# DJGPP and Protected Mode Programming

## Interrupts

I’ve found that dealing with interrupts requires a bit more thought in Protected Mode. You have to decide wether to keep your interrupt a Real-Mode or change to a Protected Mode Interrupt. There are some major differences between the two! Keep in mind that the DPMI host intercepts all interrupt requests. If it is a real mode interrupt, then the system has to switch back into Real Mode to execute the request, immediately after switching back to Protected Mode. This switching is not exactly the fastest way of getting it done. If it is a Protected Mode interrupt request then away it goes, no switching necessary! For setting a PROTECTED mode interrupt do something like this:

```
_go32_dpmi_seginfo OldINT,NewINT;

NewINT.pm_offset =(int)MY_ISR;
NewINT.pm_selector=_go32_my_cs();
_go32_dpmi_get_protected_mode_interrupt_vector(InterruptNumber,&OldINT);
_go32_dpmi_allocate_iret_wrapper(&NewINT);
_go32_dpmi_set_protected_mode_interrupt_vector(InterruptNumber,&NewINT);
```

The allocate_iret_wrapper function insures that all of the overhead of calling your interrupt will be taken care of. You don’t need this if you wish take care of this work for yourself. Here MY_ISR is your isr function. It will probably be void, but the (int) is needed. At termination time of your program, be sure to run these functions to set things back to normal:

```
_go32_dpmi_set_protected_mode_interrupt_vector(InterruptNumber,&OldINT);
_go32_dpmi_free_iret_wrapper(&NewINT);
```

*Do something like this for a REAL mode interrupt:*

```
_go32_dpmi_seginfo OldINT,NewINT;
_go32_dpmi_registers regs;

_go32_dpmi_get_real_mode_interrupt_vector(InterruptNumber,&OldINT);
_go32_dpmi_allocate_real_mode_callback_iret(&NewINT,&regs);
_go32_dpmi_set_real_mode_interrupt_vector(InterruptNumber,&NewINT);
```

And before exiting:

```
_go32_dpmi_set_real_mode_interrupt_vector(InterruptNumber,&OldINT);
_go32_dpmi_free_real_mode_callback(&NewINT);
```

## Absolute (System) Addreses

In Protected Mode, you are no longer dealing with a Seg:Off pair for memory. You are dealing with a “flat” memory model in which memory is directly accessed. You can thank good ole 32-bit for this. If you are in dire need of a system address, like Video Ram, then you can cheat a little and still get the correct result. You can go about this a variety of ways. I prefer just to multiply the segment * 16 and add the offset. Video Ram now starts at 0xA000*16. The same applies to other system addreses like the character table etc. I like to use this function defined in the DJGPP FAQ:

```
 void * MK_FP(unsigned short seg, unsigned short ofs)
  {if(!(_crt0_startup_flags & _CRT0_FLAG_NEARPTR))
    if(!__djgpp_nearptr_enable())
            return (void*)0;
    return (void *)(seg*16+ofs+__djgpp_conventional_base);
  }
```

## Virtual Memory

Virtual memory can be a blessing or a nightmare. If you are switching your code over from a real mode compiler this can be really tricky! Lets say you use some memory sensitive functions, like allocating a DMA buffer that doesn’t overlap a page. Instead of just doing a c or malloc(), you have to go a different route. This is because 98% of the memory allocating functions use virtual memory. The cool thing about this is that before when you had 64 megs of ram and wanted to access it in real mode, you had to go through cryptic addressing schemes, like EMS or XMS. Now just allocate 64 megs just like normal memory and away you go! If you absolutely need to get a hold of some normal DOS memory, use __dpmi_allocate_dos_meory(...) This will do the trick. The cool thing about DJGPP is that it gives you the option to address memory however you need it. If you are used to using the usual SEG:OFF to access memory, then it offers ways to accommodate this.

The big thing i’d like to stress is that DPMI is called Dos PROTECTED Mode Interface. The reason for this is because the system is supposed to be “protected” from your program. Please take this into consideration if you are thinking of using the SEG:OFF scheme (described int the DJGPP FAQ). It can result in program instability if you are not careful! It is only there if you Really Really Really need it. Get my point? :-)

## BIOS Issues

Some BIOS functions require the programmer to pass a Segment and Offset in order for them to do their work. This is impossible under a Protected Mode since memory is directly accessable, there are no Segments or Offsets! Also, they need to have a buffer that resides in lower memory. Since we are in Protected Mode, we use Virtual Memory, so we don’t know where a buffer will reside should we allocate it! The answer to this dilema is to do a little trick. First, DJGP provides us with a transfer buffer that is located in lower memory called __tb. We can use it like this in our example:

```
 int Video::Detect_VESA()
 {__dpmi_regs r;
   r.x.ax=0x4F00;
   r.x.di=__tb&0x0f;
    r.x.es=(__tb >>4)&0xFFFF;
    dosmemput(&vesainfo,sizeof(vesainfo),__tb);
    __dpmi_int(0x10,&r);
   dosmemget(__tb,sizeof(vesainfo),&vesainfo);
 . . .
```

This snippet is from my VESA Video Modes tutorial. The BIOS function needs a SEG and OFFSET in the DI and ES registers. We fill in __tb with a structure using dosmemput, call the interrupt and then call dosmemget to put it back into our structure. Pretty slick huh!

## Inline Assembly

The Inline Assembly syntax of DJGPP is pretty screwed up, but for good reason. It's one of the most versatile and powerful inline assembler around! I'll start with the basic syntax and provide you with a couple of examples to get you started.

```
asm("cli");
asm("sti");
```

If you only need basic assembly instructions, you can get away with enclosing them with quotes, and then you're ready to rock. For more advanced usages, which most of you will, you'll need to gain a grasp on extended assembly format!

```
asm(" Asm commands"
    : "Output Register Listing"
    : "Input Register Listing"
    : "Trashed Register Listing");
```

Ok, here's the general form. We start with the asm( keyword followed by the left parenth. All of our code and register listings are going to be contained between that heading and ended with a ); Our assembly commands are going to be in At&t syntax which means the source and destination operands are going to be src,dest or coming from the register on the left and placed in the register on the right. This is completely backwards compared to Intel syntax which is dst,src. I like At&t's syntax simply because it feels a lot more natural to go from left to right. All of you old fashioned assemblers might get a little frustrated at first with this, but believe me it's worth it! Our commands also end with a letter defining its size. For instance the mov instruction will end in l,w or c if it is of type long (4 bytes), word (2 bytes) or character (1 byte). Here's a little example that uses one assembler operand and 1 register!

```
asm("movl $4,%%eax"
     :
     :
     :"%eax");
```

| Secret Code | Register |
|---|---|
| a | eax |
| b | ebx |
| c | ecx |
| d | edx |
| S | esi |
| D | edi |
| I | constant value (1-31) |
| g | eax,ebx,ecd,edx or value in memory |
| A | eax combined with edx for a 64 bit register |

Here we simply execute 1 assembly command. We move the number 4 into the ax register 4 bytes at a time. We didn't save what was in the EAX register ourselves, so we put that register in the trashed register listing. Notice that in our code section, we refer to registers with a double %% and a single % in our trashed register listing. This is one wonderful quirk about the assembly parser that we have to deal with. We just have to do it this way :) We have not input or output registers, so we leave those blank. Now let's get into an example that uses input registers.

```
void Video::MoveMem()
{asm (“ LOOPB:
        movl (%%esi),%%eax
        movl 4(%%esi),%%ebx
        movl %%eax, (%%edi)
        movl %%ebx,4(%%edi)
        addl $8,%%esi
        addl $8,%%edi
        decl %%ecx
        jnz LOOPB”
        : /*No output*/
        :”c” (Screen_Size>>2), “D” (&video_screen[0]), “S”(&video_buffer[0])
        :”%eax”,”%ebx”);
}
```

Here's a portion of my video library. This function simply moves an offscreen buffer to video ram using inline assembly. The main assembly code isn't important here. I want you to focus on what is in the Input Register field. Here we have 3 variables being loaded into 3 different registers, but how can we tell what is going where!? This little table should answer that question:
Notice in our example how our secret codes are in quotes, immediately followed by a variable name. According to this table, I'm putting my screen size (divided by 2) into the ecx register, the beginning of video ram into the edi register, and the beginning of my offscreen buffer into the esi register! You should also notice that I only have 2 registers listed in the trashed register listing. This is because if we preload registers like the 3 we are using now, they will automatically be treated as trashed since the compiler knows for certain that they will be! The output listing has the same kind of format and requirements except that when we list our secret code, we have to use the = sign before it (=s,=D etc).

## Contact Information

I just wanted to mention that everything here is copyrighted, feel free to distribute this document to anyone you want, just don't modify it!  You can get a hold of me through my website or direct email.  Please feel free to email me about anything.  I can't guarantee that I'll be of ANY help, but I'll sure give it a try :-)

Email : deltener@mindtremors.com
Webpage : www.inversereality.org