Lab Book

Zachary Whitlock

EE-407: Embedded Systems Hardware

Oregon Institute of Technology

Professor: Allan Douglas

Fall Term 2020

Table of Contents

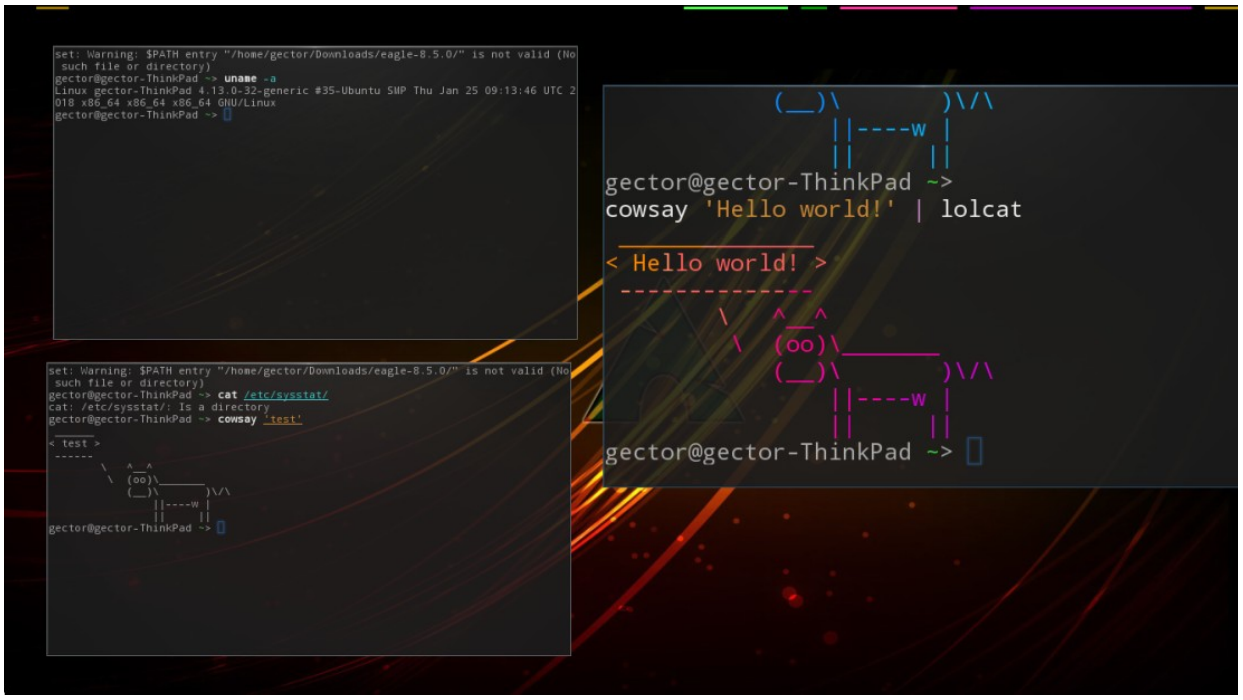
Lab 1

Introduction

Objective: To install and configure a software development environment for using throughout the school term.

Part 1 – Setting up Ubuntu Laptop

I normally use Linux as my main operating system, so I already have an ubuntu laptop. Shown is a screenshot from my normal desktop, but I have access to vanilla ubuntu as well.



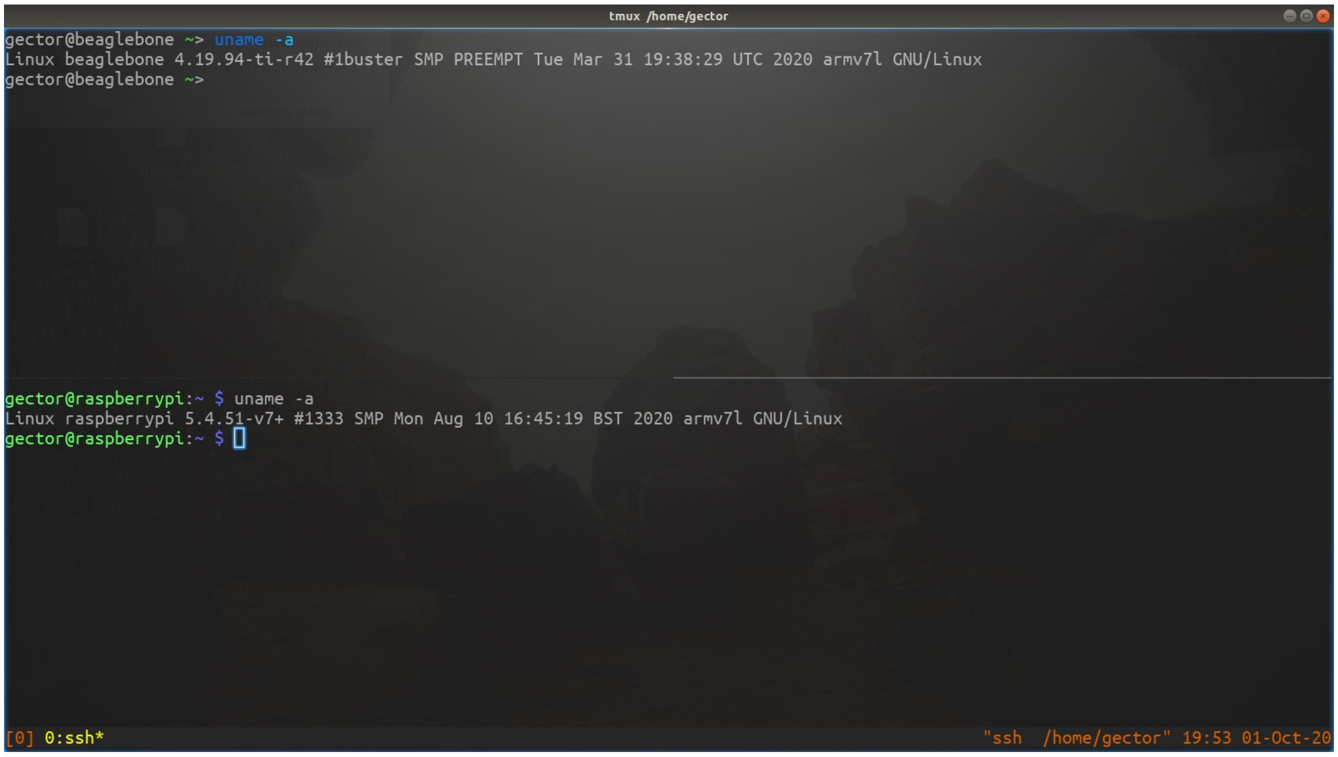
```
set: Warning: $PATH entry "/home/gector/Downloads/eagle-8.5.0/" is not valid (No
 such file or directory)
gector@gector-ThinkPad ~> uname -a
Linux gector-ThinkPad 4.13.0-32-generic #35-Ubuntu SMP Thu Jan 25 09:13:46 UTC 2
018 x86_64 x86_64 x86_64 GNU/Linux
gector@gector-ThinkPad ~>
```

```
            (__)\       )\/\
                ||----w |
                ||     ||
gector@gector-ThinkPad ~>
cowsay 'Hello world!' | lolcat
 ______________
< Hello world! >
 --------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
gector@gector-ThinkPad ~>
```

```
set: Warning: $PATH entry "/home/gector/Downloads/eagle-8.5.0/" is not valid (No
 such file or directory)
gector@gector-ThinkPad ~> cat /etc/sysstat/
cat: /etc/sysstat/: Is a directory
gector@gector-ThinkPad ~> cowsay 'test'
 ______
< test >
 ------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
gector@gector-ThinkPad ~>
```

Part 2 – Checking Linux Distributions

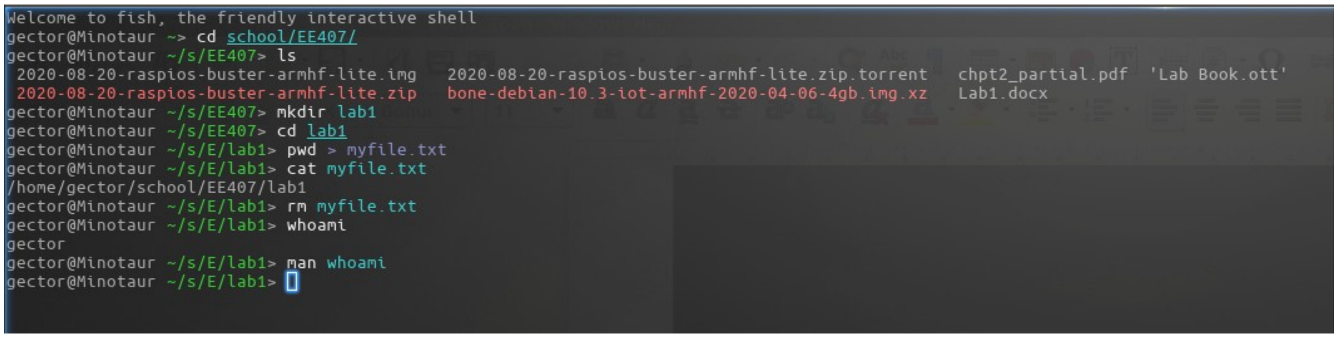
The raspberry pi and BeagleBone Black boards were configured so that I could remotely connect to them over my network. This screenshot shows both the raspberry pi and beaglebone and their linux versions.



```
tmux /home/gector
gector@beaglebone ~> uname -a
Linux beaglebone 4.19.94-ti-r42 #1buster SMP PREEMPT Tue Mar 31 19:38:29 UTC 2020 armv7l GNU/Linux
gector@beaglebone ~>

gector@raspberrypi:~ $ uname -a
Linux raspberrypi 5.4.51-v7+ #1333 SMP Mon Aug 10 16:45:19 BST 2020 armv7l GNU/Linux
gector@raspberrypi:~ $ 
[0] 0:ssh*                                                          "ssh  /home/gector" 19:53 01-Oct-20
```

Part 3 – Basic Linux Commands

Laptop commands:



```
Welcome to fish, the friendly interactive shell
gector@Minotaur ~> cd school/EE407/
gector@Minotaur ~/s/EE407> ls
 2020-08-20-raspios-buster-armhf-lite.img   2020-08-20-raspios-buster-armhf-lite.zip.torrent   chpt2_partial.pdf  'Lab Book.ott'
 2020-08-20-raspios-buster-armhf-lite.zip   bone-debian-10.3-iot-armhf-2020-04-06-4gb.img.xz   Lab1.docx
gector@Minotaur ~/s/EE407> mkdir lab1
gector@Minotaur ~/s/EE407> cd lab1
gector@Minotaur ~/s/E/lab1> pwd > myfile.txt
gector@Minotaur ~/s/E/lab1> cat myfile.txt
/home/gector/school/EE407/lab1
gector@Minotaur ~/s/E/lab1> rm myfile.txt
gector@Minotaur ~/s/E/lab1> whoami
gector
gector@Minotaur ~/s/E/lab1> man whoami
gector@Minotaur ~/s/E/lab1> 
```

ls: Lists the files in the current directory

mkdir: Makes a new directory

cd: Move into another directory

pwd: Show the current directory

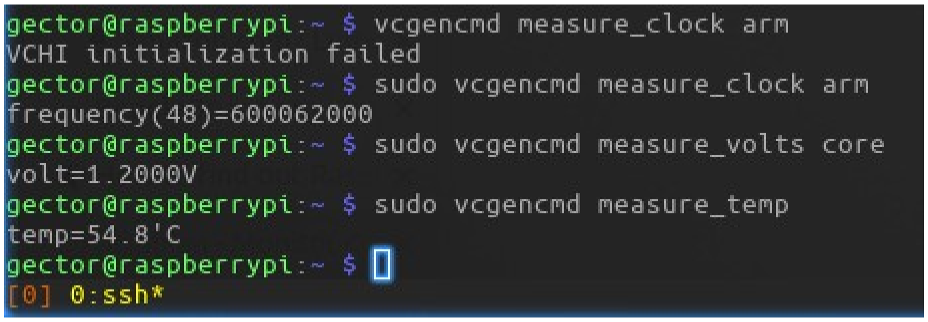
cat: Print a file into the terminal

rm: Remove/delete a file

whoami: Print your username

man: Show the manual entry for a command

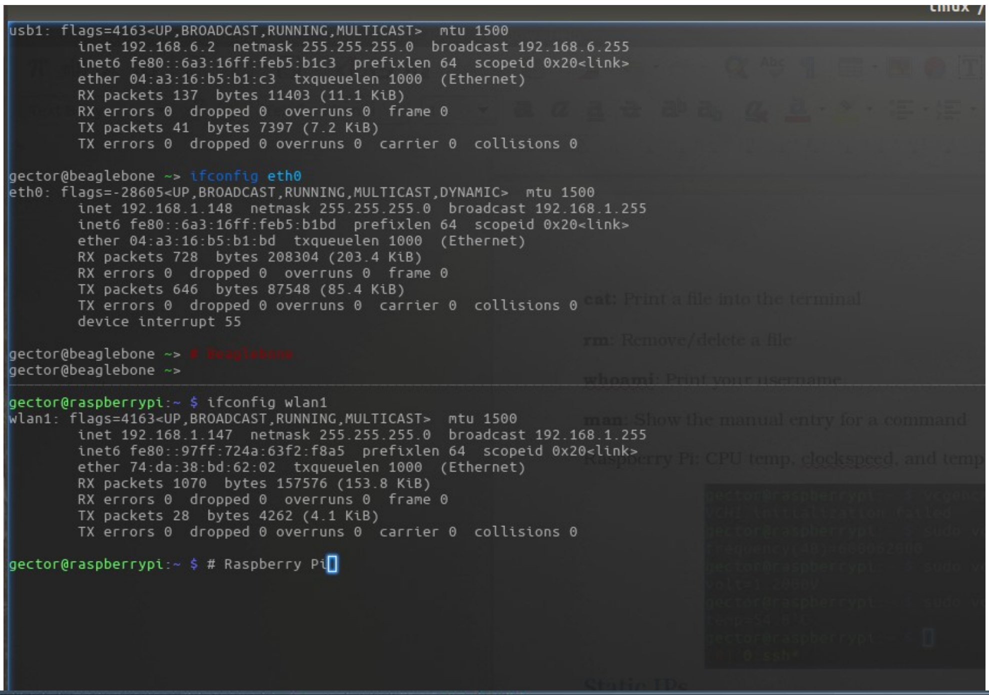
Raspberry Pi: CPU temp, clockspeed, and temperature



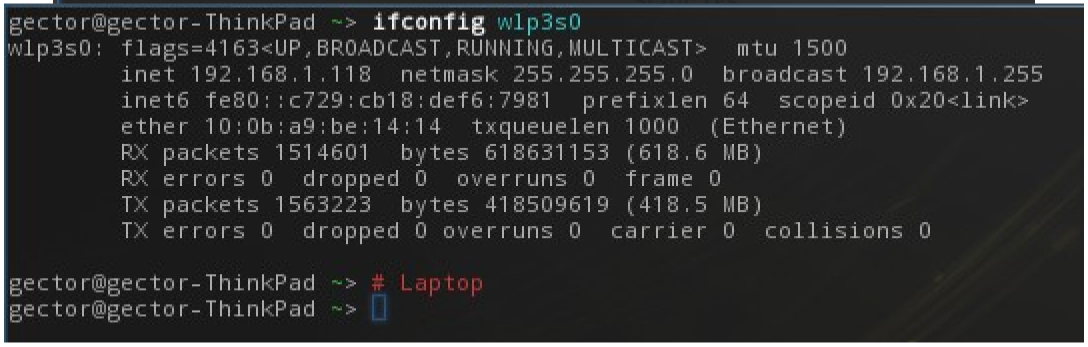
```
gector@raspberrypi:~ $ vcgencmd measure_clock arm
VCHI initialization failed
gector@raspberrypi:~ $ sudo vcgencmd measure_clock arm
frequency(48)=600062000
gector@raspberrypi:~ $ sudo vcgencmd measure_volts core
volt=1.2000V
gector@raspberrypi:~ $ sudo vcgencmd measure_temp
temp=54.8'C
gector@raspberrypi:~ $ 
[0] 0:ssh*
```

Static IPs

This also demonstrates that I've ssh'd into the beaglebone and pi.



```
usb1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.6.2  netmask 255.255.255.0  broadcast 192.168.6.255
        inet6 fe80::6a3:16ff:feb5:b1c3  prefixlen 64  scopeid 0x20<link>
        ether 04:a3:16:b5:b1:c3  txqueuelen 1000  (Ethernet)
        RX packets 137  bytes 11403 (11.1 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 41  bytes 7397 (7.2 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

gector@beaglebone ~> ifconfig eth0
eth0: flags=-28605<UP,BROADCAST,RUNNING,MULTICAST,DYNAMIC>  mtu 1500
        inet 192.168.1.148  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::6a3:16ff:feb5:b1bd  prefixlen 64  scopeid 0x20<link>
        ether 04:a3:16:b5:b1:bd  txqueuelen 1000  (Ethernet)
        RX packets 728  bytes 208304 (203.4 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 646  bytes 87548 (85.4 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 55

gector@beaglebone ~> # Beaglebone
gector@beaglebone ~>

gector@raspberrypi:~ $ ifconfig wlan1
wlan1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.147  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::97ff:724a:63f2:f8a5  prefixlen 64  scopeid 0x20<link>
        ether 74:da:38:bd:62:02  txqueuelen 1000  (Ethernet)
        RX packets 1070  bytes 157576 (153.8 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 28  bytes 4262 (4.1 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

gector@raspberrypi:~ $ # Raspberry Pi
```



```
gector@gector-ThinkPad ~> ifconfig wlp3s0
wlp3s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.118  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::c729:cb18:def6:7981  prefixlen 64  scopeid 0x20<link>
        ether 10:0b:a9:be:14:14  txqueuelen 1000  (Ethernet)
        RX packets 1514601  bytes 618631153 (618.6 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1563223  bytes 418509619 (418.5 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

gector@gector-ThinkPad ~> # Laptop
gector@gector-ThinkPad ~> 
```

Conclusion

All three devices, the raspberry pi, the beaglebone black board, and the laptop, were configured with a linux operating system and made to talk to eachother with static IP addresses.

It was difficult at first to get the static IP addresses to work correctly, but after reading a few more articles and modifying the '*/etc/dhcpd.conf*' file, the static addresses were established.

It also took a while to figure out that you have to hold a button down on my beaglebone black to make it boot to the SD card, it was booting to onboard flash and had a super old version of linux on it which wasn't working with the apt package manager.

Lab 2

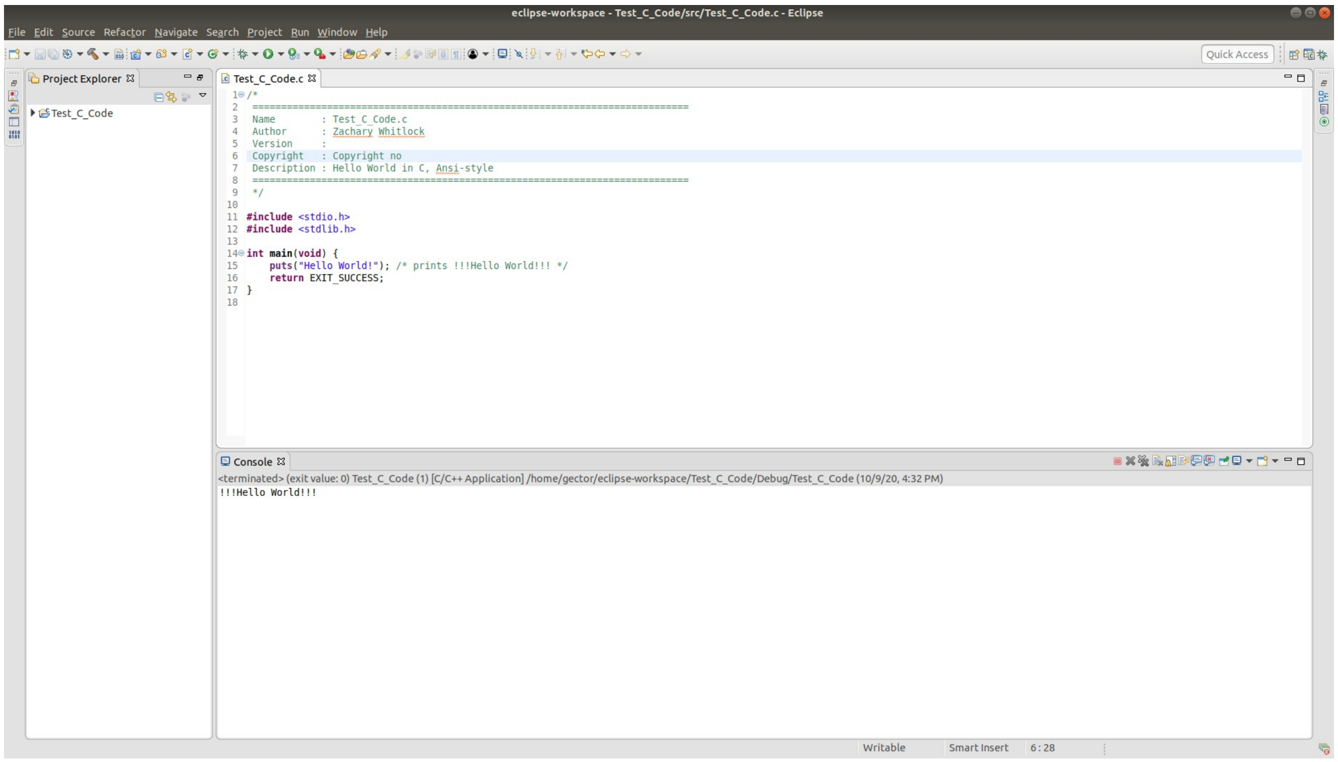
Introduction & Objective

Often times it is advantageous to use a powerful IDE for the sake of writing and compiling code. An IDE (Integrated Development Environment) has many tools built into it, but is essentially a text-editor at the core. IDEs often will catch syntax errors and provide extra functionality, such as search & replace. The issue is, it's not normally practical to run powerful development tools on the actual device you are developing for. Less powerful ARM devices like the raspberry pi and beaglebone black do not work well as development computers but are excellent embedded processors. Hence, we develop our code on a powerful desktop or laptop computer and cross-compile the code for ARM. Cross-compilation consists of using a normal (Intel X86) computer and compiling for ARM.

The objective of this lab is to use the Eclipse IDE to compile simple C and C++ programs for our ARM devices (Raspberry and Beaglebone). Additionally we will manipulate the I/O pins of the devices by use of the Linux terminal.

Part 1 – Installing Eclipse & Cross Compilation

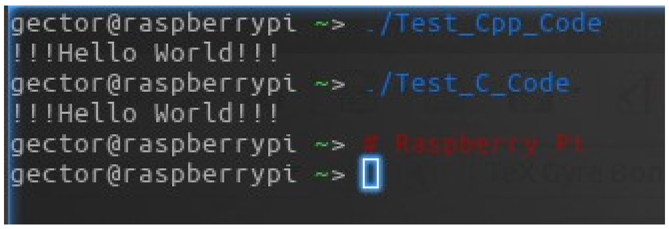
The Eclipse IDE was downloaded from www.eclipse.org/downloads/, the most recent version was used and placed on a Linux (ubuntu) system.



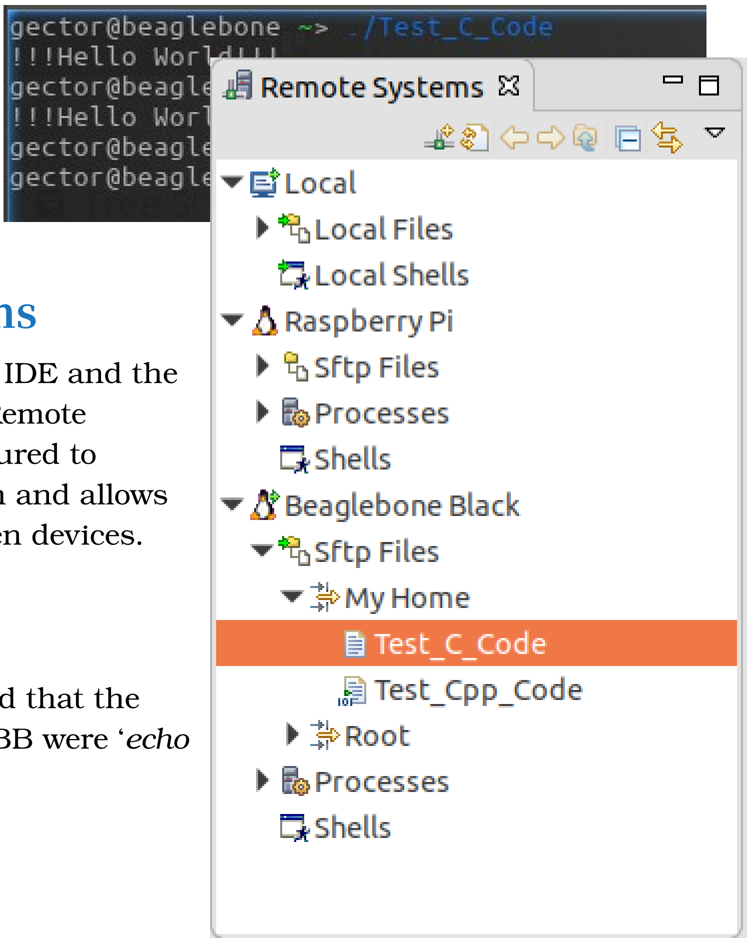
```
/*
 ============================================================================
 Name        : Test_C_Code.c
 Author      : Zachary Whitlock
 Version     :
 Copyright   : Copyright no
 Description : Hello World in C, Ansi-style
 ============================================================================
 */

#include <stdio.h>
#include <stdlib.h>

int main(void) {
	puts("Hello World!"); /* prints !!!Hello World!!! */
	return EXIT_SUCCESS;
}
```

```
<terminated> (exit value: 0) Test_C_Code (1) [C/C++ Application] /home/gector/eclipse-workspace/Test_C_Code/Debug/Test_C_Code (10/9/20, 4:32 PM)
!!!Hello World!!!
```

Additionally, by the use of the ‘apt’ package manager on Ubuntu, the cross-compilation tools were installed (‘gcc-arm-linux-gnueabi’, ‘arm-linux-gnueabi-gcc’) and then a new configuration was created in Eclipse for the ARM toolchain. The hello-world program was written in both C++ and C and compiled for ARM. The files were copied by using the `scp` (secure copy) command and then executed on the BBB and RPi.



```
gector@raspberrypi ~> ./Test_Cpp_Code
!!!Hello World!!!
gector@raspberrypi ~> ./Test_C_Code
!!!Hello World!!!
gector@raspberrypi ~>
gector@raspberrypi ~>
```



```
gector@beaglebone ~> ./Test_C_Code
!!!Hello Worl
gector@beagle
!!!Hello Wor
gector@beagle
gector@beagle
```

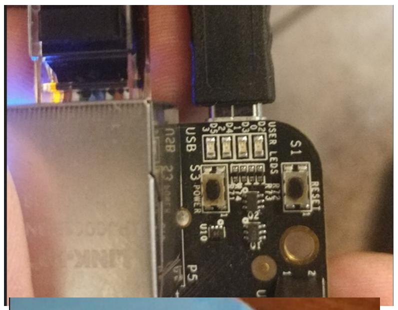
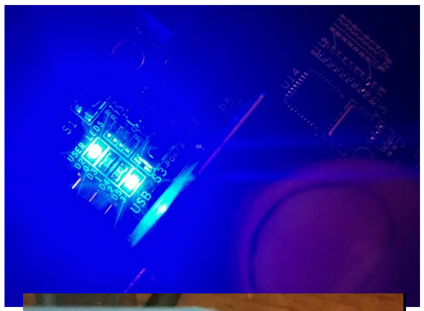
Part 2 – Eclipse Remote Systems

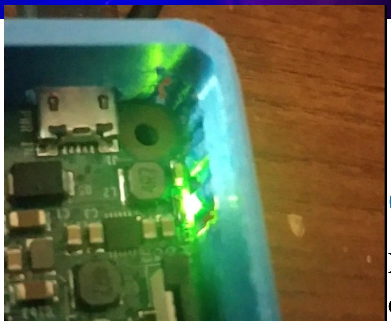
For easier transfer of files between the eclipse IDE and the ARM devices, a add-on was installed called “Remote Systems User Actions”. The plugin was configured to connect to both the RPi and the BBB with ssh and allows files to simply be dragged and dropped between devices.

Part 3 – Interacting with GPIO

From resources found online, it was discovered that the commands to write an LED on the RPi and BBB were ‘*echo*

255 > /sys/class/leds/led0/brightness' and '*echo 1 > /sys/class/leds/beaglebone\:green\:usr3/brightness*' respectively.



Conclusion

In this lab I learned how to configure Eclipse for compilation (and cross compilation) of C and C++ code. I also learned how to toggle the GPIO pins from bash in linux, and how to move files between Eclipse and the embedded devices directly.

Appendix – Code

```
//============================================================================
// Name        : Test_Cpp_Code.cpp
// Author      : Zachary Whitlock
// Version     :
// Copyright   : Copyright no
// Description : Hello World in C++, Ansi-style
//============================================================================

#include <iostream>
using namespace std;

int main() {
	cout << "Hello World! - CPP - ZW" << endl;
	return 0;
}
/*
 ============================================================================
 Name        : Test_C_Code.c
 Author      : Zachary Whitlock
 Version     :
 Copyright   : Copyright no
 Description : Hello World in C, Ansi-style
 ============================================================================
 */

#include <stdio.h>
#include <stdlib.h>

int main(void) {
	puts("Hello World! - C - ZW");
	return EXIT_SUCCESS;
}
```

Lab 3

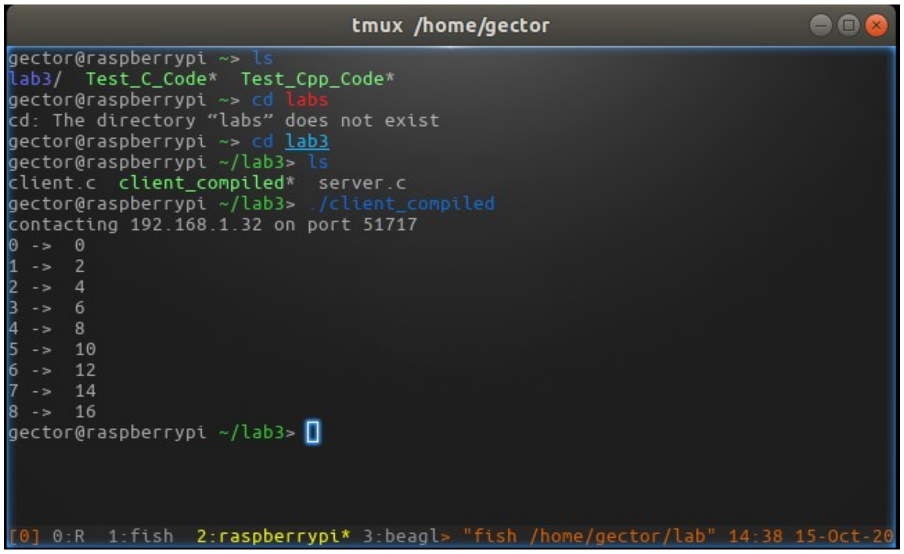
Introduction

The objective of this lab is to use the network sockets on the RPi and BBB to transmit and receive data between devices. The code was given as example code and makes use of the Linux socket system. Along the way the use of sockets should become much easier as I learn how to use them. With sockets, we can send data between different embedded devices on the same network.

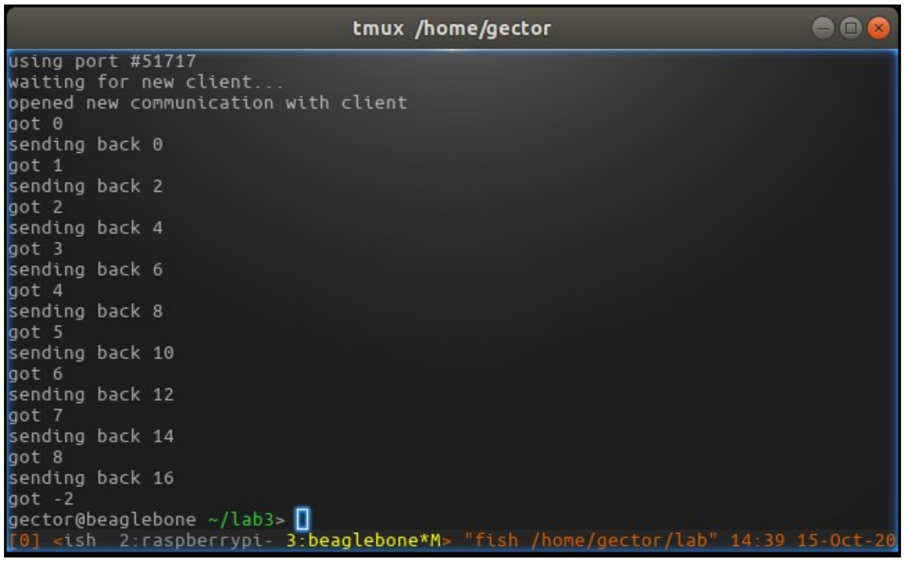
Part 1

The example code was compile for ARM Devices and transferred with *scp* before using eclipse for future revisions of the code. The client executable was placed on the RPi and the server code was placed on the BBB.

First, the server code was executed using the './' syntax, and then, switching to the RPi, the client code was executed similarly. The executable worked perfectly as can be observed in figures 1 & 2.



```
gector@raspberrypi ~> ls
lab3/  Test_C_Code*  Test_Cpp_Code*
gector@raspberrypi ~> cd labs
cd: The directory “labs” does not exist
gector@raspberrypi ~> cd lab3
gector@raspberrypi ~/lab3> ls
client.c  client_compiled*  server.c
gector@raspberrypi ~/lab3> ./client_compiled
contacting 192.168.1.32 on port 51717
0 ->  0
1 ->  2
2 ->  4
3 ->  6
4 ->  8
5 ->  10
6 ->  12
7 ->  14
8 ->  16
gector@raspberrypi ~/lab3>
[0] 0:R  1:fish  2:raspberrypi* 3:beagl> "fish /home/gector/lab" 14:38 15-Oct-20
```



```
using port #51717
waiting for new client...
opened new communication with client
got 0
sending back 0
got 1
sending back 2
got 2
sending back 4
got 3
sending back 6
got 4
sending back 8
got 5
sending back 10
got 6
sending back 12
got 7
sending back 14
got 8
sending back 16
got -2
gector@beaglebone ~/lab3>
[0] <ish  2:raspberrypi- 3:beaglebone*M> "fish /home/gector/lab" 14:39 15-Oct-20
```

Figure 2: Beaglebone

Part 2 – Dissecting the Code

Function:	Parameters:	Returns:	Header:
error()	pointer to char	N/A	errno.h
This function is used to report general problems during program execution.			
perror()	Pointer to char	N/A	stdio.h
Prints out an error			
func()	An integer	An integer	N/A
Multiplies a given integer by 2 and then returns the result			
sendData()	Two integers	N/A	N/A
Converts an integer to a string and attempts to write to socket, throwing an error if unsuccessful.			
getData()	An integer	An integer	N/A
Attempts to read data from the given socket file descriptor, throws an error if unsuccessful.			
sprintf()	Two pointers to chars	An integer	stdio.h
Takes an input, formats it, and stores the result into a string. Returns the number of characters stored in the array.			
read()	Integer, void *, size_t	ssize_t	unistd.h
Attempts to read and store data from a file given by a file descriptor.			
write()	Integer, void *, size_t	ssize_t	unistd.h
Writes a number of bytes to a given file by it's descriptor. Returns number of bytes actually written.			
strlen()	Char pointer	size_t	string.h
Returns the length of a null-terminated string in bytes.			
atoi()	Char pointer	An integer	stdlib.h
Converts a string to an integer and returns it.			
bzero()	Void pointer, size_t	N/A	string.h
Sets a number of bytes to 0.			
sizeof()	A type or expression	An integer	N/A
Returns the size of a piece of data in			
htons()	uint16_t	uint16_t	netinet/in.h
Converts input to network byte order, which is big-endian, for socket communication.			
bind()	3 ints, socket_t	An integer	sys/socket.h
Assigns an address to a specified socket with the specified address format.			
listen()	Two ints	An integer	sys/socket.h
Listens on a given socket and creates a queue of given size.			
accept()	Two integers, socklen_t *	An integer	sys/socket.h

Accepts a connection request on a socket. Returns the file descriptor of the new socket once a connection is accepted.			
socket()	Three integers	An integer	sys/socket.h
Creates a new socket given it’s domain, type, and protocol. Returns the file descriptor of the new socket.			
gethostbyname()	Char pointer	Hostent	netdb.h
Returns information about the requested host, or a null pointer.			
connect()	Two integers, A socklen_t	An integer	sys/socket.h
Opens a connection between a file of given file descriptor and a socket with given socket address.			
close()	An integer	An integer	unistd.h
Takes a file descriptor for a socket and closes it.			

Part 3 – Modifying the Code

In this part of the lab, we modify the example code given to us such that we can send and receive character arrays (strings). Both the client and server must be able to exchange strings, and the maximum length is assumed to be 32 bytes.
Both the client and server programs had a “sendChars” and “getChars” function added, and the code was modified such that it would still support integer data communication between the two devices as well as characters by means of warning of characters with a special integer.

The BBB was used as the server and the RPi was used as the client. Code was compiled inside of Eclipse and copied to each device using the Remote Systems plugin.

Code is available in the appendix for this lab.

Conclusion

It was difficult to look through every function used in the socket programs and document them. It wasn’t always clear how the function worked, and at least one of them (sizeof) wasn’t a typical function and required different research. I eventually realized that eclipse had several inbuilt tools for describing functions and also telling you which header file they were included by.

In summary, I used sockets first transmit and receive integers between two different embedded computers. Later, I upgraded the code to support sending and receiving character arrays. Jump to Index

Appendix – Code

Client.C

```c
/* A simple client program to interact with the myServer.c program on the
Raspberry.
myClient.c
D. Thiebaut
Adapted from http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html
The port number used in 51717.
This code is compiled and run on the Macbook laptop as follows:

    g++ -o myClient myClient.c
    ./myClient


*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <netdb.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <arpa/inet.h>


char myMessage[32] = "Hello from client!";
char recievedChars[32];

void error(char *msg) {
    perror(msg);
    exit(0);
}

void sendData( int sockfd, int x ) {
  int n;

  char buffer[32];
  sprintf( buffer, "%d\n", x );
  if ( (n = write( sockfd, buffer, strlen(buffer) ) ) < 0 )
      error("ERROR writing to socket");
  buffer[n] = '\0';
}

void sendChars (int sockfd, char * myData) {
      int myErr;
      char buffer[32];
      sprintf(buffer, myData);
      if ((myErr = write(sockfd, buffer, strlen(buffer))) < 0)
            error("ERROR: Failed to write string to socket");
      buffer[myErr] = '\0'; // Terminate with null character?
```

```
}


int getData( int sockfd ) {
  char buffer[32];
  int n;

  if ( (n = read(sockfd,buffer,31) ) < 0 )
       error("ERROR reading from socket");
  buffer[n] = '\0';
  return atoi( buffer );
}

void getChars(int sockfd) {
      int n;
      if ( (n = read(sockfd, recievedChars, sizeof(recievedChars)) ) < 0 )
                  error("ERROR reading from socket");
}

int main(int argc, char *argv[])
{
    int sockfd, portno = 51717, n;
    char serverIp[] = "192.168.1.32";
    struct sockaddr_in serv_addr;
    struct hostent *server;
    char buffer[256];
    int data;

    if (argc < 3) {
      // error( const_cast<char *>( "usage myClient2 hostname port\n" ) );
      printf( "contacting %s on port %d\n", serverIp, portno );
      // exit(0);
    }
    if ( ( sockfd = socket(AF_INET, SOCK_STREAM, 0) ) < 0 )
        error("ERROR opening socket");

    if ( ( server = gethostbyname( serverIp ) ) == NULL )
        error("ERROR, no such host\n");

    bzero( (char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy( (char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr, server-
>h_length);
    serv_addr.sin_port = htons(portno);
    if ( connect(sockfd,(struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
        error("ERROR connecting");


    for ( n = 0; n < 9; n++ ) {
      sendData( sockfd, n );
      data = getData( sockfd );
      printf("%d ->  %d\n",n, data );
    }
    // Put in "char" mode
    sendData(sockfd, -3);
    sendChars(sockfd, myMessage);

    getChars(sockfd);
    printf("Response: %s\n", recievedChars);
    // Stop the server
    sendData( sockfd, -2 );

    int r;
    if (r = close( sockfd ))
      error("Failed to close port!");
    return 0;
}
```

Server.c

```
/* A simple server in the internet domain using TCP.
myServer.c
D. Thiebaut
Adapted from http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html
The port number used in 51717.
This code is compiled and run on the Raspberry as follows:

    g++ -o myServer myServer.c
    ./myServer

The server waits for a connection request from a client.
The server assumes the client will send positive integers, which it sends back
multiplied by 2.
If the server receives -1 it closes the socket with the client.
If the server receives -2, it exits.
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

#define AF_INET 2
#define INADDR_ANY 0

char myMessage[32] = "Hello from server!"; // Filled in main's arguments
char recievedChars[32];

void error( char *msg ) {
      printf( "error\n");
  perror(  msg );
  exit(1);
}

int func( int a ) {
   return 2 * a;
}

void sendChars (int sockfd, char * myData) {
      int myErr;
      char buffer[32];
      sprintf(buffer, myData);
      if ((myErr = write(sockfd, buffer, strlen(buffer))) < 0)
            error("ERROR: Failed to write string to socket");
      buffer[myErr] = '\0'; // Terminate with null character?
}

void sendData( int sockfd, int x ) {
  int n;

  char buffer[32];
  sprintf( buffer, "%d\n", x );
  if ( (n = write( sockfd, buffer, strlen(buffer) ) ) < 0 ) {
    error("ERROR writing to socket");
  }
  buffer[n] = '\0';
}

void getChars(int sockfd) {
      int n;
      if ( (n = read(sockfd, recievedChars,
sizeof(recievedChars)) ) < 0 )
                  error("ERROR reading from
socket");
}

int getData( int sockfd ) {
  char buffer[32];
  int n;

  if ( (n = read(sockfd,buffer,31) ) < 0 )
    error("ERROR reading from socket");
  buffer[n] = '\0';
  return atoi( buffer );
}

int main(int argc, char *argv[]) {
     int sockfd, newsockfd, portno = 51717, clilen;
     char buffer[256];
     struct sockaddr_in serv_addr, cli_addr;
     int n;
     int data;

     printf( "using port #%d\n", portno );

     sockfd = socket(AF_INET, SOCK_STREAM, 0);
     if (sockfd < 0)
         error("ERROR opening socket");
     bzero((char *) &serv_addr, sizeof(serv_addr));

     serv_addr.sin_family = AF_INET;
     serv_addr.sin_addr.s_addr = INADDR_ANY;
     serv_addr.sin_port = htons( portno );
     if (bind(sockfd, (struct sockaddr *) &serv_addr,
              sizeof(serv_addr)) < 0)
       error("ERROR on binding");
     listen(sockfd,5);
     clilen = sizeof(cli_addr);
```

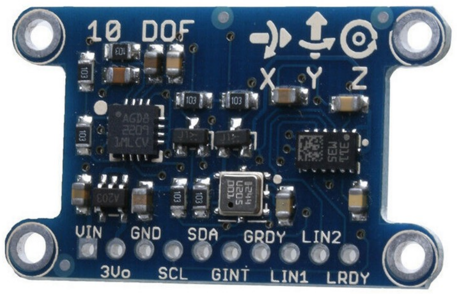


Image 1: Sensor Module

```
     //--- infinite wait on a connection ---
     while ( 1 ) {
        printf( "waiting for new client...\n" );
        if ( ( newsockfd = accept( sockfd, (struct sockaddr *) &cli_addr,
(socklen_t*) &clilen) ) < 0 )
            error("ERROR on accept");
        printf( "opened new communication with client\n" );

        int doChars = 0;
        while ( 1 ) {
             //---- wait for a number from client ---
             if (doChars > 0)
                   getChars(newsockfd);
             else
                   data = getData( newsockfd );

             if (data == -2)
                   break;
             else if (data == -3)
             {
                   doChars = 1;
                   data = 0;
                   printf("Ready for characters! \n");
                   continue; // goto next loop iteration
             }

             if (doChars == 1) {
                   printf("Rec: %s\n", recievedChars);
                   sendChars(newsockfd, myMessage);
                   doChars = 0; // stop receiving chars
                   continue; // goto next iteration
             }

             printf( "got %d\n", data );
             printf( "sending back %d\n", func(data));
                   //--- send new data back ---
                   sendData( newsockfd, func(data));

        }
        printf("Closing socket!");
        int n;
        if (n = close( newsockfd ))
            error("Failed to close socket!");

        //--- if -2 sent by client, we can quit ---
        if ( data == -2 )
          break;
     }
     return 0;
}
```

Lab 4

The objective of this part of the lab is to define and plan out the use of a sensor and actuator.

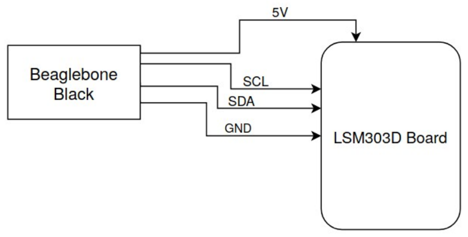
Part 1 – Sensor

The sensor I have chosen for this lab is Adafruit's LSM303D 3-axis accelerometer with 3-axis magnetometer. In addition to the accelerometer, there is a BMP180 pressure sensor and a L3G4200D 3-axis gyroscope on the same board. Drawing 1 illustrates the device hookup to the BBB.

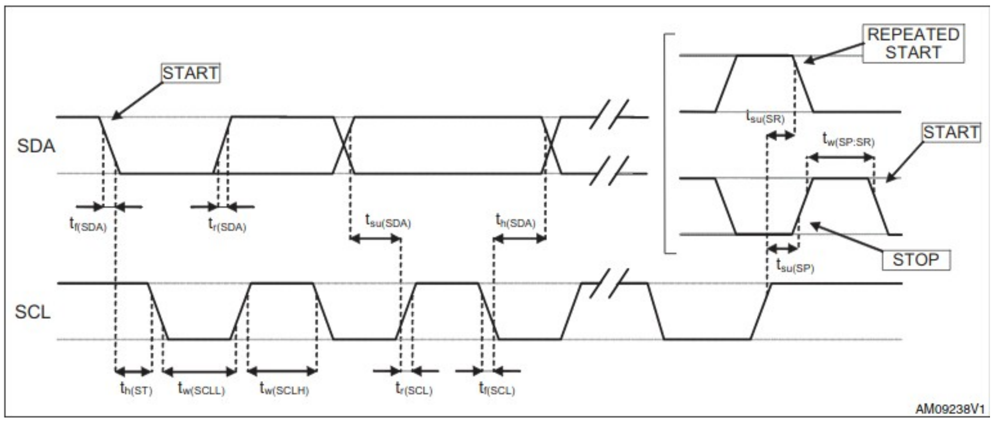
Connection

The LSM303D device supports both I2C and SPI communication. I will be using I2C in this lab because of previous experience with the protocol. The board includes the necessarily level shifting and will operate with the 3.3V GPIO pins on the BBB.

The device operates off of 5v, and will be supplied by the BBB's own power supply. The device only requires 110uA of current, which the BBB's power supply can easily handle. In combination with the other sensors on the board, the board should draw no more than 6.5mA, which is also allowable. Drawing 2 shows the I2C timing for the device.



Drawing 1: Sensor Hookup



Drawing 2: I2C Timing

Part 2- Actuator

The actuator I have chosen for this lab is a 28BYJ-48 stepper motor in combination with a ULN20003APG driver board.

Connection

The driver is not a step and direction driver. It is a seven channel darlington sink driver which is designed to interface a TTL chip with a high voltage (50V max) device such as a motor. The BBB will communicate with the device via four inputs (only four are needed for the stepper motor). The

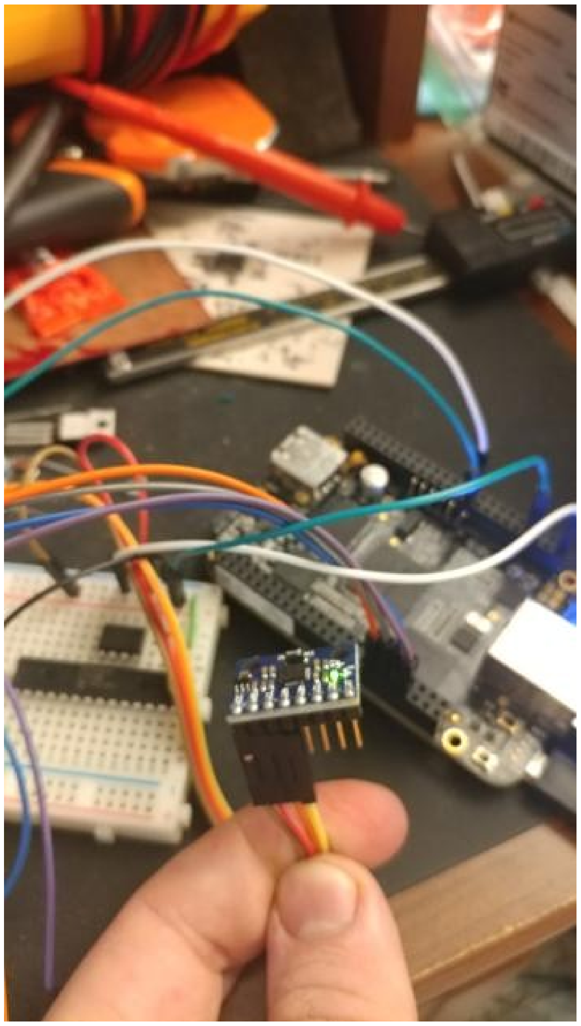
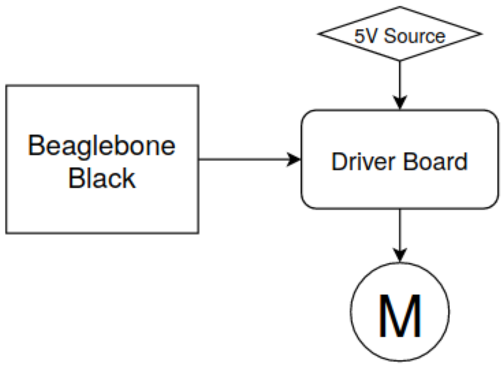


Image 3: Circuit & Mess

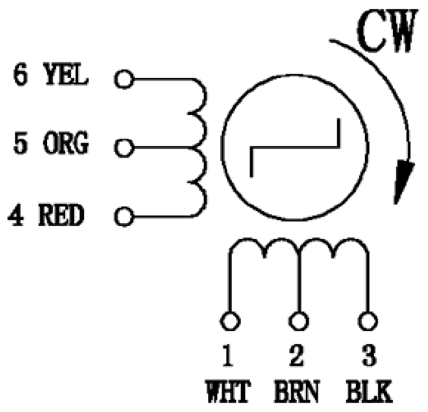


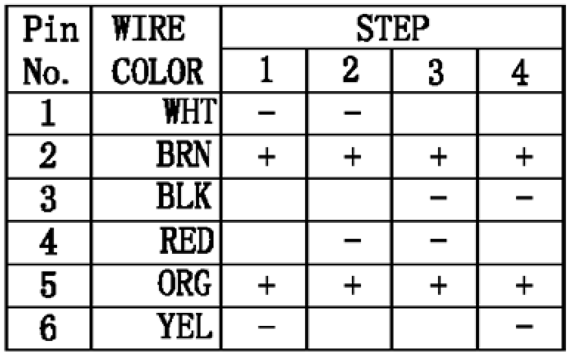
Drawing 3: Block Diagram

driver datasheet states the voltage input range is from 2.8v to 24 typically, which the BBB can supply. The current requirement is typically only 0.4mA per channel, which is allowable.

The motor is a 5Vdc device and it and the driver board will be powered from a bench-top adjustable power supply. The motor should require about 400mA max, which can both be supplied by the power supply and handled by the driver board (500mA max).

The motor is turned by a sequence of pulses to it by the driver, which is controlled by the BBB. The motor contains four windings which must be powered in a specific sequence to achieve clockwise or counter-clockwise direction.





Pin No.	WIRE COLOR	STEP 1	STEP 2	STEP 3	STEP 4
1	WHT	−	−		
2	BRN	+	+	+	+
3	BLK			−	−
4	RED		−	−	
5	ORG	+	+	+	+
6	YEL	−			−

Drawing 4: Stepper sequence

Conclusion

Researching how to use the stepper motor was one of the more challenging aspects of this lab. In the end however, in addition to noticing libraries available online, the driver architecture and pulse sequence for operation was determined. The choice of the actuator and sensor was to allow the pairing of them in the future for projects such as an automatically leveling table or balancing robot.

Jump to Index

Lab 5

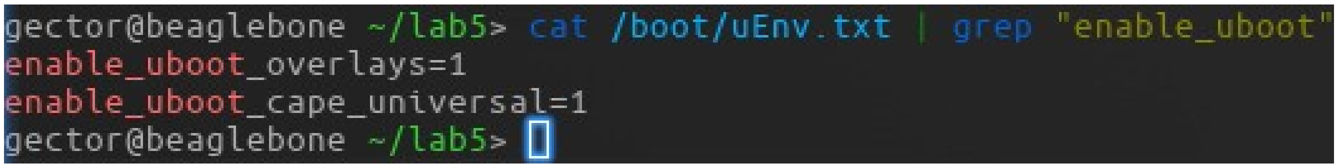
Introduction

This lab is supposed to be for setting up the circuitry and programming the sensor and actuator chosen and described in lab 4. The BeagleBone Black (BBB) single-board computer is used to operate both sub-modules. For the sensor, I am using an MPU-6050 Accelerometer/Gyroscope. Originally I was using a different module, from Adafruit, but it doesn't appear to work anymore. Due to time constraints, the MPU-6050 was another module I had on-hand and I didn't have to wait for a new module from Adafruit.

Part 1 – Sensor

The sensor has a 3.3V regulator and level shifter, and is powered by the BBB's 5V supply. It is entirely controlled and powered by the BBB. In addition to power and ground, it utilizes pins [P9, 19] and [P9, 20], which are the I2C2 SDA and SCL pins. I2C2 is the third I2C interface supplied by the BBB and can be utilized either in C code, or by the bash terminal using *i2c-utils*.

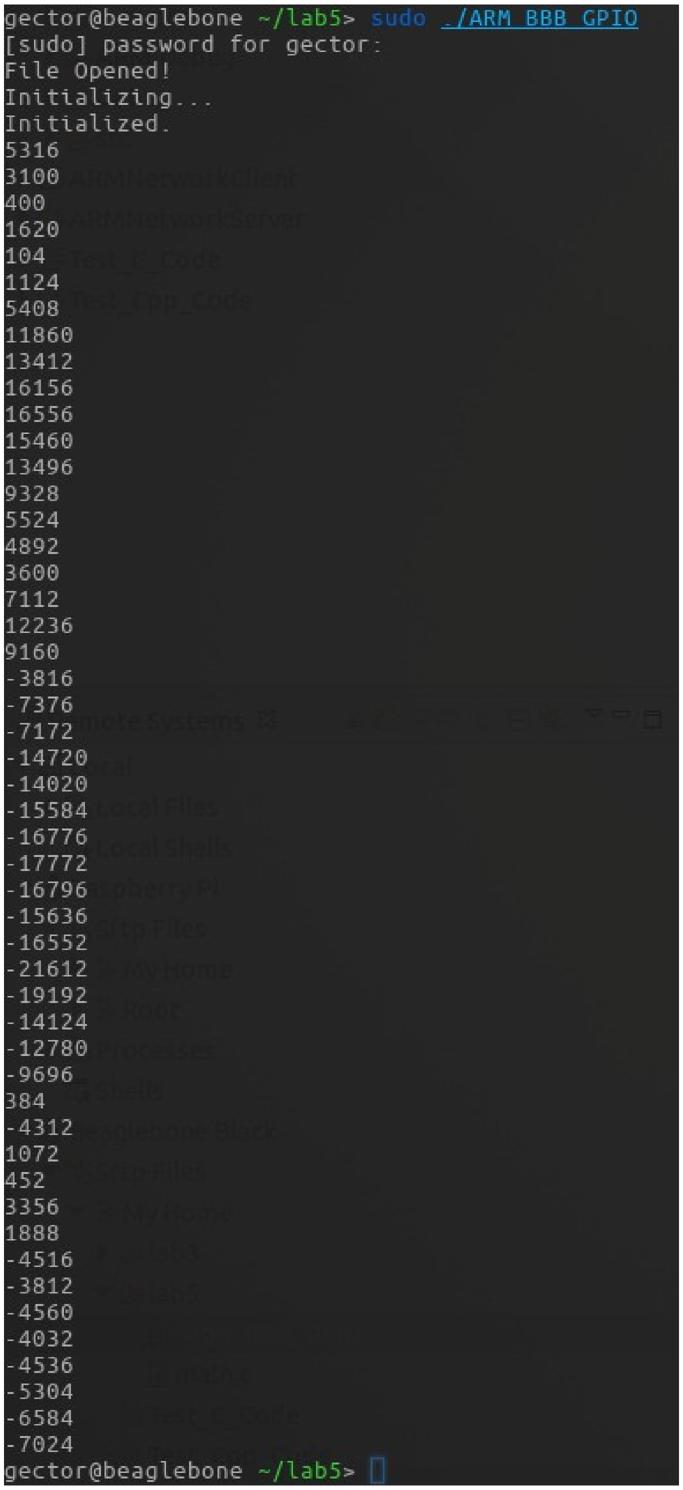
To make use of the GPIO pins of the BBB, the device tree overlays must be configured correctly. The C library I am using needs 'enable_uboot_cape_universal=1' in the device tree overlay config file. Luckily this is set by default on the Linux image I used.



```
gector@beaglebone ~/lab5> cat /boot/uEnv.txt | grep "enable_uboot"
enable_uboot_overlays=1
enable_uboot_cape_universal=1
gector@beaglebone ~/lab5> 
```

Capture 1: Proof that the device tree overlay is configured

For the I2C communication to the Accelerometer (MPU-6050), a standard Linux library called “linux/i2c.h” was used. This handles the i2C interface as a file and the use is as simple as reading and writing characters to a file after a ‘ioctl’ setup. As of this lab, the C code is in a single file which collects and prints X-axis accelerometer data over the period of a few seconds. The code is included in the appendix, and Capture 2 shows some accelerometer data read from the module.

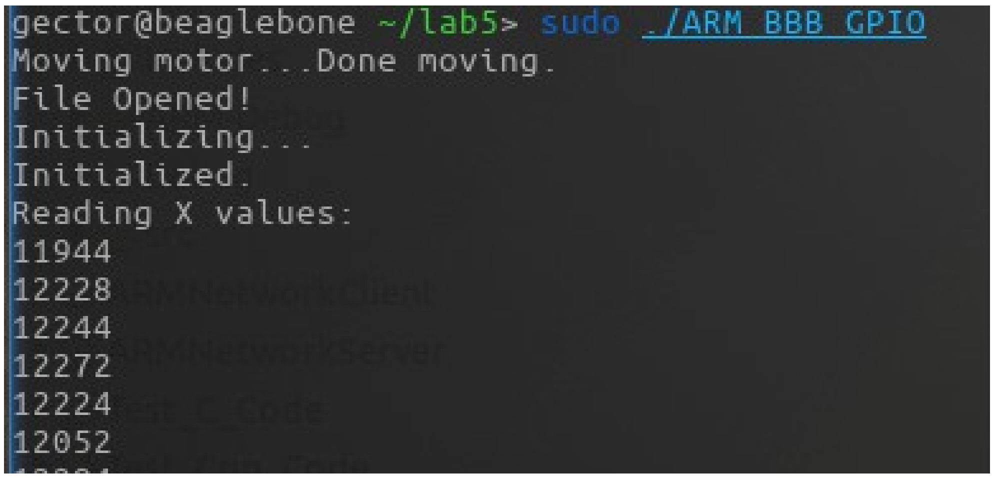


```
gector@beaglebone ~/lab5> sudo ./ARM_BBB_GPIO
[sudo] password for gector:
File Opened!
Initializing...
Initialized.
5316
3100
400
1620
104
1124
5408
11860
13412
16156
16556
15460
13496
9328
5524
4892
3600
7112
12236
9160
-3816
-7376
-7172
-14720
-14020
-15584
-16776
-17772
-16796
-15636
-16552
-21612
-19192
-14124
-12780
-9696
384
-4312
1072
452
3356
1888
-4516
-3812
-4560
-4032
-4536
-5304
-6584
-7024
gector@beaglebone ~/lab5>
```

Capture 2: X axis data printed to the terminal

Part 2 – Actuator

The Actuator uses the pins [P8, 8], [P8, 10], [P8, 12], [P8, 14]. The stepper motor is controlled by just four pins and a special sequence to rotate the motor forward or backward. The motor control board itself is grounded to the BBB, but it’s power supply is a variable power supply. The C library used is “iobb”, a great C library for manipulating the GPIO of the BBB in addition to other functions like SPI. Capture 3 shows the terminal output of the program. The code is documented and shown in the appendix. Image 4 shows the entire prototype setup with the sensor and stepper motor.



```
gector@beaglebone ~/lab5> sudo ./ARM_BBB_GPIO
Moving motor...Done moving.
File Opened!
Initializing...
Initialized.
Reading X values:
11944
12228
12244
12272
12224
12052
```

Capture 3: Terminal Output

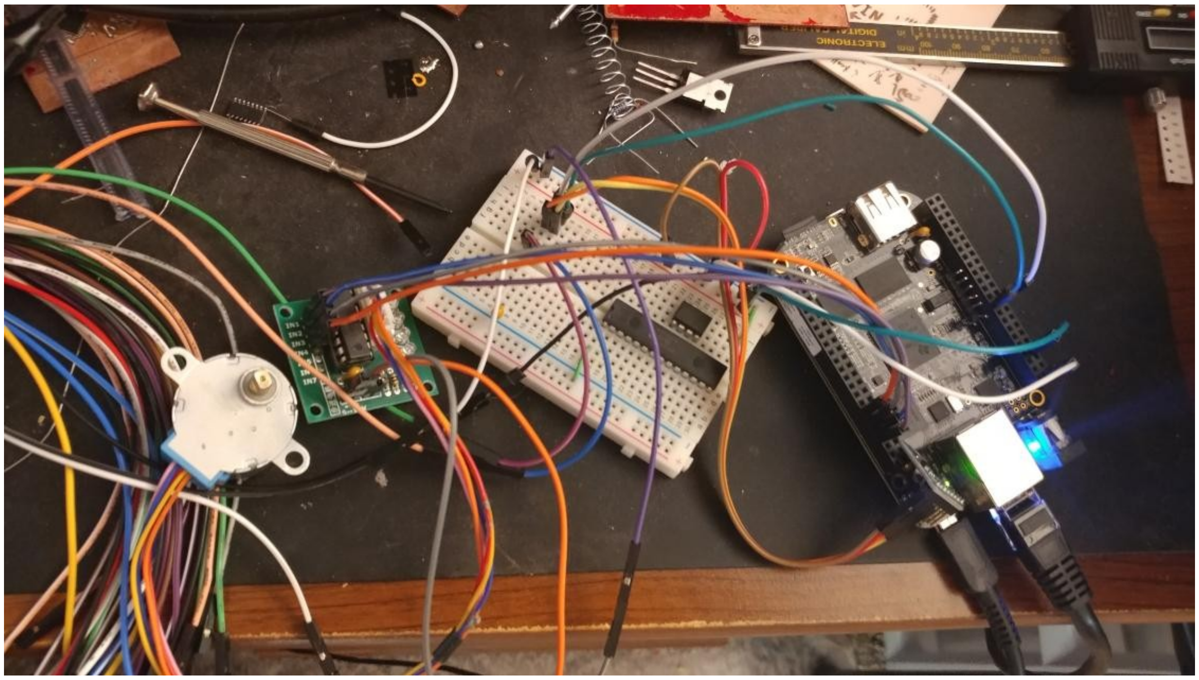


Image 4: Breadboarding prototype (ICs do nothing) Motor is shown on the left

Part 3 – Communication Protocol

The objective of this part of the lab is to outline a detailed communication protocol to implement in the next lab. The protocol will command the actuator to rotate and will have the ability to read sensor accelerometer values, gyroscope values, or temperature values.

The main program will be split into separate files for control of the actuator, of the the sensor, and of the networking handling.

Actuator Interface

The stepper takes 4096 steps to do a full rotation, 64 steps is 5.625 degrees.

STEPPER SET SPEED	– Sets the speed of the stepper in steps-per-millisecond
STEPPER CCW <STEPS>	– Turn a number of steps counter-clockwise
STEPPER CW <STEPS>	– Turn a number of steps clockwise
STEPPER BALANCE	– Will attempt to maximize the accelerometer's Z value (hold it upright) Other commands to the actuator will disable this.

Response type:

OK	– Means request was successfully executed
ERR	– Means there was an error while trying to process command

Sensor Interface

ACCEL READ X	– Returns the integer value for X-axis
ACCEL READ Y	– Returns the integer value for Y-axis
ACCEL READ Z	– Returns the integer value for Z-axis
TEMP READ	– Returns the temperature value in celsius
GYRO READ X	– Return the integer value for the gyroscope's X-Axis
GYRO READ Y	– Return the integer value for the gyroscope's Y-Axis
GYRO READ Z	– Return the integer value for the gyroscope's Z-Axis

Responses start with the command minus the "**READ**". Example responses:

ACCEL X 12280

ACCEL Y -12280

Conclusion

Initially it was greatly challenging to setup the I2C protocol. There was some configuration required in Linux, and before I could go get to C code I have to first learn how to use the Bash shell utilities for controlling the I2C bus. After I was able to initialize and read the device using the terminal I started work on the C program. Eventually, with much aid from a logic analyzer, the bugs were worked out. Initially the file was opened and closed in every function, resulting in odd errors related to file manipulation and excess code required to do file handling. This was later changed to have the main function be in charge of opening and closing the file, simply passing it's descriptor to the sub-functions.

In setting up the motor, a C library for manipulation of the IO had to be found and installed. Initially I had difficulty setting up the compiler and linker to recognize the library. Eventually the Eclipse IDE was set up to recognize the include, and after re-compiling the library for ARM (since it had been compile on my computer), the library was function. The device-tree-overlay settings were good by default and the code worked immediately.

Overall this was a great learning experience in both software and hardware. I learned how to do basic step and direction control of a stepper motor without a wrapper library, and how to use the GPIO of the BeagleBone Black. I learned how to use the MPU-6050 module as a slave I2C device and also some of the fundamentals of the Linux I2C system.

Jump to Index

Appendix

C Program

```c
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>
#include <linux/i2c.h>
#include <linux/i2c-dev.h>
#include <stdint.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <iobb.h>

#define MOTOR_PORT 8    // Connector #
#define MOTOR_DELAY 500 // uS

const char motorPins[4] = {8, 10, 12, 14};
const char CCW[8] = {0x09,0x01,0x03,0x02,0x06,0x04,0x0c,0x08};  //CouterClockWise
const char CW[8]= {0x08,0x0c,0x04,0x06,0x02,0x03,0x01,0x09};    //ClockWise

int adapter_nr = 2; /* probably dynamically determined */

const char filename[20] = "/dev/i2c-2";

// Function to simplify writing GPIO
void setPin(char port, char pin, uint8_t state) {
      if (state) // 0 = OFF, else = ON
            pin_high(port, pin);
      else
            pin_low(port, pin);

}

// Function for easily controlling the 4 GPIO pins
int writeNibbleToGPIO(char data) {
      // Write GPIO to the lower 4 bits of 'data'
      setPin(MOTOR_PORT, motorPins[0], 0x01 & data);
      usleep(MOTOR_DELAY);
      setPin(MOTOR_PORT, motorPins[1], 0x02 & data);
      usleep(MOTOR_DELAY);
      setPin(MOTOR_PORT, motorPins[2], 0x04 & data);
      usleep(MOTOR_DELAY);
      setPin(MOTOR_PORT, motorPins[3], 0x08 & data);
      usleep(MOTOR_DELAY);

      return 0;
}

// Step the motor through one sequence. (8 steps)
int stepMotor8(int numSteps, int dir) {
      if (numSteps <= 0)
            return -1;

      for (int x = 0; x < numSteps; x++) {
            if (dir) {
                  // forward
                  for (int i = 0; i < 8; i++) {
                        writeNibbleToGPIO(CW[i]);
                  }
            } else {
                  // backward
                  for (int i = 0; i < 8; i++) {
                        writeNibbleToGPIO(CCW[i]);
                  }

            }
      }

    // Turn off GPIO
    writeNibbleToGPIO(0x00);

      return 0;
}

int initAccel(int fileDesc) {
      // Register 0x6B - power management 1
      // Register 3B - AccelX Upper
      // Register 3C - AccelX Lower

      // Disable IC sleep mode
      char buf[2]; // Address + Data = 4
      buf[0] = 0x6B;
      buf[1] = 0x00; //0x43;
      if (write(fileDesc, buf, 2) != 2) {
            printf("Failed to write to device!\n");
      }


      buf[0] = 0x6B;
      buf[1] = 0x00; //0x43;
      if (write(fileDesc, buf, 2) != 2) {
            printf("Failed to write to device!\n");
      }

      // Init the i2c device and print out some accel data
      // set clock source
      // set fullscale gyro range
      // set fullscale accel range
      // set sleep disabled (power reg 1)
      return 0;
}

// Read two bytes, if sent the first address of an axis, it will return the axis
value.
int readAxis(int fileDescriptor, char regAddr){
      int retrievedValue;
      char readBytes[2];

      // Write our register address
      if (write(fileDescriptor, &regAddr, 1) != 1) {
            printf("Failed to write to device!\n");
      }
      // Makes use of the rolling register pointer inside the accelerometer
      read(fileDescriptor, readBytes, 2);
      retrievedValue = (int16_t)((readBytes[1] << 8) | readBytes[0]);
      return retrievedValue;
}


// Write a register over the I2C bus
int writeRegister(int fileDescriptor, char regAddr, char data) {
      char readBytes[1];
      char buf[2];
      buf[0] = regAddr;
      buf[1] = data;

      // Write to register
      if (write(fileDescriptor, buf, 2) != 1) {
            printf("Failed to write to device!\n");
            return -1;
      }

      return 0;
}

// Read a register from the I2C device
int readRegister(int fileDescriptor, char regAddr) {
      int retrievedValue;
      char readBytes[1];
      char buf[1];
      buf[0] = regAddr;

      // Write our register address
      if (write(fileDescriptor, buf, 1) != 1) {
            printf("Failed to write to device!\n");
      }

      // Read value
      read(fileDescriptor, readBytes, 1);
      retrievedValue = (int8_t)(readBytes[0]);

      return retrievedValue;
}

int main(int argc, char *argv[]) {
      int file = open(filename, O_RDWR);

      // Initialize GPIO and turn on a pin..
    iolib_init();
    iolib_setdir(8, 8, DigitalOut);
    iolib_setdir(8, 10, DigitalOut);
    iolib_setdir(8, 12, DigitalOut);
    iolib_setdir(8, 14, DigitalOut);


    // Step the stepper driver 180 degrees
    // with step8, it requires 512 to rotate 360 degrees
    printf("Moving motor...");
    stepMotor8(256, 0); // '0' direction means counter clockwise.
    printf("Done moving.\n");

      // Specify device address for the file access
      if (ioctl(file, I2C_SLAVE, 0x68) < 0) {
            printf("Error setting address");
      }

      // Verify the file opened successfully.
      if (file < 0) {
            printf("Error opening file\n");
            return -1;
      } else {
            printf("File Opened!\n");
      }

      // Initialize the accelerometer
      printf("Initializing...\n");
      initAccel(file);
      printf("Initialized!\n");

      // Print out 50 X values
      int xVal;
      printf("Reading X values: \n");
      for (int x = 0; x < 50; x++) {
            xVal = readAxis(file, 0x3C);
            printf("%d\n", xVal);
            usleep(100000);
      }

      close(file);
      return 0; // Success!
}
```

Lab 6

Introduction

The object of lab 6 is to implement a client and server system for controlling the BeagleBone Black and it's peripherals. Both the sensor and actuator are available for use over the network and the BBB provides a text-based command system for reading and writing data.

Part 1 – The Protocol

In this part of the lab, the actual protocol is outlined and written in a separate document to distribute to other classmates so that they can also use my devices. This file includes all of the technical information necessary to control and read from the BBB. The content of the file is included in the appendix.

Essentially, this will serve as a reference for other people attempting to network with my BBB. It will describe the command syntax and the allowable range of inputs. Additionally, it describes the response format and how to read sensor data.

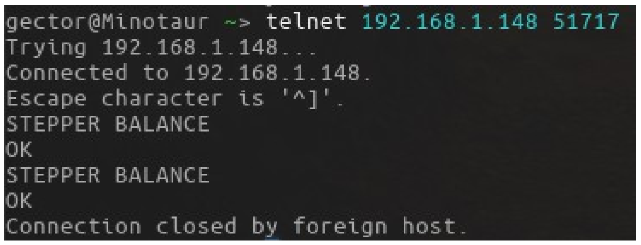
Part 2 – The Server

In this part of the lab, the sensor and actuator program for the BBB is modified to support networking. The program originally created in lab 5 was split into separate files for the motor, sensor, and networking functions.

The networking code was completely rewritten to be more robust and for the sake of learning more about socket networking. The new system uses polling to detect when data is ready to be received, when a new client connects, and when a client disconnects. When a data event is discovered, the received data is moved into a dynamic hash table (a dictionary). The command dictionary is used to dynamically process commands whenever the main program is available. Commands consist of two parts; the client that sent the command, and the text itself. Once a command is processed and replied to, it's discarded and the memory for it is freed.

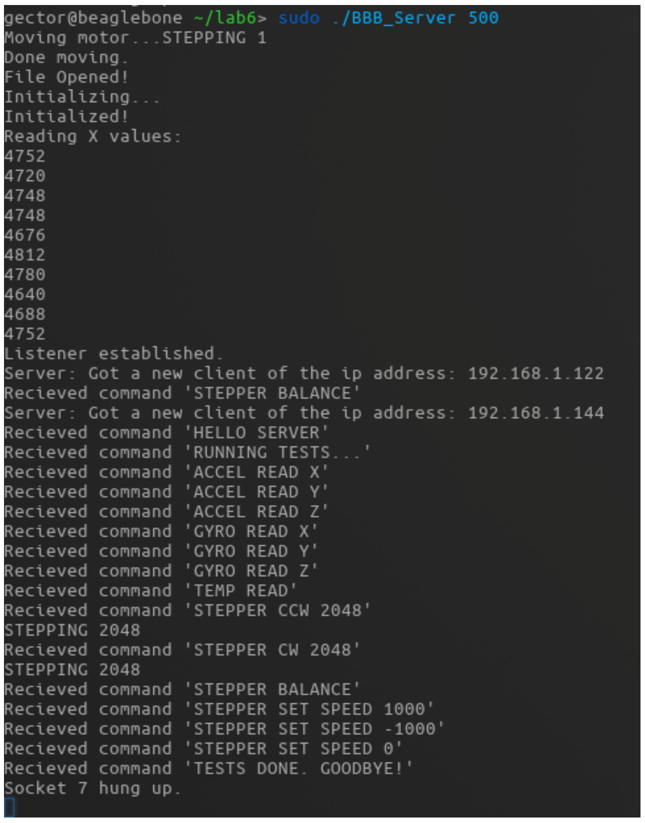
Because the BBB is driving a stepper motor directly without a sub-module, it has to stop what it is doing to send steps to the driver. By using code that won't stop and wait for events that may or may not happen, the program is free to continue running the motor even while being able to handle network clients.

Captures 4, 5, and 6 show the client and server communicating with each other. In addition to the raspberry pi connecting to the BBB, a linux PC is *telnet'd* into the BBB as well. Both the RPi and the PC can submit commands to the server at the same time, and both commands will be dealt with. If you look closely you can spot a "STEPPER BALANCE" command that the raspberry pi terminal never sent, but was in fact sent from the Linux PC.



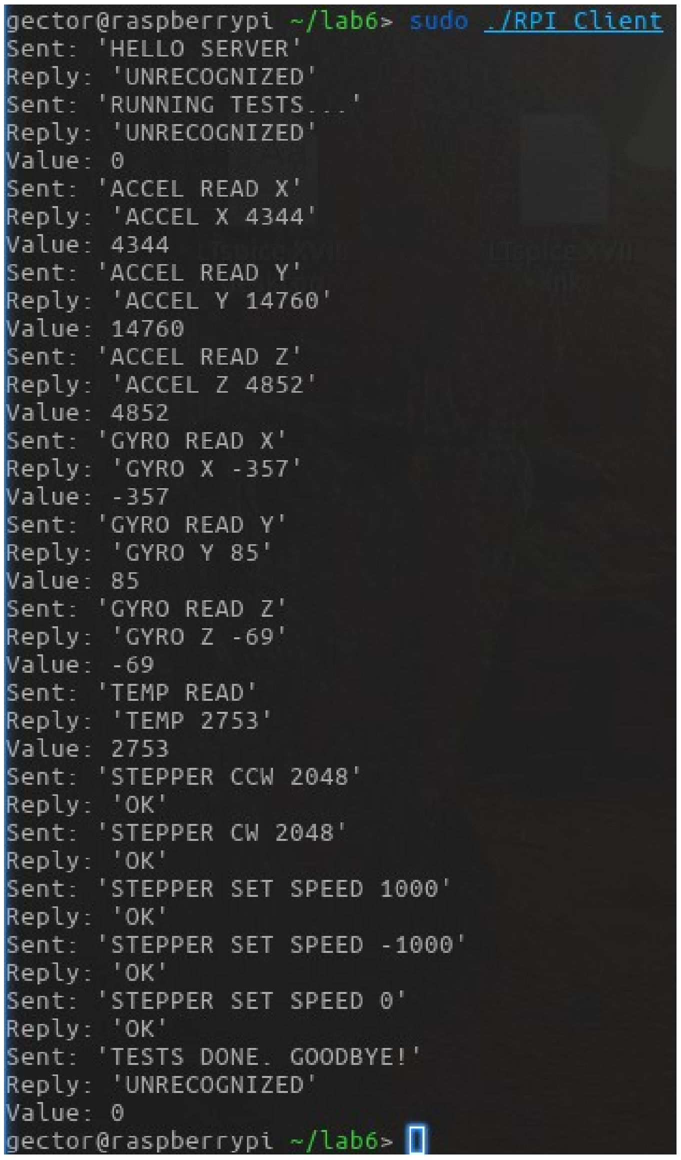
```
gector@Minotaur ~> telnet 192.168.1.148 51717
Trying 192.168.1.148...
Connected to 192.168.1.148.
Escape character is '^]'.
STEPPER BALANCE
OK
STEPPER BALANCE
OK
Connection closed by foreign host.
```

Capture 5: Telnet session from linux PC



```
gector@beaglebone ~/lab6> sudo ./BBB_Server 500
Moving motor...STEPPING 1
Done moving.
File Opened!
Initializing...
Initialized!
Reading X values:
4752
4720
4748
4748
4676
4812
4780
4640
4688
4752
Listener established.
Server: Got a new client of the ip address: 192.168.1.122
Recieved command 'STEPPER BALANCE'
Server: Got a new client of the ip address: 192.168.1.144
Recieved command 'HELLO SERVER'
Recieved command 'RUNNING TESTS...'
Recieved command 'ACCEL READ X'
Recieved command 'ACCEL READ Y'
Recieved command 'ACCEL READ Z'
Recieved command 'GYRO READ X'
Recieved command 'GYRO READ Y'
Recieved command 'GYRO READ Z'
Recieved command 'TEMP READ'
Recieved command 'STEPPER CCW 2048'
STEPPING 2048
Recieved command 'STEPPER CW 2048'
STEPPING 2048
Recieved command 'STEPPER BALANCE'
Recieved command 'STEPPER SET SPEED 1000'
Recieved command 'STEPPER SET SPEED -1000'
Recieved command 'STEPPER SET SPEED 0'
Recieved command 'TESTS DONE. GOODBYE!'
Socket 7 hung up.
```

Capture 6: Server Execution

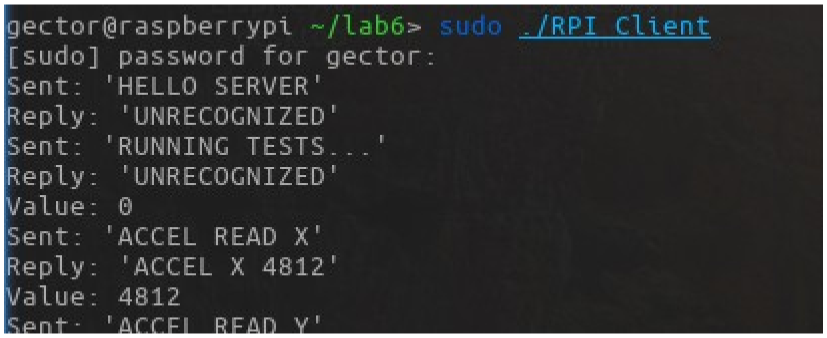


```
gector@raspberrypi ~/lab6> sudo ./RPI_Client
Sent: 'HELLO SERVER'
Reply: 'UNRECOGNIZED'
Sent: 'RUNNING TESTS...'
Reply: 'UNRECOGNIZED'
Value: 0
Sent: 'ACCEL READ X'
Reply: 'ACCEL X 4344'
Value: 4344
Sent: 'ACCEL READ Y'
Reply: 'ACCEL Y 14760'
Value: 14760
Sent: 'ACCEL READ Z'
Reply: 'ACCEL Z 4852'
Value: 4852
Sent: 'GYRO READ X'
Reply: 'GYRO X -357'
Value: -357
Sent: 'GYRO READ Y'
Reply: 'GYRO Y 85'
Value: 85
Sent: 'GYRO READ Z'
Reply: 'GYRO Z -69'
Value: -69
Sent: 'TEMP READ'
Reply: 'TEMP 2753'
Value: 2753
Sent: 'STEPPER CCW 2048'
Reply: 'OK'
Sent: 'STEPPER CW 2048'
Reply: 'OK'
Sent: 'STEPPER SET SPEED 1000'
Reply: 'OK'
Sent: 'STEPPER SET SPEED -1000'
Reply: 'OK'
Sent: 'STEPPER SET SPEED 0'
Reply: 'OK'
Sent: 'TESTS DONE. GOODBYE!'
Reply: 'UNRECOGNIZED'
Value: 0
gector@raspberrypi ~/lab6>
```

Capture 4: Client Execution

Part 3 – The Client

In this part of the lab, a program is created to allow the Raspberry Pi (RPi) to act as a client of the BeagleBone Black (BBB). The code was re-written similarly to the server; it uses standard socket networking functions and paradigms.



```
gector@raspberrypi ~/lab6> sudo ./RPI_Client
[sudo] password for gector:
Sent: 'HELLO SERVER'
Reply: 'UNRECOGNIZED'
Sent: 'RUNNING TESTS...'
Reply: 'UNRECOGNIZED'
Value: 0
Sent: 'ACCEL READ X'
Reply: 'ACCEL X 4812'
Value: 4812
Sent: 'ACCEL READ Y'
```

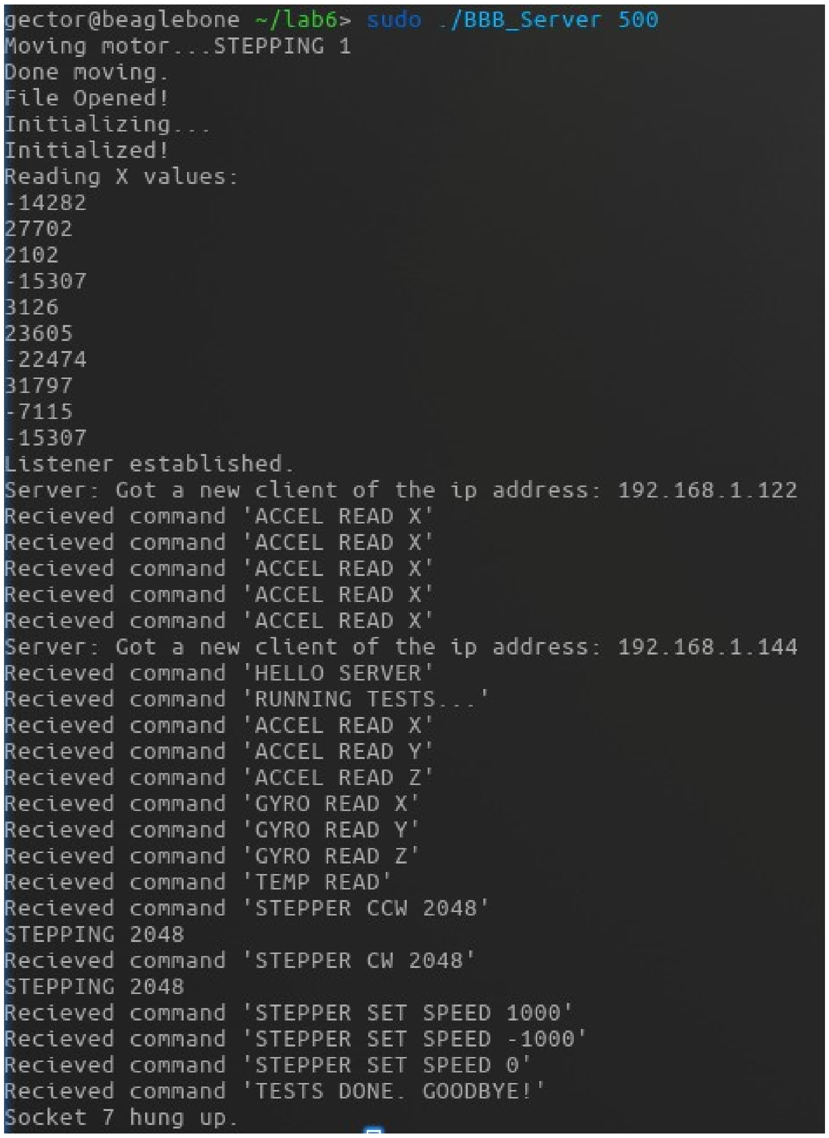
Capture 7: RPi Client

Part 4 – Test and Integration

In this part of the lab, the code from part 3 (the client) is used to test the protocol and connection methods. Essentially, every command is fired at the server and the response is read back and printed.

Shown in capture 8 is the server's terminal output while the client is communicating with it. Notice there being two clients, the RPi is the '.144' address and the Linux PC is the '.122' address.

The server will run for as many seconds are entered in the command line, in this case it runs for '500' seconds. By default, it will shut down after 20 seconds.



```
gector@beaglebone ~/lab6> sudo ./BBB_Server 500
Moving motor...STEPPING 1
Done moving.
File Opened!
Initializing...
Initialized!
Reading X values:
-14282
27702
2102
-15307
3126
23605
-22474
31797
-7115
-15307
Listener established.
Server: Got a new client of the ip address: 192.168.1.122
Recieved command 'ACCEL READ X'
Recieved command 'ACCEL READ X'
Recieved command 'ACCEL READ X'
Recieved command 'ACCEL READ X'
Recieved command 'ACCEL READ X'
Server: Got a new client of the ip address: 192.168.1.144
Recieved command 'HELLO SERVER'
Recieved command 'RUNNING TESTS...'
Recieved command 'ACCEL READ X'
Recieved command 'ACCEL READ Y'
Recieved command 'ACCEL READ Z'
Recieved command 'GYRO READ X'
Recieved command 'GYRO READ Y'
Recieved command 'GYRO READ Z'
Recieved command 'TEMP READ'
Recieved command 'STEPPER CCW 2048'
STEPPING 2048
Recieved command 'STEPPER CW 2048'
STEPPING 2048
Recieved command 'STEPPER SET SPEED 1000'
Recieved command 'STEPPER SET SPEED -1000'
Recieved command 'STEPPER SET SPEED 0'
Recieved command 'TESTS DONE. GOODBYE!'
Socket 7 hung up.
```

Capture 8: Server output while client runs tests

Conclusion

Overall, the socket programming was highly educational and will certainly provide great utility in the future. It also leaves plenty of room for adding features in the existing code, it would be possible run more than one actuator and talk to more than one sensor. On average, the CPU usage on the BBB remains at less than 4%.

For dedicated driving scenarios like this one, the BBB's CPU has two sub-cores called Programmable Real-time Units (PRUs). These can be programmed and used similarly to any high performance microcontroller, and would make ideal sub-cores for tasks like running a stepper motor. Unfortunately, due to time (and energy) constraints, I did not implement a method of programming them for my uses. There are libraries and tutorials available for them however, and for more complicated programs they would be the perfect solution.

In conclusion, after this lab I have the source code and the skills to do basic socket networking with an ARM computer, in addition to controlling an actuator and reading from a sensor. I've also learned a lot about the C programming language and the Eclipse IDE along the way. I should now be able to communicate with other kinds of motor drivers, other I2C devices, and other networked computers.

Jump to Index

Appendix

Protocol

Name:	**Zachary Whitlock**
Email:	**zachary.whitlock@oit.edu**
BBB IP Address:	**192.168.1.148**

Actuator Interface

Command Format

STEPPER SET SPEED < -2000 to +2000, 0 to stop, unit = steps/sec>
STEPPER CCW <steps, 4096 steps per rotation>
STEPPER CW <steps, 4096 steps per rotation>
STEPPER BALANCE

Note: Spaces are required

Example commands from client

STEPPER SET SPEED 1000 -- Tells the stepper motor to rotate clockwise at 1000 steps per second.

STEPPER CCW 2048	-- Turn 180 degrees counter-clockwise, goes as fast as possible.
STEPPER CW 2048	-- Turn 180 degrees clockwise, goes as fast as possible.
STEPPER BALANCE	-- Stepper will attempt to balance the accelerometer. Command is a toggle.

Example response from the server

OK	– Will always reply "OK" after done moving the stepper, or after a successful stepper command.

Sensor Interface

Command Format

<ACCEL/TEMP/GYRO> READ <X/Y/Z – do not specify axis for TEMP>

Note: Spaces are required

Example commands from client

ACCEL READ X	-- Reads the voltage on ADC channel 0
GYRO READ X	-- Reads the X axis gyroscope.
TEMP READ	-- Reads the temperature in Celcius * 100

Example response from the server

ACCEL X 12345	-- Number ±32768 = ±2g**,** '12345' = 0.753g
GYRO Z -542	-- Number ±32768 = ±250°/s**,** '-542' = -4.137°/s
TEMP 2790	-- Equivalent to 27.90C degrees.

Messages between the Client and Server

CLOSE	-- Closes the TCP/IP socket. This commend should be issued by the client. The server will respond with CLOSE.

Code

Because there are over 900 lines of code and 10 different files with conflicting names, the code can be found in a gitlab repository here: https://gitlab.com/CaptainGector/bbb_server, and here: https://gitlab.com/CaptainGector/rpi_client.

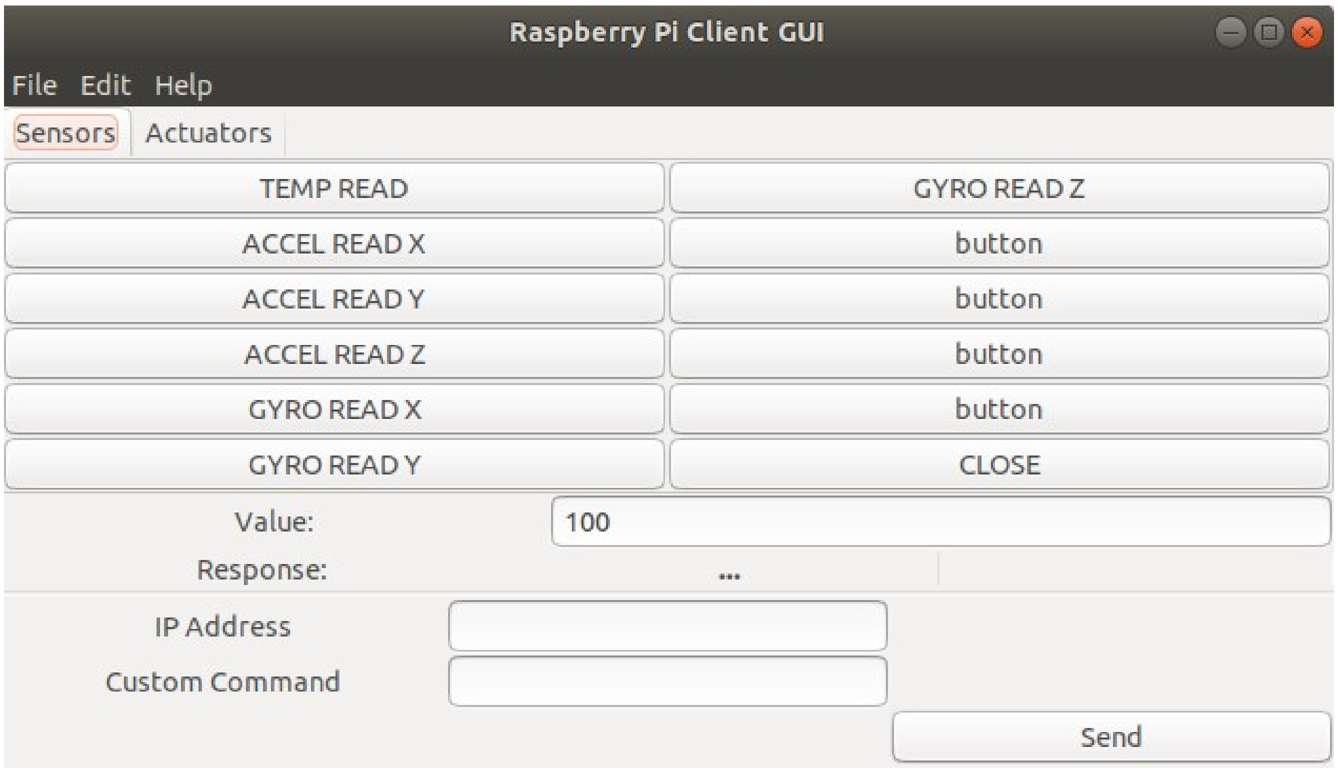
Lab 7

Introduction

The objective of this lab is to design a GUI (Graphic User Interface) using a piece of software called Glade. The GUI will be for sending commands to the BeagleBone Black from the Raspberry Pi and it's touch screen.

Part 1 – GUI Design

The first part of this lab is the GUI design itself. The user interface of the program was designed in a programm called Glade, which allows you to describe GTK-3 applications easily without spelling out everything in C code. The goal of the GUI is to support 12 sensors and 12 actuators, and callbacks were used to hook C code into the GUI's button system. Capture 9 shows the GUI in it's mostly finished form, for lab 7. It will undergo revisions to connect to more than my BeagleBone Black, and may include additional functionality.



Capture 9: The GUI as Created by Glade

Part 2 – Testing GUI on RPi

The second part of this lab is to write a program that controls the GUI and handles user input like button presses. The code was actually written on my Linux computer and a post-build command was used to run a script which copied the development files to the RPi. On the RPi, I used a Makefile to compile and run the GUI program. This worked quite well and was easier to set up than cross-compiling GTK-3 and it's associated libraries, which I also attempted.

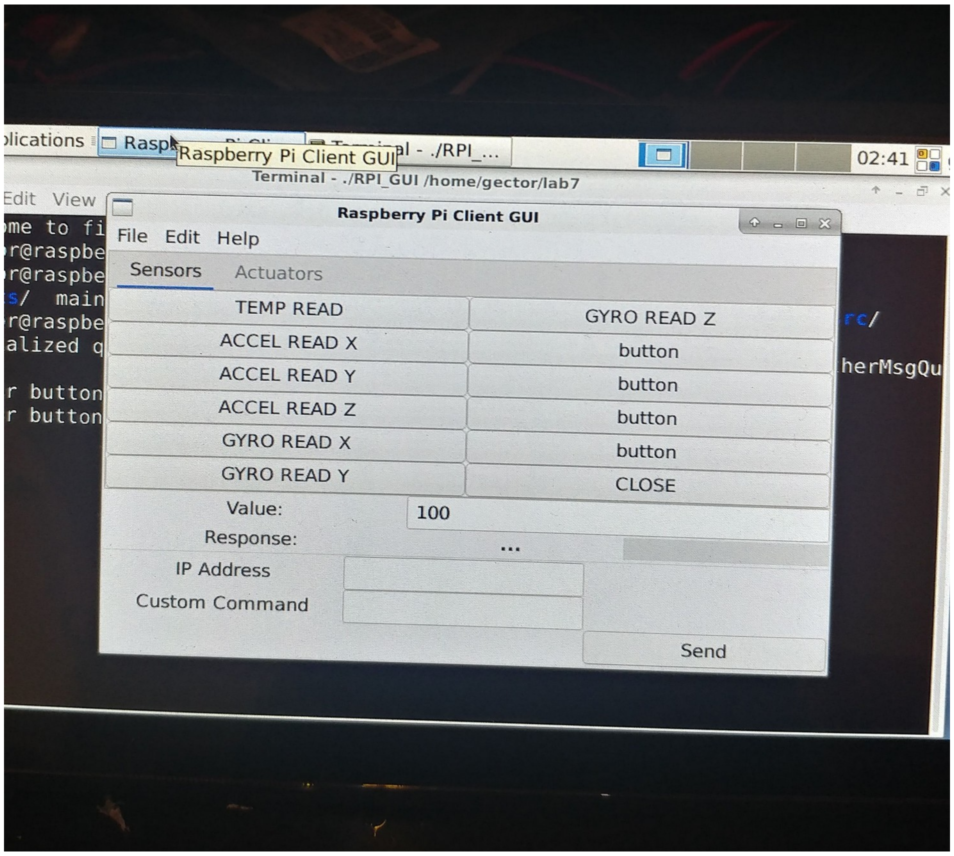
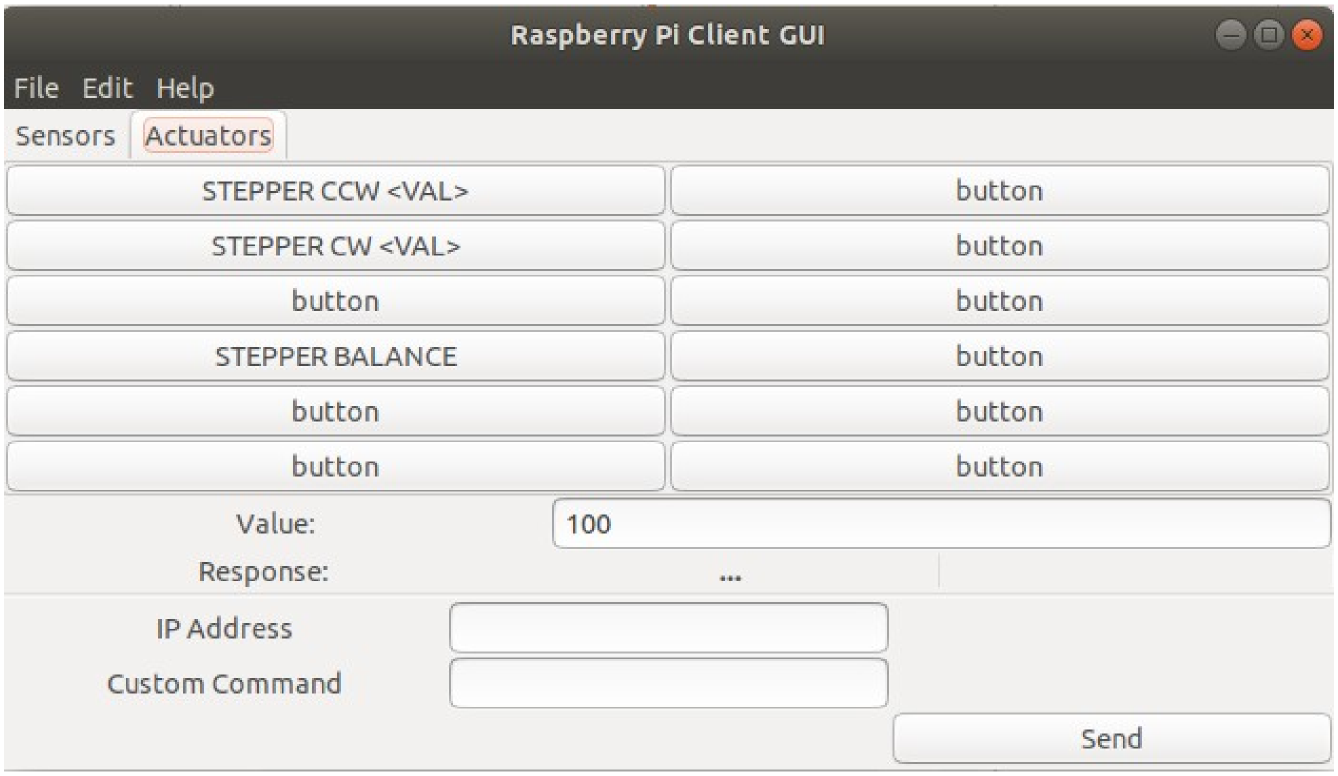


Image 5: The GUI running on the Raspberry Pi

Image 5 shows the GUI running on the Raspberry Pi after the code was compiled and executed. The text in the terminal behind the GUI reads "Sensor button clicked", which indicates that one of the 12 sensor buttons was clicked on.

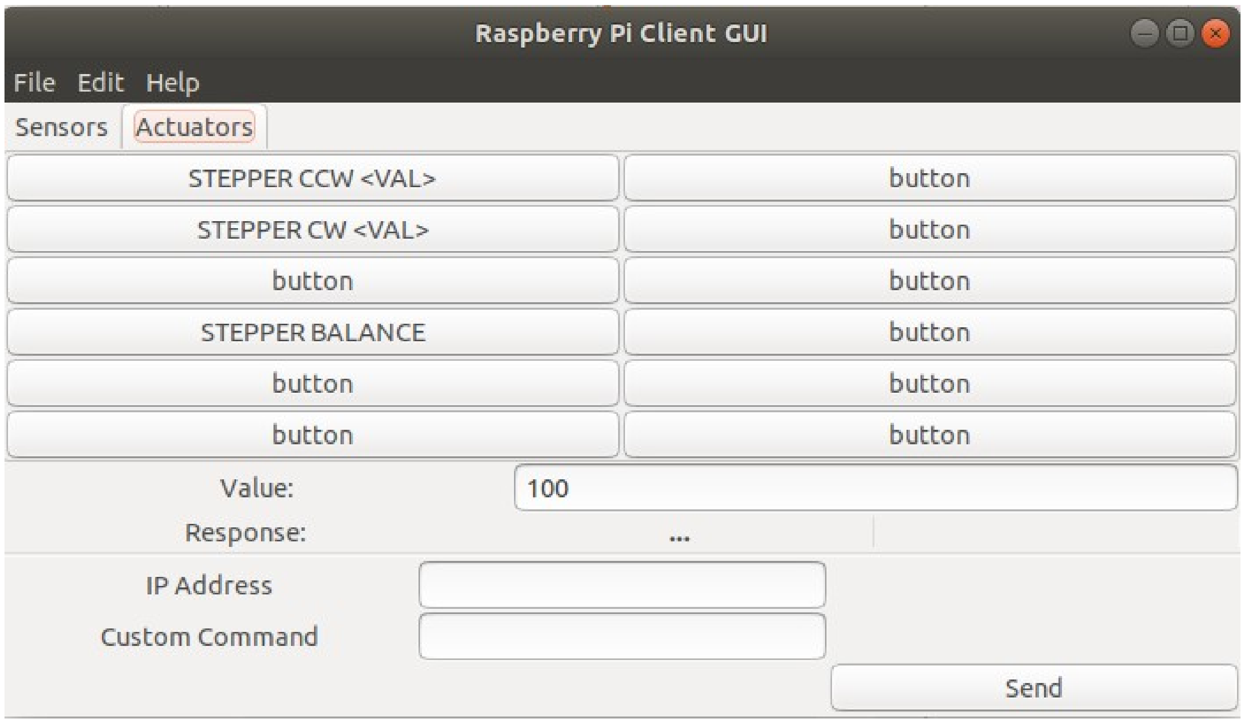


Value: 100
Response: ...
IP Address
Custom Command

Capture 10: Actuator button panel

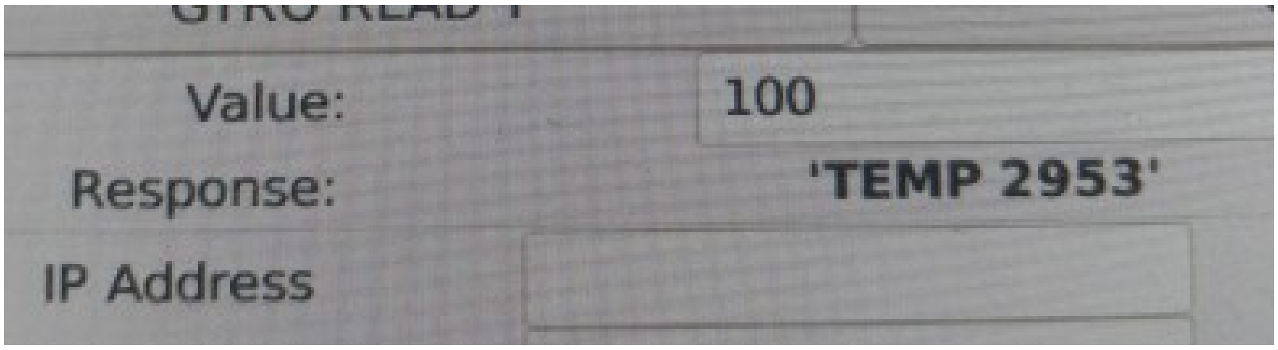
Part 3 – GUI Interfacing to client.c

The purpose of this part of the lab is to allow the GUI to utilize the connectivity of the client.c application previously written for the RPi.



Capture 11: Actuator button panel

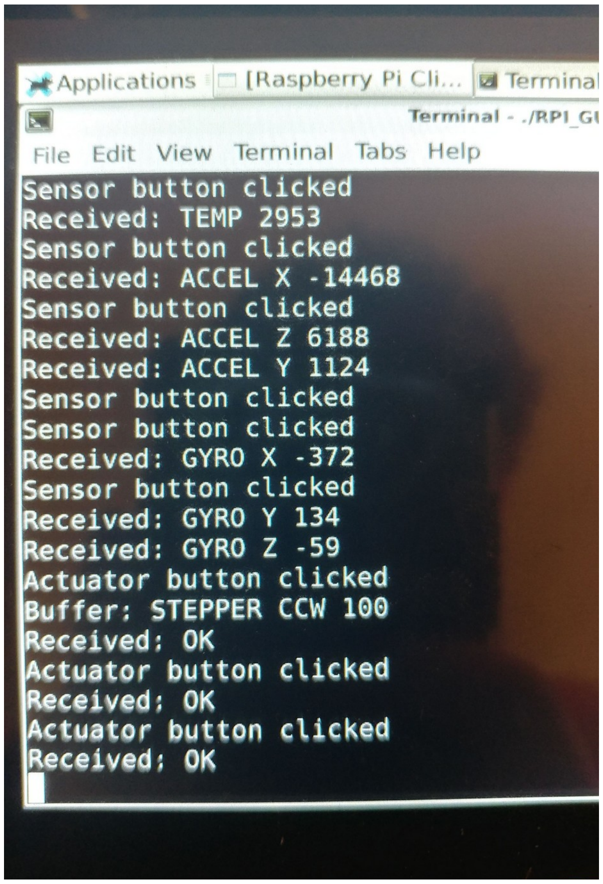
The value field is read upon an actuator button press, such as ‘STEPPER CCW <VAL>’. The ‘<VAL>’ text will replaced with whatever is in the “Value” entry box, and then the command sent to the BBB. Setting commands is as simple as setting the label of the button.



Capture 12: Proof the GUI works

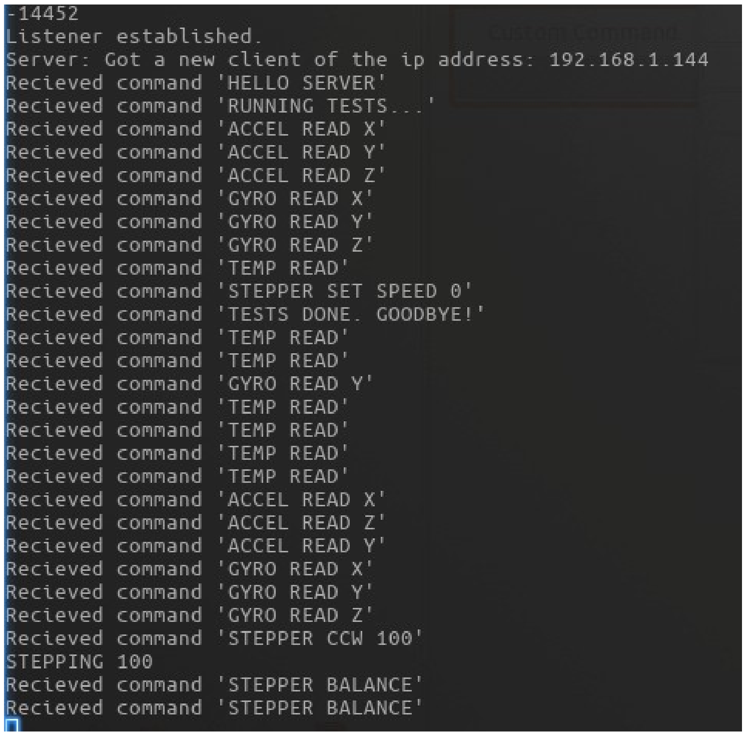
As of right now, the IP address for my BBB is hard-coded, and so the 'IP Address' entry box has no purpose. The 'Custom Command' entry box allows you to send whatever command you want to the server, however. The 'Send' button is only for the custom command feature, the buttons automatically send commands.

Commands and replies are relayed/proxied through the client program previously written for the RPi. The client program was modified to read message queues, which are used to pass information between the GUI and client which are both running on the RPi. Replies from the BBB are relayed back to the GUI which will display them in the label to the left of 'Response', which defaults to '...'.



```
Sensor button clicked
Received: TEMP 2953
Sensor button clicked
Received: ACCEL X -14468
Sensor button clicked
Received: ACCEL Z 6188
Received: ACCEL Y 1124
Sensor button clicked
Sensor button clicked
Received: GYRO X -372
Sensor button clicked
Received: GYRO Y 134
Received: GYRO Z -59
Actuator button clicked
Buffer: STEPPER CCW 100
Received: OK
Actuator button clicked
Received: OK
Actuator button clicked
Received: OK
```

Capture 13: Capture of the GUI terminal during connection



```
-14452
Listener established.
Server: Got a new client of the ip address: 192.168.1.144
Recieved command 'HELLO SERVER'
Recieved command 'RUNNING TESTS...'
Recieved command 'ACCEL READ X'
Recieved command 'ACCEL READ Y'
Recieved command 'ACCEL READ Z'
Recieved command 'GYRO READ X'
Recieved command 'GYRO READ Y'
Recieved command 'GYRO READ Z'
Recieved command 'TEMP READ'
Recieved command 'STEPPER SET SPEED 0'
Recieved command 'TESTS DONE. GOODBYE!'
Recieved command 'TEMP READ'
Recieved command 'TEMP READ'
Recieved command 'GYRO READ Y'
Recieved command 'TEMP READ'
Recieved command 'TEMP READ'
Recieved command 'TEMP READ'
Recieved command 'TEMP READ'
Recieved command 'ACCEL READ X'
Recieved command 'ACCEL READ Z'
Recieved command 'ACCEL READ Y'
Recieved command 'GYRO READ X'
Recieved command 'GYRO READ Y'
Recieved command 'GYRO READ Z'
Recieved command 'STEPPER CCW 100'
STEPPING 100
Recieved command 'STEPPER BALANCE'
Recieved command 'STEPPER BALANCE'
```

Capture 14: Capture of the BBB server during connection

Captures 13 and 14 show an example of the backend in use. Capture 13 is the output from the GUI program, and Capture 14 is the output of the BBB's server. Capture 12 (above) shows the GUI having received data from the trail of programs starting at the BBB.

Conclusion

In this lab I learned how to use GTK-3 to make some very nice graphical interfaces to programs written in different languages. Glade can be used for more than just C, and is commonly used with Python. Additionally, I learned how the POSIX message queues worked and how to compile applications using them on both ARM and x86 processors. I also learned how to set up a Raspberry Pi with an LCD with a custom resolution and a capacitive touchscreen. This was a highly educational lab with a lot of relevant experience for future projects or employments.

Jump to Index

Appendix

Code

Because there are over 900 lines of code and 10 different files with conflicting names, the code can be found in a gitlab repository here: https://gitlab.com/CaptainGector/bbb_server, and here: https://gitlab.com/CaptainGector/rpi_client, and here:

https://gitlab.com/CaptainGector/rpi_gui.

It should be noted that one would have to go back through commit history to view the changes between labs, as there isn't much distinguishing the code otherwise.

I've included the GUI code for this lab since it's not as large as the other programs

Code – main.c

```c
/*
 * main.c
 *
 *  Created on: Nov 24, 2020
 *      Author: Zachary Whitlock
 *
 * Creates a GTK3 GUI from a glade-generated file, and attaches
 * handlers.
 *
 */
#include <stdint.h>
#include <stdio.h>
#include <gtk/gtk.h>

#include "msgQueue.h"

// Global items of interest
GtkWidget   *lbl_response;
GtkEntry    *entry_command;
GtkEntry    *entry_value;

GtkTextBuffer *xCommand;
```

```c
// Fires when an actuator button is pressed
void Actuator_clicked_cb(GtkButton *button, gpointer user_data) {
      printf("Actuator button clicked\n");
      char msgBuf[MAX_MSG_SIZE];
      char msgBuf2[MAX_MSG_SIZE];
      char *subStr;

      const gchar *label = gtk_button_get_label(button);
      strcpy(msgBuf, label);
      subStr = strstr(msgBuf, "<VAL>");
      if (subStr != NULL) {

            // Strips the "<VAL>" from the string and places the result in msgbuf2
            strncpy(msgBuf2, msgBuf, subStr-msgBuf);
            msgBuf2[subStr-msgBuf] = 0; // null terminator

            // Adds the content of the value entry into the string
            const gchar *entry_text;
            guint16 entry_size;
            entry_text = gtk_entry_get_text(entry_value);
            entry_size = gtk_entry_get_text_length(entry_value);
            sprintf(msgBuf, "%s%s", msgBuf2, entry_text);

            printf("Buffer: %s\n", msgBuf);
      }

      // Send the complete command
      if (sendToOther(msgBuf, MAX_MSG_SIZE) != 0) {
            printf("Failed to pass command along!");
      }
}

// Fires when a sensor button is pressed
void Sensor_clicked_cb(GtkButton *button, gpointer user_data) {
      printf("Sensor button clicked\n");
      char msgBuf[MAX_MSG_SIZE];

      const gchar *label = gtk_button_get_label(button);
      strcpy(msgBuf, label);
      if (sendToOther(msgBuf, MAX_MSG_SIZE) != 0) {
            printf("Failed to pass command along!");
      }
}

// Send button was pressed
void Send_clicked_cb(GtkButton *button, gpointer user_data) {
      printf("Send button clicked\n");
      // Take the contents of the custom command input
      // and send it to the message queue service
      char msgBuf[MAX_MSG_SIZE] = {};

      const gchar *entry_text;
      guint16 entry_size;
      entry_text = gtk_entry_get_text(entry_command);
      entry_size = gtk_entry_get_text_length(entry_command);
      strcpy(msgBuf, entry_text);

      printf("Got text of size %d: %s\n", entry_size, msgBuf);
      if (sendToOther(msgBuf, MAX_MSG_SIZE) != 0) {
            printf("Failed to pass command along!");
      }
}


// Runs repeatedly on a timer
int timer_handler() {
      char msgBuf[MAX_MSG_SIZE] = {};
      char responseBuf[MAX_MSG_SIZE] = {};

      // Show any data we got in the queue
      if (readFromOther(msgBuf, MAX_MSG_SIZE) > 0) {
            printf("Received: %s\n", msgBuf);
            sprintf(responseBuf, "'%s'", msgBuf);
            gtk_label_set_text(GTK_LABEL(lbl_response), responseBuf);    // update
label
            return 1;
      } else {
            memset(msgBuf, 0, MAX_MSG_SIZE);
      }

      return 1; // returning 0 will destroy us.
}


int main(int argc, char *argv[]) {
      // Setup message queuing
      if (createMsgQueues(0) < 0) {
            printf("Error creating message queue!\n");
      }

      GtkBuilder  *builder;
      GtkWidget   *window;
      GError            *error = NULL;


      gtk_init(&argc, &argv);

      builder = gtk_builder_new();

      // try and load file for GUI
      if (!gtk_builder_add_from_file(builder, "assets/main.glade", &error)){
            g_warning("%s", error->message);
            g_free(error);
            return(1);
      }

      // get widgets/objects
      window = GTK_WIDGET(gtk_builder_get_object(builder, "myWindow"));
      lbl_response = GTK_WIDGET(gtk_builder_get_object(builder, "lbl_response"));
      entry_command = GTK_ENTRY(gtk_builder_get_object(builder,
"custom_command"));
      entry_value = GTK_ENTRY(gtk_builder_get_object(builder,
"enter_value_field"));
```

```
	xCommand = GTK_TEXT_BUFFER(gtk_builder_get_object(builder, "readT_cmd"));

	// Hookup our signals
	gtk_builder_connect_signals(builder, NULL);

	GtkTextBuffer test;
	gtk_builder_connect_signals(builder, &test);

	// Destroy the builder, now that we're done with it.
	g_object_unref(G_OBJECT(builder));

	// Make quitting the GUI also stop the program.
	g_signal_connect(window, "destroy", G_CALLBACK(gtk_main_quit), NULL);

	// Number is in milliseconds
	g_timeout_add(50, (GSourceFunc)timer_handler, NULL);

	gtk_widget_show(window);

	gtk_main();

	closeQueues();

	return 0;
}
```

Code – msgQueue.h

```
/*
 * msgQueue.h
 *
 *  Created on: Nov 24, 2020
 *      Author: Zachary Whitlock
 */

#ifndef SRC_MSGQUEUE_H_
#define SRC_MSGQUEUE_H_

#include <fcntl.h>
#include <sys/stat.h>
#include <stdio.h>
#include <mqueue.h>
#include <errno.h>

#define OTHER_MSG_FILE        "/otherMsgQueue"
#define CLIENT_MSG_FILE       "/clientMsgQueue"

#define QUEUE_PERMISSIONS 0660
#define MAX_MESSAGES 10
#define MAX_MSG_SIZE 32
#define MSG_BUFFER_SIZE MAX_MSG_SIZE + 10

int createMsgQueues(int client );
int sendToOther(const char *msgBuf, size_t bufSize);
int readFromOther(char *msgBuf, size_t bufSize);
int closeQueues();
```

```c
mqd_t other_server;
mqd_t client_server;   // queue descriptors

#endif /* SRC_MSGQUEUE_H_ */
```

code – msgQueue.c

```c
/*
 * msgQueue.c
 *
 *  Created on: Nov 24, 2020
 *      Author: Zachary Whitlock
 *
 * Writes to the GUI message queue and reads from the client message queue
 */

#include "msgQueue.h"


// Open the message queue
// Make the receive functionality non-blocking.
// Set 'client' to anything except 0 to make this the client.
// Returns -1 if error, anything >= 0 is good.
int createMsgQueues(int client) {
    struct mq_attr attr;

    attr.mq_flags = 0;
    attr.mq_maxmsg = MAX_MESSAGES;
    attr.mq_msgsize = MAX_MSG_SIZE;
    attr.mq_curmsgs = 0;

    if ((client_server = mq_open (CLIENT_MSG_FILE, O_RDWR | O_CREAT | O_NONBLOCK,
QUEUE_PERMISSIONS, &attr)) == -1) {
      perror("Error on opening client file: ");
      return -1;
    }

    if ((other_server = mq_open (OTHER_MSG_FILE, O_RDWR | O_CREAT | O_NONBLOCK,
QUEUE_PERMISSIONS, &attr)) == -1) {
      perror("Error on opening GUI file:");
      return -1;
    }

    if (client != 0){
      mqd_t tempHolder = client_server;
      client_server = other_server;
      other_server = tempHolder;
      printf("Initialized queues. Sending to '%s' and receiving from '%s'\n",
                  OTHER_MSG_FILE,
                        CLIENT_MSG_FILE);
    } else {
      printf("Initialized queues. Sending to '%s' and receiving from '%s'\n",
                  CLIENT_MSG_FILE,
                        OTHER_MSG_FILE);
    }

    return 0;
}

// Send a message to client program
// Returns -1 if error, anything >= 0 is good.
int sendToOther(const char *msgBuf, size_t bufSize) {
	if (mq_send(client_server, msgBuf, bufSize, 0) < 0) {
		perror("Error sending message.");
		return -1;
	}
	return 0;
}

// Close a message queue
// Returns -1 if error, anything >= 0 is good.
// TODO
int closeQueues() {
	mq_close(other_server);
	mq_close(client_server);
	return 0;
}

// Attempt to receive a message from the queue
// Returns -1 if error, anything >= 0 is good.
int readFromOther(char *msgBuf, size_t bufSize) {
	ssize_t numRecieved = mq_receive(other_server, msgBuf, bufSize, NULL);
	if (numRecieved < 0 && errno != EAGAIN) {
		perror("Error receiving! Reason");
		return -1;
	}
	return (int)numRecieved;
}
```

Lab 8

Introduction

The objective of this lab is to connect to another student's BBB and facilitate a connection to my BBB. Due to current circumstances (COVID-19), a local-network connection was not used this term. Instead, I voluntarily hosted a service on a public server to allow students to forward traffic through the internet to their local devices.

Part 1

Modifications

To connect to another server via the GUI designed and written in previous lab, some modifications were required. My client application was slightly modified to connect to a specified IP and port rather than something hard-coded. Additionally, a new tab was added to the GUI for controlling my classmate's BBB.

Setting up a Reverse SSH Tunnel

On my Virtual Private Server (VPS), I set up a user called "ee407" for my classmates. In the SSH config file, I added the following lines:

```
# Temporarily allow users to login with password
Match User ee407
        PasswordAuthentication yes
```

This enables password authentication for only the EE407 user and allows other users to connect. I also enabled the "AllowTcpForwarding" and "GatewayPorts" settings in the SSH config file in the global sense.

Students could forward traffic from my server to their BBB by running the following command from their BBB:

```
ssh -R 5000:localhost:51717 ee407@gectorsbox.net
```

The first port (<first>:localhost:<second>) is the port used on the remote server, and the second port is the port used by the program controlling the BBB locally. To connect to the tunnel created by that example, you connect telnet to that tunnel:

```
telnet gectorsbox.net 5000
```

The raspberry pi connects to the same place, although usually replacing gectorsbox.net with it's IP address.

Connecting to Another BBB Server

Figure 3 shows my RPi's screen and output after sending "SENSOR ON" to Truong's BBB.

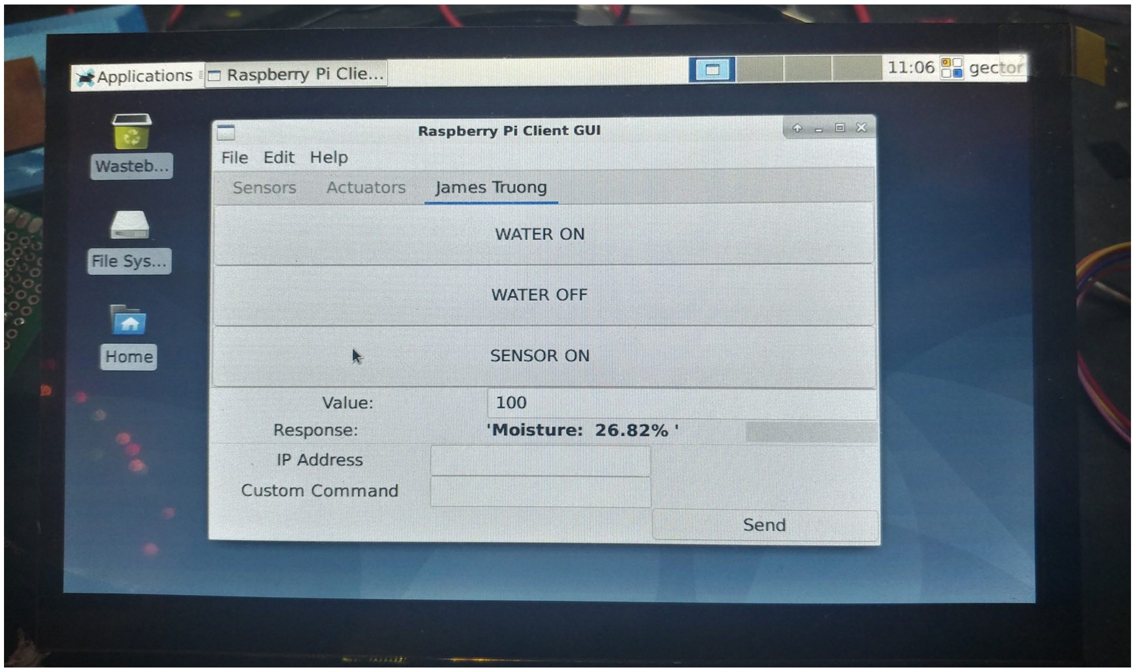
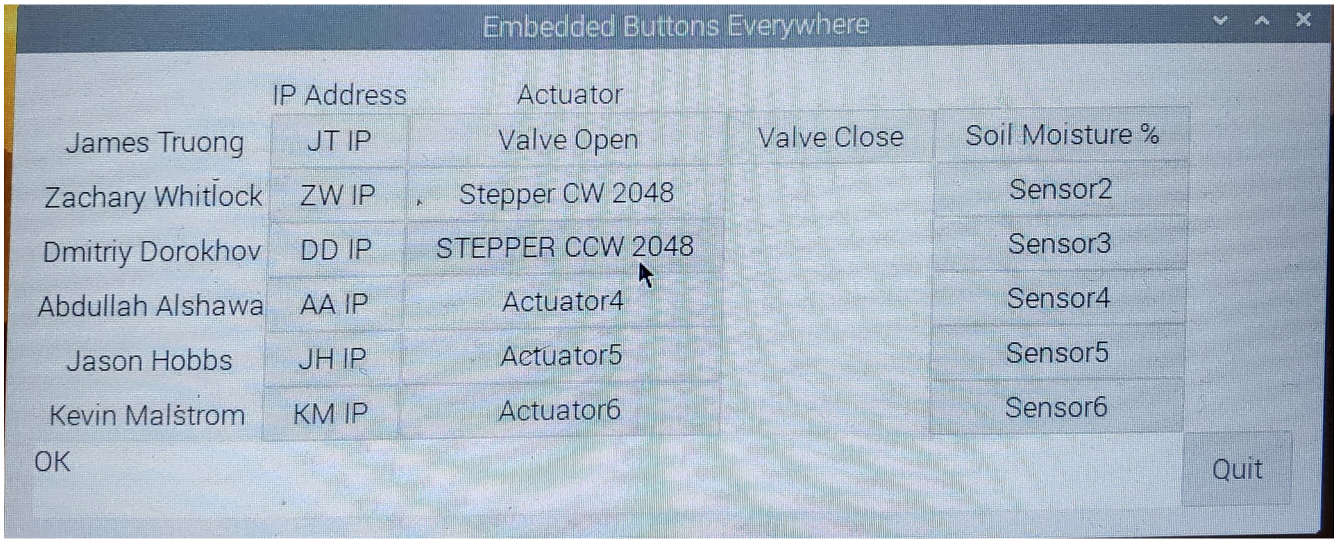


Figure 3: My GUI showing the response from the other student's BBB

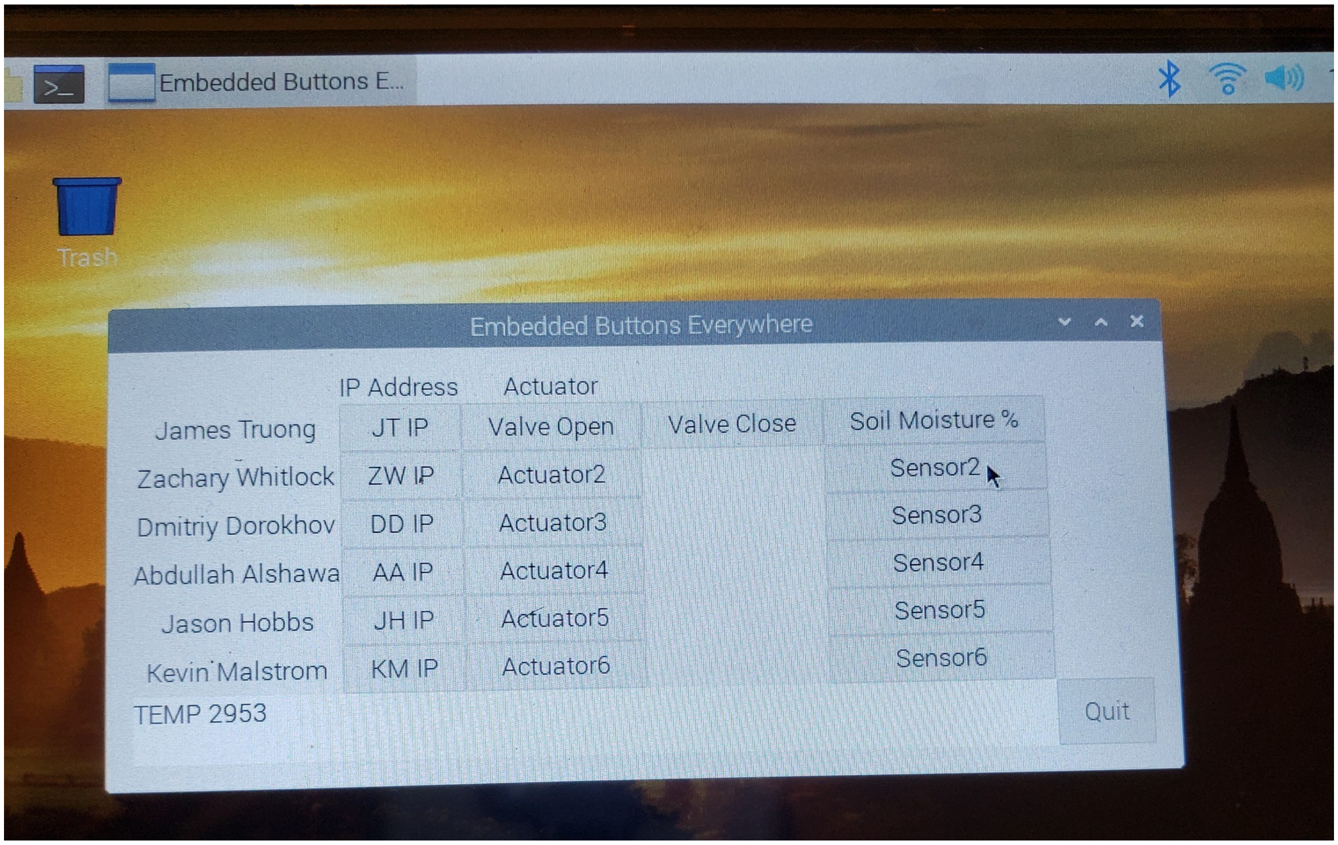
Part 2

In this part of the lab, my communication was given to another student so that they might connect to my BBB.



```
OK
```

Figure 4: Truong's Output after sending a command to my stepper motor ('OK')



```
TEMP 2953
```

Figure 5: Response from my BBB on Truong's GUI, "TEMP 2953"

Figures 4 & 5 are the proof that the other student, James Truong, was able to communicate with my server.

This video shows my stepper motor turning halfway around from Truong's command specifying 2048/4096 steps:

https://www.youtube.com/watch?v=EP_8lok31Vw

Conclusion

Truong and I spent quite a bit of time on getting his BBB server to receive messages from computers over the internet. In the end, the solution was primarily that he was running the wrong command on his BBB to create the reverse SSH tunnel. However, he also added code to handle commands with better error handling and debugging so we could tell what was going on. He also had to modify his client code to be able to talk to my BBB. In modifying *my* client code and GUI, I opted to allow a connection to only 1 BBB at a time, even though the GUI was originally designed to talk with multiple servers. For personal use, the client code could be modified to create new connections when asked to by the GUI, and I could create a inter-program protocol for setting up network connections based on commands from the GUI.

In this project, I've learned a ton about C programming, typical Linux and embedded systems conventions and practices, and basic circuit control with a Single-Board-Computer like the Raspberry Pi or BeagleBone Black. I've gotten to learn how to configure and use internal peripherals like I2C and how to make use of onboard sensors like the CPU temperature.

This is the final lab of the term. I chose not to do part 3 which was to connect with a second student, for extra credit. In this lab I learned a lot since I also set up the reverse-ssh tunnel for the rest of the students. I learned how to connect to a remote device over the internet in a secure fashion and a little bit on how SSH works. Other students in the class were also able to use their projects in a more real-world way because of the internet tunnel, we could work almost as if we were on the same Local Area Network.

In addition to the proof of remote control, I recorded a video of the balance feature of my program. https://www.youtube.com/watch?v=fa7_deQBjU4

Jump to Index

Appendix

All of the code for this lab and the rest of the project is available on gitlab.

https://gitlab.com/CaptainGector/bbb_server https://gitlab.com/CaptainGector/rpi_gui

https://gitlab.com/CaptainGector/rpi_client