Lab 1: Meeting the Beaglebone Black and the Linux Operating System

Overview of the Board

In this lab we are going to start with the basics of the Beaglebone Black board. This board is a powerful piece of hardware that is capable of driving large embedded systems projects. For this class, we will be using it to ultimately host a wirelessly controlled tank. You can think of this board as a small and simple computer that can interact with and generate signals from the outside world. This communication of input/output signals is the basis of all electrical systems, and can be seen as a thesis of this class. As embedded engineers, our job is to manage and control all the signals that the board generates or receives.

Before we start with the fun stuff, let's go over the awesome attributes this board has. The Beaglebone Black supports the following features:

- Running an AM3354 TI Sitara processor based on ARM Cortex-A8 core
- Pico ITX standard dimensions (100mm×72mm) Boot from MMC or NAND Flash
- Max. 1 GHz core clock frequency.
- Three different power supply options (5V only with 3.5mm comb icon or micro USB connector, external power module e.g. 12V-24V input voltage).
- Two RJ45 jacks for 10/100 Mbps Ethernet
- One USB OTG interface available at an USB Micro-AB connector at the back side one secure
- Digital/Multi Media memory card interface brought out to a Micro-SD connector at the back side
- CAN interface at 5x2 pin header 2.54mm
- Audio codec with stereo Line in and Line out (3x2 pin header 2.54mm) and mono speaker (2-pole Molex)
- RS-232 transceiver supporting UART1 incl. Handshake signals with data rates of up to 1 Mbps (5x2 pin header 2.54mm)
- Reset – Button
- Audio/Video (A/V)connectors
- Expansion connector
- Backup battery supply for RTC with Gold cap (lasts approx. 17 1⁄2 days)

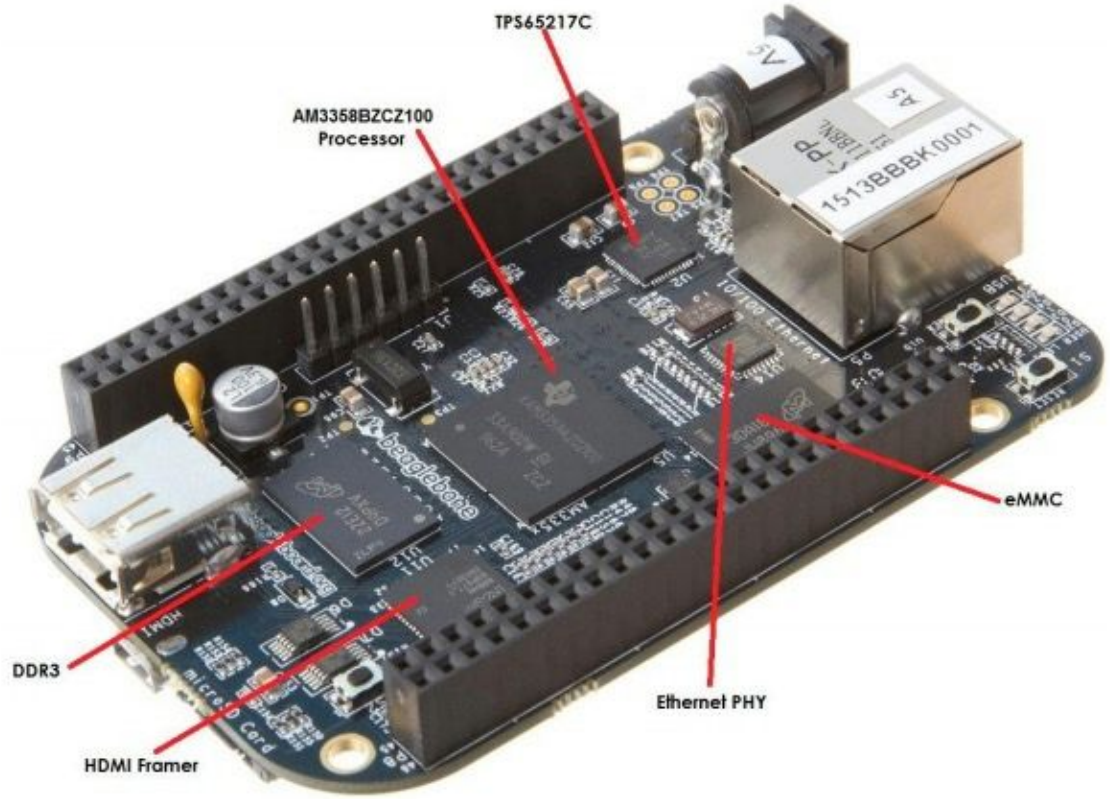

Figure 1: The overview of the Beaglebone Black

Connecting to the Board

We will primarily be communicating with this board using SSH. SSH -- or secure shell-- is a pretty common networking protocol used to remote login into other devices on a network. Start by firing up the terminal (ctrl + alt + t) and plugging in the ethernet cable to the ETH0 port on the board, see diagram above. Also make sure the power module is plugged in to the board or you won't be able to actually communicate with it. Then, on the host computer, type **ssh root@192.168.7.2** or **ssh 192.168.7.2 -l root** to enter the board. This command will attempt to grab root access to the target 192.168.3.11. This is the default local ip address of the board. Once you see the PHYTEC YOGURT logo on the top of the terminal, you have connected to the board successfully.

If you have not successfully connected to the board it's likely that the local network settings on your computer are not appropriately set up. If your local network needs to be configured manually, follow these steps:

System Settings > Network > Wired > Options > IPV4 Settings > Add

Address: 192.168.3.10 Netmask: 255.255.255.0 Gateway: 192.168.3.10

The Linux Operating Systems

Once you are connected on the board, you will be inside the Linux operating system of the board. You can type all of the Linux terminal commands (such as **ls**, **cd**, **echo**, etc.) inside the terminal. Remember, when you see the PHYTEC YOGURT logo, you are no longer inside the host computer; you are inside the board. When you are first connected, you will be in the **/home/** folder of the system. You can type **ls** to view all of the files and folders located in your current directory. At this stage, it's likely that this folder is empty for you, so don't panic if nothing shows up. To familiarize yourself with this system, you can type **ls** to list all of the files in your current directory, and **cd** to traverse into or out of these files. You can change your current directory by typing **cd <folder>** to enter that folder, or just **cd ../** to go back one level. It is a good idea to get a general understanding of the file system on this board and you will be interacting with it for this lab and the rest of the quarter.. If you want to know the version of Linux that is currently running on the board, you can enter the command **uname -r**.

Transferring Files to the Board

Before we start programming the board, we will show you how to transfer files to the board. We are also going to use the SSH protocol to send files to the board. The files could be anything and any kind. Let's see the steps:

- Make sure you are now at your host computer.
- open up a terminal, and traverse to some desired working directory.
- Create a new dummy file by typing the command **touch new_file** to make an empty file with the name "new_file" .
- You can add anything to the file, such as texts to the file. The *pure* terminal way would be to use the echo command to print text to this file. To edit the file in a text editor, you can also type **gedit new_file &**. gedit is the name of the text editor supplied with Ubuntu, and the ampersand symbol tells linux to run this in the background.
- Transfer **new_file** to the board by typing the command **scp new_file root@192.168.7.2:~**
- What the above command does is transferring **new_file** to the **~** (/home/) folder of the board. The account name is **root** with the address **192.168.7.2**, which is the address of the board.

Accessing the Pins

One essential part before we start the lab is to know where the pins are located. We will use the connections P8 and P9 on the board. The details of these connections can be found on the graph below.

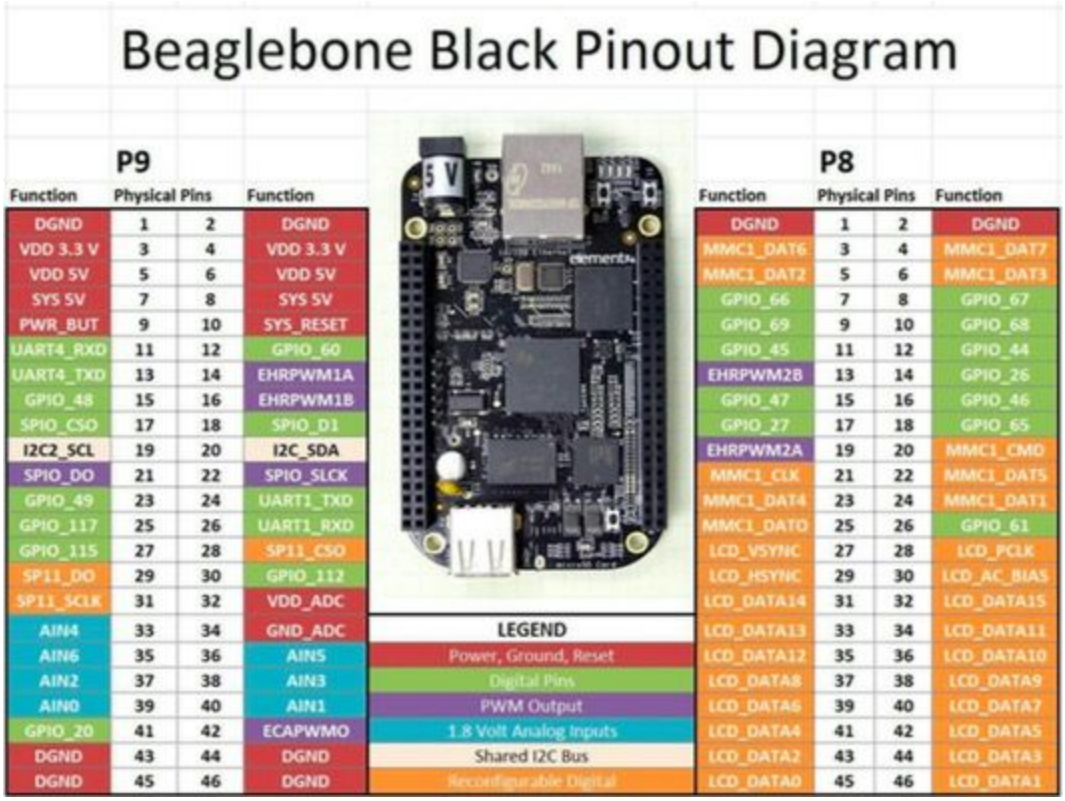

Figure 2: Beaglebone Black Pinout Diagram

Part 1

Introduction to the General-Purpose Input/Output (GPIO) pins

GPIO is a generic pin on an integrated circuit. In our case, the GPIO pins are directly mapped to the AM3354 processor. The behavior of these pins can be controlled by the user at run time. Its behavior is to output and receive high or low states. For our boards, the high state is defined to be at 3.3 V and the low state is 0 V. These pins can be used to configure other hardware devices such as LEDs and LCD screens. Let's practice using GPIOs to turn on an LED. You can use the hardware schematics and the pin configuration files to choose and configure a GPIO pin.

Let's see the steps:

- Connect an LED between a GPIO pin and ground. Remember LEDs are easy to burn out because of their small resistance, so a series resistor (>500Ω) to reduce current is a good idea.
- Before configuring the GPIOs, let's see how the GPIO numbering works. In schematics and documentations, you will see that all of the GPIO pins have the format GPIO*x*[*y*]. The letter *x* represents the bank, and the letter *y* represents which location in the bank that the GPIO pin is located. Usually, one bank will have 32 GPIO pins. So, the exact GPIO number would be *x*32 + y*. For example, GPIO1[31] can also be called GPIO63. However, for Beaglebone Black, all of the GPIO numbers are provided in **Figure 2**.
- The GPIO configurations are located in the directory **/sys/class/gpio**.
- Before we use a GPIO pin, we will have to request that GPIO pin. To do this, type this command on the target board: **echo GPIO_NUMBER > /sys/class/gpio/export**. For example, if you want GPIO63 to be available, type **echo 65 > /sys/class/gpio/export**. What this command does is creating a folder inside the **/sys/class/gpio** folder called **gpioGPIO_NUMBER** (for our example, **gpio65**).
- After exporting the folder, you will see the device folder for the GPIO that you requested. For the example above, you will see **gpio65** device folder inside your **/sys/class/gpio** folder. Inside the device folder, you will see device files:
    - "direction" ... reads as either "in" or "out".
    - "value" ... reads as either 0 (low) or 1 (high). If the GPIO is configured as an output, this value may be written; any nonzero value is treated as high
    - "edge" ... reads as either "none", "rising", "falling", or "both". Write these strings to select the signal edge(s) that will make poll on the "value" file return. This file exists only if the pin can be configured as an interrupt generating input pin.
    - "active_low" ... reads as either 0 (false) or 1 (true). Write any nonzero value to invert the value attribute both for reading and writing. Existing and subsequent poll support configuration via the edge attribute for "rising" and "falling" edges will follow this setting.

To set a configuration, you can use the command **echo** to these device files to configure a setting. For example, **echo out > direction** will configure the GPIO direction to be output.

Interacting with the board in this way has its drawbacks. While we are able to communicate with the peripherals on this board by writing to the dev files, we ideally want the process to be automated and efficient. For example, what if we want to turn on

10 LEDs with 10 GPIOs? Activating 10 GPIOs just by typing terminal commands can be a pain. You may wonder, can we program this behavior using a programming language language? Yup! For this reason, we will introduce you how to open and close files using the C language.

The User Space C Applications and Cross Compiling

To run any code on the board, we must first compile it down into an executable binary file. A brief introduction to the compiling process is helpful in understanding how to do this. The chart below shows a visual on all the steps your code will go through before it is executable by a machine.

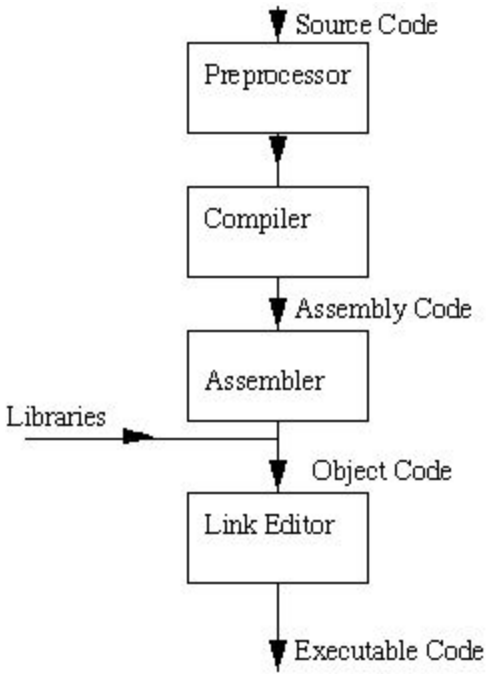

Figure 3: C Code Compilation Process

To compile and turn our code into a usable executable file, we will need the proper toolchain. To compile your codes, you have to first transfer your codes to the board, not your binaries. For your C codes, once you are in the board, you can compile the codes using “gcc”, not “arm-linux-gnueabihf-gcc” because the board is preinstalled with its own processor-specific compiler. If you are using makefiles, make sure to check your compiler type before proceeding to compile your codes.

LED User Space Application

Now, it’s your turn to do some programming. Your job is to write a C program in user space that will display a looping count of 0 to 7 in binary between 3 LEDs.

Part 2

Introduction to Pulse Width Modulation (PWM) Signals

A PWM signal is pretty much a digital signal that is alternating from high and low. It can be used to control devices such as a mechanical servo, a buzzer, or even as a clock output to another digital device. We will be using all of these by the end of the quarter. The properties of PWM include the period and the duty cycle of thy signal. The following picture shows clearly enough what a PWM is:

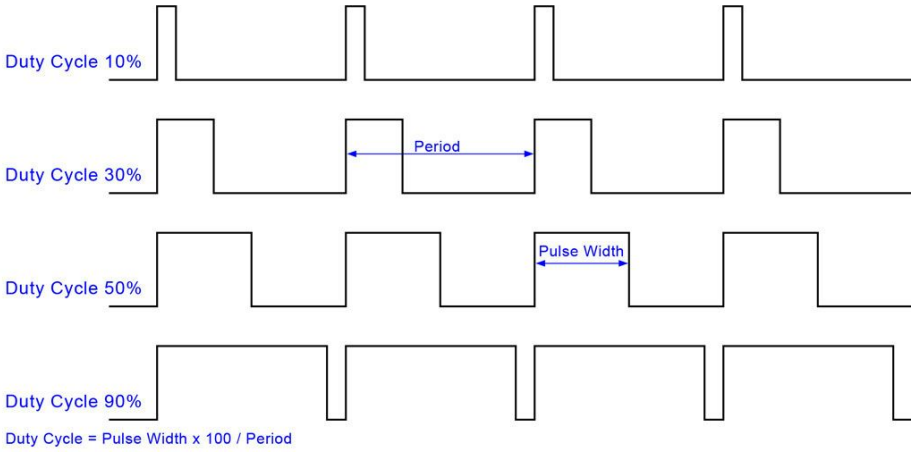

Figure 4: PWM Signals

In our boards, as you can see on the pinout diagram. There are few PWM pins like EHRPWM1A, EHRPWM1B, ECAPPWMO, etc. These commands create the device configuration folders for each channel:

"cd /sys/devices/bone_capemgr.8"

"echo am33xx_pwm > slots"

"echo bone_pwm_P9_14 > slots" → for EHRPWM1A

"echo bone_pwm_P8_19 > slots" → for EHRPWM2A

The above device can also be done in user space C programming by just writing to the **slots** file using FILE I/O operations.

The configuration folders are called:

/sys/devices/ocp.3/pwm_test_P9_14.* → for EHRPWM1A

/sys/devices/ocp.3/pwm_test_P8_19.* → for EHRPWM2A

the * at the end will be a certain number when you actually do it in bash. And it may change depending on the order you require the pwm pins. So the when you write the C program, what you probably want to do is to configure them in bash first and remember those numbers. In your C program, configure them in the same order and replace the * at the end with the numbers you just found.

Inside these folders, you will find the following device configuration files that you can open, write to, and read from them:

- period - set the period in nanoseconds
- duty_cycle - set the duty_cycle in nanoseconds
- polarity - set as “normal” or “inversed”
- enable - turn on PWM (1), turn off PWM (0)

Note: the configuration files of EHRPWM1B (P9 pin 16) cannot be changed. They have fixed values.

Scientific Pitch Notation

Although PWM is a digital signal, it can emulator an analog signal well enough to play simple (not very good) sounds on a piezo buzzer. The pitch of the sound can be changed by altering the frequency of the PWM. For example, one frequency for the pitch of C is 523.25Hz. This means that by setting the period of the PWM to (1 / 523.25Hz) will play a C note-- or at least something close. Using the chart of listed frequencies on the wikipedia page for scientific pitch notation, It’s possible to play all notes on the piano using PWM.

PWM User Space Code

For the next portion of the code, you will be appending to your blinking code from before. Within the same counting loop, you should make the PWM play a different sound for each different count value.

Makefile

When you write the C user space code for GPIOs, you have to compile it using the **gcc** toolchain. What if we have to compile 100 C codes, but we don’t know how to organize them? Don’t worry, we will introduce makefiles.

Makefiles are special format files that together with the *make* utility will help you to automatically build and manage your projects. The name of a makefile is usually just *Makefile* and you store this file alongside your C source codes and header files in one directory.

The inside of a makefile usually consists of:

```
# comment
# (note: the <tab> in the command line is necessary for make to work)
target1:  dependency1 dependency2 … dependendyM
      <tab> command
        .
        .
        .
targetN:  dependency1 dependency2 … dependencyM
      <tab> command
```

The target is the argument that the *make* utility uses to execute the command. A dependency is a file that is used as input to create the target. A target often depends on several files.

For example, for your GPIO or PWM related codes, you will have something like this inside your makefile:

```
# target1 => Create executables and object files for GPIO related code
gpio_exe: gpio_code.c gpio_helper.c gpio_header.h
       gcc -o gpio_exe gpio_code.c gpio_code_helper.c

# target2 => Create executables and object files for PWM related code
pwm_exe: pwm_code.c pwm_helper.c pwm_header.h
       gcc -o pwm_exe pwm_code.c pwm_helper.c

# target3 => remove object files and binary files
clean:
       rm -rf *.o gpio_exe pwm_exe
```

To actually use the Makefile, you can type the command *make your_target* in your host computer. For our example, if you type the command *make gpio_exe*, it will create the binary and executable files for the GPIO related codes. Simple right? To learn more about makefiles, you can read through the sources that are provided in the reference section.

Pinmuxing and the Device Tree

This section is for reference, and has already been set up for you. You may want to change this later on in the class, especially for the final lab. The pinmuxing process is

important to know, but let's first get familiar with how the operating system keeps track of the hardware on board. This information is stored within a data structure called the device tree.

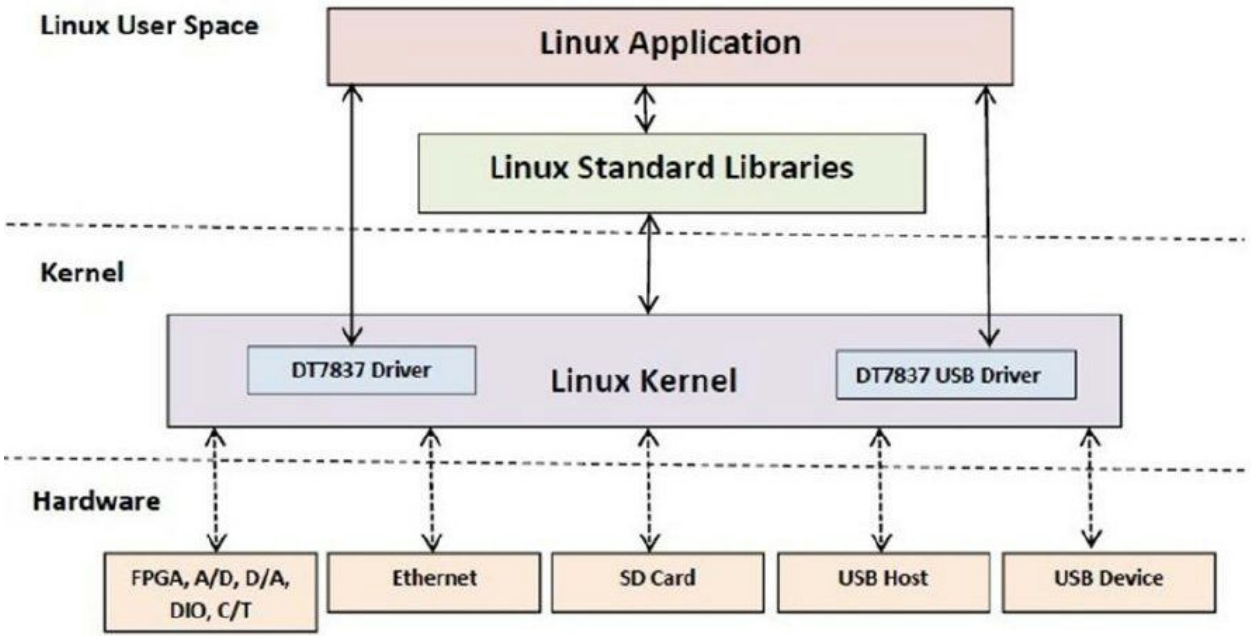

Figure 5: The different layers of the Linux operating system

Within the Kernel is the device tree. This will be the the lowest interface we use to communicate with the hardware in this class. The Device Tree is a data structure for describing hardware. Rather than hard coding every detail of a device into an operating system, many aspects of the hardware can be described in a data structure that is passed to the operating system at boot time. The data structure itself is a simple tree of named nodes and properties. Nodes contain properties and child nodes. Properties are simple name-value pairs. The structure can hold any kind of data.

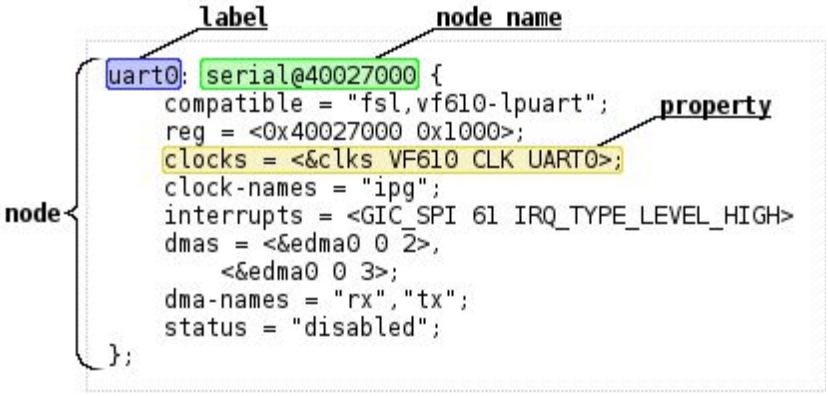

Figure 6: Example of Device Tree Node

The device tree structures are usually stored in a file that ends with .dts (device tree source) that then will be compiled in the kernel source. After the compilation process, the device tree structure will be represented in a binary format (.dtb files) that will be loaded to the board.

The way we will be interacting and modifying the device tree will be through the `am335x-wega-rdk.dts` file located inside the Board Support Package (BSP) in the host computer directory below:

`/home/ee472admin/bb-kernel/KERNEL`

The major modification that we will be making to the device tree is with pin multiplexing. Since the logic functions of the chip are greater than the amount of pins connected to it, each pin is muxed in order to have multiple functionality. As with any sort of multiplexor, we are able to select the mode we want by providing the hardware with the configuration that we desire. If we want our configuration to take effect, we have to recompile the kernel and load the new kernel to the board.

For this first lab, we will have modified the device tree file in order to enable certain pins as a GPIO (general purpose in/out) and a PWM (pulse width modulation). We have already configured the pinmux to allow use of GPIO using the BSP installed on the computers. This modified file can be found in the folder for lab 0.

What to turn in

Create a new repository on git and push the following files:

```
README.md    // simple overview to your code
music.c      // your user space code
music.h      // optional if you wrote one
Makefile     // working makefile
```

You will push your clean and well-commented .c file containing the working code from part 3. The name of your file is not important, but avoid using names like *lab0.c* Most of standard linux files try to sound cool by using short names with all lowercase letters and abbreviations. *ex: videodev.c*. Underscores are also common to split up files with multiple words. ex: *sparse_keymap.c*. We personally dislike using underscores for most cases, but you will not be docked any points for having files with underscores in their name.

If your group is familiar with header files and you want to include one, that will be accepted as well. Additional script files are also accepted. A readme file must be included explaining high level information about your program, but doesn't have to be long. There is no strict style for this first lab, but your code should be elegant and concise. If you wish to follow some style guide, Google's style guide for C++ is a good reference.

Demo

You will also need to sign up for a time to demo your code. In the demo, expect to show how to pull, compile, and upload your code from your repository to your board and run it. You also should explain your group's design choices for this project. A TA will ask questions to your group that will be graded, but you can expect these questions to be neither difficult nor critical to your grade. We mostly want to get a feel for your individual project and each member's contribution to the group.

Documentation

Hardware Manuals:
https://github.com/CircuitCo/BeagleBone-Black/blob/master/BBB_SRM.pdf?raw=true
Schematic:
https://github.com/CircuitCo/BeagleBone-Black/blob/master/BBB_SCH.pdf?raw=true
Quick Start Guide for BBB:
http://www.ee.washington.edu/class/474/ecker/
Makefiles:
http://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html
Scientific pitch notation:
https://en.wikipedia.org/wiki/Scientific_pitch_notation
Google C++ style guide:
https://google-styleguide.googlecode.com/svn/trunk/cppguide.html
Application Guide (for extra info but not necessary):
http://phytec.com/site/assets/files/2061/l-792e_0.pdf
Expansion Boards Application Guide:
http://phytec.com/site/assets/files/2061/l-793e_0.pdf