

Protected Mode Tutorial  
 V 0.02  
 Written by Till Gerken

-----  
 -- INTRODUCTION --

This text contains a short and simple step-by-step tutorial for Protected Mode beginners. It shows you everything you need to program your own PM environment and is intended for those who don't have any experiences with it yet. All you need to understand this text is your brain (you have one, do you? :) ) and a bit assembler knowledge.

The text will try to teach you the principles of the 80386's Protected Mode and contains the complete source for a mode switcher. This little program was made to show you the basic rules of Protected Mode code, so it is unoptimized and kept simple, but very well documented. If you have questions/suggestions for an extension, `_please_ mail me.`

The whole code is written in assembler, using Ideal mode TASM 2.01 syntax. Newer versions of TASM will work, too, but you'll have to add a macro to convert `DWORDS` to `WORDS` in order to compile the assembler file.

I'm currently thinking about splitting the text into several files, each containing information about a different topic. (like mode switching under DPMS, VCPI, XMS and RAW; Exception handling; etc.) This would make it a bit handier and easier to read. Someone has any opinion?

-----  
 -- GETTING STARTED --

Well, "getting started" is what this document is all about... Here you'll learn what you need to prepare for Protected Mode. There are several things you have to take care of, at first you have to check on which processor your program's currently running (trying to do PM on a 8086 may hurt... :) )

The best way to test for a 80386 is to test the processor's flag register. Since the 80386, flag bits 12-14 are used for the I/O Privilege Level (IOPL) and the Nested Task (NT) flag, so the only thing to do is to test if these bits are modifiable (the 8086, 8088, 80186 don't use these flags and set them implicitly to zero, the 80286 uses them, but they can only be modified in Protected Mode. DOS can't run in Protected Mode, so if the program runs on a 80286, it is in Real Mode, and there the flags can't be modified).

; checks for a 386

```
no386e      db      'Sorry, at least a 80386 is needed!',13,10,'$'
```

```
proc      check_processor
  pushf                    ; save flags
  xor     ah,ah            ; clear high byte
  push   ax                ; push AX onto the stack
  popf                    ; pop this value into the flag register
  pushf                    ; push flags onto the stack
  pop     ax               ; ...and get flags into AX
  and    ah,0f0h          ; try to set the high nibble
  cmp    ah,0f0h          ; the high nibble is never 0f0h on a
  je     no386             ; 80386!
  mov    ah,70h           ; now try to set NT and IOPL
  push   ax
  popf
  pushf
  pop     ax
  and    ah,70h           ; if they couldn't be modified, there
  jz     no386             ; is no 80386 installed
  popf                    ; restore the flags
```

```

        ret                ; ...and return
no386:  mov     dx,offset no386e    ; if there isn't a 80386, put a msg
        jmp     err16exit        ; and exit
endp    check_processor

; exits with a msg
; In:   DS:DX - pointer to msg

proc    err16exit
        mov     ah,9            ; select DOS' print string function
        int     21h            ; do it
        mov     ax,4cfffh       ; exit with 0fffh as exit code
        int     21h            ; good bye...
endp    err16exit

```

Now we have a function to determine if there is a 80386 installed and one which provides a quick and dirty exit. The second thing to be done now is to check in which mode we are running. Expanded Memory Managers like EMM386, QEMM, etc. usually switch to V86 mode to provide their services. Our little program only works in Real Mode, so another function has to be coded.

To distinguish between Real Mode and V86 mode, we have to look at the Control Register 0: bit 0 is clear when we are running in Real Mode, otherwise it is set.

```

; checks if we are running in Real Mode

nrme    db      'You are currently running in V86 mode!',13,10,'$'

proc    check_mode
        mov     eax,cr0        ; get CR0 to EAX
        and     al,1          ; check if PM bit is set
        jnz     not_real_mode ; yes, it is, so exit
        ret                ; no, it isn't, return
not_real_mode:
        mov     dx,offset nrme
        jmp     err16exit
endp    check_mode

```

Now we can be sure that nothing will disturb us while setting up for Protected Mode. DPMS and VCPI (even BIOS) can be used for mode switching too, but this is left as an exercise to you. (the interface however is described below)

Please notice that I used MOV EAX,CR0 in this example. Jerzy Tarasiuk pointed out that this is not allowed in a Protected Mode environment, especially not on a 286. If the program doesn't work on your computer, try SMSW AX. This instruction is only supported on the 386/486 for compatibility reasons and shouldn't be used any more, but it works in any environment.

When switching to Protected Mode, you also have to change MOV CR0,EAX to LMSW AX.

Please note that you can use LMSW only to get \_into\_ Protected Mode. The instruction cannot be used to get \_out\_ of Protected Mode. To handle this weird behaviour, you have to force the processor to enter shutdown mode and then reset it. This is, however, not described here, as it would be a too complex thing for a beginner's tutorial. ;)

-----  
-- TABLES AND DESCRIPTORS AND SELECTORS AND L O T S OF CONFUSING STUFF --

Before the actual mode switch, we have to set up a few tables and descriptors. What I'm talking about is the GDT, the LDT and the IDT.

The GDT is the Global Descriptor Table and contains basic segment descriptors. These segment descriptors are keeping information about different parts of the memory. In Real Mode, one segment is 64kb big followed by the next segment in a 16 byte distance. In Protected Mode however, you can decide yourself where to put a segment. Every segment can be as big as 4Gb (in words: four giga-bytes!). The LDT is optional and contains segment descriptors, too, but these are usually used for applications. An Operating System, for example, could set up the GDT with it's own system descriptors and for every task a LDT which contains the application descriptors.

The LDT is a descriptor table like the GDT. It's usage is to provide different tasks different memory-layouts. In our program, LDTs aren't needed.

The IDT contains the interrupt descriptors. These are used to tell the processor where to find the interrupt handlers. It contains one entry per interrupt, just like in Real Mode, but the format of these entries is totally different.

Here is the basic format of these descriptors:

Segment Descriptor

-----

```

15  14  13  12  11  10  09  08  07  06  05  04  03  02  01  00
[      Base address 31:24      ] [G] [D] [0] [AVL] [Sg. length 19:16]
[P] [DPL] [DT] [      Type      ] [      Base address 23:16      ]
[      Base address 15:00      ]
[      Segment length 15:00    ]

```

As you can see in this figure, the basic segment descriptor has a size of  $4 \cdot 16 = 64$  bits.

To give you a help understanding the structure (my ASCII graphics are not very good at all), here is the same a bit more compact in assembler:

```
; contains a segment descriptor
```

```

struc segment_descriptor
    seg_length0_15    dw    ?    ; low word of the segment length
    base_addr0_15     dw    ?    ; low word of base address
    base_addr16_23    db    ?    ; low byte of high word of base addr.
    flags             db    ?    ; segment type and misc. flags
    access            db    ?    ; highest nibble of segment length
                        ; and access flags
    base_addr24_31    db    ?    ; highest byte of base address
ends segment_descriptor

```

Using this structure makes handling the GDT and LDT much easier. The same applies to the IDT, but here the format is different:

```

15  14  13  12  11  10  09  08  07  06  05  04  03  02  01  00
[      Offset 31:16      ]
[P] [ DPL ] [0] [1] [1] [1] [0] [0] [0] [0] [0] [0] [0] [0] [0]
[      Selector 00:15    ]
[      Offset 00:15     ]

```

And, as above, the same in assembler:

```
; contains an interrupt descriptor
```

```

struc interrupt_descriptor
    offset0_15    dw    ?    ; low word of handler offset
    selector0_15  dw    ?    ; segment selector
    zero_byte     db    0    ; unused in this descriptor format
    flags         db    ?    ; flag-byte
    offset16_31   dw    ?    ; high word of handler offset

```

ends interrupt\_descriptor

Now you know the segment and interrupt descriptor format. But what to fill in? Good question. Before explaining you this, I'll give you a short table where all mentioned bit names are described:

Bit name	Meaning
G	<p>Granularity.</p> <p>G=0 =&gt; 1 Byte granularity G=1 =&gt; 4k granularity</p> <p>This bit specifies the granularity of the segment. If the bit is clear, the length field in the descriptor reflects the real length of the segment in bytes. If the bit is set, you have to multiply the length field in the descriptor by 4096 to get the real length in bytes.</p>
D	<p>Default Operand Size.</p> <p>D=0 =&gt; 16 bit operands D=1 =&gt; 32 bit operands</p> <p>This bit specifies the default operand size which has to be used by special opcodes (like REP xxx). If the bit is clear, the default operand size is 16 bit and the processor behaves similar to a 80286. If the bit is set, the default operand size is 32 bit. D=0 does not mean you can't use 32 bit instructions, it only affects the default operand sizes.</p>
AVL	<p>Available for System.</p> <p>This bit is not used in 80286/80386/80486 machines. If somebody has information about how it is used on Pentium machines, please mail me. For now, better keep it to zero to keep compatibility. However, if your program only runs on machines lower than Pentium, you can use it as a mark for your own software or whatever.</p>
P	<p>Presence.</p> <p>P=0 =&gt; segment is not present (or invalid) P=1 =&gt; segment is present and valid</p> <p>With this bit you can easily implement a virtual memory manager (VMM). If the application wants to allocate more memory than available, save the least used segment (determined with help of the A bit) to disk, then clear the P bit in its descriptor. The next access to that segment will be followed by a General Protection Fault. Catch the fault, reload the segment into memory and set its P bit. Done. The processor checks only the P bit before generating the General Protection Fault, so if P is set to zero, the rest of the descriptor is available to keep information for your VMM.</p>
DPL	<p>Descriptor Privilege Level.</p> <p>0 &lt;= DPL &lt;= 3</p> <p>The DPL bits contain the Descriptor Privilege Level. The Privilege Level has a range from 0 (highest privilege level) to 3 (lowest privilege). If a program tries to access a segment with a higher privilege level than its own, the processor will generate a General Protection Fault.</p> <p>REMARK: Every time I speak of Privilege Levels in this text, I mean that HIGH Privilege Level is a LOW number in DPL, LOW Privilege Level is a HIGH number in DPL.</p> <p>Example: segment 1: DPL=1 \</p> <p style="padding-left: 100px;">-&gt; segment 1 is more privileged</p> <p style="padding-left: 100px;">segment 2: DPL=3 /</p>
DT	<p>Descriptor Type.</p> <p>DT=0 =&gt; System Descriptor (System-Segment or Gate) DT=1 =&gt; Application Descriptor (Data or Code)</p> <p>If this bit is clear, the Descriptor describes a segment that is (a) available for the System Software, or (b) a Gate-Descriptor.</p>
Type	<p>Segment type.</p> <p>These four bits select the segment type.</p>

Bit	3	2	1	0	Type	Description
Name	T	E	W	A		
	0	0	0	0	Data	read-only
	0	0	0	1	Data	read-only, accessed
	0	0	1	0	Data	read/write
	0	0	1	1	Data	read/write, accessed
	0	1	0	0	Data	read-only, expand down
	0	1	0	1	Data	read-only, exp. down, acc.
	0	1	1	0	Data	read-write, expansion down
	0	1	1	1	Data	read-write, exp. down, acc.
	1	0	0	0	Code	exec-only

---

Name	T	C	R	A	Type	Description
	1	0	0	1	Code	exec-only, accessed
	1	0	1	0	Code	exec-read
	1	0	1	1	Code	exec-read, accessed
	1	1	0	0	Code	exec-only, conforming
	1	1	0	1	Code	exec-only, conf., acc.
	1	1	1	0	Code	exec-read, conforming
	1	1	1	1	Code	exec-read, conf., acc.

Read-only means that you are only allowed to read this segment. Read-write means that you can read and write from/to the segment.

Exec-only segments can only be executed, but no read-access is allowed.

Exec-read segments can be read and executed. Unlike as in Real Mode, you aren't allowed to use self-modifying code. A way around this can be found a few lines ahead in this text.

The Accessed bit is set everytime a program tried to access this segment and the bit isn't already set. If you want to figure out which segment to swap (the famous VMM example), increase a counter if the A bit is set and then clear this bit. The segment with the lowest counter position can safely be swapped out. BUT WATCH OUT: If the A bit is set, a program might run in this segment! Swapping these segments may hurt! Expansion Direction is a weird thing. If the bit is clear, the Expansion Direction is upwards, that means the segment grows upwards. To grow it, increase the length. You are allowed to access every address that is

$$0 \leq \text{Address} \leq \text{Limit}$$

Limit means the actual length of the segment. If the segments Granularity is set to 0, the limit is equal to the length. But if Granularity is set to 1, you first have to multiply the length by 4096 (4k) to get the length.

If the bit is set however, welcome to hell. Now the Expansion Direction is downwards, that means to grow the segment, you'll have to decrease the Length. You are allowed to access every address that is

$$G=0 : \text{Limit}-1 \leq \text{Address} \leq 0\text{ffffh}$$

$$G=1 : \text{Limit}-1 \leq \text{Address} \leq 0\text{ffffffh}$$

Because of the 4G Wrap-Around, these addresses are just the ones that would cause a General Protection Fault if E would be zero.

Conforming means that a segment with C=1 can call another segment with a lower or equal Privilege Level. The Current Privilege Level however isn't changed!

If you call directly from a segment with C=0 (and not through a Task-Gate) to a segment that has another Privilege Level than the Current Privilege Level, a General Protection Fault follows.

That wasn't too hard, wasn't it? At first all this might look a bit confusing, but when you look at the code, it will be that simple...

So the only thing left before starting to do the real thing (no, not drinking Coke, I mean coding! :) ) is to explain what a Selector is:

A Selector selects something, and this something are Segment Descriptors!

The format is simple enough to understand:

```

15  14  13  12  11  10  09  08  07  06  05  04  03  02  01  00
[           Pointer into a Descriptor Table           ]  TI  [ RPL ]

```

RPL is the Requested Privilege Level. If the Descriptor in the Descriptor Table has a higher Privilege Level than RPL, a General Protection Fault will be caused.

TI selects the Descriptor Table to get the Descriptor from : TI=0 means GDT, TI=1 means LDT.

The pointer contains the offset into the Descriptor Table where the wanted Descriptor is to find.

-----  
-- NOW FOR THE FUN STUFF!! --

If you understood the above, the rest is easy to do: set up the appropriate tables, then switch to PM. EASY!!!

At first, we declare the two segments where to put all data and code in:

```

segment code16 para public use16          ; <- the 16-bit code and data segment
assume cs:code16, ds:code16

```

```
ends    code16
```

```

segment code32 para public use32          ; <- the 32-bit code and data segment
assume cs:code32, ds:code32

```

```
ends    code32
```

There is only one little bug with these two segments: both are code segments. If you listened carefully, you'd remember that in Protected Mode it is not allowed to write to a code segment. Well, where's the problem? (I hear them saying: let's only use static data!! :) ) Everytime your program would try to write to data located in one of these segments, a General Protection Fault will occur. What about an extra 32-bit data segment? Hmm, nice, but not the best way. Here's the probably easiest way to have code and data together in one segment, even allowing self-modifying code: we have to create a so called "alias"-segment. This segment isn't really a new segment, it's just another descriptor in the GDT. The descriptor points to the same memory that is defined in the code segment descriptor. The only thing that we have to take care of now is that CS is loaded with the code selector and DS, ES, FS and GS are loaded with the data selector.

```
; GDT data
```

```

gdt_reg      dw      gdt_size,0,0
dummy_dscr   segment_descriptor    <0,0,0,0,0>
code32_dscr  segment_descriptor    <0ffffh,0,0,9ah,0cfh,0>
data32_dscr  segment_descriptor    <0ffffh,0,0,92h,0cfh,0>
core32_dscr  segment_descriptor    <0ffffh,0,0,92h,0cfh,0>
code16_dscr  segment_descriptor    <0ffffh,0,0,9ah,0,0>
data16_dscr  segment_descriptor    <0ffffh,0,0,92h,0,0>
gdt_size=$-(offset dummy_dscr)

```

The first line contains three words (better: one word and one dword) that contain information for the GDT register. To inform the CPU where our GDT is located in memory, we have to use the LGDT instruction. This instruction sets an internal CPU register with the data pointed to by the instruction parameter. The format of this data is

- GDT size in bytes (word)
- GDT base address (dword)

so to set up the register, we have to use the following line:

```
lgdt    [fword ds:gdt_reg]
```

And the same for the IDT:

```
lidt    [fword ds:idt_reg]
```

In our GDT, we have defined six descriptors: 32-bit code (4G size, 32-bit operands, code type), 32-bit data (4G size, 32-bit operands, data type), 32-bit core (4G size, 32-bit operands, data type), 16-bit code (64k size, 16-bit operands, code type) and finally 16-bit data (64k size, 16-bit operands, data type).

Whoops, there's one descriptor missing: the dummy descriptor! Why do we have to include something like this? Good question! This descriptor can't be used and has to be set to zero. But that doesn't explain why it is included! The LDT definitely does `_not_` have something like this...

The reason is the concept of the `_Protected_` Mode. The CPU provides several protection mechanisms, and one of them is the "invalid" (zero) descriptor. If a segment is loaded with a zero-selector, every try to access memory through it will be followed by a General Protection Fault, so it can be used as a "marker". This is handy for debuggers if they want to find out when a segment register is used, but there are lots of other possibilities to take advantage of this feature.

A second thing is that the CPU validates every selector before it is loaded into a segment register. This means, that if you want to load a Real Mode segment address (like 1234h) into a segment register, the CPU checks in the GDT if there is a valid descriptor. At offset 1234h, there probably won't, so again a General Protection Fault is generated. In V86 Mode however, the processor works with segment addresses like that. To solve this problem, the CPU saves every segment register before calling an interrupt handler and loads them with the zero selector. The Protected Mode interrupt handler won't notice if it has been called from Protected or V86 Mode, so one handler will work in both modes.

Now, disable interrupts (a Real Mode Interrupt in Protected Mode does you no good, a Protected Mode Interrupt with no IDT may be even worse!),

```
cli
```

and switch to Protected Mode:

```
mov     eax,cr0
or      al,1
mov     cr0,eax
```

(you may use `LMSW AX`, too)

The next thing is dirty but there's no way around it:

```
db      0eah
dw      offset start32, code32_idx
```

0EAh is the opcode for `JMP FAR`. If you use the instruction `JMP FAR`, TASM tries to resolve the jump with the optimal opcode, but we need `JMP FAR` to flush the Instruction Prefetch Queue and to load CS with the new Protected Mode Descriptor. This has to be done because the CPU doesn't set its descriptor caches and the Instruction Prefetch Queue might contain instructions decoded in a way only valid for Real Mode.

After this, the CPU is setup for Protected Mode and starts execution at the label `START32`. There we just load the other segment registers with their Protected Mode selectors and call `MAIN`.

The rest is easy.

-----  
 -- MISSING PARTS IN THE DEMO SOURCE --

There are lots of features not contained in the sample source. I've done this to keep the program as simple as possible and to demonstrate as much as possible, so there is an interrupt-system missing. This thing should intercept every interrupt and redirect them to Real Mode. Second, a routine to toggle the A20 address line should be added. Mail me if you want something to be included or if you already coded something for it. You can also look at Tran's PMODE package. It contains a complete Protected Mode header with complete environment management. You won't notice what it does, you can start coding as if you were in Real Mode, it is very powerful. However, I encourage you to write your own Protected Mode system sometime, it helps to understand the principles.

-----  
 -- SOME CODING TIPS FOR PROTECTED MODE --

This section just can't explain all of the advantages of Protected Mode, better buy a good book, but some of them I'll show you here.

1. To use the JMP FAR back to a 16-bit Real Mode segment from a 32-bit segment, you have to use

```

db      0eah
dw      offset real_mode_proc, 0 , segment_selector
                ^^^

```

This little zero-word can cost some time of debugging.

2. You can use `_every_` register as an index.

```

mov     eax,[edx]

```

3. You can use displacements `_and_` factors in pointers.

```

mov     eax,[ecx*8+edx]

```

All factors have to be powers of 2.

4. Sometimes this feature can be used for quick multiplies:

```

lea     eax,[eax*8+eax] -> EAX=EAX*9

```

5. IMUL accepts immediates:

```

imul   ecx,5                -> ECX=ECX*5

```

6. IMUL accepts immediates `_and_` a register:

```

imul   eax,ecx,5            -> EAX=ECX*5

```

7. Extended form of SHR/SHL to shift across registers:

```

shrd   eax,edx,5
shld   eax,edx,5

```

In this example, the 5 bits that become free in EAX will be filled with bits of EDX. EDX is not modified.

8. This feature can be used to get rid of one MOV instruction:

Assume ECX contains a linear address that you want to convert into a segment:offset notation. Normally, you would use something like this:

```

mov     eax,ecx

```

```
shr    eax,4           ; EAX now contains segment
and    ecx,0fh        ; ECX now contains offset
```

With the SHRD instruction, you can modify it to

```
shld   eax,ecx,28     ; EAX now contains segment
and    ecx,0fh        ; ECX now contains offset
```

REMEMBER: The CPU only uses the 5 lower bits of the shift factor, so  
shld eax,ecx,32 will not copy ECX to EAX!!!

9. You can push immediates:

```
push   12345
```

---

## -- FAULTS, TRAPS AND EXCEPTIONS --

Below is a complete list of the Faults, Traps and Exceptions that may occur. If somebody has information about Exception 17 (Arrangement Error), please mail me.

No.	Name	Type	Error Code	Cause
0	Division by Zero	Fault	no	Someone tried to divide by zero. Same as in Real Mode
1	Single Step	Trap,Fault	no	This interrupt is called after each instruction if the Trap Flag is set
2	Non Maskable Interrupt (NMI)	Abort	no	Heavy hardware failure. Same as in Real Mode.
3	Breakpoint	Trap	no	Used for debugging purposes. Called by special INT3 opcode.
4	Overflow	Trap	no	Called if INTO is executed and the Overflow Bit is set.
5	Bound Range Exceeded	Fault	no	BOUND failed
6	Invalid Opcode	Fault	no	CPU found an invalid opcode. Same as in Real Mode.
7	Coprocessor Not Available	Fault	no	Called if CPU tries to execute ESC or WAIT and EM bit is clear.
8	Double Fault	Abort	yes (always 0)	An exception occurred while another exception handler is active.
9	Coprocessor Segment Overrun	Abort	no	The middle operand of a FPU instruction can't be accessed. Dunno what this should be, i486 doesn't has this exception any more.
10	Invalid TSS	Fault	yes	Tried to switch to a task with an invalid TSS.
11	Segment not Present	Fault	yes	Someone tried to access a segment that had it's Present bit clear.
12	Stack Exception	Fault	yes	Called if stack exceeds it's limits or if selector for SS is invalid

13	General Protection Fault	Fault	yes	Someone tried to access invalid, protected or not-present data.
14	Page Fault	Fault	yes	Called if paging is enabled and an access to an invalid, protected or not-present page occurred.
16	Coprocessor Error	Fault	no	The FPU saw that it was doing something totally wrong... :)
17	Arrangement Error	????	??	This exception occurs only if AC=1, AM=1 and CPL=3. If memory isn't accessed at integral addresses, EXCP17 is generated. (see table below)
0..255	Software Interrupts	Trap	no	If you call one of these interrupts from your program (INT xx), they are handled like Traps.

Faults are documented and recoverable errors. The return address for the IRET instruction (CS:EIP) points to the opcode that caused the Fault. Some Faults have an error code on the stack (see below). To solve the problem, the handler only has to read in the failed opcode and react on it.

Traps are interrupts that are caused by your program (INT xx instruction) or by a debugging mechanism (INT3 or Trap Mode). An error code is never generated. CS:EIP points to the opcode following the one that caused the Trap.

Aborts are only caused if the system tables (GDT, IDT, LDT) are invalid or if there was a hardware failure. They don't allow you to return to your program, nor there's an error code.

The format of the error code:

```

15  14  13  12  11  10  09  08  07  06  05  04  03  02  01  00
[                                     Reserved                                     ]
[           Selector           ]  TI  IDT  EXT

```

Bit name	Meaning
Selector	Selector of segment where the error occurred
TI	Table Indicator (applies only if IDT=0) TI=0 - Selector from GDT TI=1 - Selector from LDT
IDT	Interrupt Descriptor IDT=0 - No Interrupt Gate IDT=1 - Selector from IDT
EXT	External EXT=0 - Program caused Exception EXT=1 - Exception caused by external event

#### Arrangement Check Errors

Data Type	Address has to be dividable by
Word	2
Double Word	4
Floating Point, Single Precision	4
Floating Point, Double Precision	8
Floating Point, Extended Precision	8
Selector	2
48-Bit Farpointer	4

```

Contents of GDTR and IDTR (48 Bits)      4
32-Bit Farpointer                        2
32-Bit Address                           4
Bitstrings                               4
FPU Environment Blocks or FPU State      4 or 2, depending on Operand Length

```

An example how to handle these exceptions is not yet included in the demo source. If someone would like to see more about this -> mail me! (as you might have noticed, I definitely WANT your mails! :) )

---

-- USEFUL INTERRUPTS --

In this section I'll give you some useful interrupt calls that provide handy information or services for Protected Mode.

Format of the entries:

```

Function name
Interrupt number (or CALL address) in hexadecimal numbers
Input registers
Return registers
Notes

```

```

Func:   Get RAM Size
Call:   INT 12h
Input:  ---
Return: AX - Memory size in kb
Notes:  Returns only size of conventional memory

```

```

Func:   Move Block (AH=87h)
Call:   INT 15h
Input:  AH=87h
        CX=Number of Words in Buffer
        ES:SI=Address of Descriptor Table
Return: CY=0 - ok
        CY=1 - error
        AH=Status
Notes:  Transfers a block of max. 64k (max. CX=8000h) via Protected Mode
        to any memory location.
        ES:SI -> Dummy          <--,
        Table begin           --'
        Source Descriptor
        Destination Descriptor
        BIOS Codesegment
        Stacksegment

```

Every entry is 8 bytes long. First entry has to be set to zero.

```

Entry structure: - Segment size (word)
                 - Low Word 24-bit Segment Address (word)
                 - High Byte 24-bit Segment Address (byte)
                 - Access Flag (byte, =93h)
                 - Reserved (word)

```

The last two entries have to be set to zero.

```

Error code in AH:   00 - Transfer completed without error
                   01 - RAM Parity Error
                   02 - Exception
                   03 - A20 failure

```

```

Func:   Extended Memory Size (AH=88h)
Call:   INT 15h
Input:  AH=88h
Return: CY=0 - ok
        AX=ext. memory size in kb
        CY=1 - error

```

Notes: AH=error code (80h - invalid command, 86h - function not supported)  
 Function returns extended memory size stored at address 20h and 31h  
 in CMOS RAM of clock chip. Extended Memory can only be used if  
 base memory is at least 512k big. If HIMEM.SYS is loaded, the function  
 returns 0.

Func: Virtual Mode (AH=89h)  
 Call: INT 15h  
 Input: AH=89h  
 BH=first PIC start  
 BL=second PIC start  
 CX=offset of code segment  
 ES:SI=pointer to Descriptor Table  
 Return: CY=0 - ok  
 AH=0  
 CY=1 - error  
 AH=0ffh

Notes: Function destroys all registers. ES:SI points to Descriptor Table.  
 ES:SI -> Dummy <--,  
 Table begin --'  
 Interrupt Table  
 User Data Segment (DS)  
 User Extra Segment (ES)  
 User Stack Segment (SS)  
 User Code Segment (CS)  
 Internal Code Segment

Entries structured like in function Move Block (AH=87h). Dummy has to  
 be set to zero. Third entry points to IDT built by program (Real Mode  
 structure). Last Descriptor has to be set to zero. BH contains mappings  
 for first PIC (start of first 8 hardware interrupts, i.e. BH=8 if IRQ0  
 should be shifted to IRQ8). BL contains mappings for second PIC.  
 Carry flag has to be cleared before function call. CX contains code  
 segment where execution in V86 should start. After function call no  
 BIOS is available, return to Real Mode only by resetting CPU.

-----  
 -- THE VIRTUAL DMA SPECIFICATION (VDS) --

The VDS provides services to use DMA transfers in Protected Mode with enabled  
 paging mechanism.

I am not sure about the information contained here. Please mail me if there are  
 errors. I hoped it would be handy for you, so it is included.

Error codes used in all functions:

01h	region not in contiguous memory
02h	region crossed a physical alignment boundary
03h	unable to lock pages
04h	no buffer available
05h	region too large for buffer
06h	buffer currently in use
07h	invalid memory region
08h	region wasn't locked
09h	number of physical pages greater than table length
0ah	invalid buffer ID
0bh	copy out of buffer range
0ch	invalid DMA channel number
0dh	disable count overflow
0eh	disable count underflow
0fh	function not supported
10h	reserved flag bits set in DX

Sometimes Descriptor Tables are needed (DMA Descriptor Structure - DDS).  
 Format as following:

Offset	Bytes	Meaning
--------	-------	---------

00h	4	size of region
04h	4	region offset
08h	2	region segment
0ah	2	buffer ID
0ch	4	linear address

Some functions use an extended format (EDDS):

Offset	Bytes	Meaning
00h	4	size of region
04h	4	region offset
08h	2	region segment
0ah	2	reserved
0ch	2	number available
0eh	2	number used
10h	4	linear address (region 0)
14h	4	size in bytes (region 0)
18h	4	linear address (region 1)
1ch	4	size in bytes (region 1)
	.	
	.	
	.	

If there are page tables contained, the following structure applies:

Offset	Bytes	Meaning
00h	4	size of region
04h	4	region offset
08h	2	region segment
0ah	2	reserved
0ch	2	number available
0eh	2	number used
10h	4	Page Table Entry 0
14h	4	Page Table Entry 1
	.	
	.	
	.	

Bits 1-12 of a Page Table Entry should be cleared. Bit 0 has to be set if the page is present and locked.

Func: VDS Get Version (AX=8102h)

Call: INT 4Bh

Input: AX=8102h

DX=0

Return: CY=1 - error

AL=error code

CF=0 - ok

AH=major version number

AL=minor version number

BX=product number

CX=revision number

DX=flags

SI:DI=buffer size

Notes: Flag bits in DX:

Bit	Meaning
0	PC/XT Bus System (1Mb addressable)
1	physical buffer/remap region in 1st Mb
2	automatic remap enabled
3	all memory physically contiguous
4-15	reserved

SI:DI contains maximal size of DMA buffer.

Func: VDS Lock DMA Region (AX=8103h)

Call: INT 4Bh

Input: AX=8103h

DX=Flags  
 ES:SI=DMA Descriptor  
 Return: CY=1 - error  
 AL=error code  
 CY=0 - ok  
 Notes: DX is used as flag register to control the operation.

Bit	Meaning
0	reserved (cleared)
1	copy data to buffer (ignored if bit 2 set)
2	don't allocate buffer if region not contiguous or exceeds physical boundaries (bit 4,5)
3	don't try to automatically remap
4	region must not exceed 64kb
5	region must not exceed 128kb
6-15	reserved (cleared)

Region Size Field in DDS contains size of maximal contiguous memory area. If Carry Flag is clear, area is locked and may not be swapped. Physical Address and Buffer ID are filled by the function. If Buffer ID is 0, no buffer has been allocated.

Func: VDS Unlock DMA Region (AX=8104h)  
 Call: INT 4Bh  
 Input: AX=8104h  
 DX=flags  
 ES:DI=DMA Descriptor  
 Return: CY=1 - error  
 AL=error code  
 CY=0 - ok  
 Notes: Flag bits in DX:

Bit	Meaning
0	reserved (cleared)
1	Copy data from buffer
2-15	reserved (cleared)

Region Size, Physical Address and Buffer ID in DDS have to be filled.

Func: VDS Scatter/Gather Lock Region (AX=8105h)  
 Call: INT 4Bh  
 Input: AX=8105h  
 BX=page offset (not sure about it)  
 DX=flags  
 ES:DI=DMA Descriptor  
 Return: CY=1 - error  
 AL=error code  
 CY=0 - ok  
 Notes: Function is used to lock parts of memory. Useful if parts of memory are swapped out.  
 Flag bits in DX:

Bit	Meaning
0-5	reserved (cleared)
6	return EDDS with page table entries
7	only lock existing pages, fill not existing pages with 0
8-15	reserved (cleared)

Region Size, Linear Segment, Linear Offset and Number Available have to be set. Region Size Field in EDDS will be filled with size of largest contiguous memory block. Number Used will be filled with the number of used pages. If bit 6 in DX is set, lower 12 bits of BX should contain offset of first page (not sure about that).

Func: VDS Scatter/Gather Unlock Region (AX=8106h)  
 Call: INT 4Bh  
 Input: AX=8106h  
 DX=Flags  
 ES:DI=DMA Descriptor  
 Return: CY=1 - error

AL=error code

CY=0 - ok

Notes: Flag bits in DX:

Bit	Meaning
0-5	reserved (cleared)
6	EDDS contains page table entries
7	EDDS may contain not present pages
8-15	reserved (cleared)

ES:DI contains EDDS initialised by function 8105h.

Func: VDS Request DMA Buffer (AX=8107h)

Call: INT 4Bh

Input: AX=8107h

DX=Flags

ES:DI=DMA Descriptor

Return: CY=1 - error

AL=error code

CY=0 - ok

Notes: Flag bits in DX:

Bit	Meaning
0	reserved (cleared)
1	Copy data to buffer
2-15	reserved (cleared)

ES:DI contains pointer to DDS. Region Size has to be filled. If bit 1 in DX is set, Region Offset and Region Segment have to be filled, too. Function returns Physical Address, Buffer ID and Region Size.

Func: VDS Release DMA Buffer (AX=8108h)

Call: INT 4Bh

Input: AX=8108h

DX=flags

ES:DI=DMA Descriptor

Return: CY=1 - error

AL=error code

CY=0 - ok

Notes: Flag bits in DX:

Bit	Meaning
0	reserved (cleared)
1	copy data from buffer
2-15	reserved (cleared)

Buffer ID in DDS has to be filled. If bit 1 in DX is set, Region Size, Region Offset and Region Segment have to be initialised, too.

Func: VDS Copy into DMA Buffer (AX=8109h)

Call: INT 4Bh

Input: AX=8109h

DX=0

ES:DI=DMA Descriptor

BX:CX=offset

Return: CY=1 - error

AL=error code

CY=0 - ok

Notes: BX:CX contains offset into DMA Buffer. ES:DI contains pointer to DDS. Buffer ID, Region Offset, Region Segment and Region Size have to be initialised.

Func: VDS Copy out of DMA Buffer (AX=810ah)

Call: INT 4Bh

Input: AX=810ah

DX=0

ES:DI=DMA Descriptor

BX:CX=offset

Return: CY=1 - error

AL=error code

CY=0 - ok  
 Notes: BX:DX contains offset into DMA Buffer. ES:DI contains pointer to DDS.  
 Buffer ID, Region Offset, Region Segment and Region Size have to be  
 initialised.

Func: VDS Disable DMA Translation (AX=810bh)  
 Call: INT 4Bh  
 Input: AX=810bh  
 DX=0  
 BX=DMA channel  
 Return: CY=1 - error  
 AL=error code  
 CY=0 - ok  
 Notes: Function stops DMA transfer on channel BX.

Func: VDS Enable DMA Translation (AX=810ch)  
 Call: INT 4Bh  
 Input: AX=810ch  
 DX=0  
 BX=DMA channel  
 Return: CY=1 - error  
 AL=error code  
 CY=0 - ok  
 Notes: Function starts DMA transfer on channel BX.

-----  
 -- THE VIRTUAL CONTROL PROGRAM INTERFACE (VCPI) --

The Virtual Control Program Interface (VCPI) was the first standard to manage memory in a Protected Mode or Virtual 86 Mode environment. It has been founded in 1987 by many different companies. (PharLap, Quarterdeck, Qualitas, LOTUS, Autodesk and others)

The communication between the interface and the application is divided into a Server and a Client. The program that provides the interface services is recognized as the Server. The application will be the Client.

To call the Server, there are two ways: in Real Mode, you have to use INT 67h, in Protected Mode, you have to use a FAR CALL.

Everytime I speak of page addresses, I mean the common format to address a page (bits 31-22=page directory, bits 21-12=directory entry, bits 11-0=offset). The offset part of the page address is normally always set to 0.  
 [Page Directory

I am not sure about the information contained here. Please mail me if there are errors. I hoped it would be handy for you, so it is included.

Func: VCPI Installation Check (INT 67h, AX=DE00h)  
 Call: INT 67h  
 Input: AX=DE00h  
 Return: AH=0 - VCPI is available  
 BH=major version number  
 BL=minor version number  
 AH!=0 - VCPI not available  
 Notes: Some docs say that AH=84h on return if VCPI isn't available, but EMM is enabled.

Func: VCPI Get Protected Mode Interface (INT 67h, AX=DE01h)  
 Call: INT 67h  
 Input: AX=DE01h  
 DS:SI=pointer to descriptors  
 ES:DI=pointer to client pages

Return: AH=0 - ok  
DI=pointer into page directory  
EBX=offset of entry point  
AH!=0 - error

Notes: To call the Server in Protected Mode, you have to use the returned address. The memory at DS:SI has to have enough space for three GDT Descriptors, the first descriptor will be filled with the VCPI code segment. Use a FAR CALL into this segment, offset EBX, to reach the Server dispatcher. The space has to be in the first page in the applications code segment. ES:DI has to contain a pointer to a list of pages used by the Client. In DI, a pointer to the first unused page is returned.

Func: VCPI Get Maximum Physical Memory (INT 67h, AX=DE02h)  
Call: INT 67h  
Input: AX=DE02h  
Return: AH=0 - ok  
EDX=page address  
AH!= - error

Notes: EDX contains the address of the highest 4kb page in memory. The lowest 12 bits are set to zero. Some Clients are using this call to initialize their data structures.

Func: VCPI Get Number of Free 4K Pages (INT 67h / CALL FAR, AX=DE03h)  
Call: INT 67h / CALL FAR  
Input: AX=DE03h  
Return: AH=0 - ok  
EDX=number of free pages  
AH!=0 - error

Notes: The call returns the number of free pages that are available for all tasks. This function is available in Protected Mode, too. (CALL FAR...)

Func: VCPI Allocate a 4K Page (INT 67h / CALL FAR, AX=DE04h)  
Call: INT 67h / CALL FAR  
Input: AX=DE04h  
Return: AH=0 - ok  
EDX=page address  
AH!=0 - error

Notes: The function allocates a 4K page for the Client. The lowest 12 bits of EDX are set to 0. This function is available in Protected Mode, too.

Func: VCPI Free a 4K Page (INT 67h / CALL FAR, AX=DE05h)  
Call: INT 67h / CALL FAR  
Input: AX=DE05h  
EDX=page address  
Return: AH=0 - ok  
AH!=0 - error

Notes: The page has to be allocated by function DE04h. The lowest 12 bits of EDX are set to 0. This function is available in Protected Mode, too.

Func: VCPI Get Physical Address of Page in First MB (INT 67h, AX=DE06h)  
Call: INT 67h  
Input: AX=DE06h  
CX=page number  
Return: AH=0 - ok  
EDX=page address  
AH!=0

Notes: The function returns the address of a page in the first MB. The lowest 12 bits of EDX are set to zero. The page number in CX is the address of the page SHL 12. (This is written in my VCPI docs, but quite illogical, because then there are only the 4 highest bits available for the page number)

Func: VCPI Read CR0 (INT 67, AX=DE07h)  
 Call: INT 67h  
 Input: AX=DE07h  
 Return: AH=0 - ok  
 EBX=CR0  
 AH!=0 - error

Notes: The function returns CR0 in EBX because MOV xxx,CR0 isn't allowed in V86 Mode. However, EMM386 and QEMM simulate this instruction and you don't have to use an interrupt call.

Func: VCPI Read Debug Register (INT 67h, AX=DE08h)  
 Call: INT 67h  
 Input: AX=DE08h  
 ES:DI=pointer to buffer  
 Return: AH=0 - ok  
 AH!=0 - error

Notes: ES:DI has to provide enough space for 8 entries, every entry has a size of 4 bytes. The function stores DR0, DR1, ..., DR7. DR4 and DR5 are unused.

Func: VCPI Set Debug Register (INT 67h, AX=DE09h)  
 Call: INT 67h  
 Input: AX=DE09h  
 ES:DI=pointer to buffer  
 Return: AH=0 - ok  
 AH!=0 - error

Notes: ES:DI has to point to a table with 8 entries, every entry has a size of 4 bytes. The function loads DR0, DR1, ..., DR7. DR4 and DR5 are unused.

Func: VCPI Get 8259 Interrupt Vector Mappings (INT 67h, AX=DE0Ah)  
 Call: INT 67h  
 Input: AX=DE0Ah  
 Return: AH=0 - ok  
 BX=1st PIC Vector Map  
 CX=2nd PIC Vector Map  
 AH!=0 - error

Notes: The Server returns the mapping from the Master PIC in BX (start of first 8 hardware IRQs) and the mapping from the Slave PIC in CX (start of next 8 hardware IRQs). If there's no Slave PIC installed, CX is undefined.

Func: VCPI Set 8259 Interrupt Mappings (INT 67h, AX=DE0Bh)  
 Call: INT 67h  
 Input: AX=DE0Bh  
 BX=1st PIC Vector Map  
 CX=2nd PIC Vector Map  
 Return: AH=0 - ok  
 AH!=0 - error

Notes: Master PIC is programmed with BX, Slave PIC is programmed with CX. Interrupts have to be disabled before calling this function.

Func: VCPI Switch to Protected Mode (INT 67h, AX=DE0Ch)  
 Call: INT 67h  
 Input: AX=DE0Ch  
 ESI=pointer to data structure  
 Return: AH=0 - ok  
 AH!=0 - error

Notes: The data structure has to be setup by the Client in the first MB. ESI has to contain the linear address of it. Structure as follows:

Offset	Bytes	Meaning
00h	4	new value for CR3
04h	4	linear address in first MB of value for

		GDT register (6 bytes)
08h	4	linear address in first MB of value for IDT register (6 bytes)
0Ch	2	selector for LDT register
0Eh	2	selector for Task Register
10h	4	EIP of Protected Mode entry point
14h	2	CS of Protected Mode entry point

The function loads GDTR, IDTR, LDTR and TR. SS:ESP has to point to a stack with at least 16 bytes available on entry. EAX, ESI, DS, ES, FS and GS are destroyed.

The CPU continues execution in Protected Mode at address CS:EIP specified in the table.

Interrupts are disabled on return.

Func: Switch from Protected Mode to V86 Mode (CALL FAR, AX=DE0Ch)

Call: CALL FAR

Input: AX=DE0Ch

DS=segment selector

Return: ---

Notes: The stack has to be shifted in the first MB on entry. DS has to contain a selector for a segment that includes the address area returned by function DE01h.

The function switches to V86 mode. Interrupt have to be disabled.

GDTR, IDTR, LDTR and TR are initialised by the Server. SS:ESP has to contain the following structure:

Offset	Meaning
-28h	GS
-24h	FS
-20h	DS
-1Ch	ES
-18h	SS
-14h	ESP
-10h	reserved
-0Ch	CS
-08h	EIP
00h	return address

EAX is destroyed.

---

-- THE DOS PROTECTED MODE INTERFACE (DPMI) --

After the successful VCPI standard, a new standard was founded: DPMI. With the DPMI, some "bugs" of VCPI were removed, for example VCPI allowed to run a task on CPL=0. DPMI has been published in 1990 by Microsoft and Intel with version 0.9, in 1991 version 1.0 has been published.

All DPMI functions are reentrant. Many implementations use a VMM, so be sure to lock your memory if you don't want it to be swapped to disk.

Every DPMI task uses four stacks: - Protected Mode Application Stack (CPL=0)  
- Locked Protected Mode Stack  
- Real Mode Stack  
- DPMI Host Stack (CPL=0).

The Protected Mode Stack is used by the Client when switching from Real- to Protected Mode. The Locked Protected Mode Stack is used by the DPMI Server to simulate hardware IRQs. The Real Mode Stack is used for Real Mode IRQs and the DPMI Host Stack is used by (who else could it have been... ;) ) the DPMI Host.

Unlike VCPI, all Protected Mode function can be reached via INT 31h.

I didn't include the interface here, because it is nearly 70 (140 on screen) pages "small". I'd really like to know if someone knows a source, but if there's enough demand, I'll include the whole interface here.

---

-- OTHER TEXTS --

This document has been created to be part of the comp.lang.asm.x86 FAQ, section 13 (Real Mode/Protected Mode). Jerzy Tarasiuk (the author of the directly in the FAQ included text) and I cooperate on writing and extending the FAQ contribution. (Again, please mail us about something you want to have included, I LOVE RECEIVING MAILS! ;) )

His texts are on zfja-gate.fuw.edu.pl:cpu/protect.mod/\* . There is also a listserver, to get the files mail to listserv@zfja-gate.fuw.edu.pl with the body of the letter containing

GET/CPU/PROTECT.MOD/\*.ZIP  
GET/CPU/PROTECT.MOD/\*.TXT

to get all of his files (about 200k). Subjects are switching from Real Mode to Protected Mode, V86 Mode, basic multitasking and using Protected Mode with Turbo Pascal (NOT the Turbo Pascal DPMI stuff!!).

Jerzy's internet address is: JT@zfja-gate.fuw.edu.pl

Another really good choice is Tran's Protected Mode package. I don't know which ftp server has this file available, but it should be on x2ftp.oulu.fi (teeri.oulu.fi). However, if you have Fidonet access, you can request it at 2:2426/2030 (file PMC100.ZIP). This file contains a complete DOS Extender for BC++ 4.0 (can be used with Watcom, too), including all sources.

For those German speaking people out there, I suggest reading the following book:

Author: Dr. Wolfgang Matthes  
Title: Intel's i486  
Architektur und Befehlsbeschreibung  
der xxx86er-Familie  
Published by: Elektor  
ISBN: 3-928051-29-6