California Polytechnic State University,
San Luis Obispo

Computer Engineering 462

Senior Project

Efficacy of Replacing TI Launchpad with Pyboard in CPE 329 Curriculum: Final Report

Authors:
Tony Miller
Dominic Romualdo

Advisor:
Dr. John Oliver

June 11, 2015

Table of Contents

1 Executive Summary

The goal of the current project was to determine if the PyBoard microcontroller is a suitable replacement for the Texas Instruments MSP-430 Launchpad in the Computer Engineering 329 Microcontrollers class at Cal Poly. The major projects from CPE 329 where attempted using the PyBoard in place of the MSP-430. The PyBoard was found to be a powerful and fairly capable platform for learning to develop embedded systems. Some of the benefits of the PyBoard were found to be it's ease of rapid prototyping, a fairly large collection of libraries with useful tools for microcontrollers, and many powerful abstractions that take care of tedious programming details automatically. The major downsides to the PyBoard are the opaqueness of the abstractions used and the decidedly less comprehensive documentation available for many of it's libraries. Another concern with using the PyBoard is that it does not prepare students to use the embedded industry standard C programming language.

2 Introduction

2.1 Background

2.1.1 MSP-430

The microcontroller currently used in CPE 329 is the Texas Instruments MSP430G2553, which is part of the MSP-EXP430G2 Launchpad development kit. The MSP430G2553 has 16 kilobytes of flash memory and 512 bytes of RAM. It features a ten bit analog to digital converter, two sixteen bit timers, and can communicate using the SPI, UART and I2C protocols.[6]//

The Launchpad development board itself has twenty configurable I/O pins easily accessible on it's perimeter for communication with the pins of the on-board microcontroller. It also features several LEDs and buttons that can be read from or written to through the microcontroller's I/O pins. Among other feautures, the Launchpad also has a micro USB connection for interfacing with a computer. This allows the user to easily transfer compiled code directly from the computer to the microcontroller in addition to debugging the microcontroller's currently running program from a connected

computer.

2.1.2 PyBoard

At the heart of the PyBoard is the STM32F405RGT6 microcontroller from STMicroelectronics. It is a 32bit ARM-based core featuring a RISC architecture with 1 Megabyte of flash memory. Among other features the STM32F405RGT6 offers twelve 16-bit timers, two 32-bit timers, two I2C full duplex, four USART and two UART interfaces in addition to a 32 bit Random Number Generator. The core clock runs at frequencies up to 168 MHz.

2.1.3 CPE 329

In CPE-329, students are expected to learn how to design and implement embedded systems to sense and actuate external forces. To do this, students will learn how to implement hardware interrupts, software interrupts, use hardware timers to generate pulse width modulation waves, interface and integrate sensors into their system, sample and generate analog signals through the use of A/D and D/A converters, and communicate with other devices through UART, SPI, or I2C. Other goals for this course include: learning how to read a datasheet, learning how to represent a project through state and blackbox diagrams, and developing the necessary skills to excel in a team.

3 Projects

3.1 Project 1 : Hello World

3.1.1 Assignment

This is a brief project aimed at familiarizing students with some of the most basic features of the PyBoard. The end result of this project should be a software and hardware based solution to the classic beginning programming task of displaying the text "Hello World". Since the PyBoard has no screen to write to, it is interfaced with an external LCD. This assignment familiarizes

the student with the usage of general purpose input/output (GPIO) pins and software delays on the PyBoard.

3.1.1.1 Objectives:

1. Interface with an external component
2. Use built-in timer of PyBoard to properly communicate with external LCD
3. Learn techniques of minimizing microcontroller pin usage

3.1.1.2 Introduction:

Microcontrollers are very capable devices that can control many different peripherals through General Purpose Input/Output (GPIO) pins. However, due to the limited size of the circuit board on which they are placed, they must have a limited number of available GPIO pins and consequently a limited number of peripherals with which they can communicate. To make a system communicate with more peripherals past a certain point difficult design decisions must be made. On the one hand more microcontrollers could be incorporated into the system to account for the limited number of pins. While this may seem like a good solution, it has several drawbacks. First of all, the extra microcontroller will require extra power to run, which can lead to significant energy usage over extended periods of uptime. Secondly, this microcontroller is essentially on its own and cannot communicate with the other microcontroller unless precious pins are used for communication between the two microcontrollers.

There are alternative solutions to the problem of limited pins. One such alternative is "nibble mode" on peripherals. This mode essentially reduces the number of required pins by nearly a half compared with normal operation. In nibble mode the same information is transferred, just over fewer wires. This feat is accomplished by exchanging lower pin usage with longer data transfer time. For instance, instead of using 8 pins to transfer 8 bits of information in one millisecond, nibble mode may transfer the same 8 bits of information on 4 pins in two milliseconds. The first four bits of data would be sent the first millisecond and the second four bits sent the second millisecond. Often times in the design of systems such a delay will be a much

more acceptable alternative to vastly increased pin usage, and sometimes transmission speed is so important that pin usage must go up. This is just one decision a system designer must make.

3.1.1.3 Procedure:

1. Read the datasheet for the provided external LCD module
   (a) Pay special attention to the write cycle for "nibble mode".
2. Develop a schematic for writing messages to the LCD from the PyBoard
   (a) The design must utilize the "nibble mode" of the LCD.
   (b) Remember to include an external resistor for the brightness control of the LCD.
3. Develop a script to write a string to the LCD
   (a) Pay special attention to the timings specified in the LCD's documentation.
   (b) Use the delay() function and the Pin class methods to communicate with the LCD.

3.1.2 Approach Taken

This project required cooperation between hardware and software. To successfully make characters display on the LCD requires the LCD and PyBoard to communicate in not only the right format but also with the right timing between commands. These delays where implemented in software. The software is only half the story. Physical connections between the PyBoard,Power Supply and LCD are required for the PyBoard and LCD to function at all. Also, connections between the data, enable and reset pins on the LCD must be made to the appropriate pins on the PyBoard, as specified in the program it is running.

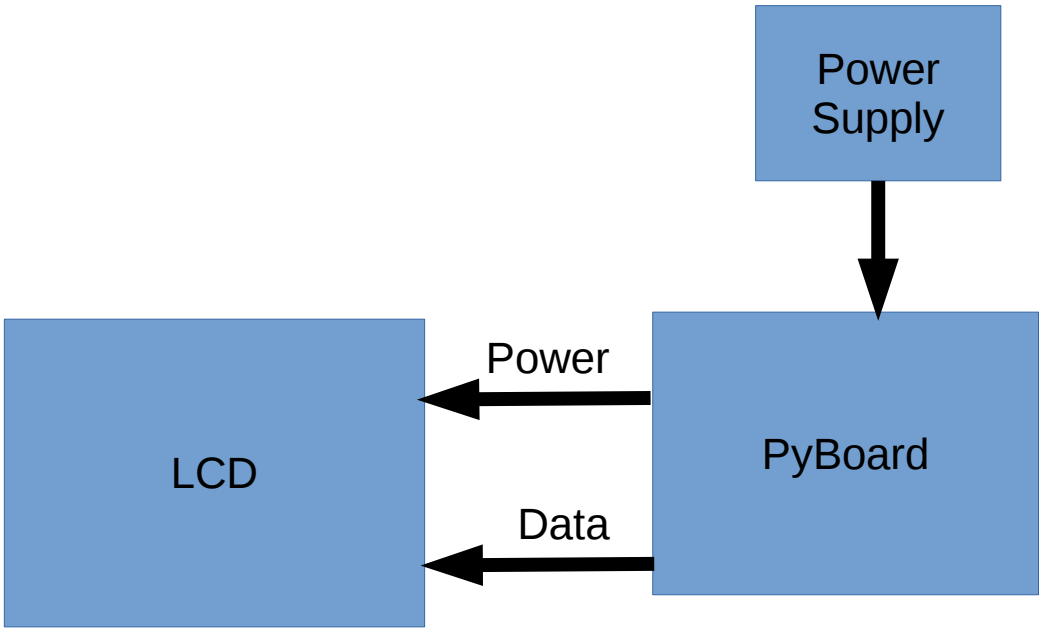

Figure 1: Block Diagram for the Hello World Project

3.1.2.1 Software Solution

The software portion of this project was accomplished by writing two functions: an LCD initialization function and a function to write one character to the initialized LCD. The initialization function sets parameters of the LCD like configuring it for nibble mode (which uses four data lines instead of eight) and the size of the characters on the screen. This function uses the pyb.delay() function to set pin values (using Pin.value(val)) at specific times in the LCD's start-up sequence as specified in the timing information provided in the LCD's documentation. The character writing function uses the same two functions to send the character data at the proper times to the LCD. The complete software solution for this project can be seen in Listing 1.

3.1.2.2 Hardware Solution

For this project a computer acted as the Power Supply shown in Figure 1. This connection was made with a standard micro USB cable. The VIN pin on the PyBoard was connected to V+ on the LCD. A GND pin on the PyBoard was connected to GND on the LCD. V_0 on the LCD controls the contrast

of the LCD. A 10kΩ led from V_0 to the common GND. Four of the GPIO pins on the PyBoard where connected to four of the data connections on the LCD. The Reset pin on the LCD was connected to X4 on the PyBoard, and the LCD's Enable pin was connected to X3. Finally the R/ W pin on the LCD was connected directly to common GND since data only needs to be written to and not read from the LCD. The wiring schematic can be seen in Figure 2.

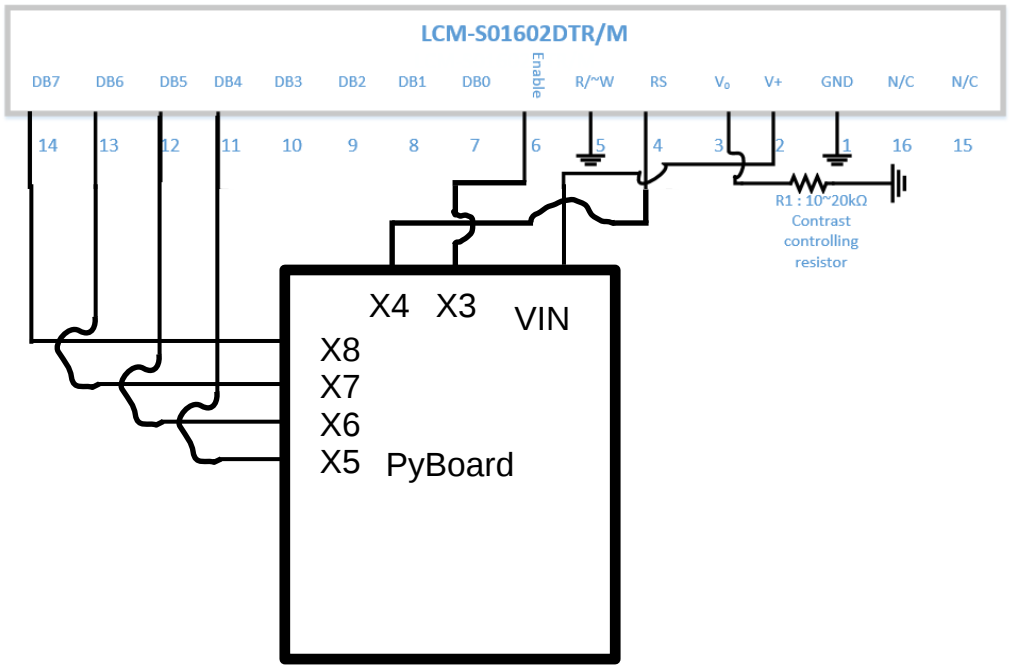

Figure 2: Schematic for Hello World Project showing LCD and PyBoard

3.1.3 Review

The chosen approach to this lab was very similar to the approach taken for the MSP-430 in CPE 329. The only major difference between the approaches taken for the two microcontrollers is the way pins on the board are accessed. On the MSP-430 the value of a pin is set or read by setting or getting the value of a bit in a particular area of memory corresponding to that pin. Accessing pins on the PyBoard is very different. Instead a Pin type object is created which corresponds to a physical pin based on the arguments passed to the

object's constructor and the Pin.value([*val*]) function is used to get that Pin's current value (if the optional argument *val* is not present) or to set the Pin's value to *val*. This is a rather significant abstraction made on the PyBoard in that it disguises the low level memory manipulation in a seemingly simple Object-Oriented set/get function. It also removes the requirement that the student understand the concept of bit-flags to access a specific bit within an integer, which often proves to be a useful skill to have in Computer Science.

3.2 Project 2 : Digital to Analog Conversion

3.2.1 Assignment

The aim of this project is to familiarize the students with hardware timers as well as how to use the Digital to Analog conversion class on the Pyboard. After this project, students should have a strong understanding of how to control the frequency of pulse width modulation waves with the Pyboard.

3.2.1.1 Objectives:

1. Use built-in timer of Pyboard to generate different frequencies
2. Use built-in Digital to Analog converter to create square, sine, and sawtooth waveforms

3.2.1.2 Introduction:

An important use of microcontrollers is the generation of pulse width modulations. Pulse width modulations are used to control the power supplied to external devices. A common use of pulse width modulations is to control the power given to a motor that is connected to the microcontroller. The power is controlled by quickly switching between an on and off state. Based on the frequency that switching occurs at, the external device will have a different average value of voltage. The duty cycle of the waveform that is generated also affects the average voltage level given to the external device. An advantage of using pulse width modulation is that it reduces the amount of power used by the device because in the off state there is no current flow.

For this project, the PyBoard will be used to create a simple function generator. The function generator will be able to create three different waveform: a square wave, sine wave, and sawtooth wave. The function generator will also create the waveforms in 5 different frequencies: 100 Hz, 200 Hz, 300 Hz, 400 Hz, and 500 Hz. Finally, the function generator should allow the adjustment of the duty cycle of the square wave. It should have duty cycles from 0 percent all the way to 100 percent in 10 percent increments.

3.2.1.3 Procedure

1. Read the documentation in [4] to understand how to use the hardware timers on the PyBoard
   (a) Pay special attention to the creation of the timers, and how to use callback functions
2. Read the documentation provided in[2] to understand how to use the internal DAC
3. Develop a function to use in the callback of a timer
4. Develop functions to handle the debouncing of buttons, and to adjust the waveform, frequency, and duty cycle of the output wave.

3.2.2 Approach Taken

This project requires an understanding of the Timer and DAC classes on the Pyboard. Once they are understood, the code can be written in main.py to create the function generator. This project will require at least two external buttons so that you can adjust the three values in the waveforms. The third button is the onboard 'usr' button that is available.

3.2.3 Software Solution

This project was accomplished by creating one timer object and one DAC object. The timer object is set to run at a certain frequency and uses a callback function to create the waveform using the DAC object. This project needed four functions: a function to process a button press and change the

waveform, a function that to process a button press and adjust the frequency, a button to process the press of the usr button, and a function to be used by the callback of the timer. To create the sine and saw waveforms, dynamic calculations or look-up tables could be created. The use of look-up tables is popular because some microcontrollers cannot dynamically calculate the next value for the DAC to output. A complete software solution can be seen in Listing 2

3.2.4 Hardware Solution

For this project, the USB connection to a computer was used to power the board. This means that on the Vin pins, you can get 5V. An oscilloscope probe will need to be attached to the pin you use for the internal DAC. Buttons will also need to be connected to some of the GPIO's on the board. A valid function generator schematic is shown in Figure 3.

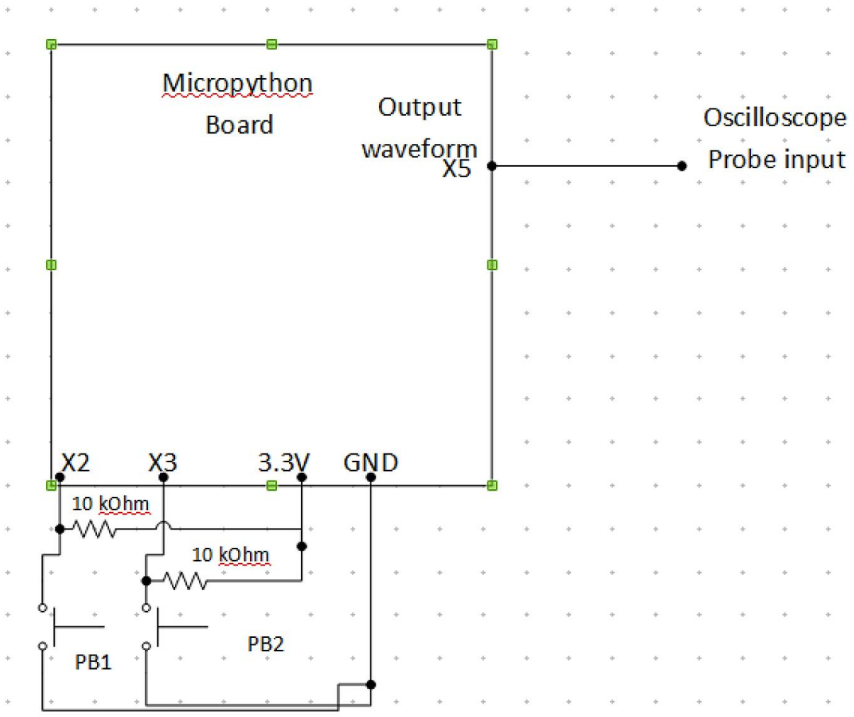

Figure 3: Function generator schematic

3.2.5 Review

The chosen approach for this microcontroller was different than on the MSP-430 or the ATMega-328P. The PyBoard has an onboard DAC so it was used instead of an external DAC. This major difference makes it so students will not have to learn how to use the SPI communication prototcol to communicate with an external DAC. When trying to use an external DAC with SPI communication some issues with the PyBoard were discovered. There seemed to be a limit in regards to how fast the PyBoard could communicate using SPI as waveforms were limited to less than 200 Hz no matter how fast the timer object was. After this was discovered, the internal DAC was used instead to prove that a function generator could be created with the PyBoard.

3.3 Project 3 : External Sensor

3.3.1 Assignment

This project's aim is to familiarize students with the ADC and I2c classes on the Pyboard.It also aims to teach the students how to read and understand data sheets as it is a valuable skill to have.

3.3.1.1 Objectives:

1. Interface with an external sensor
2. Use a standard communication protocol to read input from the external sensor

3.3.1.2 Introduction:

One of the strongest selling points of microcontrollers is their ability to easily communicate with many external devices to collect realtime data regarding their surroundings and react to this information appropriately. Standardised communication protocols make interfacing with sensors very simple. These protocols make several guarantees on the format and transfer speed of data so adding sensors to an existing product has the potential to be straightforward. This is a broad and open ended project. The student may select an external sensor of their choosing to interface with the PyBoard using one of the standard communication protocols such as SPI, I2C, or UART.

3.3.1.3 Procedure:

1. Select a sensor to interface with the PyBoard
   (a) The sensor should communicate using a standard communication protocol such as SPI, I2C, or UART.
2. If using an analog sensor, read documentation in [1] to understand how to use the ADC pins on the Pyboard.
3. Review the documentation available for the sensor being used

(a) Pay attention to power requirements for the sensor and slight deviations from the standard protocols some sensors make.

(b) Determine the neccessary steps to successfully calibrate the sensor

4. Develop a schematic for the circuit that connects the PyBoard with the external sensor

5. Develop a script to interface the PyBoard with the external sensor

   (a) The script will most likely need to perform some kind of calibration of the sensor before it can be used.

   (b) Once the sensor is calibrated, read the values it sends and output the results in some human-readable format.

      i. Potential options include printing the values read and viewing them by connecting to the serial port of the PyBoard from the computer or writing the results to the LCD using the code developed in Project 1.

3.3.2 Project 3a : Capacitve Touch Sensor

3.3.2.1 Approach Taken

The LCD touch sensor skin communicates using the I2C protocol and can be powered by the roughly 5 V VIN pin on the PyBoard. A provided library was used to interface with the four capacitive touch sensors[5]. Using this library is not required, though it does provide several useful calibration features for debouncing the output from the sensors. Even when using this library, some light adjustment had to be made to the output to get the desired functionality. The LCD on the same skin was used to verify the correctness of the touch sensors. A simple fill function was developed to fill the quadrant of the LCD corresponding to the sensor that detects touch.

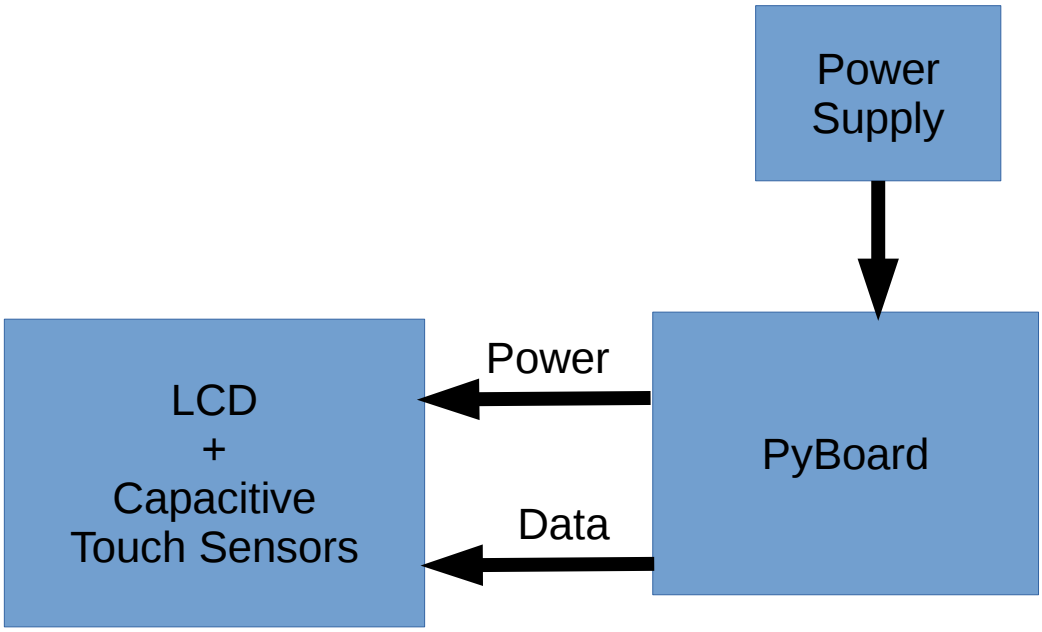

Figure 4: Block Diagram for the Capacitive Touch Sensor Project

3.3.2.2 Software Solution

Developing the software for this project was very straight-forward. Using the aforementioned mpr121 library makes getting the status of each of the four capacitive sensing buttons a single function call. The library itself is very straightforward and simple to understand. It takes care of the I2C communication between the PyBoard and the sensor. The only thing left to do was create a simple fill command that sets the pixels in the quadrant specified by the corresponding capacitive button's letter. One issue that did come up was when a touch was sensed by the 'Y' button it would occasionally also register as a touch on the 'B' button. The complete software solution for this project can be seen in Listing 3.

3.3.2.3 Hardware Solution

For this project a computer acted as the Power Supply shown in Figure 4. This connection was made with a standard micro USB cable. The VIN , 3V3, and GND pins on the PyBoard were connected to their namesakes on the

LCD and capacitive touch sensor skin. I2C communication was handled by pins Y9 and Y10 on the PyBoard being connected to TX/SCL and RX/SDA on the skin. The remaining pins allow for bidirectional communication with the LCD on the same skin. The similarities in naming are no coincidence; the LCD and Capacitive Touch Skin was developed specifically for the PyBoard. The wiring schematic can be seen in Figure 5.

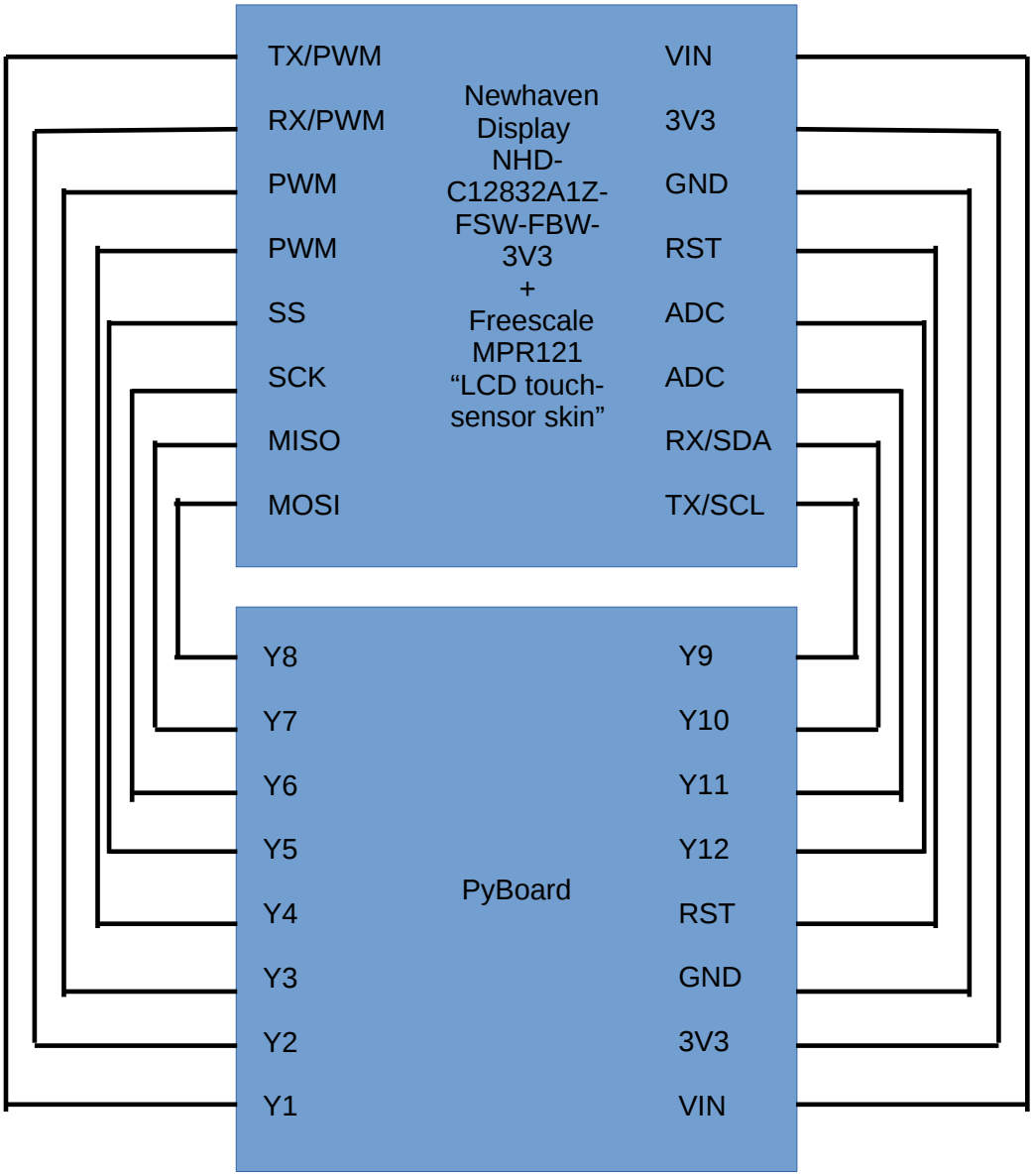

Figure 5: Schematic for Capacitive Touch Sensor Project

3.3.2.4 Review

The LCD touch sensor skin was rather easy to interface with the PyBoard. Calibration was not quite as straightforward but the sensor output was still

quite usable out of the box. It is difficult to compare the results of interfacing different sensors with different microcontrollers, though it would have been considerably less intuitive and perhaps more involved in a non-object oriented programming language such as C on the MSP-430. The I2C library for the PyBoard makes communicating with other components nearly as simple as writing to a file, with very intuitive functions such as send() and recv(). That the LCD and capacitive touch skin is designed with the PyBoard in mind can be both positve and negative for the learning experience. On the one hand, the hardware connections where very simple to make so more time could be spent on developing the program. This is also a major downside, since in the real world you may need to interface with a highly specialized sensor that is not in common use and therefore it is not in the microcontroller's "ecosystem" of plug-and-play sensor packages. This is why it is important to teach the lower-level standard protocols like I2C and SPI as they are much more broadly applicable.

3.3.3 Project 3b : Accelerometer

3.3.3.1 Approach Taken

The ADXL335 accelerometer communicates with the Pyboard with analog signals. This means that for each data pin there must be an associated ADC pin on the Pyboard. For this project, only the data from the x and y signal was used to create a protractor that would print the angle to the LCD screen.

3.3.4 Software Solution

The software for this project is very simple. This project requires functions that allow printing to the LCD screen seen in the first project. Two ADC objects will need to be created to read the two data lines from the accelerometer. Once they have been read, they must be fed into the atan() function that is include in the math library. The value returned from the atan function is then printed to the LCD. A software solution is shown in Listing 4.

3.3.5 Hardware Solution

In this project, board power is supplied by the USB connection to a computer. This means that the Vin pin will source 5V. The accelerometer uses the 3.3V pin to power itself and connects its ground pin to the ground pin on the board. The two data signals 'X' and 'Y' are connected to two ADC pins on the board. A hardware solution can be seen in Figure 6

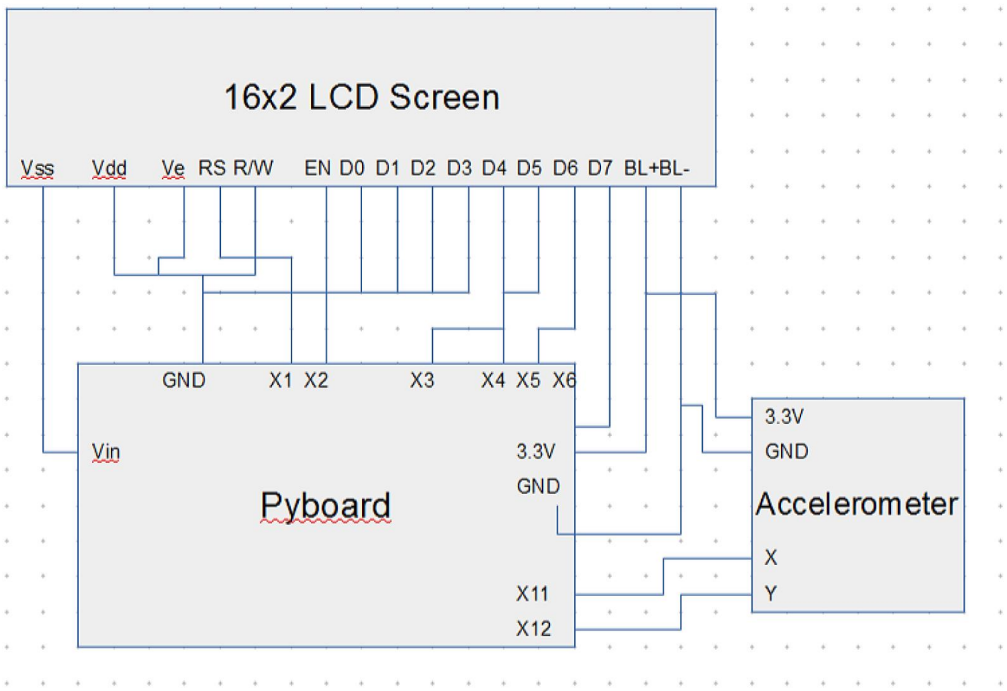

Figure 6: Circuit diagram of Accelerometer and Pyboard connnection

3.3.5.1 Review

Interfacing with analog sensors on the Pyboard is really simple due to the numerous ADC pins. To get the reading of one of the data lines, the read() method of the ADC pin is sufficient to get the information. The percent error of the angle reported by the accelerometer was higher before adding

code to simulate a Hanning filter. The filter helped clean some of the jitter that is inherent to accelerometers by using a weighted average.

3.4 Project 4 : Independent Project

3.4.1 Introduction:

In real-life microcontroller applications interfacing with a single external component is almost unheard of. Normally multiple sensors and output devices must work in tandem to observe and report useful information about the environment the microcontroller is in. Having so many different devices trying to communicate with a single microcontroller at the same time can be a complicated task, even when using standard communication protocols. Time critical tasks could potentially be preempted by device driven interupts and potentially destroy the integrity of any previously collected data. At this point the microcontroller must almost emulate a general purpose operating system, controlling the scheduling of various routines, and maintaining the integrity of memory in places that possibly two or more different routines are trying to write to at nearly the same time. Since microcontrollers do not have operating systems, these tough decisions inevitably fall into the hands of the programmer. This final project must make use of several external sensors and/or output devices.

3.4.2 Project 4a : Whack-a-Mole

3.4.2.1 Approach Taken

For this project, five force sensitive resistors, five L.E.D.s, and a 16x2 LCD screen were used in conjunction with the Pyboard to create the Whack-A-Mole game. The game last sixty seconds and at every ten seconds that have gone by, the moles are alive for a shorter amount of time. The L.E.D.s were placed near each force sensitive resistor. When the LED near a mole lit up, that means the mole was in play. The mole had to be hit with a certain amount of force for the hit to count and if it was hit properly, the LED would turn off signaling that the mole was in longer in play.

3.4.3 Software Solution

The software for this project relies on two functions included in the Pyboard, pyb.millis() and pyb.rng()[3]. pyb.millis() is used to figure out the start time of the game as well as keeping track of how long each mole has left in play. There are functions to put the moles in play and to check if they should still be in play. The function pyb.rng() is used to randomly choose which moles should in play. It is used to generate a random number and based on that number , it turns one of the moles on by turning on the LED. If the mole is already on, then nothing happens. Figure 7 shows the flow of the program of the game. A software solution is shown in Listing 5.

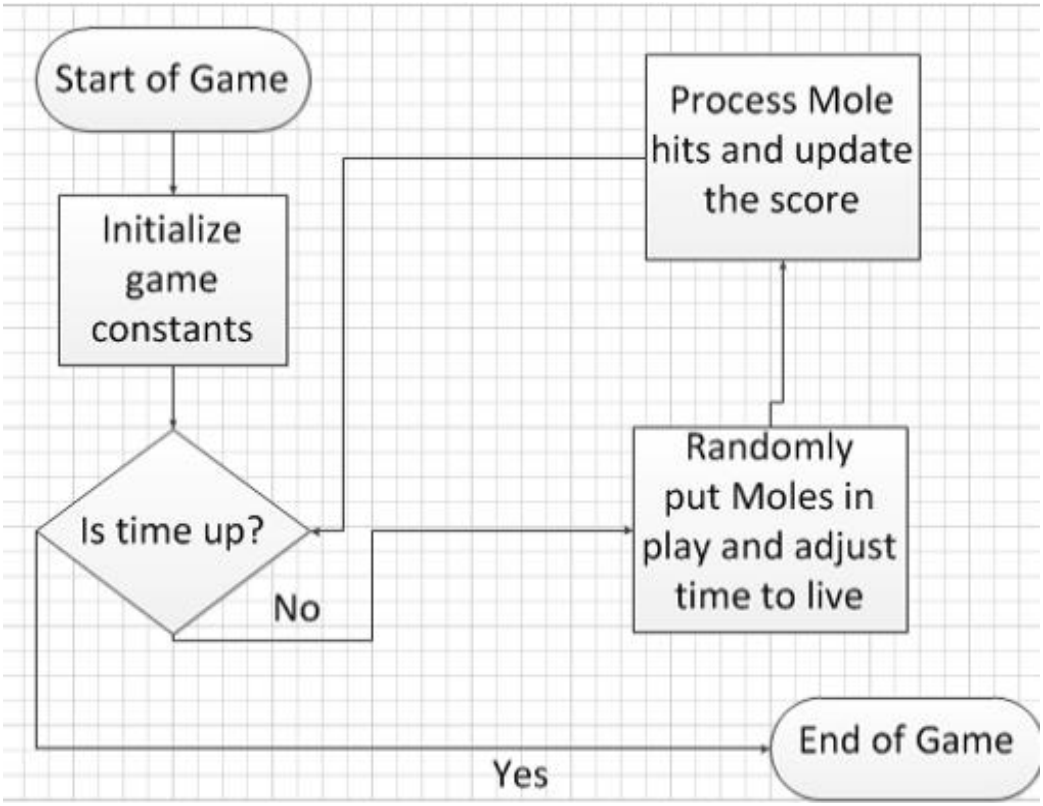

Figure 7: Flowchart of Whack-A-Mole project

3.4.4 Hardware Solution

In this project, board power is supplied by the USB connection to a computer. The LCD Vss needs to be connected to 5V and the BLK+ pin needs to be connected to the 3.3V from the Pyboard. Pins X1-X6 on the Pyboard are used to control the LCD screen. Pins Y1-5 will be used to control the LEDs. Pins X7, X8, X11, X12, and Y11 will be used to read the data from the force sensitive resistors used for the moles. To connect the force sensitive resistors, they will need a 3.3V and ground. There should also be a 27 kOhm resistor for each. Refer to Figure 8 to see how to connect each force sensitive resistor.

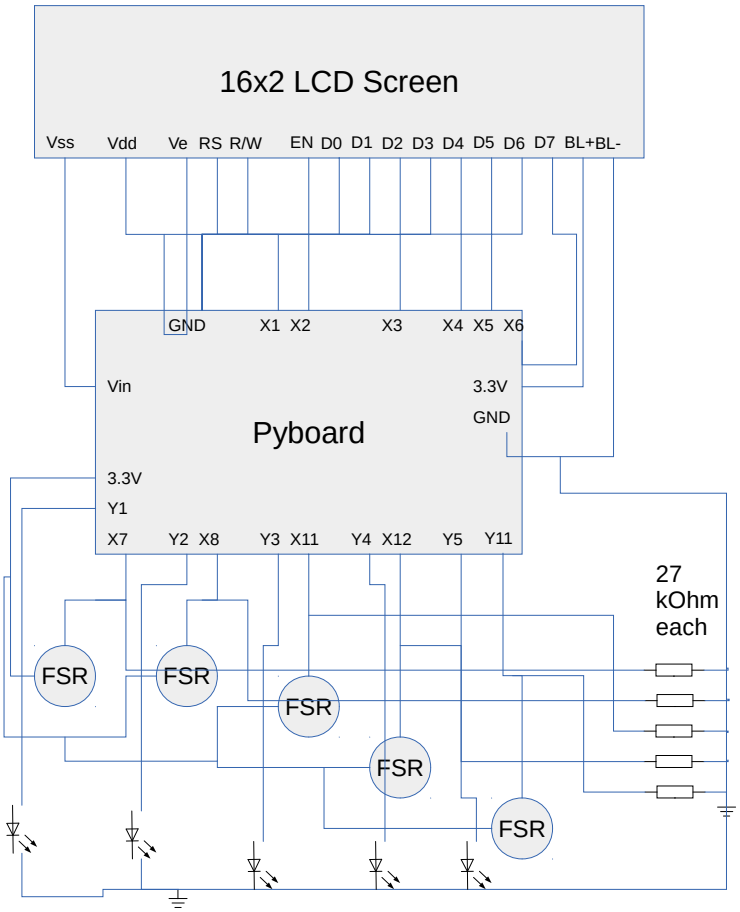

Figure 8: Whack-A-Mole Schematic

3.4.4.1 Review

The Whack-A-Mole project shows some capabilities of the Pyboard to be the brain of simple electronic games. The project highlights the ability to prototype fast using the Pyboard. This is due to the Pyboard's nature of being easily programmed and having many useful classes and methods such as pyb.millis(). The Micropython code is very easy to translate to C and use in other microcontrollers.

3.4.5 Project 4b : Lights Are On

3.4.5.1 Approach Taken

This project utilizes two sensors, an integrated A/D converter, and serial communication with a computer. The Infrared sensor's initialization requires twenty seconds to record the IR signature of a stationary room. Ideally there is no change in the IR values detected in it's field of view. After this calibration phase interfacing with the sensor is very straightforward. It outputs a logical high when a change in the IR signature is detected, otherwise a logical low. The photoresistor acts just like any other resistor, except it's resistance varies with changes in the recorded light level. The brighter the detected light, the lower the effective resistance. The voltage across the photoresistor is simply read in on one of the PyBoard's A/D pins. Communication with the PC is performed over USB, taking advantage of an easy to use MicroPython library called USB_VCP. This library allows for bidirectional communication between the PyBoard and nearly any computer. The Python script running on the PyBoard is very straightforward. It simply periodically polls the two sensors and sends their on or off values as two bytes to the connected computer. These values are read by one Python script that reads these values every two seconds and writes a string describing the output value of the two sensors to a log file. This log file is periodically read by a CGI script which then dynamically generates the HTML for a status message displayed on a web page. The script is rerun every five seconds. Calibration for this project is straightforward but very sensitive to the environment it is located. This is because different rooms can have widely varying light levels throughout the day and it is difficult to find a universal lights on or lights off range for photoresistor readings. Potentially a calibration script could be developed that prompts the user to turn the light off and send a command

to the PyBoard that causes it to store an average low light level and then prompts the user to turn on a light and record an average high light level after the user sends another command.

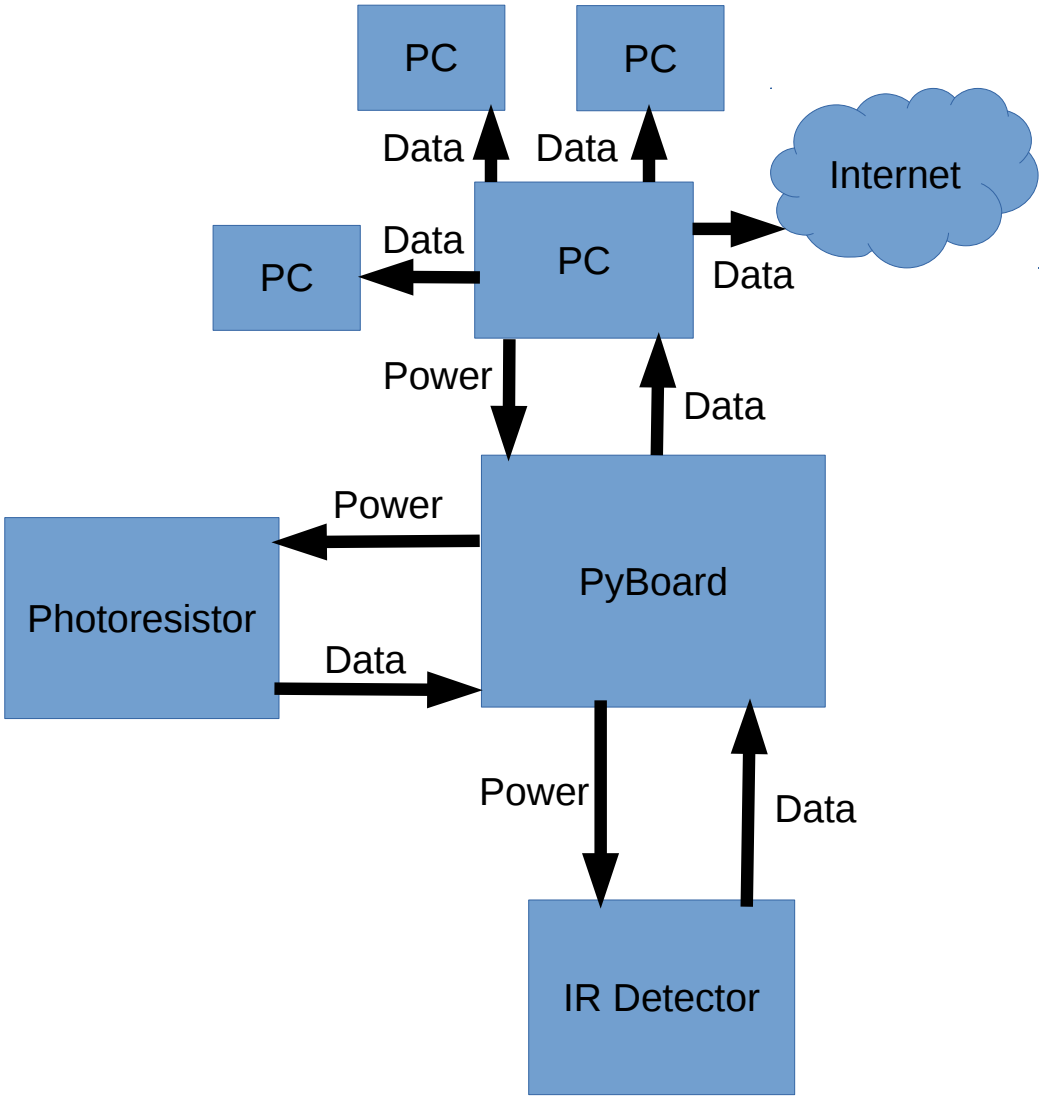

Figure 9: Block Diagram for the Lights are On Project

3.4.5.2 Software Solution

The software produced for this project consists of several simple small

Python scripts and a web-page served by the Python 3 http.server web server. The PyBoard itself polls both the photoresistor and the IR detector periodically. The output from the photoresistor is converted from analog to digital by the PyBoard's integrated A/D converter. The PyBoard script then packs the status of the two sensors in two bytes and sends the data over the USB connection to the computer. The script running on the PyBoard can be found in Listing 6. On the computer itself is another Python script running that reads the bytes sent by the PyBoard and writes the status of the two sensors corresponding to the data in English to a log file. This logging script can be found in Listing 7. The Python 3 http.server is started with CGI support enabled on the same computer, which serves an HTML page with a CGI script embedded. This script parses the log file created by the log writing script and reports the most recent status on the web page. This status is refreshed every five seconds. The markup language for the web page and the CGI script can be found in Listing 9 and Listing 8 respectively.

3.4.5.3 Hardware Solution

For this project a PC powered the PyBoard as shown in Figure 9. This connection was made with a standard micro USB cable. The PyBoard sent the sensor status data to the PC over this same USB connection. The PIR Motion Sensor's VCC pin was connected via jumper cable to the X4 pin on the PyBoard to receive five volts and to reset the sensor's IR signature of the room once a change was detected. Toggling X4 low and then leaving it high for twenty seconds effectively made the IR sensor re-image the room's IR signature. It's GND pin was connected to GND on the PyBoard. Finally the output pin, OUT, was connected to pin X5 on the PyBoard. The IR detector outputs VCC on the OUT pin when a change in the IR signature has been detected, and 0 otherwise. One end of the photoresistor was connected to a VIN pin on the PyBoard while the other went to X6 on the PyBoard which can act as an analog input pin.A 10kΩ pull-down resistor went from pin X6 on the PyBoard to the other GND connection on the PyBoard. The wiring schematic can be seen in Figure 10.

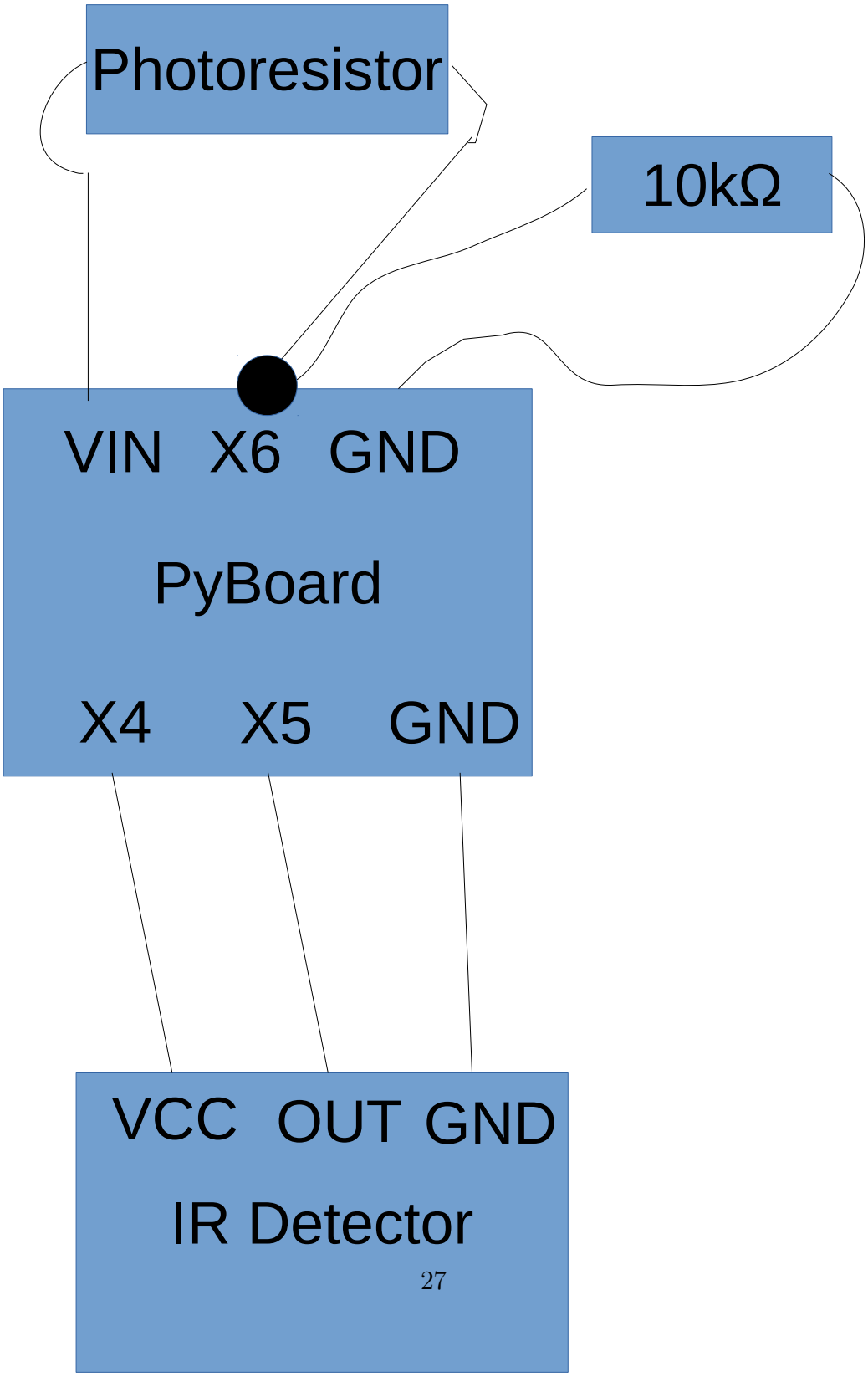

Figure 10: Schematic for the Lights On Project

3.4.5.4 Review

This project shows the versatility and ease of use of the Python language on both a microcontroller and PC. The interface is very straightforward and simple to use. While the microcontroller code could almost as easily be written in C the script that is run by the server must of course be a script and not compiled code. This project highlights the ease with which rapid prototypes can be developed using not just the PyBoard but Python in general.

4 Recommendations

The Pyboard would be a great board for a younger audience to use. With all of the available classes, it is a great way to introduce the world of microcomputing. MicroPython is a derivative of Python 3 so younger users will learn a commonly used programming language and will not have to deal with all of the intricacies of C while also learning about GPIOs, ADCs, DACs, etc.

In addition, the PyBoard is well suited for preliminary proof of concept and prototype work to quickly actualize rough drafts for projects of all but the highest complexity levels. It would likely be a good platform for a project based class focused on the Internet of Things.

The PyBoard is not well suited to replace the TI MSP-430 in the current curriculum for CPE 329. Important concepts in the curriculum are abstracted over by MicroPython. Also, the overall quality of the PyBoard and MicroPython was found to be lacking.

5 Summary

The Pyboard is not a good replacement microcontroller for CPE-329. The degree of control over the speed of the board is lost as the code becomes more complex. The students will never learn about the registers they are using as all of this information is hidden from them. Because of how MicroPython is structured, students will not learn about the bit manipulation needed to set and read registers. Although it is nice to have classes that handle things like D/A conversion, I2C, SPI, and UART; the amount of control over the Pyboard is lost because of them. It is important that the students learn how

to fully harness the capabilities of a microcontroller so that they can design and implement better embedded systems.

6 Solutions

In this section, solutions are provided for each of the projects discussed above

6.1 Project 1 : Hello World

Listing 1: Hello World!

```
from pyb import Pin
from pyb import LED

#variables, etc
hello = "Hello World!"
i = 0
led = LED(1)
led.off()
pyb.delay(100)


def LCD_init ():
        #X8:X5 on Pyboard tied to DB (7:4) of LCD
        #X4 tied to RS on LCD
        #X3 tied to Enable on LCD
        # R/W on LCD is grounded (always write, never read)

        #SET NIBBLE MODE, TWO LINES, CHARACTERS 5X8
        # 8-bit command = 001010
        # 001000 (send first nibble twice)
        p_out6.high()  # set 1's
        p_out8.low() # clear 0's
        p_out7.low()
        p_out5.low()
        p_out4.low()
        p_out3.low()
        pyb.delay(31) # wait ~30 ms for LCD power
        p_out3.high()  # raise enable line
        pyb.udelay(1)  # wait for enable high time
```

```
        p_out3.low()  # drop enable line (data latched)
        pyb.udelay(40)  # wait ~ 39 us for command execution
        p_out3.high()  # raise enable line
        pyb.udelay(1)  # wait for enable high time
        p_out3.low()  # drop enable line (data latched)
        pyb.udelay(40) # wait ~ 39 us for command execution
#       # switch to second nibble of nibble mode instruction
#       # 10--00
        p_out8.high()  # set 1
        p_out7.low() # clear 0's
        p_out4.low()
        p_out3.low()
        p_out3.high()  # raise enable line
        pyb.delay(1)  # wait for enable high time
        p_out3.low()  # drop enable line (data latched)
        pyb.udelay(40) # wait ~ 39 us for command execution

        #TURN DISPLAY ON, CURSOR ON, BLINK ON
        #8-bit command = 00001111
        #nibbles = 0000, 1111
        #000000
        p_out8.low() # clear 0's
        p_out7.low()
        p_out6.low()
        p_out5.low()
        p_out4.low()
        p_out3.low()
        p_out3.high()  # raise enable line
        pyb.udelay(1)  # wait for enable high time
        p_out3.low()  # drop enable line (data latched)
        pyb.udelay(1)  # wait for enable cycle time
        #111100
        p_out8.high() # set 1's
        p_out7.high()
        p_out6.high()
        p_out5.high()
        p_out3.low() # clear 0's
        p_out4.low()
```

```
        p_out3.high()  # raise enable line
        pyb.udelay(1)  # wait for enable high time
        p_out3.low()  # drop enable line (data latched)
        pyb.udelay(40) # wait ~ 39 us for command execution

#       #SET CURSOR TO INCREMENT, DISPLAY DOES NOT SHIFT
#       #8-bit command = 00000110
#       #nibbles = 0000, 0110
#       #000000
        p_out8.low() # clear 0's
        p_out7.low()
        p_out6.low()
        p_out5.low()
        p_out4.low()
        p_out3.low()
        p_out3.high()  # raise enable line
        pyb.udelay(1)  # wait for enable high time
        p_out3.low()  # drop enable line (data latched)
        pyb.udelay(1)  # wait for enable cycle time
        #011000
        p_out7.high() # set 1's
        p_out6.high()
        p_out8.low() # clear 0's
        p_out5.low()
        p_out4.low()
        p_out3.low()
        p_out3.high()  # raise enable line
        pyb.udelay(1)  # wait for enable high time
        p_out3.low()  # drop enable line (data latched)
        pyb.udelay(40) # wait ~ 39 us for command execution

#       #CLEAR DISPLAY
#       #8-bit command = 00000001
#       #nibbles = 0000, 0001
#       #000000
        p_out8.low() # clear 0's
        p_out7.low()
        p_out6.low()
```

```
        p_out5.low()
        p_out4.low()
        p_out3.low()
        p_out3.high()  # raise enable line
        pyb.udelay(1)  # wait for enable high time
        p_out3.low()  # drop enable line (data latched)
        pyb.udelay(1)     # wait for enable cycle time
#       #000100
        p_out5.high() # set 1's
        p_out8.low()    # clear 0's
        p_out7.low()
        p_out6.low()
        p_out4.low()
        p_out3.low()
        p_out3.high()  # raise enable line
        pyb.udelay(1)  # wait for enable high time
        p_out3.low()  # drop enable line (data latched)
        pyb.delay(2)  # wait ~1.53 ms for command execution

        #RETURN CURSOR TO HOME POSITION
        #8-bit command = 0000001-
        #nibbles = 0000, 001-
        #000000
        p_out8.low() # clear 0's
        p_out7.low()
        p_out6.low()
        p_out5.low()
        p_out4.low()
        p_out3.low()
        p_out3.high()  # raise enable line
        pyb.udelay(1) # wait for enable high time
        p_out3.low()  # drop enable line (data latched)
        pyb.udelay(1)  # wait for enable cycle time
        #001-00
        p_out6.high() # set 1's
        p_out8.low() # clear 0's
        p_out7.low()
        p_out5.low()
```

```
        p_out4.low()
        p_out3.low()
        p_out3.high()  # raise enable line
        pyb.udelay(1)  # wait for enable high time
        p_out3.low()   # drop enable line (data latched)
        pyb.delay(2)  # wait ~1.53 ms for command execution
        return
def outchar (c1):
        tempchar = c1
        p_out8.low()
        p_out7.low()
        p_out6.low()
        p_out5.low()
        p_out4.low()
        p_out3.low()
        pyb.udelay(1)

        #c1 = hhhhllll
        #send first nibble to LCD (hhhh----)
        #RS must be high to write data (instead of instruction)
        p_out4.high()   # set enable bit & RS
        p_out3.high()
        pyb.udelay(1)
        p_out8.value(1 & (ord(tempchar) >> 7)) # add first nibble of ch
        p_out7.value(1 & (ord(tempchar) >> 6))
        p_out6.value(1 & (ord(tempchar) >> 5))
        p_out5.value(1 & (ord(tempchar) >> 4))
        pyb.udelay(1)
        p_out3.low()              # clear enable bit & RS
        p_out4.low()
        pyb.udelay(1)

        #send second nibble to LCD (llll----)
        p_out8.low()
        p_out7.low()
        p_out6.low()
        p_out5.low()
        tempchar = c1
```

```
        p_out4.high()  # set enable bit & RS
        p_out3.high()
        pyb.udelay(1)
        p_out8.value(1 & (ord(tempchar) >> 3))
        p_out7.value(1 & (ord(tempchar) >> 2))
        p_out6.value(1 & (ord(tempchar) >> 1))
        p_out5.value(1 & ord(tempchar))
        pyb.udelay(1)
        p_out3.low()  # clear enable bit & RS
        p_out4.low()
        pyb.udelay(1)
        return

#Initialize LCD
p_out8 = Pin('X8', Pin.OUT_PP)
p_out7 = Pin('X7', Pin.OUT_PP)
p_out6 = Pin('X6', Pin.OUT_PP)
p_out5 = Pin('X5', Pin.OUT_PP)
p_out4 = Pin('X4', Pin.OUT_PP)
p_out3 = Pin('X3', Pin.OUT_PP)
p_out8.low()
p_out7.low()
p_out6.low()
p_out5.low()
p_out4.low()
p_out3.low()
LCD_init()

##Iterate through string, send each character to LCD
while (len(hello) > i):
        outchar(hello[i])
        led.toggle()
        pyb.delay(1000)
        i = i + 1
```

6.2 Project 2 : Digital to Analog conversion

Listing 2: Digital to Analog conversion

```
##Function Generator##

import math
from pyb import DAC
from pyb import Pin

# button and switch declarations
dutySW = pyb.Switch()
wavBut = Pin('X2', Pin.IN, Pin.PULL_UP)
freqBut = Pin('X3', Pin.IN, Pin.PULL_UP)

# global variable initialization
aSize = 45
duty = 5
wave = 1
iter = 0

# create a buffer to hold the different duty cycle values
dutyCycle = bytearray(10);
for i in range(10):
dutyCycle[i] = int (4.5 * i)

# create and fill a buffer containing a sine-wave
buf = bytearray(aSize)
for i in range(aSize):
buf[i] = 128 + int(127 *math.sin(2 * math.pi * i / aSize))

# create and fill a buffer containing a saw-wave
tbuf = bytearray(aSize)
for i in range(aSize):
tbuf[i] = int((255/aSize) * i)

# internal DAC defined here
dac = DAC(1)

# function to print out sine wave
def sineWave():
global iter
```

```
global aSize
global buf
dac.write(buf[iter])
iter = iter + 1
if (iter == aSize):
iter = 0

# function to print out the saw wave
def sawWave():
global iter
global aSize
global tbuf
dac.write(tbuf[iter])
iter = iter + 1
if (iter == aSize):
iter = 0

# function to print out square wave
def squareWave():
global iter
global aSize
global half
global dutyCycle
global duty
iter = iter + 1
if iter > dutyCycle[duty]:
dac.write(255)
if (iter > aSize):
iter = 0
else:
dac.write(0)

# call back function that will choose which wave to print
def chooseWave(timer):
global wave
if(wave == 0):
squareWave()
elif(wave == 1):
```

```
sawWave()
else:
sineWave()
# function to poll button 1
def check_wave(val):
global wave
if (val == 0):
pyb.delay(150)
wave = wave + 1
if (wave == 3):
wave = 0

# function to poll button 2
def check_freq(val):
global wave
if (val == 0):
pyb.delay(150)
wave = wave + 1
if (wave == 3):
wave = 0

# function to poll on board usr button
def check_duty(val):
global duty
if (val == True):
pyb.delay(150)
duty = duty + 1
if (duty == 10):
duty = 0

# Creation and initilization
myTimer = pyb.Timer(4)
myTimer.init(freq=18050)
myTimer.callback(chooseWave)

# while loop to run the microcontroller
# and check for button presses
```

```
while(1):
check_wave(wavBut.value())
check_freq(freqBut.value())
check_duty(dutySW())
```

6.3 Project 3a : Capacitive Touch Sensor

Listing 3: Capacitive Touch + LCD

```
import mpr121

lcd = pyb.LCD('Y')
lcd.light(True)

i2c = pyb.I2C(2, pyb.I2C.MASTER) #for touch sensors
touch_sensor = mpr121.MPR121(i2c)
def fill(quadrant):
    if quadrant == 'A':
        x = 0
        xmax = 64
        y = 0
        ymax = 16
        ystart = 0
    if quadrant == 'B':
        x = 64
        xmax = 128
        y = 0
        ymax = 16
        ystart = 0

    if quadrant == 'X':
        x = 0
        xmax = 64
        y = 16
        ymax = 32
        ystart = 16

    if quadrant == 'Y':
```

```
        x = 64
        xmax = 128
        y = 16
        ymax = 32
        ystart = 16

    while x < xmax:
        while y < ymax:
            lcd.pixel(x, y, 1)
            y = y + 1
        y = ystart
        x = x + 1
    lcd.show()

while 1:
    print ("sensor 0: " + str(touch_sensor.touch_status(0)))
    print ("sensor 1: " + str(touch_sensor.touch_status(1)))
    print ("sensor 2: " + str(touch_sensor.touch_status(2)))
    print ("sensor 3: " + str(touch_sensor.touch_status(3)))
    print ('\n')
    if touch_sensor.touch_status(3):
        fill('A')
    if touch_sensor.touch_status(2) and not touch_sensor.touch_status(0
        fill('B')
    if touch_sensor.touch_status(1):
        fill('X')
    if touch_sensor.touch_status(0):
        fill('Y')
    pyb.delay(5000)
    lcd.fill(0)
    lcd.show()
```

6.4 Project 3b : Accelerometer

Listing 4: Accelerometer

```
#main add your code here
#accel code
```

```
import pyb
import math
from pyb import Pin


# defining useful variables
LCD_WIDTH = 16
LCD_CHR = True
LCD_CMD = False
E_PULSE = 50
E_DELAY = 50

Line0 = 0x80
Line1 = 0xC0

clear = "                "

green = pyb.LED(2)
blue = pyb.LED(4)
orange = pyb.LED(3)
red = pyb.LED(1)

# defining the pins I will be using.

RS = Pin('X1',Pin.OUT_PP)
E = Pin('X2',Pin.OUT_PP)
D4 = Pin('X3',Pin.OUT_PP)
D5 = Pin('X4',Pin.OUT_PP)
D6 = Pin('X5',Pin.OUT_PP)
D7 = Pin('X6',Pin.OUT_PP)
x_val = pyb.ADC(pyb.Pin.board.X11)
y_val = pyb.ADC(pyb.Pin.board.X12)


# pin change functions
def changeRS(val):
    if val:
        RS.high()
```

```
    else:
        RS.low()

def changeE(val):
    if val:
        E.high()
    else:
        E.low()

def changeD4(val):
    if val:
        D4.high()
    else:
        D4.low()

def changeD5(val):
    if val:
        D5.high()
    else:
        D5.low()

def changeD6(val):
    if val:
        D6.high()
    else:
        D6.low()

def changeD7(val):
    if val:
        D7.high()
    else:
        D7.low()

# lcd commands
#-------------------------------------------------------------------------
def lcd_byte(bits, mode):

    changeRS(mode)
```

```
#set high bits to 0
changeD4(False)
changeD5(False)
changeD6(False)
changeD7(False)

#set the high bits now
if bits & 0x10 == 0x10:
    changeD4(True)
if bits & 0x20 == 0x20:
    changeD5(True)
if bits & 0x40 == 0x40:
    changeD6(True)
if bits & 0x80 == 0x80:
    changeD7(True)

pyb.udelay(E_DELAY)
changeE(True)
pyb.udelay(E_PULSE)
changeE(False)
pyb.udelay(E_DELAY)


#set low bits to 0
changeD4(False)
changeD5(False)
changeD6(False)
changeD7(False)

#set the low bits now
if bits & 0x01 == 0x01:
    changeD4(True)
if bits & 0x02 == 0x02:
    changeD5(True)
if bits & 0x04 == 0x04:
    changeD6(True)
if bits & 0x08 == 0x08:
```

```
        changeD7(True)

    pyb.udelay(E_DELAY)
    changeE(True)
    pyb.udelay(E_PULSE)
    changeE(False)
    pyb.udelay(E_DELAY)

def init():
    lcd_byte(0x33,LCD_CMD)
    lcd_byte(0x32,LCD_CMD)
    lcd_byte(0x28,LCD_CMD)
    lcd_byte(0x0C,LCD_CMD)
    lcd_byte(0x06,LCD_CMD)
    lcd_byte(0x80,LCD_CMD)
    lcd_clear()


def set_line(line):
    lcd_byte(line, LCD_CMD)

def lcd_clear():
    lcd_byte(0x01, LCD_CMD)


def print_string(message):
    mess_len = len(message)
    for iter in range(mess_len):
        lcd_byte(ord(message[iter]), LCD_CHR)

def clear_line(line):
    global clear
    set_line(line)
    for iter in range(16):
        lcd_byte(ord(clear[iter]), LCD_CHR)

def togLED():
    blue.toggle()
```

```
    orange.toggle()
    red.toggle()
    green.toggle()


iter = 0

#start of main code
blue.on()
orange.off()
red.off()
green.on()

init()

set_line(Line0)
pyb.delay(200)
print_string("Accelerometer")
pyb.delay(500)
set_line(Line1)
pyb.delay(500)



while (1):
    clear_line(Line1)
    set_line(Line1)
    # angle uses the arctan function to figure out the angle
    # using the y value and the x value
    angle = math.atan(y_val.read() / x_val.read())
    print_string(str(angle))
    pyb.delay(100)
```

6.5 Project 4a : Whack-A-Mole

Listing 5: Accelerometer

##Whack–A–Mole##

```
import pyb
from pyb import Pin


# defining useful variables
LCD_WIDTH = 16
LCD_CHR = True
LCD_CMD = False
E_PULSE = 50
E_DELAY = 50

Line0 = 0x80
Line1 = 0xC0

clear = "                "
sixtys = 60000
start = pyb.millis()
level2 = start + 10000
level3 = start + 20000
level4 = start + 30000
level5 = start + 40000
level6 = start + 50000

green = pyb.LED(2)
blue = pyb.LED(4)
orange = pyb.LED(3)
red = pyb.LED(1)

# defining the pins I will be using.

RS = Pin('X1',Pin.OUT_PP)
E = Pin('X2',Pin.OUT_PP)
D4 = Pin('X3',Pin.OUT_PP)
D5 = Pin('X4',Pin.OUT_PP)
D6 = Pin('X5',Pin.OUT_PP)
D7 = Pin('X6',Pin.OUT_PP)
mole1 = pyb.ADC(pyb.Pin.board.X7)
mole2 = pyb.ADC(pyb.Pin.board.X8)
```

```
mole3 = pyb.ADC(pyb.Pin.board.X11)
mole4 = pyb.ADC(pyb.Pin.board.X12)
mole5 = pyb.ADC(pyb.Pin.board.Y11)
mole1_led = Pin('Y1', Pin.OUT_PP)
mole2_led = Pin('Y2', Pin.OUT_PP)
mole3_led = Pin('Y3', Pin.OUT_PP)
mole4_led = Pin('Y4', Pin.OUT_PP)
mole5_led = Pin('Y5', Pin.OUT_PP)

# variable decalarations
random_num = 0
mole_timeup = 1800
score = 0

mole1_start = 0
mole1_hit = 0
mole1_ttl = 0

mole2_start = 0
mole2_hit = 0
mole2_ttl = 0

mole3_start = 0
mole3_hit = 0
mole3_ttl = 0

mole4_start = 0
mole4_hit = 0
mole4_ttl = 0

mole5_start = 0
mole5_hit = 0
mole5_ttl = 0


# function that adjust how long the mole is up
def adjust_molettl(curTime):
        global mole_timeup
```

```
        global level6
        global level5
        global level4
        global level3
        global level2

        if (curTime > level6):
        mole_timeup = 400
        elif (curTime > level5):
        mole_timeup = 500
        elif (curTime > level4):
        mole_timeup = 800
        elif (curTime > level3):
        mole_timeup = 900
        elif (curTime  > level2):
        mole_timeup = 1300

# pick a mole and put it into play
def mole_on(val):
        # if else logic to see which mole to turn on
        if (val >= 226): # mole 1
        if (mole1_ttl <= 0):
                whack_mole1(1, pyb.millis())
        elif (val >= 185): # mole 2
        if (mole2_ttl <= 0):
                #call function to turn mole2 on
                whack_mole2(2, pyb.millis())
        elif (val >= 144): # mole 3
        if (mole3_ttl <= 0):
                #call function to turn mole3 on
                whack_mole3(3, pyb.millis())
        elif (val >= 103): # mole 4
        if (mole4_ttl <= 0):
                #call function to turn mole4 on
                whack_mole4(4, pyb.millis())
        else: # mole 5
        if (mole5_ttl <= 0):
                #call function to turn mole5 on
```

```
                whack_mole5(5, pyb.millis())

# function to turn on and check for hits for mole 1
def whack_mole1(turnon, curTime):
        global mole1_ttl
        global mole1_led
        global mole1_hit
        global mole1_start
        global score


        if ((mole1_ttl <= 0) and (turnon > 0)):
        mole1_led.high()
        mole1_start = curTime
        mole1_ttl = curTime + mole_timeup
        mole1_hit = 0

        else:
        if (curTime > mole1_ttl):
                mole1_ttl = 0
                mole1_led.low()

        elif ((mole1_hit == 0) and (mole1.read() > 3400)):
                score = score + 1
                mole1_led.low()
                mole1_ttl = 0
                mole1_hit = 1

# function to turn on and check for hits for mole 2
def whack_mole2(turnon, curTime):
        global mole2_ttl
        global mole2_led
        global mole2_hit
        global mole2_start

        global score

        if ((mole2_ttl <= 0) and (turnon > 0)):
```

```
        mole2_led.high()
        mole2_start = curTime
        mole2_ttl = curTime + mole_timeup
        mole2_hit = 0

        else:
        if (curTime > mole2_ttl):
                mole2_ttl = 0
                mole2_led.low()

        elif ((mole2_hit == 0) and (mole2.read() > 3400)):
                score = score + 1
                mole2_led.low()
                mole2_ttl = 0
                mole2_hit = 1

# function to turn on and check for hits for mole 3
def whack_mole3(turnon, curTime):
        global mole3_ttl
        global mole3_led
        global mole3_hit
        global mole3_start

        global score

        if ((mole3_ttl <= 0) and (turnon > 0)):
        mole3_led.high()
        mole3_start = curTime
        mole3_ttl = curTime + mole_timeup
        mole3_hit = 0

        else:
        if (curTime > mole3_ttl):
                mole3_ttl = 0
                mole3_led.low()

        elif ((mole3_hit == 0) and (mole3.read() > 3400)):
                score = score + 1
```

```
                mole3_led.low()
                mole3_ttl = 0
                mole3_hit = 1

# function to turn on and check for hits for mole 4
def whack_mole4(turnon, curTime):
        global mole4_ttl
        global mole4_led
        global mole4_hit
        global mole4_start

        global score

        if ((mole4_ttl <= 0) and (turnon > 0)):
        mole4_led.high()
        mole4_start = curTime
        mole4_ttl = curTime + mole_timeup
        mole4_hit = 0

        else:
        if (curTime > mole4_ttl):
                mole4_ttl = 0
                mole4_led.low()

        elif ((mole4_hit == 0) and (mole4.read() > 3400)):
                score = score + 1
                mole4_led.low()
                mole4_ttl = 0
                mole4_hit = 1

# function to turn on and check for hits for mole 5
def whack_mole5(turnon, curTime):
        global mole5_ttl
        global mole5_led
        global mole5_hit
        global mole5_start

        global score
```

```
        if ((mole5_ttl <= 0) and (turnon > 0)):
        mole5_led.high()
        mole5_start = curTime
        mole5_ttl = curTime + mole_timeup
        mole5_hit = 0

        else:
        if (curTime > mole5_ttl):
                mole5_ttl = 0
                mole5_led.low()

        elif ((mole5_hit == 0) and (mole5.read() > 3400)):
                score = score + 1
                mole5_led.low()
                mole5_ttl = 0
                mole5_hit = 1



# pin change functions
def changeRS(val):
        if val:
        RS.high()
        else:
        RS.low()

def changeE(val):
        if val:
        E.high()
        else:
        E.low()

def changeD4(val):
        if val:
        D4.high()
        else:
        D4.low()
```

```
def changeD5(val):
        if val:
        D5.high()
        else:
        D5.low()

def changeD6(val):
        if val:
        D6.high()
        else:
        D6.low()

def changeD7(val):
        if val:
        D7.high()
        else:
        D7.low()

# lcd commands
#-------------------------------------------------------------------------
def lcd_byte(bits, mode):

        changeRS(mode)

        #set high bits to 0
        changeD4(False)
        changeD5(False)
        changeD6(False)
        changeD7(False)

        #set the high bits now
        if bits & 0x10 == 0x10:
        changeD4(True)
        if bits & 0x20 == 0x20:
        changeD5(True)
        if bits & 0x40 == 0x40:
        changeD6(True)
```

```
        if bits & 0x80 == 0x80:
        changeD7(True)

        pyb.udelay(E_DELAY)
        changeE(True)
        pyb.udelay(E_PULSE)
        changeE(False)
        pyb.udelay(E_DELAY)


        #set low bits to 0
        changeD4(False)
        changeD5(False)
        changeD6(False)
        changeD7(False)

        #set the low bits now
        if bits & 0x01 == 0x01:
        changeD4(True)
        if bits & 0x02 == 0x02:
        changeD5(True)
        if bits & 0x04 == 0x04:
        changeD6(True)
        if bits & 0x08 == 0x08:
        changeD7(True)

        pyb.udelay(E_DELAY)
        changeE(True)
        pyb.udelay(E_PULSE)
        changeE(False)
        pyb.udelay(E_DELAY)

def init():
        lcd_byte(0x33,LCD_CMD)
        lcd_byte(0x32,LCD_CMD)
        lcd_byte(0x28,LCD_CMD)
        lcd_byte(0x0C,LCD_CMD)
        lcd_byte(0x06,LCD_CMD)
```

```
        lcd_byte(0x80,LCD_CMD)
        lcd_clear()


def set_line(line):
        lcd_byte(line, LCD_CMD)

def lcd_clear():
        lcd_byte(0x01, LCD_CMD)


def print_string(message):
        mess_len = len(message)
        for iter in range(mess_len):
        lcd_byte(ord(message[iter]), LCD_CHR)

def clear_line(line):
        global clear
        set_line(line)
        for iter in range(16):
        lcd_byte(ord(clear[iter]), LCD_CHR)

def togLED():
        blue.toggle()
        orange.toggle()
        red.toggle()
        green.toggle()


iter = 0

#start of main code
blue.on()
orange.off()
red.off()
green.on()

init()
```

```
set_line(Line0)
pyb.delay(200)
print_string("Game On")
pyb.delay(500)
set_line(Line1)
print_string(str(score))
pyb.delay(500)

elapsed = pyb.millis() - start

while (elapsed < sixtys):
        togLED()
        clear_line(Line1)
        set_line(Line1)
        print_string(str(score))
        #print_string(str(random_num))
        pyb.delay(100)
        elapsed = pyb.millis() - start
        adjust_molettl(pyb.millis())
        random_num = (pyb.rng() / 4194304)
        mole_on(int(random_num))
        whack_mole1(0, pyb.millis())
        whack_mole2(0, pyb.millis())
        whack_mole3(0, pyb.millis())
        whack_mole4(0, pyb.millis())
        whack_mole5(0, pyb.millis())


clear_line(Line0)
set_line(Line0)
print_string("Game Over")
```

6.6 Project 4b : Lights Are On

Listing 6: PyBoard Script for Lights On

```
### Script that runs on PyBoard ###
import pyb
```

```
DARK = 2048 #changes from room to room, may need calibration
TWENTY_SECONDS = 20000
HALF_SECOND = 500
x6 = pyb.Pin('X6', pyb.Pin.PULL_UP)
x5 = pyb.Pin('X5', pyb.Pin.IN)
x4 = pyb.Pin('X4', pyb.Pin.OUT_PP)
adc = pyb.ADC(x6)

#virtual com port stuff
vcp = pyb.USB_VCP()
#so IR sensor can get a snapshot of environment
x4.high()
pyb.delay(TWENTY_SECONDS)

to_send = bytearray(2)
while 1:
    val = adc.read()
    if val <= DARK:
        to_send[0] = 0
    else:
            to_send[0] = 1
            x4.low()
            pyb.delay(HALF_SECOND)
            x4.high()
            pyb.delay(TWENTY_SECONDS)
            if x5.value() == 1:
                to_send[1] = 0
            else:
                    to_send[1] = 1
    pyb.delay(HALF_SECOND)
    vcp.send(data = to_send, timeout = 5000)
### /Script that runs on PyBoard ###
```

Listing 7: Logging Script for Lights On

```
#!/usr/bin/env python3
### Event Logging Script that runs on computer ###
```

```
import time, sys, os, logging, serial

ser = serial.Serial('/dev/ttyACM0') # may not be correct port, system de
logging.basicConfig(format = "%(asctime)s : %(message)s", filename="www/


while 1:
    foo = ser.read(size = 2)
    logging.info('lights = ' + str(foo[0]) + ', movement = ' + str(foo[1
    time.sleep(2)
### /Event Logging Script that runs on computer ###
```

Listing 8: CGI Script for Lights On

```
#!/usr/bin/env python3
### CGI Script that runs on computer ###


import time, sys, os, mmap

os.environ['PYTHONUNBUFFERED'] = '1'
print("Content-Type: text/html\n\n")  # html markup follows
log = open(os.path.dirname(__file__) + '/../status.log')
strlog = mmap.mmap(log.fileno(), 0, access=mmap.ACCESS_READ)
last_ndx = strlog.rfind(b'lights = ')
lights = int(chr(strlog[last_ndx + 9]))
last_ndx = strlog.rfind(b'movement = ')
motion = int(chr(strlog[last_ndx + 11]))

if lights == 1:
    light_status = "on"
else:
    light_status = "off"

if motion == 1:
    motion_status = "someone is"
else:
    motion_status = "no one is"
```

```
htmlFormat = "<p>It appears the lights are " + light_status + " and " +

print(htmlFormat, file=sys.stdout, flush=True)

### /CGI Script that runs on computer ###
```

Listing 9: HTML Page for Lights On

```
### HTML Page that displays status ###
<html>
<head>
<title>
Lights Are On But Nobody's Home
</title>
<meta http-equiv="refresh" content="5">
</head>
<body>
<h1>Welcome!</h1>
<h2>Let's take a look...</h2>
<iframe
id = "status"
src = "cgi-bin/now.cgi"
width = "100%"
frameBorder = "0">
</iframe>
</body>
</html>
### /HTML Page that displays status ###
```

References

[1] class adc. https://micropython.org/doc/module/pyb/ADC. Accessed: 2015-06-07.

[2] class dac. https://micropython.org/doc/module/pyb/DAC. Accessed: 2015-06-07.

[3] class pyb. https://micropython.org/doc/module/pyb/. Accessed: 2015-06-07.

[4] class timer. https://micropython.org/doc/module/pyb/Timer. Accessed: 2015-06-07.

[5] Mpr121 library. http://micropython.org/resources/examples/mpr121.py. Accessed: 2015-06-08.

[6] Msp430g2553 launchpad (msp-exp430g2). http://www.ti.com/ww/en/launchpad/launchpads-msp430-msp-exp430g2.html. Accessed: 2015-06-03.