The IDT

The Interrupt Descriptor Table, or IDT, is used in order to show the processor what Interrupt Service Routine (ISR) to call to handle either an exception or an 'int' opcode (in assembly). IDT entries are also called by Interrupt Requests whenever a device has completed a request and needs to be serviced. Exceptions and ISRs are explained in greater detail in the next section of this tutorial, accessible here.

Each IDT entry is similar to that of a GDT entry. Both have hold a base address, both hold an access flag, and both are 64-bits long. The major differences in these two types of descriptors is in the meanings of these fields. In an IDT, the base address specified in the descriptor is actually the address of the Interrupt Service Routine that the processor should call when this interrupt is 'raised' (called). An IDT entry doesn't have a limit, instead it has a segment that you need to specify. The segment must be the same segment that the given ISR is located in. This allows the processor to give control to the kernel through an interrupt that has occured when the processor is in a different ring (like when an application is running).

The access flags of an IDT entry are also similar to a GDT entry's. There is a field to say if the descriptor is actually present or not. There is a field for the Descriptor Privilege Level (DPL) to say which ring is the highest number that is allowed to use the given interrupt. The major difference is the rest of the access flag definition. The lower 5-bits of the access byte is always set to 01110 in binary. This is 14 in decimal. Here is a table to give you a better graphical representation of the access byte for an IDT entry.

7	6	5	4				0
P	DPL			Always 01110 (14)			

P - Segment is present? (1 = Yes)
DPL - Which Ring (0 to 3)

Create a new file in your kernel directory called 'idt.c'. Edit your 'build.bat' file to add another line to make GCC also compile 'idt.c'. Finally, add 'idt.o' to the ever growing list of files that LD needs to link together to create your kernel. 'idt.c' will declare a packed structure that defines each IDT entry, the special IDT pointer structure needed to load the IDT (similar to loading a GDT, but alot less work!), and also declare an array of 256 IDT entries: This will become our IDT.

```
#include < system.h >

/* Defines an IDT entry */
struct idt_entry
{
    unsigned short base_lo;
    unsigned short sel;        /* Our kernel segment goes here! */
    unsigned char always0;     /* This will ALWAYS be set to 0! */
    unsigned char flags;       /* Set using the above table! */
    unsigned short base_hi;
} __attribute__((packed));

struct idt_ptr
{
```

```
    unsigned short limit;
    unsigned int base;
} __attribute__((packed));

/* Declare an IDT of 256 entries. Although we will only use the
*  first 32 entries in this tutorial, the rest exists as a bit
*  of a trap. If any undefined IDT entry is hit, it normally
*  will cause an "Unhandled Interrupt" exception. Any descriptor
*  for which the 'presence' bit is cleared (0) will generate an
*  "Unhandled Interrupt" exception */
struct idt_entry idt[256];
struct idt_ptr idtp;

/* This exists in 'start.asm', and is used to load our IDT */
extern void idt_load();
```

This is the beginning half of 'idt.c'. Defines the vital data structures!

Again, like 'gdt.c', you will notice that there is a declaration of a function that physically exists in another file. 'idt_load' is written in assembly language just like 'gdt_flush'. All 'idt_load' is is calling the 'lidt' assembly opcode using our special IDT pointer which we create later in 'idt_install'. Open up 'start.asm', and add the following lines right after the 'ret' for '_gdt_flush':

```
; Loads the IDT defined in '_idtp' into the processor.
; This is declared in C as 'extern void idt_load();'
global _idt_load
extern _idtp
_idt_load:
    lidt [_idtp]
    ret
```

Add this to 'start.asm'

Setting up each IDT entry is alot easier than building a GDT entry. We have an 'idt_set_gate' function which accepts the IDT entry number, the base address of our Interrupt Service Routine, our Kernel Code Segment, and the access flags as outlined in the table introduced above. Again, we have an 'idt_install' function which sets up our special IDT pointer as well as clears out the IDT to a default known state of cleared. Finally, we would load the IDT by calling 'idt_load'. Please note that you can add ISRs to your IDT at any time after the IDT is loaded. More about ISRs later.

```
/* Use this function to set an entry in the IDT. Alot simpler
*  than twiddling with the GDT ;) */
void idt_set_gate(unsigned char num, unsigned long base, unsigned short sel, unsigned char flags)
{
    /* We'll leave you to try and code this function: take the
    *  argument 'base' and split it up into a high and low 16-bits,
    *  storing them in idt[num].base_hi and base_lo. The rest of the
    *  fields that you must set in idt[num] are fairly self-
    *  explanatory when it comes to setup */
}

/* Installs the IDT */
void idt_install()
{
    /* Sets the special IDT pointer up, just like in 'gdt.c' */
    idtp.limit = (sizeof (struct idt_entry) * 256) - 1;
    idtp.base = &idt;

    /* Clear out the entire IDT, initializing it to zeros */
    memset(&idt, 0, sizeof(struct idt_entry) * 256);
```

```
    /* Add any new ISRs to the IDT here using idt_set_gate */

    /* Points the processor's internal register to the new IDT */
    idt_load();
}
```

The rest of 'idt.c'. Try to figure out 'idt_set_gate'. It's easy!

Finally, be sure to add 'idt_set_gate' and 'idt_install' as function prototypes in 'system.h'. Remember that we need to call these functions from other files, like 'main.c'. Call 'idt_install' from inside our 'main()' function, right after the call to 'gdt_install'. You should be able to compile your kernel without problems. Take some time to experiment a bit with your new kernel. If you try to do an illegal operation like dividing by zero, you will find that your machine will reset! We can catch these 'exceptions' by installing Interrupt Service Routines in our new IDT.

If you got stuck writing 'idt_set_gate', you may find the solution to this section of the tutorial here.