The Oránge

ENGR 155 - Fall 2017

Natalie Kadonaga and Lillian Liang

Abstract

An FPGA, Raspberry Pi microcontroller, a BlueSMiRF Bluetooth module, and SEN-11574 pulse sensor were used to build a wireless heart rate zone monitor system called the Oránge. The team was inspired by the Orangetheory workout program, which defines five heart rate zones - gray, blue, green, orange, and red zones. The objective of the workout program is to stay in the "orange" heart rate zone, which is 84-91% of the user's maximum heart rate [1]. The FPGA uses an SPI protocol with an ADC to convert the raw pulse sensor data to a digital signal and uses the UART protocol and a BlueSMiRF Bluetooth device to wirelessly transmit the sensor data to the Raspberry Pi. The Raspberry Pi runs an Apache2 web server user interface with a form for the device user to submit their age. Finally, the Pi uses the age and transmitted sensor data to calculate the heart rate and drive one of five colored LEDs corresponding to the user's heart rate.

1 Introduction

1.1 Project Overview

This project involves a device that works as a heart rate zone indicator. The user inputs their age into a web server page; the Raspberry Pi calculates the user's heart rate zones and starts retrieving pulse data from the pulse sensor worn by the user. The user attaches the pulse sensor to a fingertip using Velcro. The sensor shines its LED into the user's capillary tissue and outputs an oxygen saturation level based on the light that is reflected. The pulse data is processed and sent over Bluetooth to the Pi, where a program calculates a heart rate and outputs the corresponding heart rate zone LED.

The high level block diagram is shown in Fig. 1. The raw analog pulse data from the pulse sensor is converted to a 10-bit digital code by the ADC. The ADC acts as the SPI slave to the FPGA on the MuddPi utility board. The FPGA repackages the 10-bit digital code from the ADC into a UART packet that holds the 8 most significant bits of data. The FPGA sends the UART packet serially to the BlueSMiRF Bluetooth module. This module is connected to the Pi's built-in Bluetooth peripheral. The Pi reads the pulse data from Bluetooth and runs a program that calculates the heart rate by detecting peaks in the pulse data. The program determines the corresponding heart rate zone based on the user's age submitted on the Pi's web server, and updates one of the colored LEDs.

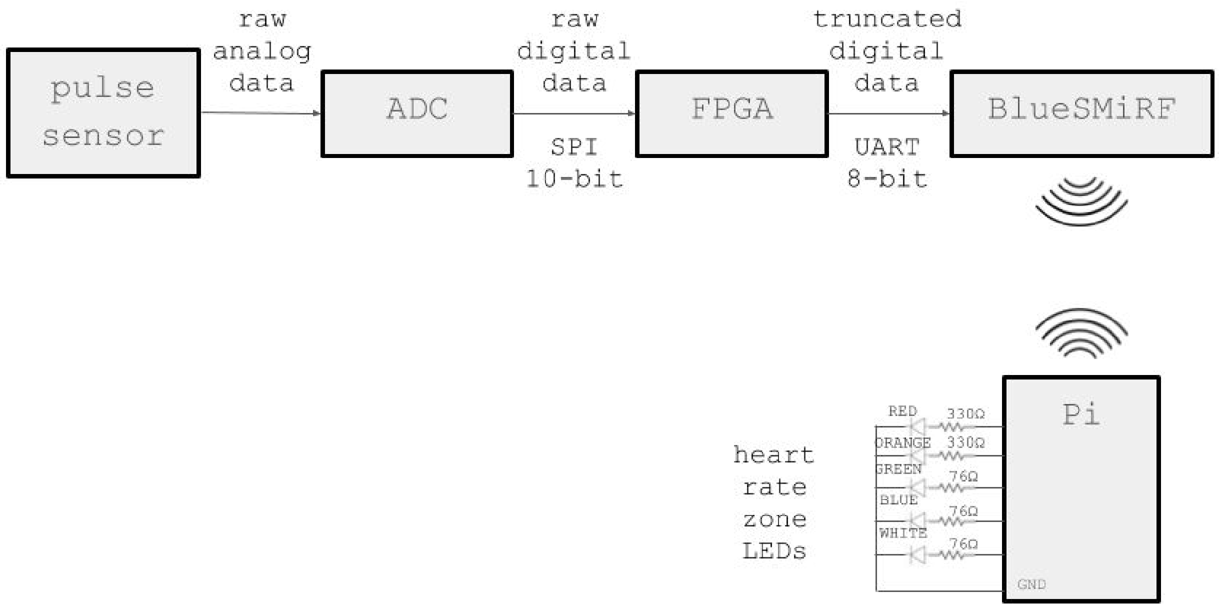


Figure 1: A block diagram displaying the components and interfaces of the system.

1.2 Deliverables

Deliverables for the project included the following:

- Successful wireless transmission of heart rate sensor data via Bluetooth
- Accurate detection of heart rate, as verified by heart rate phone app
- User interface, a web server page w/ age input
- Correct indication of heart rate zone by colored LED

2 Hardware

To accomplish these deliverables, the team used a SEN-11574 pulse sensor, MCP3002 ADC, T9JRN41-1 BlueSMiRF Bluetooth module, Raspberry Pi 3.0, and MuddPi IV, and five colored LED's, as shown in Figure 2.

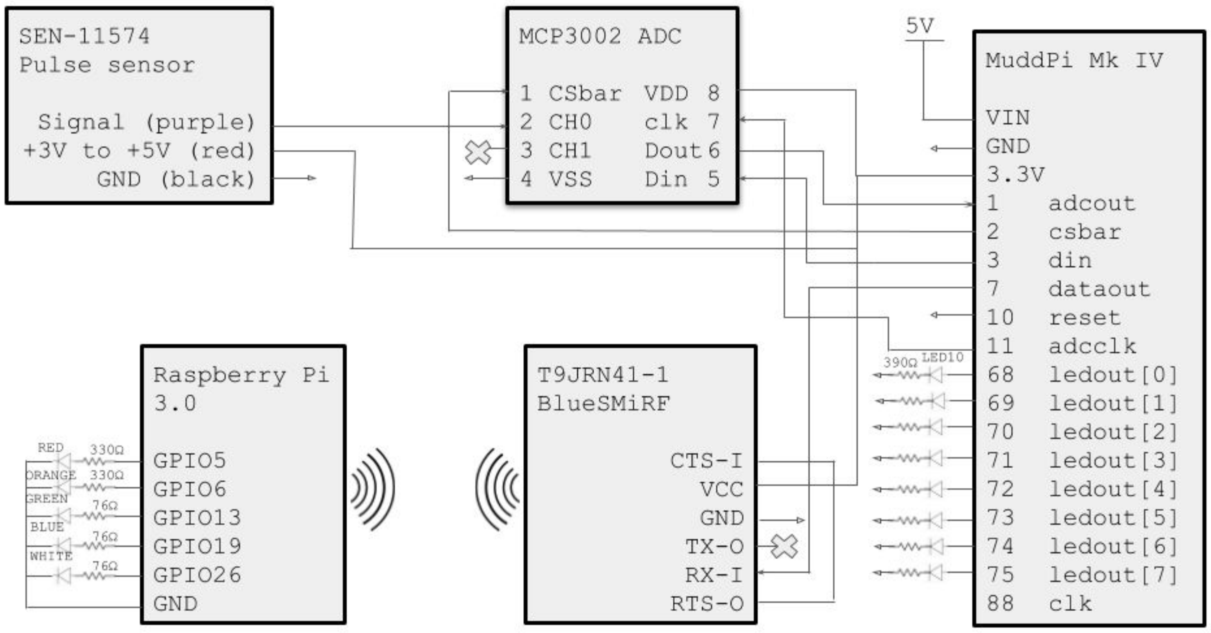


Figure 2: Breadboard circuit schematic.

2.1 New Hardware

The team used two new hardware peripherals in the Oránge: the SEN-11574 pulse sensor and the T9JRN41-1 BlueSMiRF bluetooth module, discussed below.

2.1.1 SEN-11574 Sensor

The SEN-11574 sensor is a pulse oximeter, a device that uses light to detect oxygen saturation in the blood. Typically, pulse oximeter probes are placed on the earlobe or finger [2]; in this project, the team primarily tested the SEN-11574 using fingers. The SEN-11574 has three pins: power, ground, and signal, indicated by red, black, and purple wires respectively. In this project, the pulse sensor is powered by 3.3V from the MuddPi utility board and the signal is an input to Channel 0 of the MCP3002 ADC, which converts the raw analog sensor signal to a 10-bit digital value (in the case of the Oránge, this 0-1023 digital value is proportional to voltage value of the sensor output, between 0-3.3V). Raw sensor oscilloscope outputs are shown below in Figure 3. Throughout the project, the raw sensor output was scoped on different human subjects for varying lengths of time, revealing changing DC offsets and changing peak-to-peak values, features that influenced the team's software implementation of peak detection. The sensor experienced significantly better signal integrity when gently attached to the finger with a Velcro strap.

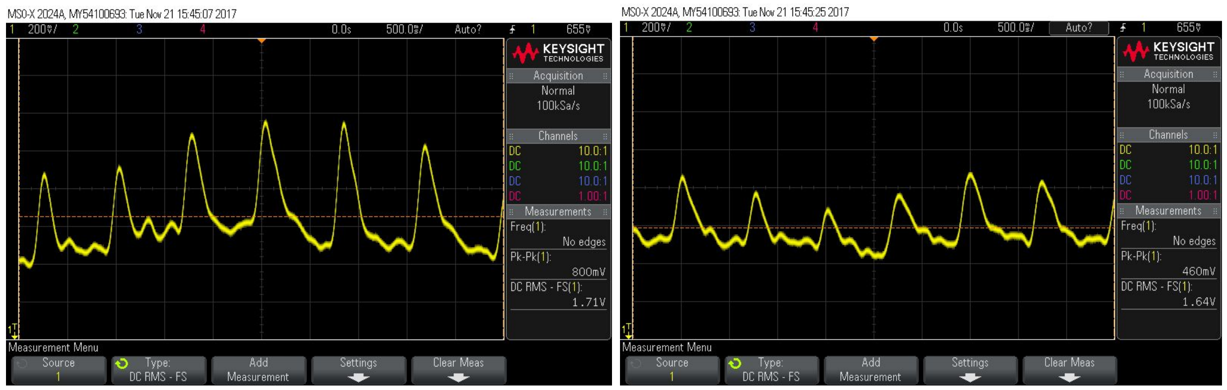


Figure 3: The oscilloscope outputs revealed that both DC offsets (left), and peak amplitude changed over time.

2.1.2 T9JRN41-1 BlueSMiRF Bluetooth Module

The BlueSMiRF Bluetooth modules has six pins: clear to send (CTS-I), power, ground, serial transmit (TX-O), serial receive (RX-I), and request to send (RTS-O). In this project, the BlueSMiRF is powered by 3.3V from the MuddPi utility board, the clear to send and request to send pins are connected, the serial receive is connected to `dataout` pin of the MuddPi board, and the serial transmit is unused.

2.2 FPGA Design

The FPGA creates signals for SPI communication with an ADC slave and UART communication with Pi via BlueSMiRF, as shown in Figure 4.

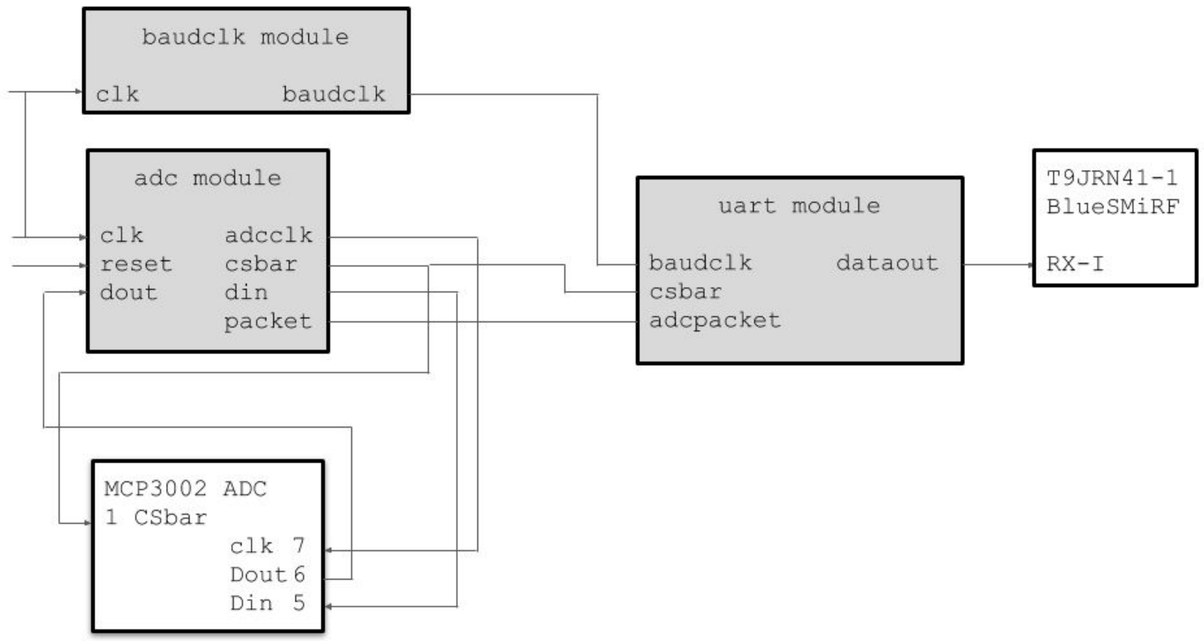


Figure 4: FPGA Block Diagram.

2.2.1 SPI with FPGA master and ADC slave

The `adc` FPGA module implements a SPI communication protocol with a FPGA master and MCP3002 ADC slave. The module takes in a `clk`, `reset`, and `adcout` (MISO) and outputs a 312.5 kHz `adcclk`, `csbar`, `din` (MOSI), and 10-bit data `packet`. The `adcclk` is generated from the 40MHz FPGA clock. The FPGA communicates with the ADC using a FSM with 32 states. The FSM switches states at 625 kHz, which is created from the 40 MHz clock from the FPGA and is twice as fast as the ADC clock. While the ADC communicates the 10-bit digital code over only 16 cycles of the ADC clock, as displayed in Figure 5, 32 states were necessary for the MOSI values to be set on the falling edge of the ADC clock.

The FSM's state 0 is when it sets the `csbar` low. States 0 through 7 output the configuration bits to D_{IN}, which include Start, SGL/DIFF, ODD/SIGN, and MSBF. The bit is changed on an even state number, and therefore the falling edge of the clock, because the value must settle before the ADC reads its D_{IN} on the clock's rising edge. States 8 and 9 wait for the null bit from the ADC D_{OUT} to pass. On the odd states from state 11 through 29, the FPGA reads the 10 bit digital code from D_{OUT}, MSB first. On states 30 and 31, the FPGA sets `csbar` high again to signal the end of data transmission.

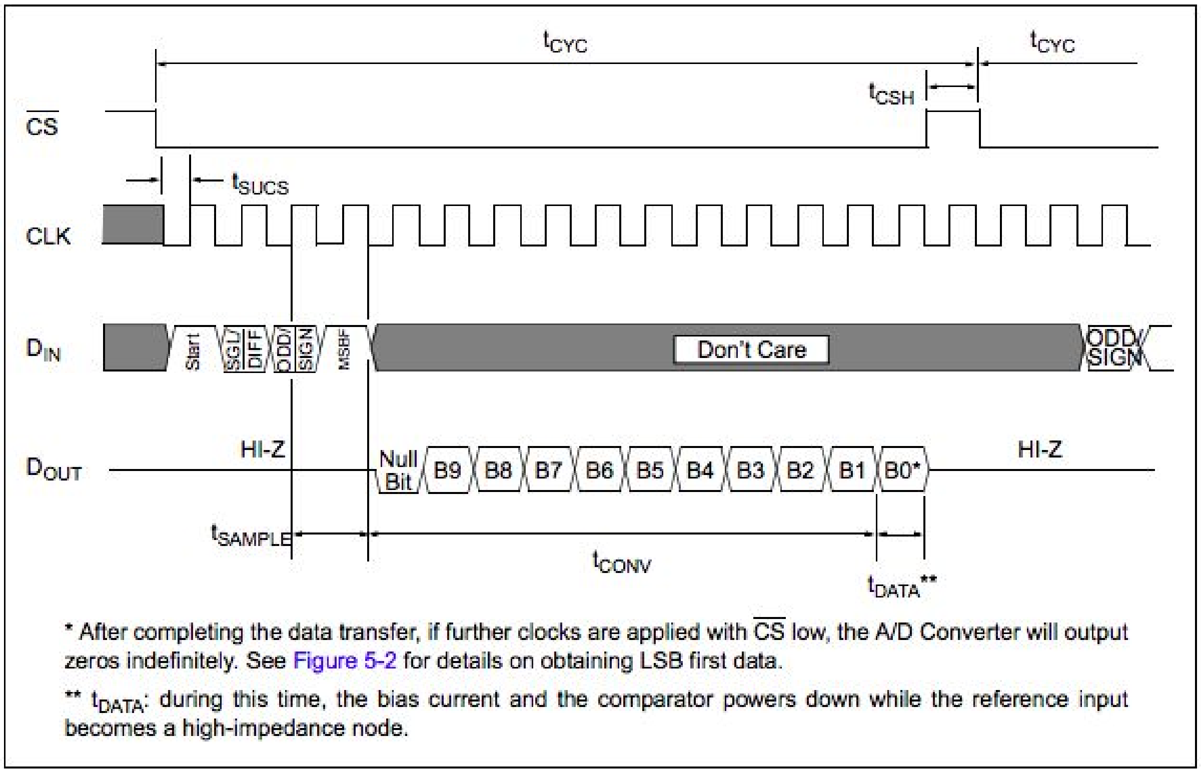

tCYC
tCYC
tCSH
CS
tSUCS
CLK
DIN
Start
SGL/DIFF
ODD/SIGN
MSBF
Don't Care
ODD/SIGN
DOUT
HI-Z
Null Bit
B9
B8
B7
B6
B5
B4
B3
B2
B1
B0*
HI-Z
tSAMPLE
tCONV
tDATA**
* After completing the data transfer, if further clocks are applied with CS low, the A/D Converter will output zeros indefinitely. See Figure 5-2 for details on obtaining LSB first data.
** tDATA: during this time, the bias current and the comparator powers down while the reference input becomes a high-impedance node.

Figure 5: SPI MCP3002 ADC waveforms [2].

2.2.2 UART BlueSMiRF signal

The BlueSMiRF communicates using UART and specifies a baud rate of 115.2 kHz. The `baudclk` FPGA module creates this baud rate using a digitally controlled oscillator that converts the FPGA's 40 MHz clock to a ~115.6 kHz clock, which is within the baud rate tolerance. The `uart` FPGA module that generates the UART signal takes in the generated baud clock from the `baudclk` module and `csbar` and the 10-bit `adcpacket` from the `adc` module, and outputs a UART signal for the BlueSMiRF to transmit to the Pi. The UART protocol includes a low start and high stop bit and 8 bits of data, transmitted LSB first, as shown by Figure 6 below. Because the UART data packet is 8 bits, the 2 least significant bits from the 10-bit data packet from the ADC are removed before transmission.

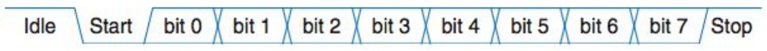


Figure 6: UART signal format [4].

3 Raspberry Pi Design

The Raspberry Pi microcontroller calculates the heart rate, drives the LED's, and hosts the user interface.

3.1 Heart Rate Calculation

The Pi program retrieves the pulse data at a sampling rate of 20 Hz from the built-in Bluetooth file. The Bluetooth file is bound to the MAC address of the BlueSMiRF.

To calculate the heart rate from the pulse data, the program checks whether the pulse data value is above or below a dynamically determined threshold. This threshold is calculated by taking the average of the minimum and maximum values from the previous heart beat. When three consecutive values match the pattern of the first value above the threshold and the second and third values below the threshold, the program detects a new heart beat. When the program detects a new heart beat, it records the number of samples since the start of the previous heart beat and stores it if the samples indicate a reasonable length of an interval between heart beats. Then, the program updates the user's heart rate by taking the average of the 20 most recent intervals and converting it to BPM. This calculation method is documented in Figure 7.

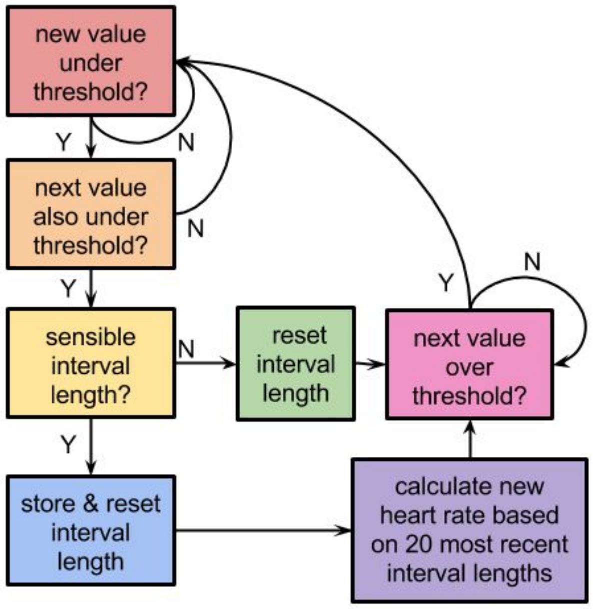

new value under threshold?
Y
N
next value also under threshold?
N
Y
sensible interval length?
N
reset interval length
next value over threshold?
Y
N
Y
store & reset interval length
calculate new heart rate based on 20 most recent interval lengths

Figure 7: State diagram for heart rate calculation algorithm.

3.2 LED driver

When the program recalculates the heart rate, it also determines the zone for the heart rate and drives the corresponding LED. The LEDs will display the heart rate zone for the entirety of the program's runtime, excepting the first 20 heart beats. The runtime length is configurable as a preprocessor directive constant.

3.3 User Interface

The Raspberry Pi runs an Apache2 web server. The first page is a form with a field to enter an age. Submitting the form will start the pulse data collection and heart rate zone indicator on the LEDs. After the program ends, a different web page will display buttons to rerun or reconfigure the program and a history of the achieved heart rate zones.

4 Results, Challenges, and Future Development

The team successfully transmitted the heart rate sensor data via Bluetooth, accurately detected heart rate, as verified by a heart rate phone app and with oscilloscope frequency measurements, implemented a user interface webpage with an age input form, and correctly indicated the heart rate zone by lighting a colored LED.

The team experienced difficulty with reading a clean, clear pulse data signal from the Pi Bluetooth file. The Bluetooth file would not retrieve new values quickly enough if it was left open for reading, so the file would need to be closed and reopened. When the Bluetooth file is first opened, the values in the first 30-40 ms would only be -1, so this required that the pulse data be recorded every 50 ms or longer. With a slower sampling rate, the likelihood of catching the peak values at precisely the correct time is more unlikely. This would introduce more variation in the pulse interval lengths. To tolerate for the variation, the heart rate calculation algorithm used averaging so that the calculated heart rate did not fluctuate too quickly. Allowing the program to reject values outside of a reasonable range also contributed to steady, accurate heart rates. Oddly, the digital 10 bit binary output of the ADC tended to favor values 63, 111, and 127. This resulted in a granular signal with sharp peaks, attached in Appendix C, rather than a smooth curve similar to the oscilloscope trace shown in Figure 3.

Potential future areas of development for The Oránge include making the device portable and testing the device during real exercise and improving the user interface to include a heart rate zone status message and a workout countdown timer.

6 References

[1] “Orangetheory takes science, fitness to the next level.” http://dcrefined.com/sports-fitness/orangetheory-takes-science-fitness-to-the-next-level. 2016.

[2] "What is Pulse Oximetry?", *Hopkinsmedicine.org*, 2017. [Online]. Available: https://www.hopkinsmedicine.org/healthlibrary/test_procedures/pulmonary/oximetry_92,P07754. [Accessed: 08- Dec- 2017].

[3] "MCP3002", *Ww1.microchip.com*, 2010. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/21294E.pdf. [Accessed: 08- Dec- 2017].

[4] Harris S, Harris D. Digital Design And Computer Architecture: ARM Edition. Waltham, MA: Elsevier; 2016:531.e17.

7 Parts List

Part	**Source**	**Part number**	**Price**
Pulse sensor	SparkFun	SEN-11574	$24.95 + $8.73 shipping
ADC	FPGA Component	MCP3002	N/A
LEDs (white, blue, green, orange, red)	Stock room	None	N/A
BlueSMiRF wireless module	Stock room	T9JRN41-1	N/A

8 Appendix

Appendix A SystemVerilog Code

```
// top level module
// creates signals for SPI communication with MCP3002 ADC and UART wireless communication with
Pi via BlueSMiRF device
// input: clk (40 MHz), reset, adcout (sensor data from ADC output)
// output: adcclk, csbar, din (SPI), dataout (UART), ledout (drives 8 LEDs to display data
output for use in testing)

module orange_final(input logic clk, reset, adcout,
                            output logic adcclk, csbar, din, dataout,
                            output logic[7:0] ledout);

       logic[9:0] packet; // internal MSBF data packet retrieved from ADC output, to be used
in uart module
       logic bclk; // baudrate for use in uart module

       // SPI communication with MCP3002 ADC
       adc adc(clk, reset, adcout, adcclk, csbar, din, packet);
       // create baudrate for UART communication with Pi via BlueSMiRF device
       baudclk baudclk(clk, reset, bclk);
       // UART communication with Pi via BlueSMiRF device
       uart uart(bclk, csbar, packet, dataout, ledout);
endmodule

// packet module
// creates signals for SPI with MCP3002 ADC slave.
// input: clk (40 MHz), reset, dout (sensor data at ADC output)
// output: adcclk (312.5 kHz), csbar, din, packet (10 bits, MSB first)

module adc(input logic clk, reset, dout,
                            output logic adcclk, csbar, din,
                            output logic [9:0] packet);
       // slowclk set to 625 kHz.
       logic slowclk;
       logic [5:0] counter;
       always_ff @(posedge clk, posedge reset) begin
              if (reset) begin
                     counter <= 0;
              end else begin
                     counter <= counter + 6'b1; // = (625000Hz/40MHz)*(2^6)
              end
       end
       assign slowclk = counter[5];
```

```
	// FSM to get data packet from dout signal
	// 16 cycles:
	// (1-5) Master drives start, SGL/DIFF, ODD/SIGN, MSBF, null.
	// (6-15) Master gets data from slave with shift register, 10 bits of data MSBF.
	// (16) drive csbar high when bits are done shifting
	// Use 32-state implementation. In (6-15), get data on the odd states so that data bits
are stable when saved.
	// State register
	logic [4:0] state; // 32 states, counting up at slowclk rate
	always_ff @(posedge clk, posedge reset)
		if (reset) state <= 0;
		else if (slowclk) state <= state + 5'b1;

	// Packet update register
	always_ff @(posedge clk, posedge reset)
		if (reset) packet <= 0;
		// Only read in from dout when ADC is transferring data.
		else if (adcclk && state >= 11 && state < 30) packet <= {packet[8:0], dout};

	// Output logic
	assign csbar = (state == 30 | state == 31); // goes high when 10 data bits are done
shifting
	assign din = (state == 0 | state == 1 | state == 2 | state == 3 | state == 6 | state ==
7); // start, SGL/DIFF, MSBF bits
	assign adcclk = (state % 2 == 1); // 625 kHz / 2 clk, high when count is odd
endmodule

// baudclk module
// creates baudrate for UART Communication
// input: clk (40 MHz), reset
// output: baudclk (115.2 kHz)

module baudclk(input logic clk, reset,
                              output logic baudclk);
	logic [7:0] counter; // count to 173
	always_ff@(posedge clk, posedge reset) begin
		if (reset) begin
			counter <= 0;
			baudclk <= 0;
		end else begin
			counter <= counter + 8'b1;
			if (counter == 173 ) begin // check if 173 is reached. If so,
				counter <= 0; // reset counter
				baudclk <= ~baudclk; // toggle baudclk
			end
		end
	end
endmodule

// uart module
// creates signals for UART communication with Pi via BlueSMiRF
// input: baudclk (output of baudclk module, baud rate of 115.2 kHz), csbar (internal signal
in adc module, marks when data bits are done shifting), adcpacket (10 bit data packet from adc
module)
// output: dataout(10 bit UART signal: start, 8 bits data LSB first, stop), ledout (drives 8
LEDs to display data output for use in testing)

module uart(input logic baudclk, csbar,
                             input logic[9:0] adcpacket,  // MSB first
                             output logic dataout,               // LSB first
                             output logic[7:0] ledout);

       // save the input data when the data is done transmitting
       logic[9:0] savedadcpacket; // where input data packet from ADC module will be saved MSB
first.
       assign savedadcpacket = csbar ? adcpacket : savedadcpacket; // bits are done shifting
into the input ADC packet when csbar goes high, so that's when the input ADC data is saved

       // truncate the 10 bit input data to 8 bits for use in the UART protocol
       logic[7:0] packet8msb; // where the truncated data packet will be saved
       assign packet8msb = savedadcpacket[9:2]; // truncate the data saved ADC data packet

       // create the UART data packet
       logic[9:0] uartpacket; // START(0), 8 data bits (LSB first), STOP (1).
       assign uartpacket = {1'b0, packet8msb[0], packet8msb[1], packet8msb[2],
              packet8msb[3], packet8msb[4], packet8msb[5], packet8msb[6],
              packet8msb[7], 1'b1};

       // create a UART signal corresponding to that data packet
       // State register
       logic [3:0] state, nextstate;
       always_ff@(posedge baudclk)
              state <= nextstate;

       // Next state logic
       assign nextstate = csbar ? 4b'1 : state + 4'b1;

       // Output logic
       always_comb
              case(state)
                     4'b0001: dataout = uartpacket[9]; // START bit (0)
                     4'b0010: dataout = uartpacket[8]; // LSB
                     4'b0011: dataout = uartpacket[7];
                     4'b0100: dataout = uartpacket[6];
                     4'b0101: dataout = uartpacket[5];
                     4'b0110: dataout = uartpacket[4];
                     4'b0111: dataout = uartpacket[3];
                     4'b1000: dataout = uartpacket[2];
                     4'b1001: dataout = uartpacket[1]; // MSB
                     4'b1010: dataout = uartpacket[0]; // STOP bit (1)
```

```
                        default: dataout = 1; // IDLE
                endcase

        // Light up LEDs to display data for use in testing and debugging
        assign ledout = uartpacket[8:1]; // LSB first (at bottom of LED array).
endmodule
```

Appendix B C Code

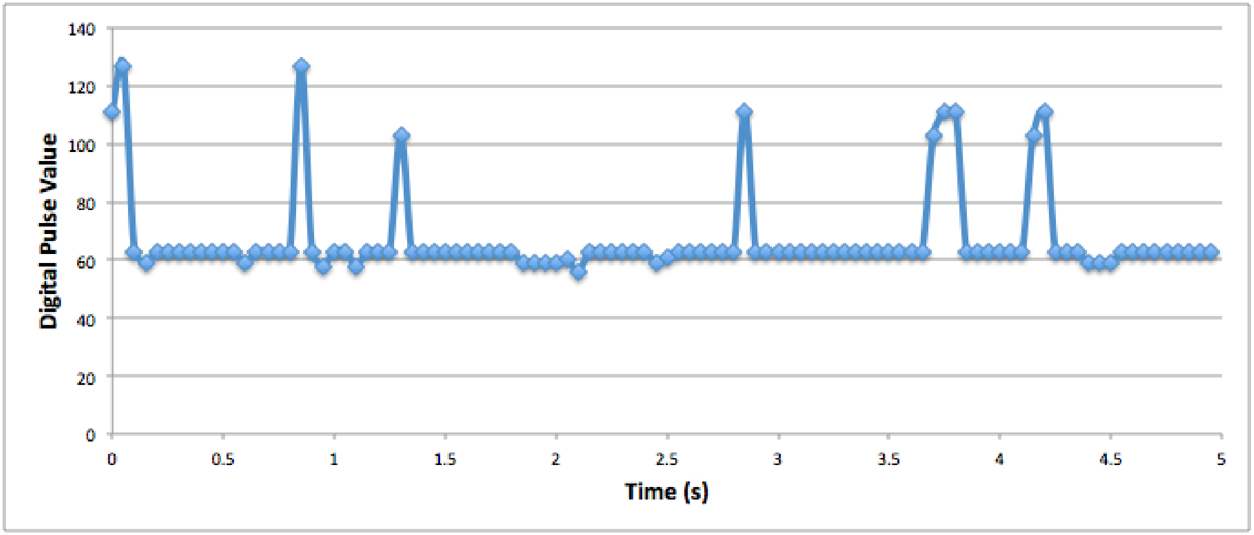
```
#include "EasyPIO.h"
#include <time.h>

/*************************************************
* CONSTANTS
*************************************************/

// Toggle debugging print statements.
#define DEBUG 0

// Milliseconds between taking pulse data samples.
#define SAMPLING_PERIOD 50
// Seconds to calculate the heart rate.
#define CALCULATE_SEC 30
// Seconds to collect and write pulse data to file.
#define COLLECT_SEC 20
// Number of recent heart beats used to calculated the heart rate.
#define NUM_RECENT 20

// Pi pinouts for each LED.
#define RED 5
#define ORANGE 6
#define GREEN 13
#define BLUE 19
#define GRAY 26

// If age is not read from server, use default age.
#define DEFAULT_AGE 20

/*************************************************
* GLOBAL VARIABLES
*************************************************/

// Pinouts for LEDs.
int pins[] = {RED, ORANGE, GREEN, BLUE, GRAY};
// Ranges for heart rate zones.
int zoneThresholds[] = {180, 160, 140, 120};

/*************************************************
* LED OUTPUT
*************************************************/
```

```c
/* Pulse all the LEDs for the given number of rounds. */
void flashLEDs(int rounds) {
  int i = 0;
  while (i < rounds) {
    digitalWrite(pins[i%5], 1);
    delayMillis(100);
    digitalWrite(pins[i%5], 0);
    delayMillis(100);
    i++;
  }
}

void turnAllLEDsOff() {
  int i;
  for (i=0; i<5; i++) {
    digitalWrite(pins[i], 0);
  }
}

/**
 * Turns on the LED output corresponding to the user's heart rate zone.
 */
void updateZoneLED(double heartRate) {
  turnAllLEDsOff();

  // Turn the relevant LED on.
  if (heartRate >= zoneThresholds[0]) {
    digitalWrite(RED, 1);
    printf("<p>INFO: RED ZONE.</p>\n");
  } else if (heartRate >= zoneThresholds[1]) {
    digitalWrite(ORANGE, 1);
    printf("<p>INFO: ORANGE ZONE.</p>\n");
  } else if (heartRate >= zoneThresholds[2]) {
    digitalWrite(GREEN, 1);
    printf("<p>INFO: GREEN ZONE.</p>\n");
  } else if (heartRate >= zoneThresholds[3]) {
    digitalWrite(BLUE, 1);
    printf("<p>INFO: BLUE ZONE.</p>\n");
  } else {
    digitalWrite(GRAY, 1);
    printf("<p>INFO: GRAY ZONE.</p>\n");
  }
}

/**************************************************
* DATA COLLECTION
**************************************************/

/* Read and return value from bluetooth and delay for SAMPLING_PERIOD. */
```

```c
unsigned int getNextAndWait() {
    // If this line segfaults, manually bind the device with:
    // sudo rfcomm bind hci0 00:06:66:04:B1:AC
    FILE *fp = fopen("/dev/rfcomm0", "r");
    if (fp == NULL) {
      printf("<p>ERROR: Could not open file. ");
      printf("Make sure BlueSMiRF is turned on and is bound to rfcomm.</p>\n");
      exit(1);
    }
    // Turn the bluetooth file into a non-blocking file.
    int fd = fileno(fp);
    int flags = fcntl(fd, F_GETFL, 0);
    fcntl(fd, F_SETFL, flags | O_NONBLOCK);

    int value = 0;

    // The first few values read from the bluetooth file are inaccurate,
    // so we throw them away.
    int i = 0;
    while (i < SAMPLING_PERIOD) {
      value = fgetc(fp);
      if (feof(fp)) {
        printf("<p>WARNING: Reached EOF.</p>\n");
        break;
      }
      delayMillis(10);
      i += 10;
    }

  // We use the value after waiting SAMPLING_PERIOD.
  if (DEBUG) printf("<p>%d</p>\n", value);
  return value;
}

/* Reads pulse data every SAMPLING_PERIOD and writes to a file when finished. */
void collectAndWriteData() {
  FILE *testfile = fopen("pulse_data_output.txt", "w");

  clock_t before = clock();

  int i = 0;
  int value = 0;

  // Run for approximately COLLECT_SEC seconds.
  while(i < COLLECT_SEC * 1000 / SAMPLING_PERIOD) {
    value = getNextAndWait();
    fprintf(testfile, "%d\n", value);
    printf("VALUE: %d\n", value);
    i++;
  }
  fclose(testfile);

  clock_t diff = clock() - before;
  printf("Time taken: ");
  printf("%d s %d ms\n", diff/CLOCKS_PER_SEC, (diff%CLOCKS_PER_SEC)/1000);
  printf("Time expected: %d s 0 ms\n", COLLECT_SEC);
}

/*************************************************
* SIGNAL PROCESSING
*************************************************/

/* Calculates and returns average of values in array. */
double getAverage(unsigned int array[NUM_RECENT]) {
  int sum = 0;
  int i;
  for (i = 0; i < NUM_RECENT; i++) {
    sum += array[i];
  }
  return 1.0*sum/NUM_RECENT;
}

/* Calculate heart rate by sampling data and update current heart rate. */
void calculateAndUpdateHeartRate() {
  // Pulse data output value.
  int value;
  // Num recorded values for current interval.
  unsigned int numValues = 0;
  // Values per interval for last NUM_RECENT intervals.
  unsigned int numValuesHist[NUM_RECENT];
  // Num intervals recorded this entire run.
  unsigned int numRecordedIntervals = 0;
  // Calculated heart rate in BPM.
  double heartRate = 0.0;
  // Threshold used to detect a new heart beat.
  // Recalculated every heart beat.
  int threshold = 100;
  // Minimum and maximum values. Used to calculate threshold.
  // Initial values of 200 and 0 will be changed.
  int min = 200;
  int max = 0;

  clock_t before = clock();

  // Find and set initial values for the min and max.
  while(1) {
    // Record values for only 2 seconds.
    if ((clock() - before)/CLOCKS_PER_SEC > 2) break;
    int value = getNextAndWait();
    if (value < min && value >= 0) min = value;
    if (value > max && value >= 0) max = value;
  }

  // Loop to continuously collect data and calculate new heart rates.
  while (1) {
    // Dynamic threshold is recalculated based on previous interval's
    // min and max.
    threshold = 1.0 * (min + max)/2;
    if (DEBUG) printf("Threshold set to %d.\n", threshold);

    int newMin = 200;
    int newMax = 0;

    // Check to stop calculating and updating the heart rate.
    if ((clock()-before)/CLOCKS_PER_SEC > CALCULATE_SEC) {
      printf("Done.\n");
      break;
    }

    // To detect a new interval, check for two values under threshold in
    // a row.
    do {
      value = getNextAndWait();
      if (value < newMin && value >= 0) newMin = value;
      if (value > newMax && value >= 0) newMax = value;
      numValues++;
    } // Wait for first occurrence of value under threshold.
    while (value >= threshold);

    // Check if the next value is also under threshold.
    value = getNextAndWait();
    if (value < newMin && value >= 0) newMin = value;
    if (value > newMax && value >= 0) newMax = value;
    numValues++;

    // If not, we haven't started a new interval and we start over from the
    // beginning of the while loop.
    if (value >= threshold) continue;
    if (DEBUG) printf("DEBUG (values per interval): %d\n", numValues);

    // After detecting a new interval, calculate what the heart rate would be
    // based on how long the previous interval was.
    double heartRateSingleInterval = 60000.0 / (numValues * SAMPLING_PERIOD);
    // Check if the heart rate would be too big or small.
    if (heartRateSingleInterval < 37.5 || heartRateSingleInterval > 250) {
      // Wait for value to go back over threshold.
      do {
        value = getNextAndWait();
        if (value < newMin && value >= 0) newMin = value;
        if (value > newMax && value >= 0) newMax = value;
      } while (value < threshold);

      // Start over in a new interval w/o recording numValues in numValuesHist.
      numValues = 0;
      continue;
    }

    // Store the number of values in this interval.
    numValuesHist[numRecordedIntervals % NUM_RECENT] = numValues;
    numRecordedIntervals++;
    // Reset number of values for new interval.
    numValues = 0;

    if (numRecordedIntervals >= NUM_RECENT) {
      double normalizedValue = getAverage(numValuesHist);
      double periodMillis = normalizedValue * SAMPLING_PERIOD;
      double periodMinutes = periodMillis/(1000 * 60);
      heartRate = 1/periodMinutes;
      updateZoneLED(heartRate);
      printf("<p>HEART RATE: %d BPM</p>\n", (int)heartRate);
    }

    // Wait for value to go back over threshold.
    do {
      value = getNextAndWait();
      if (value < newMin && value >= 0) newMin = value;
      if (value > newMax && value >= 0) newMax = value;
      numValues++;
    } while (value < threshold);

    if (newMin < newMax) {
      min = newMin;
      max = newMax;
    }
  }
}

/**************************************************
* USER INTERACTION
**************************************************/

/**
 * Returns age in years submitted by user through web server.
 */
int getAgeFromServer() {
  int age;
  char *qs = getenv("QUERY_STRING");
  if (qs == NULL) {
    printf("<p>WARNING: Could not read age from web server.</p>\n");
    return DEFAULT_AGE;
  }
  else if (sscanf(qs, "userage=%d", &age) != 1) {
    printf("<p>WARNING: Data must be numeric.</p>\n");
    return DEFAULT_AGE;
  } else {
    return age;
  }
}

/**
 * Calculates heart rate zone thresholds from user's age.
 */
void updateZoneThresholds(int age) {
  int max = 220-age;
  // Red zone: 92-100% of maximum heart-rate.
  zoneThresholds[0] = 0.92*max;
  // Orange zone: 84-91%.
  zoneThresholds[1] = 0.84*max;
  // Green zone: 71-83%.
  zoneThresholds[2] = 0.71*max;
  // Blue zone: 61-70%.
  zoneThresholds[3] = 0.61*max;
}

int main(void)
{
  pioInit();

  pinsMode(pins, 5, OUTPUT);

  flashLEDs(5);

  // HTML header.
  printf("%s%c%c\n", "Content-Type:text/html;charset=iso-8859-1",13,10);
  printf("<body>\n");

  int age = getAgeFromServer();
  updateZoneThresholds(age);

  printf("<p>Age set to %d.</p>\n", age);
  printf("<p>Heart rate zones (BPM):</p>\n");
  printf("<p>Red: %d and above</p>\n", zoneThresholds[0]);
  printf("<p>Orange: %d-%d</p>\n", zoneThresholds[1], zoneThresholds[0]-1);
  printf("<p>Green: %d-%d</p>\n", zoneThresholds[2], zoneThresholds[1]-1);
  printf("<p>Blue: %d-%d</p>\n", zoneThresholds[3], zoneThresholds[2]-1);
  printf("<p>Gray: %d and below</p>\n", zoneThresholds[3]-1);

  // Button to repeat.
  printf("<form action=\"/cgi-bin/finalproject?userage=%d\" method=\"POST\">\n", age);
  printf("<input type = \"submit\" value=\"Continue\">\n");
  printf("</form>\n");

  // Reset button.
  printf("<form action=\"/cgi-bin/orangeweb\" method=\"POST\">\n");
  printf("<input type = \"submit\" value=\"Reset\">\n");
  printf("</form>\n");

  printf("</body>\n");

  // Uncomment to collect pulse data and write to a file.
  // collectAndWriteData();

  calculateAndUpdateHeartRate();

  turnAllLEDsOff();

  return 0;
}
```

```
#include "EasyPIO.h"

int main(void)
{
  pioInit();
  printf("%s%c%c\n", "Content-Type:text/html;charset=iso-8859-1",13,10);
  printf("<body>\n");

  printf("<p>Enter age in years:</p>\n");

  printf("<form action=\"/cgi-bin/finalproject\">\n");
  printf("<input type=text name=\"userage\">\n");
  printf("<input type=\"submit\" value=\"Start\">\n");
  printf("</form>\n");

  printf("</body>\n");
  return 0;
}
```

Appendix C Digital Pulse Value vs. time


Digital Pulse Value
Time (s)