Heartrate Monitor for Rehabilitation Exercise Machine

With Fault Tolerance

Author: Curt Henrichs
Date Submitted November 9, 2017

Milwaukee School of Engineering Electrical Engineering and Computer Science Department
CE4920 : Embedded Systems IV | Professor Dr. Russ Meier

[This page was intentionally left blank.]

Abstract

The system developed for CE 4920, embedded systems IV, is a heartrate monitor and heartbeat compensated treadmill using a Cypress PSOC 5 processor. This processor provides mixed-signal customizable circuitry to signal condition and process analog inputs. The processor also provides numerous prebuilt hardware implementations of common technology such as USB UART and CapSense capacitive touch. For this application, the user uses a capacitive touch slider to wake the system and to start exercise profile selection. After profile selection, the system will ramp up the treadmill motor to the appropriate speed and start compensating the motor's speed based on the heartrate measured. If the heartrate rises then the motor is slowed down to not overtax the user, and vis-versa for decrease in heartbeat. Finally, if the system detects a fault on any critical hardware devices then it will lock the treadmill to ensure safety. This project aims to provide embedded IV students with a new development environment and a new design challenge that requires the full embedded systems model with fault tolerant considerations. The desired result is to achieve familiarity with the Cypress PSOC architecture in addition to a functional embedded system.

Table of Contents

Introduction

The heartrate compensated exercise machine is an embedded system developed using a Cypress PSOC 5 as the embedded controller. The system will accept a heartbeat signal from the user and control the treadmill motor depending on the measured heartbeat in beats per minute. The user will be able to select from several exercise profiles to match their fitness needs. This document aims to provide a description of the development progress for this project. The document is broken into a system design section, which will detail the general design developed for this application, and a milestone section that will detail the progress to implement the design through the development phase.

System Design

General design of the project is detailed in this section. Specifically, this section is broken into an overview to give context and scope to the project, hardware design to explain the current hardware solution, and software design to explain the current software application. For information on the design and implementation process, see the milestones section of this report.

Overview

The goal of this project is to develop and implement a heart rate compensated exercise machine with fault detection. The system will take the user's heartbeat as an input into the system and output a PWM signal that will control a DC motor. Additionally, an LCD and capacitive touch buttons form the user interface to allow exercise profile selection. Each profile will have a unique base speed and heartrate compensation behavior. Finally, a piezo buzzer is used to play a startup tune and a warning signal when heartbeat is exceed. While the system is designed as a standalone device, the system does provide a USB USART connection to a PC so heartbeat data can be streamed to the connected computer.

Fault detection is a major consideration in the reliability of the exercise product. Thus, the critical hardware devices will provide a signal to the ARM processor to check for fault events. If such an event were to occur then the treadmill will safely deactivate the motor and lock itself from further use. This should ensure safety until a hard reset occurs. However even if the user reboots, the system will quickly catch the offending error and proceed to lockout.

Hardware Design

The design philosophy behind this project is to use the PSOC 5 hardware when possible instead of relying on a software solution. When software is needed, a there is a preference towards interrupts to still allow for asynchronous behavior in regards to the hardware. While it does complicate the flow of the program, it will generally allow for a faster main loop, which should increase user interface responsiveness. Additionally, if in the future there exists a need to devote more processing to a specific task in the main program execution path as then the entire hardware behavior does not degrade in performance, responsive or otherwise.

Shown in Figure 1 is the hardware design for the user interface. Additionally the figure shows the three fault detection inputs into the system from the external interface hardware. Shown on the left is the CapSense module that interfaces the capacitive touch buttons and slider with the processor for the user interface. USB USART module, below the CapSense, is used to stream heartrate data to an optionally connected PC for data logging. The LCD module is used to drive the products visual user interface. On the top right of Figure 1 is the PWM module that is used to control the treadmill motor attached to this device. Below the PWM is the music generation hardware, which includes a PWM module that generates a 50% duty cycle, variable frequency waveform and a counter used to measure duration of the note being played. Finally in the lower left is the fault tolerance hardware. The three component operating properly, COP, signals are shown for the critical hardware. Under these inputs is the timer used to trigger the fault detection ISR that will either feed or expire the watchdog.

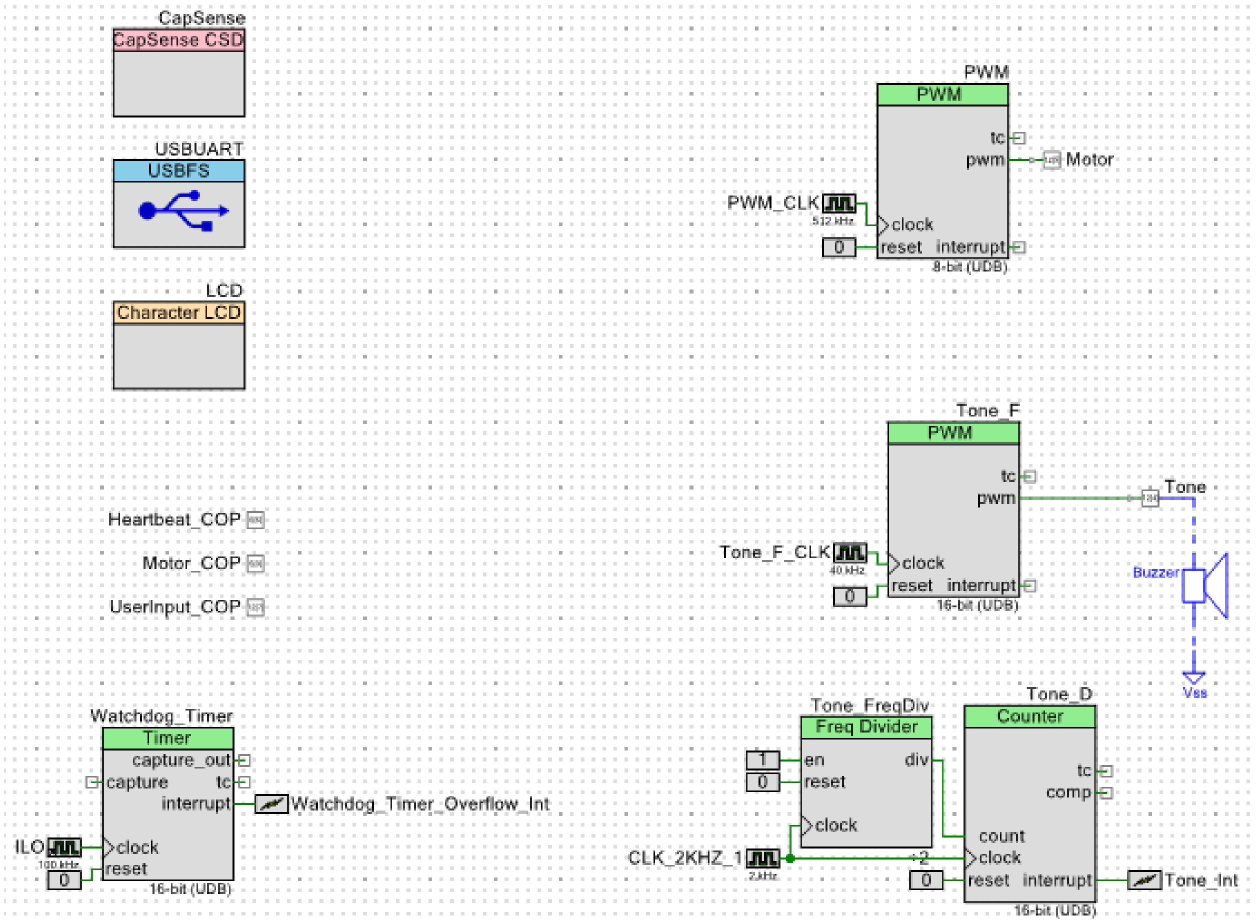


Figure 1: Block diagram for user interface hardware design.

Figure 2 depicts the hardware design for the heartbeat detection module. The module utilizes the mixed-signal environment of PSOC 5 to bring the amplifier, analog comparator, and ADC into the IC. A hardware debounce circuit is used to clean the analog comparator's result, where the resulting signal is used to generate an interrupt to the processor. A counter is used to track the duration of the heartbeat pulse for frequency calculation. This counter is attached to an interrupt that will trigger on an overflow event. Finally, it is worth noting that the program can select the input waveform source as either an internally generated waveform for testing or the external waveform from the user. Figure 2 also shows the heartbeat recreation DAC which generates a copy of the scaled, after PGA, input waveform as measured on the ADC. Note that a coupled part of the user interface is shown in this diagram; the LED that signals heartbeat detection by toggling on and off to each heartbeat peak.

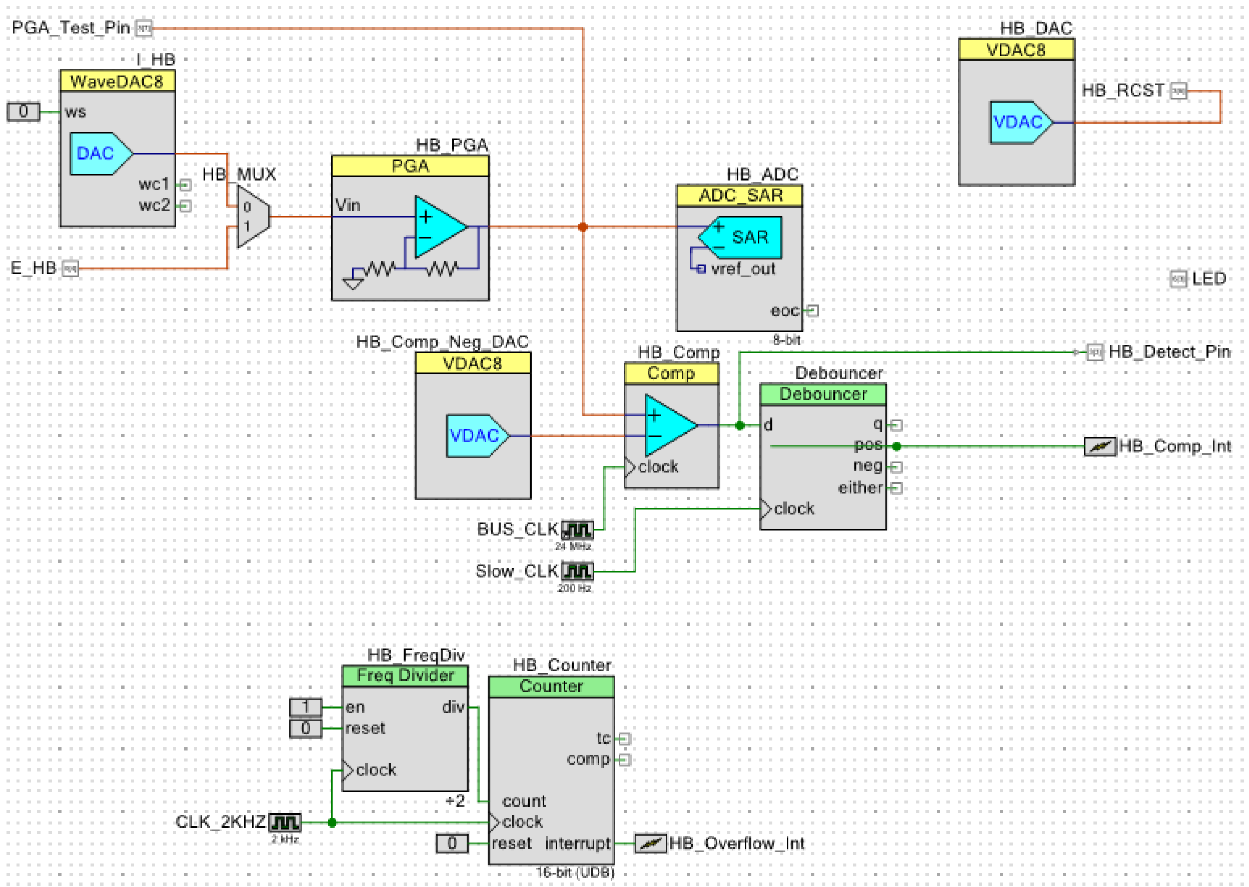


Figure 2: Block diagram for heartrate detection hardware.

Given the extensive documentation and development Cypress put into the hardware components and their editor, it was relatively simple to setup the hardware side of the project. The most complicated part was configuring the clock tree for the USB USART module but even that was well documented. For further discussion of how the modules interact with software read the milestone section of this report.

Software Design

When starting this project the first goal was to sketch out a UML diagram that would describe the general programming framework for the implementation phase. Figure 3 shows the result of this design process. Note that the UML diagram presented is not complete, as many private functions and attributes are not listed, thus it is best to consider this diagram as just a general framework to fill out during implementation.

The first prominent feature of the application is the Main module, which implements a state-machine to control the program logic and is responsible for motor control. This module is dependent directly on the Fault, Heartbeat, Display, Music, UserIn, Uart, and Profile Modules. This list along with RunningAverage become the files that makeup the program. Heartbeat, as the name suggests, will interface with the heartrate detection hardware to compute the beats per minute, BPM, of the user. To accomplish this the RunningAverage object is design to get updated with the current frequency read from hardware and to return an average result, thereby filtering unwanted noise.

User interface is composed of the Music, Uart, UserIn, and Display modules. Display encapsulates the low-level hardware and provides an application specific interface to the Main module to drive the visual user interface. The music module interfaces with the music hardware and runs almost exclusively via interrupts to allow the main program not to pay any penalty for integrating music into the application. Profile defines several singletons that describe the various exercise profile behavior. The user can interface with the program via the UserIn module, which is a wrapper on the CapSense interface to provide the required slider detection behavior not provided in Cypress' interface. Finally, a Uart module is used as a wrapper on the USB USART module to decouple the implementation from the main application.

Finally, fault tolerance is implemented in the Fault module. This module will start and update the watchdog timer in addition to generating a code for the current error. Note that the module supports the access of a critical section such as in emergency stop where the watchdog must wait before it can expire.

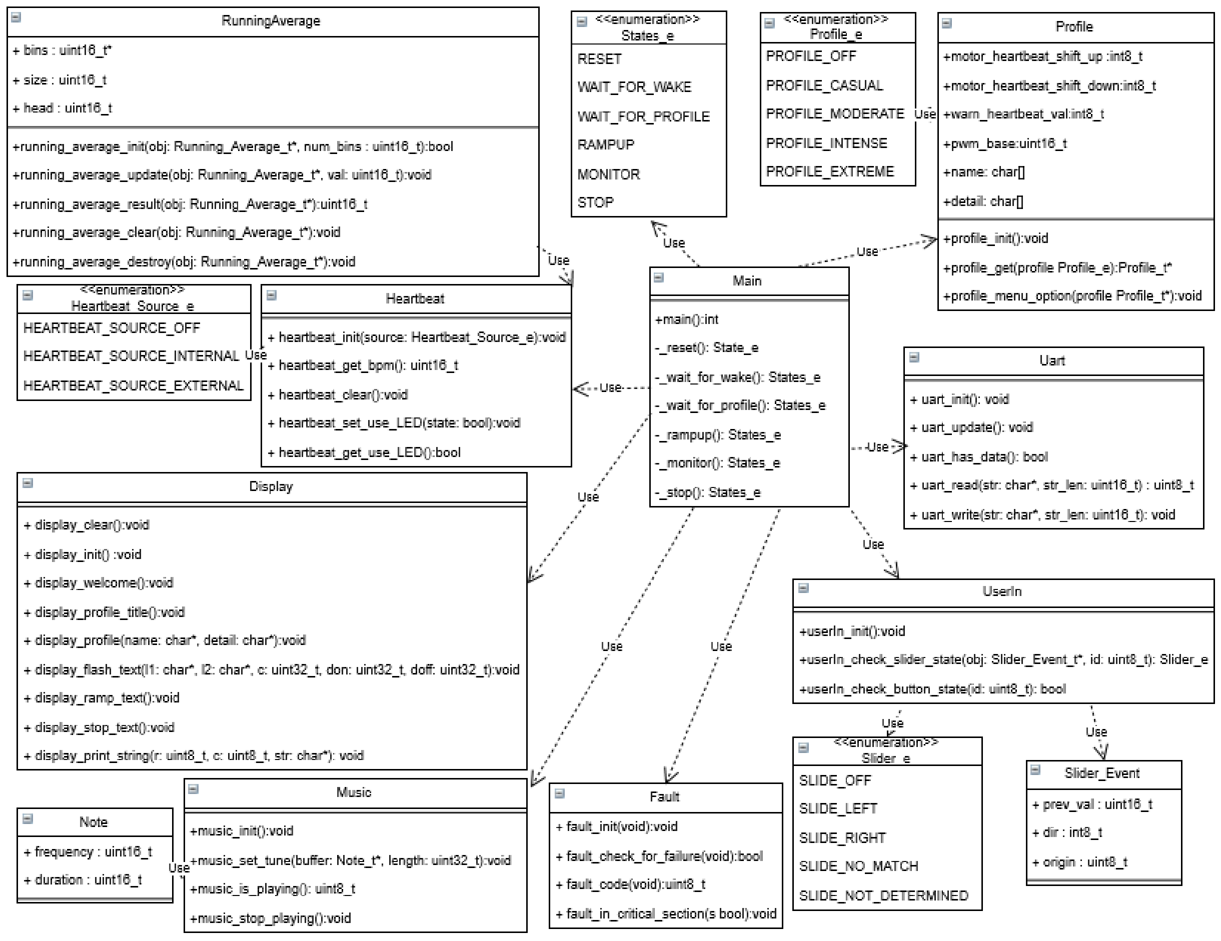

RunningAverage
+ bins : uint16_t*
+ size : uint16_t
+ head : uint16_t
+running_average_init(obj: Running_Average_t*, num_bins : uint16_t):bool
+running_average_update(obj: Running_Average_t*, val: uint16_t):void
+running_average_result(obj: Running_Average_t*):uint16_t
+running_average_clear(obj: Running_Average_t*):void
+running_average_destroy(obj: Running_Average_t*):void
<<enumeration>> States_e
RESET
WAIT_FOR_WAKE
WAIT_FOR_PROFILE
RAMPUP
MONITOR
STOP
<<enumeration>> Profile_e
PROFILE_OFF
PROFILE_CASUAL
PROFILE_MODERATE
PROFILE_INTENSE
PROFILE_EXTREME
Profile
+motor_heartbeat_shift_up :int8_t
+motor_heartbeat_shift_down:int8_t
+warn_heartbeat_val:int8_t
+pwm_base:uint16_t
+name: char[]
+detail: char[]
+profile_init():void
+profile_get(profile Profile_e):Profile_t*
+profile_menu_option(profile Profile_t*):void
<<enumeration>> Heartbeat_Source_e
HEARTBEAT_SOURCE_OFF
HEARTBEAT_SOURCE_INTERNAL
HEARTBEAT_SOURCE_EXTERNAL
Heartbeat
+ heartbeat_init(source: Heartbeat_Source_e):void
+ heartbeat_get_bpm(): uint16_t
+ heartbeat_clear():void
+ heartbeat_set_use_LED(state: bool):void
+ heartbeat_get_use_LED():bool
Main
+main():int
-_reset(): State_e
-_wait_for_wake(): States_e
-_wait_for_profile(): States_e
-_rampup(): States_e
-_monitor(): States_e
-_stop(): States_e
Uart
+ uart_init(): void
+ uart_update(): void
+ uart_has_data(): bool
+ uart_read(str: char*, str_len: uint16_t) : uint8_t
+ uart_write(str: char*, str_len: uint16_t): void
Display
+ display_clear():void
+ display_init() :void
+ display_welcome():void
+ display_profile_title():void
+ display_profile(name: char*, detail: char*):void
+ display_flash_text(l1: char*, l2: char*, c: uint32_t, don: uint32_t, doff: uint32_t):void
+ display_ramp_text():void
+ display_stop_text():void
+ display_print_string(r: uint8_t, c: uint8_t, str: char*): void
UserIn
+userIn_init():void
+userIn_check_slider_state(obj: Slider_Event_t*, id: uint8_t): Slider_e
+userIn_check_button_state(id: uint8_t): bool
<<enumeration>> Slider_e
SLIDE_OFF
SLIDE_LEFT
SLIDE_RIGHT
SLIDE_NO_MATCH
SLIDE_NOT_DETERMINED
Slider_Event
+ prev_val : uint16_t
+ dir : int8_t
+ origin : uint8_t
Note
+ frequency : uint16_t
+ duration : uint16_t
Music
+music_init():void
+music_set_tune(buffer: Note_t*, length: uint32_t):void
+music_is_playing(): uint8_t
+music_stop_playing():void
Fault
+ fault_init(void):void
+ fault_check_for_failure(void):bool
+ fault_code(void):uint8_t
+ fault_in_critical_section(s bool):void
Use

Figure 3: UML diagram for the complete software design. Note this is an abbreviated UML diagram as not all private functions, ISRs, and private attributes are noted. The purpose of this UML diagram is solely to describe the general programming interface for the project.

Figure 4 shows the design of the state-machine for the Main module. Note that the reset state is the initial state of the application when not experiencing a fault. Shown in the state-machine diagram is the slider event that is triggered by the user to either start or stop some event using the CapSense hardware. Profile selected events are triggered by the user when a CapSense button is pressed at the appropriate time. Finally, time transition events are used to allow transition states to occupy a specified unit of time instead of being instantaneous. Note that the watchdog will expire due to fault detection circuitry activating a fault event. When this occurs, the motors will be ramped down from current value to a steady and safe condition. Then the program will enter a lockout state where the user cannot use the product until a hard system reset.

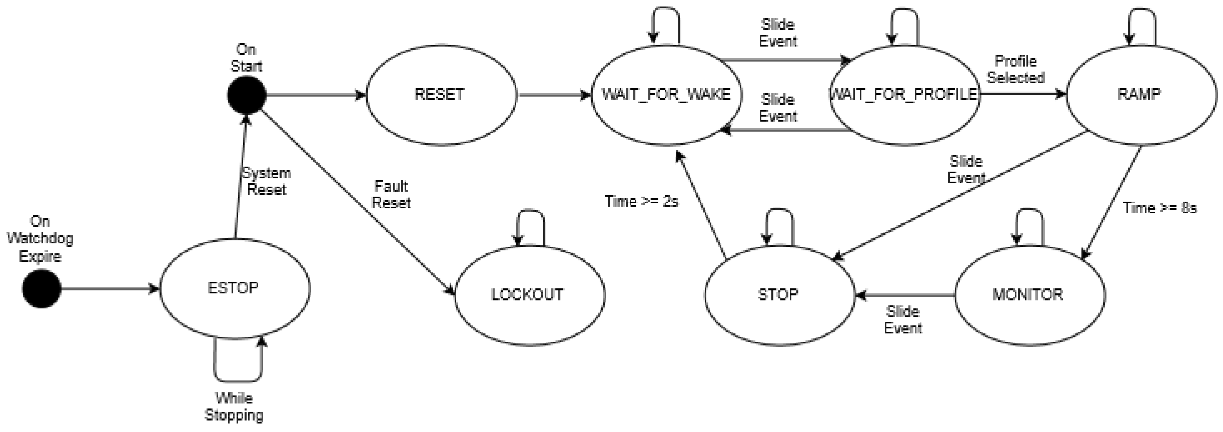


Figure 4: State-machine diagram for the software application. Note that unlabeled paths are default cases when no other transition conditions are met.

The application is implemented using the general software framework as described above. During the development process, as documented in the milestones, all of the modules will be completely fleshed out and any new considerations will be appropriately propagated through the design.

Milestones

This section of the report focuses on the design and implementation of the system described above. Each milestone outlines its objectives and then details the design and implementation to achieve them. A purpose of breaking the design explanation into milestones is to illustrate the time breakdown of the project. Note that for the first milestone, the general system design is skipped due to it being covered above.

Milestone I

Objective

Milestone one attempts to structure the development of the project by implementing the high-level functional interface for all software modules as explained in the system design. Additionally milestone one will start the state-machine implementation for application functionality. Specific implementation goals for this milestone include:

- Full implementation of startup event with music generation and LCD text.

- Full implementation of wait state behavior using CapSense user input.
- Implementation of profile selection, with profiles:
    - Casual
    - Moderate
    - Intense
    - Extreme
- Stubbed out states for operation of exercise motor and heartbeat detection.

As this milestone is designing the high-level interface, future milestones will take a more implementation only approach. However since the general design has already been described in the above sections it is only noted here.

Design and Implementation

During this milestone the UML diagram and state-machine diagram was generated, as shown in general description. As noted in the objective section, this milestone started the project thereby causing more organizational work than other milestone. Implementation was broken down to four categories: hardware interface modules, startup event implementation, profile selection implementation, and profile operation implementation. Hardware interface modules implemented included user interface, display, and music. The state-machine was implemented in main with a static inline function associated to each state.

Hardware Implementation

The hardware design was structured around the goal of fully using the PSOC's programmable hardware as opposed to solving the design challenge in software. Take the music module that is designed to play a tune through a piezo buzzer.

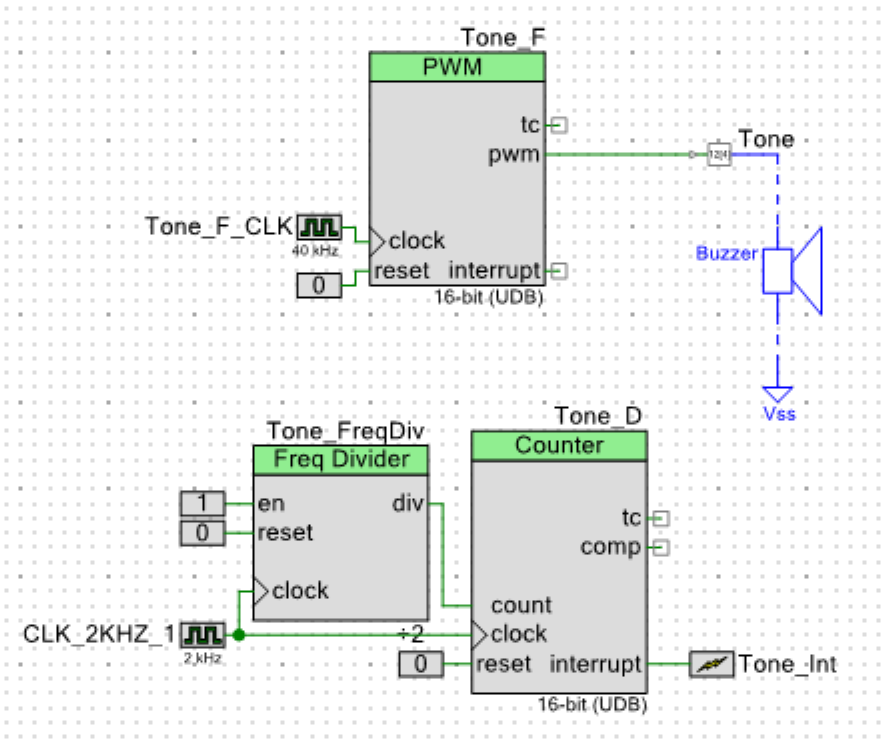


Figure 5: Hardware design for the music module. Shown is the PWM device that produces a square wave with variable frequency. Also shown is the counter that times the duration to play a note. Not shown is the 2kHz clock attached to the counter circuitry to provide millisecond timing.

This modules hardware design in shown in Figure 5 where a PWM module generates a 50% duty cycle at variable frequency. Additionally a counter is used to track the time that a note is being played that on compare match causes an interrupt. The software side of the music module has an initializer to start all hardware along with a play music function that takes an array of note objects that describe frequency and duration to play. On each, interrupt of the counter an index variable in incremented until the end of the tune see Code List 1. Note in the ISR a conditional branch exists to stop playing at end of tune or to take a note from the array as described above. Thus, the music module can play a tune while the processor is performing some other task.

```
CY_ISR(music_isr){
    static uint16_t freq = 0;
    static uint16_t per = 0;
    static uint16_t cmp = 0;
    static uint16_t dur  = 0;

    //play next note
    Tone_D_Stop();
    if(_current_buffer != NULL){
        if(_current_buffer_index >= _current_buffer_length){
            //end of tune
            _current_buffer = NULL;
            Tone_F_WriteCompare(0);
            Tone_D_Stop();
        }else{
            //play for current frequency and duration
            freq = _current_buffer[_current_buffer_index].frequency;
            per = TONE_F_CLK_FREQ_IN / ((freq > 0)? freq : 1);
            cmp = (freq > 0) ? per / 2 : 0;
            dur = _current_buffer[_current_buffer_index].duration;
            _current_buffer_index++;

            Tone_F_WritePeriod(per);
            Tone_F_WriteCompare(cmp);
            Tone_D_WritePeriod(dur);
            Tone_D_WriteCounter(0);
            Tone_D_Start();
        }
    }
    Tone_D_ReadStatusRegister();
    Tone_Int_ClearPending();
}
```

Code List 1: ISR that drives the asynchronous tone generation.

The display module hardware is composed solely of the LCD implementation provided by Cypress. Software was implemented using the design approach of encapsulation in order to reduce hardware to software coupling. Specifically, all references to the underlying hardware module must be done through a wrapper function. This may at first appear inefficient, however direct function calls are only renamed through a macro whereas complex functions like display_profile(), which provide an easy way to generate a menu, are implemented using standard function syntax. Another function example is the display_welcome() which handles the startup event textual logic.

Similar to the display module, the user input hardware is only dependent on the CapSense device provided by Cypress. CapSense provides a low-level mechanism to read both capacitive touch buttons and sliders that pressed by the user to drive the application state machine. The capacitive touch slider proved to be a challenge during implementation. Reason being that the CapSense needs time to reach a baseline before initialization and there exists a lag in data change that can span hundreds of milliseconds after the user makes a gesture. The current solution to the problem consists of a function that must be polled for state change on the slider. This function will return an enumeration with states: OFF, LEFT, RIGHT, NO_MATCH, and NOT_DETERMINED. It is then up to the state-machine logic to determine when to accept an input event from the user. Additionally the function will only trigger a LEFT or RIGHT value if the user swipes with the correct pattern.

Startup Event Implementation

On application power-on or after a reset, the system will display a loading screen where the first line of the LCD will fill up with black rectangles. Figure 6 captures the application in progress of loading. After the loading screen is complete, the display will render the welcome message to the user as shown in Figure 7. During the entire event, the system will play a startup tune on the piezo buzzer. When the tune is finished, the state machine will transition into a wait for user wakeup event. Startup is the first state of the state-machine that once transitioned out of it cannot be reentered until system reset.

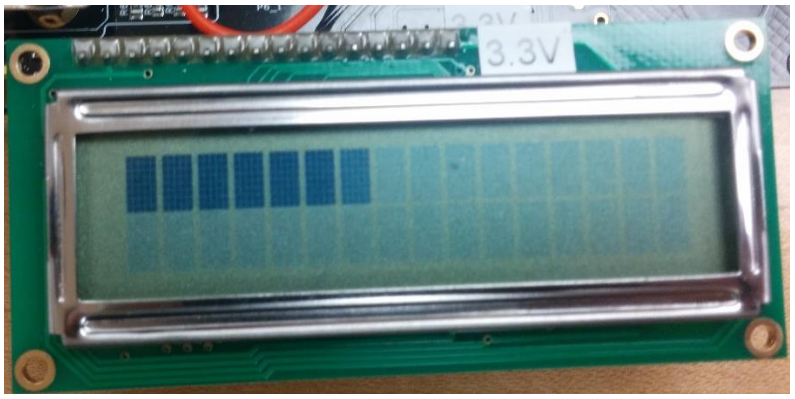


Figure 6: Loading screen for the startup event. Black rectangles fill the top row through time while the startup tune is played

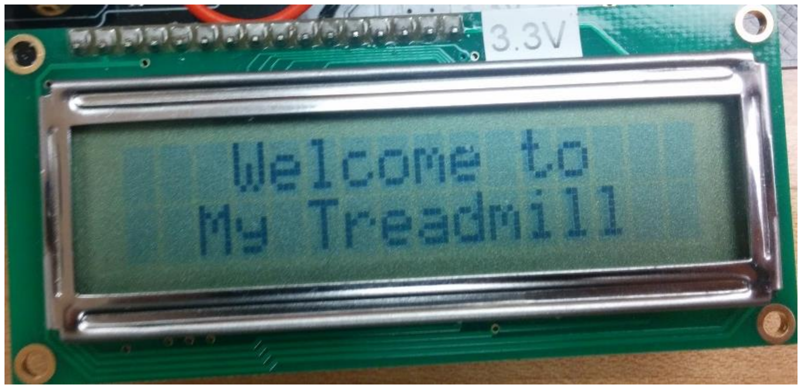


Figure 7: Welcome message displayed to user after the loading screen in completed. This screen persists until the startup tune finishes playing. Then the screen is cleared until user wakes up device.

Profile Selection Implementation

Profile selection is the state that is entered after the user wakes up the system. This state can be exited by sliding the CapSense slider in the opposite direction of the wake event. A CapSense button is used to cycle the profile options and another is used to select the profile. An explanatory message is shown as the first menu slide, see Figure 8. The first option is the causal profile, which is designed to simulate a walking experience, Figure 9. The second option is the moderate profile, which is designed to slightly elevate heartrate for fitness but not to push oneself, Figure 10. The third option is the intense profile, which will force the user to run at a fast pace, Figure 11. Finally, a tongue-in-cheek extreme profile will test the user's limits, Figure 12. After selection of any of the modes, that profile's name will flash four times to notify the user of their choice. Then the state machine will transition to the ramp state.

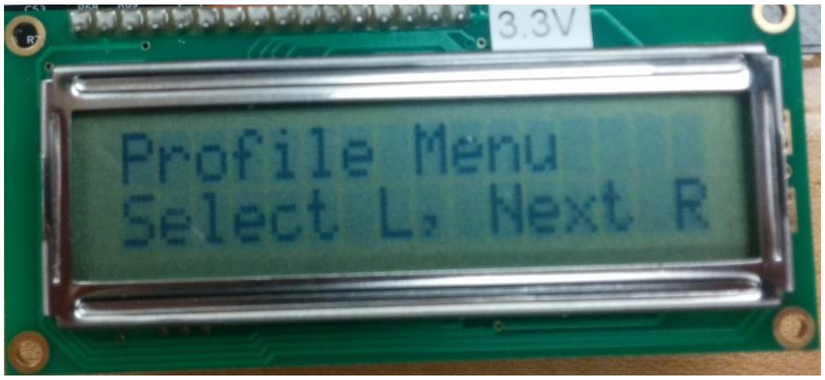


Figure 8: Profile menu selection prompt to guide user to select a profile using the CapSense buttons. This page is considered part of the profile page list for display, so looping through will cause this message to be rendered again. However, attempting to select the information page will not change state of the program.

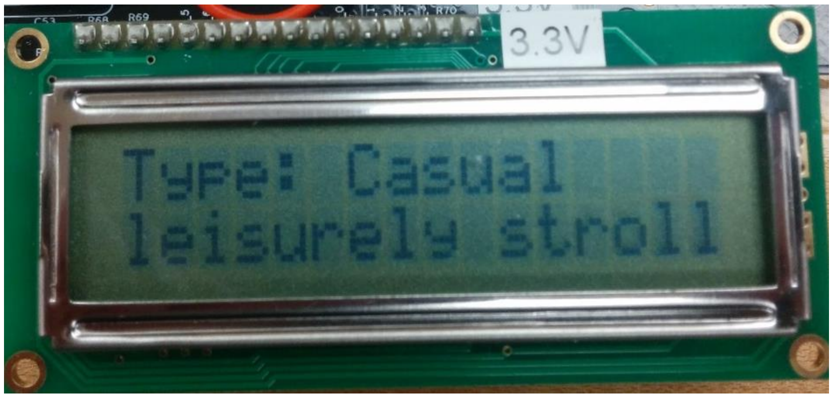


Figure 9: Causal workout profile option for those more interested in light fitness such as taking a walk.

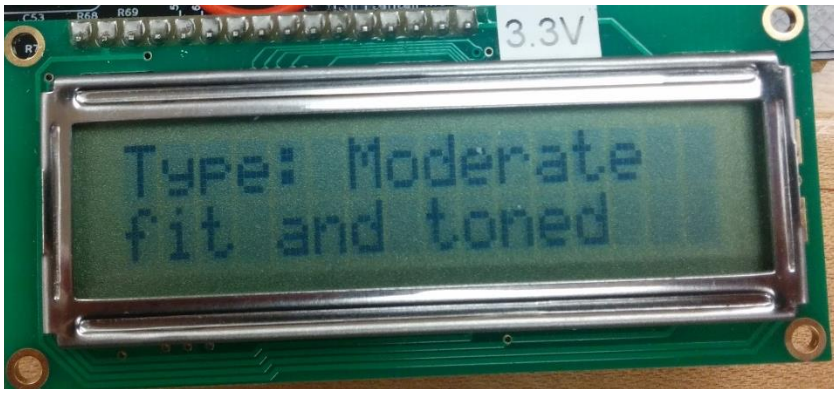


Figure 10 Moderate workout profile option for those who are interested in a jog or light run.

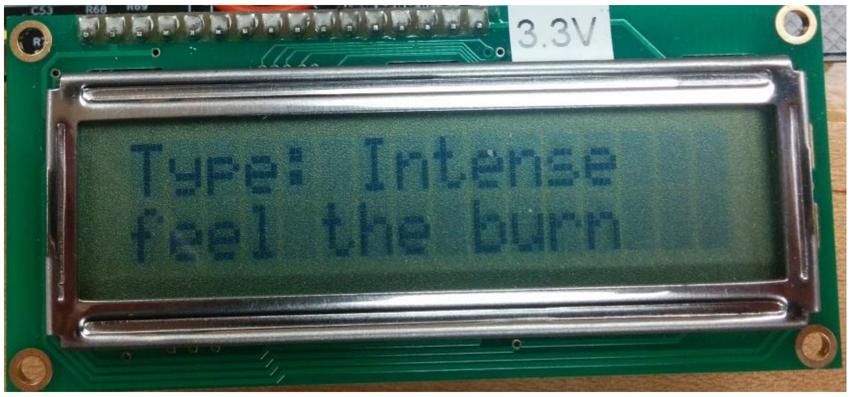


Figure 11: Intense workout profile option for those interested in cardio and endurance.

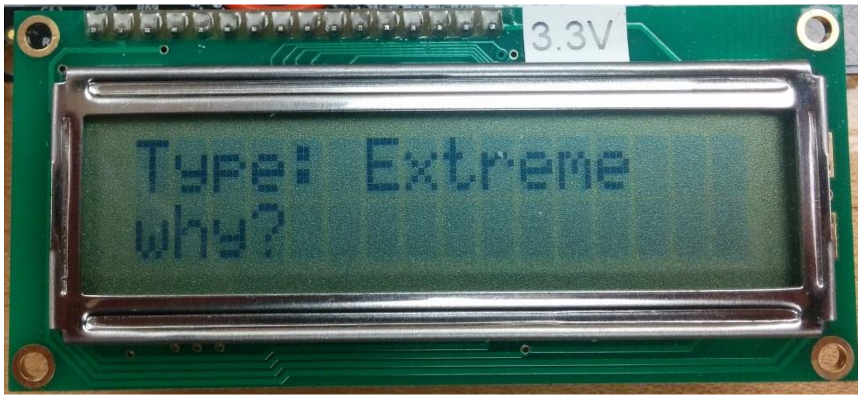


Figure 12: Limit testing profile option to train at a professional level.

Profile Operation Implementation

Profile operation implementation encompasses the ramp, monitor, and stop states. For this milestone, the functionality was a facade with textual display but no logical substance behind it. The goal then was to implement the transition logic between states. Starting at the ramp state, the function would transition to the monitor state once the PWM reaches the profile baseline. Figure 13 shows the displayed message for the ramp state. Additionally, the user can cancel the profile causing a transition to the stop state. The monitor state would loop until the user requests a shutdown event or the heartbeat signal returns an invalid value. Figure 14 shows the message for the monitor state. Note that the heartbeat value was not shown during this phase, rather it was left blank. The stop state waits until the PWM ramps down from current value to zero before transitioning to the user wait for wakeup state. Figure 15 shows the message displayed during the stop state.

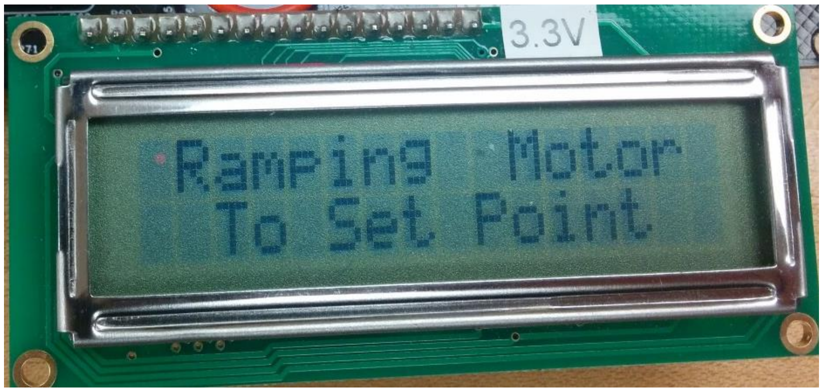


Figure 13: Message to user while exercise motor is brought to profile base level.

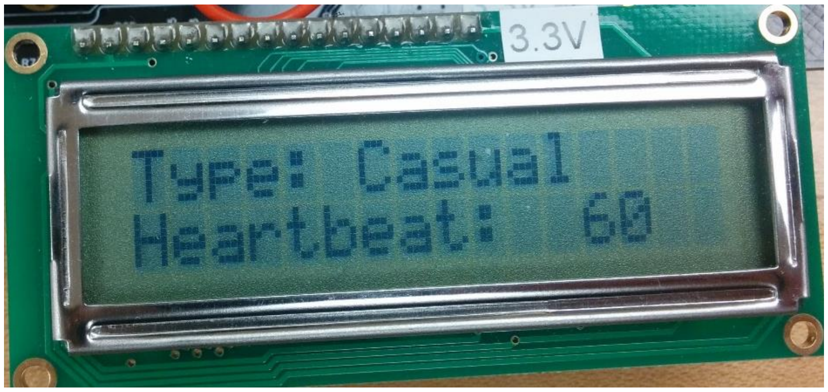


Figure 14: Message to user while in monitor state. Note that the heartbeat shown did not appear in the initial milestone version. However, no image exists from this milestone so this image of final implementation is used as approximation.

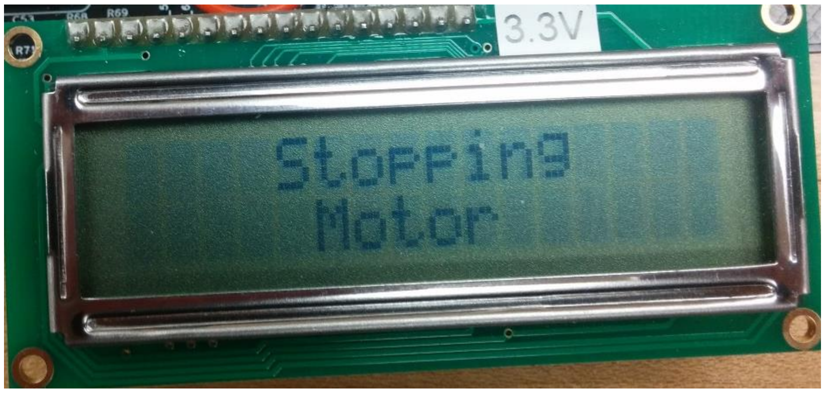


Figure 15: Message to user while motor is ramped down from current speed to off state.

Milestone II

Objective

The focus of milestone two is to develop the heartbeat detection module. This module will encompass analog signal conditioning of the input waveform, heartbeat detection, and heartbeat calculation.

Additionally, the hardware structure for milestone three starts with the waveform acquisition being accomplished by an ADC.

Note that the ideal input waveform has the voltage range of 0 mV to 100 mV.

Design and Implementation

Design philosophy for this milestone follows the pattern of utilizing the hardware as much as possible along with focusing heavily on software interrupts to drive processor interaction. Figure 16 shows the final hardware design that attempts to fulfill the requirements and to follow the design philosophy. The first important feature of the design is the analog MUX that allows the program to select either an internally generated heartbeat signal or an external signal. The goal of this feature will allow the user to test the application without the need for an external signal.

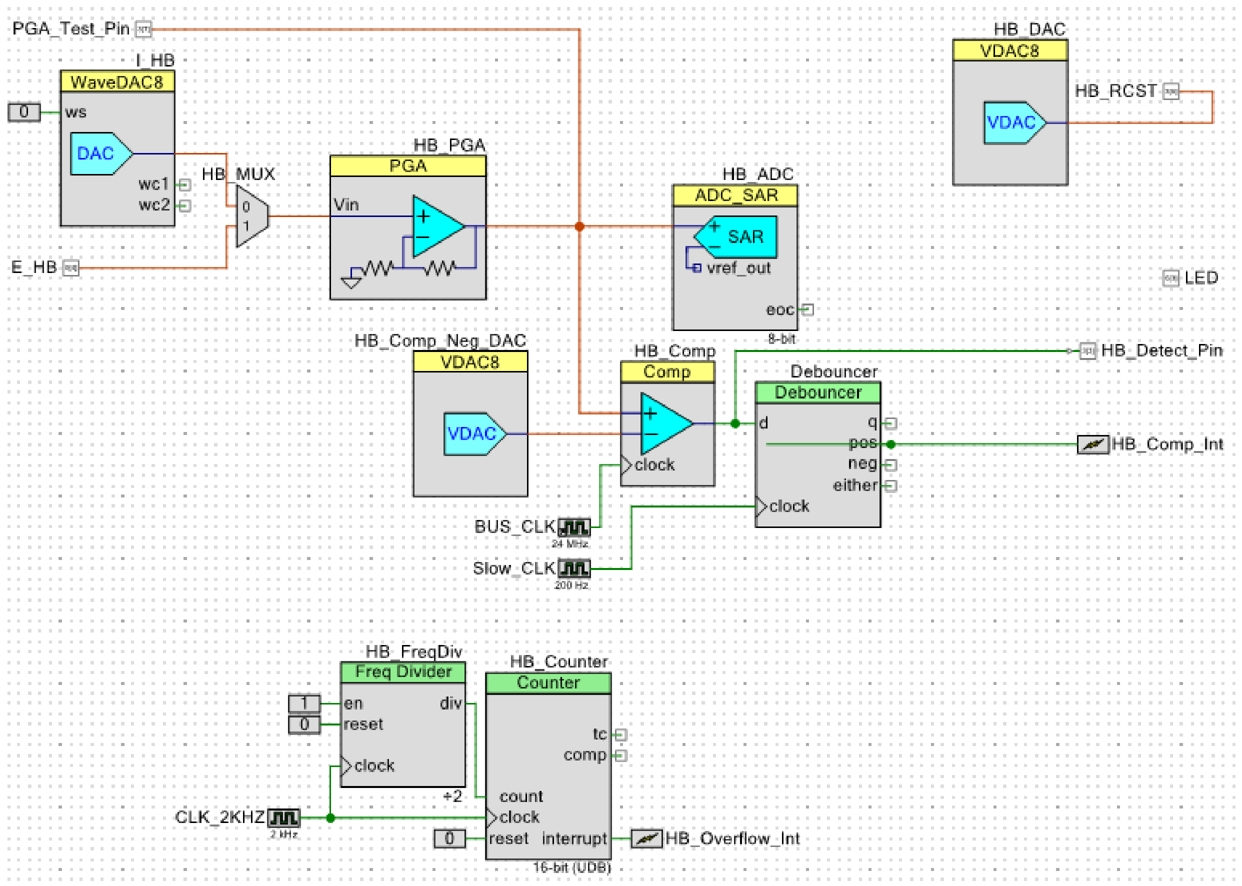


Figure 16: Hardware design for heartbeat detection and waveform acquisition module.

Next important feature is the programmable gain amplifier, PGA, using a gain of twenty-four. This will scale the input voltage from a peak of 100 mV to 2.4 V as verified in Figure 17. A test pin is supplied to verify the correct scale on the amplified signal as shown in the hardware design. Attached to the amplified signal is the ADC, which will be used in milestone three to acquire the waveform for recreation. Additionally, a voltage comparator is attached to the amplified signal to detect when the heartbeat has risen above a calibrated voltage threshold generated by the voltage DAC attached for comparative reference.

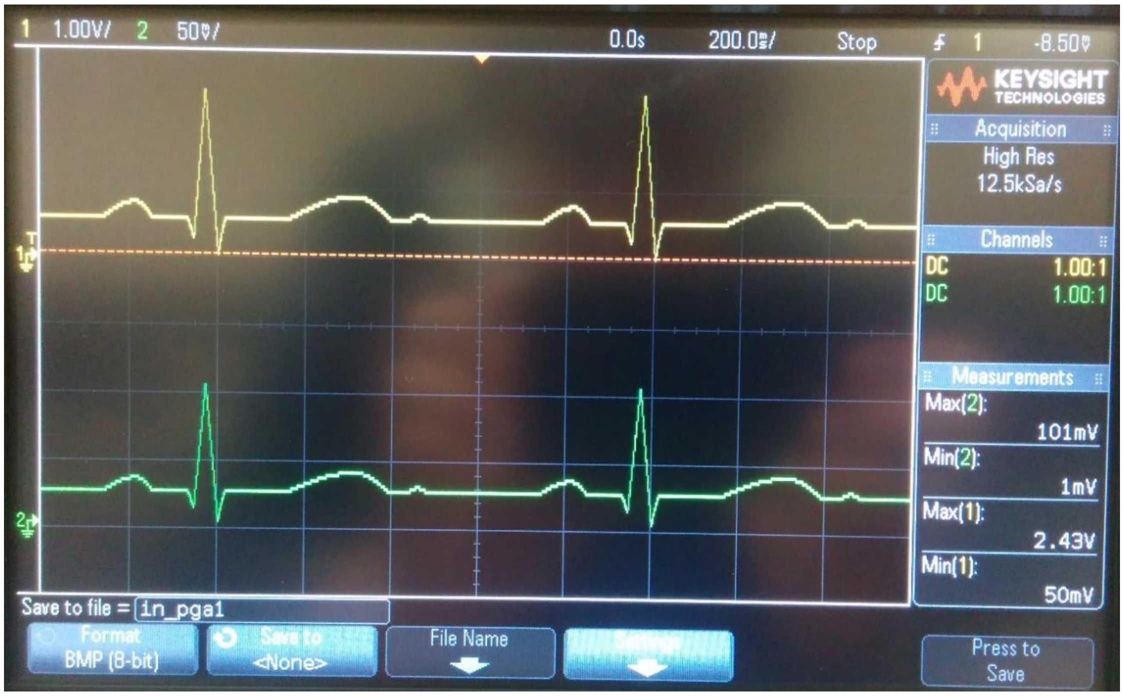


Figure 17: Amplification verification image. Lower waveform is the input heartbeat with scale of 100mV and the upper waveform is result after amplification with a result of 2.43V or approximately a gain of 24 as designed.

Verification of waveform detection is shown in Figure 18 with the upper waveform being read from the heartbeat detect test pin. Note that the comparator result waveform is noisy which leads to a hardware debouncer circuit being attached to the comparator result to drive the heartbeat detection interrupt.

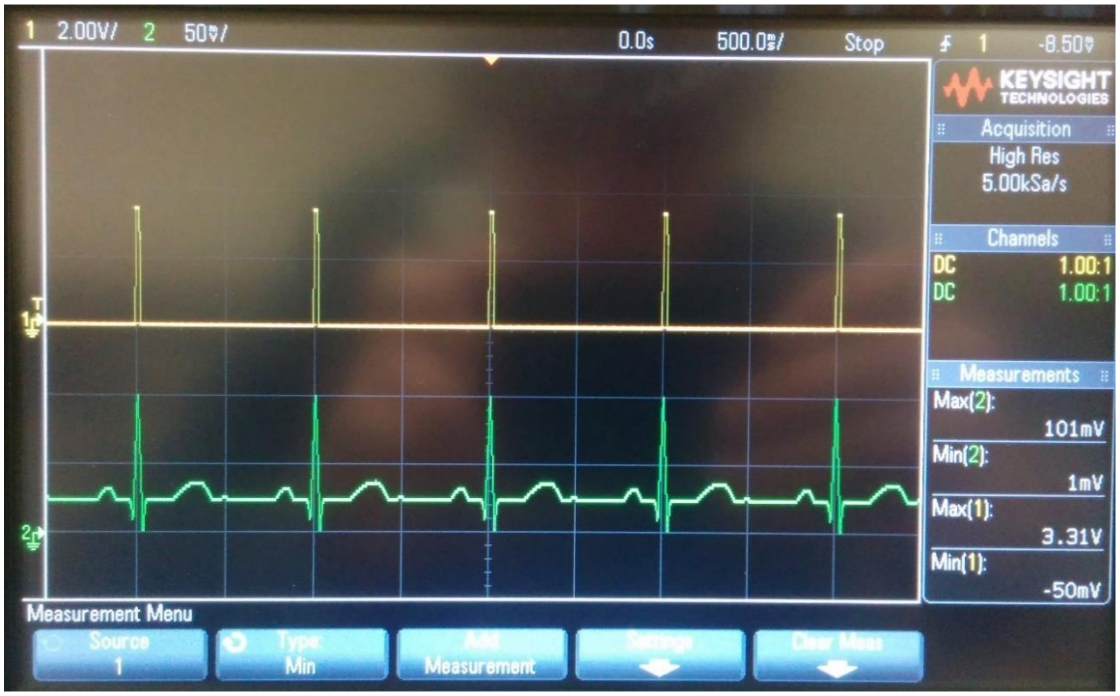


Figure 18: Heartbeat detection comparator waveform where the upper waveform is result from comparator due to the input waveform supplied. Note that the resulting waveform has not been ran through a debouncer for this image. The interrupt is attached to a debounced form of this waveform.

Finally, note that the counter component in the design that is used to track the duration between pulses, thereby allowing for frequency calculation. The counter is dependent on a 2 kHz clock to allow for millisecond level precision. As a side note, the LED shown in the hardware design is toggled on each pulse of the heartbeat for user indication of heartbeat detection. Yields a visible pulse of one half the heartbeat measured.

Software implementation for this milestone is focused entirely on the heartbeat module. As mentioned, the design philosophy for the low-level software is to make most of the control logic dependent on interrupts. For the heartbeat module, the relevant interrupts are comparator detection, counter overflow, and ADC sample complete. Comparator detection interrupt is used as the periodic update for the software module. Code Listing 2 shows the ISR implementation for the comparator interrupt. Generally, the function gets the current counter value, calculates the BPM, and updates the running average data structure to filter noise in the heartbeat signal. After which the function clears the source of the interrupts and restarts the counter. Note that the running average data structure follows an object oriented design approach where the heartbeat module owns a data structure that must be used in the running average function parameters.

```
CY_ISR(comp_isr){
    static uint16_t count = 0;
    static float bpm = 0;

    //calculate the new heartrate in BPM
    count = HB_Counter_ReadCounter(); //count is ms between peaks
    bpm = 60000 / ((count > 0) ? (count) : (1)); // in BPM
    bpm = (bpm > MAX_POSSIBLE_HEARTBEAT) ? (0) : (bpm);
    running_average_update((Running_Average_t*)(&_filter),(uint16_t)(bpm));
    //clear source of interrupt
    HB_Counter_WriteCounter(0);
    HB_Comp_Int_ClearPending();

    //pulse LED to simulate heartbeat
    if(_use_led_state){
        LED_Write(!LED_Read());
    }
}
```

Code List 2: ISR for heartbeat detection interrupt. This ISR updates the running average filter to provide heartbeat to control logic in application.

The counter overflow interrupt is used to handle larger periods than can be represented in the 16-bit counter word size. Currently, the method selected to handle the overflow event is to clear the running average as it is assumed that the pulse signal is lost. By taking this approach one can use the lack of a signal to report zero BPM quicker than if a zero was only placed into the running average thereby reporting to the control algorithm quicker. ADC interrupt is only stubbed out during this milestone, however in future milestones this will be used for DAC recreation data.

Milestone III

Objective

Design and implement the heartbeat waveform acquisition and DAC recreation for the heartbeat module. Implementation will use the ADC from the hardware design and the end of conversion interrupt stubbed out in milestone two. Additionally there is a functional requirement not to perform any software based scaling of the ADC to DAC value is proposed to further show the ability of the PSOC hardware.

Design and Implementation

Software implementation for the ADC to DAC connection is an ISR that is triggered on the end of an ADC conversion. This ISR copies an 8-bit unsigned value from the ADC data register and writes it to the DAC data register. The resulting waveform is shown in Figure 19, where the upper signal is the recreated waveform and the lower signal is the external heartbeat signal inputted into the system. A comparison between the recreated waveform and the waveform presented to the ADC is shown in Figure 20. Note that the main differences between the input and output signal are due to a voltage shift and scaling factor. Both of these differences are artifacts of the sample and recreation resolutions. Specifically, the ADC has a configurable but limited range of sampling resolutions, dependent on reference voltage. The DAC has significantly less options where the one selected ranges from 0 V to 4 V.

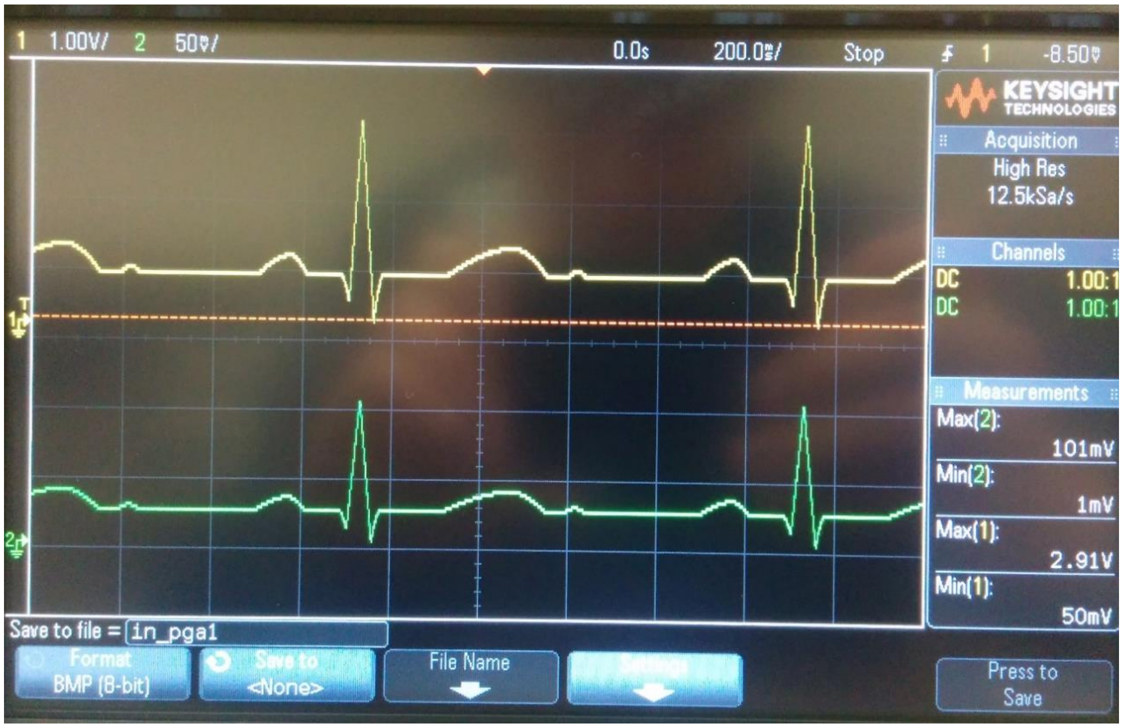


Figure 19: DAC waveform recreation shown as top signal. Bottom signal is the original external heartbeat pulse used to test system. There exists a shift of voltage due to sampling and conditioning artifacts.

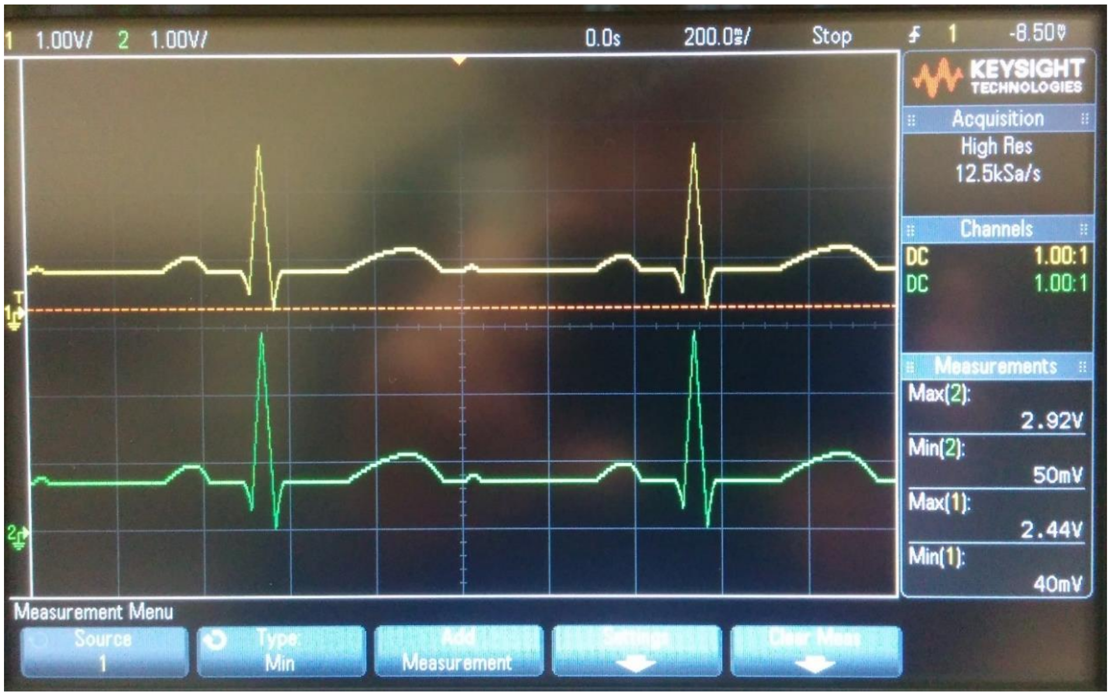


Figure 20: Comparison between ADC input waveform, bottom, and DAC recreation, top. Note the difference in scale and offset is due to difference in ADC and DAC resolutions.

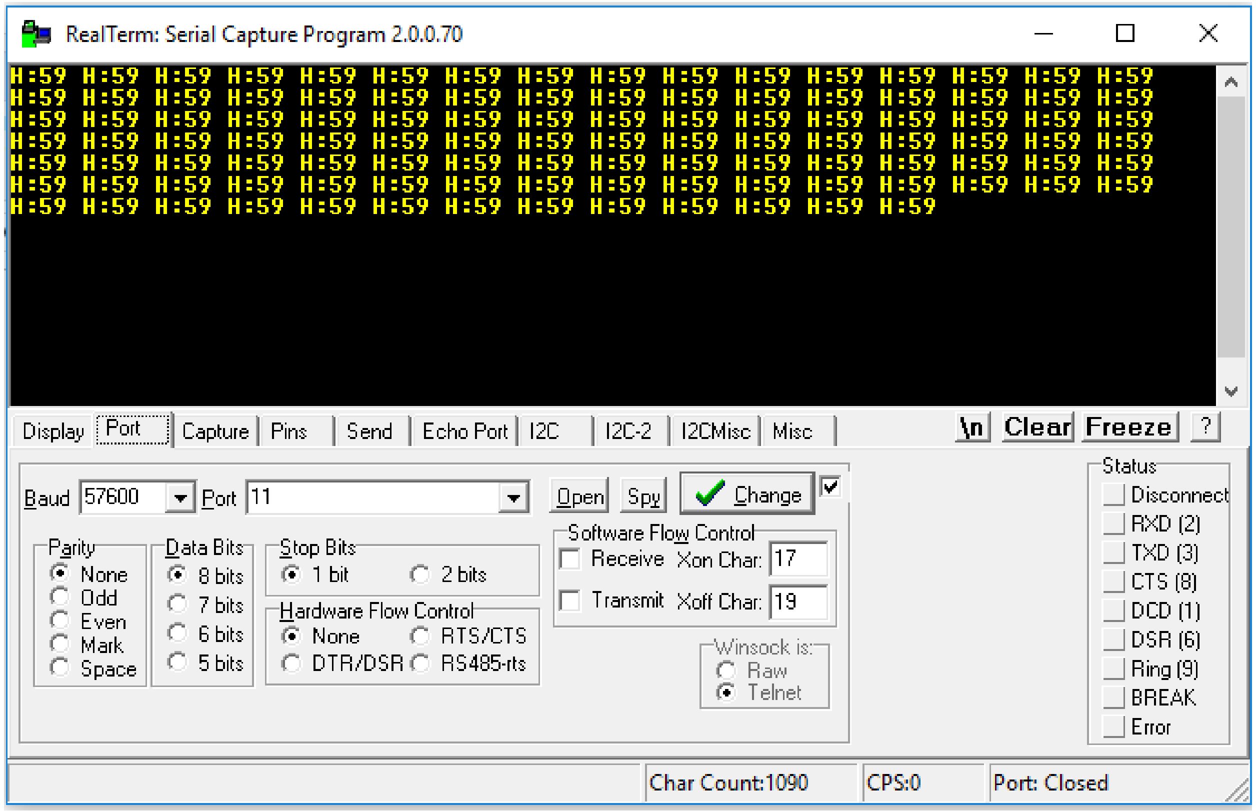
Milestone IV

Objective

For this milestone, a data stream is to be sent over the USB USART interface to a connected PC. The data will consist of the current heartbeat detected. Data should only be sent during the monitor state. Finally, the device should separate each value with white space to make stream-parsing simple on the PC side. Note that only USB USART write behavior is needed for the application, however simple reading is implemented to account for future use case.

Design and Implementation

The UART software module is designed as a wrapper of the USB USART hardware interface to provide simplified read and write behavior. This module is used only during the monitor state, specifically called right after the running average heartbeat value is calculated. Figure 21 shows a serial terminal, RealTerm, on the PC connected to the PSOC system. Shown is the current heartbeat shown as an integer that is rising as time increases. Note that the requirement of easy parsing in accomplished by placing a space between each key value pair. Where a key value pair is a letter followed by a colon finally followed by a number. While not as robust as stringified JSON, this solution does not presume the receivers' application being forced to handle the complexity of JSON parsing. Additionally, in a language like C the key-value solution allows for the use of built-in functions, scanf(), as opposed to third party libraries.



```
H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59
H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59
H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59
H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59
H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59
H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59
H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59 H:59
```

Baud 57600 Port 11

Parity: [✓ None] Odd Even Mark Space
Data Bits: [✓ 8 bits] 7 bits 6 bits 5 bits
Stop Bits: [✓ 1 bit] 2 bits
Hardware Flow Control: [✓ None] RTS/CTS DTR/DSR RS485-rts
Software Flow Control: Receive Xon Char: 17 Transmit Xoff Char: 19
Winsock is: Raw [✓ Telnet]

Status: Disconnect RXD (2) TXD (3) CTS (8) DCD (1) DSR (6) Ring (9) BREAK Error

Char Count:1090 CPS:0 Port: Closed

Figure 21: Serial terminal displaying the heartbeat stream.

Milestone V

Objective

Implement the motor control algorithm for the ramp state, monitor state, and stop state. In ramp state, the motor PWM will increase to profile base PWM over eight seconds. In monitor state, the motor PWM will be adjusted every 10% increase or decrease in heartbeat by the specified profile's defined step value with respect to the base PWM. For the stop state, the speed will be reduced from current PWM to zero to shut off the motor.

Design and Implementation

Implementation of the PWM control is done within the state-machine due to the tightly coupled nature of the motor control and the application state. All PWM information related to a specific profile is stored as a member of each profile singleton. The main application stores a pointer reference to the current profile or NULL if none is selected. When a profile is running, its properties are used to generalize the algorithm, as opposed to a switch code block. The following figures: Figure 22, Figure 23, Figure 24, Figure 25 and Figure 26 show the PWM output at 12.5%, 25%, 50%, 75%, and 100% positive duty cycles respectively. Note that a PWM is calculated every iteration of ramp, monitor, and stop states. Finally, the PWM does adjust in value as specified for each profile, however the demonstration is hard to convey on paper, thus the reason to show base line duty cycles for the profiles.

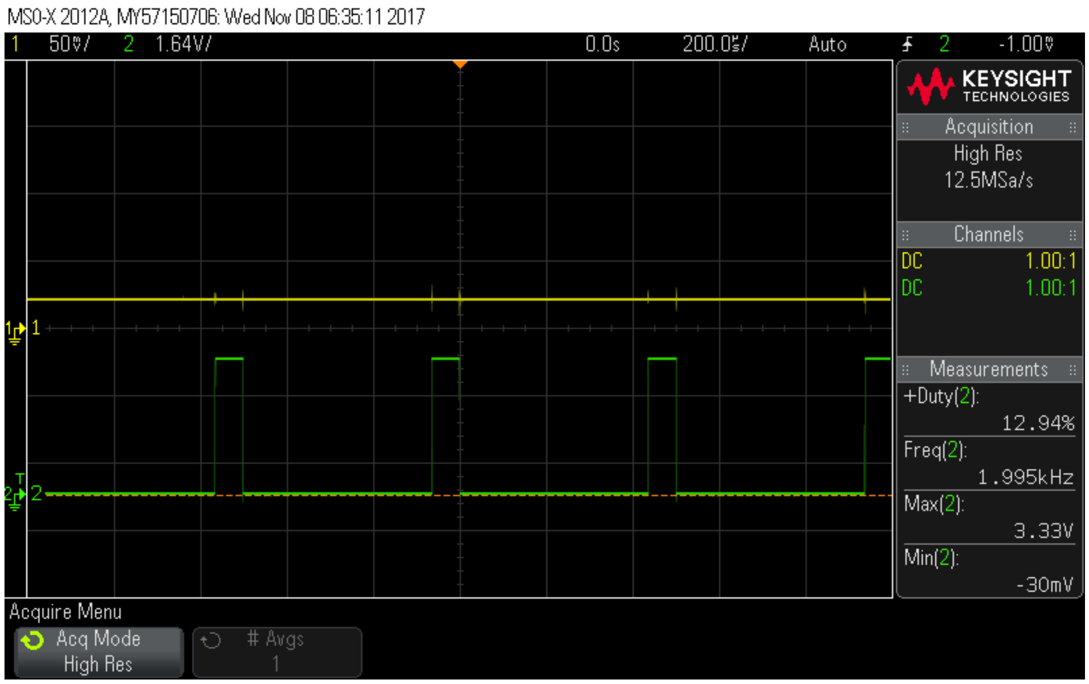


Figure 22: PWM output at 12.5%, which is the base duty cycle for the casual profile.

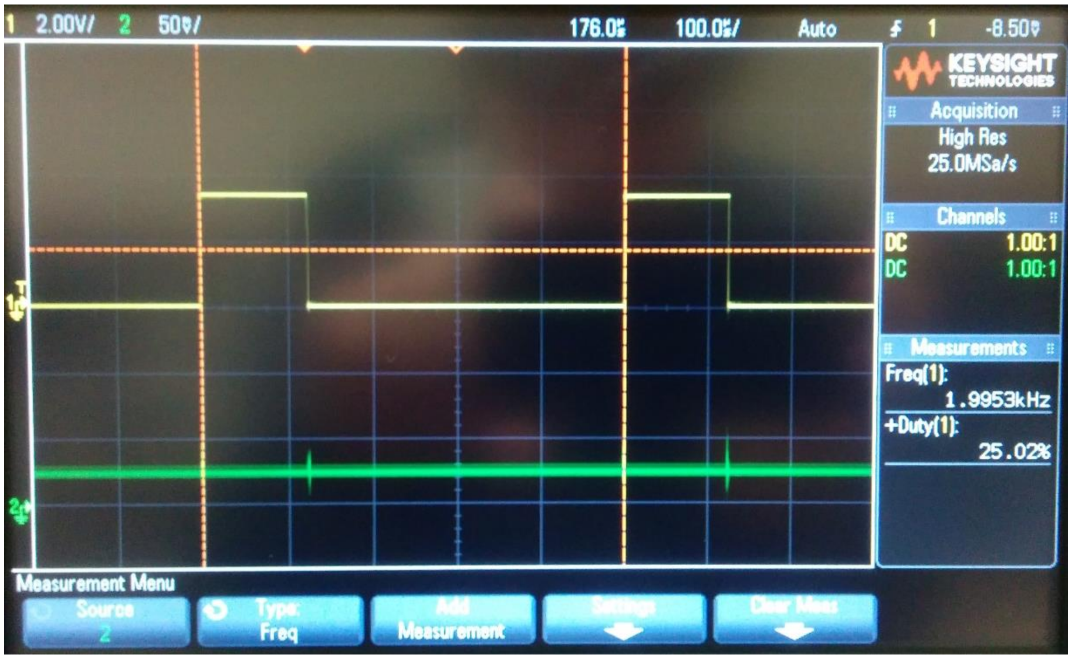


Figure 23 PWM output at 25%, which is the base duty cycle for the moderate profile.

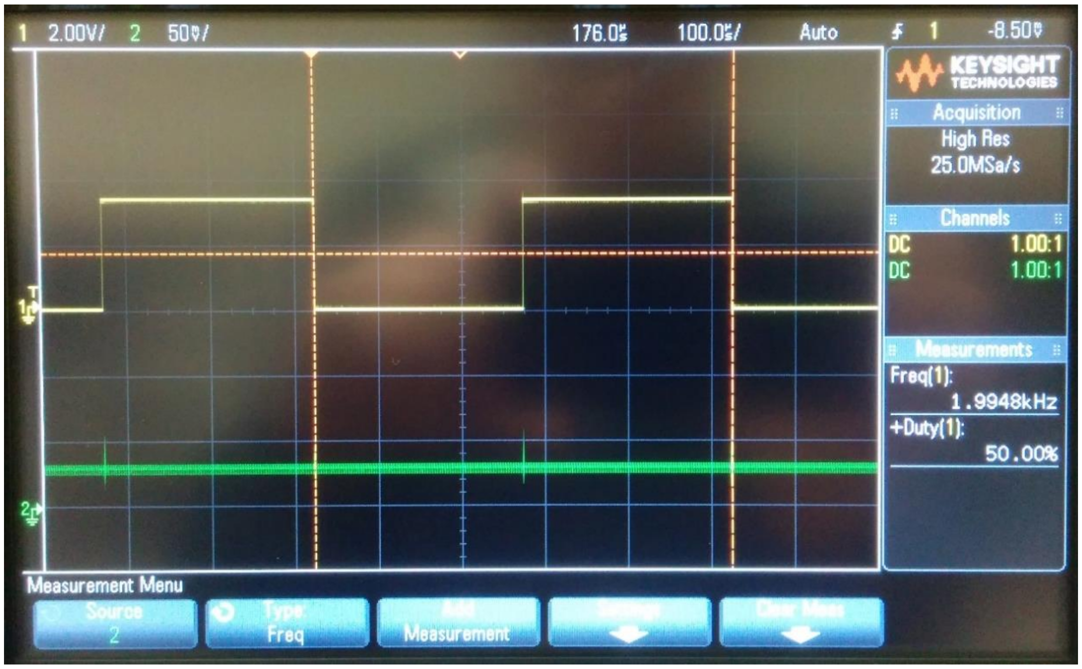


Figure 24: PWM output at 50%, which is the base duty cycle for the intense profile.

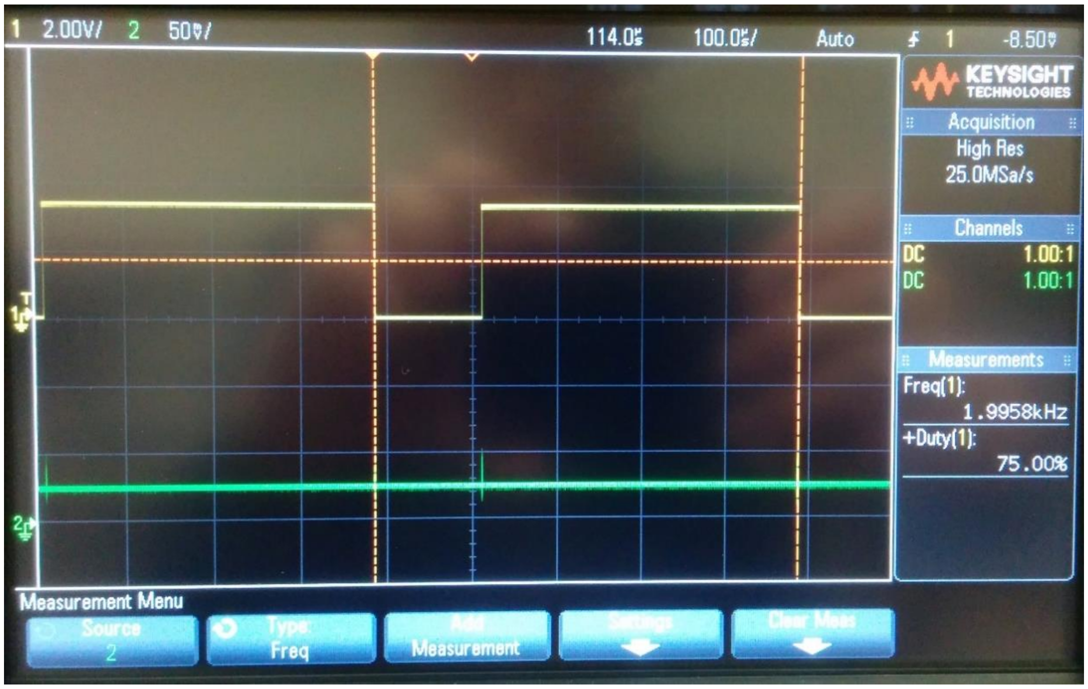


Figure 25: PWM output at 75%, which is the base duty cycle for the extreme profile.

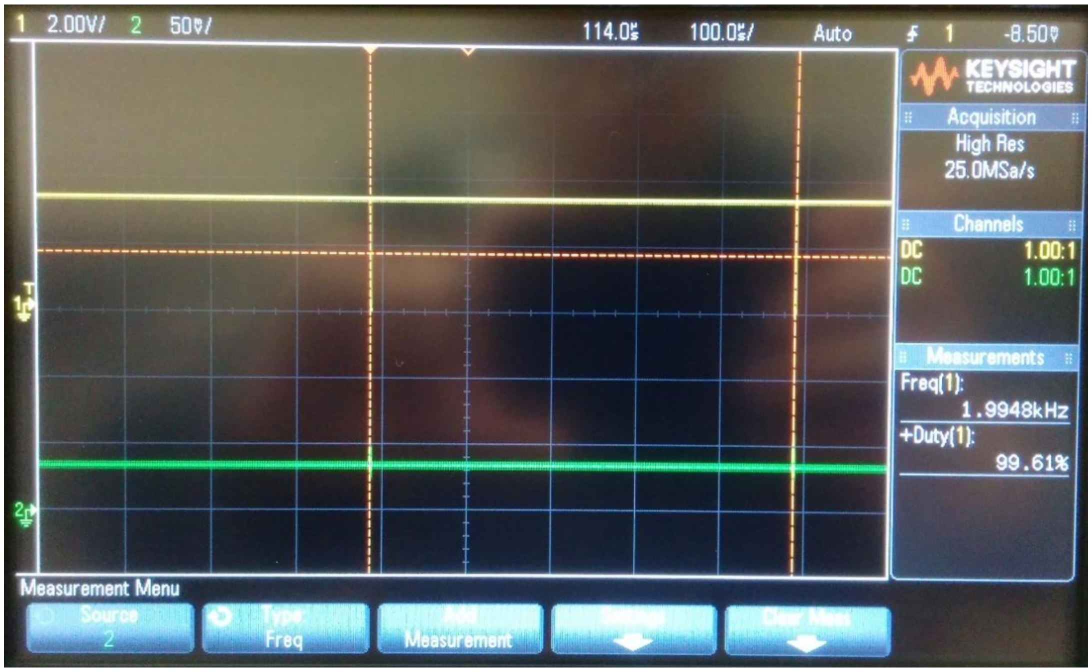


Figure 26: PWM output at approximately 100% positive duty cycle shows the upper end of the PWM range.

Milestone VI

Objective

The objective of this milestone is to integrate fault tolerance into the current treadmill project. The fault behavior expected is fail-safe so that the system will prevent harm from the user but is not expected nor desired to continue functioning. The software ought to periodically check the fault detection inputs from hardware, determine if a fault has occurred, and cause a fault restart. Additionally, to provide safety to the user, the system must ramp down the PWM from its current value so that an abrupt stop is avoided. Another safety feature is that the system will lockout the user after a fault has occurred so that they may not request an exercise profile.

Design and Implementation

During the development of this milestone, several previously finished features of the code base needed to be modified. First was the state machine which now requires a critical ESTOP state, to ramp down the motor, and a LOCKOUT state to prevent the user from attempting to use the treadmill. The ESTOP state is implemented as a critical section of code that runs before the watchdog is allowed to expire and is triggered only if the motor is running when a fault occurs. The LOCKOUT state will be entered on the reset condition only when watchdog flag is set. The display view for when the LOCKOUT state is active is shown in Figure 27. The second change made was less critical in system behavior as it affected the USB USART. Specifically the data was changed from just a number being sent to that of a key-value pair, the necessary changes were updated in the USB USART milestone. Finally, a general refactoring of the hardware and software design was performed so that the code is presentable as a final project.

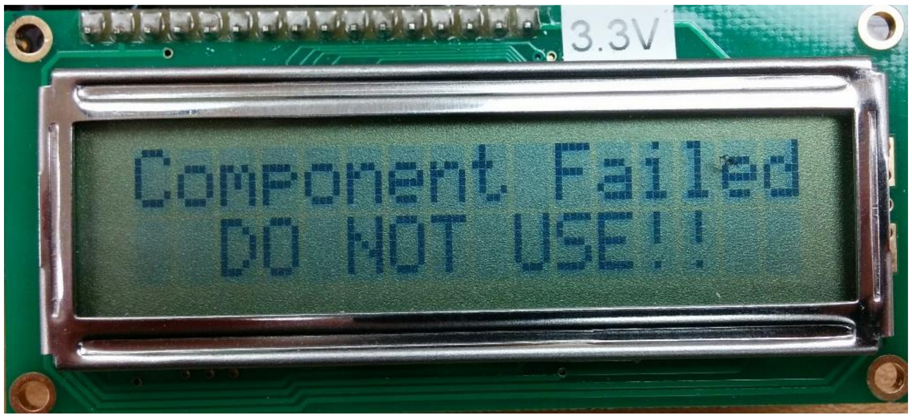


Figure 27: Lockout state display text to warn user of failure and risk to safety.

A fault detection module was implemented to keep with the modularity objective for the project. Hardware for this module consists of a timer to feed the watchdog, several component operating properly signals, and the watchdog. The timer used to track faults triggers an interrupt on overflow, which is connected to the fault detection ISR as shown Code Listing 3. The focus of this interrupt is to determine if an error occurred or the code is in a critical state and if so then feed the watchdog else let it expire so that the system restarts.

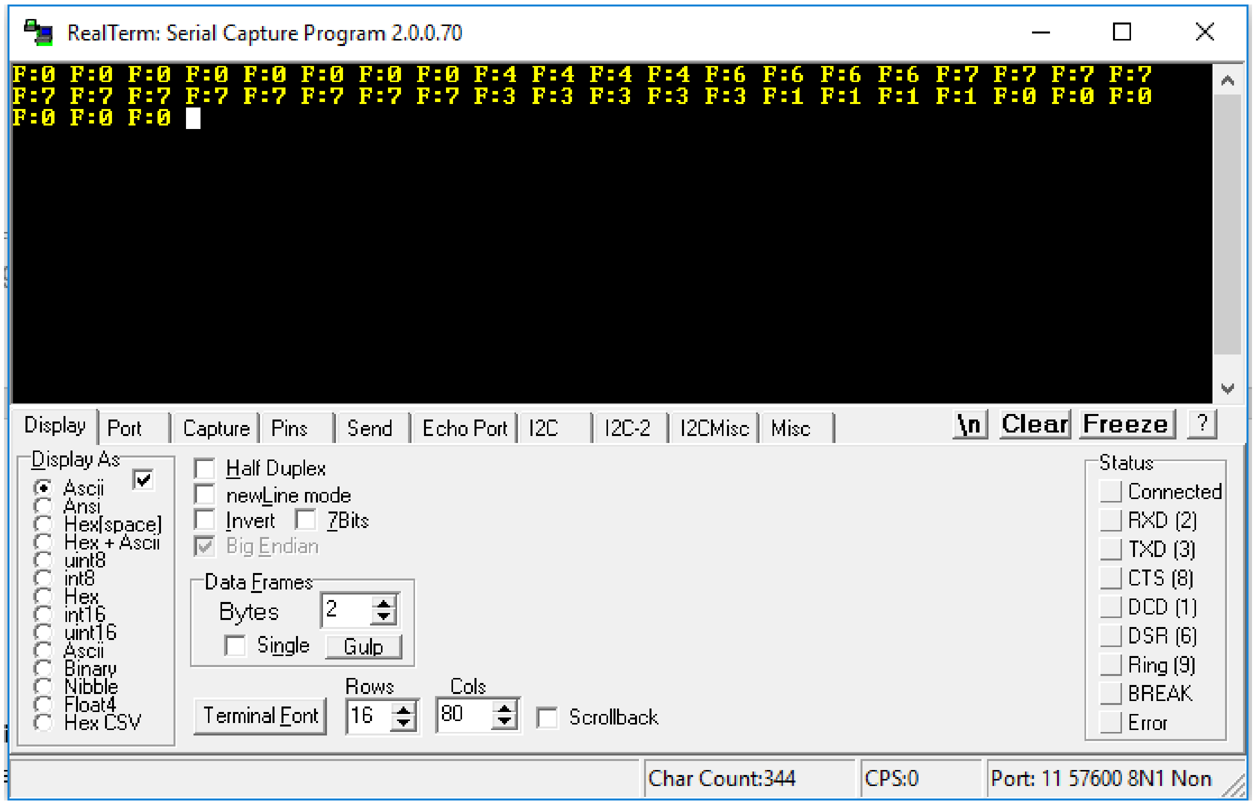
```
CY_ISR(fault_detection_isr){

    //feed watchdog only if no faults exist
    if(!fault_check_for_failure() || _critical_section){
        CyWdtClear();
    }

    //clear source of interrupt
    Watchdog_Timer_ReadStatusRegister();
    Watchdog_Timer_Overflow_Int_ClearPending();
}
```

Code List 3: ISR for the fault detection module. Feeds the watchdog only if there is not a fault or program is in a critical section.

A final requirement of fault detection is to send out an error code to the optionally connected PC. The USB USART code was used, and modified, from the previous project iteration where a key-value pair is sent out. The number attached to the result is formatted as a bit-field where presence of an active high bit signals that a fault occurred. The order from greatest to least significant bit is heartbeat fault, motor fault, and user input fault. Figure 28 shows a serial terminal that is receiving the data stream. Note that since the data is a bit field the fault code will range from 0 to 7 inclusively.



```
F:0 F:0 F:0 F:0 F:0 F:0 F:0 F:0 F:4 F:4 F:4 F:4 F:6 F:6 F:6 F:6 F:7 F:7 F:7 F:7
F:7 F:7 F:7 F:7 F:7 F:7 F:7 F:7 F:3 F:3 F:3 F:3 F:3 F:1 F:1 F:1 F:1 F:0 F:0 F:0
F:0 F:0 F:0
```

Char Count:344 | CPS:0 | Port: 11 57600 8N1 Non

Figure 28: RealTerm serial terminal displaying fault codes being generated during lockout state.

Conclusion

Given the nature of the heartrate compensated exercise machine project as a mixed-signal project, the PSOC 5 proved to be an excellent hardware platform specifically with internal, customizable analog circuitry to condition the input heartbeat waveform and with the extensive library of hardware devices to implement various devices, USB UARTs for example. The design process to develop the application proved to be a good refresher on the full embedded systems development model. In addition, being forced to develop on a new processor as opposed to the more familiar NIOS soft-core processor allowed for growth of development skill. Thus, the project achieved the learning objective. In regards to the products result, the system responds well to user input through the capacitive touch buttons and slider.

Additionally the LCD shows relevant information without unnecessary flashing. The system also demonstrates all requirements to control the motor through an adjustable PWM. Finally, the addition of fault tolerance as an extension to the project illustrated the considerations to make a reliable safe system. Thereby integrating the theory received in lecture with the current code base. Hence, the system has met its design objective. In entirety, project is regarded as a success for the embedded IV course.

Appendix

Test Setup

The following image depicts the physical system as under test. Note that the signals from the PSOC development board were not tested with a physical motor nor a physical heartbeat sensor. Rather the input signal was generated from a function generator and the outputs observed through an oscilloscope. Note that the equipment used to test are the InfiniiVision MSOX2012A mixed signal Oscilloscope and the Agilent 33220A 20 MHz function / arbitrary waveform generator.

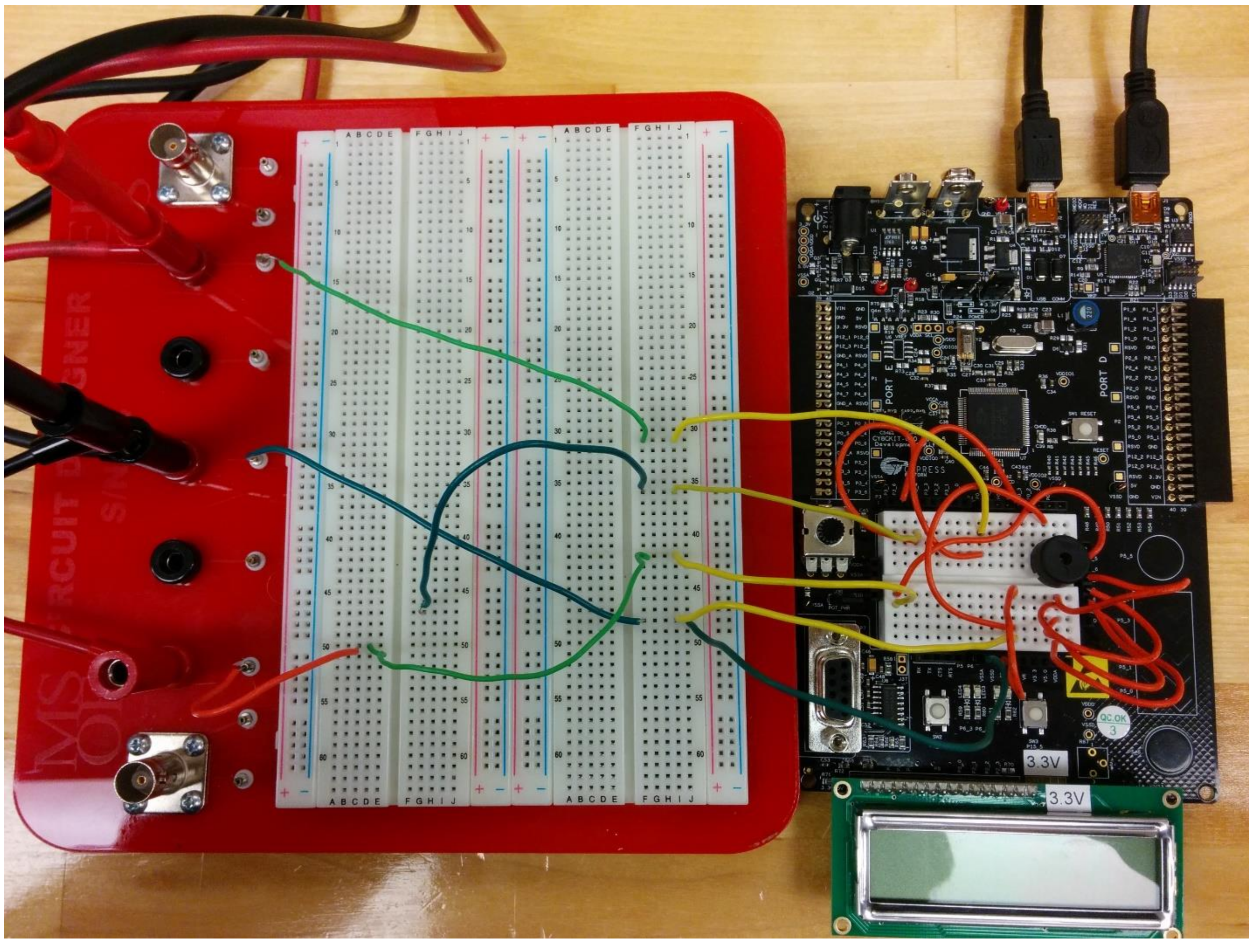


Figure 29: System under test. Shown on right is the PSOC 5 development board with the user input in lower half. On the right is a breadboard to breakout the input and output signals for testing and verification using laboratory equipment.

Code Listing

Main

```
/**
 * Main
 * @author Curt Henrichs
 * @date 9-13-17
 *
 * Heartbeat compenstated treadmill, fitness application. Application uses a
 * state machine to reset program variables, wate for user input, motor
 * rampup, heartbeat monitor plus motor compensation, and motor stop state.
 *
 * Also, fault detection software is enabled to read COP signals from critical
 * components. Under case of failure, the system will transition to an ESTOP
 * state to gracefully power down the motor before performing a system reset.
 * Once reset is complete, the system will transition to LOCKOUT state until
 * user power-cycles device. Process occurs if again if fault was not transient
 */

//==============================================================================
//                                 Library
//==============================================================================

#include <project.h>
#include <stdio.h>
#include <stdbool.h>

#include "Display.h"
#include "Heartbeat.h"
#include "Profile.h"
#include "Music.h"
#include "Uart.h"
#include "UserIn.h"
#include "MotorPWM.h"
#include "Fault.h"

//==============================================================================
//                        Constant and Macro Definitions
//==============================================================================

/**
 * Application state machine, state enumeration
 */
typedef enum States_e {
    RESET,                 //! Initalization state of application, welcome user
    WAIT_FOR_WAKE,         //! Waits for user to wakeup treadmill
    WAIT_FOR_PROFILE,      //! Waits for user to select a profile, drives menu
    RAMPUP,                //! Ramps the motor to profile setting
    MONITOR,               //! Adaptivly controls motor based on heartbeat
    STOP,                  //! Stops the motor and profile operation
    LOCKOUT,               //! Lockout the device until servied due to fault
    ESTOP                  //! Emergency Stops the motor in case of failure
} States_e;

/**
 * Startup tune for welcoming user
 */
const Note_t STARTUP_TUNE[] = {
    {NOTE_C,500},
    {NOTE_E,1000},
    {NOTE_C,500},
    {NOTE_G,1000},
    {NOTE_A,1000},
    {NOTE_C_S,500},
    {NOTE_B,500},
    {NOTE_PAUSE,100}};

/**
 * Heartbeat exceeded threshold warning to user
 */
const Note_t WARNING_TUNE[] = {
    {NOTE_A,150},
    {NOTE_PAUSE,100},
    {NOTE_A,150},
    {NOTE_PAUSE,75}};

/**
 * Component failed causing watchdog to expire tune
 */
const Note_t FAULT_TUNE[] = {
    {NOTE_G_S,500},
    {NOTE_F,200},
    {NOTE_PAUSE,300},
    {NOTE_D,250},
    {NOTE_PAUSE,100},
    {NOTE_G_S,500},
    {NOTE_F,200},
    {NOTE_PAUSE,300},
    {NOTE_D,250},
    {NOTE_PAUSE,100},
    {NOTE_G_S,500},
    {NOTE_F,200},
    {NOTE_PAUSE,300},
    {NOTE_D,250},
    {NOTE_PAUSE,100}};

#define RAMP_TIME_STEP      50   //! Delay between ramp state operations
#define MONITOR_TIME_STEP   50   //! Delay between monitor state operations
#define STOP_TIME_STEP      50   //! Delay between stop state operations
#define LOCKOUT_TIME_STEP   50   //! Delay between lockout state operations

#define RAMP_LOOP_COUNT     (5750 / RAMP_TIME_STEP)    //! Ramp iterations
#define STOP_LOOP_COUNT     (1750 / STOP_TIME_STEP)    //! Stop iterations
#define LOCKOUT_LOOP_COUNT  (1000 / LOCKOUT_TIME_STEP) //! USB Lockout timing

#define STOP_PWM_BASE       0    //! Motor PWM stop value
#define STOP_PWM_OFFSET     2.5f //! Fudge factor to allow for truncate to zero
#define MONITOR_DUTY_STEP   0.5f //! Iterations to ramp to desired PWM

#define HEARTBEAT_VALID_THRESHOLD 5 //! Less than this value is considred off

//==============================================================================
//                         Data Structure Declaration
//==============================================================================

/**
 * State Machine - Profile State attributes
 */
typedef struct State_Machine_Profile{
    uint8_t page_index;     //! Current page of profile menu
    bool page_displayed;    //! Flag noting that the page is rendered
    bool prev_press_right;  //! Flag noting that the next button is being held
}State_Machine_Profile_t;

/**
 * State Machine - Ramp State attributes
 */
typedef struct State_Machine_Ramp{
    uint8_t count;           //! Counter to cause transiton to next state
    float motor_increment;  //! Incrementer step to profile set point
}State_Machine_Ramp_t;

/**
 * State Machine - Stop State attributes
 */
typedef struct State_Machine_Stop{
    uint8_t count;           //! Counter to cause transition to next state
    float motor_decrement;  //! Decrement step to profile set point
}State_Machine_Stop_t;

/**
 * State Machine - Monitor State attributes
 */
typedef struct State_Machine_Monitor{
    bool startFlag;           //! Flag set before entry into state
    uint16_t init_heartbeat; //! Inital heartbeat for monitor state
}State_Machine_Monitor_t;

/**
 * State Machine - Lockout State attrubutes
 */
typedef struct State_Machine_Lockout{
    bool startFlag;          //! Flag set on entry into state
    uint16_t count;          //! Loop counter to time USB otuput
}State_Machine_Lockout_t;

/**
 * State Machine attributes as a collection of all states specific attrubutes
 */
typedef struct State_Machine {
    States_e state;          //! Current state of state machine
    const Profile_t* active_profile; //! Selected profile for monitor state
    State_Machine_Profile_t profile; //! Profile Select State Attributes
    State_Machine_Monitor_t monitor; //! Monitor heartbeat State attributes
    State_Machine_Ramp_t ramp; //! Ramp State Attributes
    State_Machine_Stop_t stop; //! Stop State attributes
    float current_duty;      //! Current duty cycle applied to the motor
    State_Machine_Lockout_t lockout; //! Lockout State Attributes
}State_Machine_t;

//==============================================================================
//                             Private Attributes
//==============================================================================

/**
 * Capacitive slider object
 */
static Slider_Event_t slider_event;
/**
 * USB UART transmission buffer
 */
static char txStr[25];
/**
 * State machine attributes
 */
static State_Machine_t state_machine;

//==============================================================================
//                         Private Function Prototypes
```

```
//===============================================================================

/**
 * Reset state for application state machine responsible for setting
 * application variables and sending a welcome message to the user
 * @return next state for application, WAIT_WAKE
 */
static inline States_e _reset(void);
/**
 * Wait state for waking up the treadmill. Will need user to send activation
 * sequence before a transition to waiting for profile selection
 * @return next state for application, WAIT_WAKE or WAIT_PROFILE
 */
static inline States_e _wait_for_wake(void);
/**
 * Wait state for selecting which profile to run on treadmill.
 * @return next state for application, WAIT_PROFILE, RAMPUP, or STOP
 */
static inline States_e _wait_for_profile(void);
/**
 * Rampup motor to the profile setpoint regardless of heartbeat input.
 * @return next state for application, MONITOR, STOP
 */
static inline States_e _rampup(void);
/**
 * Monitor state records user's heartbeat. Compares heartbeat trend to
 * profile response to control the speed of the motor.
 * @return next state for application, MONITOR, STOP
 */
static inline States_e _monitor(void);
/**
 * Stop state reduces the speed from current pwm speed to zero.
 * Ends session for user and goes to sleep
 * @return WAIT_WAKE
 */
static inline States_e _stop(void);
/**
 * Lockout state triggered on component failure. Prevents user from using
 * device until serviced.
 * @return LOCKOUT
 */
static inline States_e _lockout(void);
/**
 * Emergency stip state triggered when a fault is detected.
 * @return ESTOP state. However an infinite loop blocks exit
 */
static inline States_e _estop(void);

//===============================================================================
//                                  Main
//===============================================================================

/**
 * Main function will implement application state machine to run heartbeat
 * tracker treadmill.
 */
int main(void){

    //hardware initialization
    CyGlobalIntEnable;
    heartbeat_init(HEARTBEAT_SOURCE_EXTERNAL);
    music_init();
    display_init();
    profile_init();
    uart_init();
    userIn_init();
    motor_init();

    //if source of reset is watchdog then procede to lockout
    if(CY_RESET_WD == CyResetStatus || fault_check_for_failure()){
        state_machine.state = LOCKOUT;
        state_machine.lockout.startFlag = false;
        state_machine.lockout.count = 0;
    }else{
        state_machine.state = RESET;
        fault_init();
    }

    //run state machine loop indefinitly
    while(true){

        //update capacitive touch user input
        if(!CapSense_IsBusy()){
            CapSense_UpdateEnabledBaselines();
            CapSense_ScanEnabledWidgets();
        }

        //update usb interface
        uart_update();

        //check for component failure, transition to ESTOP
        if(fault_check_for_failure() && state_machine.state != LOCKOUT){
            state_machine.state = ESTOP;
        }

        //state machine update
        switch(state_machine.state){
            case RESET:
                state_machine.state = _reset();
                break;
            case WAIT_FOR_WAKE:
                state_machine.state = _wait_for_wake();
                break;
            case WAIT_FOR_PROFILE:
                state_machine.state = _wait_for_profile();
                break;
            case RAMPUP:
                state_machine.state = _rampup();
                break;
            case MONITOR:
                state_machine.state = _monitor();
                break;
            case STOP:
                state_machine.state = _stop();
                break;
            case LOCKOUT:
                state_machine.state = _lockout();
                break;
            case ESTOP:
                state_machine.state = _estop();
                break;
            default:
                state_machine.state = RESET;
                break;
        }
    }
```

```
}

//==============================================================================
//                          Private Function Implementation
//==============================================================================

/**
 * Reset state for application state machine responsible for setting
 * application variables and sending a welcome message to the user
 * @return next state for application, WAIT_WAKE
 */
static inline States_e _reset(void){
    //set default off states
    motor_stop();
    state_machine.active_profile = NULL;
    slider_event.dir = 0;
    slider_event.prev_val = NO_TOUCH_ON_SLIDER;
    slider_event.origin = 0;

    //welcome user
    music_set_tune(STARTUP_TUNE,(sizeof(STARTUP_TUNE)/sizeof(Note_t)));
    display_welcome();
    while(music_is_playing()){/* Block until welcome is done */};
    display_clear();
    return WAIT_FOR_WAKE;
}

/**
 * Wait state for waking up the treadmill. Will need user to send activation
 * sequence before a transition to waiting for profile selection
 * @return next state for application, WAIT_WAKE or WAIT_PROFILE
 */
static inline States_e _wait_for_wake(void){
    States_e retVal = WAIT_FOR_WAKE;
    Slider_e pos = userIn_check_slider_state(&slider_event,CapSense_SLIDER__LS);
    if(pos == SLIDE_RIGHT){
        retVal = WAIT_FOR_PROFILE;
        state_machine.profile.page_displayed = false;
        state_machine.profile.page_index = 0;
    }
    return retVal;
}

/**
 * Wait state for selecting which profile to run on treadmill.
 * @return next state for application, WAIT_PROFILE, RAMPUP, or STOP
 */
static inline States_e _wait_for_profile(void){
    States_e retVal = WAIT_FOR_PROFILE;
    //display page information on LCD
    if(!state_machine.profile.page_displayed){
        switch(state_machine.profile.page_index){
            case 0:
                display_profile_title();
                break;
            case 1:
            case 2:
            case 3:
            case 4:
                profile_menu_option(profile_get(state_machine.profile.page_index));
                break;
        }
        state_machine.profile.page_displayed = true;
    }

    //check user input for selection
    if(userIn_check_button_state(CapSense_LEFT__BTN)
            && state_machine.profile.page_index != 0
            && !userIn_check_button_state(CapSense_RIGHT__BTN)){
        retVal = RAMPUP;
        state_machine.active_profile =
profile_get(state_machine.profile.page_index);
        display_clear();

        //flash profile for user to see selection
        char temp[LCD_WIDTH+1];
        char padding[LCD_WIDTH+1];
        uint8_t name_size = strlen(state_machine.active_profile->name);
        int8_t pad_count = (LCD_WIDTH - name_size)/2;
        strcpy(padding,"");
        for(int8_t i=0; i<pad_count; ++i){
            strcat(padding," ");
        }
        sprintf(temp,"%s%s",padding,state_machine.active_profile->name);
        display_flash_text(temp,"",4,250,250);

        //setup for ramp state
        display_clear();
        heartbeat_set_use_LED(true);
        state_machine.ramp.count = 0;
        state_machine.ramp.motor_increment = 0;
        state_machine.current_duty = 0;
    }
    //check user input for toggle
    else if(userIn_check_button_state(CapSense_RIGHT__BTN)
            && !state_machine.profile.prev_press_right){
        state_machine.profile.prev_press_right = true;
        state_machine.profile.page_displayed = 0;
        state_machine.profile.page_index = (state_machine.profile.page_index + 1) %
5;
    }else if(!userIn_check_button_state(CapSense_RIGHT__BTN)){
        state_machine.profile.prev_press_right = false;
    }

    //check for stop slide event which cancels selection
    Slider_e pos = userIn_check_slider_state(&slider_event,CapSense_SLIDER__LS);
    if(pos == SLIDE_LEFT){
        display_clear();
        retVal = WAIT_FOR_WAKE;
    }

    return retVal;
}

/**
 * Rampup motor to the profile setpoint regardless of heartbeat input.
 * @return next state for application, MONITOR, STOP
 */
static inline States_e _rampup(void){
    States_e retVal = RAMPUP;

    //on inital rampup call, set LCD information, calculate motor increment
    if(0 == state_machine.ramp.count){
        display_clear();
        display_ramp_text();
        uint8_t target_pwm = state_machine.active_profile->pwm_base;
        state_machine.ramp.motor_increment = (target_pwm)
                                           / (RAMP_LOOP_COUNT * 1.0f);
        fault_in_critical_section(true);
    }

    //adjust motor speed
    state_machine.current_duty = motor_ramp(state_machine.active_profile->pwm_base,
            state_machine.current_duty,state_machine.ramp.motor_increment);

    //increment counter, if end of count then transition to next state
    state_machine.ramp.count++;
    if(state_machine.ramp.count >= RAMP_LOOP_COUNT){
        //setup for monitor state
        retVal = MONITOR;
        state_machine.monitor.init_heartbeat = heartbeat_get_bpm();
        state_machine.monitor.startFlag = true;
    }

    //check for stop slide event
    Slider_e pos = userIn_check_slider_state(&slider_event,CapSense_SLIDER__LS);
    if(pos == SLIDE_LEFT){
        retVal = STOP;
        //prepare state machine for stop state
        state_machine.stop.count = 0;
        state_machine.stop.motor_decrement = 0;
    }

    CyDelay(RAMP_TIME_STEP);
    return retVal;
}

/**
 * Monitor state records user's heartbeat. Compares heartbeat trend to
 * profile response to control the speed of the motor.
 * @return next state for application, MONITOR, STOP
 */
static inline States_e _monitor(void){
    States_e retVal = MONITOR;

    uint16_t current_heartbeat = heartbeat_get_bpm();
    char bpm[5];
    sprintf(bpm,"%d ",current_heartbeat);
    strcpy(txStr,"H:");
    strcat(txStr,bpm);
    uart_write(txStr,strlen(txStr));

    //display heartbeat on LCD
    if(state_machine.monitor.startFlag){
        state_machine.monitor.startFlag = false;
        display_profile(state_machine.active_profile->name,"Heartbeat:");
        fault_in_critical_section(true);
    }
    display_print_string(1,11,bpm);

    //calculate error from inital to current heartbeat
    float heartbeat_diff = (((float)(current_heartbeat)
                            / (float)(state_machine.monitor.init_heartbeat))
                            * 100.0f) - 100;

    //compensate motor state
    int pwm_steps = heartbeat_diff / 10 * ((heartbeat_diff < 0) ? -1 : 1);
    float pwm_step_size = 0;
    if(heartbeat_diff > 0){
        pwm_step_size = state_machine.active_profile->motor_heartbeat_shift_down;
    }else if(heartbeat_diff < 0){
        pwm_step_size = state_machine.active_profile->motor_heartbeat_shift_up;
    }
    float pwm_target = state_machine.active_profile->pwm_base
            + pwm_steps * pwm_step_size;
    if(pwm_target > 255){
        pwm_target = 255;
    }else if(pwm_target < 0){
        pwm_target = 0;
    }
    state_machine.current_duty = motor_ramp(pwm_target,
            state_machine.current_duty,MONITOR_DUTY_STEP);

    //check if alarm needs to be set
    if((heartbeat_diff >= 25) && !music_is_playing()){
        music_set_tune(WARNING_TUNE,sizeof(WARNING_TUNE)/sizeof(Note_t));
    }else if(heartbeat_diff < 25){
        music_stop_playing();
    }

    //check for stop slide event
    Slider_e pos = userIn_check_slider_state(&slider_event,CapSense_SLIDER__LS);
    if(pos == SLIDE_LEFT || current_heartbeat < HEARTBEAT_VALID_THRESHOLD){
        retVal = STOP;
        //prepare state machine for stop state
        state_machine.stop.count = 0;
        state_machine.stop.motor_decrement = 0;
    }

    CyDelay(MONITOR_TIME_STEP);
    return retVal;
}

/**
 * Stop state reduces the speed from current pwm speed to zero.
 * Ends session for user and goes to sleep
 * @return WAIT_WAKE
 */
static inline States_e _stop(void){
    States_e retVal = STOP;

    //on initial runthrough, display message and calculate motor step
    if(state_machine.stop.count == 0){
        heartbeat_set_use_LED(false);
        display_clear();
        display_stop_text();

        float target_pwm = state_machine.current_duty;
        state_machine.stop.motor_decrement = (target_pwm)
                                            / (STOP_LOOP_COUNT * 1.0f);
    }

    //adjust motor speed
    state_machine.current_duty = motor_ramp(STOP_PWM_BASE,
            state_machine.current_duty,state_machine.stop.motor_decrement);

    //increment the counter, if end of count then transition to next state
    state_machine.stop.count++;
    if(state_machine.stop.count >= STOP_LOOP_COUNT){
        display_clear(); //clear message before entring wait state
        motor_stop(); //make sure zero is not just approached
        retVal = WAIT_FOR_WAKE;
        fault_in_critical_section(false);
    }

    CyDelay(STOP_TIME_STEP);
    return retVal;
}

/**
 * Lockout state triggered on component failure. Prevents user from using
 * device until serviced.
 * @return LOCKOUT
 */
static inline States_e _lockout(void){

    if(state_machine.lockout.startFlag == false){
        state_machine.lockout.startFlag = true;
        motor_stop();
        music_set_tune(FAULT_TUNE,(sizeof(FAULT_TUNE)/sizeof(Note_t)));
        display_lockout();
    }

    //check if time has elapsed for usb error code
    state_machine.lockout.count++;
    if(state_machine.lockout.count >= LOCKOUT_LOOP_COUNT){
        state_machine.lockout.count = 0;
        char txStr[10];
        sprintf(txStr,"F:%x ",fault_code());
        uart_write(txStr,strlen(txStr));
    }

    CyDelay(LOCKOUT_TIME_STEP);
    return LOCKOUT;
}

/**
 * Emergency stip state triggered when a fault is detected.
 * @return ESTOP state. However an infinite loop blocks exit
 */
static inline States_e _estop(void){

    //ramp down the motor
    if(state_machine.current_duty > STOP_PWM_OFFSET){
        //gracefully shutdown motor
        state_machine.stop.count = 0;
        state_machine.stop.motor_decrement = 0;
        while(_stop() != WAIT_FOR_WAKE){/*Wait until motor is off*/}
    }
    motor_stop();

    //sit in loop until watchdog expires, triggering restart
    fault_in_critical_section(false);
    while(1);
    return ESTOP;
}
```

Code List 4: Main module state-machine implementation.

Profile

```
/**
 * Profile
 * @author Curt Henrichs
 * @date 9-13-17
 *
 * Profile module defines workout profile singletons as profile objects for
 * the treadmill application. Control of motor is dependent on the profile's
 * calculation, which is based on a heartbeat compensated PWM duty cycle.
 */

#ifndef PROFILE_H
#define PROFILE_H

//==============================================================================
//                                  Libraries
//==============================================================================

#include "Display.h"
#include <stdint.h>

//==============================================================================
//                         Constant and Macro Definitions
//==============================================================================

//! Max string size for name parameter
#define PROFILE_NAME_STR_LENGTH (MAX_PROFILE_NAME_LENGTH+1)
//! Max string size for detail parameter
#define PROFILE_DETAIL_STR_LENGTH (MAX_PROFILE_DETAIL_LENGTH+1)

/**
 * Enumeration of all profile cases
 */
typedef enum Profile_e {
    PROFILE_OFF = 0,    //! Not moving state
    PROFILE_CASUAL,     //! Easy workout for relaxation
    PROFILE_MODERATE,   //! Moderate workout for fitness
    PROFILE_INTENSE,    //! Hard-core workout, for humans
    PROFILE_EXTREME,    //! Hard-core workout, with max intensity

    //private enum to describe end of list
    _END_OF_PROFILE_LIST //! Not selectable enumeration (used for count)
} Profile_e;

#define CASUAL_MOTOR_RATE   33 //! Casual profile PWM rate
#define MODERATE_MOTOR_RATE 64 //! Moderate profile PWM rate
#define INTENSE_MOTOR_RATE  128 //! Intense profile PWM rate
#define EXTREME_MOTOR_RATE  192 //! Extreme profile PWM rate

//==============================================================================
//                           Data Structure Declaration
//==============================================================================

/**
 * Profile object defines the characteristics of each workout profile.
 */
typedef struct Profile {
    int8_t motor_heartbeat_shift_up;    //! Heartbeat increased by 10%
    int8_t motor_heartbeat_shift_down;  //! Heartbeat decreased by 10%
    int8_t warn_heartbeat_val;          //! Hartbeat increased past threshold
    uint16_t pwm_base;                  //! Starting PWM value pre-compensation
    char name[PROFILE_NAME_STR_LENGTH]; //! Name of profile as string
    char detail[PROFILE_DETAIL_STR_LENGTH]; //! Description of profile string
} Profile_t;

//==============================================================================
//                           Public Function Prototypes
//==============================================================================
/**
 * Profile singleton intialization function.
 */
void profile_init(void);
/**
 * Profile singleton selection function
 * @param profile enumeration of singletons
 * @return pointer to profile singleton given selection parameter. NULL if
 *         invalid.
 */
const Profile_t* profile_get(Profile_e profile);
/**
 * Profile menu display function
 * @param profile to display on LCD display
 */
void profile_menu_option(const Profile_t* profile);

#endif
```

Code List 5: Header file for the profile module.

```
/**
 * Profile
 * @author Curt Henrichs
 * @date 9-13-17
 *
 * Profile module defines workout profile singletons as profile objects for
 * the treadmill application. Control of motor is dependent on the profile's
 * calculation, which is based on a heartbeat compensated PWM duty cycle.
 */

//==============================================================================
//                                  Libraries
//==============================================================================

#include "Profile.h"
#include <project.h>
#include <string.h>

//==============================================================================
//                             Private Attributes
//==============================================================================

/**
 * Array of singleton profiles that is selectable by the profiles enumeration
 */
static Profile_t profiles[_END_OF_PROFILE_LIST];

//==============================================================================
//                        Public Function Implementation
//==============================================================================

/**
 * Profile singleton intialization function.
 */
void profile_init(void){
    //generate off object
    Profile_t* off = &profiles[PROFILE_OFF];
    off->motor_heartbeat_shift_down = 0;
    off->motor_heartbeat_shift_up = 0;
    off->warn_heartbeat_val = 0;
    off->pwm_base = 0;
    strcpy(off->name,"Off");
    strcpy(off->detail,"motor is off");
    //generate casual object
    Profile_t* casual = &profiles[PROFILE_CASUAL];
    casual->motor_heartbeat_shift_down = -51;
    casual->motor_heartbeat_shift_up = 13;
    casual->warn_heartbeat_val = 25;
    casual->pwm_base = CASUAL_MOTOR_RATE;
    strcpy(casual->name,"Casual");
    strcpy(casual->detail,"leisurely stroll");

    //generate moderate object
    Profile_t* moderate = &profiles[PROFILE_MODERATE];
    moderate->motor_heartbeat_shift_down = -26;
    moderate->motor_heartbeat_shift_up = 26;
    moderate->warn_heartbeat_val = 25;
    moderate->pwm_base = MODERATE_MOTOR_RATE;
    strcpy(moderate->name,"Moderate");
    strcpy(moderate->detail,"fit and toned");

    //generate intense object
    Profile_t* intense  = &profiles[PROFILE_INTENSE];
    intense->motor_heartbeat_shift_down = -13;
    intense->motor_heartbeat_shift_up = 26;
    intense->warn_heartbeat_val = 25;
    intense->pwm_base = INTENSE_MOTOR_RATE;
    strcpy(intense->name,"Intense");
    strcpy(intense->detail,"feel the burn");

    //generate extreme object
    Profile_t* extreme = &profiles[PROFILE_EXTREME];
    extreme->motor_heartbeat_shift_down = 0;
    extreme->motor_heartbeat_shift_up = 51;
    extreme->warn_heartbeat_val = 25;
    extreme->pwm_base = EXTREME_MOTOR_RATE;
    strcpy(extreme->name,"Extreme");
    strcpy(extreme->detail,"why?");
}

/**
 * Profile singleton selection function
 * @param profile enumeration of singletons
 * @return pointer to profile singleton given selection parameter. NULL if
 *         invalid.
 */
const Profile_t* profile_get(Profile_e profile){
    const Profile_t* retVal = NULL;
    if(profile < _END_OF_PROFILE_LIST){
        retVal = &profiles[profile];
    }
    return retVal;
}

/**
 * Profile menu display function
 * @param profile to display on LCD display
 */
void profile_menu_option(const Profile_t* profile){
    if(profile != NULL){
        display_profile(profile->name,profile->detail);
    }
}
```

Code List 6: Implementation file for profile module.

Display

```
/**
 * Display
 * @author Curt Henrichs
 * @date 9-10-17
 *
 * Display module abstracts the LCD interface into general high level messages
 * for user. All functions that rely on low level interface should be
 * implemented in this module.
 */

#ifndef DISPLAY_H
#define DISPLAY_H

//==============================================================================
//                                   Libraries
//==============================================================================

#include <project.h>

//==============================================================================
//                         Constant and Macro Defintions
//==============================================================================

#define LCD_HEIGHT                2  //! Rows of display
#define LCD_WIDTH                 16 //! Columns of display
#define MAX_PROFILE_NAME_LENGTH   10 //! Max size for valid display of string
#define MAX_PROFILE_DETAIL_LENGTH 16 //! Max size for valid display of string

#define display_clear() LCD_ClearDisplay() //! Aliase for clearing screen

//==============================================================================
//                          Public Function Prototypes
//==============================================================================

/**
 * Intialization function for display module starts hardware
 */
void display_init(void);
/**
 * Displays welcome message to the user, intended for startup of the module
 */
void display_welcome(void);
/**
 * Displays the profile menu title screen to explain process to user
 */
void display_profile_title(void);
/**
 * Displays a general profile page used for both selection and operation
 * display purposes
 * @param name is Profile name string
 * @param detail is second line string for specific details for profile
 */
void display_profile(const char* name, const char* detail);
/**
 * Displays a flashing text message on the LCD screen
 * @param l1 is line one text
 * @param l2 is line two text
 * @param c is count or number of times to flash
 * @param don is delay (in milliseconds) on per count
 * @param doff is delay (in millseconds) off per count
 */
```

```
void display_flash_text(const char* l1, const char* l2, uint32_t c,
        uint32_t don, uint32_t doff);
/**
 * Displays ramping motor text to user
 */
void display_ramp_text(void);
/**
 * Displays stopping motor text to user
 */
void display_stop_text(void);
/**
 * Displays a string at the provided position without clearing
 * screen.
 * @param r is row position of string origin
 * @param c is column position of string origin
 * @param str is the string to print to the display
 */
void display_print_string(uint8_t r, uint8_t c, const char* str);
/**
 * Prints lockout message to user onto the LCD
 */
void display_lockout(void);

#endif
```

Code List 7: Header file for display module.

```
/**
 * Display
 * @author Curt Henrichs
 * @date 9-10-17
 *
 * Display module abstracts the LCD interface into general high level messages
 * for user. All functions that rely on low level interface should be
 * implemented in this module.
 */

//==============================================================================
//                                  Libraries
//==============================================================================

#include "Display.h"
#include <project.h>
#include <stdint.h>
#include <stdio.h>

//==============================================================================
//                         Constant and Macro Defintions
//==============================================================================

#define BLOCK_CHAR  LCD_CUSTOM_0 //! Custom Block Character for top row
#define FILL_TO_MSG_DELAY_TIME 250 //! time to transition in welcome message

//==============================================================================
//                          Private Function Prototypes
//==============================================================================

/**
 * Fills the screen with blocks for startup to representing the loading to
 * the user.
 */
static inline void _fill_blocks(void);
/**
 * Welcome message displayed to the user
 */
static inline void _welcome_msg(void);
/**
 * Lockout message displayed to the user
 */
static inline void _lockout_msg(void);

//==============================================================================
//                        Public Function Implementation
//==============================================================================

/**
 * Intialization function for display module starts hardware
 */
void display_init(void){
    LCD_Start();
}

/**
 * Displays welcome message to the user, intended for startup of the module
 */
void display_welcome(void){
    _fill_blocks();
    CyDelay(FILL_TO_MSG_DELAY_TIME);
    _welcome_msg();
}

/**
 * Displays the profile menu title screen to explain process to user
 */
void display_profile_title(void){
    display_clear();
    LCD_Position(0,0);
    LCD_PrintString("Profile Menu");
    LCD_Position(1,0);
    LCD_PrintString("Select L, Next R");
}

/**
 * Displays a general profile page used for both selection and operation
 * display purposes
 * @param name is Profile name string
 * @param detail is second line string for specific details for profile
 */
void display_profile(const char* name, const char* detail){
    if(name == NULL || detail == NULL){
        return; //invalid input
    }else if(strlen(name) > MAX_PROFILE_NAME_LENGTH){
        return; //invalid length
    }else if(strlen(detail) > MAX_PROFILE_DETAIL_LENGTH){
        return; //invalid length
    }

    display_clear();
    LCD_Position(0,0);
    char temp[17];
    sprintf(temp,"Type: %s",name);
    LCD_PrintString(temp);
    LCD_Position(1,0);
    LCD_PrintString(detail);

}
```

```
/**
 * Displays a flashing text message on the LCD screen
 * @param l1 is line one text
 * @param l2 is line two text
 * @param c is count or number of times to flash
 * @param don is delay (in milliseconds) on per count
 * @param doff is delay (in millseconds) off per count
 */
void display_flash_text(const char* l1, const char* l2, uint32_t c,
        uint32_t don, uint32_t doff){
    //sanity check
    if(l1 == NULL || l2 == NULL){
        return;
    }else if(strlen(l1) > LCD_WIDTH || strlen(l2) > LCD_WIDTH){
        return;
    }else if(c == 0){
        return;
    }

    //flash text
    display_clear();
    for(uint32_t i=0; i<c; ++i){
        if(don > 0){
            LCD_Position(0,0);
            LCD_PrintString(l1);
            LCD_Position(1,0);
            LCD_PrintString(l2);
            CyDelay(don);
        }
        if(doff > 0){
            display_clear();
            CyDelay(doff);
        }
    }
    display_clear();
}

/**
 * Displays ramping motor text to user
 */
void display_ramp_text(void){
    LCD_Position(0,1);
    LCD_PrintString("Ramping  Motor");
    LCD_Position(1,2);
    LCD_PrintString("To Set Point");
}

/**
 * Displays stopping motor text to user
 */
void display_stop_text(void){
    LCD_Position(0,4);
    LCD_PrintString("Stopping");
    LCD_Position(1,5);
    LCD_PrintString("Motor");
}

/**
 * Displays a string at the provided position without clearing
 * screen.
 * @param r is row position of string origin
 * @param c is column position of string origin
 * @param str is the string to print to the display
 */
void display_print_string(uint8_t r, uint8_t c, const char* str){
    LCD_Position(r,c);
    LCD_PrintString(str);
}

/**
 * Prints lockout message to user onto the LCD
 */
void display_lockout(void){
    _fill_blocks();
    CyDelay(FILL_TO_MSG_DELAY_TIME);
    _lockout_msg();
}

//==============================================================================
//                        Private Function Implementation
//==============================================================================

/**
 * Fills the screen with blocks for startup to representing the loading to
 * the user.
 */
static inline void _fill_blocks(void){
    display_clear();
    LCD_Position(0,0);
    for(uint8_t c=0; c<16; ++c){
        LCD_PutChar(BLOCK_CHAR);
        CyDelay(100);
    }
}

/**
 * Welcome message displayed to the user
 */
static inline void _welcome_msg(void){
    display_clear();
    LCD_Position(0,3);
    LCD_PrintString("Welcome to");
    LCD_Position(1,2);
    LCD_PrintString("My Treadmill");
}

/**
 * Lockout message displayed to the user
 */
static inline void _lockout_msg(void){
    display_clear();
    LCD_Position(0,0);
    LCD_PrintString("Component Failed");
    LCD_Position(1,2);
    LCD_PrintString("DO NOT USE!!");
}
```

Code List 8: Implementation file for display module.

UserIn

```
/**
 * UserIn
 * @author Curt Henrichs
 * @date 9-23-17
 *
 * User Input module abstracts all user input hardware into a common module.
 * The goal is to decouple the hardware API from the software application.
 * Currently only a CapSense device is used with two buttons and a slider.
 * Future input would need to extend this module.
 */

#ifndef USERIN_H
#define USERIN_H

//==============================================================================
//                                  Library
//==============================================================================

#include <stdint.h>
#include <stdbool.h>

//==============================================================================
//                         Constant and Macro Definitions
//==============================================================================

#define NO_TOUCH_ON_SLIDER  0xFFFF  //! State returned when slider

/**
 * Enumeration of the events that the user can trigger with the slider
 */
typedef enum Slider_e {
    SLIDE_OFF,       //! User is not touching slider (default)
    SLIDE_LEFT,      //! User is moving left on slider
    SLIDE_RIGHT,     //! User is moving right on slider
    SLIDE_NO_MATCH, //! User has
    SLIDE_NOT_DETERMINED //! Not enough data to determine state yet
}Slider_e;

#define SLIDE_RIGHT_START   25 //! Origin requirment for right swipe
#define SLIDE_RIGHT_END     75 //! End requirement for right swipe
#define SLIDE_LEFT_START    75 //! Origin requirement for left swipe
#define SLIDE_LEFT_END      25 //! End requirement for left swipe

//==============================================================================
//                           Data Structure Declaration
//==============================================================================

/**
 * Slider processing object detects rising and falling edge of data to trigger
 * processing of user slide event as either left or right.
 */
typedef struct Slider_Event {
    uint16_t prev_val;  //! Previous value recorded from slider
    int8_t dir;         //! Current direction of swipe is either -1, 0, 1
    uint8_t origin;     //! Origin x value of swipe
} Slider_Event_t;

//==============================================================================
//                          Public Function Prototypes
//==============================================================================

/**
 * User input initalization function will start any hardware necessary to
 * process user input events.
 */
void userIn_init(void);
/**
 * User input process slider function will process the capacitive touch slider
 * for user input.
 * @param obj is pointer to Slider_Event_t object to assist in processing of
 *        slider data.
 * @param id is slider to check
 * @return current state of the slider
 */
Slider_e userIn_check_slider_state(Slider_Event_t* obj, uint8_t id);
/**
 * User input process button event will check a button for state of hardware.
 * @param id is the button to check
 * @return boolean where true is active state of button, false otherwise
 */
bool userIn_check_button_state(uint8_t id);

#endif
```

Code List 9: Header file for user input module.

```
/**
 * UserIn
 * @author Curt Henrichs
 * @date 9-23-17
 *
 * User Input module abstracts all user input hardware into a common module.
 * The goal is to decouple the hardware API from the software application.
 * Currently only a CapSense device is used with two buttons and a slider.
 * Future input would need to extend this module.
 */

//=============================================================================
//                                  Library
//=============================================================================

#include "UserIn.h"
#include <project.h>

//=============================================================================
//                         Constant and Macro Definitions
//=============================================================================

#define CAPSENSE_SETTLE_TIME  100 //Time to delay during initialization (ms)

//=============================================================================
//                         Public Function Implementation
//=============================================================================

/**
 * User input initalization function will start any hardware necessary to
 * process user input events.
 */
void userIn_init(void){
    CapSense_Start();
    CyDelay(CAPSENSE_SETTLE_TIME);
    CapSense_InitializeAllBaselines();
}

/**
 * User input process slider function will process the capacitive touch slider
 * for user input.
 * @param obj is pointer to Slider_Event_t object to assist in processing of
 *        slider data.
 * @param id is slider to check
 * @return current state of the slider
 */
Slider_e userIn_check_slider_state(Slider_Event_t* obj, uint8_t id){
    //parameter check
    if(obj == NULL){
        return SLIDE_NO_MATCH;
    }

    //process slider
    Slider_e retVal;
    uint16_t pos = CapSense_GetCentroidPos(id);

    //if start of button press
    if(obj->prev_val == NO_TOUCH_ON_SLIDER
            && pos != NO_TOUCH_ON_SLIDER){
        retVal = SLIDE_NOT_DETERMINED;
        obj->dir = 0;
        obj->origin = pos;
    }
    //else if up event
    else if(obj->prev_val != NO_TOUCH_ON_SLIDER
            && pos == NO_TOUCH_ON_SLIDER){
        //if slide a left, right, or no match?
        if(obj->origin <= SLIDE_RIGHT_START
                && obj->prev_val >= SLIDE_RIGHT_END){
            retVal = SLIDE_RIGHT;
        }else if(obj->origin >= SLIDE_LEFT_START
                && obj->prev_val <= SLIDE_LEFT_END){
            retVal = SLIDE_LEFT;
        }else{
            retVal = SLIDE_OFF;
        }
    }
    //else if not touching consistent event
    else if(obj->prev_val == NO_TOUCH_ON_SLIDER
            && pos == NO_TOUCH_ON_SLIDER){
        retVal = SLIDE_OFF;
    }
    //else currently sliding
    else{
        retVal = SLIDE_NOT_DETERMINED;

        //if user starts moving in the reverse direction, reset origin
        int8_t curr_dir = pos - obj->prev_val;
        curr_dir = (curr_dir > 0) ? 1 : (curr_dir < 0) ? -1 : 0;
        if(curr_dir != 0 && curr_dir != obj->dir){
            obj->dir = curr_dir;
            obj->origin = pos;
        }
    }
    obj->prev_val = pos;

    return retVal;
}

/**
 * User input process button event will check a button for state of hardware.
 * @param id is the button to check
 * @return boolean where true is active state of button, false otherwise
 */
bool userIn_check_button_state(uint8_t id){
    return CapSense_CheckIsWidgetActive(id);
}
```

Code List 10: Implementation file for user input module.

Music

```
/**
 * Music
 * @author Curt Henrichs
 * @date 9-10-17
 *
 * Music module provides interface to the Piezo buzzer hardware to produce
 * frequency modulated, variable duration waveforms. An object is developed
 * to interface with the hardware and create music arrays. Finally all
 * operations return control quickly to the caller as the hardware uses
 * interruts to handle note clockout.
 */

#ifndef MUSIC_H
#define MUSIC_H

//==============================================================================
//                                   Libraries
//==============================================================================

#include <stdint.h>

//==============================================================================
//                         Constant and Macro Defintions
//==============================================================================

#define NOTE_C       (262) //! Hz
#define NOTE_C_S     (277) //! Hz
#define NOTE_D       (294) //! Hz
#define NOTE_D_S     (311) //! Hz
#define NOTE_E       (330) //! Hz
#define NOTE_F       (349) //! Hz
#define NOTE_F_S     (370) //! Hz
#define NOTE_G       (392) //! Hz
#define NOTE_G_S     (415) //! Hz
#define NOTE_A       (440) //! Hz
#define NOTE_B       (494) //! Hz

#define NOTE_PAUSE   (0)   //! No frequency, thus no sound

//==============================================================================
//                           Data Structure Declaration
//==============================================================================

/**
 * Note object describes the piezo note to play at a frequency and duration
 * By chainging these together in an array, one can generate a song
 */
typedef struct Note {
    uint16_t frequency;     //! Frequency of note in Hz
    uint16_t duration;      //! Duration of note in milliseconds
} Note_t;

//==============================================================================
//                           Public Function Prototypes
//==============================================================================

/**
 * Initalization function will start hardware, set ISRs for hardware, and
 * set internal data structures for playback.
 */
void music_init(void);
```

```
/**
 * Sets a tune to play. This is a non-blocking function so that other things
 * may happen. Most of the music generation is in hardware but a periodic
 * interrupt is needed to update notes.
 * @param buffer is array of notes to play on the buzzer
 * @param length is the number of Notes in the buffer to play
 */
void music_set_tune(const Note_t* buffer, uint32_t length);
/**
 * Checks to see if the music is still playing
 * @return 1 if music is playing else 0
 */
uint8_t music_is_playing(void);
/**
 * Stops the music from playing then clears the music buffer assoication
 */
void music_stop_playing(void);

#endif
```

Code List 11: Header file for music module.

```
/**
 * Music
 * @author Curt Henrichs
 * @date 9-10-17
 *
 * Music module provides interface to the Piezo buzzer hardware to produce
 * frequency modulated, variable duration waveforms. An object is developed
 * to interface with the hardware and create music arrays. Finally all
 * operations return control quickly to the caller as the hardware uses
 * interruts to handle note clockout.
 */

//==============================================================================
//                                  Libraries
//==============================================================================

#include "Music.h"
#include <project.h>

//==============================================================================
//                         Constant and Macro Defintions
//==============================================================================

#define TONE_F_CLK_FREQ_IN 40000    //! Clock frequency that is divided for out

//==============================================================================
//                              Private Attributes
//==============================================================================

static const Note_t* _current_buffer = NULL; //! Music buffer association
static uint32_t _current_buffer_length = 0;  //! Size of buffer
static uint32_t _current_buffer_index = 0;   //! Index into buffer clockout

//==============================================================================
//                          Interrupt Handler Prototypes
//==============================================================================

/**
 * Music note clock out ISR is linked to the duration counter. Only active
 * when music is actively being played
 */
CY_ISR_PROTO(music_isr);
```

```c
//==============================================================================
//                         Public Function Implementation
//==============================================================================

/**
 * Initalization function will start hardware, set ISRs for hardware, and
 * set internal data structures for playback.
 */
void music_init(void){
    Tone_F_Start();
    Tone_F_WriteCompare(0);
    Tone_D_Start();
    Tone_D_Stop(); //sets up hardware but then disables timer
    Tone_Int_StartEx(music_isr);
}

/**
 * Sets a tune to play. This is a non-blocking function so that other things
 * may happen. Most of the music generation is in hardware but a periodic
 * interrupt is needed to update notes.
 * @param buffer is array of notes to play on the buzzer
 * @param length is the number of Notes in the buffer to play
 */
void music_set_tune(const Note_t* buffer, uint32_t length){
    Tone_Int_Disable();

    //copy buffer location for playback
    _current_buffer = buffer;
    _current_buffer_length = length;
    _current_buffer_index = 0;

    //set up timer counter
    Tone_D_WritePeriod(1);
    Tone_D_WriteCounter(0);
    Tone_Int_Enable();
    Tone_D_Start();
}

/**
 * Checks to see if the music is still playing
 * @return 1 if music is playing else 0
 */
uint8_t music_is_playing(void){
    return _current_buffer != NULL;
}

/**
 * Stops the music from playing then clears the music buffer assoication
 */
void music_stop_playing(void){
    _current_buffer = NULL;
    Tone_F_WriteCompare(0);
    Tone_D_Stop();
}

//==============================================================================
//                        Interrupt Handler Implementation
//==============================================================================

/**
 * Music note clock out ISR is linked to the duration counter. Only active
 * when music is actively being played
 */
CY_ISR(music_isr){
    static uint16_t freq = 0;
    static uint16_t per = 0;
    static uint16_t cmp = 0;
    static uint16_t dur  = 0;

    //play next note
    Tone_D_Stop();
    if(_current_buffer != NULL){
        if(_current_buffer_index >= _current_buffer_length){
            //end of tune
            _current_buffer = NULL;
            Tone_F_WriteCompare(0);
            Tone_D_Stop();
        }else{
            //play for current frequency and duration
            freq = _current_buffer[_current_buffer_index].frequency;
            per = TONE_F_CLK_FREQ_IN / ((freq > 0)? freq : 1);
            cmp = (freq > 0) ? per / 2 : 0;
            dur = _current_buffer[_current_buffer_index].duration;
            _current_buffer_index++;

            Tone_F_WritePeriod(per);
            Tone_F_WriteCompare(cmp);
            Tone_D_WritePeriod(dur);
            Tone_D_WriteCounter(0);
            Tone_D_Start();
        }
    }
    Tone_D_ReadStatusRegister();
    Tone_Int_ClearPending();
}
```

Code List 12: Implementation file for music module.

Heartbeat

```
/**
 * Heartbeat
 * @author Curt Henrichs
 * @date 9-17-17
 *
 * Heartbeat module is a software module tasked with collecting heartbeat
 * rate from the user for the treadmill application. Heartbeat module operates
 * via interrupts. Interrupts are on ADC (for waveform recreation), on counter
 * (for overflow, signalling no pulse), and on an analog comparator (to detect
 * heartbeat peak and therefore measure frequency).
 */

#ifndef HEARTBEAT_H
#define HEARTBEAT_H

//==============================================================================
//                                   Libraries
//==============================================================================

#include <stdint.h>
#include <stdbool.h>

//==============================================================================
//                          Constant and Macro Defintions
//==============================================================================
/**
 * Heartbeat source enumeration to direct the analog MUX channel
 */
typedef enum Heartbeat_Source {
    HEARTBEAT_SOURCE_OFF,        //! Internal with DAC off
    HEARTBEAT_SOURCE_INTERNAL,   //! Internal with DAC generating waveform
    HEARTBEAT_SOURCE_EXTERNAL    //! External pin, DAC off
} Heartbeat_Source_e;

//==============================================================================
//                           Public Function Prototypes
//==============================================================================

/**
 * Heartbeat module intialization function will start all module hardware and
 * intialize all private attributes. This module is entirely driven by
 * interrupts so it is crucial that global interrupts are enabled.
 * @param source is the heartbeat input source for analog MUX channel select
 */
void heartbeat_init(Heartbeat_Source_e source);
/**
 * @return heartbeat beats per minute as standard measure of user's heartrate
 */
uint16_t heartbeat_get_bpm(void);
/**
 * Clears the previous heartbeat data, thereby reseting the module
 * Note hardware is not modified.
 */
void heartbeat_clear(void);
/**
 * Sets the use LED flag so that heartbeat is pulsed out
 * @param state, true if activate LED heartbeat, false if disable
 */
void heartbeat_set_use_LED(bool state);
/**
 * @return gets the use LED flag state. true if using, false if disabled
 */
bool heartbeat_get_use_LED(void);

#endif
```

Code List 13: Header file for heartbeat module.

```
/**
 * Heartbeat
 * @author Curt Henrichs
 * @date 9-17-17
 *
 * Heartbeat module is a software module tasked with collecting heartbeat
 * rate from the user for the treadmill application. Heartbeat module operates
 * via interrupts. Interrupts are on ADC (for waveform recreation), on counter
 * (for overflow, signalling no pulse), and on an analog comparator (to detect
 * heartbeat peak and therefore measure frequency).
 */

//==============================================================================
//                                    Libraries
//==============================================================================

#include "Heartbeat.h"
#include <project.h>
#include "RunningAverage.h"

//==============================================================================
//                          Constant and Macro Defintions
//==============================================================================

#define HB_MUX_INTERNAL_CH 0 //! Heartbeat internal channel connected to DAC

#define HB_MUX_EXTERNAL_CH 1 //! Heartbeat external channel connected to pin

#define HB_COMP_HEARTBEAT_THRESHOLD_VALUE 100 //! DAC value for comparator

#define HEARTBEAT_FILTER_SIZE 8 //! Filter bins for running average

#define MAX_POSSIBLE_HEARTBEAT 300 //! Highest heartbeat allowed, else zero

//==============================================================================
//                               Private Attributes
//==============================================================================

/**
 * FIR filter of BPM to remove noise in measurement
 */
static volatile Running_Average_t _filter;
/**
 * Boolean flag to signal whether the heartbeat detection will flash LED
 */
static volatile bool _use_led_state = false;

//==============================================================================
//                           Interrupt Handler Prototypes
//==============================================================================

/**
 * Comparator ISR is triggered on rising edge of peak detection. Responsible
 * for calculating current beats per minute and storing that result into
 * running average. Resets counter to zero for next comparison.
 */
CY_ISR_PROTO(comp_isr);
/**
 * ADC EOC ISR is triggered on end of conversion event. Responsible for reading
 * result from ADC, scaling it, and recreating it on the DAC.
 */
CY_ISR_PROTO(ADC_EOC_isr);
/**
 * Counter overflow ISR is triggered when counter register overflows. This
 * interrupt is a fail-safe to record zero heartbeat.
 */
CY_ISR_PROTO(counter_overflow_isr);

//==============================================================================
//                           Public Function Implementation
//==============================================================================

/**
 * Heartbeat module intialization function will start all module hardware and
 * intialize all private attributes. This module is entirely driven by
 * interrupts so it is crucial that global interrupts are enabled.
 * @param source is the heartbeat input source for analog MUX channel select
 */
void heartbeat_init(Heartbeat_Source_e source){

    //initalize attributes
    running_average_init((Running_Average_t*)(&_filter),HEARTBEAT_FILTER_SIZE);

    //start analog output hardware
    HB_Comp_Neg_DAC_Start();
    HB_Comp_Neg_DAC_SetValue(HB_COMP_HEARTBEAT_THRESHOLD_VALUE);
    HB_DAC_Start();
    HB_DAC_SetValue(0);

    //start analog input hardware
    HB_ADC_Start();
    HB_ADC_StartConvert();
    HB_ADC_IRQ_Enable();
    HB_Comp_Start();
    HB_Counter_Start();

    //select source and start
    HB_MUX_Start();
    HB_PGA_Start();
    if(source == HEARTBEAT_SOURCE_INTERNAL){
        //internal state
        I_HB_Start();
        HB_MUX_FastSelect(HB_MUX_INTERNAL_CH);
    }else if(source == HEARTBEAT_SOURCE_EXTERNAL){
        //external state
        HB_MUX_FastSelect(HB_MUX_EXTERNAL_CH);
    }else{
        //off state uses turned off DAC for 0V input
        HB_MUX_FastSelect(HB_MUX_INTERNAL_CH);
    }

    //start interrupts
    HB_ADC_IRQ_StartEx(ADC_EOC_isr);
    HB_Comp_Int_StartEx(comp_isr);
    HB_Overflow_Int_StartEx(counter_overflow_isr);
}

/**
 * @return heartbeat beats per minute as standard measure of user's heartrate
 */
uint16_t heartbeat_get_bpm(void){
    return running_average_result((Running_Average_t*)(&_filter));
}

/**
 * Clears the previous heartbeat data, thereby reseting the module
 * Note hardware is not modified.
 */
void heartbeat_clear(void){
    running_average_clear((Running_Average_t*)(&_filter));
}

/**
 * Sets the use LED flag so that heartbeat is pulsed out
 * @param state, true if activate LED heartbeat, false if disable
 */
void heartbeat_set_use_LED(bool state){
    _use_led_state = state;
    if(!_use_led_state){
        LED_Write(0);
    }
}

/**
 * @return gets the use LED flag state. true if using, false if disabled
 */
bool heartbeat_get_use_LED(void){
    return _use_led_state;
}

//==============================================================================
//                       Interrupt Handler Implementation
//==============================================================================

/**
 * Comparator ISR is triggered on rising edge of peak detection. Responsible
 * for calculating current beats per minute and storing that result into
 * running average. Resets counter to zero for next comparison.
 */
CY_ISR(comp_isr){
    static uint16_t count = 0;
    static float bpm = 0;

    //calculate the new heartrate in BPM
    count = HB_Counter_ReadCounter(); //count is ms between peaks
    bpm = 60000 / ((count > 0) ? (count) : (1)); // in BPM
    bpm = (bpm > MAX_POSSIBLE_HEARTBEAT) ? (0) : (bpm);
    running_average_update((Running_Average_t*)(&_filter),(uint16_t)(bpm));

    //clear source of interrupt
    HB_Counter_WriteCounter(0);
    HB_Comp_Int_ClearPending();

    //pulse LED to simulate heartbeat
    if(_use_led_state){
        LED_Write(!LED_Read());
    }
}

/**
 * ADC EOC ISR is triggered on end of conversion event. Responsible for reading
 * result from ADC, scaling it, and recreating it on the DAC.
 */
CY_ISR(ADC_EOC_isr){
    static uint8_t adc_val = 0;

    //get value and clear interrupt source
    adc_val = HB_ADC_GetResult8();

    //send DAC new value
    HB_DAC_SetValue(adc_val);
    HB_ADC_IRQ_ClearPending();
}

/**
 * Counter overflow ISR is triggered when counter register overflows. This
 * interrupt is a fail-safe to record zero heartbeat.
 */
CY_ISR(counter_overflow_isr){
    //store zero to heartbeat
    running_average_clear((Running_Average_t*)(&_filter));

    //clear source of interrupt
    HB_Counter_ReadStatusRegister();
    HB_Overflow_Int_ClearPending();
}
```

Code List 14: Implementation file for heartbeat module.

RunningAverage

```
/**
 * RunningAverage
 * @author Curt Henrichs
 * @date 9-17-17
 *
 * Running average FIR filter object. Goal of this module is to provide an
 * object that takes in sensor data and through an averaging FIR filter,
 * produce a lowpass average. Module allows user to specify number of bins
 * which are then constructed in heap space.
 */

#ifndef RUNNINGAVERAGE_H
#define RUNNINGAVERAGE_H

//===============================================================================
//                                   Libraries
//===============================================================================

#include <stdint.h>
#include <stdbool.h>

//===============================================================================
//                         Constant and Macro Defintions
//===============================================================================

#define RUNNING_AVERAGE_BIN_MAX   256   //! Maximum number of bins
#define RUNNING_AVERAGE_BIN_MIN   1     //! Minimum number of bins

//===============================================================================
//                           Data Structure Declaration
//===============================================================================

/**
 * Running average FIR filter object structure.
 */
typedef struct Running_Average {
    uint16_t* bins; //! Array of previous values
    uint16_t size;  //! Size of the array
    uint16_t head;  //! Next value position
} Running_Average_t;

//===============================================================================
//                            Public Function Prototypes
//===============================================================================

/**
 * Running average object initalization function. Allocates the specified
 * number of bins into heap space. Note user should call the destroy function
 * to free this space.
 * @param obj valid pointer to Running_Average object
 * @param num_bins is number of previous values to store. Must follow range.
 * @return true if allocated bins, else false due to failure
 */
bool running_average_init(Running_Average_t* obj, uint16_t num_bins);
/**
 * Running average object update function. Replaces oldest value with the
 * new supplied value.
 * @param obj valid pointer to Running_Average object
 * @param val is value to place into filter
 */
void running_average_update(Running_Average_t* obj, uint16_t val);
```

```
/**
 * Running average object result calculation function. Calculates the average
 * of all values in the filter. Note that the result will default to zero
 * if the object is invalid. Additonally, at start of filter, due to lowpass
 * nature, the result will ramp up from zero to the average input value
 * @param obj valid pointer to Running_Average object
 * @return average result from the filter
 */
uint16_t running_average_result(Running_Average_t* obj);
/**
 * Running average clear function will set all bins to zero.
 * @param obj valid pointer to Running_Average object
 */
void running_average_clear(Running_Average_t* obj);
/**
 * Running average destroy function will free heap memory and clear
 * data members
 * @param obj valid pointer to Running_Average object
 */
void running_average_destroy(Running_Average_t* obj);

#endif
```

Code List 15: Header file for running average module.

```
/**
 * RunningAverage
 * @author Curt Henrichs
 * @date 9-17-17
 *
 * Running average FIR filter object. Goal of this module is to provide an
 * object that takes in sensor data and through an averaging FIR filter,
 * produce a lowpass average. Module allows user to specify number of bins
 * which are then constructed in heap space.
 */

//==============================================================================
//                                    Libraries
//==============================================================================

#include "RunningAverage.h"
#include <stdlib.h>

//==============================================================================
//                          Public Function Implementation
//==============================================================================

/**
 * Running average object initalization function. Allocates the specified
 * number of bins into heap space. Note user should call the destroy function
 * to free this space.
 * @param obj valid pointer to Running_Average object
 * @param num_bins is number of previous values to store. Must follow range.
 * @return true if allocated bins, else false due to failure
 */
bool running_average_init(Running_Average_t* obj, uint16_t num_bins){
    bool retVal = false;
    if(obj != NULL && num_bins < RUNNING_AVERAGE_BIN_MAX
            && num_bins >= RUNNING_AVERAGE_BIN_MIN){
        obj->bins = (uint16_t*)(malloc(num_bins * sizeof(uint16_t)));
        if(obj->bins != NULL){
            retVal = true;
            obj->head = 0;
            obj->size = num_bins;
            for(uint16_t i=0; i<obj->size; ++i){
                obj->bins[i] = 0;
            }
        }
    }
    return retVal;
}

/**
 * Running average object update function. Replaces oldest value with the
 * new supplied value.
 * @param obj valid pointer to Running_Average object
 * @param val is value to place into filter
 */
void running_average_update(Running_Average_t* obj, uint16_t val){
    if(obj != NULL && obj->bins != NULL && obj->head < obj->size
            && obj->size >= RUNNING_AVERAGE_BIN_MIN){
        obj->bins[obj->head] = val;
        obj->head = (++obj->head) % obj->size;
    }
}

/**
 * Running average object result calculation function. Calculates the average
 * of all values in the filter. Note that the result will default to zero
 * if the object is invalid. Additonally, at start of filter, due to lowpass
 * nature, the result will ramp up from zero to the average input value
 * @param obj valid pointer to Running_Average object
 * @return average result from the filter
 */
uint16_t running_average_result(Running_Average_t* obj){
    uint16_t retVal = 0;
    if(obj != NULL && obj->bins != NULL
            && obj->size >= RUNNING_AVERAGE_BIN_MIN){
        float sum = 0;
        for(uint16_t i=0; i<obj->size; ++i){
            sum += obj->bins[i];
        }
        retVal = (uint16_t)(sum / obj->size);
    }
    return retVal;
}

/**
 * Running average clear function will set all bins to zero.
 * @param obj valid pointer to Running_Average object
 */
void running_average_clear(Running_Average_t* obj){
    if(obj != NULL && obj->bins != NULL){
        for(uint16_t i=0; i<obj->size; ++i){
            obj->bins[i] = 0;
        }
    }
}

/**
 * Running average destroy function will free heap memory and clear
 * data members
 * @param obj valid pointer to Running_Average object
 */
void running_average_destroy(Running_Average_t* obj){
    if(obj != NULL && obj->bins != NULL){
        free(obj->bins);
        obj->bins = NULL;
        obj->head = 0;
        obj->size = 0;
    }
}
```

Code List 16: Implementation file for running average module.

Uart

```
/**
 * Uart
 * @author Curt Henrichs
 * @date 9-23-17
 *
 * USB CDC Virtual UART applicaiton interface module generalizes the hardware
 * API for the specific use cases of the application. The ability to read,
 * write, check if data is available, and handling of host configuration change
 * is abstracted by this module.
 */

#ifndef UART_H
#define UART_H

//==============================================================================
//                                  Libraries
//==============================================================================

#include <stdint.h>
#include <stdbool.h>

//==============================================================================
//                         Constant and Macro Defintions
//==============================================================================

#define USBUART_BUFFER_SIZE (64u)   //! Maximum size of USB buffer for Rx, Tx

//==============================================================================
//                          Public Function Prototypes
//==============================================================================

/**
 * USB UART initailization function will start the virtual serial port
 */
void uart_init(void);
/**
 * USB UART update function will handle configuration change from the USB host
 */
void uart_update(void);
/**
 * USB UART received new data, needs to be taken from internal buffer
 * @return true if new data else false
 */
bool uart_has_data(void);
/**
 * Get data from USB UART internal receive buffer.
 * @param str is pointer to buffer to write into
 * @param str_len is buffer size
 * @return number of bytes read from UART internal buffer into supplied
 */
uint8_t uart_read(char* str, uint16_t str_len);
/**
 * Write data to USB UART internal transmit buffer.
 * @param str is pointer to buffer to read from
 * @param str_len is size of buffer to write out.
 * @return true of successfully wrote data, else false
 */
bool uart_write(const char* str, uint16_t str_len);

#endif
```

Code List 17: Header file for UART module.

```
/**
 * Uart
 * @author Curt Henrichs
 * @date 9-23-17
 *
 * USB CDC Virtual UART applicaiton interface module generalizes the hardware
 * API for the specific use cases of the application. The ability to read,
 * write, check if data is available, and handling of host configuration change
 * is abstracted by this module.
 */

//==============================================================================
//                                   Libraries
//==============================================================================

#include "Uart.h"
#include <project.h>

//==============================================================================
//                        Constant and Macro Defintions
//==============================================================================

#define USBFS_DEVICE         (0u)    //! Device number as defined in hardware
#define WAIT_TIMEOUT_COUNT   1000    //! Iterations of loop before timeout

//==============================================================================
//                         Public Function Implementation
//==============================================================================

/**
 * USB UART initailization function will start the virtual serial port
 */
void uart_init(void){
    USBUART_Start(USBFS_DEVICE,USBUART_3V_OPERATION);
}

/**
 * USB UART update function will handle configuration change from the USB host
 */
void uart_update(void){
    //reconfiguration
    if(USBUART_IsConfigurationChanged()){
        if(USBUART_GetConfiguration()){
            USBUART_CDC_Init();
        }
    }
}

/**
 * USB UART received new data, needs to be taken from internal buffer
 * @return true if new data else false
 */
bool uart_has_data(void){
    return USBUART_GetConfiguration() ? USBUART_DataIsReady() : false;
}
```

```
/**
 * Get data from USB UART internal receive buffer.
 * @param str is pointer to buffer to write into
 * @param str_len is buffer size
 * @return number of bytes read from UART internal buffer into supplied
 */
uint8_t uart_read(char* str, uint16_t str_len){

    //parameter check
    if(str == NULL || str_len > USBUART_BUFFER_SIZE || str_len == 0){
        return 0;
    }

    //service CDC interface
    uint8_t count = 0;
    if(USBUART_GetConfiguration()){
        //input data from host
        if(USBUART_DataIsReady()){
            count = USBUART_GetAll((uint8_t*)(str));
        }
    }
    return count;
}

/**
 * Write data to USB UART internal transmit buffer.
 * @param str is pointer to buffer to read from
 * @param str_len is size of buffer to write out.
 * @return true of successfully wrote data, else false
 */
bool uart_write(const char* str, uint16_t str_len){
    bool retVal = false;

    //parameter check
    if(str == NULL || str_len > USBUART_BUFFER_SIZE){
        return false;
    }else if(str_len == 0){
        return true;
    }

    //send data
    if(USBUART_GetConfiguration()){
        //wait for uart to be ready
        int16_t timeout_count = WAIT_TIMEOUT_COUNT;
        while(!USBUART_CDCIsReady() && timeout_count > 0){
            timeout_count--;
        }
        //if not timeout
        if(timeout_count > 0){
            retVal = true;
            USBUART_PutData((const uint8_t*)(str),str_len);
        }
    }

    return retVal;
}
```

Code List 18: Implementation file for UART module.

MotorPWM

```
/**
 * MotorPWM
 * @author Curt Henrichs
 * @date 10-17-17
 *
 * Motor PWM module controls a single PWM signal that is intended to control
 * a motor. This module provides a layer of decoupling between the state
 * machine and the hardware. Note that this module does not auto-update the
 * state of the motor. Instead the client software must invoke this module
 * appropriately.
 */

#ifndef MOTOR_H
#define MOTOR_H

//==============================================================================
//                                   Libraries
//==============================================================================

#include <project.h>
#include <stdbool.h>

//==============================================================================
//                         Constant and Macro Defintions
//==============================================================================

#define MOTOR_STOP_PWM (0)  //! PWM value that represents off to motor

/**
 * Stops the PWM pulse to the motor
 */
#define motor_stop() (PWM_WriteCompare(MOTOR_STOP_PWM))

//==============================================================================
//                          Public Function Prototypes
//==============================================================================

/**
 * Initalizes the motor software and hardware modules
 */
void motor_init(void);
/**
 * Calculates a step toward the desired target PWM and sets the motor. Requires
 * external tracking of PWM current value. This function must be called in an
 * external loop to actually ramp the motor to speed.
 * @param target is the end PWM to ramp to. Note that his value must be
 *        bounded between 0 and 255 inclusively
 * @param base is the current PWM of the motor. Note that his value must be
 *        bounded between 0 and 255 inclusively
 * @param step is the increment (or decrement) to reach the target. Note that
 *        this value should be positive
 * @return base + step until it reaches target, then returns target.
 */
float motor_ramp(float target, float base, float step);

#endif
```

Code List 19: Header file for motor module.

```
/**
 * MotorPWM
 * @author Curt Henrichs
 * @date 10-17-17
 *
 * Motor PWM module controls a single PWM signal that is intended to control
 * a motor. This module provides a layer of decoupling between the state
 * machine and the hardware. Note that this module does not auto-update the
 * state of the motor. Instead the client software must invoke this module
 * appropriately.
 */

//===============================================================================
//                                  Libraries
//===============================================================================

#include "MotorPWM.h"

//===============================================================================
//                         Public Function Implementation
//===============================================================================

/**
 * Initalizes the motor software and hardware modules
 */
void motor_init(void){
    PWM_Start();
}

/**
 * Calculates a step toward the desired target PWM and sets the motor. Requires
 * external tracking of PWM current value. This function must be called in an
 * external loop to actually ramp the motor to speed.
 * @param target is the end PWM to ramp to. Note that his value must be
 *        bounded between 0 and 255 inclusively
 * @param base is the current PWM of the motor. Note that his value must be
 *        bounded between 0 and 255 inclusively
 * @param step is the increment (or decrement) to reach the target. Note that
 *        this value should be positive
 * @return base + step until it reaches target, then returns target.
 */
float motor_ramp(float target, float base, float step){
    float duty = 0;
    step = step * ((target < base) ? -1 : 1);
    if(step >= 0 && (base + step) > target){
        duty = target;
    }else if(step < 0 && (base + step) < target){
        duty = target;
    }else {
        duty = base + step;
    }
    PWM_WriteCompare((uint8_t)(duty));
    return duty;
}
```

Code List 20: Implementation file for motor module.

Fault

```
/**
 * Fault
 * @author Curt Henrichs
 * @date 10-17-17
 *
 * Fault detection module will determine if any component have failed. If
 * failure occured then the watchdog epxire unless system has entered a critical
 * section, where then the main loop takes control of the watchdog.
 */

#ifndef FAULT_H
#define FAULT_H
```

```
//===============================================================================
//                                  Libraries
//===============================================================================

#include <stdint.h>
#include <stdbool.h>

//===============================================================================
//                         Public Function Prototypes
//===============================================================================

/**
 * Initialization function for fault detection software module
 */
void fault_init(void);
/**
 * Checks hardware signals noting component failure.
 * @return true if a module has failed, false if module
 */
bool fault_check_for_failure(void);
/**
 * Generates the fault code associated to the components failed.
 * result is a bit field that follows form
 * b8 : 0
 * b7 : 0
 * b6 : 0
 * b5 : 0
 * b4 : 0
 * b3 : 1 if input failed else 0
 * b2 : 1 if motor failed else 0
 * b1 : 1 if heartbeat failed else 0
 * @return error code as detailed above
 */
uint8_t fault_code(void);
/**
 * Signals to the fault module whether to supress watchdog expiration even if
 * component has failed. Ideally used when main loop is interacting with the
 * motor, which needs to be slowly backed off.
* @param s is boolean signaling to module whether if in critical section
 */
void fault_in_critical_section(bool s);

#endif
```

Code List 21: Header file for fault module.

```
/**
 * Fault
 * @author Curt Henrichs
 * @date 10-17-17
 *
 * Fault detection module will determine if any component have failed. If
 * failure occured then the watchdog epxire unless system has entered a critical
 * section, where then the main loop takes control of the watchdog.
 */

//===============================================================================
//                                  Libraries
//===============================================================================

#include "Fault.h"
#include <project.h>
```

```c
//==============================================================================
//                          Interrupt Handler Prototypes
//==============================================================================

/**
 * Fault Detection ISR triggered on watchdog feed timer overflow. ISR will
 * check for all possible faults and if a fault occurs, it will cause the
 * system to reset.
 */
CY_ISR_PROTO(fault_detection_isr);

//==============================================================================
//                              Private Attributes
//==============================================================================

/**
 * Flag to signal main program has entered a critical section where watchdog
 * should not cause immediate reset.
 */
static volatile bool _critical_section = false;

//==============================================================================
//                          Public Function Implementation
//==============================================================================

/**
 * Initialization function for fault detection software module
 */
void fault_init(void){
    //start watchdog timer
    Watchdog_Timer_Overflow_Int_StartEx(fault_detection_isr);
    Watchdog_Timer_Start();

    //start watchdog
    CyWdtStart(CYWDT_128_TICKS,CYWDT_LPMODE_NOCHANGE);
}

/**
 * Checks hardware signals noting component failure.
 * @return true if a module has failed, false if module
 */
bool fault_check_for_failure(void){
    return  !(Heartbeat_COP_Read() && Motor_COP_Read() && UserInput_COP_Read());
}

/**
 * Generates the fault code associated to the components failed.
 * result is a bit field that follows form
 * b8 : 0
 * b7 : 0
 * b6 : 0
 * b5 : 0
 * b4 : 0
 * b3 : 1 if input failed else 0
 * b2 : 1 if motor failed else 0
 * b1 : 1 if heartbeat failed else 0
 * @return error code as detailed above
 */
uint8_t fault_code(void){
    uint8_t code = 0x00;
    code += (!Heartbeat_COP_Read()) << 0;
    code += (!Motor_COP_Read()) << 1;
    code += (!UserInput_COP_Read()) << 2;
    return code;
}

/**
 * Signals to the fault module whether to supress watchdog expiration even if
 * component has failed. Ideally used when main loop is interacting with the
 * motor, which needs to be slowly backed off.
* @param s is boolean signaling to module whether if in critical section
 */
void fault_in_critical_section(bool s){
    _critical_section = s;
}

//==============================================================================
//                       Interrupt Handler Implementation
//==============================================================================

/**
 * Fault Detection ISR triggered on watchdog feed timer overflow. ISR will
 * check for all possible faults and if a fault occurs, it will cause the
 * system to reset.
 */
CY_ISR(fault_detection_isr){

    //feed watchdog only if no faults exist
    if(!fault_check_for_failure() || _critical_section){
        CyWdtClear();
    }

    //clear source of interrupt
    Watchdog_Timer_ReadStatusRegister();
    Watchdog_Timer_Overflow_Int_ClearPending();
}
```

Code List 22: Implementation file for fault module.