Xosdev Chapter 2 [Basic Kernel] - by mr. xsism

- - - - - - - - - - - -
[Planning/Setting goals]
- - - - - - - - - - - -

Congrats, you have gotten farther than many people that have tried OSdev. So what now? Now you start to see the results of your hard work; you load the kernel. "Wow," I hear you say, "that can't be too hard :)." Well, it is and it isn't. If you keep it simple and use C it'll be a piece of pie. If you use C++, it just might be hell. You have to load class, new, and delete support to name a few, where as in C you don't because everything can be made into asm without having to add support for it. I know of a few people that use C++ and they like it. It is up to you to research C++ kernel dev. From here it is all ahead full :)

First we need some code that will be loaded by our bootloader. Something like this:

In C:
```
void k_main()
{
  int num;
  char ch;
  char *str="Kernel Loaded";
  return;
}
```

All that does is declare 3 variables and returns. That's just fine and dandy. Here are some other things that we can do:

1-output "Hello, World"
2-clear the screen
3-output colored text to the screen

- - - - - - - - - - - - - - - -
[First goal: Displaying text]
- - - - - - - - - - - - - - - -

No screen output, what's the fun in that?? Nothing >:/ What we need is Screen I/O. It has to be the easiest thing to code in the kernel. Just involves righting a simple ascii character to screen's memory and the character's color.

The way that you do this is simply by placing an ascii byte for the character followed by its attribute at 0xB8000. You don't always put it there, because each time you print one character you should increment the text pointer by two(char byte + attrib byte).

In C:
```
void _k_main()
{
  int num;
  char ch;

  char *text_video = (char*)0xB8000;
  char attrib = 0x07;
  char *str="Kernel Loaded";
```

```
  while(*str!=0)
  {
    *text_video = *str;
    *text_video++;
    *text_video = attrib;
    *text_video++;
    *str++;
  }
  return;
}
```

- - - - - - - - - -
[Clearing screen]
- - - - - - - - - -

Clearing the screen is another very easy task involving the text pointer. For each ascii character in text video, just set it to zero and the attribute byte to the current attribute.

In C:

```
void clear_screen(char clear_to, char attrib)
{
  char *text_video = (char*)0xB8000;
  int pos=0;

  while(pos<(80*25*2))
  {
    *text_video = clear_to;
    *text_video++;
    *text_video = attrib;
    *str++;
    pos++;
  }
}
```

You could just set the char to null and the attribute to 0x07 (white on black), but I did the most complicated for you. Now, about attributes. What are they!?

- - - - - - - - - - - - - - - -
[Text colors 'n' attributes]
- - - - - - - - - - - - - - - -

Let's get started by giving all the supported text colors & attributes:

-{TEXT COLORS}-

FG AND BG
0 = black
1 = blue
2 = green
3 = cyan
4 = red
5 = magenta
6 = brown
7 = white (standard text color)
FG ONLY
8 = dark grey
9 = bright blue

```
  10 = bright green
  11 = bright cyan
  12 = pink
  13 = bright magenta
  14 = yellow
  15 = bright white
  [IBBBFFFF] binary
  I = Intensity (blink)
  B = Background
  F = Foreground
```

Ok, so that's really nice, a bunch of friggin' numbers! What are we supposed to do with that!? Let's find out through example. 0x07 is White on Black. That means that the text is white and the background color is black. Well how about that! Those numbers match up with the right colors on our list. Okay, well, say we want red text and white background. What would that number be in hex? Well, red = 4 and white = 7. So then it would be 0x74.

There are a couple strange things that you migth be wondering about. For instance, why can colors 8-15 be only in the foreground? Well, if you were to have read the whole list you would have seen one very inportant point! Text can be blinking. How does it blink? It all depends on whether one bit is set. That one bit happens to be in the background WORD. See, look:

```
blink  bg  fg
  \  _/  _/_
  IBBBFFFF
  I = Intensity (blink)
  B = Background
  F = Foreground
```

111 in bin is only 7. That means that to make room for the blinker bit, you can only use the first 7 colors in our list. Is that so hard to understand? I hope not. If it is, don't think about it too long, you might hurt yourself.

```
- - - - - - - - - - - -
[Compiling the Kernel]
- - - - - - - - - - - -
```

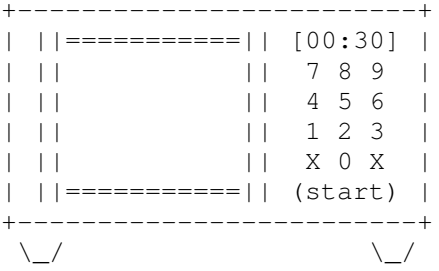
Ahhh, yes. Compiling the bootloader was easy, right? You did read the first chapter, right!? Well, all that this involves is compiling the C source code, linking it all together, and then copying the bootloader to the very from of the kernel object. This will give a kernel image that you can write too a bootable device, like a floppy disk. Let me make this even easier for you:

```
-{Step by Step}-
1-compile all *.c files
>gcc *.c

2-compile all asm files into a format like aout (not bin, C doesn't
  output to bin by default)
>nasm *.asm -f aout

3-link all C files and asm files together into a file(ie:kernel.o)
>ld -T linkscript.ld -o kernel.o a.o b.o c.o
```

4-compile & copy the bootloader to the front of the kernel object
  file(ie:kernel.img)
>nasm boot.asm
>copy /b boot.bin+kernel.o kernel.img

5-write the image file to a bootable device(ie:floppy disk)
>floppyout kernel.img a: -sector 0 -head 0 -track 0

6-now take that bootable device and put it in a microwave oven for
  30 seconds, wait for it to melt, and enjoy!



Ok, are you awake? I hope you dismissed step 6(if not please notify me so i can bash you over your head). I hope this was adequate enough for you. If you need more help, look at the given example code for this chapter.