**Writing multiboot PE kernels using Visual C++**

Oct 1, 2005  
tags: programming kernel multiboot

[Tweet](https://twitter.com/share)

Update [1/29/2012]: Source code now available [here](http://github.com/kaushiks/multibootpe).

Aspiring operating system developers who target x86 often don't get beyond writing a boot sector. Seldom do they even complete it due to the inordinate amount of time one has to spend, to understand the "tricks" required to get the processor into a "sane" (protected) mode of operation before the kernel may start executing. This is why new kernel writers are often advised to use an alternative like [Grub](http://www.gnu.org/software/grub/) to bootstrap their kernel and focus on implementing the kernel itself, rather than the plumbing. Why Grub? Because it is one of the boot loaders that implements the Multiboot specification.

The Multiboot specification details the steps that kernel and bootloader developers need to follow in order to be compatible with (and usable by) each other. In terms of code, Multiboot compliant operating systems contain a 48 byte structure called the Multiboot header (in its entirety), somewhere within the first 8192 bytes of the kernel image, longword aligned. . [Updated: 10/6/2005, 12:23 PM]. Actual details about the fields are documented in the Multiboot specification [here](http://www.gnu.org/software/grub/manual/multiboot/multiboot.html).

Most "roll your own OS" tutorials explain the steps required to make a kernel bootable by Grub. But all of them assume you are using GCC. If you are using the Visual C++ tool chain you are out of luck, until now. This post will explain how you can make GRUB boot your PE kernel image produced by the Visual C++ compiler.

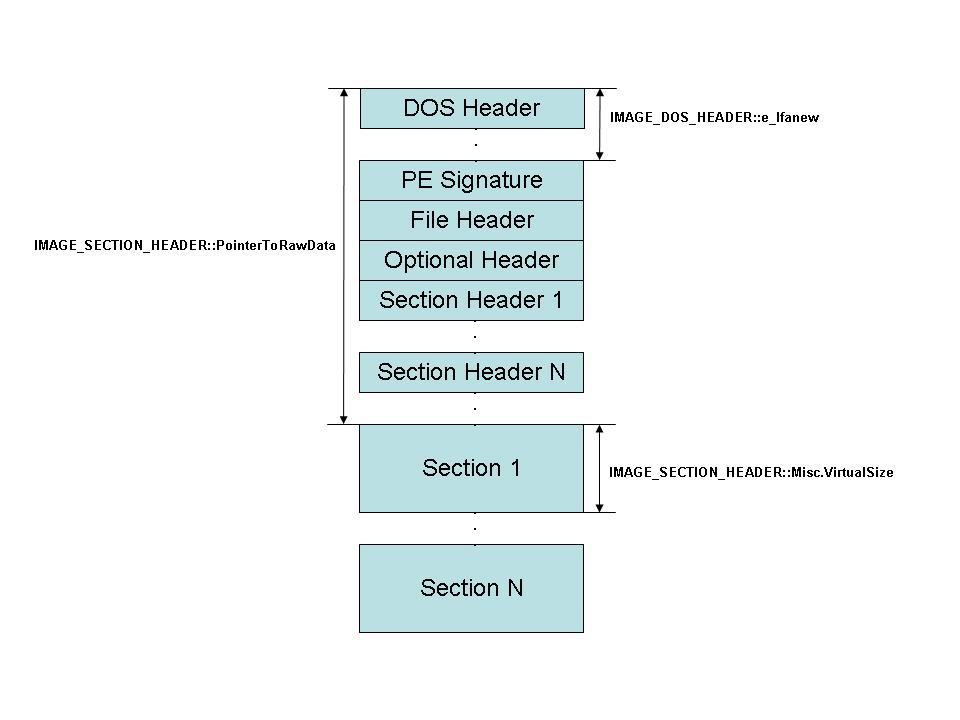
Making a boot loader like Grub boot a custom kernel is easy (at least compared to the effort it takes to create a new boot loader). The kernel itself is only a binary image in some file format - Typically AOUT, ELF, PE etc. The Windows kernel (%WINDRIVE%\Windows\System32\NTOSKRNL.EXE), for instance, uses the PE file format (Try dumpbin /ALL %WINDRIVE%\Windows\System32\NTOSKRNL.EXE) that is also used by user mode programs under windows. Similarly, the Linux kernel uses the ELF file format. Expecting a boot loader to "know" all executable file formats is probably not a good idea. The multiboot specification takes a different approach to load a kernel image onto RAM. It uses fields in the multiboot header to denote the parts of the kernel image that needs to be loaded. Grub "knows" how to load an ELF binary, not a PE. So we are going to give it "hints" in our multiboot header that will help it load the kernel image properly. Time for some code ...

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | /\* kernel.h \*/  #ifndef \_\_kernel\_h\_\_  #define \_\_kernel\_h\_\_  #define dd(x) \  \_\_asm \_emit (x) & 0xff \  \_\_asm \_emit (x) >> 8 & 0xff \  \_\_asm \_emit (x) >> 16 & 0xff \  \_\_asm \_emit (x) >> 24 & 0xff  #define KERNEL\_STACK 0x00103fff  #define KERNEL\_START 0x00101000  #define KERNEL\_LENGTH 0x0000200F  void main(unsigned long, unsigned long);  #endif |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | /\* kernel.c \*/  #include "kernel.h"  \_\_declspec(naked) void \_\_multiboot\_entry\_\_(void)  {  \_\_asm {  multiboot\_header:  dd(0x1BADB002) ; magic  dd(1 << 16) ; flags  dd(-(0x1BADB002 + (1 << 16))); checksum  dd(0x00101000) ; header\_addr  dd(0x00101000) ; load\_addr  dd(0x0010200F) ; load\_end\_addr  dd(0x0010200F) ; bss\_end\_addr  dd(0x00101030) ; entry\_addr  dd(0x00000000) ; mode\_type  dd(0x00000000) ; width  dd(0x00000000) ; height  dd(0x00000000) ; depth  kernel\_entry:  mov esp, KERNEL\_STACK  xor ecx, ecx  push ecx  popf  push eax  push ebx  call main  jmp $  }  }  void main(unsigned long magic, unsigned long addr)  {  char \*string = "Hello World!", \*ch;  unsigned short \*vidmem = (unsigned short \*) 0xB8000;  int i;  for(ch = string, i = 0; \*ch; ch++, i++)  vidmem[i] = (unsigned char) \*ch | 0x0700;  } |

The first field in the header is a magic number that the boot loader will use to locate the start of the multiboot header in the image. The second field denotes the features that the OS expects from the boot loader. To keep the code simple, I've ignored bits 0-15 (about which you can read in the multiboot specification). I've set bit 16 of this field. This means that the fields at offsets 8-24 in the Multiboot header are valid, and the boot loader should use them instead of the fields in the actual executable header to calculate where to load the OS image. This mechanism enables the boot loader load kernel images whose format it doesn't understand natively (like PE). Before examining what the fields at offsets 8-24 mean, let's take a look at the PE file format.

**The PE file format**



A PE image starts with a couple of standard headers (DOS / PE / File / Optional). Following these is a set of headers called the section headers that contain information about the different sections in the image. (For a more verbose explanation of the PE file format read Matt Pietrek's excellent [article](https://msdn.microsoft.com/en-us/library/ms809762.aspx).) A section typically contains either code or data. The above kernel if compiled with the following switches

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | cl /Gd  /Fokernel.obj  /Fm  /TC  /c  kernel.c  link /safeseh:no  /filealign:0x1000  /BASE:0x100000  /MAP: kernel.map  /ENTRY:\_\_multiboot\_entry\_\_ kernel.obj  /NODEFAULTLIB:LIBC  /SUBSYSTEM:CONSOLE  /OUT: kernel.exe |

Produces an .EXE with two sections named .text and .data. Sections are aligned on a 4K boundary using the undocumented linker switch /filealign:0x1000.

Armed with this information about the PE file format, lets examine the fields at offset 8-24 in the multiboot header.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | dd(0x1BADB002) ; magic  dd(1 << 16) ; flags  dd(-(0x1BADB002 + (1 << 16))); checksum  dd(0x00101000) ; header\_addr  dd(0x00101000) ; load\_addr  dd(0x0010200F) ; load\_end\_addr  dd(0x0010200F) ; bss\_end\_addr  dd(0x00101030) ; entry\_addr  dd(0x00000000) ; mode\_type  dd(0x00000000) ; width  dd(0x00000000) ; height  dd(0x00000000) ; depth |

The field at offset 8, Checksum, needs to be set to

-(magic + flags)

Grub loads the .text section of the kernel into physical address 0x100000 (1 MB) + Offset by default. The offset is specified indirectly using the header\_addr and load\_addr fields. According to the specification header\_addr "Contains the address corresponding to the beginning of the Multiboot header". This is a bit confusing. What it really means is, if the image file is loaded at 0x100000, the physical address of the starting of the multiboot header is header\_addr. The next field load\_addr contains the physical address of the beginning of the .text section. (In our case both are the same because the multiboot header is the first 48 bytes of the .text section). The next field load\_end\_addr is used to determine how many bytes of the image file actually needs to be loaded. (Note that the .text and .data sections need to be successive in the image for this to work). In our case 0x102000 is where data section starts and it has a size of 0xF bytes and hence the value 0x10200F for load\_end\_addr. Grub, now knows it needs to load 0x10200F - 0x101000 bytes. The next field according to the multiboot specification, needs to be set to 0 if a bss section doesn't exist. (As in our case). However Grub refuses to load the image if bss\_end\_addr is set to 0, so I set it to 0x10200F (same as the previous field).

Merely initializing this data structure correctly gives Grub all the information it needs to load our PE kernel correctly.

Continued in [Part II](http://srevas.net/notes/2005/10/02/multiboot2/).

## Writing a multiboot PE kernel: part 2

Oct 2, 2005  
tags: programming kernel multiboot

[Tweet](https://twitter.com/share)

Update [1/29/2012]: Source code now available [here](http://github.com/kaushiks/multibootpe).

Part I is [here](http://srevas.net/notes/2005/10/01/multiboot1/).

While my earlier post explained the "how?", this will explain the "why?".

* The linker (link.exe) puts machine code of functions in the source text into the .text section in the order it finds them in the source.
* All PE headers never add up to a total size that exceeds 4K. So when file alignment is 4K (== memory alignment), the .text section is guaranteed to start at offset 4K (4096) in the PE file. This is a good place to put the multiboot header (which anyways needs to be present in its entirety within the first 8192 bytes.)
* The multiboot header forms the first 48 bytes of the .text section.
* declspec(naked) is an attribute that makes the compiler generate code without a prolog or an epilog. This is important because we want the multiboot header to start at offset 4096. Without the naked attribute, the function (and hence the .text section) would start with the bytes 55 8B EC which stand for the following instructions:

|  |  |
| --- | --- |
| 1  2 | push ebp ;; 55  mov ebp, esp ;; 8B EC |

which is the prolog. Because of this the multiboot header would be pushed to offset 4099 and Grub would refuse to load the kernel because the multiboot header isn't 4K longword aligned. [Updated: 10/6/2005, 12:23 PM].

Compiler switches:

* /Gd: forces the use of cdecl calling convention.
* /Fm: names the map file which might be useful when the kernel gets large.
* /Tc: compile the file as C source.
* /c: compile only, no link.

Linker switches:

* /safeseh:no: disables generating the symbols related to Safe SEH handlers (\_\_safe\_se\_handler\_table and \_\_safe\_se\_handler\_count.)
* /filealign:0x1000: this is an undocumented switch that aligns sections in the file based on this value. I've set it to 0x1000 (4K) so that sections are aligned on a 4K boundary on the image also (this is the default in-memory alignment). This is required because, Grub doesn't seem to load images whose file alignment is different from in-memory alignment.
* /base:0x100000: this makes the linker generate code assuming that the .text section starts at physical address 0x101000 and .data section at physical address 0x102000. This is what we want because Grub actually loads the image at 0x100000 which forces the .text and .data section into these addresses automatically and we need not relocate the kernel. (Now you probably understand why we set the file alignment to 4096 bytes as well).
* /entry:\_\_multiboot\_entry\_\_: sets the entry point.
* /nodefaultlib:libc: forces the linker to ignore libc while resolving external references. The idea is to be able to use names like memcpy etc in the kernel, and make sure the libc's functions don't get linked in.
* /subsystem:console: This sets a bit in one of the headers that tells windows which subsystem to use to execute this application. This doesn't make any sense here, but I included this to keep the linker happy.
* /out: kernel.exe: the name of the kernel.