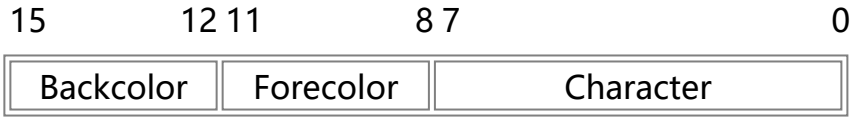
Printing to the Screen

Now, we will try to print to the screen. In order to print to the screen, we need a way to manage scrolling the screen as needed, also. It might be nice to allow for different colors on the screen as well. Fortunately, a VGA video card makes it rather simple: It gives us a chunk of memory that we write both attribute byte and character byte pairs in order to show information on the screen. The VGA controller will take care of automatically drawing the updated changes on the screen. Scrolling is managed by our kernel software. This is technically our first driver, that we will write right now.

As mentioned, above, the text memory is simply a chunk of memory in our address space. This buffer is located at 0xB8000, in physical memory. The buffer is of the datatype 'short', meaning that each item in this text memory array takes up 16-bits, rather than the usual 8-bits that you might expect. Each 16-bit element in the text memory buffer can be broken into an 'upper' 8-bits and a 'lower' 8-bits. The lower 8 bits of each element tells the display controller what character to draw on the screen. The upper 8-bits is used to define the foreground and background colors of which to draw the character.



The upper 8-bits of each 16-bit text element is called an 'attribute byte', and the lower 8-bits is called the 'character byte'. As you can see from the above table, mapping out the parts of each 16-bit text element, the attribute byte gets broken up further into 2 different 4-bit chunks: 1 representing background color and 1 representing foreground color. Now, because of the fact that only 4-bits define each color, there can only be a maximum of 16 different colors to choose from (Using the equation (num bits ^ 2) - 4^2 = 16). Below is a table of the default 16-color palette.

Value	Color	Value	Color
0	BLACK	8	DARK GREY
1	BLUE	9	LIGHT BLUE
2	GREEN	10	LIGHT GREEN
3	CYAN	11	LIGHT CYAN
4	RED	12	LIGHT RED
5	MAGENTA	13	LIGHT MAGENTA
6	BROWN	14	LIGHT BROWN
7	LIGHT GREY	15	WHITE

Finally, to access a particular index in memory, there is an equation that we must use. The text mode memory is a simple 'linear' (or flat) area of memory, but the video controller makes it appear to be an 80x25 matrix of 16-bit values. Each line of text is sequential in memory; they follow eachother. We therefore try to break up the screen into horizontal lines. The best way to do this is to use the following equation:

index = (y_value * width_of_screen) + x_value;

This equation shows that to access the index in the text memory array for say (3, 4), we would use the equation to find that 4 * 80 + 3 is 323. This means that to draw to location (3, 4) on the screen, we need to write to do something similar to this:

unsigned short *where = (unsigned short *)0xB8000 + 323;
*where = character | (attribute << 8);

Following now is 'scrn.c', which is where all of our functions dealing with the screen will be. We include our 'system.h' file so that we can use outportb, memcpy, memset, memsetw, and strlen. The scrolling method that we use is rather interesting: We take a chunk of text memory starting at line 1 (NOT line 0), and copy it over top of line 0. This basically moves the entire screen up one line. To complete the scroll, we erase the last line of text by writing spaces with our attribute bytes. The putch function is possibly the most complicated function in this file. It is also the largest, because it needs to handle any newlines ('\n'), carriage returns ('\r'), and backspaces ('\b'). Later, if you wish, you may handle the alarm character ('\a' - ASCII character 7), which is only supposed to do a short beep when it is encountered. I have included a function to set the screen colors also (settextcolor) if you wish.

```
#include < system.h >

/* These define our textpointer, our background and foreground
*  colors (attributes), and x and y cursor coordinates */
unsigned short *textmemptr;
int attrib = 0x0F;
int csr_x = 0, csr_y = 0;

/* Scrolls the screen */
void scroll(void)
{
    unsigned blank, temp;

    /* A blank is defined as a space... we need to give it
    *  backcolor too */
    blank = 0x20 | (attrib << 8);

    /* Row 25 is the end, this means we need to scroll up */
    if(csr_y >= 25)
    {
        /* Move the current text chunk that makes up the screen
        *  back in the buffer by a line */
        temp = csr_y - 25 + 1;
        memcpy (textmemptr, textmemptr + temp * 80, (25 - temp) * 80 * 2);

        /* Finally, we set the chunk of memory that occupies
        *  the last line of text to our 'blank' character */
        memsetw (textmemptr + (25 - temp) * 80, blank, 80);
        csr_y = 25 - 1;
    }
}

/* Updates the hardware cursor: the little blinking line
*  on the screen under the last character pressed! */
void move_csr(void)
{
    unsigned temp;

    /* The equation for finding the index in a linear
    *  chunk of memory can be represented by:
    *  Index = [(y * width) + x] */
    temp = csr_y * 80 + csr_x;
```

```
    /* This sends a command to indicies 14 and 15 in the
    *  CRT Control Register of the VGA controller. These
    *  are the high and low bytes of the index that show
    *  where the hardware cursor is to be 'blinking'. To
    *  learn more, you should look up some VGA specific
    *  programming documents. A great start to graphics:
    *  http://www.brackeen.com/home/vga */
    outportb(0x3D4, 14);
    outportb(0x3D5, temp >> 8);
    outportb(0x3D4, 15);
    outportb(0x3D5, temp);
}

/* Clears the screen */
void cls()
{
    unsigned blank;
    int i;

    /* Again, we need the 'short' that will be used to
    *  represent a space with color */
    blank = 0x20 | (attrib << 8);

    /* Sets the entire screen to spaces in our current
    *  color */
    for(i = 0; i < 25; i++)
        memsetw (textmemptr + i * 80, blank, 80);

    /* Update out virtual cursor, and then move the
    *  hardware cursor */
    csr_x = 0;
    csr_y = 0;
    move_csr();
}

/* Puts a single character on the screen */
void putch(unsigned char c)
{
    unsigned short *where;
    unsigned att = attrib << 8;

    /* Handle a backspace, by moving the cursor back one space */
    if(c == 0x08)
    {
        if(csr_x != 0) csr_x--;
    }
    /* Handles a tab by incrementing the cursor's x, but only
    *  to a point that will make it divisible by 8 */
    else if(c == 0x09)
    {
        csr_x = (csr_x + 8) & ~(8 - 1);
    }
    /* Handles a 'Carriage Return', which simply brings the
    *  cursor back to the margin */
    else if(c == '\r')
    {
        csr_x = 0;
    }
    /* We handle our newlines the way DOS and the BIOS do: we
    *  treat it as if a 'CR' was also there, so we bring the
    *  cursor to the margin and we increment the 'y' value */
    else if(c == '\n')
    {
        csr_x = 0;
        csr_y++;
    }
    /* Any character greater than and including a space, is a
    *  printable character. The equation for finding the index
    *  in a linear chunk of memory can be represented by:
```

```
    *  Index = [(y * width) + x] */
    else if(c >= ' ')
    {
        where = textmemptr + (csr_y * 80 + csr_x);
        *where = c | att;     /* Character AND attributes: color */
        csr_x++;
    }

    /* If the cursor has reached the edge of the screen's width, we
    *  insert a new line in there */
    if(csr_x >= 80)
    {
        csr_x = 0;
        csr_y++;
    }

    /* Scroll the screen if needed, and finally move the cursor */
    scroll();
    move_csr();
}

/* Uses the above routine to output a string... */
void puts(unsigned char *text)
{
    int i;

    for (i = 0; i < strlen(text); i++)
    {
        putch(text[i]);
    }
}

/* Sets the forecolor and backcolor that we will use */
void settextcolor(unsigned char forecolor, unsigned char backcolor)
{
    /* Top 4 bytes are the background, bottom 4 bytes
    *  are the foreground color */
    attrib = (backcolor << 4) | (forecolor & 0x0F)
}

/* Sets our text-mode VGA pointer, then clears the screen for us */
void init_video(void)
{
    textmemptr = (unsigned short *)0xB8000;
    cls();
}
```

Printing to the screen: 'scrn.c'

Next, we need to compile this into our kernel. To do that, you need to edit 'build.bat' in order to add a new gcc compile command. Simply copy the command in 'build.bat' that corresponds to 'main.c' and paste it right afterwards. In our newly pasted line, change 'main' to 'scrn'. Again, don't forget to add 'scrn.o' to the list of files that LD needs to link! Before we can use these in main, you must add the function prototypes for putch, puts, cls, init_video, and settextcolor into 'system.h'. Don't forget the 'extern' keyword and the semicolons as these are each function prototypes:

```
extern void cls();
extern void putch(unsigned char c);
extern void puts(unsigned char *str);
extern void settextcolor(unsigned char forecolor, unsigned char backcolor);
extern void init_video();
```

Add these to 'system.h' so we can call these from 'main.c'

Now, it's safe to use our new screen printing functions in out main function. Open 'main.c' and add a line that calls init_video(), and finally a line that calls puts, passing it a string: puts("Hello World!"); Finally, save all your changes, double click 'build.bat' to make your kernel, debugging any syntax errors. Copy your 'kernel.bin' to your GRUB floppy disk, and if all went well, you should now have a kernel that prints 'Hello World!' on a black screen in white text!