressing, i.e. additionally to variable identifiers or number expressions, the registers of the CPU can be used.

Some examples of 16 bit addressing:

```
mov al,bytevar+bx+di    ; you can leave square brackets out
mov al,[bx+di]          ; you must use square bra ckets!
mov dx,word ptr [byt evar+DI]
mov bx,es:[WordVar+S I]
```

You can leave square bracket out, if the first item makes sure that this is an addressing form. If the first item is a register, the assembler only recognizes the register, .eg: `mov ax,bx+di` makes `mov ax,bx` and `+di` which produces an error[1]. If you are using 16 bit address modes in protected mode programs - you must make sure, that the base address is below 64K! This might happen, for example, if you like to index a variable with the SI register and your program code exceeds 64K:

```
mov al,[ByteVar+SI]     ; forces an error, when the offset of ByteValue
                        ; is above 0ffff h !!!
```

Some examples of 32 bit addressing:

```
mov ax,es:[eax+ecx*4]
mov eax,dword ptr bytevar+edi
mov [esp+8],edi
mov ax,[eax+edi*8+01 11h]
cmp word ptr [eax+edx*2],0
```

1.This is due to speed reasons. The instructions, which occur statistically at most use at least one register.

You can use any extended register as base and index register. The scale factor for the index register must be 2,4 or 8. A segment override should be placed before the square brackets / before the expression.

Square brackets or a variable identifier refer to a memory location. To load a value direct into a register don't use square brackets:

```
mov dx,01111b and 47h or 1000000b
mov ecx,4*1000h + 2*0100h +88h
```

Pass32 as well allows to load character or string values in a register:

```
mov al,'$'
mov eax,'.COM'
mov ax,'DX'
```

Against the TASM convention, the string is stored in the register, like it would be stored in memory; so you can use this method to scan the memory for string expressions:

```
mov eax,'.COM'
mov edi,OFFSET Parameter_1 Sc anParameter
cmp [edi],eax
je ok_is_COM_File
inc edi ...
jmp ScanParameter
```

You can also calculate string expressions:

Upcase:

```
mov al,'a'-'A'          ; al = 32
mov edx,'HEY!'+' '      ; edx = 'hey!'
```

8.2.2 Defining Labels

Labels are part of the program code. They define the beginning of a loop, an address of a procedure, the entry point of the code, etc. In the TINY model a label is a 16 bit offset, in the FLAT model a label is a 32 bit offset.

A label is defined by a colon at the end of the name. The label name must not begin with a digit 0..9. It can consist of letters A-Z,a-z, digits 0-9, special characters as _@#., some examples:

```
.PUBLIC ProgramStart:
....
ProgramExit:
     mov ah,4ch
     int 21h

FirstChoice:
....
SecondChoice:
....
@loop:
     mov es:[ecx*2+0b800h],al
loop @loop

END ProgramStart
```

The END [label | procedure] defines the entry point of a program. Without this directive any assembler source will start at the first .CODE instruction or at the first .CONST entry. The start label / procedure must be defined before the END directive. The .PUBLIC directive defines a label globally.

Labels are always treated as local in the procedure they are used. You can use the same label name in different procedures:

```
PROC ProcA
...
@Start:
...
     je @Start
ENDP ProcA

PROC ProcB
...
     je @Start
...
@Start:
ENDP ProcB
```

If you want to use a label globally, for example if you want to jump from one procedure into a specific offset in another procedure, you have to define this Label as public:

```
PROC ProcA
...
```

```
.PUBLIC Exit:
     ...
ENDP ProcA

PROC ProcB
...
     jmp Exit
...
ENDP ProcB
```

Without the directive `.PUBLIC` the instruction `jmp Exit` would produce an error. You can't use a labelname more than once inside a procedure, you can't use the same labelname for different global labels and you can't use the name of a public label for a local label:

```
PROC ProcA
@loop:
@loop: ; wrong, dup licate label
.PUBLIC Exit:
ENDP ProcA

PROC ProcB
@loop: ; right, use d for the first time
Exit: ; wrong, 'Exi t' already defined as public!
ENDP ProcB
```

You can use the `.EXTERN` directive to make a label public after the label definition:

```
PROC ProcA ...
Exit: ...
ENDP ProcA

.EXTERN Exit
PROC ProcB ...
jmp Exit ...
ENDP ProcB
```

Note, Pass32 will make the first label which is found public! You should prefer the `.PUBLIC` method, as it is not ambiguous!

8.2.3 Definig a Procedure

You can define a procedure with the PROC directive. With PROC, the name and attributes of a procedure are defined. The procedure attributes NEAR, FAR and START are optional. There are two ways of using the PROC directive:

```
PROC Main ; same as
Main PROC
```

Or, the same definition with attributes:

```
PROC Main NEAR START
Main PROC N EAR START
```

A procedure should end with the ENDP directive. This directive is especially necessary in combination with code optimization! There are again two ways of using the ENDP directive:

```
ENDP Main ; same as
Main ENDP
```

A procedure can have three attributes: NEAR, FAR or START. Generally all procedures of the main program are NEAR procedures. This means, all procedures share the same code segment. A FAR procedure is a procedure outside the main code segment. If you use a **DLL**[1], for example, the procedures of the **DLL** are resident in another code segment: therefore all procedures in the **DLL** are defined as FAR procedures.

NEAR is the default attribute for TINY or FLAT model procedures. You can explicitly force the assembler to generate a near procedure with

```
PROC Main NEAR
```

a far procedure with

```
PROC Main FAR
```

Far procedures must be defined before they can be called. If you are working with far procedure and you want a forward reference, you can use the directive .FAR. All procedure calls are now treated as far calls, so every far procedure has a forward definition. If you don't want far procedures to be treated like that, you can disable the function by .NOFAR.

1.This refers to the Pass32 DLL type and not to a Win32 DLL, which is defined as near.

The attribute START produces an implicit call to this procedure at the beginning of the program (before the start label is called). For example **DOSX.INC** uses such a procedure type to install the new DOS interrupt handler, to create extended DOS functions. A procedure with the attribute start must be defined as a near procedure, the attribute START is not valid in the DLL / OVL model. In these models, Pass32 generates a warning message, that these procedures are not called implicitely by program execution.

You can define a variable type as a jump target. The variable type can be a 16 bit offset = WORD PTR (TINY only), a 16:16 pointer = DWORD PTR (TINY only), a 32 bit offset = DWORD PTR (FLAT ONLY) and a 16:32 pointer = FWORD PTR (FLAT ONLY).

If you have two alternative procedures and you want to use a variable identifier to access them, this could be done like this:

```
.MODEL FLAT
.DATA GraphPutPixel DD ?
.CODE
PROC VGAPutPixel NEAR ...
ENDP VGAPutPixel

PROC SVGAPutPixel NEAR ...
ENDP SVGAPutPixel

START: ....
     mov eax,offset VGAPutPixel
     mov GraphPutPixel,eax ; initialize one of the Procs ...
     call GraphPutPixel ...
     call GraphPutPixel ...
END START ; define START as entry point END
```

Conditional jumps or jumps to a forward reference can be optimized with the SHORT directive, if the target is in the next 127 bytes.

For some instructions you must explicitly distinguish between 32 bit and 16 bit instructions:

The loop instruction can be used with the CX and ECX registers. Usually Pass32 uses loop with the CX register, as this is in most cases more efficient (small loops). To use the ECX register you must use the (pseudo) instruction loopd:

```
.CODE
     mov cx,10
     @loop:
     mov al,[esi]
     inc esi
```

```
     loop @loop ; repeat ten times ...

     mov ecx,0b80000h+100h
     @@loop: mov fs:[ecx],al ; from b80100 to b80001
     loop @@loop ; repeat  100h times (low word part of ECX) ...

     mov ecx,10000h
extloop:
     mov fs:[ecx*4+esi]
     loopd extloop ; repe at 65536 times (ECX)
```

The `push` immediate instruction can push a 16 bit or a 32 bit immediate. We use again the suffix 'D' to distinguish the instructions:

```
     push 1234h         ; pushes an immediate of 16 bit in the TINY model
     push 1234h         ; pushes an imme diate of 32 bit in the FLATmodels
     push 12345678h     ; pushes an imm ediate of 32 bit in the FLAT models
                        ; you'll get an error in the TINY model
     pushw 1234h        ; pushes an imme diate of 16 bit in all models
     pushd 12345678h    ; pushes an imme diate of 32 bit in all models
```

Note: Pass32 always creates the push instruction according to the current model: In Tiny a 16 bit push, in FLAT a 32 bit push, to use a different operand size (or to use the operand size model independend) use the suffix 'W' for 16 bit and 'D' for 32 bit.

The `JCXZ` and `JECXZ` instructions already show their difference in the instruction name!

8.3 Pre-processor, Macros and Conditional Assembly

The pre-processor generates the source code for the assembler with all modules included.. Thus, the pre-procesoor reads the source code first. The pre-processor can change the source code before the actual assembling begins. Macros are typically expanded by the pre-processor. The pre-processor actually understands only the following commands:

.EQU	replace
.TYPE	enumeration type
.SMART	code optimization
.INCLUDE	include another source module
.INCLUDEDIR	set the search path for include
.MACRO	Macro definition
.LOCAL	to define a local identifier inside a macro
.DOSX	Dos Extender Defintion
.UCU	Pro32: Target for uncomercial use only
.NM	Pro32: No further messages during program loading
.NB	Pro32: CRTL+C/CRTL+BREAK is deaktivated
.PLUG	Pro32 Gold: Add plug-in
.KEY	Pro32 Gold: Define License Key

Tab. 8.7 *Pre-processor Commands*

8.3.1 The .EQU Directive

We can define a variable for values and text strings with the `.EQU` directive. For example if we want a special color design for our program, we can use the `.EQU` directive:

```
BackColor .EQU 0
WindowFrameColor .EQU 14
ScriptColor .EQU 15
GraphColor .EQU 7
```

It is easier to change the `.EQU` command at the beginning of our source, than changing all colour attributes in the source, .eg `mov al,14.` The `.EQU` command is a pre-processor command. Actually all text strings `'BackColor'` are replaced by the text string `'0'`. You can not define a storage with this! You can not store a value in `BackColor` for example, this would be like: `mov BackColor,1` ===> `mov 0,1` (!!!). As the `.EQU` directive forces a string replacement you can even replace text strings:

```
HelloMesg .EQU 'Hello, World!',0
```

Note, the `HelloMesg` is no variable and has no offset! You can use the replacement in a data definition:

```
Message db HelloMesg
```

You can even replace instructions with the .EQU directive:

```
     ClearRegs .EQU xor eax,eax//xor ecx,ecx//xor ebx,ebx//xor edx,edx
.CODE
     ClearRegs
     mov ah,4ch
     int 21h
END
```

The // in the replacement is understood as a line feed. Note, that the name can be 40 characters long and the replacement 60 characters. The total number of .EQU replacements is 640.

You can declare multiple .EQU replcaements with the .TYPE directive. The .TYPE directive can be used to declare an enumeration type. For Example:

```
     .TYPE (Red, Green, Blue)
```

The .TYPE definition assigns the strings RED the value 0, GREEN the value 1 and BLUE the value 2. Basically .TYPE generates .EQU replacements for integer expressions. For every item, the value is increased. You can define a value explicit:

```
     .TYPE (Red=1, Green, Blue=4, Black=0fh)
```

Here, the result would be: RED = 1, GREEN = 2, BLUE = 4, BLACK = 15.

The .EQU identifiers are all replaced, when the regular assembly begins. You should not define with the .EQU directive in a conditional assembly!

8.3.2 Including Assembler Modules

With the .INCLUDE directive you can include another source file. Note that the assembler includes the same file only once. Two files are equal for the assembler, if the name and extension are equal. You can include with the correct path:

```
     .INCLUDE E:\PASS32\INC\DPMI.INC
```

And without a path:

```
     .INCLUDE Module
```

The extension **.ASM** is added if the name is without extension. Without a path, the file is searched in the current directory and then in a parallel directory of the **PASS32.EXE** directory called **\INC**. Usually the assembler is located in the **\BIN** directory, all include files in the **\INC** directory. You can define an alternative directory for include files with the .INCLUDEDIR directive:

```
.INCLUDEDIR C:\TEST\INC
```

Now, Pass32 will search in this directory before searching any other directory. So Pass32 will look in the current directory and then in the directory defined with .INCLUDEDIR.

Unlike to C/C++, any model is included only once, so you don't need to test if a module is already included!

8.3.3 Defining Macros

Another powerful tool of the preprocessor is the .MACRO directive. A macro is a storage for assembler commands, which can be used several times in your code. A macro is a kind of subroutine. But in the opposit of a procedure defined with PROC, the code inside a macro definition is directly placed into your code. The best way to explain the macro method is to show an example:

```
.MACRO Exit(ExitCode)
     mov al,ExitCode
     mov ah,4ch
     int 21h
ENDMACRO
```

The usage of the macro in your code can be for example:

```
.CODE
START: ...
     Exit(3)
END START
END
```

This example will be expanded by the preprocessor to the following code:

```
.CODE
START: ...
     mov al,3
     mov ah,4ch
     int 21h
```

```
END START
END
```

Wherever we use the macro `Exit` in our code, these three instructions will be expanded. We can see, that the parameter `Exitcode` is replaced by its value. We could use for example `Exit(Ah)` and the result would be `mov al,ah` / `mov ah,4ch` / `int 21h` - parameters are similar to an `.EQU` expression textual replacements, but they are only valid inside the macro.

The definition of a macro generally has the following syntax:

```
.MACRO name [(parameter[,parameter] )] ENDM [name] | ENDMACRO [name]
```

You can end a macro with `ENDM` or `ENDMACRO`.

As we learned, when a macro occurs in your source, it will be replaced by its definition contents. This can cause problems, when you are defining labels, or data storages in your macro, because all labels or data definitions will have the same name. We need the `.LOCAL` directive to use a label locally inside a macro. The `.LOCAL` directive simply extends a symbol name by an '@' following a macro specific hex number. This method guarantees, that every label / data definition has an individual name. The following example uses the `.LOCAL` directive to define a label:

```
.MACRO WRITE(stringoffs)
     mov edi,stringoffs
.LOCAL @start:
     mov dl,[edi]
     cmp dl,0
     je short .LOCAL @end
     mov ah,2
     int 21h ; display s tring char by char
     inc edi
     jmp .LOCAL @start
.LOCAL @end:
ENDMACRO


.DATA mesg db 'Hello, World!',0
.CODE
     Write(offset mesg)
     Exit(0)
END
```

As `.LOCAL` only extends the label name, we must use the directive also when we refer to the definition.

You can not nest macros in the definition, but you can use a macro inside other macros, when the macro is already known to Pass32. The number of macros in total is limited to 256 macros, the number of words in a macro is limited to 256. As macros are expanded by the preprocessor, the number of macros does not influenze the amount of memory for the main assembler pass.

Note, that a macro definition doesn't produce code, even, if you place the definition inside your code segment. Therefore, you won't get an error, as long as you don't use the macro, if the macro definition is wrong. Another thing is the error report: Pass32 dosen't remember the macro definition in the assembler pass, so it can't display the line of the error. If a macro contains an error, Pass32 reports:

```
Error in Macro: <name> : <instruction>
```

8.3.4 Conditional Assembling

Conditional assembling is not a part of the pre-processor, as the pre-processor can not understand symbol names and identifiers.

If you want to test if a module is included, you can use the `.IFM` directive. The `.IFM` directive is a part of the `.IF` directives. The `.IF` directives are:

`.IF expr`	tests an expression for unequal zero
`.IFM module`	tests, if a module is included
`.IFPM`	tests, if the target is for protected mode
`.IFE expr1,expr2`	tests, if expr1 equal expr2
`.IFS expr`	tests, if expression is a string
`.IFR expr`	tests, if expr is a register (AL..DH, AX..SI, EAX..ESI, CS..SS)
`.IFR8 expr`	tests, if expr is an 8 bit register (AL..DH)
`.IFR16 expr`	tests, if expr is a 16 bit register (AX..SP)
`.IFR32 expr`	tests, if expr in a 32 bit register (EAX-ESP)
`.ELSE`	alternative assembly
`.ENDIF`	end of conditional assembly

Tab. 8.8 *Conditional Assembly*

All .IF directives (except .IFPM) can be used in the opposite way by writing:

```
.IF NOT expr
.IFR NOT R16 expr
.IFE NOT expr1, expr2
```

With the .IFPM you can create modules for real and protected mode. You can make sure that a module is only used for real mode:

```
.IFPM
.DISPLAY Modul Only for Real Mode!
.ERR ;force an error
.ENDIF
```

The .IF directive tests if an expression is unequal to zero. You can test if a variable is defined, if an .EQU definition has been made:

```
.IF NOT VideoBuffer
.DATA? VideoBuffer DB 256*1024 DUP(?)
.ENDIF

OVL_ERROR .EQU 1

.IF NOT OVL_ERROR
call printerror
.ENDIF
```

With .IFS, .IFR and .IFE, you can write multiply functional macros. An example for a multiple write string macro could look like this:

```
.MACRO SWriteLn(String)
.IFS String
.DATA .LOCAL mesg DB String ,0
.CODE
     mov edi,OFFSET .LOCAL mesg
.ELSE
.IFR String
     mov edi,String
.ELSE
     mov edi,offset String
.ENDIF
.ENDIF
     call SystemWriteLn
ENDMACRO
```

Now this macro can be called with three different types of parameters:

```
 .DATA mesg db 'Hello, World!',0
.CODE
     mov eax,offset mesg
     Swriteln(eax)
     Swriteln(mesg)
     Swriteln('Hello, World!')
```

The Macro library **SYSTEM.MAC** has already included this macro definition for the `WriteLn` macro.

8.4 The OVL model

The OVL model is a model defined for protected mode use. The idea is simple: Several procedure can share the same memory; at run time you can load special driver functions to adapt the software to the given hardware. The first idea of course is nearly unimportant: If you can use so much memory, procedures don't need to share memory. The second idea is far more important: To access several music boards, graphic adapters etc. you can add several different optimized OVLs to your code. You can set-up the program, so that only the best fitting OVL is load and used. The OVL model of the Pass32 is very simple. The OVL code is load to a given offset in the code and data segment. Usually somewhere in the heap. A short 256 byte long interface tells the assembler at run time where the different routines are stored in memory.

This is the syntax of an OVL source:

```
.MODEL OVL
.INTERFACE
.ORG 50000h ; OVL start address in the heap

PROC FirstOVLProcedure OFFSET OvlProc1
PROC SecondOVLProcedure OFFSET OvlProc2

.DATA Mesg db 'OVER LAY load!',0
.CODE
OvlProc1 PROC FAR
     mov edi,offset Mesg ...
OvlProc1 ENDP

OvlProc2 PROC FAR . ..
OvlProc2 ENDP END
```

The .INTERFACE directive is the start of the 256 byte long OVL interface. The .ORG directive defines the address where the overlay is loaded to. The PROC directive defines a public procedure. This identifier is public and can be called from the main program. The OFFSET identifier makes a connection between the OVL procedure and the public procedure identifier. The names could be the same of course.

To use an OVL, you include the OVL source interface part into your code. Independent of the OVL size, the 256 byte long interface will be part of the code.

A program which uses the OVL could look like this:

```
.MODEL FLAT
.INCLUDE TSTOVL.ASM
.DATA ovlname db 'TSTOVL.OVL',0
.CODE

     mov esi,offset tstovl ; OFFSET TO DATA BUFFER TSTOVL
     mov edi,offset ovlname ; OFFSET TO Filename
     call initovl ; to initialise the OVL
     call loadovl ; to load the OVL
     call FirstOVLProcedure ; use the OVL procedure after loading
     call SecondOVLProcedure
     mov ah,4ch
     int 21h
END
```

The loadovl procedure is part of the **OVLSYS.INC** file, which is part of the Pass32 assembler. If any overlay module is included, this module is automatically included. You must make sure that the OVL is loaded.

The loadovl procedure will set the carry flag, if the OVL could not be load. You can define an identifier called OVL_ERROR, the loadovl procedure will then terminate the program with an error message:

```
     OVL_ERROR .EQU 1
```

The OVL interface structure has the following format:

00-03	load offset: address of the overlay
04-1F	reserved
20-23	offset of first ovl procedure
24-25	selector of first ovl procedure
26-29	offset of second ovl procedure
...	
FA-FD	offset of 37th ovl procedure
FE-FF	selector of 37th ovl procedure

Tab. 8.9 *The header of an overlay*

The `loadovl` procedure does the following jobs:

- load the interface and determine the load offset
- load the ovl
- alter the selectors in the interface

The `initovl` procedure simply initialises all procedure calls with a far return. You should run this procedure on all overlays at the beginning of the program.

If procedures share the same memory, you should use the first procedure as an id procedure, which tells the program which overlay is load at the moment.

8.5 The DLL model

The DLL model is a model defined for protected mode use only. It's idea is similar to the OVL model, but far more effective. A DLL is a library file containing several procedures, which can be load at run time. The library is no part of the program code segment; a DLL therefore does not limit the heap memory for the program. You can use single procedures of a DLL and you can write whole programs as DLL to use the memory more efficient: If you want to write a program which needs for example 8 MByte memory and you want to make sure that your program can even run if only 3 or 4 MByte memory are available, you can split your program in different DLLs and a global data area. If the computer offers enough memory the program can load all DLLs at the beginning. If not, it can for example hold only one DLL in memory at the same time. Besides, a DLL is usually stored in real memory, if enough real memory is available. A DLL again consists of a 256 byte long interface and its code. The program entry of the DLL is usually at the offset 00000100h. The first 100h bytes represent the DLL interface.

00-01	DS selector of DLL data
02-03	ES - Video Selector
04-05	Zero Selector
06-07	Real Mode File Buffer Selector
08-09	Real Mode File Buffer Segment
0A-0D	Memory size for the DLL
0E-11	Memory handle
12-13	CS (DLL Code descriptor)
14-17	Linear Address of DLL
18-19	DS of the main application
1A-1F	reserved
20-23	offset of first dll procedure
24-25	selector of first dll procedure
26-29	offset of second dll procedure
...	
FA-FD	offset of 37th dll procedure
FE-FF	selector of 37th dll procedure

Tab. 8.10 *The header of a Pass32 DLL*

You can load those data in the DLL via CS. Note, that when the DLL is called usually DS is the data descriptor of the main program. You can either use this descriptor to access global data, or save the descriptor and load the DLL data descriptor from CS:[0]:

```
     push ds
     mov ds,word ptr CS:[ 0] ; set DLL - DS
...
     pop ds
```

This is the syntax of a DLL source:

```
.MODEL DLL
.INTERFACE

PROC MainDLLProcedure OFFSET DLLMain
PROC FirstDLLProcedure OFFSET DLLProc1
PROC SecondDLLProcedure OFFSET DLLProc2
```

```
.CODE
DLLMain PROC FAR
...
ENDP DLLMain

DLLProc1 PROC FAR
push ds
mov ds,word ptr CS: [0] ; set DLL - DS
mov edi,offset Mesg
...
pop ds
ret
DLLProc1 ENDP

DLLProc2 PROC FAR . ..
DLLProc2 ENDP
END
```

The .INTERFACE directive is the start of the 256 byte long DLL interface. The PROC directive defines a public procedure. This identifier is public and can be called from the main program. The OFFSET identifier makes a connection between the original DLL procedure and the public procedure identifier. The names can be the same, of course.

The use of a DLL is the same as the use of an overlay: you include the DLL interface part into your code. Independent of the DLL size, the 256 byte long interface will be part of the code.

A program which uses the DLL example from above could look like this:

```
.MODEL FLAT
.INCLUDE TSTDLL.ASM

.DATA DLLname db 'T STDLL.DLL',0
.CODE
     mov esi,offset tstdll ; OFFSET TO DATA BUFFER TSTDLL
     mov edi,offset dllname ; OFF SET TO Filename
     call initDLL ; to initialise  the DLL
     call loadDLL ; to load the D LL
     call DLLMAIN ; use a DLL procedure after loading
     call SecondDLLProcedure ...
     call FirstDLLProcedure
     mov esi,offset tstdll ; OFFS ET TO DATA BUFFER TSTDLL
     call FreeDLL ; to free the D LL ...
     mov ah,4ch
     int 21h
```

```
END
```

The procedures `LoadDLL`, `InitDLL` and `FreeDLL` are part of the **`DLLSYS.INC`** file, which is part of the Pass32 assembler. If any DLL interface is included, this module is automatically included. You must make sure that the DLL is loaded or at least initialised, before using any DLL procedure.

The loadovl procedure will set the carry flag, if the DLL could not be found. You can define an identifier called `DLL_ERROR` and the `loadDLL` procedure will terminate the program with an error message:

```
DLL_ERROR .EQU 1
```

The DLL interface structure has the same format as shown in Tab. 8.10. When the DLL is generated, only the memory size (offset 0ah) is defined in the header.

The first argument for the `LoadDLL` procedure is the offset to the zero terminated filename string in the `EDI` register. The second parameter is the offset to the interface structure in the `ESI` register. The interface structure has always the same name as the filename of the DLL source file without extension. If your DLL source file is called TSTDLL, the interface structure has the name TSTDLL[1].

Note, the `LoadDLL` procedure will search in the current directory and in the path for the DLL filename.

The `InitDLL` and the `FreeDLL` procedure both need the second parameter TSTDLL. If you are good at assembler you can use or even modify the **`DLLSYS.INC`** or **`OVLSYS.INC`** file to create your own DLL/OVL handling.

8.6 Debugging and Code Optimization

When you already wrote your first 32 bit program, you'll know, that debugging in protected mode is hard. The Turbo Debugger from Borland, Inc. for example, hangs when the processor is switched into protected mode. The current version of the Pro32 Dos Extender does not support debugging, so we need a special method for the debugging. The easiest method (and safest method - as it is compatible to Dos, Windows, Linux-DPMI and WinNT) is to include the debugger in your program - and this is exactly, what the `.DEBUG` directive does.

1.Filenames are treated by Pass32 to be 8 characters long

8.6.1 The integrated debugger

When the pre-processor discovers a `.DEBUG` directive, he appends the **`DEBUG.INC`** module to the source code. Every instruction which is debugged is extended to the form:

```
PUSH DebugInfoOffset
CALL DebugProc [ instruction  ]
```

The debugger (**`DEBUG.INC`**) is compared to a real debugger, e.g. Turbo Debugger, Pro32 Debugger, only small utility, which displays the contents of all registers, segment registers and flags. The debugger waits for a keystroke before the next instruction is executed. It allows to set break points and to alter the register contents. Here is a list of all commands:

`[SPACE]`	Execute next instruction : the program is executed until the next debugged instruction occurs.
`[ESC]`	Run until the next debugged RET/RETF instruction occurs
`[RETURN]`	Run until the next debugged JMP/CALL (INC/DEC) instruction occurs
`[BCK SPC]`	Run until the breakpoint occurs
`[a][b][c][d][e][f]`	display hex dump at [EAX], [EBX], [ECX], [EDX], [EDI], [ESI]
`[h]`	to display hex dump every time (hold)
`[A][B][C][D][E][F]`	to alter the register contents[a] of EAX, EBX, ECX, EDX, EDI, ESI
`[Cursor Up]`	decrement hex dump by 10h
`[Page Up]`	decrement hex dump by 100h
`[Cursor Down]`	increment hex dump by 10h
`[Page Down]`	increment hex dump by 100h
`[Crtl]+C`	terminate the program
`[Crtl]+B`	set a breakpoint[b]

a. Note: altering a register value might force an exception, if the register is used as index or base register.
b. Note, that only one breakpoint may be set. The breakpoint is marked by an '*' before the debug line information.

Tab. 8.11 *Debugger Functions*

You can enter a new hex value. The resulting value for the register is the displayed value. If the former value of EAX = 12345678h, and you alter only the first two digits to zero, the result is 00345678h. Pressing [ESC] during the input restores the old value. The input is finished either with [RETURN] or when all 8 digits are entered. During the input only the keys 0..9, a..f, A..F, [ESC],[BCK SPC],[RETURN] are valid.

When the program runs with, you can interrupt the execution by hitting any key.

A 'debugged instruction' is an instruction between the .DEBUG and the .NODEBUG directive.

Within the .DEBUG and the .NODEBUG directive you can go step by step through the program. If you want to set a breakpoint at a certain instruction, you can set the .DEBUG here; either around a single instruction or around a group, a procedure, etc. To generally debug a source file you can use the option **-D**.

The debugger supports video mode swapping. If the program to debug runs for example in graphics mode, you can enable video mode swapping by setting the debugvideo identifier with the .EQU directive:

```
debugvideo .EQU 1
```

8.6.2 The Debug File Format DMP

The DMP file format is actually a listing of the source code with additional offset infromation. The module **DLOADS.INC** is able to load the debug file format. The procedure DISPLAYSDEBUGSOURCE loads and shows the debug source file (program file name with the extension **.DMP**) at the offset address in EDI.

The **-MM** or **-DMP** option produces a debug file with the extension **.DMP**. This file contains the whole source code (including all submodules), with the offset address for every line. The output could look like this:

```
00000000: ;Pass32 DEBUG FILE (c) 1996 by Dieter Pawelczak
00000000: .MODEL TINY
0000010C: .DATA
0000010C: HelloMesg db 'Hello,World',10,13,'$'
0000011A: .CODE
00000100: START:
00000100: mov dx,OFFSET HelloMesg ; offset of the text string
00000103: mov ah,9 ; print string function number
00000105: int 21h ; dos call
00000107: mov ah,4ch ; terminate function number
00000109: int 21h ; dos call
0000010B: END START ; marks the entry procedure of the program
```

```
0000010B:
```

The first 8 bytes of a line contain the hexadecimal offset of the line. This offset information is analysed by **DLOADS.INC**.

A combination of the **-MM** option and the **-D** option (or the equivalent .DEBUG directive) automatically adds **DLOADS.INC** to the source code. The debugger realizes the presents of **DLOADS.INC** and displays - if possible the debug source file. (The debug source file must be in the current directory, when the first instruction is debugged!)

Try for example to debug the **DISPLAY.ASM** example (see part 2 for further information) with the command: **Pass32 DISPLAY -mm -d**

DLOADS.INC loads the whole debug source file into a buffer. This buffer is 256 Kbytes in size. If your debug source file is above 256 KBytes, you must alter the buffer in **DLOADS.INC**:

```
DebugFILEBUFFER DB 256*1024 DUP(?)
```

You can use the debugger as well in DLL or OVL files. When displaying the source code is enabled in combination with the OVL model, you must make sure that the main program allocates enough memory. The overlay will need about 260 KBytes more heap memory because of the source code file buffer. You can test for example debugging the **OVLTEST** demo (see part 2), with assembling the overlay **TESTOVL** with the debug option:

```
Pass32 testovl -ovl -d -mm
Pass32 ovltest -uc ovltest
```

You can test debugging an DLL source as well...

The advantage of the internal debugger is of course, that you can write your own debugger, or alter the given debugger. If you want to alter the debugger, you should save **DEBUG.INC** under another name and alter this file. To use this special debugger, you can use the .DEBUGFILE directive:

```
.DEBUGFILE Filename
```

Instead of **DEBUG.INC** the filename (standard extension **.ASM**) is used as debugger. The debug module is appended when a .DEBUG directive or the option **-D** is used (.DEBUGFILE itself does not append, but names the debugger module). Your debugger procedure must be called DEBUGPROC and must save and restore all registers and flags!

8.6.3 Usage of an external Debugger

If you want to use an external debugger (for example the Pro32 Debugger **PRODB32.EXE**), you should assemble with the **-MM** or **-DMP** option. The external debugger is then able to load the debug file, and display the source code, add watches etc.

If you want to use a disassembler, you can use **DISS32**. This is a simple 32 bit disassembler, which is part of the Pass32 package. The disassembler comes with complete source code and can be easily integrated in other applications.

8.6.4 Detailed Information - The Map File

To gain general information of the assembled program, we can create a map file: We assemble with the **-M** option. A map file shows all symbols which had been created during the assembling. A typical map file could look like this(eg. **HELLO1.MAP**):

```
Pass32 MAP FILE (c) 1996 by Dieter Pawelczak
SOURCE :HELLO1.ASM
DESTINATION:HELLO1.COM
PROGRAM TYPE DOS TINY
SEGMENT TYPE CODE

ENTRYPOINT :00000100
OFFSET TYPE NAME
00000100 LABEL START

SEGMENT TYPE DATA
OFFSET TYPE NAME
0000010C DATA HELLOMESG
```

8.6.5 Code Optimization

With the directive .SMART, .SMART1, .SMART2, .SMART3 or the equivalent options **-o**, **-os**, **-or**, **-oj**, you can enable code optimization. Code optimization covers two main aspects: optimization of the instructions and optimization of the linker.

Some optimizations are always done: For example if you work with the (E)AX register, the faster and shorter (E)AX instruction are used, when possible. Relative 8 bit jump instructions are used when possible.

With .SMART3, or the option **-oj**, Pass32 optimizes the jump instructions: Pass32 tries to assemble all jump instructions as short jump instructions, even if the target is unknown. This is exactly the optimization, a user can do with the SHORT directive...

With .SMART2, or the option **-or**, Pass32 optimizes the register instructions:

- any register immediate load with a zero constant will be replaced by the shorter register XOR register instruction.
- any immediate add,sub,cmp etc. with 16 or 32 bit registers/ memory locations and an immediate byte value will be replaced by the shorter rm16, imm8 / rm32, imm8 instruction.
- any instruction like MOV AL,AL will be removed completely (such an instruction might appear, when using macro parameters)

The .SMART1 (**-o1** option) optimization excludes all procedures from the linker, which are not explicitly called in the source file, addressed via the OFFSET directive or included with the .PUBLIC directive.

The .SMART directive (**-o** option) includes all other optimizations: .SMART1, SMART2. and .SMART3. Furthermore, with .SMART Pass32 scans through the source file (similar to a debugger) and excludes all procedure which are not called or addressed via the OFFSET directive in the assumed course of the program at run time. This is the best optimization, because only actually used procedures remain part of the program.

Three main rules for the use of .SMART or .SMART1:

- you must define every procedure correctly between the PROC and ENDP directive.
- you should not use the optimization until the unoptimized code is assembled correctly.
- you should not optimize and debug at the same time; the optimize function tries to shorten the code, whereas the debug function expands the code (actually, the debug option even disables some optimizations like short jumps, etc.)

The optimization with .SMART and .SMART1 (**-o** and **-o1**) might take until 30% longer than usual assembling, because of a third pass. The other optimizations with .SMART2 and .SMART3 (**-or** and **-oj**) might take 1-2% longer.

Appendix

A The Pass32 Assembler

A.1 Operators

`()`	Marks priority evaluation
`[]`	Marks memory location
`*`	Multiplies two integer expressions
`+`	adds to integer expressions
`-`	sign of integer expression
`-`	subtracts two integer expressions
`/`	divides two integer expressions
`%`	modulo of two integer expressions
`and`	logical and of two expressions
`not`	logical invert of expression
`or`	logical or of two expressions
`xor`	logical xor of two expressions
`byte ptr`	forces memory location to be byte size
`word ptr`	forces memory location to be word (2 bytes) size
`dword ptr`	forces memory location to be doubleword (4 bytes) size
`qword ptr`	forces memory location to be quadword (8 bytes) size
`tbyte ptr`	forces memory location to be 10-byte size
`fword ptr`	forces memory location to be 32-bit far pointer size
`size`	returns reserved byte count for the data identifier

A.2 Directives

`.ALIGN`	forces data alignment or code alignment - use in the segment
`.ALLWARN`	enables all warnings
`.BLOCK`	the following data are stored as one block (alignment ignored)
`.CODE`	the following instructions belong to the code segment; assembler instructions following
`.COMMENT`	marks the start of a comment
`.CONST`	the following instructions belong to the code segment; constant data definitions following

.DATA	the following instructions belong to the data segment; data definitions following
.DATA?	the following instructions belong to the uninitialized data segment; indeterminate data definitions following
.DEBUG	use debugger for the following instructions
.DEBUGFILE	use other debugger module instead of DEBUG32.INC/DEBUG.INC
.DISPLAY	displays message during assembling
DB	allocates and initializes a byte storage for data and code
DD	allocates and initializes a doubleword (4 bytes) storage
DF, DP	allocates and initializes a 32-bit far pointer (6 bytes) storage
DQ	allocates and initializes a quadword (8 bytes) storage
DT	allocates and initializes a ten bytes storage
DW	allocates and initializes a word (2 bytes) storage
DUP	duplicate storage
.ELSE	alternative conditional assembly block - part of the
.IF	directive
END Name	specifies the entry point of the program
END	marks the end of the assembler source
.ENDIF	end of conditional assembly block - marks the end of the .IF directive
ENDP	marks the end of a procedure
EMDM	marks the end of a macro
ENDMACRO	marks the end of a macro
.EQU	definition for a replacement
.ERR	forces the assembler to generate an error
.ERROR	forces the assembler to generate an error
.EXTERN	marks identifier of other module as public - marks label of other procedure as public
.FAR	all procedure calls are treated as far (forward far calls)
FAR	marks a procedure as far
.IF	initiates a conditional assembly, expression must be unequal 0
.IFE	initiates a conditional assembly, expression must be equal
.IFM	initiates a conditional assembly, module must be included
.IFPM	initiates a conditional assembly, target must be for protected mode
.IFR	initiates a conditional assembly, expression must be a register
.IFR R8	initiates a conditional assembly, expression must be an 8 bit reg.
.IFR R16	initiates a conditional assembly, expression must be a 16 bit reg.

.IFR R32	initiates a conditional assembly, expression must be a 32 bit reg.
.IFS	initiates a conditional assembly, expression must be a string
.INTERFACE	marks the start of interface definitions for DLL/OVL type
.INCLUDE	includes assembly file, any file will be included only once
.INCLUDEDIR	sets directory to search for include files
.INVOKE	to call a win32 API function (only Win32 model)
.LOADBIN	links binary file directly into program code
.MACRO	defines a MACRO
NEAR	marks a procedure as near
.NOALIGN	no data alignment
.NOBLOCK	marks the end of a data block
.NODEBUG	disables debugger: no debugging beyond this line
.NOFAR	far calls are treated individually (no forward reference)
.NOWARN	disables warnings
.MODEL	defines the memory model TINY/FLAT/OVL/DLL
.ORG	defines the code segment start
.OUT	displays message during assembling
PROC	defines procedure [attributes FAR / NEAR / START]
RS	allocates and initializes a single real constant (4 bytes)
RD	allocates and initializes a double real constant (8 bytes)
RT, RE	allocates and initializes an extended (temp) real constant (10 bytes)
SHORT	uses a 8 bit rel. jump instruction
.SMART	enables all code optimizations
.SMART1	enables code optimizations for subroutines
.SMART2	enables code optimizations for register instructions
.SMART3	enables code optimizations for jump instructions
START	marks a procedure as a start-up procedure
.TYPE	multiple .EQU declaration for enumeration types
.USE16	code segment attribute is 16 bit - default model TINY
.USE32	code segment attribute is 32 bit - default model FLAT
.WARN	forces the assembler to generate a warning message
.WIN32	to insert a win32 API function (only Win32 model)

A.3 Extender/Linker Variables

.MEM	specifies the mainmemory variable (FLAT/WIN32 model)
.MIN	specifies the minimum amount of memory needed to run the program
.MAX	specifies the maximum amount of memory used by the program

`.NM`	disables extra messages during loading
`.NB`	disables Pro32 break function
`.PLUGIN`	forces the dos extender to load plugin
`.STACK`	defines the stack size in KBytes (FLAT/WIN32 model)
`.UCU`	disables 1 sec loop for copyright message (for uncommercial use only)
`.DOSX`	to load other dos extender as stub

A.4 Pass32 Arguments

The general syntax for the Pass32 Assembler is:

```
PASS32 Filename[.ASM] [-OPTIONS]
```

or

```
PASS32 Filename[.ASM] [/OPTIONS]
```

Here is a list of the Pass32 options:

- Linker options:

`-t`	create com file (.COM) in combination with the TINY model
`-f`	create flat model binary (.BIN) in combination with the FLAT memory model
`-w`	create Win32 PE file (.EXE)
`-ovl`	create flat model overlay (.OVL)
`-dll`	create flat/Win32 model DLL (.DLL)
`-out``:name`	specify output (.EXE/.COM) file name

- Assembling options:

`-a`	enable all warnings
`-nw`	enable no warnings
`-e`	do not halt on first error
`-o`	optimize maximal
`-os`	optimize for size (exclude unused procedures)
`-or`	optimize instructions
`-oj`	optimize short jump instructions
`-i``:name`	use directory name to search for include files
`-im``:name`	include assembler file in source

-s silent: no output during the assembling

-spp skip pre-processor: assemble without using the pre-processor

- Debugger options:

-m create map file: A file with the extendion .MAP is created.
This file displays all symbols created during the assembling.

-mm / -dmp create debug file: A file with the extendion .DMP is created.
This file displays the whole source (including all sub-modules) with the corresponding offset address for each line.

-d add debug information: Debug information is added to the executable file[1].
A combination of -mm and -d allows to display the source code during debugging...

-error:HHHHH displays error at offset `HHHHH` (hex): The source is assembled without creating an executable program file. When the error address is found, the corresponding source line will be displayed.

- Extender/Linker options: (ignored when assembling with **-t -dll -ovl -f** option)

-mem:XXXXX -allocate xxxxx KBytes of XMS: The initial program code/data segment is xxxxx KBytes in size.
-allocate xxxxx Kbytes heap for the Win32 program

-min:XXXXX the program needs at least xxxxx KBytes of XMS[2].

-max:XXXXX -the program uses max. xxxxx KBytes of XMS.
-the maximum program heap for the Win32 program

-st:XXXXX sets the size of stack in KBytes (only FLAT and Win32 Model)

-nm display no message: Usually the Dos Extender displays the linear address of the program segment. When DPMI is emulated, the DPMI Version number is displayed. These messages won't be displayed when assembling with this argument.

-nb no control of `CRTL-Break` / `CRTL+C`: In emulated DPMI the Dos Extender takes control of `[CRTL]+[Break]` and `[CRTL+C]`.
To avoid program abortion with `[CRTL]+[Break]`/`[CRTL+C]` you should use this option.

1.Note: Debug information is added until the first `.NODEBUG` directive!
2. This option is equal to -mem

-c write core on error: For post debugging the emulated DPMI can generate a core image of the program. When an exception occurs, the current program code (and data) segment will be stored in the file CORE.COR, which can be analysed by a disassembler / debugger. (not supported by the Pro32 GOLD series)

-core:XXXXX write xxxxx KBytes core on error: Usually the core size is the .CODE and .DATA size. To store uninitialized data (.DATA?) you can alter the core size. (not supported by the Pro32 GOLD series)

-plug:NAME forces the Pro32 dos extender to load a plug-i (Pro32 GOLD).

-key:NAME To enter the license key for the Pro32 Dos Extender (Pro32 GOLD). The license key makes sure, that no other user can change the application settings with ProSet or any other Pro32 utility.

A.5 Run Time Library

A.5.1 SYSTEM.INC - Some useful system routines:

```
Function SystemKeyPressed: returns Zero Flag, if no key pressed
Function SystemGetKey: waits for key stroke, returns key in al
Function SystemGetMem: allocates eax bytes of memory on the heap
Function SystemFreeMem: frees eax bytes of memory on the heap
Function SystemMemAvail: returns largest memory block available on the heap
Function MathToString: 32 bit (EAX) number to string at Offset EDI
Function SystemPrint: prints at screen CX=X, DX=Y, EDI=string
Function SystemWrite: prints string EDI at cursor
Function SystemWriteLn: prints string EDI at cursor with LineFeed
Function SystemNewLine: creates line feed and scrolling at cursor
Function SystemClrSrc: clears the text screen
Function SystemReadLn: reads line from keyboard to string in EDI
Function SystemGotoXY: set cursor at CX, DX
Function SystemSound: turn speaker on with frequency in AX
Function SystemNoSound: turn speaker of
Function SystemDelay: delays execution for ms in AX
Function SystemExec: executes a program; name EDI, command line ESI
Function SystemGetRandom: returns a 16 bit unsigned random number, range in AX
Function GetParamStr: gets paramstr in EDI for parameter(AL)
Function GetEnvStr returns environment string to matching string in EDI
Function GetPrgDir: returns Directory of the program path in EAX
Variable TextColor:  DB    foreground color
Variable Background: DB    background color
Macro CheckCPU(CPUType) macro to check the CPU Type
```

A.5.2 SYSTEM.MAC - The Macro Version of SYSTEM.INC

```
Macro GetMem(Mem): allocates mem bytes of memory on the heap
Macro FreeMem(Ptr,Mem): frees mem bytes of pointer ptr
Macro Print(X,Y,String): prints string at screen
Macro Write(String): prints string at cursor
Macro WriteLn(String): prints string EDI at cursor with LineFeed
```

```
Macro ReadLn: reads line from keyboard
Macro Color(Text,Back) sets color variables
Macro GotoXY(x,y) set cursor
Macro Exit(ExitCode) terminate program
```

A.5.3 IO.INC - Disk Access

```
Function FileLength: EDI offset to filename, length in EAX
Function LoadFile: EDI offset to filename, ESI destination, result:size in EAX
Function OpenFileToRead: EDI offset to filename, returns handle in ebx
Function OpenFileToWrite: EDI offset to filename, returns handle in ebx
Function CloseFile: EBX handle
Function SaveFile: EDI offset to filename, ESI Source, ECX: size
Function BlockRead: EBX handle, ECX size, EDI dest., returns bytes read in EAX
Function BlockWrite: EBX handle, ECX size, EDI dest., returns bytes written
Function GetDir: copies directory name of DRIVE in DL to STRING in EDI
Function ChDir: changes to the directory in String EDI
```

A.5.4 STRING.INC - routines for zero terminated strings

```
Function strcat: appends string EDI with string ESI, returns new length
Function strupcase: converts string in EDI to upcase string, returns length
Function strcopy : copys string ESI to EDI, returns length
Function strlength: returns the EDI string length
Function strpos: Checks, if string ESI contains substring EDI
Function strcmp: compares string ESI with string EDI result in eax
```

A.5.5 GAME.INC - Joystick Access

```
Function GetXAxis : Returns value in EAX proportional to the X-Value
Function GetYAxis : Returns value in EAX proportional to the Y-Value
Function CheckFire : Zero-Flag set if Fire is pressed
Function CheckButton : Zero-Flag set if Fire is pressed
```

A.5.6 GRAPH.INC - VGA 320x200x256 graphic routines

```
Function INITGRAPH : initializes graphic mode
Function SETTEXTMODE : returns to textmode
Function PUTPIXEL : sets pixel at CX:DX with color BL
Function GETPIXEL : gets pixel color at CX:DX in BL
Function OUTCHAR : displays a single character at CX:DX, color BL, char AL
Function OUTTEXTXY : displays a string at CX:DX, color BL, string offset EDI
Function FILLBLOCK : fills block CX:DX to SI:DI with color BL
Function CIRCLE : draws a circle at CX:DX with radius SI and DI and color BL
Function LINE : draws a line between CX:DX, SI:DI with color BL
Function PUTOBJECT : displays a 'sprite' at CX:DX with size SI:BX and source EDI
Function PUTIMAGE : displays an image at CX:DX with size SI:BX and source EDI
Function GETIMAGE : gets an image from CX:DX with size SI:BX and image buffer EDI
Function COLOROUTCHAR : displays a single character in different colors
Function COLOROUTTEXTXY : displays a string in different colors (s. outtextxy)
```

A.5.7 MSDEMO.INC - PMode mouse driver for COM1/COM2 port

```
Function InitMouse:   Initializes pm mouse driver for COM1 or COM2
Variable MSX:     DW  current mouse X position
Variable MSY:     DW  current mouse Y position
Variable MSLEFT:  DB  status of left mouse button
```

```
Variable MSRIGHT: DB  status of right mouse button
Pointer  MSDRAW:  DF  far pointer for mouse draw procedure;
                   called with every mouse movement
```

A.5.8 `GRAPHIC.INC` - DLL based graphic driver library

```
Function PUTPIXEL    : ECX = X-Axis, EDX = Y-Axis, EBX = color
Function GETPIXEL    : ECX = X-Axis, EDX = Y-Axis, EBX = color
Function LINE        : ECX = X1, EDX = Y1, ESI = X2, EDI = Y2, EBX = color
Function RECTANGLE   : ECX = X1, EDX = Y1, ESI = X2, EDI = Y2, EBX = color
Function FILL        : ECX = X1, EDX = Y1, ESI = X2, EDI = Y2, EBX = color
Function CIRLCE      : ECX = X, EDX =Y, ESI = r1, EDI = r2, EBX = color
Function OUTTEXTXY  : ECX = X-Axis, EDX = Y-Axis, EBX = color, EDI = str. offset
Function PUTSPRITE  : ECX = X, EDX = Y, EBX = Size X, EDI = Size Y, ESI spr. offset
Function FLUSHBUFFER : copys activ buffer to screen
Function CLEARSCREEN
Function FASTFILL    : ECX = X1, EDX = Y1, ESI = X2, EDI = Y2, EBX = color
Function PUTIMAGE   : ECX = X, EDX = Y, EBX = Size X, EDI = Size Y, ESI img. offs.
Function GETIMAGE   : ECX = X, EDX = Y, EBX = Size X, EDI = Size Y, ESI img. offs.
Function PUT16x16   : ECX = X, EDX = Y, EBX = Size X, EDI = Size Y, ESI img. offs.
Function FLUSHWINDOW : ECX = X1, EDX = Y1, ESI = X2, EDI = Y2
Function INITGRAPH   : EAX = video mode
Function SETTEXTMODE : return to co80 mode
Function GETMAXX     : returns max value for X
Function GETMAXY     : returns max value for Y
Function SETPAGE     : EAX = page (0=screen, 1..n = BUFFER)
Function GETACTPAGE  : returns in EAX active page
Function LOADPALETTE : loads palette
Function WAITFORVERTICALRETRACE : EAX and EDX destroyed
Macros DrawText(X,Y,String,Color)
Macros DrawLine(X1,Y1,X2,Y2,Color)
Macros DrawRectangle(X1,Y1,X2,Y2,Color)
Macros DrawFill(X1,Y1,X2,Y2,Color)
Macros DrawImage(X1,Y1,X2,Y2,offs)
Macros DrawIcon(X1,Y1,offs)
```

A.5.9 `DOSX.INC` - Extended DOS Support for PRO32[1]

Include DOSX.INC at the beginning of your source to enable extended DOS support.

1.should not be used with Pro32 Gold - use DOSX plug-in instead.

A.6 Supported Assembler Instructions

```
AAA

AAD

AAM

AAS

ADC reg8, r/m8
ADC reg16, r/m16
ADC reg32, r/m32
ADC r/m8, imm8
ADC r/m8, reg8
ADC r/m16, imm8
ADC r/m16, imm16
ADC r/m16, reg16
ADC r/m32, imm8
ADC r/m32, imm32
ADC r/m32, reg32
ADC AL, imm8
ADC AX, imm16
ADC EAX, imm32

ADD reg8, r/m8
ADD reg16, r/m16
ADD reg32, r/m32
ADD r/m8, imm8
ADD r/m8, imm8
ADD r/m8, reg8
ADD r/m16, imm8
ADD r/m16, imm16
ADD r/m16, reg16
ADD r/m32, imm8
ADD r/m32, imm32
ADD r/m32, reg32
ADD AL, imm8
ADD AX, imm16
ADD EAX, imm32

AND reg8, r/m8
AND reg16, r/m16
AND reg32, r/m32
AND r/m8, imm8
AND r/m8, reg8
AND r/m16, imm8
AND r/m16, imm16
AND r/m16, reg16
AND r/m32, imm8
AND r/m32, imm32
AND r/m32, reg32
AND AL, imm8
AND AX, imm16
AND EAX, imm32

ARPL r/m16

BOUND r/m16
BOUND r/m32

BSF reg16, r/m16
BSF reg32, r/m32

BSR reg16, r/m16
BSR reg32, r/m32

BSWAP reg32

BT r/m16, imm8
BT r/m16, reg16
BT r/m32, imm8
BT r/m32, reg32

BTC r/m16, imm8
BTC r/m16, reg16
BTC r/m32, imm8
BTC r/m32, reg32

BTR r/m16, imm8
BTR r/m16, reg16
BTR r/m32, imm8
BTR r/m32, reg32

BTS r/m16, imm8
BTS r/m16, reg16
BTS r/m32, imm8
BTS r/m32, reg32

CALL rel16
CALL rel32
CALL mem16
CALL mem32
CALL mem48

CBW

CDQ

CLC

CLD

CLI

CLTS

CMC

CMP reg8, r/m8
CMP reg16, r/m16
CMP reg32, r/m32
CMP r/m8, imm8
CMP r/m8, reg8
CMP r/m16, imm8
CMP r/m16, imm16
CMP r/m16, reg16
CMP r/m32, imm8
CMP r/m32, imm32
CMP r/m32, reg32
CMP AL, imm8
CMP AX, imm16
CMP EAX, imm32

CMPSB

CMPSD

CMPSW

CMPXCHG r/m8, reg8
CMPXCHG r/m16, reg16
CMPXCHG r/m32, reg32

CWD

CWDE

DAA

DAS

DEC reg16
DEC reg32
DEC r/m8
DEC r/m16
DEC r/m32

DIV r/m8
DIV r/m16
DIV r/m32

ENTER imm16, imm8

F2XM1
F2XM1

FABS

FADD st, st(i)
FADD st(i), st
FADD mem32
FADD mem64

FADDP st(i), st

FBLD mem80

FBSTP mem80
```

```
FCHS

FCLEX

FCOM st(i)
FCOM mem32
FCOM mem64

FCOMP st(i)
FCOMP mem32
FCOMP mem64

FCOMPP

FCOS

FDECSTP

FDIV st, st(i)
FDIV st(i), st
FDIV mem32
FDIV mem64

FDIVP st(i), st

FDIVR st, st(i)
FDIVR st(i), st
FDIVR mem32
FDIVR mem64

FDIVRP st(i), st

FFREE st(i)

FFREEP st(i)

FIADD mem16
FIADD mem32

FICOM mem16
FICOM mem32

FICOMP mem16
FICOMP mem32

FIDIV mem16
FIDIV mem32

FIDIVR mem16
FIDIVR mem32

FILD mem16
FILD mem32
FILD mem64

FIMUL mem16
FIMUL mem32

FINCSTP

FINIT

FIST mem16
FIST mem32

FISTP mem16
FISTP mem32

FISUB mem16
FISUB mem32

FISUBR mem16
FISUBR mem32

FLD st(i)
FLD mem32
FLD mem64
FLD mem80

FLD1

FLDCW mem16

FLDENV r/m16

FLDL2E

FLDL2T

FLDLG2

FLDLN2

FLDPI

FLDZ

FMUL st, st(i)
FMUL st(i), st
FMUL mem32
FMUL mem64

FMULP st(i), st

FNINIT

FNOP

FNSTENV r/m16

FPATAN

FPREM

FPREM1

FPTAN

FRNDINT

FRSTOR r/m16

FSAVE r/m16

FSCALE

FSETPM

FSIN

FSINCOS

FSQRT

FST st(i)
FST mem32
FST mem64

FSTCW mem16

FSTENV r/m16

FSTP st(i)
FSTP mem32
FSTP mem64
FSTP mem80

FSTSW r/m16
FSTSW AX

FSUB st, st(i)
FSUB st(i), st
FSUB mem32
FSUB mem64

FSUBP st(i), st

FSUBR st, st(i)
FSUBR st(i), st
FSUBR mem32
FSUBR mem64

FSUBRP st(i), st

FTST

FUCOM st(i)

FUCOMP st(i)

FUCOMPP

FWAIT
```

```
                        IRET
FXAM                                        JNB rel8
                        IRETD               JNB rel16
FXCH st(i)                                  JNB rel32
                        JA rel8
FXTRACT                 JA rel16            JNBE rel8
                        JA rel32            JNBE rel16
FYL2X                                       JNBE rel32
                        JAE rel8
FYL2XP1                 JAE rel16           JNC rel8
                        JAE rel32           JNC rel16
HLT                                         JNC rel32
                        JB rel8
IDIV r/m8               JB rel16            JNE rel8
IDIV r/m16              JB rel32            JNE rel16
IDIV r/m32                                  JNE rel32
                        JBE rel8
IMUL reg16, r/m16       JBE rel16           JNG rel8
IMUL reg32, r/m32       JBE rel32           JNG rel16
IMUL reg16, r/m16, imm8                     JNG rel32
IMUL reg32, r/m32, imm8 JC rel8
IMUL r/m8               JC rel16            JNGE rel8
IMUL r/m16              JC rel32            JNGE rel16
IMUL r/m16, imm8                            JNGE rel32
IMUL r/m16, imm16       JCXZ rel8
IMUL r/m32              JECXZ rel8          JNL rel8
IMUL r/m32, imm8                            JNL rel16
IMUL r/m32, imm32       JE rel8             JNL rel32
                        JE rel16
IN AL, imm8             JE rel32            JNLE rel8
IN AL, DX                                   JNLE rel16
IN AX, imm8             JG rel8             JNLE rel32
IN AX, DX               JG rel16
IN EAX, imm8            JG rel32            JNO rel8
IN EAX, DX                                  JNO rel16
                        JGE rel8            JNO rel32
INC reg16               JGE rel16
INC reg32               JGE rel32           JNP rel8
INC r/m8                                    JNP rel16
INC r/m16               JL rel8             JNP rel32
INC r/m32               JL rel16
                        JL rel32            JNS rel8
INSB                                        JNS rel16
                        JLE rel8            JNS rel32
INSD                    JLE rel16
                        JLE rel32           JNZ rel8
INSW                                        JNZ rel16
                        JMP rel8            JNZ rel32
INT                     JMP rel16
                        JMP rel32           JO rel8
INT3                                        JO rel16
                        JNA rel8            JO rel32
INTO                    JNA rel16
                        JNA rel32           JP rel8
INVD                                        JP rel16
                        JNAE rel8           JP rel32
INVLPG                  JNAE rel16
                        JNAE rel32          JPO rel8
```

```
JPO rel16
JPO rel32

JS rel8
JS rel16
JS rel32

JZ rel8
JZ rel16
JZ rel32

LAHF

LAR reg16, r/m16

LDS reg16, mem32
LDS reg32, mem48

LEA reg16, mem16
LEA reg32, mem32

LEAVE

LES reg16, mem32
LES reg32, mem48

LFS reg16, mem32
LFS reg32, mem48

LGDT mem16
LGDT mem32

LGS reg16, mem32
LGS reg32, mem48

LIDT mem16
LIDT mem32

LLDT mem16

LMSW r/m16

LOCK

LODSB

LODSD

LODSW

LOOP rel8

LOOPD rel8

LOOPDE rel8

LOOPDNE rel8

LOOPDNZ rel8

LOOPDZ rel8

LOOPE rel8

LOOPNE rel8

LOOPNZ rel8

LOOPZ rel8

LSL reg16, r/m16

LSS reg16, mem32
LSS reg32, mem48

LTR r/m16

MOV reg8, imm8
MOV reg8, r/m8
MOV reg16, imm16
MOV reg16, r/m16
MOV reg32, imm32
MOV reg32, r/m32
MOV r/m8, imm8
MOV r/m8, reg8
MOV r/m16, imm16
MOV r/m16, reg16
MOV r/m16, sreg
MOV r/m32, imm32
MOV r/m32, reg32
MOV AL, mem8
MOV AX, mem16
MOV EAX, mem32
MOV sreg, r/m16
MOV mem8, AL
MOV mem16, AX
MOV mem32, EAX
MOV reg32, CR0-CR7
MOV CR0-CR7,reg32
MOV reg32, DR0-DR7
MOV DR0-DR7,reg32
MOV reg32, TR0-TR7
MOV TR0-TR7,reg32

MOVSB

MOVSD

MOVSW

MOVSX reg16, r/m8
MOVSX reg32, r/m8
MOVSX reg32, r/m16

MOVZX reg16, r/m8
MOVZX reg32, r/m8
MOVZX reg32, r/m16

MUL r/m8
MUL r/m16
MUL r/m32

NEG r/m8
NEG r/m16
NEG r/m32

NOP

NOT r/m8
NOT r/m16
NOT r/m32

OR reg8, r/m8
OR reg16, r/m16
OR reg32, r/m32
OR r/m8, imm8
OR r/m8, reg8
OR r/m16, imm8
OR r/m16, imm16
OR r/m16, reg16
OR r/m32, imm8
OR r/m32, imm32
OR r/m32, reg32
OR AL, imm8
OR AX, imm16
OR EAX, imm32

OUT imm8, AL
OUT imm8, AX
OUT imm8, EAX
OUT DX, AL
OUT DX, AX
OUT DX, EAX

OUTSB

OUTSD

OUTSW

POP reg16
POP reg32
POP mem16
POP mem32
POP DS
POP ES
POP SS
POP FS
POP GS

POPA

POPAD
```

```
POPF
POPFD
PUSH imm16
PUSHD imm32
PUSH mem16
PUSH mem32
PUSH reg16
PUSH reg32
PUSH CS
PUSH DS
PUSH ES
PUSH SS
PUSH FS
PUSH GS
PUSHA
PUSHAD
PUSHF
PUSHFD
RCL r/m8, imm8
RCL r/m8, CL
RCL r/m16, imm8
RCL r/m16, CL
RCL r/m32, imm8
RCL r/m32, CL
RCR r/m8, imm8
RCR r/m8, CL
RCR r/m16, imm8
RCR r/m16, CL
RCR r/m32, imm8
RCR r/m32, CL
REP
REPNE
RET
RET imm16
RETF
RETF imm16
ROL r/m8, imm8
ROL r/m8, CL
ROL r/m16, imm8
ROL r/m16, CL
ROL r/m32, imm8
ROL r/m32, CL
ROR r/m8, imm8
ROR r/m8, CL
ROR r/m16, imm8
ROR r/m16, CL
ROR r/m32, imm8
ROR r/m32, CL
SAHF
SAL r/m8, imm8
SAL r/m8, CL
SAL r/m16, imm8
SAL r/m16, CL
SAL r/m32, imm8
SAL r/m32, CL
SAR r/m8, imm8
SAR r/m8, CL
SAR r/m16, imm8
SAR r/m16, CL
SAR r/m32, imm8
SAR r/m32, CL
SBB reg8, r/m8
SBB reg16, r/m16
SBB reg32, r/m32
SBB r/m8, imm8
SBB r/m8, reg8
SBB r/m16, imm8
SBB r/m16, imm16
SBB r/m16, reg16
SBB r/m32, imm8
SBB r/m32, imm32
SBB r/m32, reg32
SBB AL, imm8
SBB AX, imm16
SBB EAX, imm32
SCASB
SCASD
SCASW
SETA r/m8
SETAE r/m8
SETB r/m8
SETBE r/m8
SETC r/m8
SETE r/m8
SETG r/m8
SETGE r/m8
SETL r/m8
SETLE r/m8
SETNA r/m8
SETNAE r/m8
SETNB r/m8
SETNBE r/m8
SETNC r/m8
SETNE r/m8
SETNG r/m8
SETNGE r/m8
SETNL r/m8
SETNLE r/m8
SETNO r/m8
SETNP r/m8
SETNS r/m8
SETNZ r/m8
SETO r/m8
SETP r/m8
SETPE r/m8
SETPO r/m8
SETS r/m8
SETZ r/m8
SGDT mem16
SGDT mem32
SHL r/m8, imm8
SHL r/m8, CL
SHL r/m16, imm8
SHL r/m16, CL
SHL r/m32, imm8
SHL r/m32, CL
SHLD r/m16, reg16, imm8
SHLD r/m16, reg16, CL
SHLD r/m32, reg32, imm8
SHLD r/m32, reg32, CL
```

```
SHR r/m8, imm8
SHR r/m8, CL
SHR r/m16, imm8
SHR r/m16, CL
SHR r/m32, imm8
SHR r/m32, CL

SHRD r/m16, reg16, imm8
SHRD r/m16, reg16, CL
SHRD r/m32, reg32, imm8
SHRD r/m32, reg32, CL

SIDT mem16
SIDT mem32

SMSW r/m16

STC

STD

STI

STOSB

STOSD

STOSW

STR r/m16

SUB reg8, r/m8
SUB reg16, r/m16
SUB reg32, r/m32
SUB r/m8, imm8
SUB r/m8, reg8
SUB r/m16, imm8
SUB r/m16, imm16
SUB r/m16, reg16
SUB r/m32, imm8
SUB r/m32, imm32
SUB r/m32, reg32
SUB AL, imm8
SUB AX, imm16
SUB EAX, imm32

TEST r/m8, imm8
TEST r/m8, reg8
TEST r/m16, imm16
TEST r/m16, reg16
TEST r/m32, imm32
TEST r/m32, reg32
TEST AL, imm8
TEST AX, imm16
TEST EAX, imm32

VERR r/m16
VERW r/m16

WAIT

WBINVD

XADD r/m8, reg8
XADD r/m16, reg16
XADD r/m32, reg32

XCHG reg8, r/m8
XCHG reg16, r/m16
XCHG reg32, r/m32
XCHG r/m8, reg8
XCHG r/m16, reg16
XCHG r/m32, reg32
XCHG AX, reg16
XCHG EAX, reg32

XLATB

XOR reg8, r/m8
XOR reg16, r/m16
XOR reg32, r/m32
XOR r/m8, imm8
XOR r/m8, reg8
XOR r/m16, imm8
XOR r/m16, imm16
XOR r/m16, reg16
XOR r/m32, imm8
XOR r/m32, imm32
XOR r/m32, reg32
XOR AL, imm8
XOR AX, imm16
XOR EAX, imm32
```

A.7 Pass32 Limits

There is no limit in source file size. The limit for the memory size is 4G, although only 64MB are supported by older XMS Versions.

As Pass32 is (still) a real mode product, the amount of symbols is limited by the available Dos memory. Symbols are: Labels, Procedures and Data identifier.

The maximal symbol length is 127 characters for labels and 128 characters for procedures and variable identifiers.

The maximal number of symbols is depending on the symbol length. With an average of 11 characters per symbol, the maximum is about 18000 symbols, with prepass optimization about 16000 symbols. With an average of 35 characters per symbol, the maximum is about 8000 symbols. The number of procedures is limited to 4000. The number of modules (actually included) is limited to 50.

B Pro32 Dos Extender

B.1 The Dos Extender Loader

The DOS extender program (**PRO32.EXE**) is copied to the beginning of the protected mode binary - done by the assembler and linker **PASS32.EXE** or by the linker tool **PROSET.EXE**. The DOS extender is therefore a part of the protected mode program. When called from DOS only the DOS extender is load into DOS Memory and executed.

The extender first checks, if a DPMI host is available. If so, the processor is switched into protected mode and the extender continues with the loading of the program. If there is no DPMI available (or only a 16 bit DPMI host), the Dos Extender checks for VCPI. If there is no VCPI, the DOS extender checks for XMS-memory manager. If there is no XMS-manager, the dos extender emulates the XMS memory manager. Now, as XMS memory can be accessed, the dos extender starts with the DPMI emulation. If VCPI is present, the extender uses the VCPI methods to switch between real mode and protected mode. Otherwise, the dos extender initiates its own GDT and uses CPU instructions to switch between real mode and protected mode (see more section B.2 *The Integrated DPMI Server* on page 121).

The protected mode parts of the dos extender are independent of the interface DPMI, VCPI, XMS or RAW. In protected mode, Pro32 tries to allocate the program memory: Pro32 first checks how much memory is available in the system. If the amount of memory is below the MinMemory variable, Pro32 terminates with the error message: *too less memory available*. If the amount of memory is above the MaxMemory variable, Pro32 allocates memory according to the MaxMemory value. Otherwise, Pro32 allocates all available Memory. So Pro32 allocates at least an amount of memory according to the MinMemory value and maximal according to MaxMemory.

Now the DOS Extender allocates real mode memory for the PM-stack. It uses real mode memory, as this memory is according to the DPMI and Windows-Specification always locked. Note, if the memory used by the stack is not locked, it will produces a stack exception under any Windows version. Therefore a real mode stack provides a more stable application. Pro32 needs at least 100h bytes of stack and can use max. 512 Kbytes. As default, Pro32 allocates 32KByte stack.

Afterwards, the dos extender tries to open the program file. The progam name is usually the parameter 0.

At last the DOS extender checks the video configuration. If the current video mode is not 80x25 textmode, the DOS extender changes the video mode to 80x25 textmode.

The extender establishes several descriptors:

- 32 bit Code Descriptor
- 32 bit Data Descriptor
- 32 bit Stack Descriptor
- 16 bit Video Descriptor
- 32 bit Zero Base Descriptor (Basis:00000000h)
- 16 bit Real Mode File Buffer Descriptor
- 16 bit DOS Environment Descriptor

If the message flag is enabled, the extender displays the basis of the 32 bit code and data descriptor: „load to address:xxxxxxxx“

The current PSP is copied with the High Data Descriptor into XMS at offset 00000000. The DOS extender loads the program to CS:00000100 (The load address can be changed, see PRO32.DOC / PROSET.DOC).

When the whole program is copied into the XMS memory, all protected descriptors and additional system information are copied into the PSP. They can be reached by CS:

00-01	DS - data selector
02-03	ES - video selector
04-05	FS, GS - zero selector
06-07	Real Mode File Buffer Selector
08-09	Real Mode File Buffer Segent
0A-0D	Actual allocated XMS Memory
0E	Flag, if windows has been detected[a]
0F	Flag, if other DPMI host is active[a]
2C-2F	selector to DOS environment
80-FF	command line with arguments

a. available with Pro32 Version 1.47 and newer versions.

Tab. B.1 *The selector register contents and the PMode PSP*

The value of EAX,EBX,ECX,EDX,ESI,EDI and EBP is zero. ESP holds the maximum stack size.

B.2 The Integrated DPMI Server

The main part of the DOS extender is using DPMI calls. If the system already provides a 32 bit DPMI host, its DPMI functions are used. For this reason, Pro32 programs are able to run under Windows 3.x, Windows 9x and under Windows NT. Pro32 does not support all DPMI functions. For this reason you should use only those that are listed. There are some differences between the DPMI specifications and the Pro32 DPMI emulation. The main reason for these differences is the speed of DPMI. DPMI is mainly slowed down, because all real mode interrupts must be provided by the DPMI host. For this reason DPMI hosts typically uses an exception handler which is checking the interrupt type (external, software,exception...) and reacting according to the type. Pro32 has a very fast interrupt handler for each individual interrupt. It supports the real mode interrupts 00..7fh.

00	Division by zero*	16	keyboard
01	Single step*	17	Printer
02	NMI	1b	CTRL-Break**
03	Break Point*	1c	Clock
04	Overflow*	21	DOS-API
05	Bound Check*	23	CRTL-C Exit**
06	Invalid Opcode*	24	FATAL ERROR Handler**
07	no numeric co-processor*	31	DPMI API

08	double exception* HW IRW0-Timer	33	Mouse driver API
09	Segment overrun* HW IRQ1-Keyboard	70	HW IRQ8 Real time clock
0A	Invalid Task STate Segment* HW IRQ2	71	HW IRQ9 Lan adaptor
0b	Segment not present* HW IRQ3 COM2	72	HW IRQ10
0c	Stack Exception* HW IRQ4 COM1	73	HW IRQ11
0d	General Protection Fault* HW IRQ5	74	HW IRQ12
0e	HW IRQ6 Floppy Drive	75	HW IRQ13
0f	HW IRQ7 Printer	76	HW IRQ14 Fixed disk
10	VGA Bios	77	HW IRQ15
11 12 13 14 15	BIOS	7A	Novell Netware API

* Exceptions are handled by Pro32

** These interrupts are also transfered from real mode to protected mode.

Tab. B.2 *SW/HW Interrupts and Exceptions*

The interrupts 80h-0FFh cause an exception.

The exceptions 00 - 0ah, 0ch and 0dh from any other DPMI host are (if the DPMI host offers this function) controlled by the Pro32 Dos Extender. So when your program is running under Windows, and an exception occurs, Pro32 will handle the exception. The exception handler usually prints the error address, the error code, the exception type, the contents of all registers, the basis addresses and the limits of the selectors CS, DS, ES, FS, GS, SS.

The Pass32 Assembler offers a find error option, so you can easily trace the error by searching for the error address:

```
Pass32 dummy.asm -error:0127
```

B.3 The DPMI Service API

The DPMI function numbers are passed in AX. The DPMI function is invoked by calling the INT 31h. Parameters are passed in BX, CX, DX, DI, SI and not in 32 bit registers due to the 16 bit origin of the DPMI. 32 bit parameters are typically expressed by BX:CX, CX:DX, SI:DI.

When the DPMI function has been succesful, the carry flag is clear. Otherwise, the carry flag is set and AX holds an error code (see section B.4 *DPMI Error Codes in AX:* on page 137).

The appendix refers to the 32 bit version of the DPMI.

B.3.1 Function 0000h - Allocate Descriptor

Allocates one or more descriptors in the descriptor table. These desciptors have a base and limit of 00000000h, they will be set to expand-up writeable data, with the present bit set. If more than one desciptor was allocated, the returnd selector is a base selector. You must add the value of INT 31h function 0003h to get the next selector. You should request only one selector by one, to avoid gaps in the descriptor table.

Function call: INT 31h

```
AX      = 0000h
CX     = number of descript  ors to allocate (usually CX=0001)
```

Results, if successful

```
carry flag clear
AX     = base selector
```

Example:

```
 mov CX,1
 mov AX,0000h
 int 31h
 jc DPMIERROR                    ; probably no more system resources...
 mov NewSelector,AX
   ....
 DPMIERROR:
 call printerror
```

B.3.2 Function 0001h - Free Descriptor

Frees a descriptor allocated by the function 0000h. You should not free descriptors allocated by the DPMI host (for example the initial CS,SS,DS,ES,FS,GS) and descriptors allocated by function 00002h.

Function call: INT 31h

```
AX      = 0001h
BX     = selector for the d  escriptor  to free
```

Results, if successful:

```
carry flag clear, selector freed
```

Example:

```
mov BX,NewSelector              ; no need for NewSelector anymore...
mov ax,0001h
int 31h
```

Notes:

Any use of a free selector will cause a general exception. If a selector register holds a freed selector, Pro32 usually loads the selector register with the dummy selector.

B.3.3 Function 0002h - Get Real Mode Segment Descriptor

Converts a real mode segment into a protected mode descriptor. The default size attribute is 16 bit, the descriptor type expand-up writeable data, with a base to the Real Mode segment and a limit of 0FFFFh.

Function call: INT 31h

```
AX      = 0002h
BX     = real mode   segment
```

Results, if successful:

```
carry flag clear
AX     = selector
```

Examples:

```
mov BX,0A000h               ;Video Graphic Memory
mov AX,0002h
int 31h
jc DPMIERROR                ; probably no more system resource s...
mov GraphicSel,ax
```

Notes:

According to the DPMI specification you should not alter base or limit of these descriptors.

B.3.4 Function 0003h - Get Selector Increment Value

You can allocate more than one descriptors with INT 31h function (0000h). To get the next descriptor you must add the increment value to the base selector.

Function call: INT 31h

```
AX      = 0003h
```

Results:

```
AX      = selector increment value
```

B.3.5 Function 0006h - Get Segment Base Address

Returns the 32bit linear base address of the selector.

Function call: INT 31h

```
AX      = 0006h
BX     = selector
```

Results, if successful:

```
carry flag clear
CX:DX  = 32bit line ar base address of the selector
```

B.3.6 Function 0007h - Set Segment Base Address

Sets the 32bit linear base address field in the descriptor for the specified segment.

Function call: INT 31h

```
AX      = 0007h
BX     = selector
CX:DX  = 32bit linear base a ddress of segment
```

Results, if successful:

```
carry flag clear
```

Example:

```
mov BX,NewSelector
mov CX,0Ah
mov DX,0                    ; CX:DX points to 0A0000h
mov AX,7                    ; so we create another Graphic Screen
int 31h                     ; Selector!
```

B.3.7 Function 0008h - Set Segment Limit

Sets the limit field in the descriptor for the specified segment.

Function call: INT 31h

```
AX      = 0008h
BX     = selector
CX:DX  = 32bit segment limit
```

Results, if successful:

```
carry flag clear
```

Example:

```
mov BX,NewSelector
mov CX,0
mov DX,0FFFFh                           ; set 64K Limit to NewSelector
mov AX,8                                    ; Limit = Size -1
int 31h
```

Note: The granularity may change depending on the value of CX:DX

B.3.8 Function 0009h - Set Descriptor Access Rights

Modifies the access rights field in the descriptor for the specified segment. The access rights stored in CX have the following format:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	G	BD	0	AVL	?				1	DPL		1	CD	EC	RW	A

G: Limit Granularity: 0: byte granular (Limit=Limit field)
1: page granular (Limit=4096*Limit field)

B/D: Segment Attribute Size: 0: 16 bit = use16
1: 32 bit = use32 (default)

AVL: Available Flag: (unused)

DPL: Privileg Level use the LAR instruction to examine the DPL. Typically:

	00: kernel level	
	01: device driver level	
	10: operating system level	
	11: user application level	
C/D: Segment Type	0: data	
	1: code	
E/C: Expand/Conforming	0: data=expand-up	code=non-conforming (default)
	1: data=expand-down	code=conforming
R/W: Read / Write	0: data=read	code=non-readable
	1: data=read/write	code=readable
A: Access	0: not accessed,	
	1: accessed	

0: must be 0

1: must be 1

?: will be ignored

Tab. B.3 *Descriptor Access Rights*

Function call: INT 31h

```
AX      = 0009h
BX     = selector
CX     = access rig  hts/type word
```

Results, if successful:

```
carry flag clear
```

B.3.9 Function 000Ah - Create Alias Descriptor

Creates a so called alias descriptor that has the same base and limit as the specified descriptor. The alias descriptor is always an expand-up data type. You can create alias descriptors as well from data and code descriptors.

Function call: INT 31h

```
AX      = 000ah
BX     = selector
```

Results, if successful:

```
carry flag clear
AX     = alias sele  ctor (data)
```

B.3.10 Function 000Bh - Get Descriptor

Copies the descriptor table entry of the specified selector into an 8 byte buffer. The buffer contents are described in Tab. 4 .

Bits / offset	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Descriptor Limit 15..0															
2	Descriptor Base 15..0															
4	1	DPL		1	CD	EC	RW	A	Descriptor Base 23..16							
6	Descriptor Base 31..24								G	BD	0	AVL	Limit 19..16?			

Tab. B.4 *The layout of a 32 bit descriptor*

The abreviations are according to Tab. 3 .

Function call: INT 31h

```
AX      = 000bh
BX     = selector
ES:EDI = selector:offset of  8 the byte buffer
```

Results, if successful:

```
carry flag clear, buffer pointed to by ES:EDI contains descriptor data
```

B.3.11 Function 000Ch - Set Descriptor

To write a complete descriptor into the descriptor table. The contents of an 8 bytes buffer is copied into the descriptor table for the specified selector. The descriptor format is shown in Tab. 3 .

Function call: INT 31h

```
AX      = 000ch
BX     = selector
ES:EDI = selector:offset of the 8 byte buffer with vaild
         descriptor format.
```

Results, if successful:

```
carry flag clear
```

Notes:

The function does not check if the descriptor entries are valid. Invalid descriptor values will cause an exception if the descriptor is load into a segment register.

B.3.12 Function 0100h - Allocate DOS Memory

Allocates DOS memory through DOS function 48h and creates a descriptor for the memory. If more than 64 K is allocated, the descriptor will have the limit set above 64K. According to the DPMI Specification this function allocates an array of descriptors with regard to the 16-bit DPMI Version. The 16-bit DPMI function allocates as many 64K descriptors as are necessary to select the whole memory. The 32-bit DPMI function allocates as well an array of descriptors with the limit of 64K, but with the exception, that the first descriptor has a limit of the total amount of allocated memory.

The DPMI emulation does not allocate an array of 64K descriptors (these are limited system resources!), but a single descriptor with a limit of the allocated memory. I think this is the only useful way to handle this function!

Function call: INT 31h

```
AX      = 0100h
BX     = paragraphs  (in 16 bytes) of DOS memory to allocate
```

Results, if successful:

```
carry flag clear
AX         = real   mode segment address
DX         = prote  cted mode selector for memory block
```

Results, if failed:

```
carry flag set
AX         = DOS e  rror code
BX         = size   of largest available block
```

B.3.13 Function 0101h - Free DOS Memory Block

Frees a low memory block previously allocated by function 0100h.

Function call: INT 31h

```
AX      = 0101h
DX      = protected mode selector for memory block
```

Results, if successful:

```
carry flag clear
```

B.3.14 Function 0200h - Get Real Mode Interrupt Vector

Returns the real mode interrupt vector for a specified interrupt.

Function call: INT 31h

```
AX      = 0200h
BL     = interrupt   number
```

Results:

```
CX:DX   = segment:offset of real mode interrupt handler
```

B.3.15 Function 0201h - Set Real Mode Interrupt Vector

To set a real mode interrupt vector.

Function call: INT 31h

```
AX      = 0201h
BL     = interrupt number
CX:DX  = segment:offset of r eal mode interrupt handler
```

Notes: The altered interrupts are only valid as long as the program is running - after the DOS exit function 4Ch all interrupt vectors are restored! The interrupt procedure must be located in real memory with a segment attribute of 16 bit!

B.3.16 Function 0202h - Get Exception Handler

Returns the 16:32 vector of the current protected mode exception handler for the specified exception.

Function call: INT 31h

```
AX      = 0202h
BL     = exception number
```

Results:

```
CX:EDX = selector:offset of protected mode exception handler
```

Note: Pro32 supports only values between 00h and 0bh as exceptions.

B.3.17 Function 0203h - Set Exception Handler

Sets the 16:32 address of the protected mode exception handler for the specified exception.

Function call: INT 31h

```
AX      = 0203h
BL     = exception number
CX:EDX = selector:offset of  protected mode exception handler
```

Note: Pro32 supports only values between 00h and 0bh as exceptions. CX and EDX should contain the protected mode address of an exception handler. The exception handler is called by a far call from the DPMI host. The stack contains the following values according to Tab. 5 .

	Error Value of SS
Error Value of ESP	
Error Value of E-Flags	
	Error Value of CS
Error Value of EIP	
Error Code	
	Caller CS
Caller EIP	

Tab. B.5 *Stack Contents for the Exception Handler*

The exception handler is invoked by a far call and should return with `retf`. Look at the **`DEMOEXC.ASM`** example file how to create an exception handler.

B.3.18 Function 0204h - Get Protected Mode Interrupt Vector

Returns the 16:32 vector of the current protected mode interrupt handler for the specified interrupt.

Function call: INT 31h

```
AX      = 0204h
BL     = interrupt  number
```

Results:

```
CX:EDX = selector:offset of protected mode interrupt handler
```

B.3.19 Function 0205h - Set Protected Mode Interrupt Vector

Sets the 16:32 address of the protected mode interrupt handler for the specified interrupt.

Function call: INT 31h

```
AX      = 0205h
BL     = interrupt  number
CX:EDX = selector offset of protected mode interrupt handler
```

Note: You should use this function to set any protected mode interrupt vector 00-7fh (see section B.2 *The Integrated DPMI Server* on page 121). If you want to create an exception handler, use function 0203h. The different Windows DPMI hosts differ especially in this function: `int 3`, for example, can be under Windows DPMI either an interrupt or an exception.

B.3.20 Function 0300h - Call Real Mode Interrupt

This function must be used to pass real mode segment register values to real mode interrupts.

Function call: INT 31h

```
AX      = 0300h
BX     = interrupt number (  BH must be 0)
CX     = number of words to copy from the protected mode stack to the
         real mode stack
ES:EDI = selector:offset of real mode register transfer data structure
```

Results[1]:

```
ES:EDI = selector offset of modified real mode register transfer data
         structure
```

The real mode transfer data structure holds the values of all CPU registers, that are passed to the real mode interrupt.

Offset	Type	Contents
00	DD	EDI register contents
04	DD	ESI register contents
08	DD	EBP register contents
0c	DD	reserved 0
10	DD	EBX register contents
14	DD	EDX register contents
18	DD	ECX register contents
1c	DD	EAX register contents
20	DW	Flags contents
22	DW	ES segment register contents
24	DW	DS segment register contents
26	DW	FS segment register contents
28	DW	GS segment register contents
2a	DW	IP[a]
2c	DW	CS segment register contents[b]
2e	DW	SP register contents[c]
30	DW	SS segment register contents[d]

a. defines the code offs. in DPMI function #0301,#0302, ignored in DPMI function #0300
b. defines the code seg. in DPMI function #0301,#0302, ignored in DPMI function #0300
c. if the value is zero, the DPMI host provides its own real mode stack
d. same as c). Must be zero or hold a correct real mode segment value.

Tab. B.6 *The real mode register data transfer structure as defined in* **DPMI.INC**

Example:

1.Make sure, that the execution of the real mode function does not effect the stability of the system. There is no exception handling in real mode.

```
;we assume our structure variables are called intedi, inteax etc
mov inteax,0900h   ; DOS Function c all AH=09h
mov intedx,0       ; Offset of our  Real Mode Message
mov ax,RealSegment  ; a real mode segment value, segment of message
mov intds,ax
push ds
pop es
mov edi,offset inted i; ES : EDI points to structure
mov cx,0           ; No parameters
mov bx,21h         ; DOS Interrupt  Number
mov ax,0300h       ; Call Real Mode  Interrupt Function
int 31h
```

B.3.21 Function 0301h - Call Real Mode Procedure With Far Return

Calls a real mode procedure which ends with a `RETF` instruction.

Function call: INT 31h

```
AX      = 0301h
BH     = must be 0
CX     = number of  words to copy from protected mode stack to real
          mode stack
ES:EDI = selector:offset of real mode register data transfer structure
```

Results:

```
ES:EDI = selector offset of modified real mode register transfer data
         structure
```

B.3.22 Function 0302h - Call Real Mode Procedure With IRET Return

This Function calls a real mode procedure which ends with one of the following instructions: `IRET` or `RETF 2`.

Function call: INT 31h

```
AX      = 0302h
BH     = must be 0
CX     = number of  words to copy from the protected mode stack to
          the real mode stack
ES:EDI = selector:offset of real mode register data transfer structure
```

Results:

```
ES:EDI = selector offset of modified real mode register transfer data
         structure
```

B.3.23 Function 0303h -Install Real Mode Call Back Functions

This Function returns a real mode procedure, which will call a specific protected mode handler.

Function call: INT 31h

```
AX      = 0303h
DS:ESI = selector:of fset to protected mode handler
ES:EDI = selector:of fset of real mode register data transfer
```

Results:

```
CX:DX  = segment:offset to real mode call back function.
```

The PM Handler (procedure in DS:ESI) is called by the real mode call back function with the following parameters:

```
DS:ESI = point to real mode stack
ES:EDI = points to real mode register data transfer structure.
```

Notes:

The PM Handler returns with an IRET instruction. The PM Handler must store the real mode return address in the data transfer structure. (Usually done by reading the offset and segment address from the real mode stack provided by DS:ESI and writing the results into the real mode register transfer structure). And the PM Handler is responsible for popping the return address from the real mode stack. See the **`EVENT.ASM`** example file.

B.3.24 Function 0304h - Free Real Mode Call Back Functions

This Function frees a real mode call back function allocated by function 0303.

Function call: INT 31h

```
AX      = 0304h
CX:DX  = segment:offset to r eal mode call back function.
```

Results:

```
Carry flag clear
```

Notes:

You must make sure, that the real mode call back function won't be called again, before freeing the call back! Pro32 offers 16 call backs per client!

B.3.25 Function 0400h - Get Version

Returns the version of the DPMI host.

Function call: INT 31h

```
AX      = 0400h
```

Results:

```
AH      = DPMI major version number
AL      = DPMI minor version number
BX      = Bits     Description
          0   : 1 = host is 32bit
          1   : 0  = CPU running V86 mode for reflected interrupts
                1 = CPU running real mode for reflected interrupts
          2   : 0  = virtual memory not supported
                1 = virtual memory supported
CL     = processor type:
         03h = 80386
         04h = 80486
         05h = 80586
         06h = 80686 / Pentium
```

```
DH      = value of master PIC base interrupt
DL     = value of   slave PIC base interrupt
```

B.3.26 Function 0500h - Get Free Memory Information

Returns Information about the amount of free memory.

Function call: INT 31h

```
AX      = 0500h
ES:EDI = selector:offset of 48 byte buffer with the following format
```

Results:

```
modified entries of the 48 byte buffer at ES:EDI
```

Offset	Type	Contents
00	DD	Size of larges available free memory block in bytes
04	DD	Size available with locking
08	DD	Size available without locking
0c	DD	Size of total Memory in pages
10	DD	number of locked pages
14	DD	number of unlocked pages
18	DD	number of free pages
1c	DD	number of all available pages
20	DW	free linear memory in pages
22	DW	size of swap file

Tab. B.7 *Free Memory Table*

Notes:

Function should never fail! At least first entry must be valid!

B.3.27 Function 0501h - Allocate Memory Block

Allocates a block of extended memory.

Function call: INT 31h

```
AX      = 0501h
BX:CX  = memory siz e to allocate in bytes
```

Results, if successful:

```
carry flag clear
BX:CX  = linear add ress of allocated memory block (*)
SI:DI  = memory han dle
```

Example:

```
mov ax,0       ;Funktion 0
```

```
mov cx,1       ;Allocate 1 Descriptor
int 31h
mov MemDesk,AX
mov ax,0501h  ;Allocate Memo ry
mov cx,0
mov bx,1      ;010000h Byte  s of Memory
int 31h       ;DPMI CALL
jc TooLessMemory
mov AX,0007h  ;Function 7:Se t Basis Address
mov BX,MemDesk
mov dx,cx     ;Low Part of   Linear Address
mov cx,bx     ;upper Part o  f Linear Address
int 31h       ;DPMI CALL
mov BX,MemDesk
mov ax,0008h  ;Function 8:Se t Limit of descriptor
mov cx,0ffffh
mov dx,0
int 31h       ;set Limit
mov BX,MemDesk
mov fs,BX     ; FS selector   to 64K XMS Memory Location!
```

B.3.28 Function 0502h - Free Memory Block

Frees a previously allocated extended memory block.

Function call: INT 31h

```
AX      = 0502h
SI:DI  = memory handle
```

Results, if successful:

```
carry flag clear
```

B.3.29 Function 0600h - Lock Linear Region

The function locks a specified linear address range.

Function call: INT 31h

```
AX      = 0600h
BX:CX  = start of linear add ress in memory
SI:DI  = size of region in b  ytes
```

Results, if successful:

```
carry flag clear
```

Notes:

Pro32 DPMI sets up a locked linear address space between 0 and the maximum available bytes in the DPMI emulation, therefore this function is alway successful under the DPMI emulation.

B.3.30 Function 0601h - Unlock Linear Region

The function unlocks a specified linear address range previously locked with function 0600h.

Function call: INT 31h

```
AX      = 0601h
BX:CX  = start of l inear address in memory
SI:DI  = size of re gion in bytes
```

Results, if successful:

```
carry flag clear
```

B.3.31 Function 0602h - Mark real mode region as pagable

This function allows to make real mode memory pagable.

Function call: INT 31h

```
AX      = 0602h
BX:CX  = start of l inear address in memory
SI:DI  = size of re gion in bytes
```

Results, if successful:

```
carry flag clear
```

Notes:

You should relock all memory before terminating your program. You should not mark regions pageable, if they are not part of your application.

B.3.32 Function 0603h - Relock real mode region

This function locks a previously unlocked real mode memory region.

Function call: INT 31h

```
AX      = 0603h
BX:CX  = start of l inear address in memory
SI:DI  = size of re gion in bytes
```

Results, if successful:

```
carry flag clear
```

B.3.33 Function 0800h - Map physical address

This function returns the linear address region to address a physical memory region.

Function call: INT 31h

```
AX      = 0800h
BX:CX  = physical a ddress in memory
SI:DI  = size of re gion in bytes
```

Results, if successful:

```
carry flag clear
BX:CX  = linear address
```

Notes: Pro32 provides a linear memory region equal to the physical address region between 0 and the maximum of available bytes. Therefore under the Pro32 DPMI emulation, this funvtion will always succed with BX:CX unchanged.

B.3.34 Function 0900h - Get and Disable Virtual Interrupt State

Replacement for the CLI instruction, which is a priviledged(!) instruction. Especially when running under V86 mode the CLI instruction will be very slow.

Function call: INT 31h

```
AX      = 0900h
```

Results:

```
AL      = 0 if virtual interrupts were previously disabled
AL     = 1 if virtual inter  rupts were previously enabled
```

B.3.35 Function 0901h - Get and Enable Virtual Interrupt State

Replacement for the STI instruction, which is a priviledged(!) instruction. Especially when running under V86 mode the STI instruction will be very slow.

Function call: INT 31h

```
AX      = 0901h
```

Results:

```
AL      = 0 if virtual interrupts were previously disabled
AL     = 1 if virtual inter  rupts were previously enabled
```

B.3.36 Function 0902h - Get Virtual Interrupt State

Returns the current state of the virtual interrupt flag.

Function call: INT 31h

```
AX      = 0902h
```

Results:

```
AL      = 0 if virtual interrupts were previously disabled
AL     = 1 if virtual inter  rupts were previously enabled
```

B.4 DPMI Error Codes in AX:

AX = 0007h (DOS ERROR Function 01xx): memory control blocks damaged
0008h (DOS ERROR Function 01xx): insufficient memory available

0009h (DOS ERROR Function 01xx): incorrect memory segment
8001h invalid DPMI function (The requested function is not available).
8003h function would lead to a protection fault
8010h no more system ressources
8011h illegal descriptor
8012h insufficient linear memory
8013h insufficient physical memory
8016h invalid handle
8022h illegal selector

B.5 Error Messages

Pro32 provides the following error messages:

.

Error: Invalid DPMI FUNCTION	The DPMI host in your system doesn't provide a basic dpmi function needed to load and execute the program. Or the resources of the system are exhausted.
Error: Too less Memory available	The amount of free memory is below the size of memory needed for the program. Install more memory in the system, or lower the size of necessary memory (if possible!).
Error: DOS: Int 24 failure	The default Pro32 Int24 handler, occurs in connection with disk errors.You can hook this interrupt to create your own exception handler.
Error: System failure	The CHECKSUM of the Pro32 dos extender is wrong. This error occurs, if the pro32 extender is destroyed.
Error: Real Mode: Stack Overflow	A real mode function either uses more than 8 Kbytes stack or the function destroys the real mode stack.
Error: Invalid Processor Type	The processor used is identified as 80286 or minor processor type.Pro32 needs at least a 80386 processor.
Error: Processor already in virtual real mode	The system is running in virtual real mode. Pro32 can't access the protected mode. This error occurs, if the system neither provides DPMI, nor VCPI.
Error: DPMI Host: Error Switchting to protected mode	The PM initialisation done by another DPMI host, for exsample WINDOWs fails. You are already running other programs under the same DPMI host, or the host has insufficient memory
Error: VCPI Init failed	The PM initialisation with VCPI fails.

Tab. B.8 *Pro32 Error Messages*

List of Tables

Index