BeagleBone Robotics

T Bleything, J Strawson 2014

1 Introduction

When embarking on a new robotics project, your first decision is usually to choose which embedded microprocessor to use to control and communicate with your robot. We use the term "embedded microprocessor" to refer to any one of the thousands of types of processors designed to be *embedded* into larger devices and systems ranging from your microwave or television to jumbo jets. Obviously the amount of memory and processing power required of the embedded processor will vary greatly depending on its use.

Most small robotics projects in the educational space will have fairly minimal processing requirements, but will benefit greatly from having a wide variety of connectivity interfaces. This will allow a single embedded system to be used on different projects with unique requirements and functions. For this reason, the following text will focus on the use of the BeagleBone Black development board as it offers the power and functionality of a full Linux-based operating system with a small form factor that is well-suited for robotics projects.

Here we will outline the usage of the BeagleBone Black as a robotics control unit and describe in detail the most commonly used interfaces and techniques in the field of robotics prototyping.

1.1 Your BeagleBone's Operating System

Unlike many cheap embedded development kits like Arduinos and PIC development boards, your BeagleBone has enough flash storage and volatile RAM to support a full-featured operating system. Thanks to the efforts of the BeagleBoard open-source community and Texas Instruments, your BeagleBone comes preinstalled with a custom build of the Debian Linux operating system. In the context of your robotics project, this means that it takes very little effort to allow your robot to support high-level functions such as networking, file system management, and the execution of several programs at once.

Unlike popular 8-bit microcontrollers which have on the order of kilobytes of flash storage, you can store many programs and projects on your BeagleBone at once. This accelerates your software development process by allowing you to edit, compile, and test your programs while keeping your BealgeBone powered on. Without an operating system, your embedded system would need to be flashed and restarted with each code revision. Furthermore, your BeagleBone has the ability to drive an HDMI display and render a graphical user interface. By connecting a USB keyboard and mouse, your BeagleBone can serve as a low-power home desktop computer.

Just like your personal computer at home, you should cleanly power off your BeagleBone before disconnecting power. This can be done with either the poweroff command or by momentarily pressing the power button closest to the ethernet jack. Since the operating system is writing to the file system occationally in the background, shutting down your BeagleBone properly will help prevent file system errors that will require you to need to flash your BeagleBone.

1.2 Flashing Your BeagleBone to a Clean Image

While your BeagleBone comes pre-flashed with the latest Debian image from the factory, it is sometimes a good idea to flash back to a clean image if you are unsure of the state of your BeagleBone's memory or just want to ensure a clean start. Follow the steps on beagleboard.org/Getting+Started to create a microSD card and flash the official BeagleBone Debian image to the on-board eMMC flash storage. All instructions and code in this guide are tested on the 2014-05-14 release of Debian found at beagleboard.org/latest-images.

2 Connecting and Communicating With Your BeagleBone

One of the many benefits of having an operating system on your robot is that you can communicate with it through standard network protocols in your own home without sophisticated telemetry systems. Typically, mobile robots are used in a headless configuration, meaning without a graphical user interface. Therefore, you must control your BeagleBone through network protocols instead of with a desktop environment rendered by the BeagleBone. Luckily, we can still use another computer's GUI when developing your BeagleBone. We will call this your host computer while your BeagleBone will be the robot.

The first step when learning to use your BeagleBone is connecting through USB and getting a working console open. This will allow you command-line access to your BeagleBone's operating system that will let you move forward to more sophisticated techniques like setting up an enthernet or WiFi connection.

2.1 Network over USB

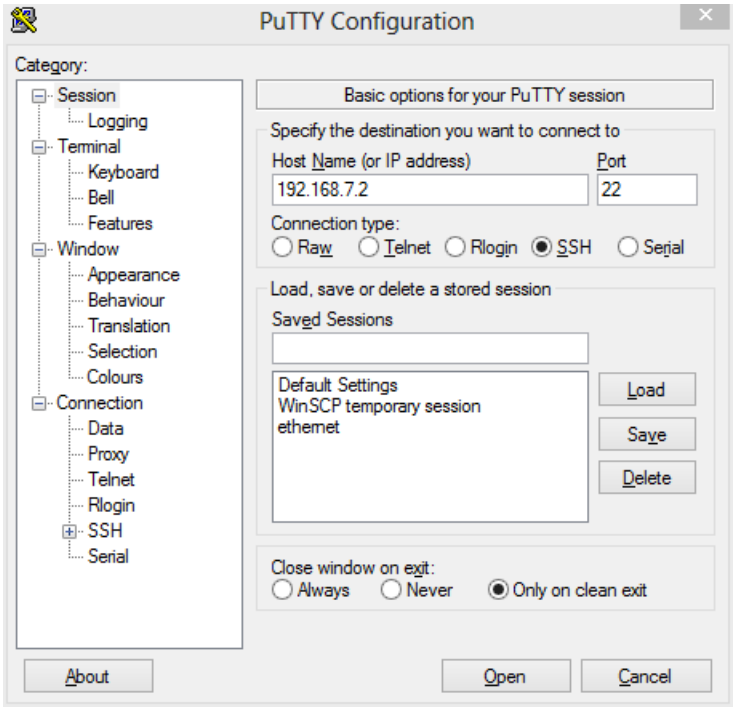
A remarkably convenient method of getting a network connection with your BeagleBone while also powering it is over a mini-USB cable. This is accomplished with a USB driver which generates a network interface when your BeagleBone is connected, similar to an ethernet or WiFi interface. If you are using a major distribution of Linux then the USB drivers to generate the network interface will probably load automatically when you connect your BeagleBone over USB. A set of drivers and setup guide are available at the BeagleBoard Getting Started page. Note that the windows drivers distributed on beagleboard.org are unsigned. Getting Windows 8 to install unsigned drivers is annoying, so Jason Kridner provides us signed drivers here.

Connecting to your BeagleBone over a USB cable will probably remain your primary communication technique. Optionally, you can connect to your BeagleBone through a home network over either an ethernet or WiFi connection. We will cover these techniques later.

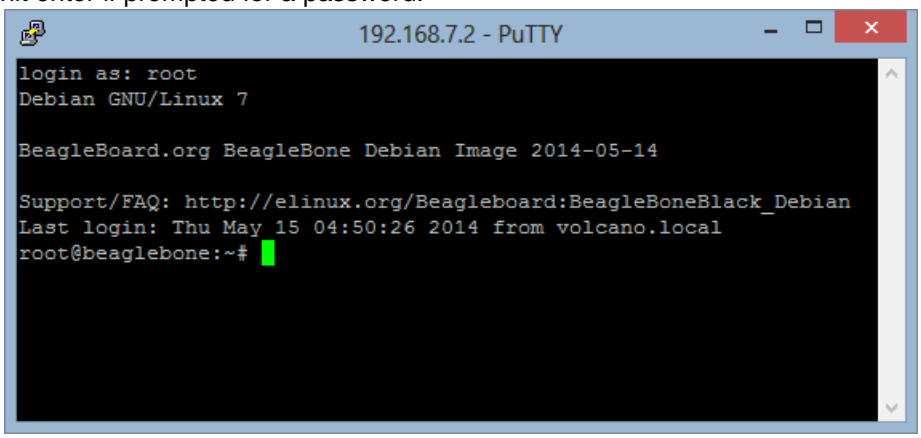
2.2 SSH

Once you have followed the Getting Started page linked above and installed the drivers, you are ready to "SSH into" your BeagleBone. Secure Shell, or SSH, is a network protocol allowing a user to initiate a command line session on the BeagleBone from a host computer. Mac and Linux operating systems generally come pre-installed with an SSH client and server. This allows you to SSH into your BeagleBone from a local command line application. For Windows users, we recommend the free and popular application PuTTY.

A convient feature of the USB network connection is that your BeagleBone will always be accessible at the same IP address: 192.168.7.2, exemplified by the below image using PuTTy.



At this stage, you should be able to log into your BeagleBone for the first time with the default username "root." Simply hit enter if prompted for a password.



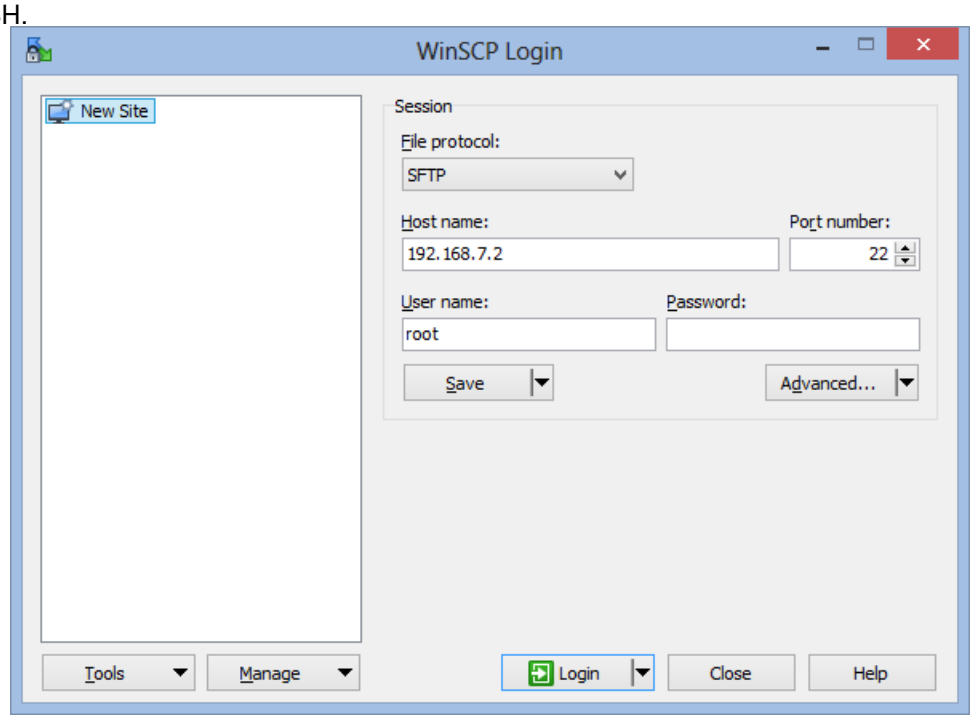
If you can see a similar login screen to mine, then you have successfully "SSH'd into" your BeagleBone. Now is good time to check the operating system version you have installed. You should use the 2014-05-14 Debian image if you intend to complete the Robotics Cape based exercises.

At this point you can interact with the command line just like any standard Linux operating system. If you are not familiar with using a Linux command line, I suggest reviewing the following most commonly used commands: ls, mkdir, rm, cd, nano, cat, echo, reboot, poweroff, make, gcc and top.

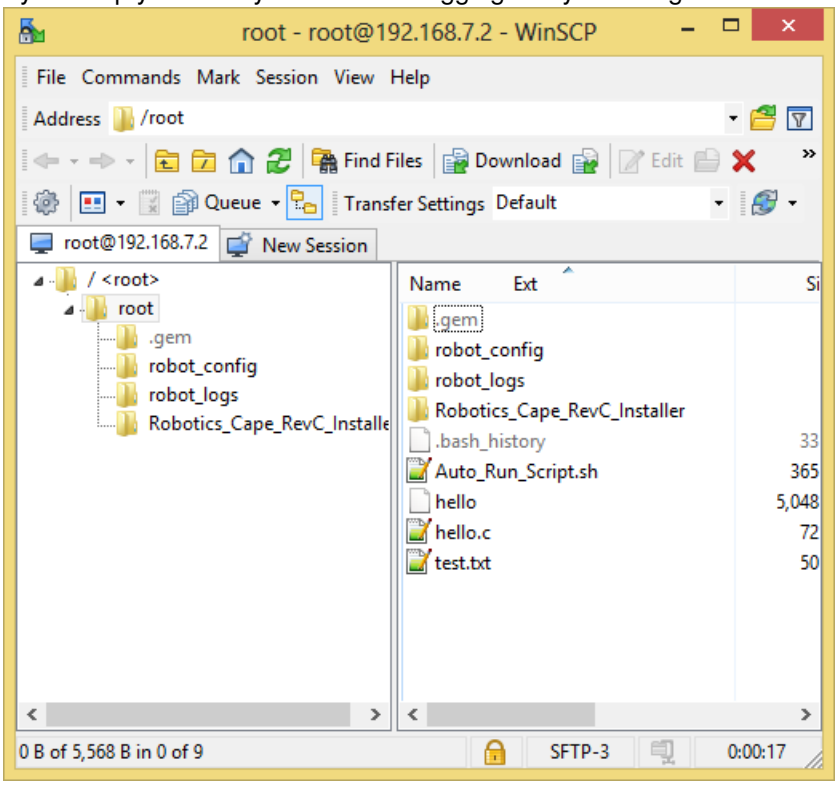
2.3 SFTP

We will not expect you to use command line file editors like nano or vim for writing long and complicated robotics control programs. Instead, we can save time by editing files locally on a host computer with a graphical interface and then sending the files back and forth to your BeagleBone with Secure File Transfer Protocol, or SFTP. Linux and Mac computers provide the FTP command-line program out of the box and there are many popular graphical FTP programs available. We recommend WinSCP for Windows users.

You should now try logging into your BeagleBone through an SFTP client at the same IP address you used for SSH.



If this is successful, you should be presented with a complete file system starting with the /root folder. This will be your main working directory. As you can see here, I have already put files and folders in my /root directory. Yours may be empty if this is your first time logging into your BeagleBone.



You can now send folders, text files, source code, and any other sort of file back and forth through the network.

Note that Windows operating systems place return carriage characters at the end of every new line in

text documents as well as a newline character whereas Linux only uses newline characters. Stray return carriage characters in files that you send to your BeagleBone can cause failures and should be avoided. Some Windows text editors such as Notepad++ will allow you to convert a file between the two standards with the end-of-line (EOL) conversion option. It is usually safest to create a new file with the command line while SSH'd into your BeagleBone using a command line text editor like nano or vim, then copy it to your computer for editing.

2.4 Ethernet

A more flexible way of acheiving a networking connection to your BeagleBone is by attaching it to a router through its ethernet port much like a desktop computer. However, with no graphical user interface on your BeagleBone, you must first discover the IP address of the BeagleBone before you can connect to it. This can be done one of three ways after you have powered up your BeagleBone with a DC power source and an ethernet connection.

1. Log into your network router's control interface and look for a list of connected network devices. One should appear with the hostname "beaglebone" with its corresponding IP address.

2. Attempt to connect using the hostname "beaglebone" directly. Note that this only works with routers that have internal DNS servers for local machines. This is the same way you can connect to a website knowing only the domain name.

3. In addition to an ethernet or wireless connection, you can plug in the USB cable to log into the BeagleBone and achieve two simultaneous network connections. Through the USB connection, you can log into 192.168.7.2 like before and ask the BeagleBone for the state of all network connections with the Linux tool ifconfig. Below you will see the result of me logging into my BeagleBone with both a USB and ethernet connection. You can see that my BeagleBone has two IP addresses. The familiar 192.168.7.2 address over USB, and also 192.168.1.82 over ethernet. You can now close your SSH session and try logging back in with the new network IP address to connect over USB.

```
root@beaglebone:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 90:59:af:82:b8:81
          inet addr:192.168.1.82  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::9259:afff:fe82:b881/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:555 errors:0 dropped:0 overruns:0 frame:0
          TX packets:125 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:71389 (69.7 KiB)  TX bytes:16418 (16.0 KiB)
          Interrupt:40

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
```

```
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

usb0      Link encap:Ethernet  HWaddr da:25:0d:36:09:69
          inet addr:192.168.7.2  Bcast:192.168.7.3  Mask:255.255.255.252
          inet6 addr: fe80::d825:dff:fe36:969/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:298 errors:0 dropped:0 overruns:0 frame:0
          TX packets:111 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:58871 (57.4 KiB)  TX bytes:25976 (25.3 KiB)
```

Two benefits of using an ethernet connection is that you can connect to multiple BeagleBones at once and you are no longer thethered to your USB cable. However, you do need to supply a DC power source to your BeagleBone either through the 5V input jack, the USB port, or with a cape that supplies DC power to the BeagleBone's VDD input pin.

2.5 WiFi

Thanks to the widely used and supported Debian operating system, your BeagleBone supports many USB WiFi dongles straight out of the box. Start by powering off your BeagleBone, installing your WiFi dongle to the USB host port, and then powering up your BeagleBone through the mini-USB port. Check that the drivers and network interface loaded by using the ifconfig like bebore. Mine is called wlan1, yours might be wlan0.

```
root@beaglebone:~# ifconfig -a
wlan1     Link encap:Ethernet  HWaddr 00:0c:13:09:1b:de
inet addr:192.168.1.12  Bcast:192.168.1.255  Mask:255.255.255.0
```

Now open the network config file for editing.

```
root@beaglebone:~# nano /etc/network/interfaces
```

Somewhere in this file will be an example entry that is commented out and looks similar to the one below. Uncomment it and edit the SSID and password to match your network.

```
# WiFi Example
auto wlan1
iface wlan1 inet dhcp
    wpa-ssid "beaglebase"
    wpa-psk  "beaglebone"
```

Now restart your BeagleBone safely from the command line with the reboot command to make the changes take effect. When your BeagleBone has booted back up, log in again through USB to check the connection with the ifconfig utility. If all was successful your router should have assigned your BeagleBone a unique IP address. You should write this down as this is the new IP address you will need to talk to your BeagleBone wirelessly instead of the 192.168.7.2 address which is through USB. It is possible to SSH into

your BeagleBone simultaneously through both the USB port and a wireless network.

To prevent you needing to remember the unique IP address of each device on a network, many routers support the use of hostnames to identify devices as well as their IP address. Your BeagleBone defaults to a hostname of "beaglebone". To change the hostname, edit the /etc/hostname file while logged in as the user 'root' with your favorite command line text editor such as nano or over sftp.

```
root@beaglebone:~# nano /etc/hostname
```

2.6 Exercise: Hello World

Once you have successfully SSH'd into your BeagleBone through one of the above-listed networking options, you have all you need to write and compile your first program. For this exercise we will walk through the standard Hello World program to ensure you have a functioning toolkit which is the collection of libraries, compilers, and editors necessary to develop software. Start by creating a new folder to put your project in.

```
root@beaglebone:~# cd /root/
root@beaglebone:~# mkdir hello
root@beaglebone:~# cd hello
root@beaglebone:~/hello#
```

Now create a new C file nammed helloworld.c

```
root@beaglebone:~/hello# nano helloworld.c
```

Paste or type in the following code.

```c
#include <stdio.h>

int main(){
    printf("Hello BeagleBone\n");
    return 0;
}
```

Hit Ctrl-O to save and Ctrl-X to exit the nano program. Now compile your program with gcc and provide the output file name matching the .c source file. If gcc returns with no errors or warning, execute the program and see the result.

```
root@beaglebone:~/hello# gcc helloworld.c -o helloworld
root@beaglebone:~/hello# ./helloworld
Hello BeagleBone
root@beaglebone:~/hello#
```

Congratulations, you have your first BeagleBone program working!

3 Capes

The expansion headers on your BeagleBone allow for accessory boards called Capes to be stacked on top of your BeagleBone. You will be using a Robotics Cape for most of the exercises in this document. Much like shields for the Arduino platform, BeagleBone capes allow you to add sensors and hardware specific to your application.

3.1 Slots

Since multiple capes can be stacked on a single BeagleBone, each installed cape occupies a slot in the stack of capes. This concept of a slot exists both in the physical position as well as in software. The BeagleBone cape manager software keeps track of the installed capes and which slot they are in. Out of the box, you will see that three virtual capes are loaded to enable EMMC flash memory and the HDMI interface. You can check which capes are loaded with the cat command as shown below. Note that the device tree overlay files can be loaded into cape slots without a physical cape needing to be installed.

```
root@beaglebone:~# cat /sys/devices/bone_capemgr.*/slots
 0: 54:PF---
 1: 55:PF---
 2: 56:PF---
 3: 57:PF---
 4: ff:P-O-L Bone-LT-eMMC-2G,00A0,Texas Instrument,BB-BONE-EMMC-2G
 5: ff:P-O-L Bone-Black-HDMI,00A0,Texas Instrument,BB-BONELT-HDMI
 6: ff:P-O-L Bone-Black-HDMIN,00A0,Texas Instrument,BB-BONELT-HDMIN
```

3.2 Exercise: Installing the Robotics Cape

To save you time, the Robotics Cape provides a single installation package which installs the device tree overlay file and many useful libraries. Simply follow the instructions here. I suggest selecting no program to run on boot if you are to proceed straight to the LED exercises in the next chapter. Note that the cape does not have to be physically installed to run the installer and use the libraries.

```
root@beaglebone:~#  cat /sys/devices/bone_capemgr.*/slots
 0: 54:PF---
 1: 55:PF---
 2: 56:PF---
 3: 57:PF---
 4: ff:P-O-L Bone-LT-eMMC-2G,00A0,Texas Instrument,BB-BONE-EMMC-2G
 5: ff:P-O-- Bone-Black-HDMI,00A0,Texas Instrument,BB-BONELT-HDMI
 6: ff:P-O-- Bone-Black-HDMIN,00A0,Texas Instrument,BB-BONELT-HDMIN
 7: ff:P-O-L Override Board Name,00A0,Override Manuf,SD-101C
```

If you are on an older Debian image such as 2014-04-23 then you will get an error from update-rc.d and the auto-run bootscript will not work. However, other library functions and the device tree overlay will still

function properly for this assignment. The proper solution is to upgrade to the 2014-05-14 Debian image.

Note that due to conflicting pin use, the Robotics Cape installer must disable HDMI functions. Since we typically use robots in a headless configuration without a display, this is generally not an issue. After installing the Robotics Cape and restarting, you can confirm that HDMI is disabled and that the new cape is loaded into a slot.

3.3 Pin Multiplexing

While there are many (92) pins on the BeagleBone expansion headers, there are even more IO channels used by the various subsystems on the AM335x processor used on the BeagleBone. The solution to this problem is called a pin multiplexer that lets you configure each pin to be used for one of up to 8 different functions. Each cape that is installed must have an associated device tree overlay file installed which configures the BeagleBone pin multiplexer to set up the header pins for use with that particular cape. This file also configures hardware parameters such as enabling serial ports and setting PWM frequencies.

Due to its complexity, the Robotics Cape uses almost all of the BeagleBone's header pins. To see how the pin multiplexer is set up with the Robotics Cape installed, download the header pin table from the Robotics Cape documentation page.

4 Circuit Design

Now that you can write and compile a “hello world” program, it’s time to start using General Purpose IO pins to control electric circuits. Regardless of the complexity of your robotics project, it is good practice to draw up a detailed schematic before making your wiring harness. Thorough documentation will reduce the likelyhood of damaging components from miswiring and make repairs easier in the future.

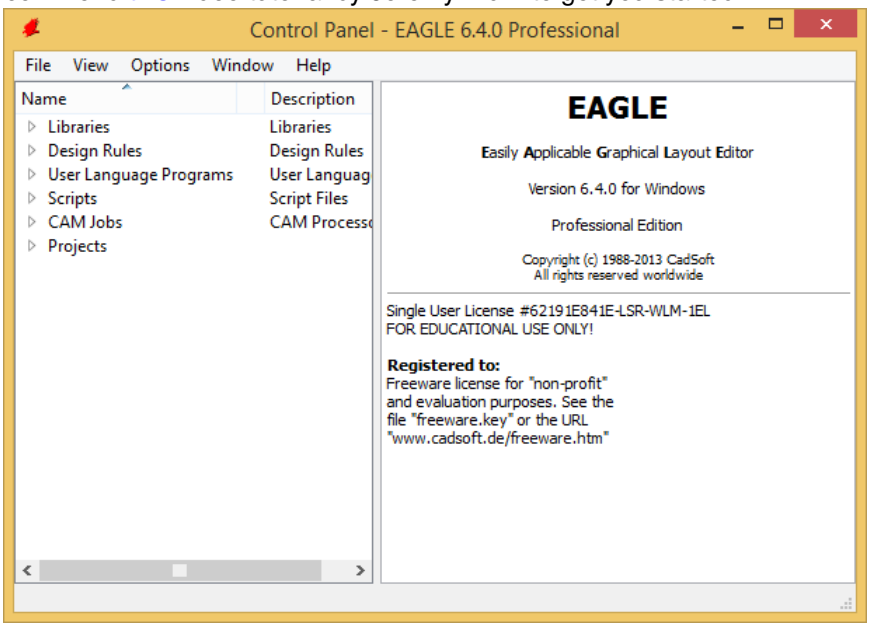
4.1 LED Kit Components

You will design three circuits in this chapter and you are provided with a kit containing the following parts which will allow you to construct and test them.These parts are meant to be reused. Be careful to push ICs square into the breadboard. Please leave ICs in the breadboard to protect pins and return the kit after the assignment is complete.

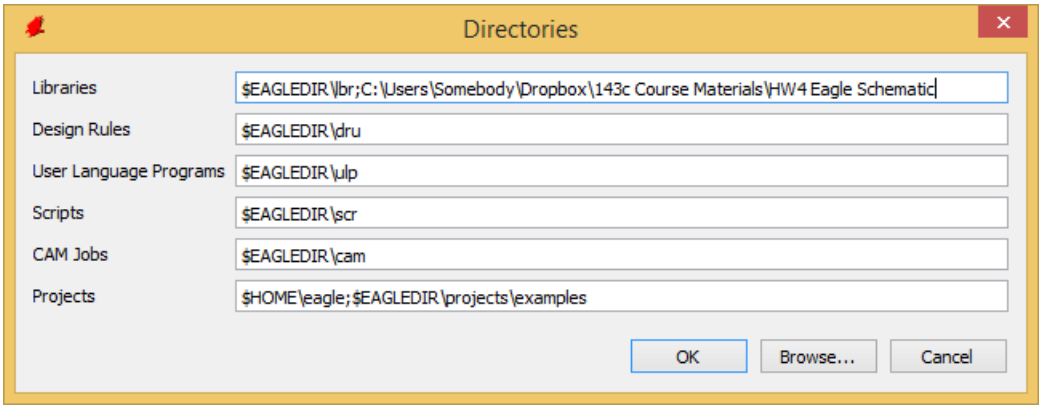
1. 4x7-segment LED Display datasheet
2. 270 Ohm resistor DIP datasheet
3. Logic level Converter datasheet
4. 4 MOSFETs datasheet
5. MAX7221 LED Driver datasheet
6. 30k Ohm Resistor
7. Solderless Prototyping Breadboard
8. 60-Pack of Jumper Cables

4.2 Installing EAGLE and Library

We will use the free version of the EAGLE schematic and PCB layout tool because of it’s wide use in the hobbyist and open-source community which has resulted in a wide array of freely available libraries. Once you’ve downloaded and set up the software, open it up and you should be greeted with the main control panel. I highly recommend this video tutorial by Jeremy Blum to get you started.

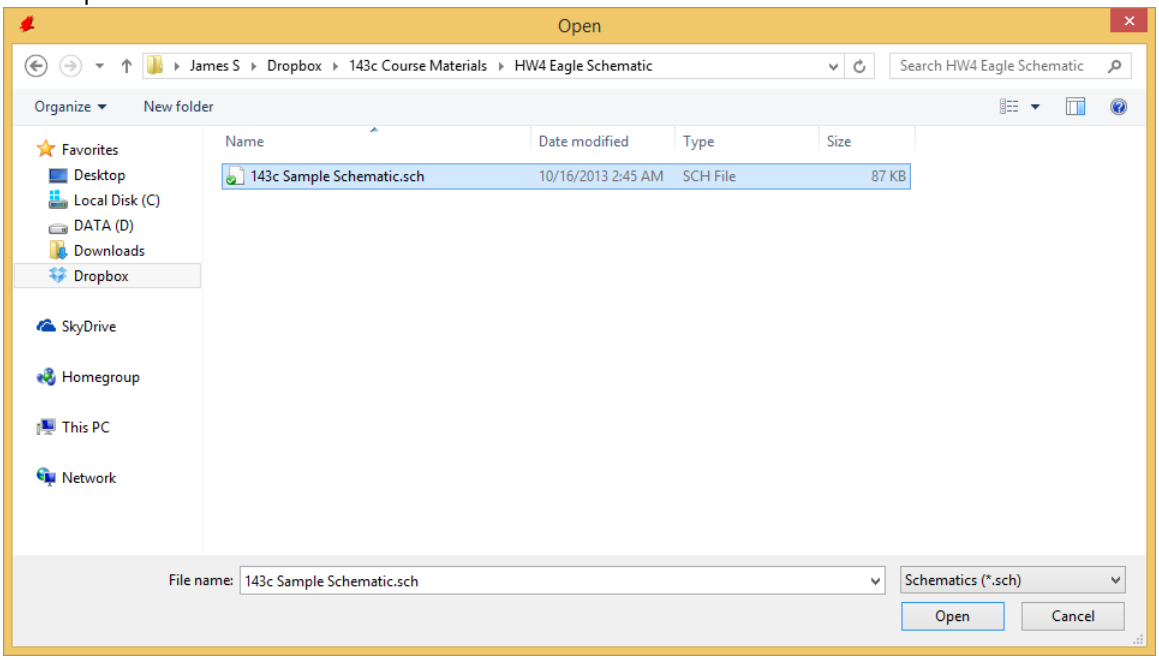


Included with this assignment is an EAGLE library containing the parts included in your LED kit, a blank BeagleBone cape, and a few of my favorite things. Please download BeagleBoneRobotics.lbr to a new directory for this exercise. Then from the EAGLE control panel, click Options -> Directories and add the location where you put the library file. If you just want to include a library without adding its directory to the search path, then in the menu bar at the top click Library -> Use then select the library manually. However this method requires you to "Use" the library each time you open EAGLE.

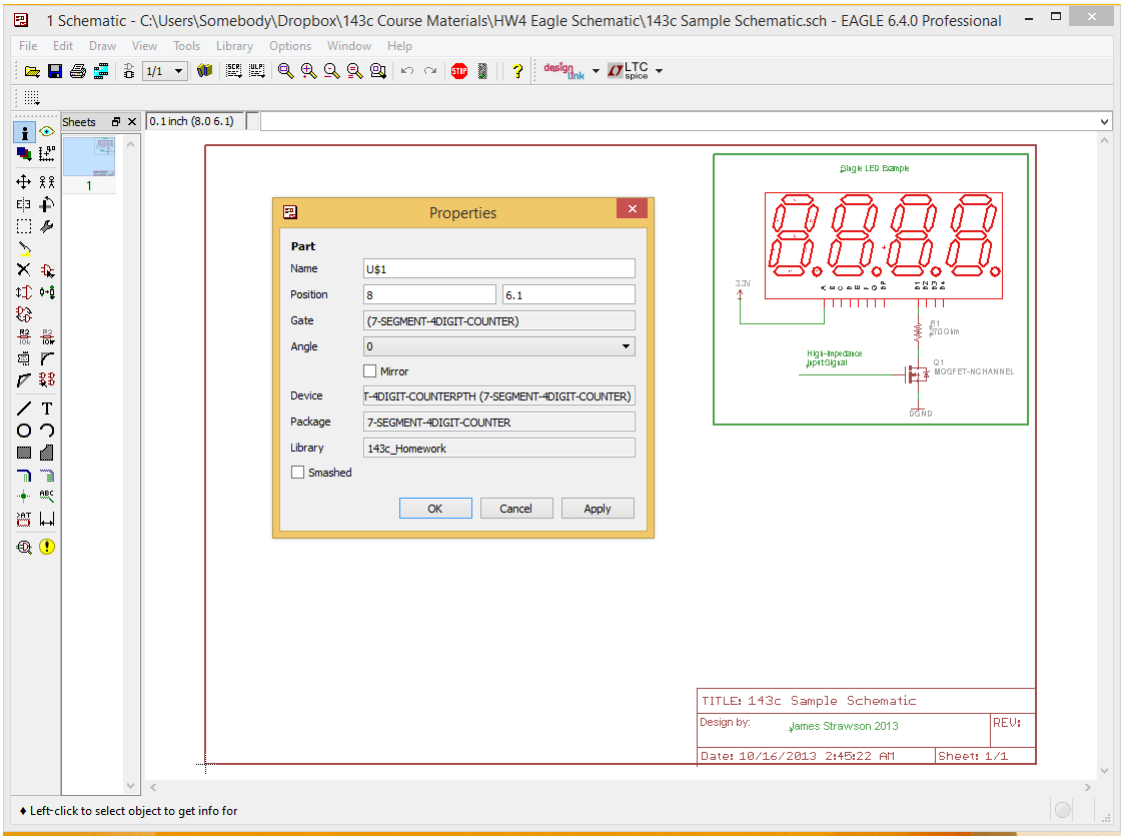


4.3 Modifying the Sample Schematic

Now download the sample schematic SampleSchematic.sch to the same directory and open this file from the control panel.

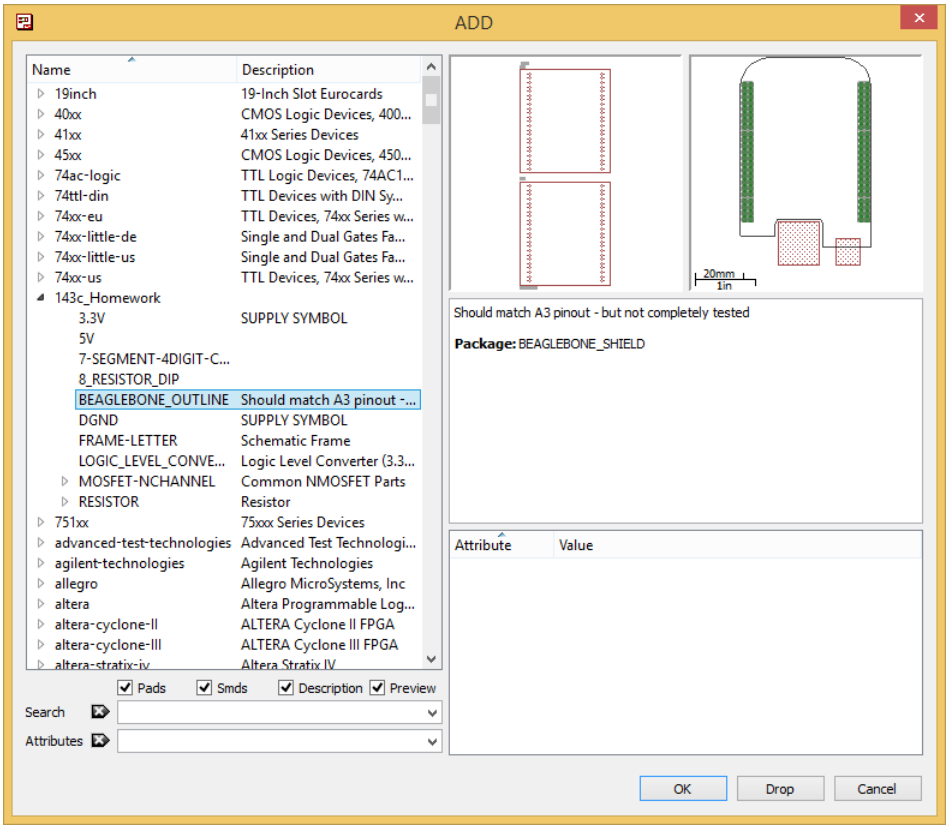


Modify the sample schematic to have your name. To do this, select the info tool on the left and left-click on the green text you wish to change. A properties window for whichever text or component you selected will appear that resembles the one below. Finally save as a new file name.



4.4 Exercise: Driving One LED Digit

With the BeagleBoneRobotics library added, click and add component tool and browse through the list of libraries to find the components included with your LED kit. Add the BeagleBone outline and 8-resistor DIP to your schematic.

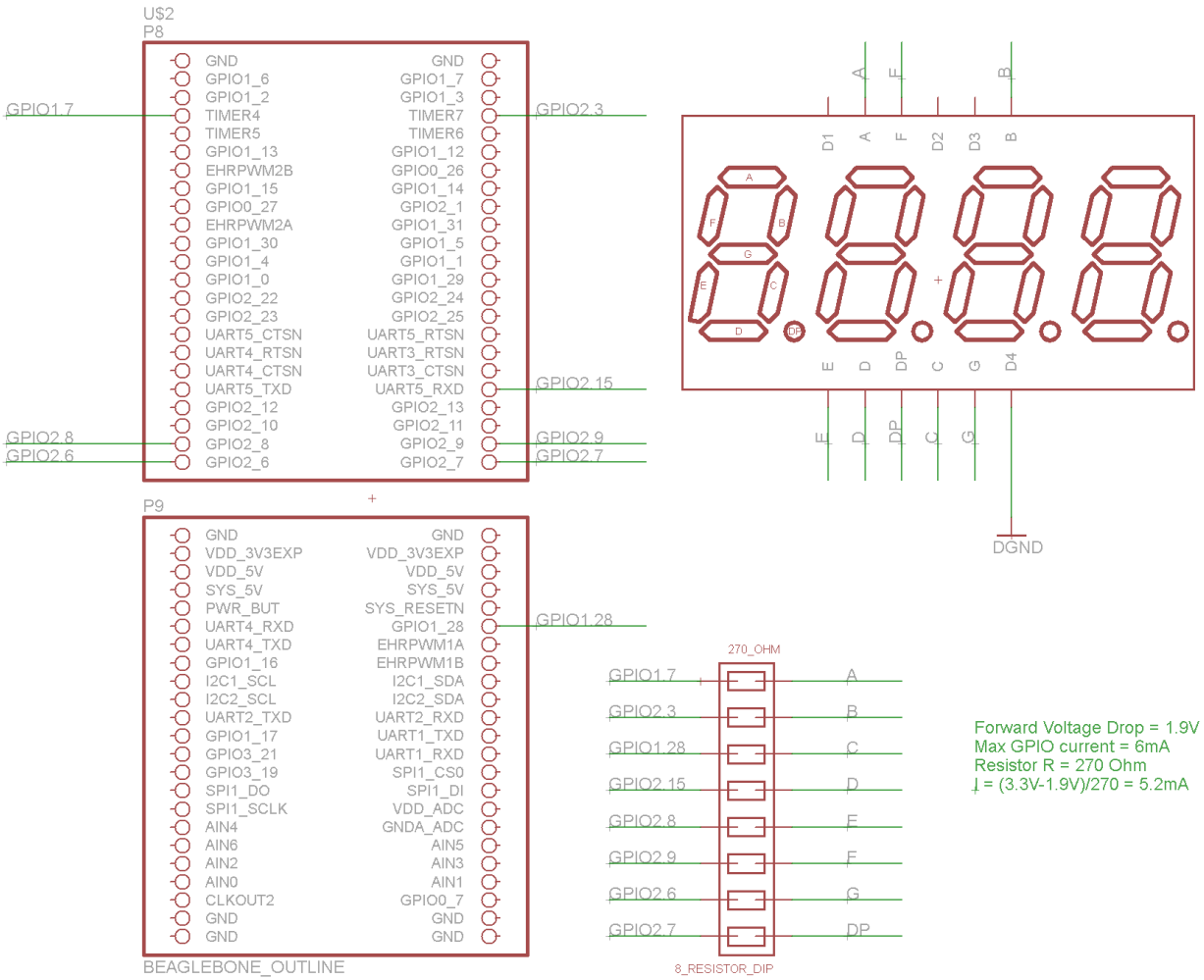


Now check the LED display datasheet to find the forward voltage drop of each LED and the peak continuous current. Write these details as a note in your schematic. Now draw wires such that the first LED digit is controlled by 8 parallel GPIO pins. Add comments, names, and values where appropriate.

We must now choose GPIO channels to control the LEDs. To avoid configuring the pin multiplexer yourself, pick pins which are already configured by the Robotics Cape device tree overlay to be GPIO outputs. This information is in the header pin table here. Avoid input pins such as P8.9, P8.10, and P9.25 as these are configured as inputs with interrupts and are therefore resources reserved by the operating system. The motor H-Bridge direction pins (MDIR) are good choices as you will not be using the motor libraries for this exercise.

I highly suggest using nets instead of just wires to keep your schematic clean. For help on using EAGLE tools such as nets I point you again to Jeremy Blum's video tutorial.

Stop and save. Take a screenshot of this circuit for submission. It should look similar to this, but with different header pin connections.



4.5 Exercise: Multiplexing 4 Digits

See how wiring up all 4 digits would result in a mess if you had to use 8 wires for each digit? Now try designing a circuit to multiplex the 8 GPIO signals across all 4 digits. This is a common cathode display: all 8 LEDs in each digit share a cathode which is accessible on pins D1-D4. The 8 segment pins are connected to every digit. Therefore, if more than 1 cathode is grounded while a segment pin is driven, the same current must be shared between multiple digits and they will dim. This is fine.

Now make a new schematic file with a circuit to drive all 4 digits using the 4 FETS as current sinks connected to ground. Don't forget to refer to the datasheets and take a screen capture of your schematic.

4.6 Exercise: Wiring an IC with SPI

Finally, we are going to replace all 12 signal wires from the beaglebone with just 3 signal wires using a Serial Peripheral Interface. Create a new symbol in the BeagleBone Robotics library titled "MAX7221" and draw a complete symbol using the datasheet as a reference. If you would like, you can create a package and device too, but this is not necessary as you are only making a schematic for this assignment. Insert that symbol into a new schematic and design a circuit to control the MAX7221 and display with a Beaglebone over SPI.

In your schematic, connect the MAX7221 to the BBB using the SPI1 CLK and MOSI pins. Since the BBB operates on a 3.3V logic level and the LED driver runs on a 5V logic level, use the two TX channels on the Sparkfun logic level converter to connect the Clock and MOSI lines. As with many 5v devices, 3.3v is just enough to register a logic level HIGH. Since the slave select line is less timing-sensitive than the serial data lines, we can connect directly from the BeagleBone to the MAX7221. Consult the header pin table again to find the pins used for SPI.

To give 5v to the display driver and high voltage side of the logic level converter, use pin P9.7. To provide 3.3V to the low volt side of the logic level converter, use pin P9.3.

When you are confident in your schematics, save and screen capture the circuit for submission. We will move on to controlling the display with these three circuits with userspace programs next chapter.

5 GPIO and Controlling Hardware with File IO

When controlling hardware such as GPIO and serial ports on a microprocessor, we have to use software to control the appropriate hardware control registers that exist in the microprocessor. To simplify life, the operating system on your BeagleBone uses drivers to expose these hardware interfaces as easy to use files in the file system. Here we will show how to control various GPIO hardware both from the command line and with software written in C.

5.1 Exporting GPIO Pins with the Command Line

There are three GPIO subsystems in the AM335x processor in your BeagleBone black, and they are all controlled with one GPIO driver. First let's pick a pin to play with. Opening the Robotics Cape header pin table again, you will find that the green led is connected to header pin 8. This corresponds to gpio2[3] which means gpio subsystem number 2 and channel 3 on that subsystem. This information can be found in the header pin table. The GPIO driver designates a single number to that pin for simplicity. Since there are 32 pins per GPIO subsystem and 4 subsystems (0,1,2,3), we multiply the subsystem number by 32 and add the channel number. For this pin we have 2*32 + 3 = 67.

Now we can navigate at the command line to the gpio driver directory and export the pin to enable it.

```
root@beaglebone:~# cd /sys/class/gpio
root@beaglebone:/sys/class/gpio# echo 67 > export
```

Now there is a /sys/class/gpio/gpio67 directory and we can see all of the files to control that pin. You may get the error "-bash: echo: write error: Device or resource busy" if the pin is in use by the operating system or another program such as one of the Robotics Cape example programs.

```
root@beaglebone:/sys/class/gpio# cd gpio67
root@beaglebone:/sys/class/gpio/gpio67# ls
active_low  direction  edge  power  subsystem  uevent  value
```

Now lets configure it for output and turn on the pin. If you have the robotics cape installed or connect an LED with a current limiting resistor to this pin, it should turn on.

```
root@beaglebone:/sys/class/gpio/gpio67# echo out > direction
root@beaglebone:/sys/class/gpio/gpio67# echo 1 > value
```

If you wish to use the GPIO pin as an input to read the state of a digital signal connected to that pin, you can set the direction to 'in' and then read the value file instead of writing to the value file.

```
root@beaglebone:/sys/class/gpio/gpio67# echo in > direction
root@beaglebone:/sys/class/gpio/gpio67# cat value
1
```

5.2 Controlling GPIO with C code

The cat and echo commands used before to control the hardware driver from the command line can be translated directly to file read and write commands in C code. To try this out, copy the following program into a new file on the your BeagleBone.

```
// gpio_file_io.c
// illuminates an LED connected to a gpio pin with only file IO
// use expansion header P8, pin8
// GPIO2_3 designated as gpio 67

#include <stdio.h>
#include <stddef.h>
#include <time.h>

#define PIN 67
int main (){

    // file handles
    FILE *ofp_export;
    FILE *ofp_gpio67_value;
    FILE *ofp_gpio67_direction;

    // export gpio pin for use
    ofp_export = fopen("/sys/class/gpio/export", "w");
    if(ofp_export == NULL){
        printf("Unable to open export.\n");
        return -1;
    }
    fseek(ofp_export, 0, SEEK_SET); // seek to beginning of file
    fprintf(ofp_export, "%d", PIN); // write the pin number to export
    fflush(ofp_export);             // finish writing file

    // configure gpio for writing
    ofp_gpio67_direction = fopen("/sys/class/gpio/gpio67/direction", "w");
    if(ofp_gpio67_direction==NULL){
        printf("Unable to open gpio67_direction.\n");
        return -1;
    }
    fseek(ofp_gpio67_direction, 0, SEEK_SET);   // seek to beginning of file
    fprintf(ofp_gpio67_direction, "out");       // configure as output pin
    fflush(ofp_gpio67_direction);               // write file

    // Open file pointer for writing the value
    ofp_gpio67_value = fopen("/sys/class/gpio/gpio67/value", "w");
    if(ofp_gpio67_value == NULL){
        printf("Unable to open gpio67_value.\n");
        return -1;
    }
    fseek(ofp_gpio67_value, 0, SEEK_SET);

    // start blinking loop
    printf("blinking LED\n");
```

```
    int i = 0;
    while(i<10){
        // turn pin on
        fprintf(ofp_gpio67_value, "%d", 1);
        fflush(ofp_gpio67_value);
        printf("ON\n");
        sleep(1);

        // turn pin off
        fprintf(ofp_gpio67_value, "%d", 0);
        fflush(ofp_gpio67_value);
        printf("OFF\n");
        i++; // increment counter
        sleep(1);
    }

    //close all files
    fclose(ofp_export);
    fclose(ofp_gpio67_direction);
    fclose(ofp_gpio67_value);
    return 1;
}
```

To compile, you can just use GCC as follows.

```
root@beaglebone:~# gcc gpio_file_io.c -o gpio
root@beaglebone:~# ./gpio
blinking LED
ON
OFF
...
```

To clean up this code, the Robotics Cape library includes simple GPIO commands for exporting, reading, and writing. The equivalent code is as follows.

```
// gpio_library_example.c
// illuminates an LED connected to a gpio pin with gpio library functions
// use expansion header P8, pin8
// GPIO2_3 designated as gpio 67

#include <robotics_cape.h>

#define PIN 67

int main (void){
    // export gpio pin for use
    if(gpio_export(PIN)){
        printf("Unable to open export.\n");
        return -1;
    }
    // set pin for output
    if(gpio_set_dir(PIN, OUTPUT_PIN)){
```

```
        printf("Unable to open gpio67_direction.\n");
        return -1;
    }

    // start blinking loop
    printf("blinking LED\n");

    int i = 0;
    while(i<10){
        // turn pin on
        gpio_set_value(PIN, 1);
        printf("ON\n");
        sleep(1);

        // turn pin off
        gpio_set_value(PIN, 0);
        printf("OFF\n");
        i++; // increment counter
        sleep(1);
    }

    return 1;
}
```

To compile this code, you need to have the Robotics Cape package installed and tell the GCC linker to include the robotics cape library as follows.

```
root@beaglebone:~# gcc gpio_library_example.c -lrobotics_cape -o gpio
root@beaglebone:~# ./gpio
blinking LED
ON
OFF
...
```

5.3 Exercise: Drive a Digit

Now wire up the single digit schematic from exercise 4.4. Using the gpio functions in the Robotics Cape library, write a program to make a single digit sequence from 0-9 and repeat again. Take a picture of one number being displayed to include with your code submission.

5.4 Exercise: Multiplex 4 Digits

Now add the 4 mosfets to your breadboard to complete your second circuit from exercise 4.5. Now you can display multiple digits by individually turning on the 4 mosfets. This is a process called multiplexing. By swapping between which digit is turned on very quickly, we can make it look like all 4 segments are displaying their own number, even though only one is turned on at a time. I suggest writing to the GPIO pins connected to the 8 anodes of the digit you wish to display, then turning on the corresponding mofset via GPIO to let current flow from the common cathode of that digit. Then wait for roughly 5ms before writing to the next

digit and rotate between the 4 digits every 20ms. This is fast enough that your eyes should not perceive the flashing.

Write a second program which displays a timer with a resolution of one tenth of a second. Take a picture with all 4 digits displaying something with a decimal point indicating the tenths digit.

6 Serial Peripheral Interface

6.1 SPI device tree overlay

To enable the SPI driver and multiplex the SPI pins correctly, you must first make sure the appropriate lines are included in a device tree overlay and loaded into a slot with the cape manager. This is all done for you in the Robotics Cape installer. However, if you are curious to see what needs to be done, here is the device tree overlay provided by the beagleboard open source project to enable SPI1. More device tree overlays can be found here.

```
/*
 * Copyright (C) 2013 CircuitCo
 *
 * Virtual cape for SPI1 on connector pins P9.29 P9.31 P9.30 P9.28
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */
/dts-v1/;
/plugin/;

/ {
    compatible = "ti,beaglebone", "ti,beaglebone-black";

    /* identification */
    part-number = "BB-SPI1";
    version = "00A0";

    /* state the resources this cape uses */
    exclusive-use =
        /* the pin header uses */
        "P9.31",    /* spi1_sclk */
        "P9.29",    /* spi1_d0 */
        "P9.30",    /* spi1_d1 */
        "P9.28",    /* spi1_cs0 */
        // "P9.42", /* spi1_cs1 */
        /* the hardware ip uses */
        "spi1";

    fragment@0 {
        target = <&am33xx_pinmux>;
        __overlay__ {
            /* default state has all gpios released and mode set to uart1 */
            bb_spi1_pins: pinmux_bb_spi1_pins {
                pinctrl-single,pins = <
                    0x190 0x33  /* mcasp0_aclkx.spi1_sclk, INPUT_PULLUP | MODE3 */
```

```
                    0x194 0x33  /* mcasp0_fsx.spi1_d0, INPUT_PULLUP | MODE3 */
                    0x198 0x13  /* mcasp0_axr0.spi1_d1, OUTPUT_PULLUP | MODE3 */
                    0x19c 0x13  /* mcasp0_ahclkr.spi1_cs0, OUTPUT_PULLUP | MODE3 */
                    // 0x164 0x12   /* eCAP0_in_PWM0_out.spi1_cs1 OUTPUT_PULLUP | ↩
                        MODE2 */
                >;
            };
        };
    };

    fragment@1 {
        target = <&spi1>;   /* spi1 is numbered correctly */
        __overlay__ {
            status = "okay";
            pinctrl-names = "default";
            pinctrl-0 = <&bb_spi1_pins>;

            #address-cells = <1>;
            #size-cells = <0>;

            channel@0 {
                #address-cells = <1>;
                #size-cells = <0>;

                compatible = "spidev";

                reg = <0>;
                spi-max-frequency = <16000000>;
                spi-cpha;
            };


            channel@1 {
                #address-cells = <1>;
                #size-cells = <0>;

                compatible = "spidev";

                reg = <1>;
                spi-max-frequency = <16000000>;
            };
        };
    };
};
```

Note here that the SPI bus frequency is set to 16Mhz in this device tree overlay. This is likey too fast for many devices, therefore the equivalent lines in the Robotics Cape overlay set the bus frequency to 1Mhz to

improve compatibility and reduce bus errors over long or noisy signal wires.

6.2 Sending through SPI

Sending packets out of a serial port can be done similarly to controlling GPIO pins. It is possible to send SPI packets by writing to the /dev/spidev1.0 device through the file system. Like with writing to GPIO pins, this can be done with the echo command line program, or in C code with fprintf() or write(). However, to have further control of the phase an speed of the serial clock, we can use the ioctl.h linux library and spidev.h to control the BeagleBone's SPI driver.

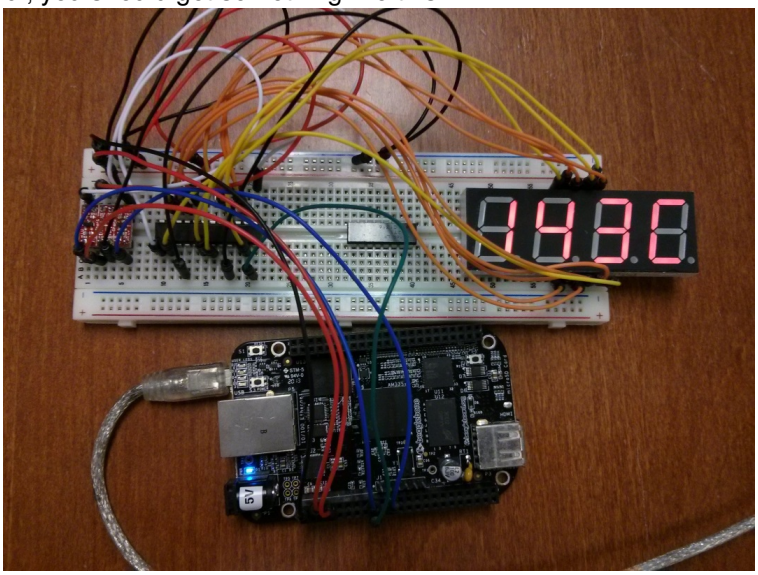
```
// test_max7221.c
// 2014 James Strawson
// Sample code to print "143C" to a display using MAX7221 IC
// Requires SPI1 to be set up in the device tree
// And the Robotics Cape library installed

// Connections
// Two signals through Logic Level Converter
// Clock  SPI1_SCK -> P9_31
// MOSI   SP1_MOSI   -> P9_30

// CS Chip select aka SS slave select
// directly from beaglebone to MAX7221
// SP1_SS1 -> P9_28  gpio3.17 gpio113

#include <robotics_cape.h>
#include <linux/spi/spidev.h>
#include <sys/ioctl.h>

#define ARRAY_SIZE(array) sizeof(array)/sizeof(array[0])
#define SS_PIN 113  //gpio number of slave select

int fd; // file pointer to SPI device
struct spi_ioc_transfer xfer[1]; // io transfer struct
unsigned char wr_buf[2], rd_buf[1]; //IO buffers, only write used here
int status; // return status of spi transfer call

int write_max7221(unsigned char byte0, unsigned char byte1){
    memset(xfer, 0, sizeof xfer);   // clean out buffers
    memset(wr_buf, 0, sizeof wr_buf);
    wr_buf[0] = byte0;
    wr_buf[1] = byte1;
    xfer[0].tx_buf = (unsigned long) wr_buf;
    xfer[0].len = 2;
    xfer[0].speed_hz = 2000000; //2Mhz
    xfer[0].bits_per_word = 8;
    // select slave and write bytes
    gpio_set_value(SS_PIN, LOW); //select slave
    status = ioctl(fd, SPI_IOC_MESSAGE(1), xfer);
    if (status < 0) {
        printf("SPI_IOC_MESSAGE_FAILED");
        return -1;
    }
```

```
    gpio_set_value(SS_PIN, HIGH);
    usleep(500); //allow IC to clear input buffer
    return 0;
}

int main(){
    //open SPI device
    fd = open("/dev/spidev1.0", O_RDWR);
    if (fd<=0) {
        printf("Device not found\n");
        exit(1);
    }
    // set write mode of SPI
    int spi_mode = SPI_MODE_3;
    status = ioctl(fd, SPI_IOC_WR_MODE, &spi_mode);
    if (status == -1){
        printf("can't set spi mode");
        return -1;
    }
    // export slave select pin as output
    if(gpio_export(SS_PIN)){
        printf("failed to export slave select pin\n");
        return -1;
    }
    gpio_set_dir(SS_PIN, OUTPUT_PIN);

    // start with slave deselected briefly
    gpio_set_value(SS_PIN, HIGH);
    usleep(500000);

    // wake unit up from sleep
    if(write_max7221(0x0C, 0x01) < 0){
        printf("failed to write, check SPI is enabled in device tree overlay\n");
    }
    // turn on first 4 digits only
    write_max7221(0x0B, 0x03);

    // set decoding font B for the first 3 digits
    write_max7221(0x09, 0x07);

    // write "143" to first 3 digits using font B decoding
    write_max7221(0x01, 0x01);
    write_max7221(0x02, 0x04);
    write_max7221(0x03, 0x03);

    // write "C" to the last digit by manually selecting segments
    // each 1 is a lit segment
    write_max7221(0x04, 0b001001110);

    printf("Packets Written\n");
    close(fd);
    return 0;
}
```

Since this uses the Robotics Cape shared library, you must also include the robotics cape flag to tell the linker to look for the installed shared library.

```
root@beaglebone:~# gcc max7221_test.c -lrobotics_cape -o spi
root@beaglebone:~# ./spi
Packets Written
```

If all was successful, you should get something like this.



Note that the MAX7221 integrated circuit is performing the same multiplexing routine that you programmed in exercise 5.4. However, your interface to the display is now much simpler and requires only a serial port instead of a bundle of GPIO signal wires.

6.3 Exercise: SPI Clock

Now rewrite your timer from exercise 5.4 to use the LED driver IC instead of GPIO pins. Take a picture of the working display to submit with your source code.

6.4 Reading through SPI

Reading registers from an IC over SPI is a little more complicated to program due to the strict timing that's necessary to read the incomming data from the serial buffer. The common ioctl() function in C will take care of managing the serial buffer for you and allows you to configure the SPI driver to use a specific SPI mode and speed as well. Here is a sample from the test adns9800 example in the Robotics Cape installer package. The ADNS9800 is an optical mouse sensor which can be hooked up to your beaglebone and read directly through SPI.

```
int adns_mode = SPI_MODE_3;
int adns_speed = 2000000; // 2 mhz max
int adns_bits = 8;

unsigned char adns_read_reg(int fd, unsigned char reg_addr){
    struct spi_ioc_transfer xfer[1];
```

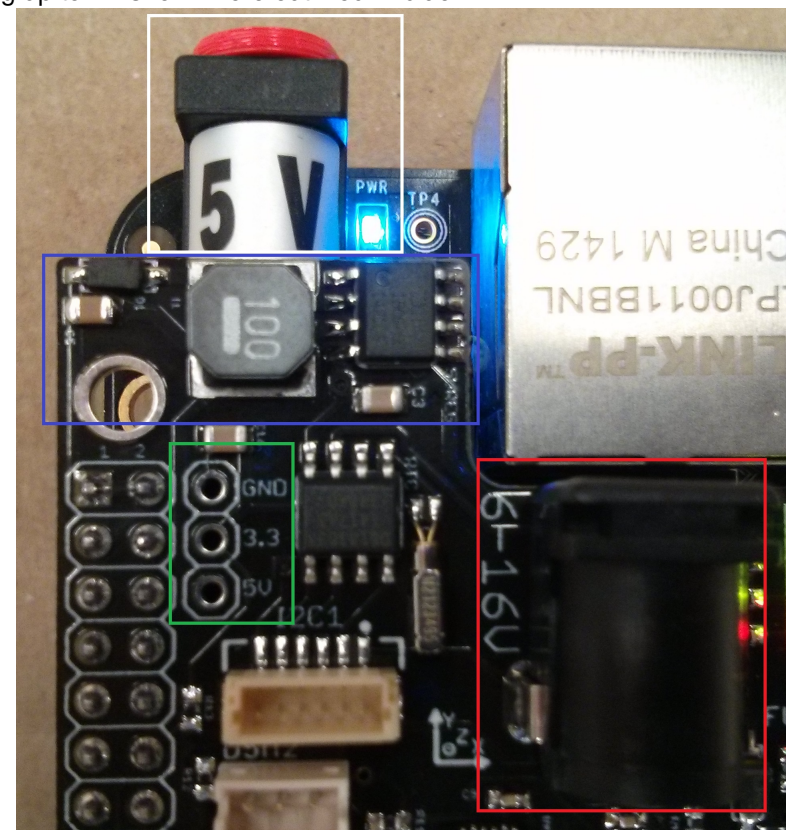
```
    unsigned char wr_buf[1], rd_buf[1];
    int status;

    memset(xfer, 0, sizeof xfer);
    memset(wr_buf, 0, sizeof wr_buf);

    // send adress of the register, with MSBit = 0 to indicate it's a read
    wr_buf[0] = reg_addr & 0x7f;
    select_spi0_slave(0);
    xfer[0].tx_buf = (unsigned long) wr_buf;
    xfer[0].len = 1;
    xfer[0].speed_hz = adns_speed;
    xfer[0].bits_per_word = adns_bits;
    status = ioctl(fd, SPI_IOC_MESSAGE(1), xfer);
    usleep(100);

    //read response
    xfer[0].tx_buf = 0;
    xfer[0].rx_buf = (unsigned long) rd_buf;
    xfer[0].len = 1;
    xfer[0].speed_hz = adns_speed;
    xfer[0].bits_per_word = adns_bits;
    status = ioctl(fd, SPI_IOC_MESSAGE(1), xfer);
    usleep(1);
    deselect_spi0_slave(0);
    usleep(1); // wait for chip to deselect

    if (status < 0) {
        printf("SPI_IOC_MESSAGE_FAILED");
        return -1;
    }
    return rd_buf[0];
}
```

7 Robotics Power Management

While your beaglebone can be conveniently powered over a USB port or with a 5V power supply, we need a more sophisticated power management system if we are to let our robots roam wirelessly. In this chapter we will review the various voltage regulators and battery management techniques used in robotics and on your Robotics Cape.

7.1 BeagleBone Power Restrictions

Your BeagleBone contains a Power Management Integrated Circuit (PMIC) from Texas Instruments which contains 1.8V and 3.3V regulators along with the charging circuit for a single cell lithium battery. This PMIC makes the decision to draw power from either the USB port or the 5VDC input which can be accessed either by the black barrel connector on the BBB or through the VDD_5V rail accesible on header pins P9_5 and P9_6. Unfortunately, multiple cell lithium battery packs used in robots have unregulated terminal voltages much higher than 5V. For this reason the Robotics Cape includes a high efficiency switching voltage regulator capable of supplying up to 2A shown here outlined in blue.



Also available near the 5V regulator are 3 test points which provide access to both the Robotics Cape's 5V line and a regulated 3.3V rail from the BeagleBone header pins. These three vias can be populated with 0.1" header pins if you desire to use them with your robotics project. Note that while the 5V regulator is rated to supply 2A, the BeagleBone will draw roughly 500mA under load in addition to the power used by any USB device plugged into the BeagleBone's USB host port. According to page 38 of the BeagleBone System Reference Manual, the allowable current for this 3.3V rail is 500mA.

Note in the above picture that the 5V DC input jack outlined in white has a red plug installed to prevent accidental damage to the BBB since it is the same sized connector as the 6-16V DC jack on the Robotics

Cape. You are provided with both this safety plug and a more common 12V DC power supply to power and charge your Robot with.

The 5V regulator on the Robotics Cape will draw power from either the 6-16V DC input outlined in red or a 2-Cell lithium battery plugged into the JST XH balance connector. More on batteries in the next section.

7.2 Lithium Batteries and Protection

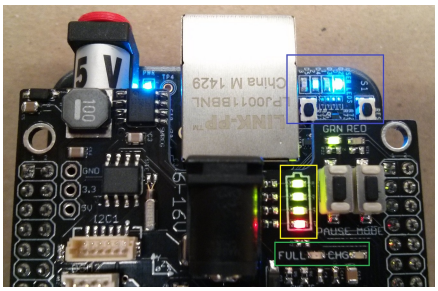
Prismatic Lithium Polymer battery packs have become the energy storage medium of choice in small robots and remotely controlled vehicles in the hobby space due to their high energy density. To reduce cost and weight, lithium polymer battery packs (LiPos) sold in the hobby industry do not typically have integrated protection circuitry like you would find on a more expensive laptop battery.

If a lithium cell drops below roughly 2.5V, it begins the process of under-voltage degradation. Very quickly, the Lithium Cobalt Oxide or Lithium Manganese Oxide will break down and release Oxygen. This results in the very common physical symptom of puffy batteries and unfortunately permanent capacity loss. The normal charging cutoff point is 4.2V for Lithium based cells. Charging above 4.3V can result in overheating and further capacity loss. While fires are not likely with small batteries under 2000mAh due to lack of total energy, be aware that Lithium fires can occur if packs are physically damaged, shorted, or overcharged.

For this reason, unprotected Lithium battery packs must be treated with care and are normally removed from a robot to be charged on a dedicated Lithium charger with a balance connector that gives the charger the ability to read and monitor individual cell voltages within a pack. We include 2-Cell under and over voltage protection and a 1A charging circuit on the Robotics Cape so all you need to charge and use you robot is a common DC power supply such as the one provided.

7.3 Robotics Cape Battery Management

Your Robotics Cape is designed to charge and protect a 2-Cell Lithium battery pack connected to the cape with a standard 3-pin JST-XH connector. This is how you will power your BeagleBone wirelessly as well as powering DC motors and servos directly from the cape. Once you plug in a 2-cell lithium pack you must arm the battery protection circuit by plugging in a DC power supply to the 6-16V input jack. After a few seconds charging will begin and the protection circuit will re-arm, enabling current to be drawn from the battery to keep your BeagleBone powered.



To insure the charging circuit is working when a power supply is connected, examine the charging circuit LEDs outlined in green. While charging, the CHG LED will illuminate yellow and when the pack is full the FULL LED will illuminate green.

While charging or discharging, you can monitor the state of the battery with the 4 battery indicator LEDs outlined in yellow. All 4 LEDs in the battery indicator will flash to indicate critical battery voltage shortly before the battery protection circuit prevents over discharge protection of the pack.

We suggest disconnecting the 2-Cell battery if you wish to store your robot unused for longer than a week to prevent unnecessary discharge. However, to avoid needing to rearm the battery protection circuit, you may leave the battery connected and simply power off your BeagleBone with the power button outlined in blue. A momentary press of the power button next to the ethernet port will shut the Beaglebone down in roughly 4 seconds depending on what is running. It will also boot up in roughly 12 seconds with the same button. Instead of power cycling your BeagleBone in the event of a software crash, you can use the neighboring reset button to force restart the BeagleBone.

8 Robotics Cape Library

Your Robotics Cape is provided with an open source installer package that sets up your BeagleBone with libraries to use cape features along with a device tree overlay, example programs, and a startup script to manage what programs you want your robot to run automatically. In this chapter we will outline the functions and features available to you.

8.1 Bare Minimum

In the examples directory of the BeagleBone installer you will find over a dozen programs demonstrating Robotics Cape functionality. You may notice that they all have a few things in common. Firstly, they all contain a C file with the same name as the program directory. This will also be the same name as the compiled program and is not a requirement but is considered good practice. Each example also has its own Makefile. This is a configuration file that allows you to compile your programs with the make utility instead of typing out long lists of linker options with GCC. Finally, you will notice that they all have README files with a description of the program and special instructions for using it. Let's examine the bare_minimum example to see what needs to be included in each of your robotics projects.

```
// Bare Minimum Skeleton for Robotics Cape Project
// James Strawson - 2014

#include <robotics_cape.h>

int main(){
    initialize_cape();

    printf("\nHello BeagleBone\n");

    //Keep Running until program state changes
    while(get_state() != EXITING){
        usleep(1000000);
    }

    cleanup_cape();
    return 0;
}
```

First note the header contains the name of the program, the author, and date. This is good practice to keep track of your programs over time or when working in a group. Next the Robotics Cape library header is included. This header file contains most common Linux libraries and all function definitions in the Robotics Cape library. C programming gurus will note that this include statement is in angular brackets and not in quotation marks. This is because the Robotics Cape installer placed the Robotics Cape header file in the /usr/include/ and the library shared object file in the /usr/lib directory so they can be accessed wherever you place your project directory.

Next we will be sure to put the initialize_cape() function at the beginning of your main function. This performs initialization functions such as exporting GPIO pins, configuring interrupts, initializing the PRU, and stopping any other Robotics program which may be running to prevent conflicting use of resources. This means that you can start any Robotics program without going through the hassle of opening the top task

manager and manually killing background programs.

Next we print something to the console to indicate the program is alive and working and then enter the main while loop. This loop is where you can run any ongoing routines. Be careful to give use of the processor back to the operating system with the usleep() function, typically at the end of the loop. The standard sleep() function only provides a resolution of one second and is not thread safe, so we used the usleep() function.

Note that the loop exists when the get_state() function call returns EXITING. EXITING is part of an enumerated type defined in robotics_cape.h which exists to manage the state of your program. When you press Ctrl-C at the console to quit your program, normally the program is stopped immediately by the operating system. When you call the initialize_cape() function, a signal handler is set up to intercept the SIGINT (Ctrl-C) signal and changes the state variable to EXITING. This allows you to cleanly exit all of your functions and threads if you keep them running in similar while loops. You can also call the function set_state(EXITING) yourself to tell other threads to exit cleanly.

Finally, we see the last statement in main() before returning is the cleanup_cape() function. This will disable the motors, PRU, and remove the lockfile indicating the program shut down cleanly. For more details see the provided library source code robotics_cape.c.

8.2 Makefiles and Creating a New Project

To start a new project, I suggest copying the bare_minimum directory and its contents to your root directory and renaming it to match your new project name.

```
root@beaglebone:~# cp -r /root/Robotics_Cape_RevC_Installer/examples/bare_minimum /↩
    root/new_project
root@beaglebone:~# cd /root/new_project
root@beaglebone:~/new_project/# mv bare_minimum.c new_project.c
```

Now we must rename our source code and the compiled output file names in the Makefile to match your project.

```
root@beaglebone:~/new_project/# nano Makefile

#project name change to match your main c file
TARGET = new_project



TOUCH    := $(shell touch *)
CC      := gcc
LINKER   := gcc -o
CFLAGS  := -c -Wall -g
LFLAGS  := -lm -lrt -lpthread -lrobotics_cape
....
```

Take this opportunity to browse the Makefile for all of the special compiler and linker flags that are used in building a Robotics Cape project. We use the make utility to avoid typing out all of these options ourselves

in a GCC command.

The make utility gives us three primary commands we can use. Before compiling a project, it is optional but advised to call the "make clean" command to delete previously compiled files. Next we call the simple "make" command to build the project and create an executable file in the project directory.

```
root@beaglebone:~/new_project/# make clean
Cleanup Complete
root@beaglebone:~/new_project/# make
root@beaglebone:~/new_project/# ./new_project
Hello BeagleBone!
root@beaglebone:~/new_project/#
```

When you are happy that your program functions as expected, you can install it to the /usr/bin directory so that it can be accessed from any directory.

```
root@beaglebone:~/new_project/# make clean
Cleanup Complete
root@beaglebone:~/new_project/# make install
root@beaglebone:~/new_project/#cd ../
root@beaglebone:~# new_project
Hello BeagleBone!
root@beaglebone:~#
```

Note that most of the example programs are installed by default and can be launched from anywhere for quick testing.

8.3 Buttons and LEDs

Here we will examine two basic but commonly used Robotics Cape functions. Along with the battery indicator and charge LEDs, the cape also includes a green and a red LED for the user to control themselves to indicate robot function and state. There are also two buttons configured with interrupts within the operating system. Below we demonstrate the method by which we define which functions are called when they are pressed or released. The buttons are labeled Pause and Mode but can be used for any purpose.

```c
// Button and LED tester for the Robotics Cape
// Pressing either button makes an LED blink
// Hold the pause button or ctrl-c to exit cleanly
// James Strawson - 2014

#include <robotics_cape.h>

int mode;   // 0, 1, 2  slow medium fast blink rate
int paused; // 0 for running, 1 for paused
int toggle; // toggles between 0&1 for led blink

// Print to the console when buttons are pressed
int on_pause_press(){
```

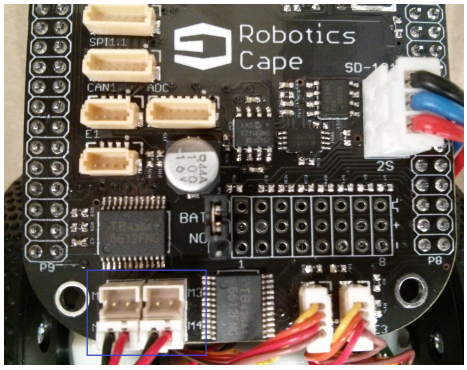
```
    printf("pressed Pause\n");
    if(get_mode_button_state() == 1){
        //both buttons pressed exit cleanly
        set_state(EXITING);
    }
    return 0;
}

int on_mode_press(){
    printf("pressed mode\n");
    if(get_pause_button_state() == 1){
        //both buttons pressed exit cleanly
        set_state(EXITING);
    }
    return 0;
}

// toggle paused state when button released
int on_pause_release(){
    if(paused) paused=0;
    else paused=1;
    return 0;
}

// increment mode
int on_mode_release(){
    if(mode<2)mode++;
    else mode=0;
    return 0;
}

int main(){
    initialize_cape();

    printf("\nPress mode to change blink rate\n");
    printf("hold pause to exit\n");

    //Assign your own functions to be called when events occur
    set_pause_pressed_func(&on_pause_press);
    set_pause_unpressed_func(&on_pause_release);
    set_mode_pressed_func(&on_mode_press);
    set_mode_unpressed_func(&on_mode_release);

    // start in slow mode
    mode = 0;

    //toggle leds till the program state changes
    while(get_state() != EXITING){
        usleep(500000 - (mode*200000));
        if(!paused && toggle){
            setGRN(LOW);
            setRED(HIGH);
            toggle = 0;
        }
        else if(!paused && !toggle){
```

```
            setGRN(HIGH);
            setRED(LOW);
            toggle=1;
        }
    }

    cleanup_cape();
    return 0;
}
```

Note that the way we set the interrupt service routines for the buttons is by passing a pointer to a function in the same way that we can pass pointers to variable locations in memory.

8.4 Motors

Your Robotics Cape has four H-bridges with flyback diodes which draw power from the 2-Cell lithium battery pack. They are capable of supplying 1A continuously provided there is sufficient airflow to keep them from overheating. Check that your motors do not draw more than this at stall with a peak battery voltage of 8.4 volts. Use 2-Pin JST-ZH connectors to quickly connect to the 4 motor output connectors outlined below in blue.



Controlling the H-bridges is done with the set_motor() function which takes in two arguments. First, provide the integer motor channel number from 1 to 4, and a floating point value to -1 to 1 to indicate the direction and duty cycle. The PWM frequency is configured in the Robotics Cape device tree overlay and comes set at 40khz.

You also have control over the motor standby signal which puts the H-bridges into a lower power state when logic level low. You should call enable_motors() before using the set_motors() function. It is also advisable to use disable_motors() when the motors are not in use and as part of a safety shutdown routine triggered by the pause button. Also available is the kill_pwm() command to stop all 4 PWM generators.

```
// test_motors.c
// Moves all 4 motors forward and back
// James Strawson - 2014

#include <robotics_cape.h>

int main(){
    initialize_cape();

    // bring H-bridges of of standby
```

```
    enable_motors();
    setGRN(HIGH);
    setRED(HIGH);
    int i;

    // Drive all motors forward at %30 duty cycle
    for(i=1;i<=4;i++){
        set_motor(i,.3);
    }
    printf("\nAll Motors Forward\n");
    sleep(2);

    // Drive all motors back at %30 duty cycle
    for(i=1;i<=4;i++){
        set_motor(i,-.3);
    }
    printf("All Motors Reverse\n");
    sleep(2);

    kill_pwm();         // set all duty cycles to 0
    disable_motors();   //put H-bridges into standby
    printf("All Motors Off\n\n");
    cleanup_cape();
    return 0;
}
```

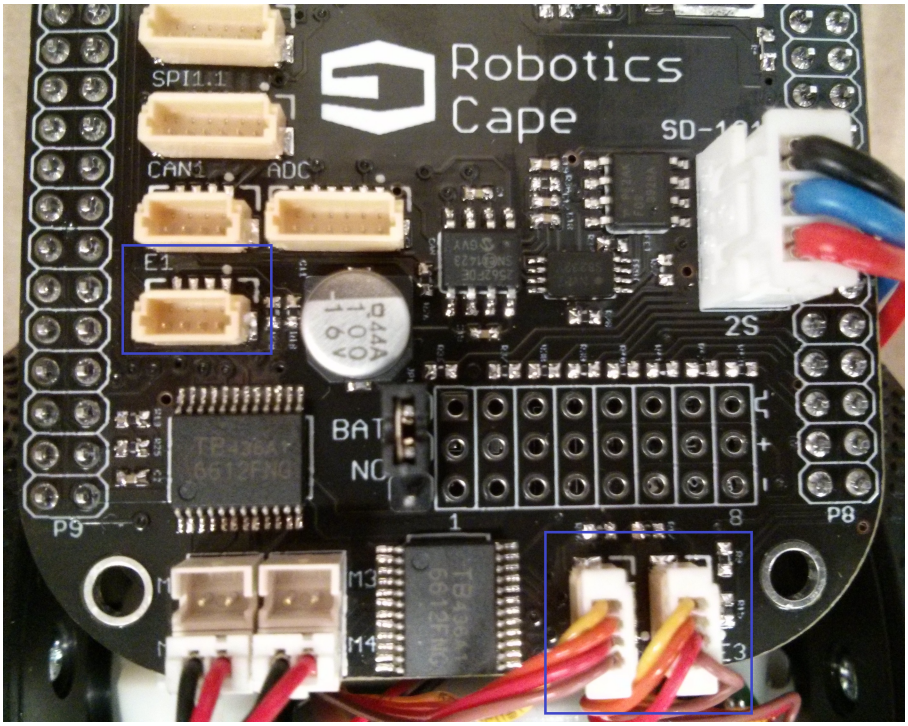
8.5 Encoders

While it is possible to count quadrature encoders in software with interrupt service routines, your BeagleBone has three hardware encoder counters called eQEP modules which are part of the PWM subsystems. To accelerate reading of the eQEP counters, the Robotics Cape library uses the C command mmap to directly access the PWM subsystem registers instead of using a kernel driver. This is one of the reasons you must run as the root user or with root privileges to use the robotics cape library.

The three encoder channels are broken out to three connectors outlined here in blue. Each of these can be used with a 4-pin JST-SH connector like the ones provided. Pin 1 of these connectors is indicated with a dot. Checking the Robotics Cape schematic available here we can see that pin 1 is ground. This is a convention common among the Robotics Cape connectors. The entire pinout is as follows.

1. Ground (brown) 2. 3.3V (Red) 3. Signal A (Orange) 4. Signal B (Yellow)

You will also notice from the schematic that 1k Ohm pullup resistors are installed on the signal pins. This is because most small optical and hall effect encoder modules require strong pullup resistors to operate and are therefore included for convenience.



Reading and setting the encoder counters is straightforward. Here is a sample program to read the current position. This is a pre-installed program and can be called from anywhere for quick testing of your encoder hardware.

```c
// test_encoders.c
// Prints out current encoder ticks
// James Strawson - 2014

#include <robotics_cape.h>

int main(){
    initialize_cape();

    // reset encoder counters to 0
    set_encoder_pos(1, 0);
    set_encoder_pos(2, 0);
    set_encoder_pos(3, 0);

    setGRN(HIGH);
    setRED(HIGH);
    printf("\n\nRaw data for encoders 1,2,3\n");

    while(get_state() != EXITING){
        printf("\r%3li %3li %3li  ", get_encoder_pos(1),get_encoder_pos(2),get_encoder_pos(3)↩
            );
        fflush(stdout);
        usleep(50000);
    }

    cleanup_cape();
    return 0;
}
```

8.6 IMU

Your robotics cape comes with an Invensense MPU-9150 9-axis inertial measurement unit , a popular sensor among hobbyists and roboticists. This sensor contains sets of three orthogonal accelerometers, gyroscopes, and magnetometers which allow us to estimate the robot's orientation in space. Furthermore, the MPU-9150 contains a small microprocessor which Invensense call their Digital Motion Processor or DMP for short. This microprocessor continuously runs digital low and high pass filters on the accelerometer and gyroscope signals respectfully in addition to estimating orientation which can be read directly over an I2C bus and then interpreted as either a Quaternion vector or as Euler angles.

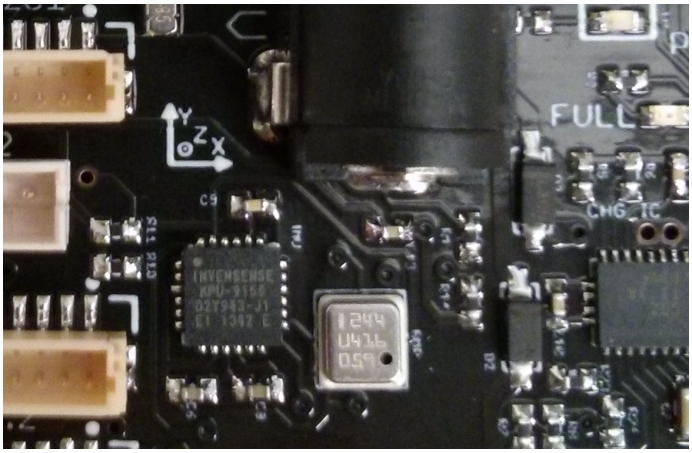
Since this sensor data is commonly used as part of a discrete time feedback controller, we provide in the Robotics Cape library a simple method for configuring the DMP to sample sensors at a constant rate. This offloads the timing of the discrete controller from the operating system to the DMP and allows you to set an interrupt service routine to be called by the DMP as soon as the sensors have been sampled. This ensures consistent timing and minimal sensor latency.

Here is the source code for the built in test_imu function which displays Euler angles in degrees and raw discretized readings from the gyroscope ADC. Note that the DMP is capable of 5-200 hz sample rate.

```
// Sample Code for testing MPU-9150 operation
// Takes in an integer to use as sample rate.
// Valid range: 5-200
//
// James Strawson - 2014

#include <robotics_cape.h>

// This is the fastest rate the DMP will do
#define DEFAULT_SAMPLE_RATE 200

// IMU interrupt service routine
int print_imu_data(){
    mpudata_t mpu; //struct to read IMU data into
    memset(&mpu, 0, sizeof(mpudata_t)); //make sure it's clean before starting
    if (mpu9150_read(&mpu) == 0) {
        printf("\r");

        printf("X: %0.1f Y: %0.1f Z: %0.1f ",
        mpu.fusedEuler[VEC3_X] * RAD_TO_DEGREE,
        mpu.fusedEuler[VEC3_Y] * RAD_TO_DEGREE,
        mpu.fusedEuler[VEC3_Z] * RAD_TO_DEGREE);

        printf("Xg: %05d Yg: %05d Zg: %05d ",
        mpu.rawGyro[VEC3_X],
        mpu.rawGyro[VEC3_Y],
        mpu.rawGyro[VEC3_Z]);

        // printf("Xa: %05d Ya: %05d Za: %05d ",
        // mpu.calibratedAccel[VEC3_X],
        // mpu.calibratedAccel[VEC3_Y],
        // mpu.calibratedAccel[VEC3_Z]);

        // printf("Xm: %03d Ym: %03d Zm: %03d ",
```

```
        // mpu.calibratedMag[VEC3_X],
        // mpu.calibratedMag[VEC3_Y],
        // mpu.calibratedMag[VEC3_Z]);

        // printf("W: %0.2f X: %0.2f Y: %0.2f Z: %0.2f ",
        // mpu.fusedQuat[QUAT_W],
        // mpu.fusedQuat[QUAT_X],
        // mpu.fusedQuat[QUAT_Y],
        // mpu.fusedQuat[QUAT_Z]);

        fflush(stdout);
    }
    return 0;
}

// main function declaration to accept command line arguments
int main(int argc, char *argv[]){
    int sample_rate;

    // uncomment desired equilibrium orientation
    signed char orientation[9] = ORIENTATION_FLAT;
    //signed char orientation[9] = ORIENTATION_UPRIGHT;

    // If the user gave no arguments, use default rate
    if (argc==1){
        sample_rate = DEFAULT_SAMPLE_RATE;
    }
    // If the user gave a valid sample rate argument, use that
    else{
        sample_rate = atoi(argv[1]);
        if((sample_rate>MAX_SAMPLE_RATE)||(sample_rate<MIN_SAMPLE_RATE)){
            printf("sample rate should be between %d and %d\n", MIN_SAMPLE_RATE,↩
                MAX_SAMPLE_RATE);
            return -1;
        }
    }
    // start cape and imu interrupt handler
    initialize_cape();
    initialize_imu(sample_rate, orientation);
    set_imu_interrupt_func(&print_imu_data);

    // now just wait, print_imu_data will run until closed
    while (get_state() != EXITING) {
        usleep(1000000);
    }
    cleanup_cape();
    return 0;
}
```

Note that this will print out angles of roughly 0 degrees when the BeagleBone and cape sit flat on a table. This is because the default orientation matrix (identity) preserves the cartesian coordinate system visible on the Robotics Cape. You may select the upright orientation matrix which reverses the Z and Y axis to provide zero Euler angles when upright with the ethernet port pointed towards the ceiling.



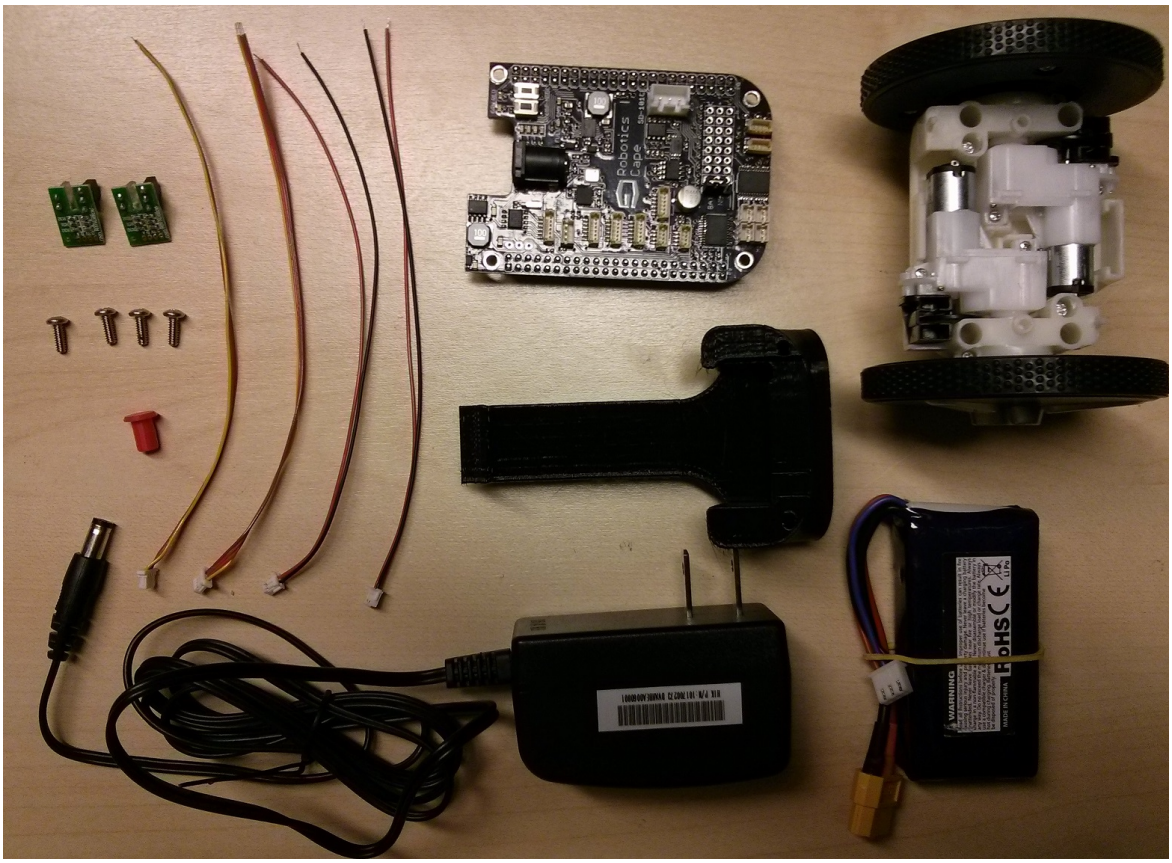
You may notice that the gyroscope readings are not centered around zero. To remedy this, call the installed calibrate_gyro function from the command line. This will sample the gyro for a second and save the offsets to your /root/cape_calibration directory to be used to calibrate the gyro on subsequent calls to the initialize_imu() function.

9 Balancing BeagleMiP

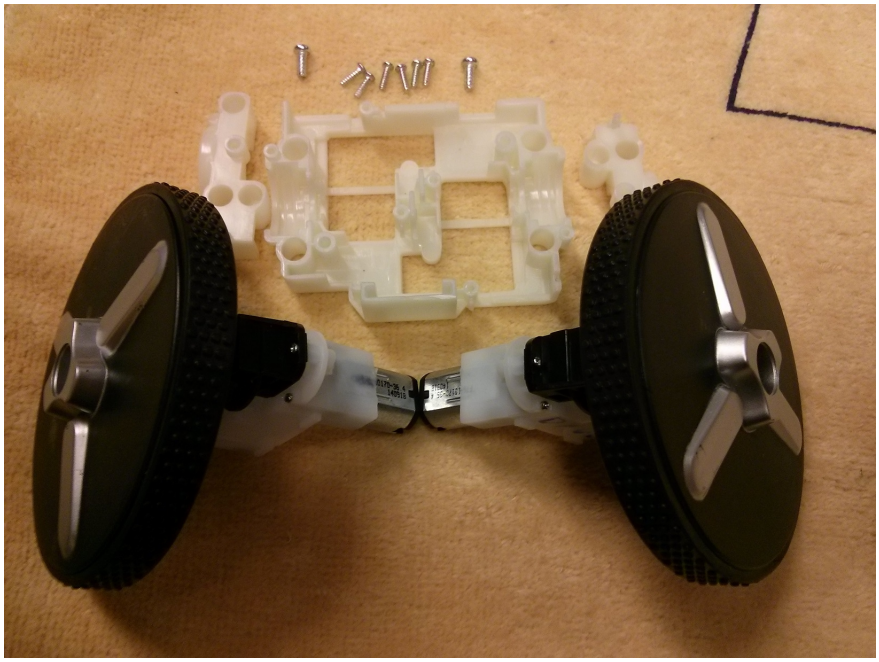
9.1 BeagleMiP Assembly

Included in your BeagleMiP kit are all of the components necessary to make your BeagleBone balance. Open your kit and inspect the following components.

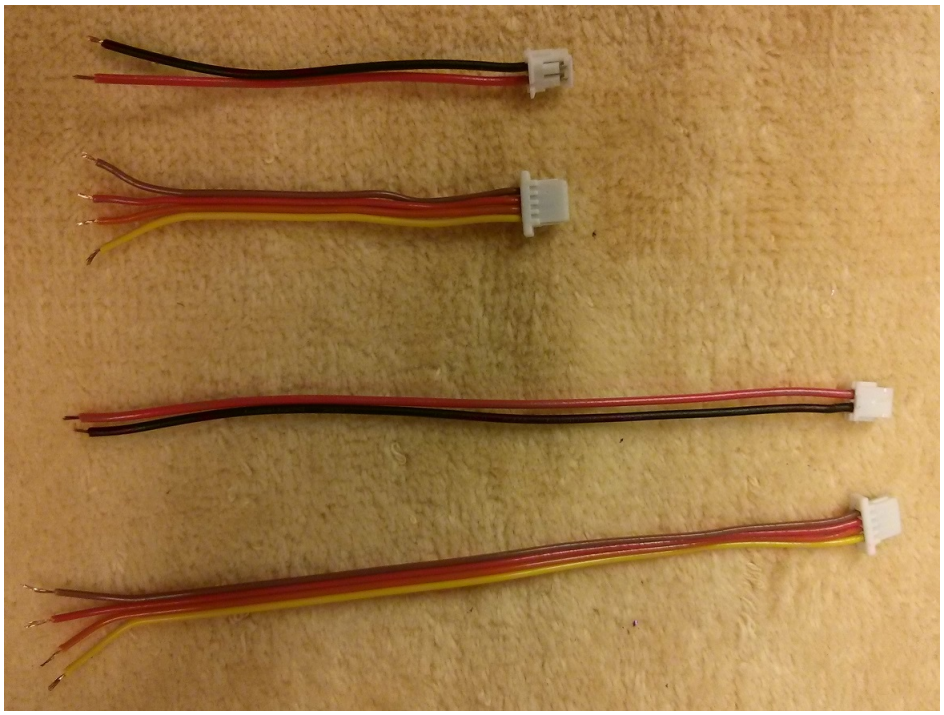
1. Gearbox unit with motors and wheels attached (1)
2. Optical encoder boards (2)
3. Robotics Cape (1)
4. 2-Pin JST ZH Pigtails (2)
5. 4-Pin JST SH Pigtails (2)
6. 2-Cell LiPo Battery (1)
7. 12v 1A DC Power Supply (1)
8. Barrel Jack Plug (1)
9. Chassis (1)
10. 4-40 x 3/8” Screws (4)
11. Screwdriver (1)??????????????????



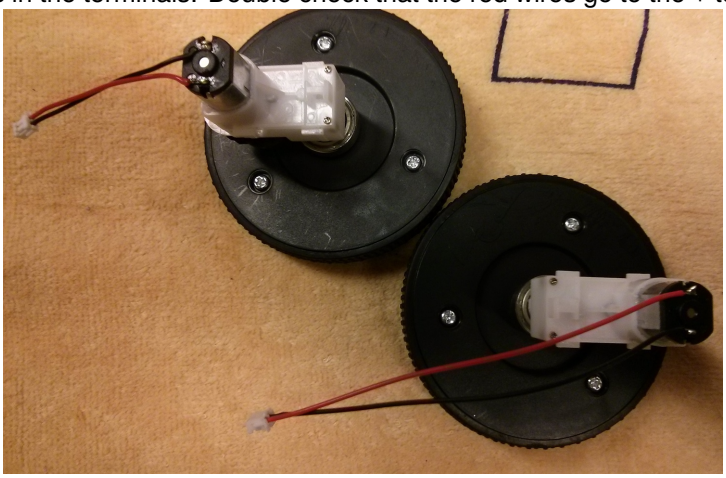
Step 1: Disassemble the powertrain unit by removing the 8 screws holding the gearboxes in. You may keep the wheels installed.



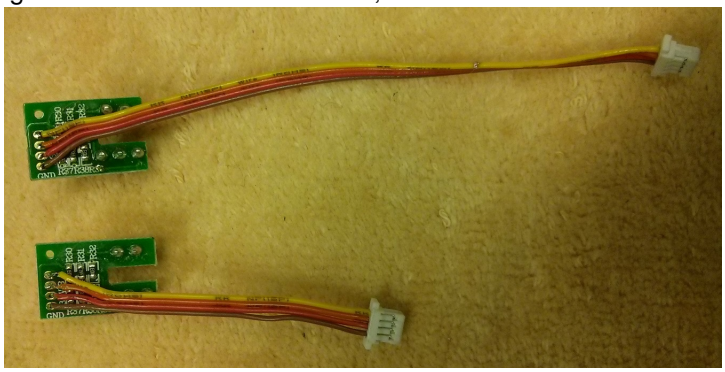
Step 2: Now that you have access to the motor terminals, trim one of the 2-pin pigtails and one of the 4-pin pigtails to roughly 50mm for the front motor and encoder. Then trim, without stripping yet, the remaining two pigtails to roughly 100mm for the rear motor and encoder. If you are new to stripping wires, you may want to leave yourself a few more mm extra as it is easy to cut straight through the wire.



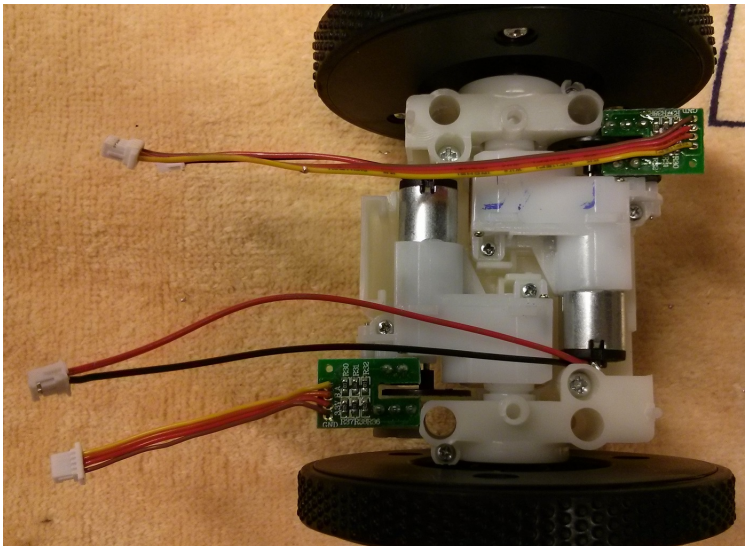
Step 3: Strip and tin the pigtail ends to prepare them for soldering. Also tin the motor terminals. There are plenty tutorials online for learning the basics of soldering, including tinning. When you strip the wires, make sure not to strip off more than 1.5mm of insulation as the insulation will shrink to expose more wire under heat. Carefully solder the motor wires to match the orientation shown below. The wires do not need to go through the holes in the terminals. Double check that the red wires go to the + terminals of the motors.



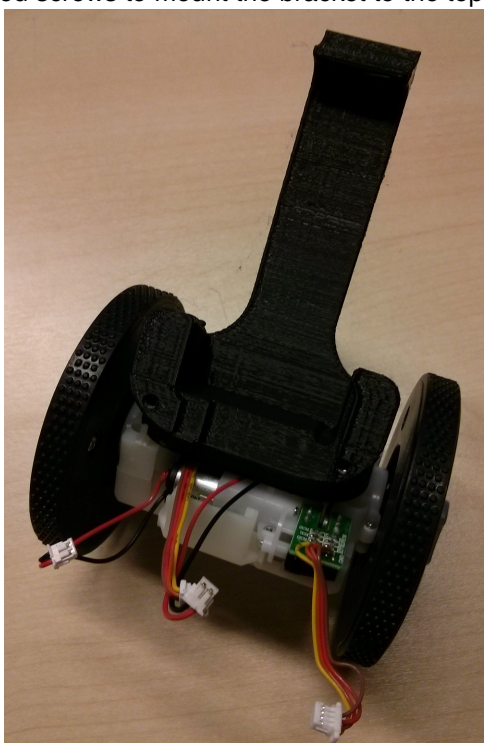
Step 4: With the soldering iron still hot, solder the encoder boards. This time thread the wires through the holes. It is not necessary to tin the wires here since the board is already tinned for you. You'll want to heat the tinned portion of the board and then apply solder rather than attempting to heat the wires themselves. The brown wire should go to the terminal labeled GND, the rest follow in order.



Step 5: Put the powertrain unit back together with wires and press the encoder boards in to match the orientation in the following picture.

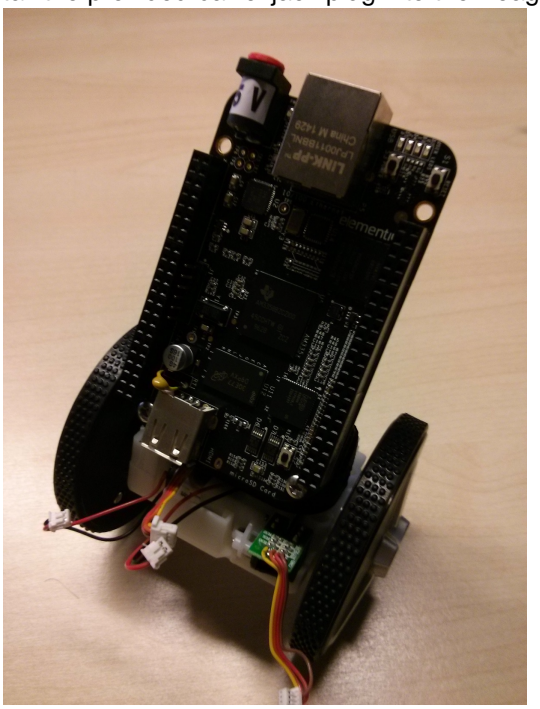


Step 6: Use two of the provided screws to mount the bracket to the top of the powertrain unit.

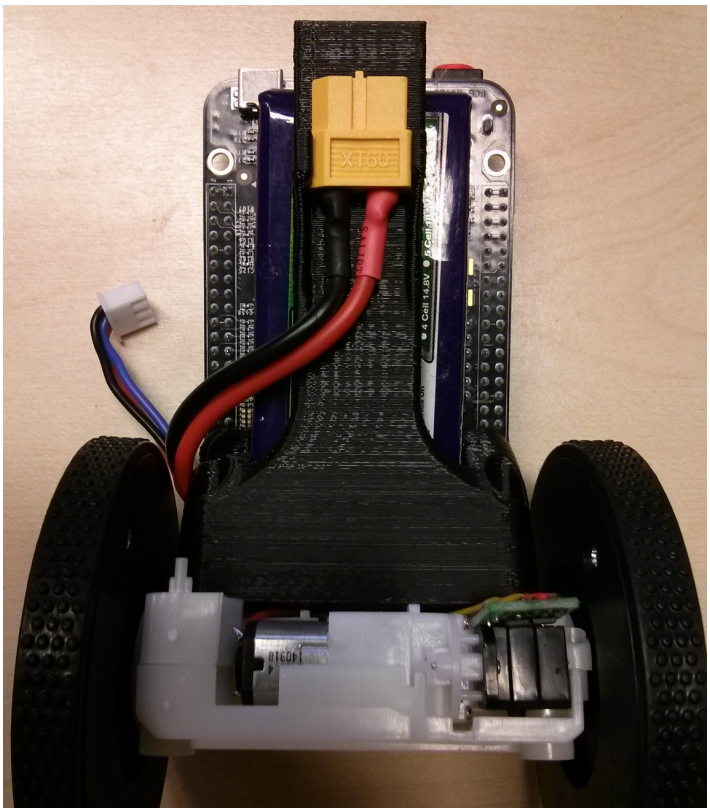


Step 7: Carefully bend the battery leads back with the thicker leads pulled towards the label like the following picture.

Step 8: Place the battery into it's slot and capture it in place by installing your BeagleBone with the remaining two screws. Also install the provided barrel jack plug into the BeagleBone's 5V DC input.

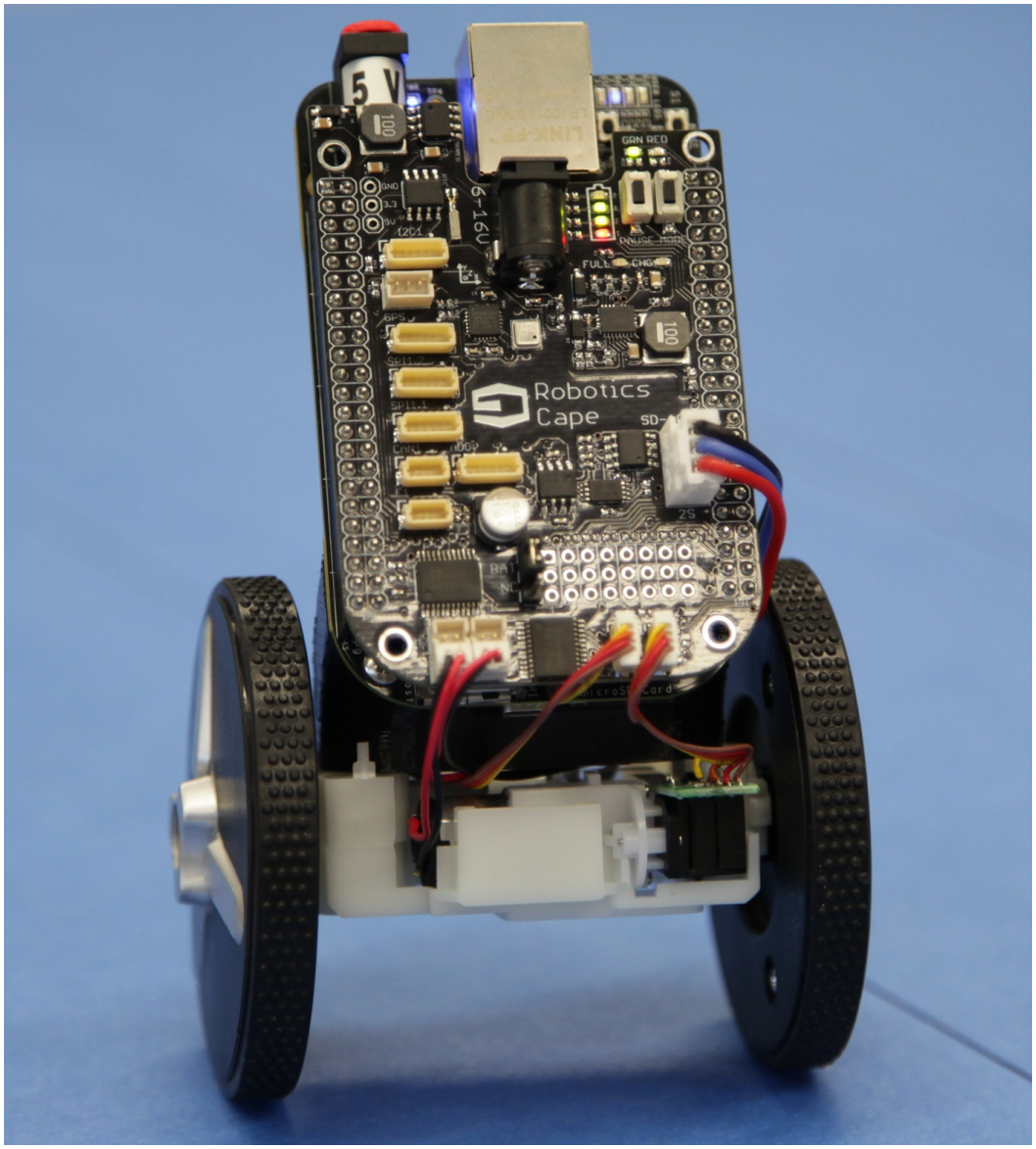


Step 9: Slide the yellow XT60 connector from the top down into the clip on the back side of the BeagleMiP bracket.



Step 10: Finally plug in the Robotics Cape and connectors. Note that your BeagleBone will not turn on immediately after connecting the battery. You must briefly connect the included 12V DC power supply to the Robotics Cape to arm the battery protection circuit.

With your MIP in the orientation depicted below, we suggest connecting the leads coming from the left motor and encoder to the M1 and E2 headers. Then connect the leads from the right motor to the M4 and E3 headers.



9.2 BeagleMiP Modeling

Referring to Example 17.6 Final Equations of Motion for the MIP are as follows.

$$\boxed{(m_r R L \cos\theta)\ddot{\phi} + (I_r + m_r L^2)\ddot{\theta} = m_r g L \sin\theta - \tau} \tag{1}$$

$$\boxed{(I_w + (m_r + m_w)R^2)\ddot{\phi} + (m_r R L \cos\theta)\ddot{\theta} = m_r R L \dot{\theta}^2 \sin\theta + \tau} \tag{2}$$

You will notice these are functions of torque, whereas we wish to design a controller that outputs a PWM duty cycle to our motors. Therefore we must also include the dynamics of the motors themselves. We can reasonably model a DC motor's output torque as a function of it's speed.

$$\tau = \bar{s} * u - b * \omega - \zeta * \omega \tag{3}$$

Where:

$\bar{s} = \frac{\eta k \gamma V}{Res} = StallTorque$

$b = \frac{\eta (k\gamma)^2}{Res} = DampingCoefficient$

$\zeta = \frac{k^2}{R} = viscousfriction$

η = Motor Efficiency

k = Motor Constant (Torque Constant)

γ = Gear Reduction

V = Voltage

Res = Resistance

ω = motor speed ($\dot{\phi} - \dot{\theta}$)

u = Motor Input (value between -1 and 1)

Your assembled BeagleMiP has roughly the following physical properties.

1. Encoder disks have 15 slots and therefore provide a resolution of 60 counts per motor revolution.
2. Motors have free run speed of 1760 rad/s and stall torque of 0.003Nm at 7.4V.
3. Motors are connected to the wheel with a 35.57:1 gearbox.
4. The motor armature has inertia 3.6E-8 Kg*m^2.
5. Wheels have a radius of 34mm and have mass 27g each.
6. Total assembled MiP has a mass of 263g.
7. MiP center of mass is 36mm above the wheel axis.
8. MiP body inertia is 0.0004 Kg*m^2 about the wheel axis.

Remember that you must account for the torque and inertia of both motors in your MiP model. Also note that when estimating the inertia of the wheels and gearbox, you must multiply the motor armature inertia by the square of the gearbox ratio before summing with the wheel inertia. You may treat the wheels as solid disks when estimating their inertia.

9.3 Exercise: Stabilizing Body Angle

Using the physical properties given above and the equations of motion provided in example 17.6 of Numerical Renaissance, develop a model G1(s) as a transfer function from duty cycle of the motors to angle theta of the MiP body in Radians. Now design a stabilizing transfer function D1(s) to keep MiP upright. I suggest plotting the angle of the system as the MiP falls over from a small angle with no input to the motors as a sanity check to make sure your unstable transfer function models the speed at which you predict MiP will fall over.

Submit the following:

1. Transfer function G1(s) from motor duty cycle to angle theta.
2. Plot of the MiP angle Theta as it falls over with time scale in seconds at the bottom and angle in radians on the left.
3. Your design for a stabilizing controller D1(s).
4. Bode and Nyquist plots of the open-loop system G1(s)*D1(s) indicating phase margin.
5. Discrete time equivalent controller D1(z) along with the equivalent difference equation.
6. BeagleBone C-code implementing your controller.

9.4 Exercise: Stabilizing Wheel Position

You may notice that although your controller D1 keeps your MiP upright, it still wants to drive away. Now design a second controller, D2, to stabilize the wheel position Phi. Since the dynamics of Theta and Phi are very different, you may use the successive loop closure method described in section 18.3.4 of Numerical Renaissance.

You may initially design the controller D2 assuming the inner loop has a constant gain of 1. This is to say that we assume the feedback of D1(s)*G1(s) is sufficiently fast that we treat it as 1 for the purpose of simplifying the design of D2 at a much slower timescale. When you think you have a stabilizing controller, include your model for G1(s) and controller D1(s) to check performance. You may need to add a prescaler P to the output of your controller D2(s) if thee feedback of G1(s)*D1(s) has steady state gain not equal to 1. With the entire successive loop closure system modeled in Matlab, plot the step response for a unit step in wheel position. This should model the MiP wheels moving forward one radian. You should observe non-minimum phase behavior where the wheels must roll backwards briefly before rolling forward.

Submit the following in a single PDF.

1. Your model for G2(s) and controller D2(s)
2. Nyquist plot demonstrating stability of outer loop with inner loop gain of 1
2. Continuous time step response assuming inner loop gain of 1
3. Continuous time step response with your G1(s) and D1(s) inner loop model included
4. Discrete time transfer function D2(z)
5. Difference equation for D2(z)
6. MiP Balancing code.

Also include a video with your email submission demonstrating balancing performance.