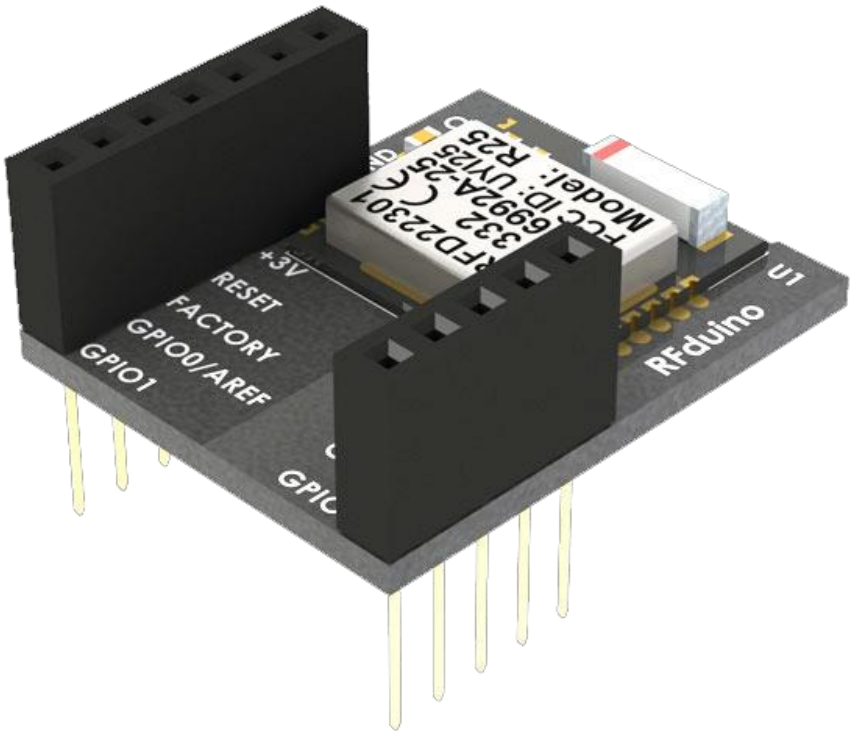
A PID Controller Using An RFduino

ECE 480: Design Team 1

Stephanie Le

04/03/2015



Abstract:

A proportional-integral-derivative controller (PID controller) is a control loop feedback mechanism widely used in industrial control systems. A PID controller calculates an error value as the difference between a measured process variable and a desired set point. The controller attempts to minimize the error by adjusting the process through use of a manipulated variable. This note will describe in detail how a PID controller is implemented in the system and how it is used to monitor the flow rate of the tool.

Table Of Contents

Keywords:

PID Controller, RFduino, Feedback, Java

Introduction:

PID controllers have been around for many years and have historically been considered to be the best controller. Originally developed from a governor device, which was used to measure and regulate the speed of a machine; it was subsequently developed and used within automatic ship steering and then for use as a pneumatic controller [1]. More recently PID controllers are now used within a wide range of applications from industrial ovens to packaging machines and are the "go-to" for most control systems.

PID Controller:

The PID controller algorithm involves three separate constant parameters, and is accordingly sometimes called three-term control: the proportional, the integral and derivative values, denoted *P*, *I*, and *D*. Simply put, these values can be interpreted in terms of time: *P* depends on the present error, *I* on the accumulation of past errors, and *D* is a prediction of future errors, based on current rate of change. The weighted sum of these three actions is used to adjust the process via a control element such as the position of a control valve, a damper, or the power supplied to a heating element [2].

Figure 1 shows the block diagram of a PID controller implemented into a system. As the diagram shows, the output is subtracted from the set point of the system to produce the error. The error is then computed into the three different control terms and then added back together. This sum is then applied to the main process in hopes of using it to minimize error in the output. Once

the process is complete, the output is then looped back to the beginning where it is subtracted from the set point once more and the PID controller is used again.

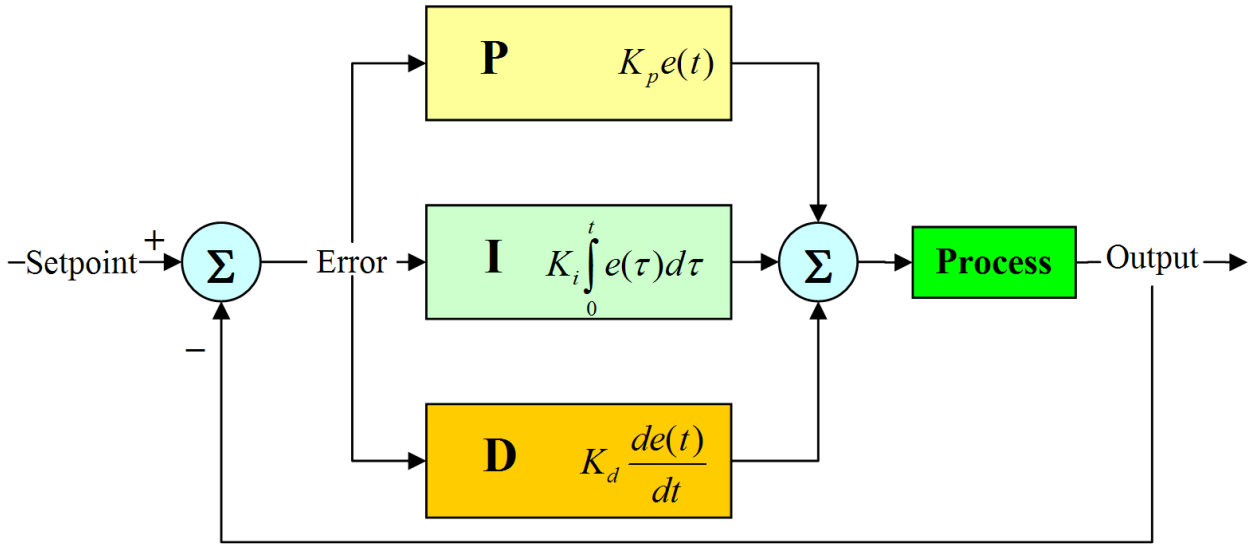


Figure 1: Block diagram of PID controller[5]

By tuning the three parameters in the PID controller algorithm, the controller can provide control action designed for specific process requirements. The response of the controller can be described in terms of the responsiveness of the controller to an error, the degree to which the controller overshoots the set point, and the degree of system oscillation.

RFduino:

The PID controller for this project will be implemented on an RFduino microcontroller. This board is a low energy BLE RF module with built-in ARM Cortex M0 Microcontroller for rapid development and prototyping projects [3]. It is a very small board, about the size of a fingertip, and is completely compatible with the Arduino interface, as seen in Figure 2.

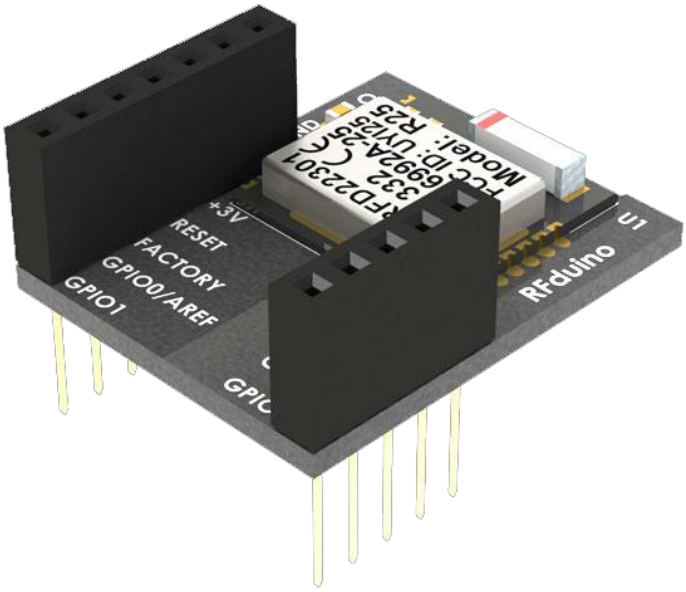


Figure 2: RFduino BLE RF module with built-in ARM Cortex M0 microcontroller[4]

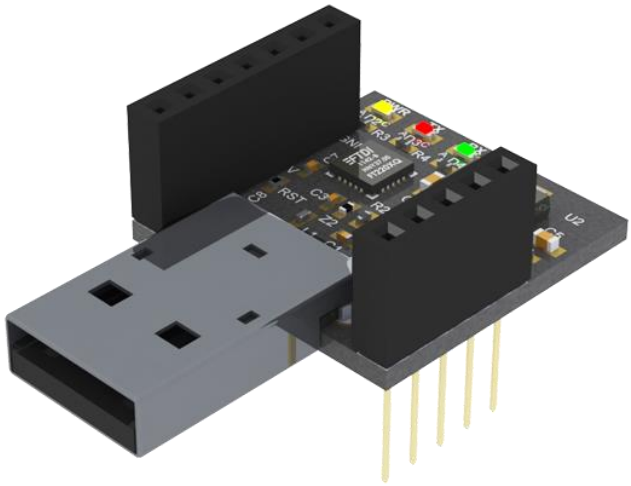


Figure 3: RFduino USB attachment module[6]

The RFduino can be programmed using the Arduino IDE by using the USB module shown in Figure 3. This microcontroller was chosen specifically for its low energy usage and its Bluetooth capabilities which will be used to connect to the user inputted information via a phone app. It is also small in size and will allow for a compact system.

Implementation:

In this project, the PID controller is used to adjust the pressure infusion bag, which is used to control the flowrate of the tool. The microcontroller will receive the user defined set point from the Bluetooth connection with the phone app. It will also be connected to a pressure sensor and will receive readings from that to use in its calculations.

The code to implement the PID controller is separated into its own public class, PIDController. The class contains various private member variables to hold values critical to computing the output including, but not limited to, the PID terms and coefficients, the output range, the max and min set point values and the error tolerance. All member variables can be seen in Figure 4 below.

```
private double m_P;                // factor for "proportional" control
private double m_I;                // factor for "integral" control
private double m_D;                // factor for "derivative" control
private double m_input;            // sensor input for pid controller
private double m_maximumOutput = 1.0;       // |maximum output|
private double m_minimumOutput = -1.0;      // |minimum output|
private double m_maximumInput = 0.0;                // maximum input - limit setpoint to this
private double m_minimumInput = 0.0;                // minimum input - limit setpoint to this
private boolean m_continuous = false;       // do the endpoints wrap around? eg. Absolute encoder
private boolean m_enabled = false;                  //is the pid controller enabled
private double m_prevError = 0.0;   // the prior sensor input (used to compute velocity)
private double m_totalError = 0.0; //the sum of the errors for use in the integral calc
private double m_tolerance = 0.05;  //the percetage error that is considered on target
private double m_setpoint = 0.0;
private double m_error = 0.0;
private double m_result = 0.0;
```

Figure 4: Member variables of PIDController class

The PIDController class is equipped with accessors and mutators for all its member variables as well as a couple other methods. The most important method where the bulk of the calculation is done is in the method *calculate()*. This method takes the external input and

calculates the adjustment value for the desired output using the P, I and D coefficients to

```
/**
 * Read the input, calculate the output accordingly, and write to the output.
 * This should only be called by the PIDTask
 * and is created during initialization.
 */
private void calculate() {

    // If enabled then proceed into controller calculations
    if (m_enabled) {

        // Calculate the error signal
        m_error = m_setpoint - m_input;

        // !!!!DEBUG!!!
        System.out.println(m_setpoint);

        // If continuous is set to true allow wrap around
        if (m_continuous) {
            if (Math.abs(m_error) >
                    (m_maximumInput - m_minimumInput) / 2) {
                if (m_error > 0) {
                    m_error = m_error - m_maximumInput + m_minimumInput;
                } else {
                    m_error = m_error +
                            m_maximumInput - m_minimumInput;
                }
            }
        }

        /* Integrate the errors as long as the upcoming integrator does
           not exceed the minimum and maximum output thresholds */
        if (((m_totalError + m_error) * m_I < m_maximumOutput) &&
                ((m_totalError + m_error) * m_I > m_minimumOutput)) {
            m_totalError += m_error;
        }

        // Perform the primary PID calculation
        m_result = (m_P * m_error + m_I * m_totalError + m_D * (m_error - m_prevError));

        // Set the current error to the previous error for the next cycle
        m_prevError = m_error;

        // Make sure the final result is within bounds
        if (m_result > m_maximumOutput) {
            m_result = m_maximumOutput;
        } else if (m_result < m_minimumOutput) {
            m_result = m_minimumOutput;
        }
    }
}
```

calculate the error values. The code for the *calculate()* method can be seen in Figure 5.

This is a private method and is called by *performPID()*, which returns the result of all the calculations and is called after all values have been initialized and the PID controller has been enabled.

Conclusion:

PID controllers are widely used and seen as the most helpful controller. By combining the three different error prediction terms, the controller can give an accurate adjustment of the output to compensate for the possible error. In this project it is used to track and adjust the pressure infusion bag which controls the flowrate of the tool. This is an instrumental part of the system and is achieved by using the PIDController class as well as external inputs from the phone app and pressure sensor.

References

[1] N/A, "Tecnologic UK," [Online]. Available: http://www.t-uk.co.uk/articles/history-pid-controllers. [Accessed 3 April 2015].

[2] A. O'Dwyer, "PID control: the early years," in *Dublin Institute of Technology*, Dublin, 2005.

[3] "RFduino Datasheet," RFduino, Hermosa Beach, 2013.

[4] [Online]. Available: http://www.rfduino.com/wp-content/uploads/2014/03/RFD22102.Prospective.Top_.png.

[5] [Online]. Available: http://stm32f4-discovery.com/wp-content/uploads/pid-controller-diagram.png.

[6] [Online]. Available: http://www.rfduino.com/wp-content/uploads/2014/03/RFD22121.Prospective.Top_.png.