Mixing Assembly and C-code

by Gergor Brunmar

Why mix programming languages?

After the last tutorial, you now feel like king of the world! =) You're eager to jump into the action, but there's one problem. Even though assembly is a powerful language, it takes time to read, write and understand. This is the main reason there ARE more programming languages than just assembly =).

Now that we have a working 32-bit boot sector, we want to be able to continue our development in a higher language, whenever possible. C is my main choice, because it's common and powerful. If you think C is old and want to use C++ instead, I'm not stopping you. The choice is your's to make.

Say that we want a print() function instead of addressing the video memory directly. Also, we want a clrscr() to clear the screen. This could easily be done by making a for-loop in C. We can't make function calls from a binary file (eg. our boot sector). For this purpose, we create another file, from which we will operate after the boot sector is done. So now we need to create a file, called '*main.c*'. It will contain the main() function - yes, even operating systems can't escape main() =). As I said, a boot sector can't call functions. Instead, we read the following sector(s) from the boot disk, load it/them into memory and finally we jump to the memory address. We can do this the hard way using ports or the easy way using the BIOS interrupts (when we're still in Real mode). I choose the easy way, as always.

How do I do this?

We start as always, by creating a file (*tutor3.asm*) and typing:

```
[BITS 16]

[ORG 0x7C00]
```

When the BIOS jumps to our boot sector, it doesn't leave us empty handed. For example, to read a sector from the disk, we have to know what disk we are resident on. Probably a floppy disk, but it could as well be one of the hard drives. To let us know this, the BIOS is kind enough to leave that information in the *DL register*.

To read a sector, the INT 13h is used. First of all, we have to 'reset' the drive for some reason. This is just for security. Just put 0 in AH for the RESET-command. DL specifies the drive and this is already filled in by our friend, the BIOS. The INT 13h returns an error code in the AH register. This code is 0 if everything went OK. We assume that the only thing that can go wrong, is that the drive was not ready. So if something went wrong, just try again.

```
reset_drive:
mov ah, 0
   int 13h
   or ah, ah
   jnz reset_drive
```

The INT 13h has a lot of parameters when it comes to reading and loading a sector from the disk to the memory. This table should clearfy them a bit.

Register	Function
ah	Command - 02h for 'Read sector from disk'
al	Number of sectors to read
ch	Disk cylinder
cl	Disk sector (starts with 1, not 0)
dh	Disk head
dl	Drive (same as the RESET-command)

Now, where shall we put our boot sector. We have the whole memory by our selves. Well, not the reserved parts, but almost the whole memory. Remember, we placed our stack in 090000h-09FFFFh. I choose 01000h for our 'kernel code'. In real mode (we haven't switched yet), this is represented by 0000:1000. This address is read from es:bx by the INT 13h. We read two sectors, just in case our code happends to get bigger than 512 bytes (likely).

```
mov ax, 0
mov es, ax
mov bx, 0x1000
```

Followed by the INT 13h parameters and the interrupt call itself.

```
mov ah, 02h
mov al, 02h
mov ch, 0
mov cl, 02h
mov dh, 0
int 13h
or ah, ah
jnz reset_drive
```

Now, we should have the next sector on the disk in memory address 01000h. Just continue with the code from tutorial 2 with two little ajustments. First, now that we're going to clear the screen, we don't need our 'P' at the top right corner anymore. And instead of hanging the computer, we will now jump to our new C-code.

```
cli
xor ax, ax
.
.
.
mov ss, ax
mov esp, 090000h
```

Now, we want to jump to our code segment (08h) and offset 01000h. Remember, we didn't want our 'P' either. Change the following four lines:

```
   mov 0B8000, 'P'
   mov 0B8001, 1Bh

hang:
   jump hang
```

To:

```
jump 08h:01000h
```

Don't forget to fill the rest of the file...

```
gdt:
gdt_null:
.
.
.
times 510-($-$$) db 0
   dw 55AAh
```

Moving on to actually writing the second sector =). This should be our main(). Our main() function should be declared as void and not as int. What should it return the integer to? We must declare the constant message string here, because I don't know how to relocate constant strings within a file (anyone know how to do this?). This works, but it's kind of ugly...

```
const char *tutorial3;
```

I always put the word *const* in, whenever possible. That's because it keeps me from making mistakes. Sometimes, it's good and some times it ain't. Most of the time it's good to have it.

First of all, we wan to clear the screen, then we print our message and go into an infinite loop (hang). Simple as that.

```
void main()
{
   clrscr();
   print(tutorial3);
   for(;;);
}
```

But wait a minute?! You haven't declared clrscr() or print() anywhere? What's up with that? No, that's true. Because of my lack of knowledge of the linker, I don't know how to do that. This way, if we spelled everything right, the linker finds the appropriate function. If not, our OS will tripple fault and die/reset. Ideas are welcome here...

After main(), we place our string. After that, main.c is complete!

```
const char *tutorial3 = "MuOS Tutorial 3";
```

Now for our other functions. We place them in a file called '*video.c*'. clrscr() is the easy one, so let's start with that.

```
void clrscr()
{
```

We know that the video memory is resident at 0xB8000. So we start by assigning a pointer to that location.

```
   unsigned char *vidmem = (unsigned char *)0xB8000;
```

To clear the screen, we just set the ASCII character at each position in the video memory to 0. A standard VGA console, is initialized to 80x25 characters. As I told you in tutorial 2, the even memory addresses contains the ASCII code and the odd addresses, the color attribute. By default, our color attributes should be 0Fh, white on black background, non-blinking. All we have to do, is to make a simple for-loop.

```
   const long size = 80*25;
   long loop;

   for (loop=0; loop<size; loop++) {
      *vidmem++ = 0;
      *vidmem++ = 0xF;
   }
```

Now for the cursor position. If we cleared the screen, we also want our cursor to be in the top right corner. To change the cursor position, we have to use two assembly commands: *in* and *out*. The computer has *ports* which is a way to communicate with the hardware. If you want to learn more, have a look at Chapter 9 in Intel's first manual(1.1MB PDF).

It's a little tricky to change the cursor position. We have two ports: 0x3D4 and 0x3D5. The first one is a index register and the second a data register. This means that we specify what we want to read/write with 0x3D4 and then do the actual reading and/or writing from/to 0x3D5. This register is called CRTC and contains functions to move the cursor position, scroll the screen and some other things.

The cursor position is divided into two registers, 14 and 15 (0xE and oxF in hex). This is because one index is just 8 bits long and with that, you could only specify 256 different positions. 80x25 is a larger than that, so it was divided into two registers. Register 14 is the MSB of the cursor offset (from the start of the video memory) and 15 the LSB. We call a function out(unsigned short _port, unsigned char _data). This doesn't exist yet, but we'll write it later.

```
   out(0x3D4, 14);
   out(0x3D5, 0);
   out(0x3D4, 15);
   out(0x3D5, 0);
}
```

Now, to write the out() and in() functions, we need some assembly again. This time, we can stick to C and use inline assembly. We put them in a seperate file called '*ports.c*'. First, we have the in() function.

```
unsigned char in(unsigned short _port)
{
```

This is just one assembly line, so if you want to know more about the in command, look in Intel's second manual(2.6MB PDF). Inline assembly is kind of special in GCC. First you program all your assembly stuff and then you specify inputs and outputs. We have one input and one output. The input is our port and the output is our value recieved from *in*.

```
   unsigned char result;
   __asm__ ("in %%dx, %%al" : "=a" (result) : "d" (_port));
   return result;
}
```

This looks rather messy, but I'll try to explain. The two %% says that this is a register. If we don't have any inputs or outputs, only one % is required. After the first ':', the outputs are lined up. The "=a" (result), tells the compiler to put result = EAX. If I'd write "=b" instead, then result = EBX. You get the point. If you want more than one output, just put a ',' and write the next and so on. Now to the outputs. "d" specifies that EDX = _port. Same as output, but without the '='. Plain and simple =).

Now to the out(). Same as for in(), but with no outputs and two inputs instead. I hope this speaks for itself.

```
void out(unsigned short _port, unsigned char _data)
{
   __asm__ ("out %%al, %%dx" : : "a" (_data), "d" (_port));
}
```

Then we have the print(). Three variables are needed. One pointer to the videomemory, one to hold the offset of the cursor position and one to use in our print-loop.

```
void print(const char *_message)
```

```
{
   unsigned char *vidmem = (unsigned char *)0xB8000);
   unsigned short offset;
   unsigned long i;
```

We want print() to write at the cursor position. This is read from the CRTC registers with the in() function. Remember that register 14 holds bits 8-15, so there we need to left shift the bits we read. We increase the vidmem pointer by two times offset, because every character has both ASCII code and a color attribute.

```
   out(0x3D4, 14);
   offset = in(0x3D5) << 8;
   out(0x3D4, 15);
   offset |= in(0x3D5);

   vidmem += offset*2;
```

With a correct vidmem pointer, we're all set to start printing our message. First we initialize our loop variable *i*. The loop should execute as long as the value we are next to print, is non-zero. Then we simply copy the value into vidmem and increase vidmem by two (we don't want to change the color attribute).

```
   i = 0;
   while (_message[i] != 0) {
      *vidmem = _message[i++];
      vidmem += 2;
   }
```

Our message is printed and all that is left to do is to change the cursor position. Again, this is done with out() calls.

```
   offset += i;
   out(0x3D5, (unsigned char)(offset));
   out(0x3D4, 14);
   out(0x3D5, (unsigned char)(offset >> 8));
}
```

To compile, we start with the boot sector.

```
nasmw -f bin tutor3.asm -o bootsect.bin
```

For the rest of the C-files, we first compile each file seperatly and then link them together.

```
gcc -ffreestanding -c main.c -o main.o
gcc -c video.c -o video.o
gcc -c ports.c -o ports.o
ld -e _main -Ttext 0x1000 -o kernel.o main.o video.o ports.o
ld -i -e _main -Ttext 0x1000 -o kernel.o main.o video.o ports.o
objcopy -R .note -R .comment -S -O binary kernel.o kernel.bin
```

'-i' says that the build should be incremental. First link without it, because when '-i' is used, the linker doesn't report unresolved symbols (misspelled function names for example). When it linkes without errors, put '-i' to reduce the size. '-e _main' specifies the entry symbol. '-Ttext 0x1000' tells the linker that we are running this code at memory address 0x1000. Then we just specify what output format we want, the output file name and list out .o-files, starting with main.o (important!). The objcopy line make the .o-file to a plain binary file, by removing some sections.

We're not done yet. We have our boot sector and our kernel. The boot sector assumes that the kernel is resident the two following sectors on the same disk. So, we need to make them into one file. For this, I've made a special program in C. I'm not going into any details about it, but I'll include the source code.

The program is called 'makeboot' and takes at least three parameters. The first one is the output file name.

This can be 'a.img' in our case. The rest of the parameters are input files, read in order. We want our boot sector to be placed first and then our kernel.

```
makeboot a.img bootsect.bin kernel.bin
```

Just run bochs with a.img and this is what you should get:

Download the complete source for this tutorial, including makeboot and a .bat-file for compiling.

Any comments, improvments or found errors? Mail me: gregor.brunmar@home.se.