# The GDT

A vital part of the 386's various protection measures is the Global Descriptor Table, otherwise called a GDT. The GDT defines base access privileges for certain parts of memory. We can use an entry in the GDT to generate segment violation exceptions that give the kernel an opportunity to end a process that is doing something it shouldn't. Most modern operating systems use a mode of memory called "Paging" to do this: It is alot more versatile and allows for higher flexibility. The GDT can also define if a section in memory is executable or if it is infact, data. The GDT is also capable of defining what are called Task State Segments (TSSes). A TSS is used in hardware-based multitasking, and is not discussed here. Please note that a TSS is not the only way to enable multitasking.

Note that GRUB already installs a GDT for you, but if we overwrite the area of memory that GRUB was loaded to, we will trash the GDT and this will cause what is called a 'triple fault'. In short, it'll reset the machine. What we should do to prevent that problem is to set up our own GDT in a place in memory that we know and can access. This involves building our own GDT, telling the processor where it is, and finally loading the processor's CS, DS, ES, FS, and GS registers with our new entries. The CS register is also known as the Code Segment. The Code Segment tells the processor which offset into the GDT that it will find the access privileges in which to execute the current code. The DS register is the same idea, but it's not for code, it's the Data segment and defines the access privileges for the current data. ES, FS, and GS are simply alternate DS registers, and are not important to us.

The GDT itself is a list of 64-bit long entries. These entries define where in memory that the allowed region will start, as well as the limit of this region, and the access privileges associated with this entry. One common rule is that the first entry in your GDT, entry 0, is known as the NULL descriptor. No segment register should be set to 0, otherwise this will cause a General Protection fault, and is a protection feature of the processor. The General Protection Fault and several other types of 'exceptions' will be explained in detail under the section on Interrupt Service Routines (ISRs).

Each GDT entry also defines whether or not the current segment that the processor is running in is for System use (Ring 0) or for Application use (Ring 3). There are other ring types, but they are not important. Major operating systems today only use Ring 0 and Ring 3. As a basic rule, any application causes an exception when it tries to access system or Ring 0 data. This protection exists to prevent an application from causing the kernel to crash. As far as the GDT is concerned, the ring levels here tell the processor if it's allowed to execute special privileged instructions. Certain instructions are privileged, meaning that they can only be run in higher ring levels. Examples of this are 'cli' and 'sti' which disable and enable interrupts, respectively. If an application were allowed to use the assembly instructions 'cli' or 'sti', it could effectively stop your kernel from running. You will learn more about interrupts in later sections of this tutorial.

Each GDT entry's Access and Granularity fields can be defined as follows:

| 7 | 6 | 5 | 4 | 3 | 0 |
|---|---|---|---|---|---|
| P | DPL | | DT | Type | |

P - Segment is present? (1 = Yes)

| 7 | 6 | 5 | 4 | 3 | 0 |
|---|---|---|---|---|---|
| G | D | 0 | A | Seg Len. 19:16 | |

DPL - Which Ring (0 to 3)
DT - Descriptor Type
Type - Which type?

G - Granularity (0 = 1byte, 1 = 4kbyte)
D - Operand Size (0 = 16bit, 1 = 32-bit)
0 - Always 0
A - Available for System (Always set to 0)

In our tutorial kernel, we will create a GDT with only 3 entries. Why 3? We need one 'dummy' descriptor in the beginning to act as our NULL segment for the processor's memory protection features. We need one entry for the Code Segment, and finally, we need one entry for the Data Segment registers. To tell the processor where our new GDT table is, we use the assembly opcode 'lgdt'. 'lgdt' needs to be given a pointer to a special 48-bit structure. This special 48-bit structure is made up of 16-bits for the limit of the GDT (again, needed for protection so the processor can immediately create a General Protection Fault if we want a segment whose offset doesn't exist in the GDT), and 32-bits for the address of the GDT itself.

We can use a simple array of 3 entries to define our GDT. For our special GDT pointer, we only need one to be declared. We call it 'gp'. Create a new file, 'gdt.c'. Get gcc to compile your 'gdt.c' by adding a line to your 'build.bat' as outlined in previous sections of this tutorial. Once again, I remind you to add 'gdt.o' to the list of files that LD needs to link in order to create your kernel! Analyse the following code which makes up the first half of 'gdt.c':

```
#include < system.h >

/* Defines a GDT entry. We say packed, because it prevents the
*  compiler from doing things that it thinks is best: Prevent
*  compiler "optimization" by packing */
struct gdt_entry
{
    unsigned short limit_low;
    unsigned short base_low;
    unsigned char base_middle;
    unsigned char access;
    unsigned char granularity;
    unsigned char base_high;
} __attribute__((packed));

/* Special pointer which includes the limit: The max bytes
*  taken up by the GDT, minus 1. Again, this NEEDS to be packed */
struct gdt_ptr
{
    unsigned short limit;
    unsigned int base;
} __attribute__((packed));

/* Our GDT, with 3 entries, and finally our special GDT pointer */
struct gdt_entry gdt[3];
struct gdt_ptr gp;

/* This will be a function in start.asm. We use this to properly
*  reload the new segment registers */
extern void gdt_flush();
```

*Managing your GDT with 'gdt.c'*

You will notice that we added a declaration for a function that does not exist yet: gdt_flush(). gdt_flush() is the function that actually tells the processor where the new GDT exists, using our special pointer that includes a limit as seen above. We need to reload new

segment registers, and finally do a far jump to reload our new code segment. Learn from this code, and add it to 'start.asm' right after the endless loop after 'stublet' in the blank space provided:

```
; This will set up our new segment registers. We need to do
; something special in order to set CS. We do what is called a
; far jump. A jump that includes a segment as well as an offset.
; This is declared in C as 'extern void gdt_flush();'
global _gdt_flush     ; Allows the C code to link to this
extern _gp            ; Says that '_gp' is in another file
_gdt_flush:
    lgdt [_gp]        ; Load the GDT with our '_gp' which is a special pointer
    mov ax, 0x10      ; 0x10 is the offset in the GDT to our data segment
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax
    jmp 0x08:flush2   ; 0x08 is the offset to our code segment: Far jump!
flush2:
    ret               ; Returns back to the C code!
```

*Add these lines to 'start.asm'*

It's not enough to actually reserve space in memory for a GDT. We need to write values into each GDT entry, set the 'gp' GDT pointer, and then we need to call gdt_flush() to perform the update. There is a special function which follows, called 'gdt_set_entry()', which does all the shifts to set each field in the given GDT entry to the appropriate value using easy to use function arguments. You must add the prototypes for these 2 functions (at very least we need 'gdt_install') into 'system.h' so that we can use them in 'main.c'. Analyse the following code - it makes up the rest of 'gdt.c':

```
/* Setup a descriptor in the Global Descriptor Table */
void gdt_set_gate(int num, unsigned long base, unsigned long limit, unsigned char access, unsigned char gran)
{
    /* Setup the descriptor base address */
    gdt[num].base_low = (base & 0xFFFF);
    gdt[num].base_middle = (base >> 16) & 0xFF;
    gdt[num].base_high = (base >> 24) & 0xFF;

    /* Setup the descriptor limits */
    gdt[num].limit_low = (limit & 0xFFFF);
    gdt[num].granularity = ((limit >> 16) & 0x0F);

    /* Finally, set up the granularity and access flags */
    gdt[num].granularity |= (gran & 0xF0);
    gdt[num].access = access;
}

/* Should be called by main. This will setup the special GDT
*  pointer, set up the first 3 entries in our GDT, and then
*  finally call gdt_flush() in our assembler file in order
*  to tell the processor where the new GDT is and update the
*  new segment registers */
void gdt_install()
{
    /* Setup the GDT pointer and limit */
    gp.limit = (sizeof(struct gdt_entry) * 3) - 1;
    gp.base = &gdt;

    /* Our NULL descriptor */
    gdt_set_gate(0, 0, 0, 0, 0);

    /* The second entry is our Code Segment. The base address
```

```
    *  is 0, the limit is 4GBytes, it uses 4KByte granularity,
    *  uses 32-bit opcodes, and is a Code Segment descriptor.
    *  Please check the table above in the tutorial in order
    *  to see exactly what each value means */
    gdt_set_gate(1, 0, 0xFFFFFFFF, 0x9A, 0xCF);

    /* The third entry is our Data Segment. It's EXACTLY the
    *  same as our code segment, but the descriptor type in
    *  this entry's access byte says it's a Data Segment */
    gdt_set_gate(2, 0, 0xFFFFFFFF, 0x92, 0xCF);

    /* Flush out the old GDT and install the new changes! */
    gdt_flush();
}
```

*Add this to 'gdt.c'. It does some of the dirty work relating to the GDT!*
*Don't forget the prototypes in 'system.h'!*

Now that our GDT loading infrastructure is in place, and we compile and link it into our kernel, we need to call gdt_install() in order to actually do our work! Open 'main.c' and add 'gdt_install();' as the very first line in your main() function. The GDT needs to be one of the very first things that you initialize because as you learned from this section of the tutorial, it's very important. You can now compile, link, and send our kernel to our floppy disk to test it out. You won't see any visible changes on the screen: this is an internal change. Onto the Interrupt Descriptor Table (IDT)!