The MicroPs (Prisoner's) Racetrack

Final Project Report
December 13, 2019
E155

Russell Bingham and Jane Cho Watts

Abstract:

The goal of this project was to create an interactive HotWheels race track with motor actuated parts and an LCD screen to display race information like game state and score. Key technical work for the project included a custom C library to interact with the LCD screen, PWM control implemented on the FPGA, 3D printed mechanical parts, and multi-mode game logic on the FPGA to unify the elements of the system. Two game modes were implemented for the final design, both based on classic game theory decision problems: the Prisoner's Dilemma (hence the name) and the Chicken Game. All elements of the stated problem were solved to specification, and the behavior of the game components was tuned such that the game is playable with variable results in both modes.

Introduction

The motivation for this project was to use HotWheels cars in an interesting and motorized context. The LACMA Metropolis exhibit was a showcase of this to the extreme [1-2], with hundreds of cars constantly circling in a motorized city, and when MATTEL gave out HotWheels cars at the HMC Career Fair, the idea of a motorized HotWheels race track was cemented.

The block diagram of the eventual system is shown in Figure 1. In general, the system is subdivided such that the FPGA handles the system inputs and the action of the game, while the FPGA handles the system output to the user via the LCD screen. As such, the FPGA code includes a custom FPGA PWM routine, a state FSM that acts to handle switch debouncing, and outputs game state information to the ATSAM via GPIO pins. By contrast, the ATSAM takes in game state information from the FPGA, outputs appropriate information to the LCD, and reads the finish switches to keep track of the score.

Mechanically, the race track is functionally segmented into three subsections. The “send station”, with its custom 3D-printed holding slots and start gates, functions as the staging area for the two cars. This section includes two micro-servos to release the cars at the same time, and is thus connected to the FPGA. The second station, the track itself, is primarily made up of stock HotWheels track running the length of the race course. The track includes a motor-actuated ramp at the beginning of its flat run-out, and is thus connected to the FPGA. At the end of the course, the so-called “receive station” contains the limit switches that detect the end of the race when hit by a car. These switches are connected to the FPGA to signal the end-of-game state change, and are also connected to the ATSAM so that it can keep track of the score.

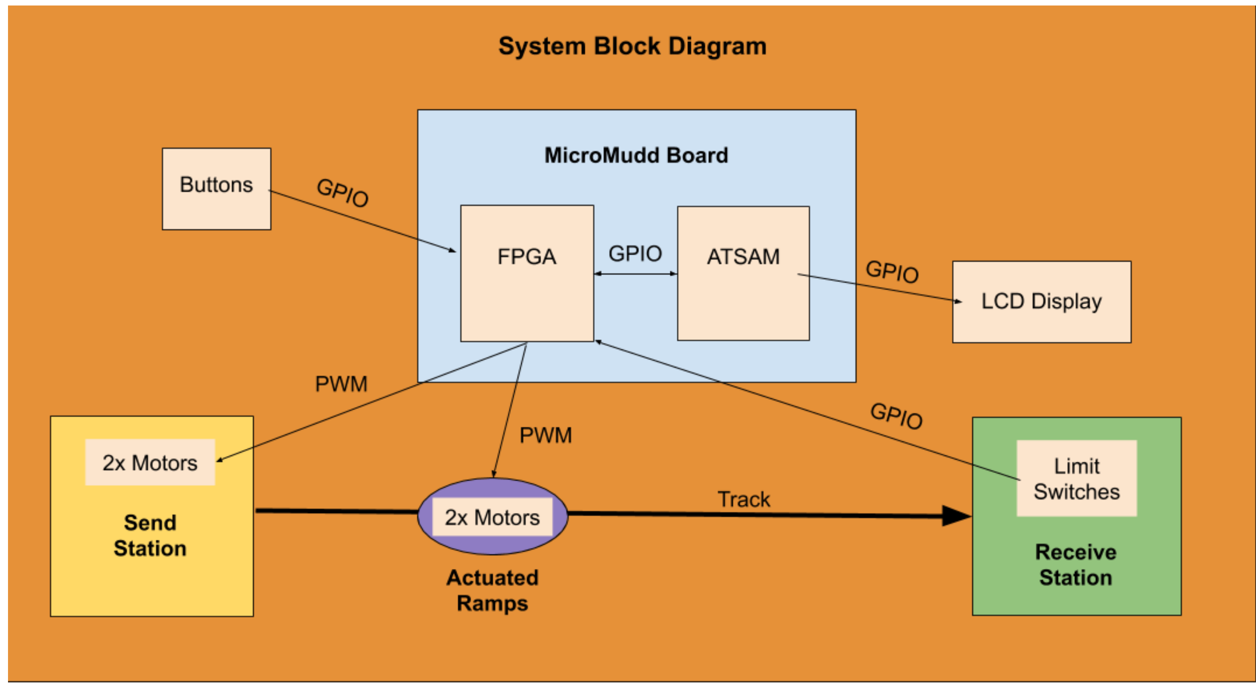

Figure 1: System block diagram

New Hardware

This project includes three primary new hardware components: the HD44780U LCD screen, the four TowerPro SG92R micro-servos, and the two MXRS KW11-3Z-2 limit switches. Their use considerations and implementation details are outlined below.

1. *HD44780U LCD*

The LCD screen (HD44780U) contains 2 rows of display characters, each with 16 characters. The screen requires various setup procedures to run and take in data, as detailed in its datasheet [3]. Once set up, the screen requires the user to sequentially write each character to the screen by moving its internal write address (referred to as the "cursor") to the desired spot, then sending the value of the desired ASCII character via its 8-bit GPIO bus. One of the primary technical accomplishments of this project was implementing a custom ATSAM library to accomplish this functionality. The library was based on the screen's datasheet and on the Arduino LiquidCrystal library the screen is designed to run on. The LCD interface software also includes functionality to take in entire strings and print them sequentially to the screen, making the interface very easy to use.

The screen is wired to the GPIO pins of the ATSAM with a status pin, an enable pin, and an 8-pin data bus, as shown in the system schematic.

2. *SG92 Tower Pro Servo Motors*

The TowerPro SG92R servo motors are standard servos, in a very small size, rated for 2.5 kg-cm of torque load [4]. They take in 5V power supply and a PWM signal to control their rotational position within a 180-degree range. The position of the motor within this range is controlled by the width of the incoming PWM pulse, which can range from 1 ms to 2 ms. Values outside of this range are rejected by the servo's internal input controls. For this project, the PWM signals are generated by a custom FPGA routine, which uses a counter and a slowed clock of 78 kHz to vary the length of the PWM pulse in the output signal across the desired range with 150 different discrete positions. Specifics of Verilog code and block diagram can be found in the FPGA section.

These motors exhibited a consistent issue with oscillating around the desired value. This was addressed with bypass capacitors of 0.1uF and a separate power supply.

3. *MXRS KW11-3Z-2 Limit Switches*

The MXRS KW11-2Z-2 limit switches are a simple model of limit switch which output a high voltage when their contact bar is pressed by a force. As such, they function similarly to a push button, and are wired similarly, with a 2.2 kOhm pull-down resistor between the ground terminal and ground. The data terminal of the switch is merely pulling from the same contact as the ground terminal, and the switch is thus electrically identical to a push button. For this project, the output terminal of both limit switches was wired to both the ATSAM and the FPGA via GPIO pins for both devices.

Schematics

The full schematic of all components on our breadboard can be found in Figure 2 below.

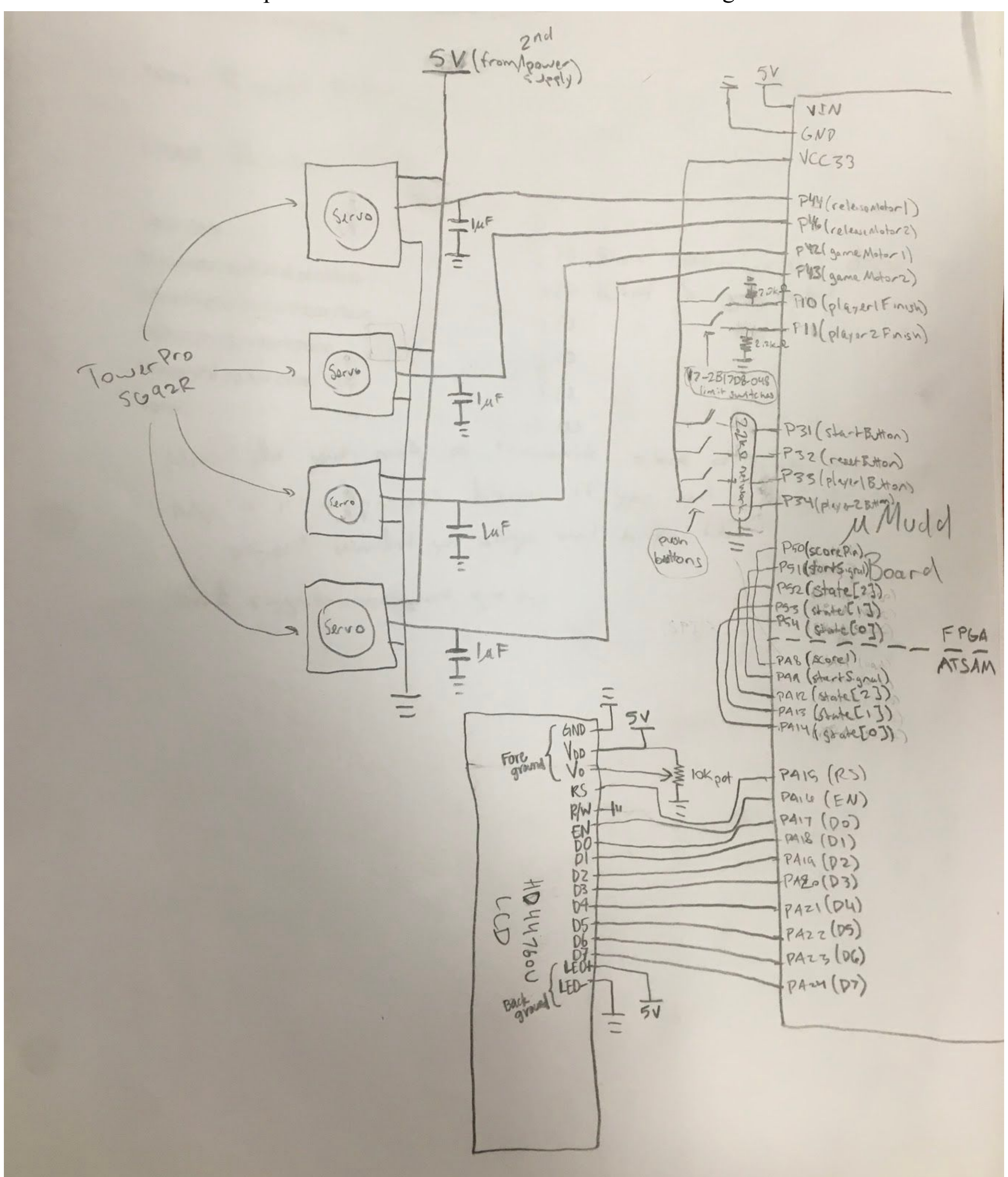

Figure 2: Full schematic

Mechanical Design

The project also required customized mechanical design of the racetrack, including a Send Station, Actuated Ramps, and a Receive Station. Figure 3 shows a block diagram of these components' locations.

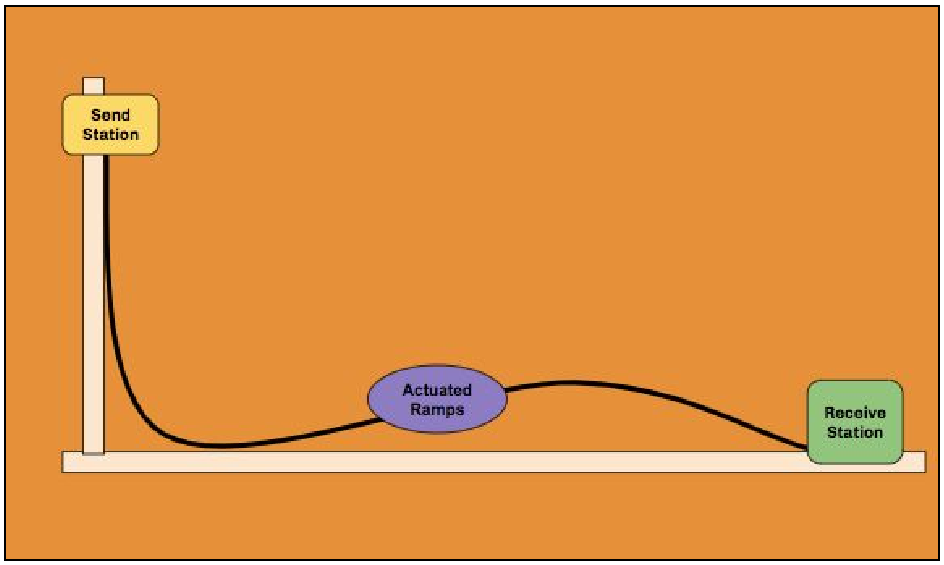

Figure 3: Block diagram of station locations

The mechanical setup, entirely completed, can be seen in Figure 4. A large wood board and a tall 2x4 wood block were assembled using wood screws, then spray painted black to contrast with the bright orange HotWheels tracks and custom components that were affixed to the base with hot glue.

Figure 4: Front/Side view of physical setup

Part Design:
The following parts were custom designed for the project's application, and 3D printed in the HMC Machine Shop. All six of the individual printed parts performed as expected and formed an effective interface between the microMudd board, the new hardware, and the physical system.

1. *Send Station Case & L-shaped Guides*

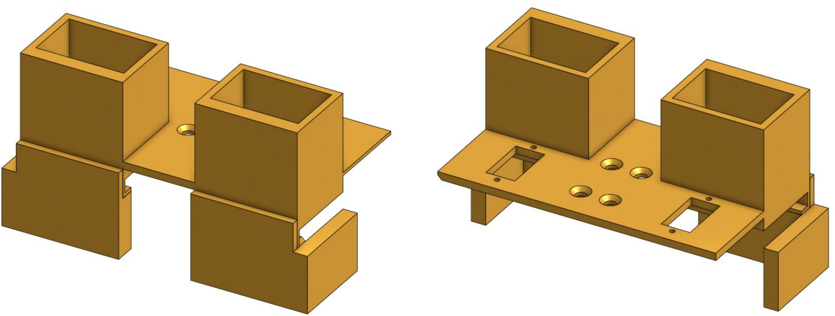

Figure 5: Front/back view of send station sase

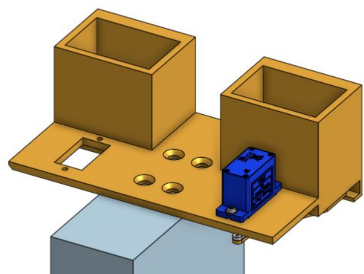

Figure 6: Servo motor and mounting block locations for send station

The send station case has two slots for Player 1 and Player 2's cars to be inserted into. Underneath, there is a lip extruded at the correct height for the servo extension arm to rest on, to ensure that the weight of the car will not dislocate the arm. The case also has cut outs for wood screws into the wooden column and 2 servo motors. The L-shaped guides were printed afterwards to increase stability of the cars during release.

2. *Send Station Servo Fan Extensions*

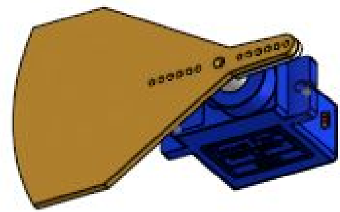

Figure 7: Servo arm fan extension

The extension for the servo arm serves to more effectively block the cars from going down the track as they rest in the send station. This shape covers the full area of the drop zone, and when the motors turn 90 degrees at the start of the race, the cars fall consistently through the chute. The fan's curved shape allows for its rotation across the flat lip surface of the Send Stati

3. *Ramp Servo Extension & Pivot Block*

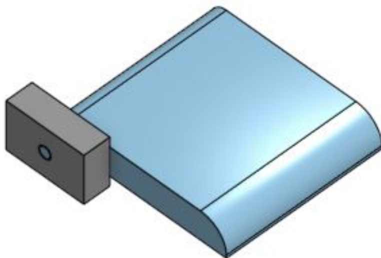

Figure 7: Track ramp and structural block

The track ramps are the width of the HotWheels track and have a pin extension on the side facing away from the servo that is secured in the hole of a structural support block. The ramps have a rounded edge to save material, distinguish one side from the other, and provide a straight launch path on the uncurved upper side. This system was very structurally sound, and remained in place and actuatable even after the ramp servos were internally damaged by car impacts.

FPGA Design

As noted above, the FPGA handles the game logic, the user inputs via push buttons and limit switches, and the PWM control for the servo motors in the race course. The design of each of these subsections is described below.

A. Game Logic

The game logic for the race is described by a seven-state FSM, implemented on the FPGA. The full state transition diagram can be seen in Figure 8, and the motor state output for each game mode can be seen in Tables 1 and 2 for Prisoner's and Chicken mode, respectively.

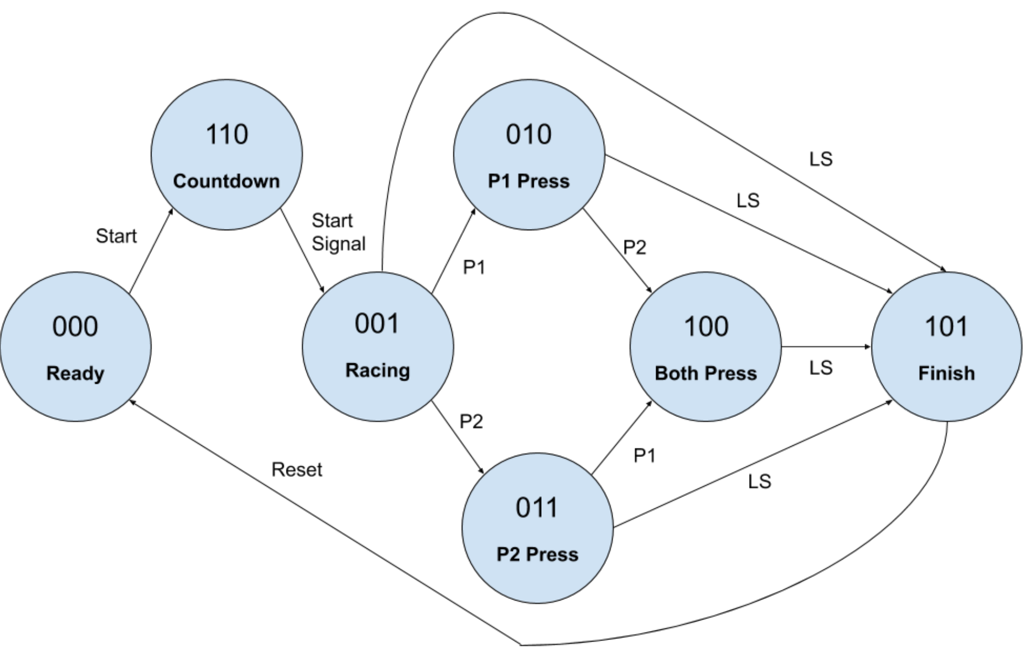

Figure 8: State transition diagram of game logic FSM

In prose, the logic of the game is as follows:

1. Reset game with reset button
First, to reset the state of the game logic and track state, the players will press the reset button on the breadboard. This will set the release station gates to closed and the track ramps to down. Once the two cars are placed in their respective slots in the release station, the game is ready to begin.

2. Start
Once the cars are placed in their slots and the players are ready, one player can press the start button, which will begin a 3 second countdown on the LCD display. When the count reaches zero, the motors in the release station will toggle, releasing the two cars to go down the tracks.

3. Control player buttons

As the cars are moving down the vertical drop below the release station, the players must each make a snap decision about whether to press their control button, and if so, when. The different effects of a button press by either or both players is shown in the tables below, but effectively, in Prisoner's mode, pressing the button makes the other player's ramp activate, and vice versa for Chicken mode.

4. Ramp sections

The ramp sections will be at the bottom of the initial vertical drop on the track. In testing, we have found that this gives the players less than a second to make their button decision, and fractions of a second to time their press correctly. We believe that this time interval will be so small that there will be a significant possibility that a player will press their button too slowly and the cars will pass the ramps before they can react. This will generate an element of skill and suspense to the game, making it fun to play.

5. Finish

At the end of the tracks, there will be two limit switches to sense when the cars have reached the end of the race. As soon as either switch is tripped, the FPGA changes state and records the winner. This increments the score count in the ATSAM and the LCD changes to display the running score between the players. The game flow loops back to reset, and the players should press the reset button and get ready for a new race.

	Player 1 - No Press	**Player 1 - Press**
Player 2 - No Press	Both ramps down	Player 2 ramp at 'STOP'
Player 2 - Press	Player 1 ramp at 'STOP'	Both ramps at 15˚ Ready to launch

Table 1: Prisoner's Dilemma Game Mode Ramp Logic

	Player 1 - No Press	**Player 1 - Press**
Player 2 - No Press	Both ramps at 'STOP'	Player 1 ramp at 15˚
Player 2 - Press	Player 2 ramp at 1˚	Both ramps down

Table 2: Chicken Game Mode Ramp Logic

B. PWM Module

The FPGA also housed the PWM module to drive the servo motors to desired angles. To run the PWM's counter, generate_slowclk( ) module from Lab 3, which uses a multi-bit counter and observes the most significant bit to slow down the FPGA's 40 MHz clock, was reused. Using the following equation:

$$f_{out} = f_{clk} * p / N$$

and counter value p = 1 and number of counter bits N = 9, the slowclk module generated a clock of 78 kHz, or a period of 0.013 ms.

Knowing the period, or amount of time for a single count, the number of slowclk counts for each desired PWM duty cycle was calculated. For example:

$$Duty\ cycle\ of\ 1\ ms * \frac{1\ slowclk\ count}{0.013\ ms} = 77\ counts$$

In this case, a PWM duty cycle of 1 ms (the lower limit for the servo) would be equivalent to 77 slowclk counts. With the calculated count values, the PWM compares the duty cycle count and current counter value, then outputs 1 if the counter is less than the duty cycle count and outputs 0 if the counter exceeds the duty cycle count. Finally, the PWM resets its counter once the PWM period (20 ms, or 1563 counts) is reached to produce a periodic signal. The block diagram for this logic can be found in Appendix C.

Microcontroller

At a high level, the ATSAM reads the current FSM state from 3 GPIO pins and then reacts to that state by printing the appropriate text to the LCD screen. Its only other output function is communicating to the FPGA when the 3 second-long countdown state has finished, at which point it sets a separate GPIO pin to high. Thus, the majority of the code in the primary C file is devoted to reading the game state and using the separate LCD screen interface, developed for the ATSAM from scratch with reference to the Arduino LiquidCrystal library's functionality.

The ATSAM routines to interface with the LCD are organized in a header file (lcdControl.h - see Appendix B). To operate the LCD with this code, one must call the lcdBegin() function, which writes the appropriate settings into the LCD's settings memory. After initialization, the user can use the printToLCD() function to print two strings to the two rows of the display. This function loops through the passed-in length of each string, and moves the cursor and writes the character at each spot. The moveCursor() function sets the current DDRAM write address (cursor) using the appropriate signal flags and digital bus values, and the write() function sets the digital bus lines to the correct values then pulses the enable pin for 1 ms to cause the screen to accept the character and write it to the cursor's position on the screen. The lcdControl.h interface also includes a range of other utility functions to operate the display that are C implementations of each function available in the LiquidCrystal library.

Results

In short, this project was successful and completed all of its stated objectives. The game state machine was accurate and reliable, and the FPGA consistently sent the correct PWM signals to the motors. The ATSAM reacted to the game state information as desired and printed informative text to the attached LCD screen without bugs. Beyond the stated objectives of the project, a second game mode was also implemented that allows players to test out a different game theory problem in the HotWheels race track context. The mechanical parts designed to interface with the servo motors and race cars behaved as intended, and the 3D printed structures were solid and consistent.

The primary shortcoming of the project was that the forces involved in actuating the various moving parts eventually did internal damage to the TowerPro SG92R servos, resulting in power problems, skipping motors, and by the end of Demo Day, a couple completely dead motors. While this was not entirely unexpected given the speed of the cars racing down the track, the magnitude and frequency of these motor malfunctions had a detrimental effect on the entertainment value of the final product. Future work on the system would focus partially on integrating more robust servos (see the TowerPro SG5010) so that the system holds up under extended use.

Other future work would include adding an option for the racers to tie in the event of both cars jumping off or past the race track, optimizing the track shape for consistency, and collecting more race data to quantify the effectiveness of different race configurations.

References

[1] C. Burden, Metropolis II (the movie), https://www.youtube.com/watch?v=llacDdn5yIE
[2] Metropolis II - Hot Wheels Kinetic Sculpture - at LACMA, https://www.youtube.com/watch?v=TA8fj-MJe5s&t=40s
[3] Hitachi, HD44780U (LCD-II) Datasheet, https://cdn-shop.adafruit.com/datasheets/HD44780.pdf
[4] TowerPro, SG92R, http://www.towerpro.com.tw/product/sg92r-7/
[5] MXRS KW-11-3Z-2 Limit Switches: https://www.amazon.com/MXRS-Hinge-Momentary-Button-Switch/dp/B07MW2RPJY/
[6] Arduino LiquidCrystal Library: https://www.arduino.cc/en/Reference/LiquidCrystal

Parts List

Item	Qty	Source	**Vendor Part #**	Price
LCD Display	1	Adafruit (SparkFun)	1447 (HD44780)	$10.95
Servo Motor	4	Adafruit (TowerPro)	4326 (SG92R)	$23.80 ($5.95 each)
Hot Wheels Car & Mega Track Pack	1 (40 ft)	Amazon (MATTEL)	B0721CGJMT	$19.99
Hot wheel Cars	1 (Pack of 5)	Amazon (MATTEL)	B002ZCZ0F6	$4.99
Limit switches	2	Amazon (MXRS)	MXRS KW11-3Z-2	$6.49 ($0.54 each, pack of 12)
Total				$63.39

Appendix A: Verilog Code

```
/*
Jane Cho Watts
Russell Bingham

Email: jwatts@hmc.edu, rbingham@g.hmc.edu
Date Created: 11/20/2019
Purpose: Run Game Logic, send/receive signals to/from connected components, including
motors, buttons, limit switches, and microcontroller (controlling the LCD screen)
*/

//definitions for angle case # of each motor state
`define down 3'd0
`define up 3'd1
`define jump 3'd2
`define leftClosed 3'd3
`define leftOpen 3'd4
`define rightClosed 3'd5
`define rightOpen 3'd6

//////////////////////////////////////////////////////////////////
module FinalProject_fpga(input clk, reset, // 40 Mhz clock
                                        input logic startButton, resetButton,
player1Button, player2Button, player1Finish, player2Finish, modeSwitch, startSignal,
// input button signals & input signal from MCU
                                        output logic releaseMotor1, releaseMotor2,
gameMotor1, gameMotor2, //output PWM motor signals
                                        output logic statePin2, statePin1, statePin0,
scorePin // GPIO comms pins
                                        );

        logic slowclk;
        logic [2:0] state, nextState; // state holders

        //generates a slower clock to run the entire system
        generate_slowclk slowCLK(clk, reset, slowclk);

        // state register
        always_ff @(posedge clk, posedge reset)
                if (reset) state <= 0;
                else            state <= nextState;

        // nextState logic
        // see FSM diagram
        always_comb
                case (state)
                        3'b000:         if (startButton)    nextState = 3'b110; // start
                                        else                nextState = 3'b000;
                        3'b110:  if (startSignal)           nextState = 3'b001; // countdown
```

```
                    else                nextState = 3'b110;
            3'b001:      if (player1Button)  nextState = 3'b010; // racing
                         else if (player2Button) nextState = 3'b011;
                         else if (player1Finish) nextState = 3'b101;
                         else if (player2Finish) nextState = 3'b101;
                         else                    nextState = 3'b001;
            3'b010:      if (player2Button)    nextState = 3'b100; // p1 press
                         else if (player1Finish) nextState = 3'b101;
                         else if (player2Finish) nextState = 3'b101;
                         else                    nextState = 3'b010;
            3'b011:      if (player1Button)   nextState = 3'b100; // p2 press
                         else if (player1Finish) nextState = 3'b101;
                         else if (player2Finish) nextState = 3'b101;
                         else                   nextState = 3'b011;
            3'b100:      if (player1Finish)  nextState = 3'b101; // both press
                         else if (player2Finish) nextState = 3'b101;
                         else                nextState = 3'b100;
            3'b101:      if (resetButton)    nextState = 3'b000; // race over
                         else                nextState = 3'b101;
            default:                         nextState = 3'b000;
        endcase

    // set these correctly for the different race states and modes
    logic [2:0] motor1State, motor2State, startGateStateLeft, startGateStateRight;

    // output logic
    // see FSM diagram

    // left game motor logic
    always_comb
        case (state)
            3'b000:      motor1State <= modeSwitch ? `up : `down; //load
            3'b110:      motor1State <= modeSwitch ? `up : `down; //countdown
            3'b001:      motor1State <= modeSwitch ? `up : `down; //race
            3'b010:      motor1State <= modeSwitch ? `jump : `down; //p1 press
            3'b011:      motor1State <= modeSwitch ? `down : `up;//p2 press
            3'b100:      motor1State <= modeSwitch ? `down : `jump;//both
            3'b101:      motor1State <= `down; //finish
            default: motor1State <= `down;
        endcase

    // right game motor logic
    always_comb
        case (state)
            3'b000:  motor2State <= modeSwitch ? `up : `down; //load
            3'b110:      motor2State <= modeSwitch ? `up : `down; //countdown
            3'b001:      motor2State <= modeSwitch ? `up : `down;  //race
            3'b010:      motor2State <= modeSwitch ? `down : `up;//p1 press
            3'b011:      motor2State <= modeSwitch ? `jump : `down;//p2 press
            3'b100:      motor2State <= modeSwitch ? `down : `jump;//both
```

```
                3'b101:       motor2State <= `down; //finish
                default: motor2State <= `down;
            endcase

    // left release motor logic
    always_comb
            case (state)
                3'b000:       startGateStateLeft <= `leftClosed;
                3'b110:       startGateStateLeft <= `leftClosed; // countdown
                3'b001:       startGateStateLeft <= `leftOpen; // start
                3'b010:       startGateStateLeft <= `leftOpen;
                3'b011:       startGateStateLeft <= `leftOpen;
                3'b100:       startGateStateLeft <= `leftOpen;
                3'b101:       startGateStateLeft <= `leftClosed;
                default: startGateStateLeft <= `leftClosed;
            endcase

            // right release motor logic
            always_comb
            case (state)
                3'b000:       startGateStateRight <= `rightClosed;
                3'b110:       startGateStateRight <= `rightClosed; // countdown
                3'b001:       startGateStateRight <= `rightOpen; // start
                3'b010:       startGateStateRight <= `rightOpen;
                3'b011:       startGateStateRight <= `rightOpen;
                3'b100:       startGateStateRight <= `rightOpen;
                3'b101:       startGateStateRight <= `rightClosed;
                default: startGateStateRight <= `rightClosed;
            endcase

    // run motors based on output logic
    PWMsignal startMotor1(slowclk, reset, startGateStateLeft, releaseMotor1);
    PWMsignal startMotor2(slowclk, reset, startGateStateRight, releaseMotor2);
    PWMsignal trackMotor1(slowclk, reset, motor1State, gameMotor1);
    PWMsignal trackMotor2(slowclk, reset, motor2State, gameMotor2);

    // give state and score info to ATSAM
    assign statePin2 = state[2];
    assign statePin1 = state[1];
    assign statePin0 = state[0];

endmodule

//////////////////////////////////////////////////////////////////////
module generate_slowclk(input clk, reset, output slowclk);
/*
Jane Cho Watts

Email: jwatts@hmc.edu
Date Created: 9/25/19
```

```
Purpose: This module takes in the 40 MHz clk from FPGA Pin 88, and outputs a slower
clk 'slowclk'
                    This is to match the slower clock that the Dual Digit Display
needs to time-multiplex, and avoid asynchronous design.
*/
       logic [8:0] N; //counter bit to generate slower frequency

       always_ff@(posedge clk, posedge reset) //counter
              if(reset) N <= 0;
                     else  N <= N + 1;
       //we look at the most significant bit of N, which will be switching at a lower
frequency than clk
       assign slowclk=N[8];
       //fout = fclk p / N = (40MHz)*1/2^9 = 78 kHz // new slowclk_period = .0128 ms
endmodule

//////////////////////////////////////////////////////////////////////
module PWMsignal(input slowclk, reset,
                                   input [2:0] angle,
                                   output PWM_out);

       logic resetPWM;
       logic [10:0] PWMperiod_cnts, //# of counts in PWM waveform period
                                   dutycycle_cnts, //# of counts in duty cycle
                                   PWMcnts; //# of current counts

       assign PWMperiod_cnts = 1563; //20 ms (1563 slowclk_periods)

       // allowable range roughly 75 to 150
       always_comb
              case(angle)
                     3'd0:  dutycycle_cnts = 134; // down
                     3'd1:  dutycycle_cnts = 100; // up
                     3'd2:  dutycycle_cnts = 129; // jump
                     3'd3:  dutycycle_cnts = 140; // leftClosed
                     3'd4:  dutycycle_cnts = 70;  // leftOpen
                     3'd5:  dutycycle_cnts = 65;  // rightClosed
                     3'd6:  dutycycle_cnts = 140; // rightOpen
                     default: dutycycle_cnts = 86;
              endcase

       //counter for PWM clock signal
       always_ff @(posedge slowclk)
              if (resetPWM) PWMcnts <= 0;
              else PWMcnts <= PWMcnts + 1;

       assign resetPWM = (PWMcnts > PWMperiod_cnts) | (reset); //new cycle
       //set as 1 during duty cycle, set as 0 otherwise
       assign PWM_out = (PWMcnts < dutycycle_cnts);
       endmodule
```

Appendix B: C Code

```
// main_code.c
// Russell Bingham and Jane Watts
// rbingham@g.hmc.edu, jwatts@g.hmc.edu
// 12/12/19
//
// Run Prisoner's Racetrack system
// displays data to screen, keeps track of countdown and score

////////////////////////////////////////////////
// #includes
////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include "SAM4S4B.h"
#include "lcdControl.h"

////////////////////////////////////////////////
// Constants
////////////////////////////////////////////////

volatile unsigned long* PMC_WPMR  = (unsigned long*) 0x400E04E4; // Pointer to the
write protect mode register
volatile unsigned long* PMC_PCER0 = (unsigned long*) 0x400E0410; // Pointer to the
PMC_PCER0 register
volatile unsigned long* WDT_MR    = (unsigned long*) 0x400E1454; // Pointer to the
WDT_MR register

////////////////////////////////////////////////
// Messages
// The following are the LCD display message routines for each game state
////////////////////////////////////////////////
void state000(void) { //Loaded
    //Screen 1
    printToLCD("E155 Prisoner's ", "   Racetrack    ", 16, 16);

    tcDelayMillis(200);

    //Screen 2
    //Message will change depending on Mode
    if(pioDigitalRead(PIO_PB3)==0){ //read mode
        printToLCD("Mode: Prisoner's", "START to begin! ", 16, 16);
    } else {
        printToLCD("Mode: Chicken   ", "START to begin! ", 16, 16);
    }
}

void state110(void) { //Countdown
```

```
    printToLCD("  Ready to Race!  ", "       3        ", 16, 16);
    tcDelayMillis(150);
    printToLCD("  Ready to Race!  ", "       2        ", 16, 16);
    tcDelayMillis(150);
    printToLCD("  Ready to Race!  ", "       1        ", 16, 16);
    tcDelayMillis(150);

    //send StartSignal to the FPGA to indicate motors to release
    pioDigitalWrite(PIO_PA9, 1); //startSignal = 1
}

void state001(void) { //Start Racing
    if(pioDigitalRead(PIO_PB3)==0){
        printToLCD("VroOoo0o0ooOoom!", "Mode: Prisoner's", 16, 16);
    } else {
        printToLCD("VroOoo0o0ooOoom!", "Mode: Chicken   ", 16, 16);
    }

    pioDigitalWrite(PIO_PA9, 0); //startSignal = 0, reset for next game
}

void state010(void) { //P1 only Press
    printToLCD("Player 1 Button ", "Press Recognized", 16, 16);
}

void state011(void) { //P2 only Press
    printToLCD("Player 2 Button ", "Press Recognized", 16, 16);
}

void state100(void) { //Both Press
    printToLCD("  Both Players  ", "Press Recognized", 16, 16);
}

int state101(char* temp) { //Done!

    if ((strcmp(temp, "1") + 1) == 1) {
        printToLCD("Race Completed! ", "Winner: Player 1", 16, 16);
        tcDelayMillis(300);
        return 0; // return 0 if p1 won
    }
    else {
        printToLCD("Race Completed! ", "Winner: Player 2", 16, 16);
        tcDelayMillis(300);
        return 1; // return 1 if p2 won
    }

}

////////////////////////////////////////////////
// Main
```

```
///////////////////////////////////////////////
int main(void) {
    *PMC_WPMR = 0x504D4300; // Writes a password to the write protect mode register
    *WDT_MR    |= 1 << 15;  // Set the WDDIS bit to 1 to disable the watchdog timer

    //GPIO initializiation
    pioInit();
    pioPinMode(PIO_PA12, PIO_INPUT); //state[0]
    pioPinMode(PIO_PA13, PIO_INPUT); //state[1]
    pioPinMode(PIO_PA14, PIO_INPUT); //state[2]
    pioPinMode(PIO_PA25, PIO_INPUT); // score signal
    pioPinMode(PIO_PA9, PIO_OUTPUT); //start signal
    pioDigitalWrite(PIO_PA9, 0); //initialize start signal as 0
    pioPinMode(PIO_PB3, PIO_INPUT); //mode swtich

    //initialize LCD display
 lcdBegin(16, 2);

    // intialize variables for print loops
    char state[3] = "000";
    char lastState[3] = "000";
    char p1Score[11] = "Player 1: _";
    char p2Score[11] = "Player 2: _";
    char temp[1];
    int mode, lastMode;
    int score1 = 0;
    int score2 = 0;
    int scoreTemp = 0;
    int doneFlag = 0;

    while (1) {
        strcpy(lastState, state); // keep track of most recent state
        lastMode = mode; // most recent mode
        // read state vals through GPIO
        state[0] = pioDigitalRead(PIO_PA12) + '0';
        state[1] = pioDigitalRead(PIO_PA13) + '0';
        state[2] = pioDigitalRead(PIO_PA14) + '0';
        mode = pioDigitalRead(PIO_PB3); // read mode from SW3
        temp[0] = pioDigitalRead(PIO_PA25) + '0';

        // if state is different or mode has changed
        // call print function for each state (see above)
        if ((strcmp(lastState, state) + 1) != 1 || lastMode != mode) {

            if ((strcmp(state, "000") + 1) == 1) {
                state000();
                doneFlag = 0; // new game, allow another score
            }
            else if ((strcmp(state, "110") + 1) == 1) {
             state110();
```

```
            }
            else if ((strcmp(state, "001") + 1) == 1) {
                pioDigitalWrite(PIO_PA9, 0); // countdown over //CHANGEME
                state001();
            }
            else if ((strcmp(state, "010") + 1) == 1){
                state010();
            }
            else if ((strcmp(state, "011") + 1) == 1){
                state011();
                }
            else if ((strcmp(state, "100") + 1) == 1){
                state100();
            }
            else if ((strcmp(state, "101") + 1) == 1){
                if (doneFlag != 1) {
                    doneFlag = 1; // allow only one score per race
                    // see state101() for scoreTemp return vals
                    scoreTemp = state101(temp);
                    // increment scores
                    if (scoreTemp == 0)      {score1 = score1 + 1;}
                    else                                         {score2 = score2 +
1;}
                    // play game to 10 points, then reset
                    if (score1 > 4) {
                        printToLCD("Player 1 Wins!!!", "Woo0oO0OoOooO00o", 16, 16);
                        score1 = 0;
                        score2 = 0;
                    }
                    else if (score2 > 4) {
                        printToLCD("Player 2 Wins!!!", "Woo0oO0OoOooO00o", 16, 16);
                        score1 = 0;
                        score2 = 0;
                    }
                    else {
                        p1Score[10] = (score1 + '0');
                        p2Score[10] = (score2 + '0');
                        printToLCD(p1Score, p2Score, 11, 11);
                    }
                    tcDelayMillis(300);
                }
                printToLCD("  Press RESET  ", "  to start over  ", 15, 16);
            }
            else {printToLCD("Unexpected State", "  ", 16, 2);}
        }
    }

    return 0;
}
```

```
/*

lcdControl.h
Author: Russell Bingham
Email: rbingham@g.hmc.edu
E155 MicroPs Final Project

Header file to provide functions to control the HD445780U LCD
Based on the Arduino LiquidCrystal library with Prof Harris' permission

*/

#include "SAM4S4B.h"

// pinouts
#define RS 15
#define EN 16
#define D0 17
#define D1 18
#define D2 19
#define D3 20
#define D4 21
#define D5 22
#define D6 23
#define D7 24

// commands
#define LCD_CLEARDISPLAY 0x01
#define LCD_RETURNHOME 0x02
#define LCD_ENTRYMODESET 0x04
#define LCD_DISPLAYCONTROL 0x08
#define LCD_CURSORSHIFT 0x10
#define LCD_FUNCTIONSET 0x20
#define LCD_SETCGRAMADDR 0x40
#define LCD_SETDDRAMADDR 0x80

// flags for display entry mode
#define LCD_ENTRYRIGHT 0x00
#define LCD_ENTRYLEFT 0x02
#define LCD_ENTRYSHIFTINCREMENT 0x01
#define LCD_ENTRYSHIFTDECREMENT 0x00

// flags for display on/off control
#define LCD_DISPLAYON 0x04
#define LCD_DISPLAYOFF 0x00
#define LCD_CURSORON 0x02
#define LCD_CURSOROFF 0x00
#define LCD_BLINKON 0x01
#define LCD_BLINKOFF 0x00
```

```
// flags for display/cursor shift
#define LCD_DISPLAYMOVE 0x08
#define LCD_CURSORMOVE 0x00
#define LCD_MOVERIGHT 0x04
#define LCD_MOVELEFT 0x00

// flags for function set
#define LCD_8BITMODE 0x10
#define LCD_4BITMODE 0x00
#define LCD_2LINE 0x08
#define LCD_1LINE 0x00
#define LCD_5x10DOTS 0x04
#define LCD_5x8DOTS 0x00

uint8_t _displaycontrol;
uint8_t _displayfunction;
uint8_t _displaymode;
uint8_t _numlines;

int _row_offsets[4] = {0x00, 0x40, 0x00 + 16, 0x40 + 16};

/************ low level data pushing commands **********/

// pulses the EN pin for 1ms
void pulseEnable(void) {
    pioDigitalWrite(EN, 0);
    tcDelayMicroseconds(1);
    pioDigitalWrite(EN, 1);
    tcDelayMicroseconds(1);    // enable pulse must be >450ns
    pioDigitalWrite(EN, 0);
    tcDelayMicroseconds(100);   // commands need > 37us to settle
}

// write character to d0-7 bus, then pulses enable to push
void writeBits(uint8_t value) {

    pioDigitalWrite(D0, (value & (1 << 0)) >> 0);
    pioDigitalWrite(D1, (value & (1 << 1)) >> 1);
    pioDigitalWrite(D2, (value & (1 << 2)) >> 2);
    pioDigitalWrite(D3, (value & (1 << 3)) >> 3);
    pioDigitalWrite(D4, (value & (1 << 4)) >> 4);
    pioDigitalWrite(D5, (value & (1 << 5)) >> 5);
    pioDigitalWrite(D6, (value & (1 << 6)) >> 6);
    pioDigitalWrite(D7, (value & (1 << 7)) >> 7);

    pulseEnable();
}

// write either command or data, with automatic 4/8-bit selection
void send(uint8_t value, uint8_t mode) {
```

```
    pioDigitalWrite(RS, mode);
    writeBits(value);
}

/*********** mid level commands, for sending data/cmds */

void command(uint8_t value) {
    send(value, 0);
}

void write(uint8_t value) {
    send(value, 1);
}

/********** high level commands, for the user! */
void clear()
{
    command(LCD_CLEARDISPLAY);  // clear display, set cursor position to zero
    tcDelayMicroseconds(2000);  // this command takes a long time!
}

void home()
{
    command(LCD_RETURNHOME);  // set cursor position to zero
    tcDelayMicroseconds(2000);  // this command takes a long time!
}

// assumes input is valid
// row <
void setCursor(uint8_t col, uint8_t row)
{
    command(LCD_SETDDRAMADDR | (col + _row_offsets[row]));
}

// Turn the display on/off (quickly)
void noDisplay() {
    _displaycontrol &= ~LCD_DISPLAYON;
    command(LCD_DISPLAYCONTROL | _displaycontrol);
}
void display() {
    _displaycontrol |= LCD_DISPLAYON;
    command(LCD_DISPLAYCONTROL | _displaycontrol);
}

// Turns the underline cursor on/off
void noCursor() {
    _displaycontrol &= ~LCD_CURSORON;
    command(LCD_DISPLAYCONTROL | _displaycontrol);
}
void cursor() {
```

```
    _displaycontrol |= LCD_CURSORON;
    command(LCD_DISPLAYCONTROL | _displaycontrol);
}

// Turn on and off the blinking cursor
void noBlink() {
    _displaycontrol &= ~LCD_BLINKON;
    command(LCD_DISPLAYCONTROL | _displaycontrol);
}
void blink() {
    _displaycontrol |= LCD_BLINKON;
    command(LCD_DISPLAYCONTROL | _displaycontrol);
}

// These commands scroll the display without changing the RAM
void scrollDisplayLeft(void) {
    command(LCD_CURSORSHIFT | LCD_DISPLAYMOVE | LCD_MOVELEFT);
}
void scrollDisplayRight(void) {
    command(LCD_CURSORSHIFT | LCD_DISPLAYMOVE | LCD_MOVERIGHT);
}

// This is for text that flows Left to Right
void leftToRight(void) {
    _displaymode |= LCD_ENTRYLEFT;
    command(LCD_ENTRYMODESET | _displaymode);
}

// This is for text that flows Right to Left
void rightToLeft(void) {
    _displaymode &= ~LCD_ENTRYLEFT;
    command(LCD_ENTRYMODESET | _displaymode);
}

// This will 'right justify' text from the cursor
void autoscroll(void) {
    _displaymode |= LCD_ENTRYSHIFTINCREMENT;
    command(LCD_ENTRYMODESET | _displaymode);
}

// This will 'left justify' text from the cursor
void noAutoscroll(void) {
    _displaymode &= ~LCD_ENTRYSHIFTINCREMENT;
    command(LCD_ENTRYMODESET | _displaymode);
}

// Allows us to fill the first 8 CGRAM locations
// with custom characters
void createChar(uint8_t location, uint8_t charmap[]) {
    location &= 0x7; // we only have 8 locations 0-7
```

```
    command(LCD_SETCGRAMADDR | (location << 3));
    for (int i=0; i<8; i++) {
        write(charmap[i]);
    }
}


// initializer function
void lcdBegin(uint8_t cols, uint8_t lines) {

    pioInit();
    tcDelayInit();

    _displayfunction = LCD_8BITMODE | LCD_1LINE | LCD_5x8DOTS;
    _displayfunction |= LCD_2LINE;

    if (lines > 1) {
        _displayfunction |= LCD_2LINE;
    }
    _numlines = lines;

    // set pinouts with PIO header
    pioPinMode(RS, PIO_OUTPUT);
    pioPinMode(EN, PIO_OUTPUT);
    pioPinMode(D0, PIO_OUTPUT);
    pioPinMode(D1, PIO_OUTPUT);
    pioPinMode(D2, PIO_OUTPUT);
    pioPinMode(D3, PIO_OUTPUT);
    pioPinMode(D4, PIO_OUTPUT);
    pioPinMode(D5, PIO_OUTPUT);
    pioPinMode(D6, PIO_OUTPUT);
    pioPinMode(D7, PIO_OUTPUT);

    // SEE PAGE 45/46 FOR INITIALIZATION SPECIFICATION!
    // from datasheet, need at least 40ms after power rises above 2.7V, wait 50
    tcDelayMicroseconds(50000);
    // pull both RS and R/W low to begin commands
    pioDigitalWrite(RS, 0);
    pioDigitalWrite(EN, 0);

    //put the LCD into 8 bit mode
    // this is according to the hitachi HD44780 datasheet
    // page 45 figure 23

    // Send function set command sequence
    command(LCD_FUNCTIONSET | _displayfunction);
    tcDelayMicroseconds(4500);  // wait more than 4.1ms

    // second try
    command(LCD_FUNCTIONSET | _displayfunction);
```

```
    tcDelayMicroseconds(150);

    // third go
    command(LCD_FUNCTIONSET | _displayfunction);

    // finally, set # lines, font size, etc.
    command(LCD_FUNCTIONSET | _displayfunction);

    // turn the display on with no cursor or blinking default
    _displaycontrol = LCD_DISPLAYON | LCD_CURSOROFF | LCD_BLINKOFF;
    display();

    // clear it off
    clear();

    // Initialize to default text direction (for romance languages)
    _displaymode = LCD_ENTRYLEFT | LCD_ENTRYSHIFTDECREMENT;
    // set the entry mode
    command(LCD_ENTRYMODESET | _displaymode);

}

// clears the screen and prints text to both rows of the LCD
void printToLCD(uint8_t row1[], uint8_t row2[], uint8_t length1, uint8_t length2) {

    clear();

    // loop through top row
    for (int i = 0; i < length1; ++i) {
        setCursor(i, 0);
        write(row1[i]);
    }
    // loop through bottom row
    for (int i = 0; i < length2; ++i) {
        setCursor(i, 1);
        write(row2[i]);
    }

}
```

Appendix C: PWM Module Block Diagram