Appendix C

Robotics Beaglebone Programming

MEEN 408 Team 1

Fall 2016 Texas A&M University

Table 0.0.1: Revision History

Version	Date	Author	Reviewed	Details
1.0	12/15/2016	AJE	—	Document Creation

Table of Contents

A Note to The Reader

Digital Document

This document was prepared in LaTeX– a dynamic document processor. As such, the document is intended for digital use. Throughout, there are hyperlinks, highlighted in blue, like this.

1 Hardware Interaction

1.1 Device Tree Overlays and .dts Files

Computers (especially those aimed at embedded operations such as the Beaglebone) typically come fitted with a smoregasboard of useful I/O devices–such as gpio pins, pulse width modulating signals, analog-to-digital converters, etc–built right into the board. And while the Beaglebone does indeed come with all of these devices physically attached, it is the joy and responsibility of the operating system (in conjunction with the user) to access them in a programmatic and useful way. The solution originally adopted by the Linux community for accessing these devices was to create a system in which the Linux Kernel (the lowest level of the Linux operating system that already performs all the bare-metal operation on the computer components) would be recompiled when needed with additional hardware access to these other peripheral devices. Unfortunately, this method required the kernel to be recompiled whenever the devices or their connections changed, which is quite a bother, especially when device changes happen frequently. To the rescue comes the Device Tree Overlay, which is a way in which kernel (linux) fragments can bed added to the kernel at runtime, meaning device connections that once required a recompile could now be added without even turning the computer off. In this way, users keep the benefit of being able to communicate with low-level devices without having the inconvenience of a constantly recompiling kernel. The common method for adding these fragments is through a compiled .dts (device tree) file.

The structure of a dts file is not too complicated. It begins with information about the system the particular dts file was made for (which type of computer, supported devices, version number, etc) and any other programs it has as dependencies. The rest of the file consists of kernel fragments, as shown in the dts file below. Each fragment consists of a target (physical device) and any sequence of operations be performed on that device (provide access to device, activate device, etc.) In the file below, which is activating two of the PWMs on the Beaglebone, the pin chip (am33x) is first activated, followed by the two pwm chips, and finally the actual pwm modules on each chip.

These dts files are compiled to dtbo files and added to the kernel. What happens after this point is dependent on the particular system, but on the Beaglebone a typical result is that a directory is created on the Beaglebone containing files that point to various properties of the activated device. The properties of the corresponding device can be read and changed by reading/writing to/and from these files. This directory and its files are part of a virtual filesystem monitored by the operating system.

```
/*
 * Copyright (C) 2016 Seeed Studio.
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */
/dts-v1/;
/plugin/;

/{
    compatible = "ti,beaglebone", "ti,beaglebone-black", "ti,beaglebone-
       green";
    part-number = "BB-PWM12";
    version = "00A0";

    fragment@0 {
        target = <&am33xx_pinmux>;
            __overlay__ {
            pinctrl_spec: Panel_Pins {
                pinctrl-single,pins = <
                        0x048 0x06 /* P9_14 MODE6 ehrpwm1A PWM */
                        0x04c 0x06 /* P9_16 MODE6 ehrpwm1B PWM */
                        0x020 0x04 /* P8_19 MODE4 ehrpwm2A PWM */
                        0x024 0x04 /* P8_19 MODE4 ehrpwm2A PWM */
                    >;
                };
            };
    };

fragment@1 {
        target = <&ocp>;
                __overlay__ {
                test_helper: helper {
                        compatible = "bone-pinmux-helper";
                        pinctrl-names = "default";
                        pinctrl-0 = <&pinctrl_spec>;
                        status = "okay";
                        };
                };
};

fragment@2 {
        target = <&epwmss2>;
                __overlay__ {
                        status = "okay";
                };
};

fragment@3 {
        target = <&ehrpwm2>;
                __overlay__ {
                        status = "okay";
                };
```

```
};

fragment@4 {
        target = <&epwmss1>;
                __overlay__ {
                        status = "okay";
                };
};

fragment@5 {
        target = <&ehrpwm1>;
                __overlay__ {
                        status ="okay";
                };
};

};
```

1.2 C++ Access

As stated in the subsection above, devices on the Beaglebone are accessed through a virtual files system (though to the user this appears to be a regular file system). Thus , one of the easiest ways to access peripheral device in code is to read and write from these device files. And that is exactly what we do. A class was created for each type of peripheral device needed for the project that could be used in any C++ code to access the peripheral devices.

1.2.1 General Purpose Input/Output (GPIO) Pins

General Purpose Input/Output (gpio) pins can be used to either get a boolean reading of a signal (on/off) or to perform simple voltage outputs (3.3V on / 0V off). They were used to control motor enable and direction switches.

```
// Author: Augustus Ellis
#ifndef GPIO_H
#define GPIO_H

#include <unistd.h>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>

class GPIO {
 private:
  // port parameters
  int gpioPinNumber;
  std::string direction;
  int value;
  // filename strings
  std::string GPIOFile;
  std::string GPIOValueFile;
  std::string GPIODirectionFile;

 public:
  GPIO(int gpioPinNumberr, std::string directionn);
  ~GPIO();
  void setValue(int Valuee);
  int getValue();
};

#endif
```

```cpp
// Author: Augustus Ellis
#include "GPIO.h"

GPIO::GPIO(int gpioPinNumberr, std::string directionn) {
  // check that inputs are valid
  if (!(directionn.compare("out") || directionn.compare("in"))) {
    printf("Invalid Direction Input. \n");
    // throw exception;
  }
  // Set internal port parameters
  std::cout << "GPIO Constuctor.\n";
  gpioPinNumber = gpioPinNumberr;
  direction = directionn;
  // Set file strings
  std::stringstream ss;
  // Pin Number String
  ss << gpioPinNumber;
  GPIOFile = ss.str();
  ss.clear();
  ss.str(std::string());
  // Value File String
  ss << "/sys/class/gpio/gpio" << gpioPinNumber << "/value";
  GPIOValueFile = ss.str();
  ss.clear();
  ss.str(std::string());
  // Direction File String
  ss << "/sys/class/gpio/gpio" << gpioPinNumber << "/direction";
  GPIODirectionFile = ss.str();
  ss.clear();
  ss.str(std::string());

  // Export the Pin Number (this will make the Pin directory we can then
     use)
  // First we see if it is possible to create the directory. If not, quit.
  std::ofstream ofs;
  ofs.open(std::string("/sys/class/gpio/export").c_str(), std::ios::app);
  if (!(ofs.is_open())) {
    std::cout << "Cannot export the GPIO Pin\n";
    // throw exception;
  } else {
    // If connecting to the export file works, we export the pin number.
    ofs << gpioPinNumber;  // write pin number to export file
    ofs.close();           // and close the file
  }

  // Set the pin direction
  ofs.open(GPIODirectionFile.c_str(), std::ios::trunc);
  if (!(ofs.is_open())) {
    std::cout << "Cannot set the GPIO direction.";
    // throw exception;
  } else {
    // If connecting to the direction file works, we export the direction
    ofs << direction;
    ofs.close();
```

```
54    }
55  }
56
57  GPIO::~GPIO() {
58    std::ofstream ofs;
59    ofs.open(std::string("/sys/class/gpio/unexport").c_str(), std::ios::app)
          ;
60    if (!(ofs.is_open())) {
61      std::cout << "Cannot unexport the GPIO Pin\n";
62      // throw exception;
63    } else {
64      // If connecting to the export file works, we export the pin number.
65      ofs << gpioPinNumber;  // write pin number to export file
66      ofs.close();           // and close the file
67    }
68  }
69
70  void GPIO::setValue(int valuee) {
71    std::ofstream ofs;
72    ofs.open(GPIOValueFile.c_str(), std::ios::trunc);
73    if (!(ofs.is_open())) {
74      std::cout << "Cannot set the GPIO Value.";
75      // throw exception;
76    } else {
77      value = valuee;
78      ofs << value;
79      ofs.close();
80    }
81  }
82
83  int GPIO::getValue() {
84    std::ifstream ifs;
85    int value = 0;
86    ifs.open(GPIOValueFile.c_str());
87    if (!(ifs.is_open())) {
88      std::cout << "Cannot get the GPIO Value.";
89      // throw exception;
90    } else {
91      ifs >> value;
92      ifs.close();
93    }
94    return value;
95  }
```

1.2.2 Analog to Digital Converters

Analog to digital converters take an analog signal between 0 and 1.8 V and return a digital reading between 0 and 4095 corresponding to this input signal. They are used for low level voltage analog readings.

```
// Author: Augustus Ellis
#ifndef ADC_H
#define ADC_H

#include <unistd.h>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>

class ADC {
 private:
  // ADC parameters
  double ADCVoltage;
  int ADCRawReading;
  int ADCPinNumber;
  int maxRaw = 4095;            // maximum returned bit reading
  int minRaw = 0;               // minimum returned bit reading
  double maxADCVoltage = 1.8;   // maximum allowed voltage -- DO NOT EXCEED
  double minADCVoltage = 0.0;   // minimum measureable voltage (lowest
      output)
  // filename strings
  std::string ADCDeviceFile;
  std::string ADCVoltageFile;

 public:
  ADC(int ADCPinNumberr);
  double readVoltage();
  int readRaw();
};

#endif
```

```cpp
// Author: Augustus Ellis
#include "ADC.h"

ADC::ADC(int ADCPinNumberr) {
  // check that inputs are valid - for now we assume they are valid
  // Set internal port parameters
  ADCPinNumber = ADCPinNumberr;

  // Set filename strings
  std::stringstream ss;
  ss << "/sys/bus/iio/devices/iio:device0/";  // ADCDeviceFile
  ADCDeviceFile =
      ss.str();  // put the contents of the stringstream into the device string
  ss.clear();    // clear any status flags on the stringstream
  ss.str(std::string());  // delete the contents of the stringstream so we can
                          // use it again
  ss << ADCDeviceFile << "in_voltage" << ADCPinNumber
     << "_raw";               // ADCVoltageFile
  ADCVoltageFile = ss.str();  // see above for device file
  ss.clear();                 //
  ss.str(std::string());      //
}

double ADC::readVoltage() {
  int rawADC = readRaw();  // get the raw bit voltage reading from the adc pin
                           // using a function defined below
  ADCVoltage = ((maxADCVoltage - minADCVoltage) / (maxRaw - minRaw)) * rawADC +
               minADCVoltage;  // convert to actual voltage from bit reading
  return ADCVoltage;
}

int ADC::readRaw() {
  std::ifstream ifs;  // filestream variable used to read from voltage file
  int rawADC = 0;     // variable used to store direct reading
  ifs.open(ADCVoltageFile.c_str());  // open the voltage file
  if (!(ifs.is_open())) {
    std::cout << "Cannot get the ADC Voltage.\n";
    // throw exception;
  } else {
    ifs >> rawADC;  // read the file voltage (in bits) into the temporary
                    // storage variable
  }
  ifs.close();              // close the file
  ADCRawReading = rawADC;   // put the reading in the class variable
                            // ADCRawReading -- not necessary, but done
  return rawADC;            // return the voltage reading
}
```

1.2.3 Pulse Width Modulated (PWM) Signal

Pulse width modulated signals were used to control motor drivers and servomotors. The Beaglebone black has 6 PWMs in 3 A/B pairs.

```
// Augustus Ellis
#ifndef PWM_H
#define PWM_H

#include <unistd.h>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>

class PWM {
 private:
  // PWM parameters
  int Period;
  int DutyCycle;
  int PWMNumber;
  std::string Polarity;
  int Enabled;
  // filename strings
  std::string PWMFile;
  std::string PWMPeriodFile;
  std::string PWMDutyCycleFile;
  std::string PWMPolarityFile;
  std::string PWMEnableFile;
  std::string PWMExportFile;
  std::string PWMUnexportFile;

 public:
  PWM(int chipNumberr, int PWMNumberr, int Periodd, int DutyCyclee);
  ~PWM();
  void setPeriod(int Periodd);
  void setDutyCycle(int DutyCyclee);
  void setPolarity(std::string Polarityy);
  void enable(int enablee);
  int getPeriod();
  int getDutyCycle();
  std::string getPolarity();
};

#endif
```

```cpp
// Author: Augustus Ellis
#include <project/PWM.h>

PWM::PWM(int chipNumberr, int PWMNumberr, int Periodd, int DutyCyclee) {
  // check that inputs are valid - for now we assume they are valid
  // Set internal port parameters
  PWMNumber = PWMNumberr;
  DutyCycle = DutyCyclee;
  Period = Periodd;
  std::cout << "PWM Constructor Start\n.";
  if (!(chipNumberr == 0 || chipNumberr == 2)) {
    std::cout << "Cannot Access pwmchipx." << std::endl;
  }
  if (!(PWMNumberr == 0 || PWMNumberr == 1)) {
    std::cout << "Cannot Access pwmx." << std::endl;
  }
  // Set filename strings
  std::stringstream ss;
  ss << "/sys/class/pwm/pwmchip" << chipNumberr << "/pwm" << PWMNumber
     << "/";                        // PWMFile
  PWMFile = ss.str();               //
  ss.clear();                       //
  ss.str(std::string());            //
  ss << PWMFile << "period";        // PeriodFile
  PWMPeriodFile = ss.str();         //
  ss.clear();                       //
  ss.str(std::string());            //
  ss << PWMFile << "duty_cycle";    // DutyCycleFile
  PWMDutyCycleFile = ss.str();      //
  ss.clear();                       //
  ss.str(std::string());            //
  ss << PWMFile << "polarity";      // PolarityFile
  PWMPolarityFile = ss.str();       //
  ss.clear();                       //
  ss.str(std::string());            //
  ss << PWMFile << "enable";        // EnableFile
  PWMEnableFile = ss.str();         //
  ss.clear();                       //
  ss.str(std::string());            //

  // Check if the PWM device is in slots

  std::string PWMSlots = "/sys/devices/platform/bone_capemgr/slots";
  std::ofstream ofs;
  /*  ofs.open(PWMSlots.c_str(),
             std::ios::app);
    if (!(ofs.is_open())) {
      std::cout << "Cannot export the PWM Device to Slots\n";
      // throw exception;
    } else {
      ofs << "BB-PWM2";  // write pin number to export file
      std::cout << "Wrote to Slots" << std::endl;
      //usleep(100000);
      ofs.close(); // and close the file
```

```
55      }
56    */
57    // Export the PWM  Number (this will make the pwm directory we can then
          use)
58    ss << "/sys/class/pwm/pwmchip" << chipNumberr << "/export";  // PWMFile
59    PWMExportFile = ss.str();
60    ss.clear();                                                    //
61    ss.str(std::string());                                         //
62    ss << "/sys/class/pwm/pwmchip" << chipNumberr << "/unexport";  //
          PWMFile
63    PWMUnexportFile = ss.str();
64    ss.clear();             //
65    ss.str(std::string());  //
66
67    ofs.open(PWMExportFile.c_str(), std::ios::app);
68    if (!(ofs.is_open())) {
69      std::cout << "Cannot export the PWM Pin\n";
70      // throw exception;
71    } else {
72      ofs << PWMNumber;  // write pin number to export file
73      ofs.close();       // and close the file
74      int dummy;
75      std::cout << "Exported PWM pin." << std::endl;
76    }
77    // Disable pin Control
78    enable(0);
79    // Set Period and Duty Cycle
80    std::cout << "Setting Period and Duty Cycle." << std::endl;
81    setPeriod(Periodd);
82    setDutyCycle(DutyCyclee);
83    std::cout << "Period and Duty Cycle set." << std::endl;
84    std::cout << "PWM Constructor End." << std::endl;
85  }
86
87  PWM::~PWM() {
88    // Unexport the PWM
89    setDutyCycle(0);
90    std::ofstream ofs;
91    ofs.open(PWMUnexportFile.c_str(), std::ios::app);
92    if (!(ofs.is_open())) {
93      std::cout << "Cannot unexport the PWM Pin\n";
94      // throw exception;
95    } else {
96      ofs << PWMNumber;  // write pin number to unexport file
97    }
98    ofs.close();  // and close the file
99  }
100
101 void PWM::setPeriod(int Periodd) {
102   std::ofstream ofs;
103   ofs.open(PWMPeriodFile.c_str(), std::ios::trunc);
104   if (!(ofs.is_open())) {
105     std::cout << "Cannot set the PWM Period.";
106     // throw exception;
```

```
107   } else {
108     Period = Periodd;
109     ofs << Period;
110     ofs.close();
111     std::cout << "Period Set." << std::endl;
112   }
113 }
114 
115 void PWM::setDutyCycle(int DutyCyclee) {
116   std::ofstream ofs;
117   ofs.open(PWMDutyCycleFile.c_str(), std::ios::trunc);
118   if (!(ofs.is_open())) {
119     std::cout << "Cannot set the PWM Duty Cycle.";
120     // throw exception;
121   } else {
122     DutyCycle = DutyCyclee;
123     ofs << DutyCyclee;
124     ofs.close();
125   }
126 }
127 
128 void PWM::setPolarity(std::string Polarityy) {
129   std::ofstream ofs;
130   ofs.open(PWMPolarityFile.c_str(), std::ios::trunc);
131   if (!(ofs.is_open())) {
132     std::cout << "Cannot set the PWM Polarity.";
133     // throw exception;
134   } else {
135     Polarity = Polarityy;
136     ofs << Polarityy;
137     ofs.close();
138   }
139 }
140 void PWM::enable(int enablee) {
141   // First we truly disable the PWM if we unenable by setting duty cycle
       to
142   // zero.
143   std::ofstream ofs;
144   ofs.open(PWMEnableFile.c_str(), std::ios::trunc);
145   if (!(ofs.is_open())) {
146     std::cout << "Cannot enable the PWM.";
147     // throw exception;
148   } else {
149     Enabled = enablee;
150     ofs << enablee;
151     ofs.close();
152   }
153   if (enablee == 0) {
154     int temp = DutyCycle;
155     setDutyCycle(0);
156     DutyCycle = temp;
157   } else if (enablee == 1)  // otherwise reactivate PWM before enabling
158   {
159     setDutyCycle(DutyCycle);
```

```
160   }
161 }
162
163 int PWM::getPeriod() {
164   //  std::ifstream ifs;
165   //  int PeriodValue = 0;
166   //  ifs.open(PWMPeriodFile.c_str());
167   //  if (!(ifs.is_open())) {
168   //    std::cout << "Cannot get the PWM Period.\n";
169   //    // throw exception;
170   //  } else {
171   //    ifs >> PeriodValue;
172   //  }
173   //  ifs.close();
174   return Period;
175 }
176
177 int PWM::getDutyCycle() {
178   //  std::ifstream ifs;
179   //  int DutyCycleValue = 0;
180   //  ifs.open(PWMDutyCycleFile.c_str());
181   //  if (!(ifs.is_open())) {
182   //    std::cout << "Cannot get the PWM Duty Cycle.\n";
183   //    // throw exception;
184   //  } else {
185   //    ifs >> DutyCycleValue;
186   //  }
187   //  ifs.close();
188   return DutyCycle;
189 }
190
191 std::string PWM::getPolarity() {
192   //  std::ifstream ifs;
193   //  std::string PolarityValue;
194   //  ifs.open(PWMPolarityFile.c_str());
195   //  if (!(ifs.is_open())) {
196   //    std::cout << "Cannot get the PWM Polarity.\n";
197   //    // throw exception;
198   //  } else {
199   //    ifs >> PolarityValue;
200   //  }
201   // ifs.close();
202   return Polarity;
203 }
```

1.2.4 Quadrature Encoders

Pulse width modulated signals were used to control motor drivers and servomotors.

```
// Author: Augustus Ellis
#ifndef EQEP_H
#define EQEP_H

#include <unistd.h>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>

class EQEP {
 private:
  // ADC parameters
  int EQEPPosition;
  int EQEPNumber;
  // filename strings
  std::string EQEPDeviceFile;
  std::string EQEPPositionFile;

 public:
  EQEP(int EQEPNumberr);
  int readPosition();
};

#endif
```

```cpp
// Author: Augustus Ellis
#include "EQEP.h"

EQEP::EQEP(int EQEPNumberr) {
  // check that inputs are valid - for now we assume they are valid
  // Set internal port parameters
  EQEPNumber = EQEPNumberr;
  std::cout << "EQEP Constructor.\n";
  // Set filename strings
  std::stringstream ss;
  ss << "/sys/devices/platform/ocp/48304000.epwmss/48304180.eqep/";  //
     DeviceFile
  EQEPDeviceFile = ss.str();                                          //
  ss.clear();                                                         //
  ss.str(std::string());                                              //
  ss << EQEPDeviceFile << "position";  // PositionFile
  EQEPPositionFile = ss.str();         //
  ss.clear();                          //
  ss.str(std::string());               //
}

int EQEP::readPosition() {
  std::ifstream ifs;
  ifs.open(EQEPPositionFile.c_str());
  if (!(ifs.is_open())) {
    std::cout << "Cannot get the EQEP Position.\n";
    // throw exception;
  } else {
    ifs >> EQEPPosition;
  }
  ifs.close();
  return EQEPPosition;
}
```

1.2.5 DC Motor

Motor control and motor encoder readings were abstracted into a DCMOTOR class containing a PWM signal class and a EQEP encoder class. This class was used to drive the joint motors.

```
// Author: Augustus Ellis, Blake Leiker, Anthony Hresko
#ifndef DCMOTOR_H
#define DCMOTOR_H

#include <unistd.h>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include "../EQEP/EQEP.h"
#include "../GPIO/GPIO.h"
#include "../PWM/PWM.h"

class DCMOTOR {
 private:
  // Motor Parameters
  PWM motorPWM;        // pwm to do basic control signal to motor
  EQEP motorEQEP;      // for reading back the angle
  GPIO motorHighGPIO;  // used to set the motor direcion
  GPIO motorLowGPIO;   // used to set the motor direction
  double gear_r;  // the motor's gear ratio. Not sure how to use this
     right now.
  double k_emf;   // emf motor constant  w = k_emf * Voltage
  double k_trq;   // torque constant     Torque = k_trq * current
  // Motor Tracking
  double angle;
  // Control Parameters
  int motorOn;
  int direction;  // 1 = forward, -1 = backward ==>these depend on how you
                  // install the motor

 public:
  DCMOTOR(int PWMChipNumberr, int PWMNumberr, int EQEPNumberr,
          int GPIONumberHigh, int GPIONumberLow);
  //~DCMOTOR();
  int getAngle();
  void setPWMPeriod(int Periodd);
  int getPWMPeriod();
  void setPWMDutyCycle(double DutyCyclee);
  int getPWMDutyCycle();
  double setk_emf(double k_emff);
  double getk_emf();
  double setk_trq(double k_trqq);
  double getk_trq();
  void enable(int enablee);  // disables or enables the motors
  void setDirection(int directionn);
};

#endif
```

```
// Author: Augustus Ellis, Blake Leiker, Anthony Hresko
#include "DCMOTOR.h"

DCMOTOR::DCMOTOR(int PWMChipNumberr, int PWMNumberr, int EQEPNumberr,
                 int GPIONumberHigh, int GPIONumberLow)
    : motorPWM(PWMChipNumberr, PWMNumberr, 1000000, 0),
      motorEQEP(EQEPNumberr),
      motorHighGPIO(GPIONumberHigh, "out"),
      motorLowGPIO(GPIONumberLow, "out") {
  // the commands above with : pwmconstructor, eqepconstructor are called an
  // initializer list. We used to initialize the member variables, especially
  // when they are a instances of our own classes
  setk_trq(0);
  setk_emf(0);
  // We now unenable the motor
  enable(0);
  // Set the direction to the default direction.
  std::cout << "About to set direction." << std::endl;
  setDirection(1);
  std::cout << "Direction has been set." << std::endl;
  // and are done with the constructor.
}
int DCMOTOR::getAngle() {
  return motorEQEP.readPosition();  // return the current EQEP position.
}
void DCMOTOR::setPWMPeriod(int Periodd) {
  motorPWM.setPeriod(Periodd);  // set the pwm period in the pwm object
}
int DCMOTOR::getPWMPeriod() {
  return motorPWM.getPeriod();  // get the pwm period from the pwm object
}
void DCMOTOR::setPWMDutyCycle(double DutyFractionn) {
  int DutyCycleActual = int(DutyFractionn / 100.0 * getPWMPeriod());
  motorPWM.setDutyCycle(DutyCycleActual);
}
int DCMOTOR::getPWMDutyCycle() {
  return motorPWM.getDutyCycle();  // same as getPWMPeriod above
}
double DCMOTOR::setk_emf(double k_emff) { k_emf = k_emff; }
double DCMOTOR::getk_emf() { return k_emf; }
double DCMOTOR::setk_trq(double k_trqq) { k_trq = k_trqq; }
double DCMOTOR::getk_trq() { return k_trq; }
void DCMOTOR::enable(int enablee) {
  if (enablee !=
      0) {  // so long as the input is not zero, we accept it as enable
    enablee = 1;
  }
  motorPWM.enable(enablee);
  motorOn = 1;
}
void DCMOTOR::setDirection(int directionn) {
  if (directionn == -1) {
```

```
53      motorHighGPIO.setValue(0);
54      motorLowGPIO.setValue(1);
55      direction = directionn;
56    } else if (directionn = 1) {
57      motorLowGPIO.setValue(0);
58      motorHighGPIO.setValue(1);
59      direction = directionn;
60    } else {
61      // We could do something, but not needed for now.
62    }
63  }
```

1.2.6 Servomotor

Servomotor control takes place in a SERVOMOTOR class containing a PWM signal class and methods for relating a desired angle to the duty cycle of that PWM.

```
// Author: Blake Leiker, Augustus Ellis
#ifndef SERVOMOTOR_H
#define SERVOMOTOR_H

#include <unistd.h>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include "../PWM/PWM.h"

class SERVOMOTOR {
 private:
  PWM servoPWM;  // pwm which will send control signal to servomotor
  int Angle;
  int lower_limit;
  int upper_limit;
  int theta_low;
  int theta_high;

 public:
  SERVOMOTOR(int PWMNumberr);
  //  int getAngle();
  void setAngle(int Anglee);
  void gripperOpen();
  void gripperClose();
  ~SERVOMOTOR();
};

#endif
```

```cpp
// Author: Blake Leiker, Augustus Ellis
#include "SERVOMOTOR.h"

SERVOMOTOR::SERVOMOTOR(int PWMNumberr) : servoPWM(2, PWMNumberr, 20000000, 0) {
  lower_limit = 1100000;
  upper_limit = 1500000;
  theta_low = 0;
  theta_high = 90;
}

// SERVOMOTOR::int getAngle();

void SERVOMOTOR::setAngle(int Anglee) {
  if ((Anglee < theta_high) && (Anglee > theta_low)) {
    Angle = Anglee;
    int pulsewidth = 0;
    pulsewidth =
        (upper_limit - lower_limit) / (theta_high - theta_low) * Angle +
        lower_limit;
    servoPWM.setDutyCycle(pulsewidth);
  } else {
    // error
    std::cout << "Could not set Angle.\n";
  }
}

void SERVOMOTOR::gripperOpen() {
  servoPWM.enable(1);
  servoPWM.setDutyCycle(upper_limit);
}

void SERVOMOTOR::gripperClose() {
  servoPWM.enable(1);
  servoPWM.setDutyCycle(lower_limit);
}

SERVOMOTOR::~SERVOMOTOR() {
  servoPWM.setDutyCycle(upper_limit);
  servoPWM.enable(0);
}
```

2 Robot Control - ROS

Since it was not guaranteed or desired that each motor be connected to the same physical device, the Robot Operting System and it's ability to automate this distributed computing through nodes was taken advantage of through the creation of several Nodes for controlling DC Motors, Servomotors, and communicating commands.

2.0.1 DC Motor ROS Node

```
// Author: Augustus Ellis and Blake Leiker
#include <project/DCMOTOR.h>
#include <ros/ros.h>
#include <std_msgs/String.h>
#include <unistd.h>

class DC1 {
 private:
  DCMOTOR myDCMOTOR1;

 public:
  DC1(int PWMChipNumberr, int PWMNumberr, int EQEPNumberr, int GPIONumberHigh,
      int GPIONumberLow)
      : myDCMOTOR1(PWMChipNumberr, PWMNumberr, EQEPNumberr, GPIONumberHigh,
                   GPIONumberLow) {}
  void DC1_Callback(const std_msgs::String::ConstPtr &msg) {
    std::stringstream ss;
    ss.str(msg->data.c_str());
    int temp = 0;
    ss >> temp;
    std::cout << "Setting to " << temp << std::endl;
    myDCMOTOR1.setPWMDutyCycle(temp);
  }
  int getAngle() { return myDCMOTOR1.getAngle(); }
};

int main(int argc, char **argv) {
  ros::init(argc, argv, "DC1_node");
  DC1 myDC1(0, 0, 0, 30, 60);  // INPUTS IN PROGRESS
  ros::NodeHandle nh;
  ros::Rate loop_rate(10);

  ros::Publisher pub = nh.advertise<std_msgs::String>("q1", 1000);
  ros::Subscriber sub = nh.subscribe("u1", 1000, &DC1::DC1_Callback, &myDC1);

  while (ros::ok()) {
    std_msgs::String msg;

    std::stringstream ss;
    ss << myDC1.getAngle();
    msg.data = ss.str();

```

```
    pub.publish(msg);

    ros::spinOnce();
    loop_rate.sleep();
  }
}
```

2.0.2 GRIPPER ROS Node

Servomotor control takes place in a SERVOMOTOR class containing a PWM signal class and methods for relating a desired angle to the duty cycle of that PWM.

```
// Author: Blake Leiker, Augustus Ellis
#include <project/SERVOMOTOR.h>
#include <ros/ros.h>
#include <std_msgs/String.h>
#include <unistd.h>

class GRIPPER {
private:
  SERVOMOTOR mySERVO;

public:
  GRIPPER(int PWMNumberr) : mySERVO(PWMNumberr) { mySERVO.gripperClose(); }
  void GRIPPER_Callback(const std_msgs::String::ConstPtr &msg) {
    mySERVO.gripperOpen();
    Position = 1;
  }
  int Position;
};

int main(int argc, char **argv) {
  ros::init(argc, argv, "GRIPPER_node");
  GRIPPER myGRIPPER(0);
  myGRIPPER.Position = 0;
  ros::NodeHandle nh;
  ros::Rate loop_rate(10);
  ros::Publisher pub = nh.advertise<std_msgs::String>("grip_position",
      1000);
  ros::Subscriber sub = nh.subscribe("grip_release", 1000,
                                     &GRIPPER::GRIPPER_Callback, &
                                        myGRIPPER);
  while (ros::ok()) {
    std_msgs::String msg;
    std::stringstream ss;
    ss << myGRIPPER.Position;
    msg.data = ss.str();
    pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
  }
}
```

3 GUI

Human interaction with the program was accomplished through a simple Graphical User Interface (GUI) created in matlab and connected to the ROS network via the Robot Control Toolbox, which allows one to create ROS nodes in MATLAB.

3.0.1 ROS_READER

Matlab program that produces a ROS node that sends trajectory information and control signals to the motor.

```
function ROS_READER
clc;
% ROS_READER continuously reads data and plots in a GUI. CAN YOU SAY WOW
    ???

f = figure('Visible','off','Position',[260,50,800,600]);

hstart = uicontrol('Style','pushbutton','String','Start Plot','Position'
    ,[700,220,100,30],...
    'Callback',@startbutton_Callback,'Value',1);

hdatasend = uicontrol('Style','pushbutton','String','Send Data','Position'
    ,[700,120,100,30],...
    'Callback',@datasend_Callback,'Value',1);

himageprocess = uicontrol('Style','pushbutton','String','Image Processing'
    ,'Position',[700,320,100,30],...
    'Callback',@imageprocess_Callback,'Value',1);


% hOrig = axes('Units','pixels','Position',[500,350,100,100],'Visible','
    off');

% hBin = axes('Units','pixels','Position',[500,50,200,200],'Visible','off'
    );
% imshow(Bin_Imag,'Parent',handles.hBin);


hstart.Units = 'normalized';
f.Visible = 'on';

end
```

3.0.2 datasend_Callback

```
function datasend_Callback(src,event,handles)
    prompt = {'Enter X distance (ft):','Enter Y distance (ft)','Enter
       theta (degrees)',...
        'Enter theta_1 release (degrees)', 'Enter theta_2 release (degrees
           )',...
        'Time span (s)','Frequency (hz)'};
    dlg_title = 'Kill me slowly.';
    answer = inputdlg(prompt,dlg_title);


    x_given = cell2mat(answer(1)); x_given = str2double(x_given);
    y_given = cell2mat(answer(2)); y_given = str2double(y_given);
    theta_given = cell2mat(answer(3)); theta_given = str2double(
       theta_given);
    theta_1 = cell2mat(answer(4)); theta_1 = str2double(theta_1);
    theta_2 = cell2mat(answer(5)); theta_2 = str2double(theta_2);
    tf = cell2mat(answer(6)); tf = str2double(tf);
    hz = cell2mat(answer(7)); hz = str2double(hz);
    [qd_1, qdot_d_1, ~, qd_2, qdot_d_2, ~, t] = PathFollowPendulum(x_given
       ,y_given,theta_given,theta_1,theta_2,tf,hz);
    Kp = 6000; Kd = 50;

    qdot_1_previous = 0;
    qdot_2_previous = 0;

    q1sub = rossubscriber('/q1');
    q2sub = rossubscriber('/q2');

    u1pub = rospublisher('/u1','std_msgs/String');
    u1message = rosmessage(u1pub);

    u2pub = rospublisher('/u2','std_msgs/String');
    u2message = rosmessage(u2pub);

    for i = 1:length(t)

        q1 = receive(q1sub);
        q2 = receive(q2sub);
        plotQ1 = q1.Data;
        plotQ2 = q2.Data;
        plotQ1 = str2num(plotQ1);
        plotQ2 = str2num(plotQ2);
        e11 = qd_1(i)*1/0.0033 - plotQ1;
        e21 = qd_2(i)*1/0.0033 - plotQ2;
        e11_dot = qdot_d_1(i)*1/0.0033 - qdot_1_previous;
        e21_dot = qdot_d_2(i)*1/0.0033 - qdot_2_previous;
        qdot_1_previous = e11_dot;
        qdot_2_previous = e21_dot;

        u1 = abs(Kp*e11 + Kd*e11_dot);
        u2 = abs(Kp*e21 + Kd*e21_dot);

```

```
        if u1 > 1000000
            u1 = 999999;
        end
        if u2 > 1000000
            u2 = 999999;
        end

        u1message.Data = num2str(u1/1000000*100);
        send(u1pub,u1message);
        u2message.Data = num2str(u2/1000000*100);
        send(u2pub,u2message);

        if plotQ1*0.0033+plotQ2*0.0033 > 10 || i == length(t)
            gr = rospublisher('/grip_release','std_msgs/String')
            grmessage = rosmessage(gr);
            grmessage.Data = num2str(1);
            send(gr,grmessage);
            u1message.Data = num2str(0);
            send(u1pub,u1message);
            u2message.Data = num2str(0);
            send(u2pub,u2message);
        end

    end
end
```

3.0.3 startbutton_Callback

```
function startbutton_Callback(src,event, handles)
    q1sub = rossubscriber('/q1');
    q2sub = rossubscriber('/q2');
    ha = axes('Units','pixels','Position',[100,50,300,300]);
    ha.Units = 'normalized';
    i = 1; hold on;
    while 1
        q1data = receive(q1sub);
        q2data = receive(q2sub);
        plotQ1 = q1data.Data;
        plotQ2 = q2data.Data;
        plotQ1 = str2num(plotQ1);
        plotQ2 = str2num(plotQ2);
        plot(i,plotQ1,'om');
        plot(i,plotQ2,'xc');
        xlabel('iterations'); ylabel('qD'); title('Angle');
        legend('q_1 measured');
        axis([0 100 0 40]);
        i = i + 1;
    end
end
```

3.0.4 PathFollowPendulum

```
function [qd_1,qdot_d_1,qdotdot_d_1,qd_2,qdot_d_2,qdotdot_d_2, t] =
    PathFollowPendulum(x_given,y_given,theta_given,theta_1,theta_2,tf,hz)
% Kinematics With Selected Release Point
    % Kinematic Formulas:
    % dx = v0_x*t
    % dy = v0_y*t + 1/2*g*t^2

    % Units: ALL UNITS IN DEGREES, METERS, SECONDS.
    x_given = x_given .* 0.3048;
    y_given = y_given .* 0.3048;
    % Fixed Inputs
    g = -9.81; h_frame = .7; D_basket = 0.2032;
    l_1 = 0.33; l_2 = 0.33;
    theta_1_back = 0; theta_2_back = 0;

    % Variables:
    h_robot = l_1.*cosd(theta_1) + l_2.*cosd(theta_1 + theta_2);
    x0_robot = l_1.*sind(theta_1) + l_2.*sind(theta_1 + theta_2);
    y0_bot = -h_robot;
    y0_hoop = -h_frame - (-(y_given + 1/2.*D_basket.*sind(theta_given)));
    dy = y0_bot - y0_hoop;
    x0_hoop = x_given - 1/2.*D_basket.*cosd(theta_given);
    dx = x0_hoop - x0_robot;

    % Solve Kinematic Formulas:
    vf = 1./cosd(theta_1 + theta_2) .* sqrt(1/2.*g.*dx.^2 .* (1./(dy-dx./
        tand(theta_1 + theta_2))));

    % Initial conditions.
    q0_tot = theta_1_back + theta_2_back;
    v0_tot = 0;
    ac0_tot = 0;
    q1_tot = theta_1 + theta_2;
    v1_tot = vf;
    ac1_tot = 1;

    q0_1 = theta_1_back;
    v0_1 = 0;
    ac0_1 = 0;
    q1_1 = theta_1;
    v1_1 = vf/3;
    ac1_1 = 0.25;

    t0 = 0;

    t = [t0:1/hz:tf];
    c = ones(size(t));
    M = [1 t0 t0^2 t0^3 t0^4 t0^5;...
        0 1 2*t0 3*t0^2 4*t0^3 5*t0^4;...
        0 0 2 6*t0 12*t0^2 20*t0^3;...
        1 tf tf^2 tf^3 tf^4 tf^5;...
        0 1 2*tf 3*tf^2 4*tf^3 5*tf^4;...
```

```
51          0 0 2 6*tf 12*tf^2 20*tf^3];
52      b_tot = [q0_tot; v0_tot; ac0_tot; q1_tot; v1_tot; ac1_tot];
53      b_1 = [q0_1; v0_1; ac0_1; q1_1; v1_1; ac1_1];
54      a_tot = inv(M)*b_tot;
55      a_1 = inv(M)*b_1;
56
57
58      qd_tot = a_tot(1).*c + a_tot(2).*t + a_tot(3).*t.^2 + a_tot(4).*t.^3 +
            a_tot(5).*t.^4 + a_tot(6).*t.^5;
59      qdot_d_tot = a_tot(2).*c + 2*a_tot(3).*t + 3*a_tot(4).*t.^2 + 4*a_tot
            (5).*t.^3 + 5*a_tot(6).*t.^4;
60      qdotdot_d_tot = 2*a_tot(3).*c + 6*a_tot(4).*t + 12*a_tot(5).*t.^2 +
            20*a_tot(6).*t.^3;
61
62      qd_1 = a_1(1).*c + a_1(2).*t + a_1(3).*t.^2 + a_1(4).*t.^3 + a_1(5).*t
            .^4 + a_1(6).*t.^5;
63      qdot_d_1 = a_1(2).*c + 2*a_1(3).*t + 3*a_1(4).*t.^2 + 4*a_1(5).*t.^3 +
            5*a_1(6).*t.^4;
64      qdotdot_d_1 = 2*a_1(3).*c + 6*a_1(4).*t + 12*a_1(5).*t.^2 + 20*a_1(6)
            .*t.^3;
65
66      qd_2 = qd_tot - qd_1;
67      qdot_d_2 = qdot_d_tot - qdot_d_1;
68      qdotdot_d_2 = qdotdot_d_tot - qdotdot_d_1;
```

3.0.5 Camera Image Processing

```
function [Orig_Imag,Bin_Imag,x,y,theta]=MEEN408_Vision()
%% Computer Vision function for our MEEN 408 Robotics Project
% Created by Tony Hresko, Blake Leiker, Augustus Ellis
% Created for the honors portion of the MEEN 408 Robotics class project
clear('cam');
cam = webcam(1);  %Creating the camera object
% cam.Resolution='1280x960';  %Adjusting the resolution to the max value
cam.Brightness=50;

%% Processing primary image
img=snapshot(cam); %Taking the picture of the target
Orig_Imag=img; % First output, the original image
color=1; %%%%%%%%%%%%%% CHANGE THIS TO CHANGE WHAT COLOR IS PULLED (1=RED,
    2=GREEN, 3=BLUE)
img=img(:,:,color); %Only pulling red colors because hoop is red
height=size(img,1);   %Height of image in pixels
width=size(img,2);    %Width of image in pixels

img=img(size(img,1):-1:1,:); %Flipping it so plot makes sense so that the
   scatter plot has the right orientation

%Finding all of the pixels above a certain threshhold of brightness
threshold = 80; %%%%%%%%%%%%%%%%%%%%%% CHANGE THIS VALUE TO CHANGE THE
   THRESHOLD

counter = 1;    %Initializing the counter
% Loop to save all of the bright images
for i=1:height
    for j=1:width
        if img(i,j)>threshold
            Brights(:,counter)=[j,i]; %#ok<AGROW>
            counter=counter+1;
        end
    end
end

Bin_Imag=im2bw(Orig_Imag,threshold/255); %Converting original image to
   binary based on the threshold chosen

%% Processing secondary image
% Secondary image is for identifying the location of the green marker
% relative the center of the elipse so we know if theta is up or down.

img_g=Orig_Imag(:,:,2); %Only pulling green colors because hoop marker is
   green

img_g=img_g(size(img_g,1):-1:1,:); %Flipping it so plot makes sense so
   that the scatter plot has the right orientation

%Finding all of the pixels above a certain threshhold of brightness
threshold_g = 200; %%%%%%%%%%%%%%%%%%%%%% CHANGE THIS VALUE TO CHANGE THE
    THRESHOLD
```

```
46
47  counter = 1;    %Initializing the counter
48  % Loop to save all of the bright images
49  Brights_g(:,1)=[width/2,140]; %#ok<AGROW>
50  for i=1:height
51      for j=1:width
52          if img_g(i,j)>threshold_g
53              Brights_g(:,counter)=[j,i]; %#ok<AGROW>
54              counter=counter+1;
55          end
56      end
57  end
58  % figure;  Originally included to see green plot
59  % scatter(Brights_g(1,:),Brights_g(2,:));
60  yrange_g=Brights_g(2,:);
61  green_height=mean(yrange_g);%%%% This is in pixels right now
62
63  %% Calculating outputs
64  xrange=Brights(1,:);
65  yrange=Brights(2,:);
66
67  % Calculating axes of the ellipse
68  Maj_Ax=max(xrange)-min(xrange);
69  Min_Ax=max(yrange)-min(yrange);
70
71  y=(max(yrange)+min(yrange))/2; % Height of the target in pixels
        %%%%%%%%%%%%%% Figure out how to convert this to feet later
72
73  % Calculating theta
74  theta_mag=asind(Min_Ax/Maj_Ax);
75  % If statement to determine if sign of theta
76  if y>green_height
77      sign=1;
78  else
79      sign=-1;
80  end
81
82  % Calculating x
83  % Need to figure out how to do this
84
85  % Equations based on calibration
86  x=-.0284*Maj_Ax+8.405;  %Equation for x as f(Maj_ax Pixels). Outpuuts in
        ft
87  y=0.0053*green_height+.7416;    %Lacking equation relating pixel height of
         green to height in feet
88  theta=real(sign*theta_mag);
89  clear('cam');
90
91  end
```

3.0.6 imageprocess_Callback

```
function imageprocess_Callback(src,event, handles)
[Orig_Imag,Bin_Imag,x,y,theta]=MEEN408_Vision();
figure()
imshow(Orig_Imag);
pause(1);
figure()
imshow(Bin_Imag);
x
y
theta
% results = msgbox('X,Y,Theta
end
```