Teaching Microcontrollers with Emphasis on Control Applications in the Undergraduate Engineering Technology Program

Dr. Wangling Yu, Purdue University, North Central

Dr. Wangling Yu is an assistant professor in the Electrical & Computer Engineering Technology Department of the Purdue University Northwest. He was a test engineer over 15 years, providing technical leadership in the certification, testing and evaluation of custom integrated security systems. He received his PhD degree in Electrical Engineering from the City University of New York in 1992, specializing in control theory and electronic technology.

Prof. Omer Farook, Purdue University Northwest

Omer Farook is a member of the faculty of Electrical and Computer Engineering Technology at Purdue University, Nothwest. Farook received the diploma of licentiate in mechanical engineering and B.S.M.E. in 1970 and 1972, respectively. He further received B.S.E.E. and M.S.E.E. in 1978 and 1983, respectively, from Illinois Institute of Technology. Farook's current interests are in the areas of embedded system design, hardware-software interfacing, digital communication, networking, image processing, and biometrics, C++, Python, PHP and Java languages. He has a keen interest in pedagogy and instruction delivery methods related to distance learning. He has a deep commitment to social justice and in achieving economic and educational equity.

Dr. Jai P. Agrawal, Purdue University Northwest

Jai P. Agrawal is a professor in electrical and computer engineering technology at Purdue University Northwest. He received his Ph.D. in electrical engineering from University of Illinois, Chicago, in 1991, dissertation in power electronics. He also received M.S. and B.S. degrees in electrical engineering from Indian Institute of Technology, Kanpur, India, in 1970 and 1968, respectively. His expertise includes analog and digital electronics design, power electronics, and optical/wireless networking systems. He has designed several models of high frequency oscilloscopes and other electronic test and measuring instruments as an entrepreneur. He has delivered invited short courses in Penang, Malaysia and Singapore. He is also the author of a textbook in power electronics, published by Prentice-Hall, Inc. His other books are, Analog and digital communication laboratory, and First course in Digital Control, published by Creatspace (Amazon). His professional career is equally divided in academia and industry. He has authored several research papers in IEEE journals and conferences. His current research is focused on renewable energy technology and wireless power transfer.

Prof. Ashfaq Ahmed P.E., Purdue University Northwest

Ashfaq Ahmed is a Professor of Electrical and Computer Engineering Technology at Purdue University Northwest. Ahmed received his bachelor's of science degree in electrical engineering from the University of Karachi in 1973 and master's of applied science degree in 1978 from University of Waterloo. He is the author of a textbook on power electronics, published by Prentice-Hall. He is a registered Professional Engineer in the state of Indiana. He is a senior member of IEEE. Ahmed's current interests include embedded system design, electric vehicle, and VHDL design.

Teaching Microcontroller with Emphasis on Control Applications In The Undergraduate Engineering Technology Program

Abstract

The paper expounds the practices utilized in teaching introductory undergraduate microcontroller's class. The microcontrollers have become ubiquitous in our daily life. They have been the engine behind automatically-controlled products and devices. As a result this course is taken by many of the non-electrical majoring students.

In this paper, we present our pedagogies for teaching a microcontroller introductory course with emphasis on detection and control applications. The proposed course uses Arduino [1], which is an open-source electronics platform, based on easy-to-use hardware and software. The course cover the architectural details of ATmega328P. The course is unique in instructing students utilizing standard C (C11 (formerly C1X) is an informal name for ISO/IEC 9899:2011) [2], the current standard for the C programming language. This approach is a departure from the plethora of code written by non-standardized coding schemes, so prevalent on the Arduino net based community. Another unique feature of instructions of the course is coding methodology. The instruction for the course is done following strict adherence to Structured Coding methodology.

Most of the technology students prefer visualization activities and hands-on experiences in their learning environment. The SparkFun Inventor's Kit [3] with Arduino Uno and other open source resources have become an effective tool for the entry-level microcontroller course. In this course, we teach necessary programming skills and knowledge of computer interfacing with input and output devices. Various types of transducers, sensors and actuators used in the course are described in the paper. Through class examples and lab experiments, students establish the concept of using microcontrollers to make open-loop and closed-loop control systems, and demonstrate knowledge learned by their course projects.

The course adhere to the teaching philosophy of Outcome Based Education [4] (OBE), as such utilizes and employ various standard tools and techniques. The paper discuss the pedagogies implemented in the course.

I. Introduction

The subject course which is the subject of this paper is a 200 level course in the Electrical and Computer Engineering Technology Department. It is an entry level microcontroller course, which is preceded by (1) a C or C++, programming course that covers the C or C++ language constructs, with emphasis on Structured Programing Methodology[5] with emphases on bit manipulation; (2) Two digital circuit's application courses that covers Combinational and Sequential Logic design utilizing Programmable Logic Devices (PLDs).

The course is offered to Electrical Engineering Technology students, who have proficient programing in C/C^{++} and electronics knowledge and want to get started to make a career out of using microcontrollers. The main purpose is to teach technology students the architectural details and application of microcontrollers.

Students taking this course are generally interested in the topic "microcontroller" because it is "tiny" yet it can "control" others. In order to help them stay interested and get familiar with the subject quickly, fast prototyping is an effective way. Students like to have something on-hand to touch and see to help them understand the knowledge, which is confirmed by student survey. For this reason, Arduino platform is a very good tool for teaching this course.

All components of the Arduino platform are open source, and it comes with a free Integrated Development Environment (IDE). There are many online resources easily obtainable by students, which makes it easy and quick for students to reach their "desire to control" before getting frustrated. However, the concept of control, sensor, and actuator are rarely found to be mentioned or taught in other courses. This often causes students having good project ideas but not knowing how to fulfill them. Therefore, we experimented including in this course the concept of closed-loop control system with variety of sensors and actuators.

II. Arduino Uno

Arduino Uno is the most popular Arduino platform in the family of the Arduino product line. The following table (Figure No. 1) compares the basic features of the various Arduinos and Arduino Compatibles platforms presently available. The user has a choice among the many Arduino platforms with regard to 1) Processor and its speed, 2) Physical footprint, 3) Number of I/O s, 4) Memory size, 5) Compatibility with the daughter boards (Shield in Arduino terminology), etc. A very important consideration to note is that the user has a large list of daughter boards to choose from in order to further define the user's specific considerations.

Arduino Uno for our considerations is typically suited due to its low cost and its versatility for class room use. Arduino Uno is based on ATmega 328P processor, belonging to AVR family of microcontrollers developed by Atmel[6]. These are modified Harvard architecture 8-bit Reduced Instruction Set Computing (RISC) single-chip microcontrollers. AVR was one of the first microcontroller families to use on-chip flash memory for program storage. Arduino Uno is available to operate at 16 MHz. out of the box. The user has three memory pools to choose from: 1) Flash memory (program space), is where the Arduino application program utilizes (sketch in Arduino terminology). 2) SRAM (static random access memory) is where the application

program creates and manipulates variables when it runs. 3) EEPROM is memory space that programmers can use to store long-term information.

Flash memory and EEPROM memory are non-volatile (the information persists after the power is turned off). SRAM is volatile and will be lost when the power is turned off. The ATmega328 chip found on the Uno has the following amounts of memory: 1) Flash 32k bytes (of which .5k is used for the bootloader), 2) SRAM 2k bytes, 3) EEPROM 1k byte.

	Processor	Processor Voltage	Supply Voltage	Flash	SRAM	Digital I/O Pins	PWM Pins	Analog Inputs	Hardware Serial Ports	Dimensions	Shield Compatibility	Notes and Special Features
Uno	16MHz Atmega 328	5v	7-12v	32Kb	2Kb	14	6	6	1	2.1"x2.7" 53x75mm	Excellent (most will work)	
Uno Ethernet	16MHz Atmega 328	5v	7-12v	32Kb	2Kb	14	6	6	1	2.1"x2.7" 53x75mm	Very Good (some pin conflicts)	Has Ethernet Port. Requires FTDI cable to program.
Mega	16MHz Atmega 2560	5v	7-12v	256Kb	8Kb	54	14	16	4	2.1"x4" 53x102mm	Good (some pinout differences)	
Mega ADK	16MHz Atmega 2560	5v	7-12v	256Kb	8Kb	54	14	16	4	2.1"x4" 53x102mm	Good (some pinout differences)	Works with Android Development Kit.
Leonardo	16MHz Atmega 32U4	5v	7-12v	32Kb	2.5Kb	20*	7	12*	1	2.1"x2.7" 53x75mm	Fair (many Pinout Differences)	Native USB capabilities. USB Micro B programming port.
Due	84MHz ARM SAM3X8E	3.3v	7-12v	512Kb	96Kb	54	12	12	4	2.1"x4" 53x102mm	Poor (voltage and pinout differences)	Fastest processor. Most memory. 2-channel DAC. USB micro B programming port. Native micro AB port.
Micro	16MHz Atmega 32U4	5v	5v	32Kb	2.5Kb	20*	7	12*	1	0.7"x1.9" 18x49mm	N/A	Smallest board size. Native USB capabilities
Flora	8MHz Atmega 32U4	3.3v	3.5-16v	32Kb	2.5Kb	8*	4	4*	1	1.75" dia 44.5mm dia	N/A	Sewable Pads. Fabric-friendly design. Native USB Capabilities
DC Boarduino	16MHz Atmega 328	5v	7-12v	32Kb	2Kb	14	6	6	1	0.8"x3" 20.5x76mm	N/A	Can build without headers or sockets for smaller size. Requires FTDI cable for programming
USB Boarduino	16MHz Atmega 328	5v	5v (USB)	32Kb	2Kb	14	6	6	1	0.8"x3" 20.5x76mm	N/A	Can build without headers or sockets for smaller size. USB Mini B programming port.
Menta	16MHz Atmega 328	5v	7-12v	32Kb	2Kb	14	6	6	1	0.8"x3" 20.5x76mm	Excellent (most will work)	Mint-Tin Size and Prototyping Area. Requires FTDI cable for programming.

Figure 1: Arduino Comparison Chart [7]

III. Sparkfun's Inventors Kit

The class utilizes Sparkfun's Inventors Kit. The kit has all the required set of parts and Sparkfun provides a set of following 16 experiments[8] on line:

Introduction: SIK RedBoard & Sparkfun Mini Inventor's Kit
Introduction: SIK Arduino Uno
Experiment 1: Blinking an LED
Experiment 2: Reading a Potentiometer
Experiment 3: Driving an RGB LED
Experiment 4: Driving Multiple LEDs
Experiment 5: Push Buttons
Experiment 6: Reading a Photoresistor
Experiment 7: Reading a Temperature Sensor
Experiment 8: Driving a Servo Motor
Experiment 9: Using a Flex Sensor
Experiment 10: Reading a Soft Potentiometer
Experiment 11: Using a Piezo Buzzer
Experiment 12: Driving a Motor
Experiment 13: Using Relays
Experiment 14: Using a Shift Register
Experiment 15: Using an LCD
Experiment 16: Simon Says

The kit provides the following set of parts:

- SparkFun RedBoard
- Arduino and Breadboard Holder
- SparkFun Inventor's Kit Guidebook
- Translucent Red Bread Board
- Carrying Case
- 16x2 White on Black LCD (with headers)
- 74HC595 Shift Register
- 2N2222 Transistors
- 1N4148 Diodes
- DC Motor with Gear
- Small Servo
- SPDT 5V Relay
- TMP36 Temp Sensor
- Flex sensor
- Softpot
- 6' SparkFun USB Cable
- Jumper Wires
- Photocell
- Tri-color LED
- Red and Yellow LEDs

- 10K Trimpot
- Piezo Buzzer
- Big 12mm Buttons
- 330 and 10K Resistors

The class utilizes the set of experiments provided as an open resource. The most important and positive aspect of these experiments is that each experiment has extensive part of theoretical engineering aspect of the discussion and reasoning in the form of comments. We have utilized these experiments as such, but the negative aspect of these experiments' code is they have not utilized standard C, and furthermore they were not designed using Structured Programing Methodology. In this class we utilized the skill set of the prerequisite class, where students learn and mastered C^{++} code design, strictly following Structured Programing Methodology. Thus our students have designed the code using standard C and following the principles of Structured Programing Methodology. All the inter-functional data communication is carried through utilizing Pointers.

IV. Architectural details of ATmega328 microcontroller

A specific objective of this course was to study the architectural details of ATmega328 microcontroller and the flow of data within a microcontroller bus system. Significant portion of the course was devoted to hardware interrupts, timers and counters and timer based counters and their use in control applications. Furthermore SPI and I2C protocols have been utilized in the labs. Students are required to refer to the ATmega328 data manual. The text book for the class utilized is, "The AVR Microcontroller and Embedded System" by Mazidi.[9]

V. Structured Programing Methodology

Here the subject is briefly introduced for the purposes of relevance with the discussion at hand. Structured programing Methodology was proposed and is being practiced as a solution to the classical problems of spaghetti code. This provide economy, reusability, and security of code. In short, avoid 1) goto statements, 2) global variables and 3) monotony of huge formless code, instead, utilize 1) break code into well-defined tasks into functions, 2) replace goto statements with function calls, 3) use local variables, and 4) use inter-functional data communication with the pointers. Inter-functional data communication with the pointers provides autonomy to functions, without writing straight jacketed code specific to memory location references, instead it make the function more abstract and lend them versatility to operate without specific reference to specific memory location.

Structured programming methodology as taught in our class and discussed here lends very well to embedded applications, hardware software interacting, control applications, robotics and Digital Signal Processing (DSP) applications to name a few. Granted there is Object Oriented Programing (OOP) methodology the choicest approach but that requires more course work in OOP, which many of Engineering / Technology students lack.

VI. Sample Code with Structured Programing Methodology

In the following we reproduce a typical Lab experiment conducted with Arduino

```
/*Lab_14
Design an application that will receive the analog signal from a pot connected to 0 - 5 volts.
The center tap of the pot is connected to ADC0 (A0). The application is going to capture the
analog signal, convert the analog value to digital value and by proper range conversion, and
display the result back to serial monitor.
*/
#include <avr/io.h>
#include <util/delay.h>
// Function prototyping or Declaring a Function
void setup (void);
void my_analog (int *, int *);
void my_serial (int *);

int main ()
{

//Invoking a Function or calling a Function
 setup ();

//Declaring local variables of main function
 int sensorPin = A0;    // select the input pin for the potentiometer
 int sensorValue = 0;  // variable to store the value coming from the sensor

//Perpetual while loop

        while (1)
        {
        //Invoking a Functions or calling a Functions

        my_analog (&sensorPin, &sensorValue);
        my_serial (&sensorValue);
        }
}

 //Body of setup function
void setup (void)
        {
        init ();
        Serial.begin (9600);
  }
```

//Body of my_analog function

```
void my_analog (int *p1, int *p2)

        {

        *p2 = analogRead (*p1);
        }
```

//Body of my_serial function

```
void my_serial (int *p3)

{
        Serial.print ("Voltage is :");
        Serial.println ((*p3 *5.0)/1023);

}
```

//Typical Output from the Application

```
Voltage is: 4.24
Voltage is: 4.23
Voltage is: 4.24
Voltage is: 4.24
Voltage is: 4.24
Voltage is: 4.24
```

Figure 2: Sample Code with Structured Programing Methodology

VII. Code Comments and Elaboration

In the sample code provided the authors have demonstrated a sample of well documented Sutured Programing Code.

The code has a “Programmer’s Block” presented with comment block demarcated with /*…. */ This is followed by #include preprocessor directive to include the header files

```
#include <avr/io.h>
#include <util/delay.h>
```

Next Functions declaration is achieved for three functions, setup, my_analog and my_serial. Each of these functions are of type “void” indicating there is no return type outputs for these functions. Setup has no input so in parenthesis is void. my_analog and my_serial has two and pointers which are local to these functions to hold the addresses of other variable to hold and operate upon.

This is followed by the main function. The code execution starts with main, it need not be declared. Every C program has to have a main function.

In the main function very first thing is call a setup function, there are arguments sent to the setup function, and does not return anything back.

Next we declare local variables, sensorPin and sensorValue of type integer meaning they are memory locations cable to hold integer type data. They are initialized meaning data of A0 and 0 are placed with assignment operators.

```
//Declaring of main function
 int sensorPin = A0;    // select the input pin for the potentiometer
 int sensorValue = 0;  // variable to store the value coming from the sensor
```

This follows by Perpetual while loop, that continuously repeats and in its body keep on calling, my_analog and my_serial functions, and proving these functions as arguments (things passed to the function) the addresses of (with '&' address of operator) variables sensorPin, sensorValue and sensorValue respectively to the two functions.

```
        my_analog (&sensorPin, &sensorValue);
        my_serial (&sensorValue);
```

Then we have three bodies of each of the three functions namely: setup, my_analog and my_serial.

Setup, function in its body is calling two more functions (init and begin functions in Serial module) which are defined in the header files attached to our application for the purpose of configuring. Serial.begin (9600) is setting up the baud rate of 9600 for serial communication of data to serial monitor.

```
        init ();
        Serial.begin (9600);
```

my_analog, function in its body is calling 'analogRead' function defined in the Arduino Library and passing the pointer in-directionally the address of variable A0.

```
*p2 = analogRead (*p1);
```

my_serial function is called and as its argument receives the address of sensorValue.

```
my_serial(&sensorValue);
```

my_serial function in its body assigns this address to pointer *p3 as its parameter. The pointer *p3 in-directionally gets to the sensorValue data is scaled, and then through the print function of serial module prints the results to serial monitor.

The perpetual while loop of main function repeats these two function's code endlessly.

VIII. Control Theory Experiments

All the classical Proportional Integral Derivative (PID) experiments could be performed because of its simplicity. No need for a "plant" model. No design to be performed. The user just installs the controller and adjusts 3 gains to get the best achievable performance. Most PID controllers nowadays are digital [10].

The classic textbook equation of PID controller is presented here:

$$u(t) = MV(t) = K_p\, e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t) \qquad [1.1]$$

Process variable (PV) is the voltage corresponding to present state of the system representing speed or temperature, etc.
Where u, the control voltage, is the signal sent to the system to take PV approaching toward a Set value (set point - SP). On the right hand side are the three contributing components, related to Proportional, Integral, and Derivative portions.

K_p , K_i , and K_d are the gains pertaining to each of the contributing components.
e(t) is the proportional error function of time corresponding to each of the contributing factors, and it corresponds to (SP – PV).

In our classroom practice we rely on the Arduino PID library [11]. For the case of detailed understanding the user is best served going through this library. An understanding of Object Oriented Programing is most helpful in fully appreciating the task at hand. For this purpose we include the Library in appendix which consist of two files, 1) PID_v1.h, which defines the class PID, and 2) PID_v1.cpp, which consists of all the member functions belonging to the class PID.

In the following example we provide a template PID control application that will read an analog input at 0, to control analog Pulse Width Modulation (PWM) output at 3.

IX. Sample Code for PID Control Applications

```
/*Lab_PID_Template
Design an application that will receive the analog signal from a pot connected to 0 - 5 volts. The
center tap of the pot is connected to ADC0 (A0). The application is going to control analog
PWM output pin 3.
*/

#include <avr/io.h>
#include <util/delay.h>
#include <PID_v1.h>
#define PIN_INPUT 0
#define PIN_OUTPUT 3
```

```
void setup(void);
void my_analog (int *, int *);
void my_serial (int *);

int main()
{
//Define Variables we'll be connecting to
double Setpoint, Input, Output;

//Specify the links and initial tuning parameters
double Kp=2, Ki=5, Kd=1;

PID myPID(&Input, &Output, &Setpoint, Kp, Ki, Kd, DIRECT);

setup();

 while(1)
 {
  Input = analogRead(PIN_INPUT);
  myPID.Compute();
  analogWrite(PIN_OUTPUT, Output);
 }
}

void setup()
{
  //initialize the variables we're linked to
  Input = analogRead(PIN_INPUT);
  Setpoint = 100;

  //turn the PID on
  myPID.SetMode(AUTOMATIC);
}
```

X. Student Satisfaction Survey

The following survey is a measurement of Students Satisfaction with regard to Course Learning Objectives:

ECET 209 Introduction to Microcontrollers – Survey Fall 2016					
Course Objectives	Students Evaluation				
	Strongly Agree	Agree	Neither Agree or Disagree	Disagree	Strongly Disagree
1. A specific objective of this course was to study the architectural details of ATmega328P microcontroller and the flow of data within a microcontroller bus system. How well did this course meet this objective?	56%	22%	22%	0%	0%
2. A specific objective of this course was to write C language program and demonstrate structured coding methodology using a microcontroller. How well did this course meet this objective?	50%	45%	5%	0%	0%
3. A specific objective of this course was to demonstrate a working knowledge of the necessary steps and methods used to interface a microcontroller system to input and output devices such as motors, sensors, displays, etc. How well did this course meet this objective?	67%	28%	5%	0%	
4. A specific objective of this course was to demonstrate the use of interrupts, timer/counters, PWM and other advanced concepts related to microcontrollers. How well did this course meet this objective	50%	34%	16%	0%	0%
5. A specific objective of this course was to complete the design, development, programming, and testing of microcontroller based open-loop and closed-loop control systems. How well did this course meet this objective?	50%	28%	22%	0%	0%

Figure 3: Course learning objectives survey

The following survey is a measurement of Students Satisfaction with regard to ABET [12] Criteria Satisfied with regard to a, b, c, d and f:

ECET 209 Introduction to Microcontrollers – ABET Survey Fall 2016					
ABET Criteria	Students Evaluation				
	Strongly Agree	Agree	Neither Agree or Disagree	Disagree	Strongly Disagree
a. As a result of this course my mastery of the knowledge, skills, and modern tools of the discipline can be rated as:	33%	45%	22%	0%	0%
b. As a result of this course my ability to apply current knowledge and adapt to emerging applications of science, engineering, and technology can be rated as:	50%	28%	22%	0%	0%
c. As a result of this course my ability to conduct, analyze, and interpret experiments can be rated as:	45%	39%	16%	0%	
d. As a result of this course my ability to apply creativity in the design of embedded systems appropriate to program objectives can be rated as:	39%	39%	22%	0%	0%
f. As a result of this course my ability to identify, analyze, and solve technical problems can be rated as:	45%	34%	21%	0%	0%

Figure 4: ABET Criteria Satisfied with regard to a, b, c, d, f

XI. Pedagogy of the Course

The pedagogy of the course is based on Outcome Based Education and utilizes the interactive model of learning. The students maintain an online portfolio of their work. The microcontroller based system designed in the laboratory to perform a specific task is the core measurement of the learning outcome of the course. The laboratory exercises are performed in teams of two students. This mode provides a platform for horizontal learning through active and engaged discourse and discussion. Students are empowered to charter their learning and feed their curiosity. The course culminates in a Final Project using AVR microcontrollers to make closed-loop control systems, and demonstrate knowledge learned in the course. These projects are assessed based upon its comprehensiveness and originality. Students are required to master the soft skills of comprehensive report writing on a weekly basis and of technical Project Report writing and project oral presentation based upon the Final Project. These classroom practices and laboratory environment provides a challenging and invigorating environment that prepares them for a lifelong learning process and career path [13].

XII. Conclusion

This paper provides the reader with a logical framework for an introductory undergraduate microcontroller's course with an emphasis on open-loop and close-loop control systems. The course guides through details of necessary C programming skills following strict observance to Structured Coding methodology. The course also demonstrates how microcontrollers interface to the real world using various types of transducers and actuators. Through the class examples and lab experiments, students establish the concept of using microcontrollers based systems and demonstrate what they have learned and to what degree they have achieved expected learning outcome through the final project.

Bibliography

[1] www.arduino.org/, Arduino - Open Source Products for Electronic Projects

[2] https://en.wikipedia.org/wiki/C11_(C_standard_revision)

[3] https://www.sparkfun.com/products/12060

[4] Omer Farook, Jai P. Agrawal, Chandra R. Sekhar, Essaid Bouktache, Ashfaq Ahmed and Mohammad Zahraee "Outcome Based Education And Assessment", Proceedings of the 2006 American Society for Engineering Education Annual Conference & Exposition June 20 -23, 2006. Chicago, IL.

[5] http://www.comp.nus.edu.sg/~hugh/TeachingStuff/cs1101c.pdf

(Structured Programing Methodology)

https://en.wikipedia.org/wiki/Structured_programming

[6] http://www.atmel.com/

[7] https://learn.adafruit.com/adafruit-arduino-selection-guide/arduino-comparison-chart

[8] https://learn.sparkfun.com/tutorials/sik-experiment-guide-for-arduino---v32/all; https://cdn.sparkfun.com/datasheets/Kits/SFE03-0012-SIK.Guide-300dpi-01.pdf; https://learn.sparkfun.com/tutorials/sik-experiment-guide-for-arduino---v32/all

[9] The Avr Microcontroller and Embedded Systems - Using Assembly and C, by Mohamed Ali Mazidi, Sarmad Naimi, and Sepehr Naimi. Prentice Hall Publiction, 2011, ISBN No. 13: 978-0-13-800331-9

[10] Digital PID Controllers, Varodom Toochinda, June 2011 https://pdfs.semanticscholar.org/49c5/d5eaf8cc27ec05bb9c5689b46a62e98d289d.pdf

[11] http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/, http://playground.arduino.cc/Code/PIDLibrary

[12] http://www.abet.org/

[13] Embedded System Design Based on Beaglebone Black with Embedded Linux. Farook, O., & Agrawal, J. P., & Ahmed, A., & Kulatunga, A., & Koyi, N. K., & Alibrahim, H. A., & Almenaies, M. (2016, June), Paper presented at 2016 ASEE Annual Conference & Exposition, New Orleans, Louisiana.

Appendix – A: PID_v1.h

```
#ifndef PID_v1_h
#define PID_v1_h
#define LIBRARY_VERSION	1.1.1

class PID
{

  public:

  //Constants used in some of the functions below
  #define AUTOMATIC	1
  #define MANUAL	0
  #define DIRECT  0
  #define REVERSE  1

  //commonly used functions
**************************************************************************
    PID(double*, double*, double*,        // * constructor.  links the PID to the Input, Output, and
        double, double, double, int);     //   Setpoint.  Initial tuning parameters are also set here

    void SetMode(int Mode);               // * sets PID to either Manual (0) or Auto (non-0)

    bool Compute();                       // * performs the PID calculation.  it should be
                                          //   called every time loop() cycles. ON/OFF and
                                          //   calculation frequency can be set using SetMode
                                          //   SetSampleTime respectively

    void SetOutputLimits(double, double); //clamps the output to a specific range. 0-255 by default, but
//it's likely the user will want to change this depending on
//the application
//available but not commonly used functions ********************************************************
    void SetTunings(double, double,       // * While most users will set the tunings once in the
                    double);              //   constructor, this function gives the user the option
                                          //   of changing tunings during runtime for Adaptive control
          void SetControllerDirection(int);       // * Sets the Direction, or "Action" of the controller. DIRECT
                                          //   means the output will increase when error is positive. REVERSE
                                          //   means the opposite.  it's very unlikely that this will be needed
                                          //   once it is set in the constructor.
    void SetSampleTime(int);              // * sets the frequency, in Milliseconds, with which
                                          //   the PID calculation is performed.  default is 100



  //Display functions ****************************************************************
          double GetKp();             // These functions query the pid for interal values.
          double GetKi();             //  they were created mainly for the pid front-end,
          double GetKd();             // where it's important to know what is actually
          int GetMode();              //  inside the PID.
          int GetDirection();                   //

  private:
          void Initialize();
```

```
        double dispKp;              // * we'll hold on to the tuning parameters in user-entered
        double dispKi;                      //   format for display purposes
        double dispKd;              //

        double kp;                      // * (P)roportional Tuning Parameter
    double ki;                      // * (I)ntegral Tuning Parameter
    double kd;                      // * (D)erivative Tuning Parameter

        int controllerDirection;

    double *myInput;            // * Pointers to the Input, Output, and Setpoint variables
    double *myOutput;           //   This creates a hard link between the variables and the
    double *mySetpoint;         //   PID, freeing the user from having to constantly tell us
                        //   what these values are.  with pointers we'll just know.

        unsigned long lastTime;
        double ITerm, lastInput;

        unsigned long SampleTime;
        double outMin, outMax;
        bool inAuto;
};
#endif
```

Appendix – B: PID_v1.cpp

```
/**********************************************************************************************
*
 * Arduino PID Library - Version 1.1.1
 * by Brett Beauregard <br3ttb@gmail.com> brettbeauregard.com
 *
 * This Library is licensed under a GPLv3 License

**********************************************************************************************
*/

#if ARDUINO >= 100
  #include "Arduino.h"
#else
  #include "WProgram.h"
#endif

#include <PID_v1.h>

/*Constructor (...)*********************************************************
 *    The parameters specified here are those for for which we can't set up
 *    reliable defaults, so we need to have the user set them.
 *********************************************************************************/
PID::PID(double* Input, double* Output, double* Setpoint,
       double Kp, double Ki, double Kd, int ControllerDirection)
{

    myOutput = Output;
    myInput = Input;
    mySetpoint = Setpoint;
            inAuto = false;

            PID::SetOutputLimits(0, 255);                //default output limit corresponds to
                                                         //the arduino pwm limits

    SampleTime = 100;                                    //default Controller Sample Time is 0.1 seconds

    PID::SetControllerDirection(ControllerDirection);
    PID::SetTunings(Kp, Ki, Kd);

    lastTime = millis()-SampleTime;
}


/* Compute() **********************************************************************
 *     This, as they say, is where the magic happens.  this function should be called
 *   every time "void loop()" executes.  the function will decide for itself whether a new
 *   pid Output needs to be computed.  returns true when the output is computed,
 *   false when nothing has been done.
 **********************************************************************************/
bool PID::Compute()
{
   if(!inAuto) return false;
   unsigned long now = millis();
   unsigned long timeChange = (now - lastTime);
```

```
   if(timeChange>=SampleTime)
   {
     /*Compute all the working error variables*/
            double input = *myInput;
     double error = *mySetpoint - input;
     ITerm+= (ki * error);
     if(ITerm > outMax) ITerm= outMax;
     else if(ITerm < outMin) ITerm= outMin;
     double dInput = (input - lastInput);

     /*Compute PID Output*/
     double output = kp * error + ITerm- kd * dInput;

            if(output > outMax) output = outMax;
     else if(output < outMin) output = outMin;
            *myOutput = output;

     /*Remember some variables for next time*/
     lastInput = input;
     lastTime = now;
            return true;
   }
   else return false;
}


/* SetTunings(...)*************************************************************
 * This function allows the controller's dynamic performance to be adjusted.
 * it's called automatically from the constructor, but tunings can also
 * be adjusted on the fly during normal operation
 ******************************************************************************/
void PID::SetTunings(double Kp, double Ki, double Kd)
{
   if (Kp<0 || Ki<0 || Kd<0) return;

   dispKp = Kp; dispKi = Ki; dispKd = Kd;

   double SampleTimeInSec = ((double)SampleTime)/1000;
   kp = Kp;
   ki = Ki * SampleTimeInSec;
   kd = Kd / SampleTimeInSec;

  if(controllerDirection ==REVERSE)
   {
     kp = (0 - kp);
     ki = (0 - ki);
     kd = (0 - kd);
   }
}

/* SetSampleTime(...) *********************************************************
 * sets the period, in Milliseconds, at which the calculation is performed
 ******************************************************************************/
void PID::SetSampleTime(int NewSampleTime)
{
   if (NewSampleTime > 0)
```

```
  {
    double ratio  = (double)NewSampleTime
                    / (double)SampleTime;
    ki *= ratio;
    kd /= ratio;
    SampleTime = (unsigned long)NewSampleTime;
  }
}

/* SetOutputLimits(...)****************************************************
 *     This function will be used far more often than SetInputLimits.  while
 *  the input to the controller will generally be in the 0-1023 range (which is
 *  the default already,)  the output will be a little different.  maybe they'll
 *  be doing a time window and will need 0-8000 or something.  or maybe they'll
 *  want to clamp it from 0-125.  who knows.  at any rate, that can all be done
 *  here.
 **************************************************************************/
void PID::SetOutputLimits(double Min, double Max)
{
  if(Min >= Max) return;
  outMin = Min;
  outMax = Max;

  if(inAuto)
  {
          if(*myOutput > outMax) *myOutput = outMax;
          else if(*myOutput < outMin) *myOutput = outMin;

          if(ITerm > outMax) ITerm= outMax;
          else if(ITerm < outMin) ITerm= outMin;
  }
}

/* SetMode(...)****************************************************************
 * Allows the controller Mode to be set to manual (0) or Automatic (non-zero)
 * when the transition from manual to auto occurs, the controller is
 * automatically initialized
 ******************************************************************************/
void PID::SetMode(int Mode)
{
   bool newAuto = (Mode == AUTOMATIC);
   if(newAuto == !inAuto)
   {  /*we just went from manual to auto*/
      PID::Initialize();
   }
   inAuto = newAuto;
}

/* Initialize()****************************************************************
 *          does all the things that need to happen to ensure a bumpless transfer
 *  from manual to automatic mode.
 ******************************************************************************/
void PID::Initialize()
{
  ITerm = *myOutput;
  lastInput = *myInput;
```

```
   if(ITerm > outMax) ITerm = outMax;
   else if(ITerm < outMin) ITerm = outMin;
}

/* SetControllerDirection(...)*************************************************
 * The PID will either be connected to a DIRECT acting process (+Output leads
 * to +Input) or a REVERSE acting process(+Output leads to -Input.)  we need to
 * know which one, because otherwise we may increase the output when we should
 * be decreasing.  This is called from the constructor.
 ******************************************************************************/
void PID::SetControllerDirection(int Direction)
{
   if(inAuto && Direction !=controllerDirection)
   {
              kp = (0 - kp);
      ki = (0 - ki);
      kd = (0 - kd);
   }
   controllerDirection = Direction;
}

/* Status Funcions*************************************************************
 * Just because you set the Kp=-1 doesn't mean it actually happened.  these
 * functions query the internal state of the PID.  they're here for display
 * purposes.  this are the functions the PID Front-end uses for example
 ******************************************************************************/
double PID::GetKp(){ return  dispKp; }
double PID::GetKi(){ return  dispKi;}
double PID::GetKd(){ return  dispKd;}
int PID::GetMode(){ return  inAuto ? AUTOMATIC : MANUAL;}
int PID::GetDirection(){ return controllerDirection;}
```