Portland State University

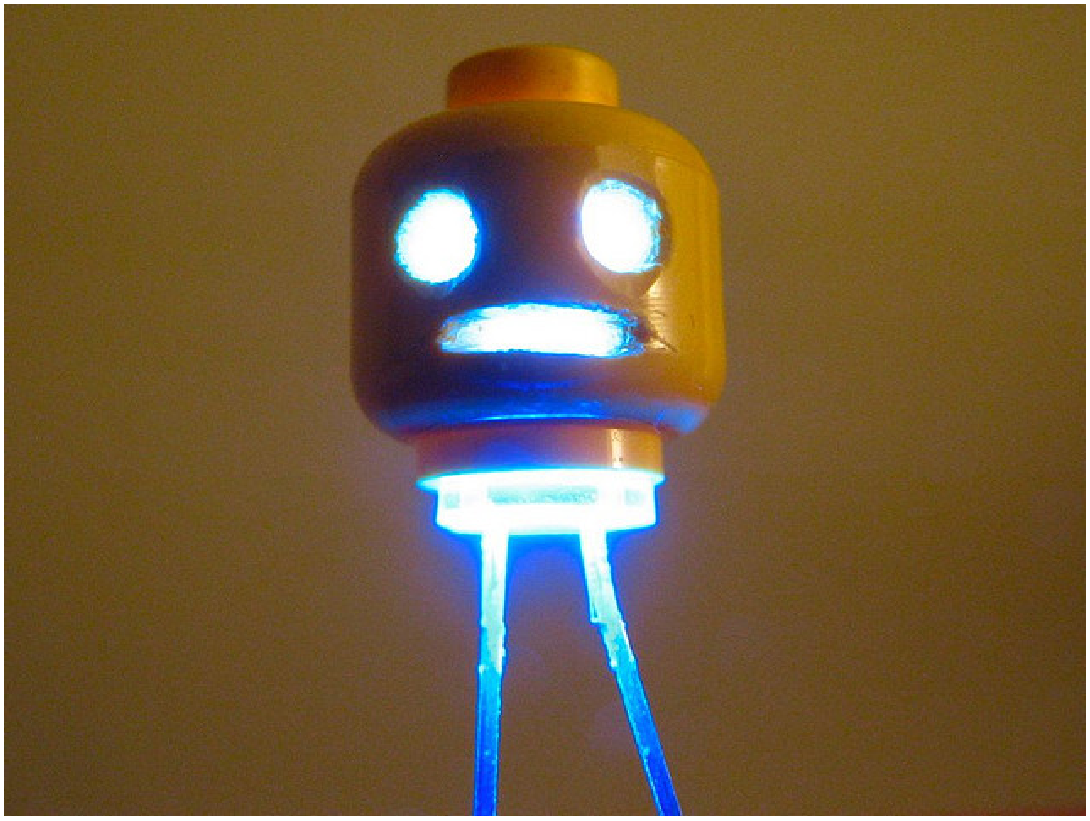
Embedded Systems on FPGAs

ECE544

Closed-Loop Control With PWM

Erik Rhodes Caren Zgheib

May 12, 2014



1 Introduction

This project demonstrates how closed-loop control can be used in modern applications. Our embedded system monitored the output (brightness) of an LED with a light sensor, and calculated the corresponding PWM output to stabilize the brightness at the desired setpoint.

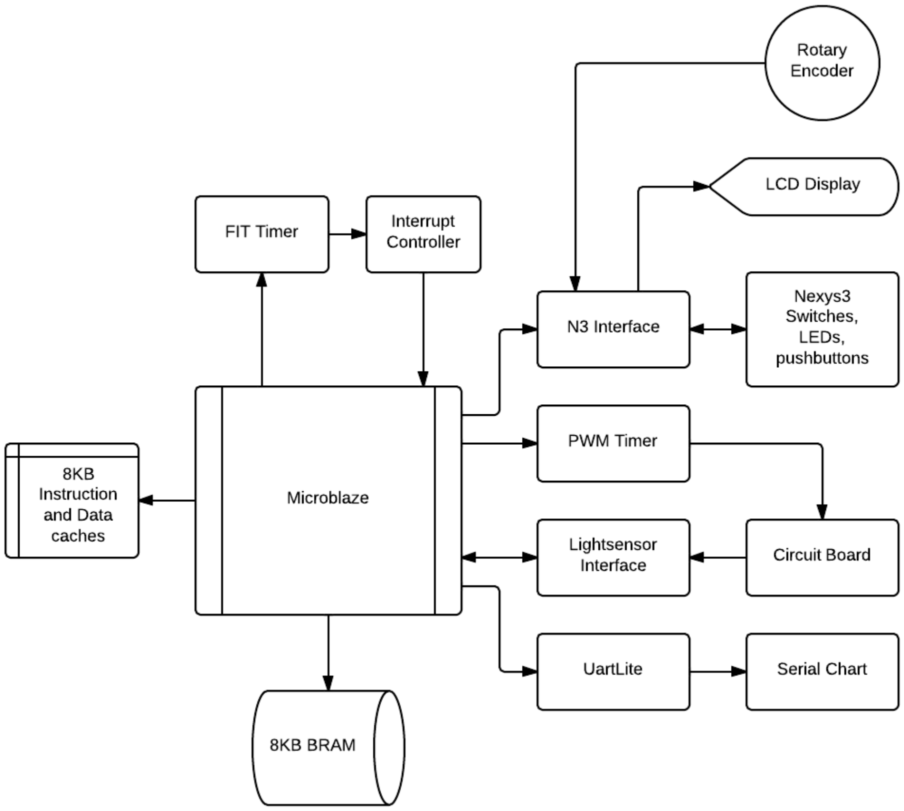


Figure 1: Block Diagram of Control System

2 System Design

As seen in figure 1, our embedded system uses... It also includes instruction and data caches, and an extra 8Kb local BRAM. The GPIO pins are only used for debugging purposes. Our light sensor peripheral interacts with the circuit we built on a breadboard. This connects the PWM generator to the LED, and receives the detected brightness from the light sensor through another pin. The Uartlite hardware sends the computed data to the external hardware serially through a USB port. This data can then be seen using different programs, such as Putty or Serial Charter. The graphs showing our results (reference) were created using Serial Charter. The N3EIF interface controls the communication to the LCD display and the rotary encoder.

3 Implementation

3.1 Control Methodology

Being able to control a dynamic system is an integral part of many everyday systems. Accomplishing this requires a feedback loop with an external measurement of the output being fed back into the system. These systems can be controlled in a number of ways. Two popular methods are **Bang Bang** and **PID**.

3.1.1 Bang Bang

Bang bang is quite straightforward: if the output is lower than desired, the controller puts the control signal at its highest amount. Likewise, if the output is higher than desired, the signal is set to the lowest level. While sometimes effective, this is a crude and cheap method for controlling systems, and is usually only used in devices where accuracy is not extremely important.

Listing 1: Bang Bang Algorithm

```
for (smpl_idx = 1; smpl_idx < NUM_FRQ_SAMPLES; smpl_idx++)
    {
        sample[smpl_idx] = LIGHTSENSOR_Capture(LIGHTSENSOR_BASEADDR, slope, offset
            , is_scaled, freq_min_cnt);
        volt_out = (-3.3 / 4095.0) * (sample[smpl_idx]) + 3.3;
        if (volt_out < setpoint)
        {
            Status = PWM_SetParams(&PWMTimerInst, pwm_freq, MAX_DUTY);
            delay_msecs(1);
            if (Status == XST_SUCCESS)
            {
                PWM_Start(&PWMTimerInst);
            }
        }
        else
        {
            Status = PWM_SetParams(&PWMTimerInst, pwm_freq, MIN_DUTY);
            delay_msecs(1);
            if (Status == XST_SUCCESS)
            {
                PWM_Start(&PWMTimerInst);
            }
        }
        delay_msecs(100);
    }
```

3.1.2 PID

The more prevalent method, PID, involves making multiple calculations to predict the accurately control the behavior of the output. The proportional (P) method involves using the previous error margin to calculate the next appropriate one. The integral (I) method calculates how the system behaves over time, and the derivative (D) calculates how fast the output is changing at that point in time. Using a specific arrangement of these parameters will yield a large increase in accuracy, so tuning the application is desirable. When tuning, we first attempted to follow the Ziegler-Nichols method. After understanding how each method changes the output, we modified the parameters as we saw fit to acquire the most accurate configuration for our control system.

Listing 2: PID Algorithm

```
for (smpl_idx = 1; smpl_idx < NUM_FRQ_SAMPLES; smpl_idx++)
{
    delay_msecs(100);

    // get count from light sensor and convert to voltage
    sample[smpl_idx] = LIGHTSENSOR_Capture(LIGHTSENSOR_BASEADDR, slope, offset,
        is_scaled, freq_min_cnt);
    volt_out = (-3.3 / 4095.0) * (sample[smpl_idx]) + 3.3;

    // calculate derivative;
    error = setpoint - volt_out;
    deriv = error - prev_error;

    // calculate integral
    if (error < setpoint/10) integral += error;
    else integral = 0;

    // Control offset is gotten from characterization
    volt_out = offset + (error * prop_gain) + (deriv * deriv_gain) + (integral
        * integral_gain);
    duty_out = (volt_out)* (MAX_DUTY+1)/VOLT_MAX;

    // establish bounds
    if (duty_out < 1) duty_out = 1;
    if (duty_out > 99)duty_out = 99;

    // activate PWM
    Status = PWM_SetParams(&PWMTimerInst, pwm_freq, duty_out);
    if (Status == XST_SUCCESS)  PWM_Start(&PWMTimerInst);
}
```

3.2 Peripheral Interface

The **TSL237** light to frequency converter outputs a period that directly corresponds to the intensity of the light emitted from the LED. The PWM detection module receives this information and converts it to a "count", which is then scaled to fit within the parameters (between 0 and 4095) needed for our control calculations. These functions are handled in the light sensor driver. The driver was supposed to convert the scaled counts to a voltage however we decided to move that functionality to the program application. The peripheral has the following user-visible registers:

Register	Number	Description
Control	slv_reg0	Enable of the peripheral. Format: RESERVED[30:0], EN
Status	slv_reg1	Current Settings of the peripheral. Format: RESERVED[30:0], EN
HighTime	slv_reg2	Detected high time count. Format: HighTime[31:0]
Period	slv_reg3	Detected period count. Format: Period[31:0]
SpareReg1	slv_reg4	Spare register. No specific purpose. Format: RESERVED[31:0]
SpareReg2	slv_reg5	Spare register. No specific purpose. Format: RESERVED[31:0]

Table 1: Lightsensor peripheral registers

3.3 Peripheral Driver

The `lightsensor` driver enables the software application to communicate with the external peripheral. It has six functions which are as follows:

- **LIGHTSENSOR_Init()** Initializes the `lightsensor` peripheral driver. It waits until the light sensor self test is done then it sets the control enable bit to 0.
- **LIGHTSENSOR_Start()** - Starts the PWM detection in the peripheral. This function sets the Control Enable bit to 1.
- **LIGHTSENSOR_Stop()** - Stops the PWM detection in the peripheral. This function sets the Control Enable bit to 0.
- **LIGHTSENSOR_Capture()** - This function returns the period count by reading the PERIOD register. If the characterize function has been called already, the `is_scaled` boolean would be set to true and the capture function returns the scaled period count based on this formula: count = (Xuint32)(slope * (period - min)+ 1).
- **LIGHTSENSOR_SetScaling()** - Not implemented since we decided to do the scaling directly in the application program.
- **LIGHTSENSOR_Count2Volts()** - Also not used because we do the conversion in the program directly.

3.4 Frequency detection

The measurements output by the light sensor were received by a hardware module that interpreted the data. This Verilog program closely resembles the PWM detection module done in project 1. It increments the count on each clock edge, depending on whether the input is high or low. Once a full period is detected, the high count and period are sent to registers which are able to be read by the light sensor driver.

3.5 User Controls

The program starts by characterizing the system. This allows the initial scaling to be perform, which varies by the amount of light detected by the light sensor. After this is done, the default control parameter values are displayed and the user is given a chance to change them. The type of test and starting input voltage can also be selected. Once these have been configured, the user starts the test with the rotary button. When it has finished, long pressing on the rotary button sends the data to the computer connected serially via the UART port. If the user wants to change the test, they can modify the switch values and update the LCD by pressing on the rotary button. While he user interface remained close to the recommended specifications, it also included a few features that made it amazingly incredible.

4 Results

4.1 Characterization

Fairly linear

Name	**Value**	**Function**
Switch[1:0]	00	Bang Bang
Switch[1:0]	01	PID
Switch[1:0]	10	Unused
Switch[1:0]	11	Characterization
Switch[2]	0	Vin Low
Switch[2]	1	Vin High
Pushbutton	North	Move Cursor
Pushbutton	East	Increase Value
Pushbutton	West	Decrease Value
LCD Display	Row 1	PID Values
LCD Display	Row 2	Setpoint and Offset
Rotary Encoder	Clockwise	Increase Setpoint
Rotary Encoder	Counter-Clockwise	Decrease Setpoint
Rotary Encoder	Press Button	Next section
Rotary Encoder	Long Press Button	Initiate test

Table 2: Nexys3 Controls

4.2 Bang Bang

Oscillates back and forth with an error of ...?

4.3 PID

Show a few different examples of some good results and list the parameter values.

5 Conclusion

Our work was split up as follows:

	Erik	Caren
Peripheral		✓
Lightsensor Driver		✓
User application	✓	
Integration	✓	✓
Documentation	✓	✓

Table 3: Division of Tasks

5.1 Challenges

- **"Tools"**: The Xilinx tool chain was quite a hassle to deal with. There was also a problem connecting one of our circuit boards to the Nexys3. Additionally, the UART serial USB connection also caused repeated BSOD's on our Windows 8 computer.

- **Hardware interfacing**: Although we took the same module used in project one for the hardware PWM detection, it did not work. After narrowing down this problem, we swapped the module out and were able to receive reliable results.
- **Debugging**: Incorrect readings,
- **Typecasting**: In several different places, using the wrong type gave us incorrect values. Making sure each variable was initialize and cast to the correct type took time, and before this was done we received incorrect or imprecise values.
- **Integration**: We divided the project into two different parts, which were both written in a relatively quick amount of time. Given the number of components in this system, however, it was quite a challenge to integrate and fix all the different issues that occurred when they were merged together.

While this project is certainly good practice for the real world, the amount of problems we faced was quite numerous. It is especially frustrating when the roadblocks we faced were not actually involving our code or the algorithms we were using. Instead, much of our time was spent wasted on getting the tools to cooperate. This lead to us spending at least 40 hours each on this project. With that being said, the concept of closed-loop control was quite interesting, and the results we finally achieved were satisfying.