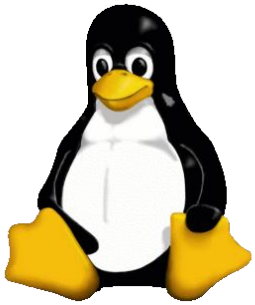
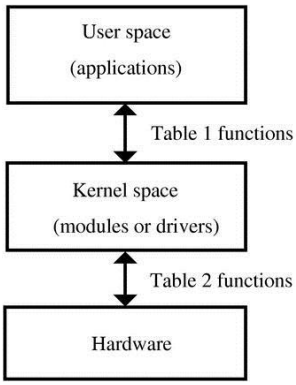
Lab 4: An Introduction to Linux Device Drivers



This lab will teach you the basics of writing a device driver in Linux. By the end of the lab, you will be able to (1) control the GPIO in a number of ways and (2) talk to an SPI device in user space. The first bit of this lab is based on the fantastic device driver tutorial by Xavier Calbet at Free Software Magazine[1]. We recommend taking a look at his article before starting the lab.



1. Prelab

Q1. In the context of Linux device drivers, what is a major number? What is a minor number?

Q2. What are three different types of Linux device files? Give an example of a device file and a hardware device for each type.

Q3. What is a pseudo-device, and how is it different from a normal device? Give an example of a pseudo-device in Linux.

Q4. What is the difference between a Beagleboard and Beaglebone? Your answer should be 2-3 sentences in length.

Q5. Look at http://beagleboard.org/static/beaglebone/latest/Docs/Hardware/BONE_SRM.pdf. Where on the board is GPIO1_6? GPIO2_11? Provide a port & pin number.

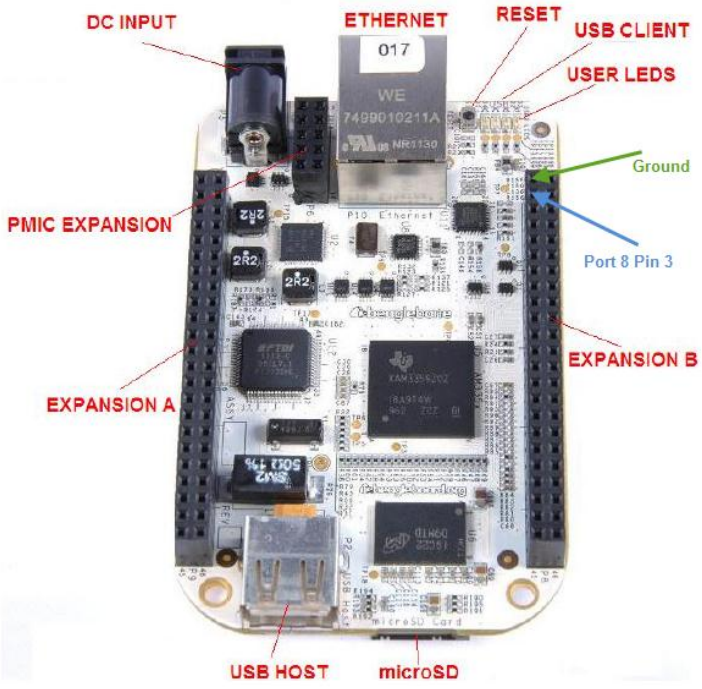


Figure 1: Labeled Beaglebone, from http://www.gigamegablog.com and BONESRM.

[1] http://www.freesoftwaremagazine.com/articles/drivers_linux

Q6. The documentation for our processor is at http://www.ti.com/lit/ug/spruh73f/spruh73f.pdf. You may find it useful to save a local copy. Use it to answer the following questions:

a) At what memory location do the GPIO0 registers start? (Hint: go to memory map and look for GPIO0) GPIO1?

b) Describe the behavior of the following registers: GPIO_OE, GPIO_CLEARDATAOUT, GPIO_SETDATAOUT.

c) At what memory location is the GPIO_OE for GPIO1?

Q7. Write a C function which returns the address of a specific GPIO register given:

- An integer that specifies which bank of GPIO registers you are concerned with (0, 1, 2, or 3)
- An integer or enum that specifies which specific register you wish to use (OE, CLEARDATAOUT or SETDATAOUT).
  - If you don't use an enum, use #defines to provide a more useful interface for identifying the specific register.

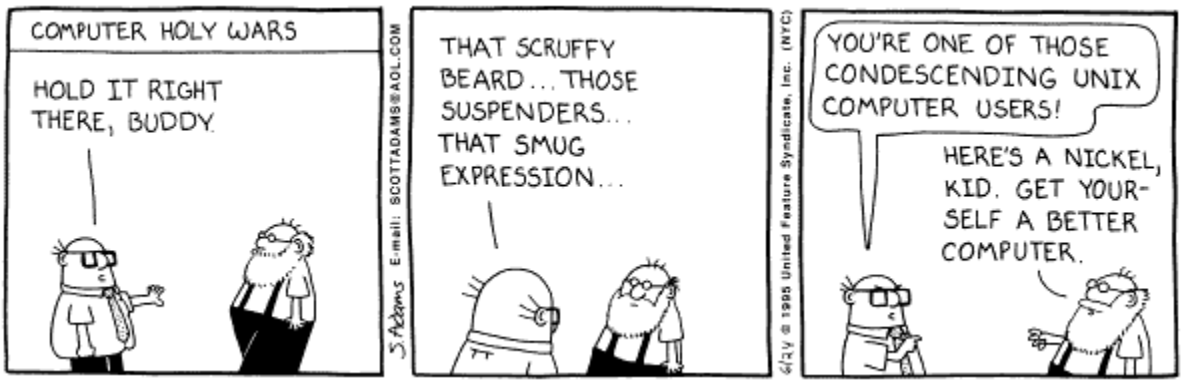
Q8. Look up the function call `mmap()`.

a) What does the function do? In particular, what does the function return?

b) Detail each of the arguments to mmap().

In addition you will find it helpful to take a look at these links before you come to lab.

- The first part of http://squidge.sourceforge.net/gpio/ (up to, but not including, "Find that GPIO pin!")
- http://www.makelinux.net/ldd3/chp-9-sect-4
- http://www.freesoftwaremagazine.com/articles/drivers_linux
- http://oreilly.com/catalog/linuxdrive3/book/ch03.pdf (same book as makelinux link, but a full chapter).



2. Inlab

This lab is broken into a number of small parts. First, you'll need to get everything ready to go. We are working with a cross compiler (that is we'll be compiling code for the Beaglebone on the host x86 machine and moving the files over). Once that's done, parts A, B and the first bit of C get you used to getting around the Linux device driver environment. We supply all needed code for the pseudo devices you'll build those parts—you just need to get them up and running. The rest of part C will have you modifying our pseudo device to create a slightly more advanced pseudo device.

In part D you'll start playing with real IO—a single GPIO pin. You'll do this in a number of ways including using a prebuilt device, writing a user space device that talks directly to the pin and writing a kernel space device. Part D will probably be fairly time consuming. In part E you will be doing a bit with a built-in SPI driver and learning to use our scopes to see the SPI transactions.

We'd love to do more; there are large chucks of information missing that you'll need if you do embedded systems development in Linux. In particular we have to entirely skip interrupts and almost entirely skip ioctl. The post lab will get you at least vaguely familiar with interrupts.

Lab computer setup

Before we dive into device driver land, we need to make a few changes to the lab computers and otherwise configure things.

1. **Boot into Linux**
   When you restart your computer, the bootloader (GNU GRUB) will show a list of boot options with Windows XP at the top of the list. Choose the Linux opion.

   When Ubuntu prompts you for a username and password, use the following administrator account:
   username: eecs2334
   password: eecs2334rules
   Note: You have (and will need) superuser permissions. Others will be using the same account on the same computer. And others (373 or us) will sometimes be using the computer you are using later. As such:
   - As such, you'll want to back up your work. Some of the directories you will be creating will be huge, so you want to only backup the stuff you are creating. Dropbox, github or something else might be a good way to go. A USB drive might also be doable.
   - Please clear your work off the machine when you are done.
   - Try (really hard) not to trash the OS install.

   With great power comes great responsibility!

2. **Hook up a Beaglebone**
   Get a Beaglebone, a power supply, an Ethernet switch or hub, and two Ethernet cables. Be sure to only connect a 5V power supply to the Beaglebone (it can't take anything else). It's easy to confuse the switch's power supply with the Beaglebone, so be careful.

   You'll need to disconnect the computer you are working on from the wall's Ethernet, and connect the wall, computer and Beaglebone to the switch with Ethernet cables. You'll

need to power the router and the Beaglebone (taking care to use the correct adapters!) **Don't** hook up the USB connection.

3. **Dual monitors (optional)**
   - On the top bar across your desktop click the gear and select **displays**…. We suggest you turn off mirrored displays and otherwise configure it as you wish. Turning off sticky edges is suggested.

4. **Get the host ready for cross compiling**
   Download the files needed to the host machine (Linux box) and uncompress the tarball (tar xfz tarball_name.tar.gz). The rest of these directions assume you untar it in your home directory (where you get to if you just type **cd**). It's pretty big so it might take a while.[2] It is on the lab webpage.

5. **Get the Beaglebone running the correct version of the OS.**
   - Check that you can talk to the Beaglebone. This is, sadly, harder than you'd think. The problem is that the Beaglebone puts itself on the network as "beaglebone.local". If there is already something with that name (which in lab there will be) it will instead be named "beaglebone-x.local" where x is an integer.
     - The trick is to figure out the name of _your_ board. The best bet is check to see what's on the network (**avahi-browse -ac | grep beagle**), plug yours in and wait a minute and see what got added (typing the same thing again). Now you have a number. **The rest of the document assumes you are "beaglebone.local" rather than anything else, so you'll need to adjust any relevant command appropriately**.
   - Try to ssh into root@beaglebone.local (no password) and see what happens. Note that after you've connected everything it may take a while (3 minutes?) for the Beaglebone to be visible on the network.
   - While in ~/ type **scp kernel/kernel/arch/arm/boot/uImage root@beaglebone.local:/boot/uImage-3.2.25+**
   - Create a tarball of ~/kernel/rootfs and use sftp to move it onto the Beaglebone to the / (root) directory. (You will want to be in that directory when you create the tarball)
   - Login to the Beaglebone and untar the file in / (root) directory.
   - Type the following from the host (or do the same thing from the Beaglebone but without the ssh and the single quotes)

```
ssh root@beaglebone.local 'cd /boot; rm uImage'
ssh root@beaglebone.local 'cd /boot; ln -s uImage-3.2.25+ uImage'
ssh root@beaglebone.local 'mount /dev/mmcblk0p1 /mnt'
ssh root@beaglebone.local 'cp /boot/uImage-3.2.25+ /mnt/uImage'
ssh root@beaglebone.local 'umount /mnt'
```

   - Reboot the board. It will take a while for things to start working, but you should see a heartbeat on the board (LED blinking like a heartbeat) in about a minute. Even after that, it will likely take a few more minutes for your computer to see it correctly. Go for a walk or something…

[2] FYI: The code in this tarball was built following the directions at https://github.com/beagleboard/kernel/tree/6682025752d0b807119c1e363a0b1b9bfe2ab453,

- Type **`ssh root@beaglebone.local 'uname -a'`**
    - That should result in a message about version 3.2.25+ being installed.

6. **Get the host set to do cross-compiling.**
   On the host type

   `sudo apt-get install -y git lzop gcc-arm-linux-gnueabi uboot-mkimage.` This will install the cross compiler.

   You need to either type the following each time you open a new window or put it in your .bashrc file. ***Unlike the other steps, you'll need to do this every time you create a new window or log in.*** Though you could put it in your .bashrc file…

```
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabi-
```

A. The "Nothing" driver

The purpose of this section is to familiarize yourself with the tools you will need to build more exciting device drivers in the later sections.

Note: you will do this on the host!

In an empty folder, create a new file called "nothing.c".

```
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
```

As discussed in class, MODULE_LICENSE is a way for the kernel to know if it has been tainted with proprietary code. (As an additional exercise, consider checking what happens when you try to insert the module without specifying the license.)

Next, create the following Makefile. obj-m (object-module) provides a list of the kernel modules to build during make:

```
obj-m := nothing.o
```

We need to compile our module with the same kernel that will be receiving our module. Run the following make command in the same directory as our two files from before.

```
$ make -C ~kernel/rootfs/lib/modules/3.2.25+/build M=$PWD modules
```

Notice that make produces a nothing.ko file; this file is our module. Move the .ko file to your beaglebone then insert our module into the kernel, run insmod as root:

```
$ insmod nothing.ko
```

Q1. What is a .ko file? Why didn't we just compile our code into an executable?

Use lsmod to check that your module has been inserted correctly. You should a "nothing" entry in the list of modules.

```
$ lsmod
```

Use the following command to remove the module. Verify that the module has been removed from the list with lsmod.

```
$ rmmod nothing
```

B. The "Hello World!" driver

In this section, you will write a device driver that prints simple messages to the kernel during module insertion and removal.

Create a new "hello.c" file in the same directory as our nothing module on the host:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void) {
  printk("<1> Pork chop sandwiches!\n");
  return 0;
}

static void hello_exit(void) {
  printk("<1> What are you doing? Get out of here!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Add hello.o to the list of object modules in the Makefile:

```
obj-m := nothing.o hello.o
```

After you build and insert the module into the kernel on the target, you can view the module's printk messages by running the following command:

```
$ dmesg
```

Dmesg simply prints the kernel's message buffer. Equivalently, we could also cat the syslog:

```
$ cat /var/log/syslog
```

C. Memory module

In this section, we will write a driver that will allow us to read and write a character to a pseudo-device. Download the memory module code from the website, build it and insert it into the kernel.

After you've inserted the memory module into the kernel, we need to create a device file that will let us interact with our driver from userspace. In the command below, we're telling mknod (make node) to create a new character device file (c) with major number 60 and minor number 0:

```
$ mknod /dev/memory c 60 0
```

We also need to unprotect the device file before normal users can interact with it:

```
$ chmod 666 /dev/memory
```

We should now be able to echo characters to our device file from the terminal. In your terminal, type the following command:

```
$ echo -n honeybadger >/dev/memory
```

To check the contents of our memory driver, use the cat command:

```
$ cat /dev/memory
```

Q2. What was the output of cat, and why did it output this letter?

G1. Write a userspace program that opens your memory driver as a file, writes the string "Honey badger", reads a character back, and prints that character to the terminal.

Now go back and look carefully at that code. In the case of read, you are effectively returning three values and you'll need to understand them to handle the next part. At least read over the section in http://oreilly.com/catalog/linuxdrive3/book/ch03.pdf on "read and write" (starts on page 63) but looking over that whole chapter might be good.

G2. Modify the memory driver so that it outputs the last 5 characters sent to it. Test it with both your userspace program and by writing 2 characters at a time to the device and checking that the output is as expected.

D. Driving GPIO

The Beaglebone has a fair bit of general purpose I/O. In this section we will work with that I/O in three different ways:

1. User space using pre-built devices
2. User space using mmap **`/dev/mem`**
3. Kernel space

D1: Driving GPIO in user space using existing drivers

Here we'll use device drivers already written for us. Let's use GPIO1_6. In Linux space this maps to GPIO38 (1*32+6, GPIO2_3 would be 2*32+3). Look in **`/sys/class/gpio`** on the target. You should see a few files in there, including **`export`**, but no gpio38. Type **`echo 38 > export`** when you are in that directory. Now a gpio38 directory should show up. Change directory into that and you'll see a few device files. Read the first part of http://squidge.sourceforge.net/gpio/ for a background these files and directors (that documentation is for a different board, but it's the best explanation we've found).

Write a short C program which toggles GPIO1_6 as fast as you can by using the device files in /dev/class/gpio/gpio38.

Q3. Take the following measurements:

1. What period does the signal have? Use the measurement tool on our scope to get a min, max and standard deviation.
2. Use the trigger tool on the scope to trigger if the period is 2x the average you've measured above. Use the "single" run-control option to see what is going on.
3. What is the longest period you get in 10K samples?
4. Explain the results you saw in part b.
5. Why might the above parts indicate a significant problem with using the Beaglebone with Linux as the basis for an embedded system?

GS. Show your GSI that you can capture an instance of the period being significantly larger than it should be.

D.2: Driving GPIO in user space via mmap

Consider the following C code. It opens **`/dev/mem`** to provide a direct map into physical memory. When you finish you'll have a pointer into a specific place in physical memory.

```
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>

volatile unsigned long* init_memmap()
```

```
{

   int gpio_fd = open("/dev/mem", O_RDWR | O_SYNC);
   if (gpio_fd < 0) {
      printf("Could not open GPIO memory!?\n");
      exit(0);
   }

   gpio = (unsigned long*) mmap(NULL, MMAP_SIZE, PROT_READ |
           PROT_WRITE, MAP_SHARED, gpio_fd, MMAP_OFFSET);
   if (gpio == MAP_FAILED) {
      printf ("GPIO Mapping failed\n");
      close(gpio_fd);
      exit(0);
   }
    return gpio;
}
```

Go back and review your answers to the prelab questions as they should help quite a bit with the next few parts.

Q4. What would be a reasonable value of MMAP_OFFSET and MMAP_SIZE if you want to be able to address all GPIO pins? Try to keep the size of the map as small as is reasonable.

You might want to check on your answer to that question with your GSI before proceeding as you'll really struggle if you don't get a reasonable answer to that question.

G4. Using the code above, write a short program which sets GPIO1_6 to be an output and then toggles the output every half a second (a 1 Hz signal +/- 10%). Demonstrate this to your GSI on the oscilloscope.

Q5. Redo problem Q3 but for this scheme. Are there significant differences?

NOTE: Be aware that digital inputs greater than 3.3V can harm the board[3]!!!

Write GPIO functions which can manage the basic I/O needed to talk to any one GPIO pin. You should be able to set direction, clear, set and read a given pin. Try not to have much in the way of cut-and-paste code. Using those functions write a short program where one pin outputs the negation of the value read on a different pin (e.g. if the input pin is "1" the output is "0"). Use the

[3] Not that we are using it, but the analog maximum is 1.8V. Just so you know…

3.3V output from the board (P9, pins 3 and 4) to avoid getting too high of a voltage.

G5. Once you've got your code working, your GSI will give you specific pins to use for your demonstration.

Q6. Print those GPIO functions and attach them to your report.

D.3. Driving GPIO in kernel space

Now you are going to create a device named "bob" in /dev which controls GPIO1_6 as a loadable kernel module. Read http://www.makelinux.net/ldd3/chp-9-sect-4. You probably should copy your memory device and use that as a basis for your driver. There is a fair bit of flexibility here on exactly how to write this.

G6. Demonstrate your working code to your GSI.

Q7. Print the code you wrote.

E. IOCTL

While we can communicate to our device through the device driver, we currently have no way of communicating with the device driver itself. For example, we might have a CD-ROM driver that can read and write blocks of data, but have no natural way to request that the driver eject the disc. For this purpose, Linux provides the ioctl syscall. We can define request codes that we pass through ioctl from user space to implement device-specific functions. To request a parameter or data structure from the device driver, we can pass in a reference to that data structure through ioctl.

Modify the device driver we gave you in part C (not the one you modified) by adding the following code in the appropriate sections:

```
int memory_ioctl (struct inode *inode, struct file *filp,
                  unsigned int cmd, unsigned long arg);

struct file_operations memory_fops =
{
  .read = memory_read,
  .write = memory_write,
  .open = memory_open,
  .release = memory_release,
  .unlocked_ioctl = memory_ioctl
}
```

```
int
memory_ioctl (struct inode *inode, struct file *filp,
```

```
                unsigned int cmd, unsigned long arg)
{
  printk("<1>in ioctl");
  if(cmd==0)
   *memory_buffer^=0x20;
  else
        *memory_buffer|=0x20;
  return(0); // success!
}
```

Once you've done that install it in the same way you did the single-character driver from part C.

Now build and run the following code on the Beaglebone:

```
#include <sys/ioctl.h>
#include <stdio.h>
#include <fcntl.h>
main()
{
   int a=open("/dev/memory",O_RDWR);
   ioctl(a,2,3,4);
}
```

Q8. What does the userspace program do? What is the value of cmd that memory_ioctl is seeing?

F. SPI using prebuilt drivers

The Beaglebone comes with a SPI driver for user space. Sadly, it isn't quite as easy to use as one might like. That said, on the website there is some code that uses the SPI driver. Download it and get it running. Note: the code assumes you are in loopback mode so jumper P9/29 to P9/30. You should see a printout of the output that matches the input array (in the file). If things aren't working you'll likely see all FF packets (rather than just most).

Q9. Look at the pins we connected. Explain exactly what those pins are and what adding the jumper did.

G7. Look at the documentation for the MSO-X 3012A[4] scopes. Get the SPI display mode working and show your GSI.

[4] http://cp.literature.agilent.com/litweb/pdf/75019-97051.pdf, chapter 25…

3. Postlab

We really need to cover interrupts, but this lab is already too long. It's possible, but unlikely, we'll squeeze Linux interrupts into the next lab. But for now, take a look at chapter 10 of http://lwn.net/Kernel/LDD3/ and answer a few questions.

Q10. On the Beaglebone, what do you get when you type cat /proc/interrupts? What does that output mean?

Q11. Consider the function short_interrupt(). In your own words, what is this function's purpose?