Nano-Kernel : A Bare Metal OS

## Part 4 - The Global Descriptor Table

Before we can go much further, we need to deal with the tricky aspects of Intel’s memory configuration layout. Fair warning, we will keep coming back to memory, over, and over again. It should not be a surprise. As a bare-metal machine, memory is just an array of bytes, but the OS and CPU architecture work together to create a rich and powerful view of memory that can be allocated to processes, and keep the memory protected, and even create virtual memory that doesn’t really exist in the hardware.

Part of the multiboot standard is that the processor is left in “protected mode” before the bootloader hands-off control to the kernel. Protected mode operation is a little different than what we experienced in the microcontrollers course. Protected mode adds restrictions on the operation of the processor. In particular, we lose the direct access to some hardware resources (such as the BIOS and its low-level interrupt services) that we had in real mode. Losing the BIOS means we will be taking more responsibility for the resources in our computer. For example, we cannot call on the BIOS to update the position of the cursor in the video display, we must now access the video hardware ourselves to do this.

### 

### Global Descriptor Table

The global descriptor table (GDT) holds the memory protection configuration for the protected mode for the processor. The GDT is a table, which must be allocated from contiguous memory.

Each entry in the GDT is a 64-bit entry. There can be at most 1024 GDT table entries. The fields of each GDT entry are as follows:

|  |  |
| --- | --- |
| 15:0 | Limit<15:0> |
| 39:16 | Base<23:0> |
| 47:40 | Access Byte |
| 51:48 | Limit<19:16> |
| 55:52 | Flags |
| 63:56 | Base<31:24> |

**Little-Endian - Byte Mapping of GDT**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Byte: 0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| limit<7:0> | limit <15:8> | base<7:0> | base<15:8> | base<23:16> | Access | Flags <7:4>| Limit<19:16> | base<31:24> |

Fields:

* **Base<31:0>** 32-bit value containing the ***linear address*** of the start of the segment of memory. Linear means w/out segmented addressing (like MIPS), so 0 … 232-1 bytes
* **Limit<19:0>** 20-bit value containing the ***number of granules*** (see Flags.GR field) of the segment

**Access Byte - An 8-bit field describing the permitted access of the segment:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 7 | 6:5 | 4 | 3 | 2 | 1 | 0 |
| Pr | Privl | 1 | Ex | DC | RW | Ac |

* **Present bit** - 1 indicates page in memory, 0 not present in memory
* **Privl** - 2 bits, Ring-Level:  
  0 - Kernel (highest)  
  3 - Lowest (user)
* **Ex** - Executable bit - 1 if code contains executable instructions, 0 if data
* **DC** - Direction / Conforming Bit  
  When EX = 1 (Executable) - then Conforming
  + 1 - Code can be executed from an equal or lower privilege   
    Code in ring 3 can jump into code in ring 2 - then the privilege represents the highest privilege level that is allowed to execute this code
  + 0 - Code can only be executed from the ring set in Privl

When EX = 0 (Data) - then Direction

* 1 - Segment grows down (offset greater than limit)
* 0 - Segment grows up (offset less than limit
* **RW** - Readable / Writable bit   
  When EX = 1 (Executable) - Code segments are never writable
  + 1 - Code can be read and executed
  + 0 - Code can only be executed (not read)

When EX = 0 (Data) - Data segments are never executable

* 1 - Read and Write access is allowed
* 0 - Only read access is allowed
* **AC** - Accessed bit - set to 0, changes to 1 when the segment is accessed

Typical access bits and the corresponding byte are shown below:

Table 1 – Typical GDT Privilege Settings

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bits** | **7** | **6:5** | **4** | **3** | **2** | **1** | **0** |  |
| **Type** | **Pr** | **Privl** | **1** | **Ex** | **DC** | **RW** | **Ac** | **Byte** |
| Kernel Code | 1 | 00 | 1 | 1 | 0 | 1 | 0 | 1001\_1010=9A |
| Kernel Data | 1 | 00 | 1 | 0 | 0 | 1 | 0 | 1001\_0010=92 |
| User Code | 1 | 11 | 1 | 1 | 0 | 1 | 0 | 1111\_1010=FA |
| User Data | 1 | 11 | 1 | 0 | 0 | 1 | 0 | 1110\_0010=E2 |

**Flag Field**

|  |  |  |  |
| --- | --- | --- | --- |
| 7 | 6 | 5 | 4 |
| Gr | Sz | 0 | 0 |

* **Gr - Granularity bit** 
  + 1 - Limit is number of 4KB pages
  + 0 - Limit is number of bytes
* **Sz - Size bit**
  + 0 - Selects 16-bit protected mode
  + 1 - Selects 32-bit protected mode

Since the limit field is only 20-bits long but the address space is 32-bits long (in 32-bit mode), using the byte granularity would only allow a program to access 220 bytes = 1MB. Most programs select Gr=1, allowing:



Since we are only interested in 32-bit devices, the flag field, for us, will always be: 0b1100\_0000 = 0xC0.

### Mapping a GDT Entry to Bytes

Mapping a GDT entry to memory image requires splitting the **base** and **limit** fields across multiple byte ranges. A simple union employing bit-fields is shown in the following code:

|  |
| --- |
| typedef struct {  unsigned int limit:20;  unsigned int base:32;  unsigned int access:8;  unsigned int flags:4;  } gdt\_entry\_t; |

This must be mapped to memory such that the CPU will interpret the protection fields correctly. Consider a mapping of the GDT entry:

|  |
| --- |
| gdt\_entry\_t entry;  entry.base = 0xbeeffeed;  entry.limit = 0x12345;  entry.access = 0b10011010;  entry.flags = 0b1100; |

This should map out to the following eight bytes:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Byte: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 45 | 23 | ED | FE | EF | 9A | C1 | BE |

**Pro-Tip:** When you are debugging your code, you can use GDB to inspect the GDT table and verify that the bytes are in the proper order. I got stuck here for a little while because I had the access and flags fields flipped in the ordering and the machine just kept rebooting.

### C-Preprocessor isn’t really just for C!

The #define statements used in the C-preprocessor also work in .S files, so this can be put into a “.h” file and the “#include”’d from a “.S” file or else it can be put into a “.S” file.: Either way, the macro expands like it would in C.

The start of my “gdt.h” file is shown in the following table. This is a fairly typical example of a low-level definition for hardware features. There are a series of #define’s that represent each of the flag values - giving them a symbolic name.

#define ACC\_PRESENT ((1 << 7) | (1 << 4))

Each bit is shifted into position and then Boolean OR’d together. I personally prefer this syntax to the alternative:

#define ACC\_PRESENT 0x90

The shifts-and-ORs will be evaluated at compile time, so the first version doesn’t change the run-time of the program, and it adds clarity to what bits each flag contributes to.

The other typical technique is to include the “zero” shifts, such as ACC\_DATA\_RDONLY, which shifts 0 over 1 bit - which is still zero. This is also for clarity - it makes it easier to detect that this is the complement of ACC\_DATA\_WR.

There are also several macros that combine the fields into one symbolic name, such as ACC\_KERN\_CODE.

You will need to fill in values for: ACC\_USER\_CODE and ACC\_USER\_DATA following the pattern of ACC\_KERN\_CODE and ACC\_KERN\_DATA (but changing the security ring), and you will need to fill in the code for the GDTENTRY macro to shift the proper bits into the proper byte.

|  |
| --- |
| #ifndef \_GDT\_H  #define \_GDT\_H  #define ACC\_PRESENT ((1 << 7) | (1 << 4))  #define ACC\_PRIV\_KERN (0 << 5)  #define ACC\_PRIV\_USER (3 << 5)  #define ACC\_EXEC\_CONFORM (1 << 3)  #define ACC\_EXEC\_NON\_CONFORM ((1 << 3 ) | (1 << 2))  #define ACC\_DATA (0 << 3)  #define ACC\_DATA\_DN ((0 << 3) | (1 << 2))  #define ACC\_DATA\_WR (1 << 1)  #define ACC\_DATA\_RDONLY (0 << 1)  #define ACC\_CODE\_RD (1 << 1)  #define ACC\_CODE\_EXONLY (0 << 1)  #define ACC\_KERN\_CODE (ACC\_PRESENT | ACC\_PRIV\_KERN | \  ACC\_EXEC\_CONFORM | ACC\_CODE\_RD)  #define ACC\_KERN\_DATA (ACC\_PRESENT | ACC\_PRIV\_KERN |\  ACC\_DATA | ACC\_DATA\_WR)  #define ACC\_USER\_CODE 0  #define ACC\_USER\_DATA 0  #define FLAG\_DEFAULT (FLAG\_GRAN\_4K | FLAG\_SIZE\_32)  // you will need to map the fields to bytes using shifts and masks  #define GDTENTRY(base,limit,access) \  .byte 0, \  1, \  2, \  3, \  4, \  5, \  6, \  7 #endif |

### Null Descriptor

The first entry in the GDT (position 0) is ignored by the CPU and is, by convention a “null entry”. Convention is that the null entry can be used to store the address and length of the GDT itself and thus not have to store these values in parallel. The value is optional but there must be an initial entry that is not used. This is required by the Intel cpu.

Using the macro defined above, the Null descriptor would be:

GDTENTRY(0,0,0)

### Minimum GDT Entries[[1]](#footnote-1)

* NULL Descriptor
* Code segment (for kernel) starting at 0 with a limit of 0xff\_ffff of type 0x9A
* Data segment (for kernel) starting at 0 with a limit of 0xff\_ffff of type 0x92

### Loading the GDT after Multiboot

Multiboot leaves the CPU in protected mode but with flat addressing (no paging). We need to use the “LGDT” instruction to load the address of the GDT description field into the GDT register (GDTR).

|  |
| --- |
| gdtrdesc:  **DW** num\_gdt\_bytes-1 *; For limit storage*  **DD** gdt\_address *; For base storage*   setGdt:  ...  **LGDT** gdtrdesc  ... |

### Creating a GDT

We need to create a GDT that contains at least the sections described in the previously described minimums section:

### Bare Minimum GDT for a Kernel

|  |  |  |
| --- | --- | --- |
| GDT 0 | Base 0, Limit 0, Type 0 | GDTENTRY(0,0,0) |
| GDT 1 | Base 0, Limit 0xffff\_ffff, Type: 9A (Selector 0x08 is code) | GDTENTRY(0, 0xffffffff, ACC\_PRESENT | ACC\_PRIV\_KERN | ACC\_EXEC\_CONFORM | ACC\_CODE\_RD) |
| GDT 2 | Base 0, Limit 0xffff\_ffff, Type 92 (Selector 0x10 is data) | GDTENTRY(0, 0xffffffff, ACC\_PRESENT | ACC\_PRIV\_KERN | ACC\_DATA | ACC\_DATA\_WR) |

This simply enables all of the 4GB of memory to be accessed as code or data, its not really protected mode at all!

### Entries for a Small and Protected Kernel

|  |  |  |
| --- | --- | --- |
| GDT 0 | Base 0, Limit 0, Type 0 | GDTENTRY(0,0,0) |
| GDT 1 | Base 0x0400\_0000, Limit 0x03ff\_ffff, Type: 9A (Selector 0x08 is code) | GDTENTRY(0x04000000, 0x3ffffff, ACC\_PRESENT | ACC\_PRIV\_KERNEL | ACC\_EXEC\_CONFORM | ACC\_CODE\_RD) |
| GDT 2 | Base 0x0800\_0000, Limit 0x03ff\_ffff, Type 92 (Selector 0x10 is data) | GDTENTRY(0x08000000, 0x3ffffff, ACC\_PRESENT | ACC\_PRIV\_KERNEL | ACC\_DATA | ACC\_DATA\_WR) |
| GDT 3 | Base = task address, Limit = size of task, type=0x89 (ring 3) | GDTENTRY(address, nr\_pages, ACC\_PRESENT | ACC\_EXEC\_CONFORM | ACC\_CODE\_EX\_ONLY) |

In this example, code loaded at physical address 4MB will be CS:0000 (and extending for 4MB), and physical address 8MB will be DS:0000 (and extending another 4MB).

At the end of “boot.S” you can add:

|  |
| --- |
| # Bootstrap GDT  .p2align 2 # force 4 byte alignment  gdt:  GDTENTRY( 0, 0, 0, 0) # null seg  GDTENTRY(0, 0xffffffff, ACC\_KERN\_CODE, FLAG\_DEFAULT) # kernel code  GDTENTRY(0, 0xffffffff, ACC\_KERN\_DATA, FLAG\_DEFAULT) # kernel data  GDTENTRY(0, 0xffffffff, ACC\_USER\_CODE, FLAG\_DEFAULT) # user code  GDTENTRY(0, 0xffffffff, ACC\_USER\_DATA, FLAG\_DEFAULT) # user data    gdtdesc:  .word (gdtdesc - gdt - 1) # sizeof(gdt) - 1  .long gdt # address gdt |

Note the use of the arithmetic involving Assembler symbols in determining the length of the GDT. The GNU assembler will compute the difference between the “gdt” and “gdtdesc” labels to get the length (in bytes) of the table during assembling. Another version of this line is fairly common:

.word ($ - gdt - 1)

Where the “naked $” is used to represent the current location in he Assembler (but not the Linker)

### Reloading the Segment Registers[[2]](#footnote-2)

The 16-bit segment registers must be reloaded after the GDT is loaded. The segment register must be loaded with the *offset* to the GDT entry that the segment refers to. If the first entry is the NULL entry (it must be), and the second entry is the executable code, and the third entry is the data, then the CS register should be loaded with 0x0008 - the offset of the code entry in bytes. All of the other data registers should be loaded with 0x0010 (1610), the offset of the data entry in the GDT. The following function will load the segment registers. Note the odd use of the “long jump” instruction LJMP. The normal “JMP” instruction can only jump a relative distance to the CS register (much like the MIPS J instruction). The “LJMP” instruction can load both the CS and the IP register. In this case, we “long jump” to the very next instruction which has the useful side-effect of loading the CS register with the desired 0x0008 (code entry in GDT).

|  |
| --- |
| reloadSegments:  *; Reload CS register containing code selector:*  **LJMP** $0x08,reload\_CS *; 0x08 points at the new code selector* .reload\_CS:  *; Reload data segment registers:*  **MOV** 0x10, %AX *; 0x10 points at the new data selector*  **MOV** %AX, %DS  **MOV** %AX, %ES  **MOV** %AX, %FS  **MOV** %AX, %GS  **MOV** %AX, %SS  **RET** |

The rest of the instructions simply move 0x0010 (the data entry in GDT) into each of the remaining segment registers.

The complete boot.S is shown on page 11.

At this point, we have developed a boot program that:

1. Provides the necessary symbols for multiboot compatibility
2. Handles the transition from multiboot at the “\_start” symbol
3. Enables protected mode if it wasn’t already
4. Loads the GDT data that is assembled into the program
5. Reloads the segment registers to the proper protected mode values
6. Creates a stack for the kernel
7. Calls “kmain” (a C program)
8. Handles the possibility of “kmain” terminating by entering a “halt spin-loop”

# Deliverables and Demos

Arrange a time for us to meet, and show be prepared to show me the following:

1. Demonstrate that your system works
2. Show me how you implemented the boot-up and the GDT initialization
3. Show me how you implemented the “halt-spin-loop”
4. Again, I want to see code quality

Also, be prepared to talk about:

1. What is a ‘segment descriptor”
2. What is a ‘segment selector’
3. What is the difference? There is actually more to the “0x10” for GDT selector, do a little research and find out what is actually going on there, and what fields we are actually skipping over.

Points: \_\_\_\_\_\_\_\_\_ / 40

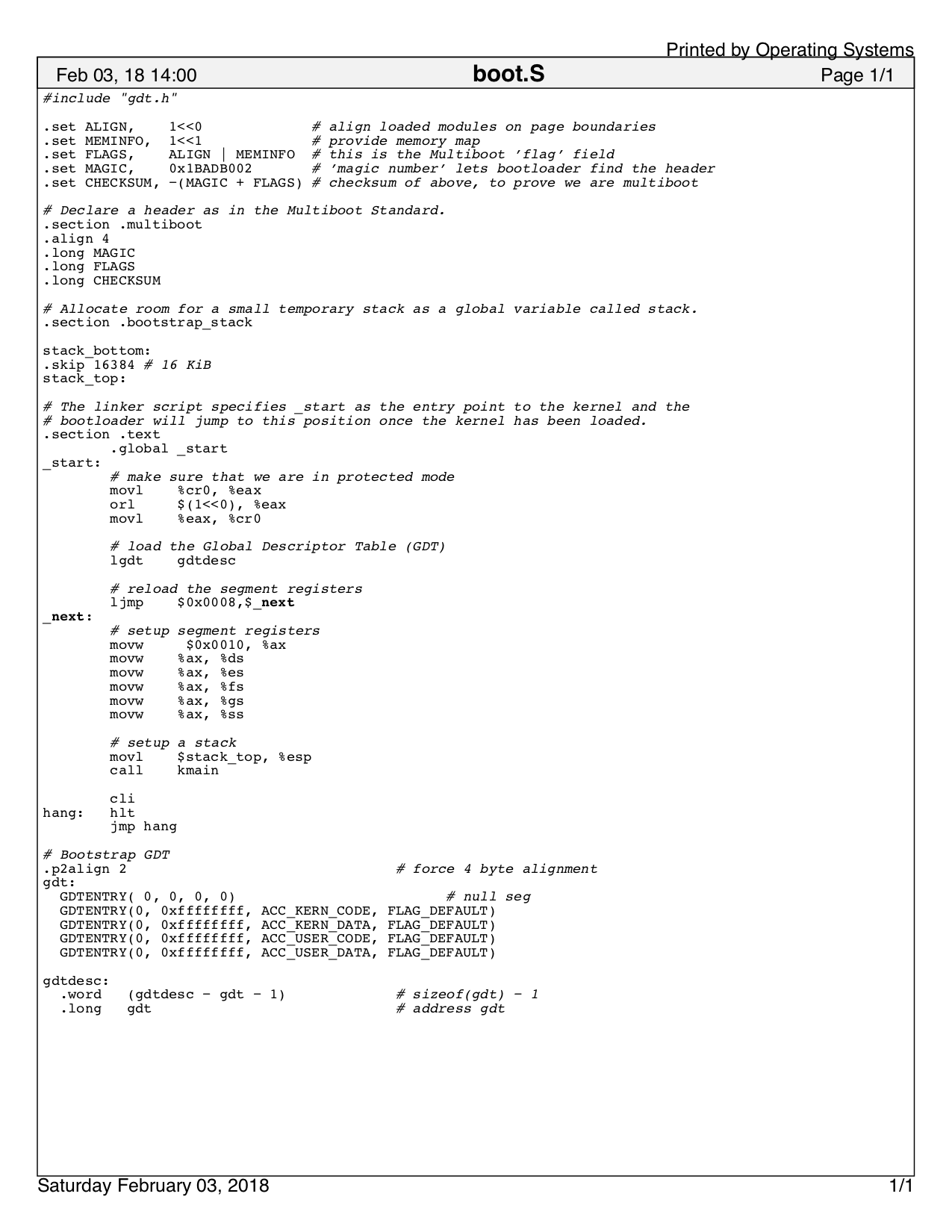


Figure 1 - Complete "boot.S"

1. https://wiki.osdev.org/GDT\_Tutorial [↑](#footnote-ref-1)
2. https://wiki.osdev.org/GDT\_Tutorial.. [↑](#footnote-ref-2)