Development of the Codebase for ECE Design: Smartwatch Device and App for Continuous Glucose Monitoring

by

Keelin Becker-Wheeler

Computer Engineering

Supervisor: Prof. John Chandy

An honors thesis submitted to the

University of Connecticut in partial fulfillment

of the requirements for graduation

with honors for the degree of

Bachelor of Science in Engineering

Department of Electrical and Computer Engineering

2018

Becker-Wheeler, Keelin (B.S.E., Computer Engineering)

Development of the Codebase for ECE Design: Smartwatch Device and App for Continuous Glucose Monitoring

Supervisor: Prof. John Chandy

Abstract – This honors thesis describes the development of the codebase for the 2018 University of Connecticut ECE Senior Design project titled "Smartwatch Device and App for Continuous Glucose Monitoring". The goal for the senior design project is to remove the dependencies imposed by a previously-designed prototype of a glucose-monitoring smartwatch, by creating a new custom device without those dependencies. Development of the new codebase involves splitting the tasks of an Arduino codebase into a number of logical components that are then implemented in C. This new codebase makes substantial improvements to the overall program structure and organization, as well as numerous functional additions. An API for writing graphics to a given display is written and implemented for two LCD controllers as well as for the standard output stream. This graphics API uses a buffering strategy and partial screen rendering to achieve faster display writing. Functions that plot incoming data on an auto-scaling bar graph are implemented as well. The resulting codebase realizes the firmware for a custom smartwatch controlled by an ATSAMD21G18A microcontroller. This smartwatch is designed to periodically read the frequency of an incoming signal to determine and display the current glucose level of a user. The program supports Bluetooth communication with a smartphone app and a deep sleep mode to save battery during inactivity. Current time and date are kept and displayed to the user, as well as battery and Bluetooth connection information. A user interacts with the smartwatch using two buttons.

Acknowledgements

This honors thesis is completed in parallel with the 2018 University of Connecticut ECE Senior Design project titled "Smartwatch Device and App for Continuous Glucose Monitoring". The senior design project is completed as a group effort between Alexander Valdes, Richard Mullen, and myself as well as a Mechanical Engineering team consisting of Chloe Vollaro and Danielle Browning. The senior design project is sponsored by Biorasis, with the advising support of Allen Legassey. The faculty advisor for the senior design project is also the supervisor for this honors thesis; that person is Professor John Chandy.

Contents

Chapter

Figures

Figure

Chapter 1

Introduction

The "Smartwatch Device and App for Continuous Glucose Monitoring" senior design project[8] is sponsored by Biorasis[1], located in Storrs, Connecticut. The company goal is to develop a smartwatch-like device that communicates with an implantable, glucose-monitoring device created by the company. The senior design project is a continuation of a project[7] completed last year that developed a system and working prototype for a smartwatch-like device as described.

The goals for the senior design team this year are to remove the dependencies imposed by last year's prototype and to create a custom device with a larger screen and added visualization of graphed data. Described in this honors thesis will be the process for developing the new codebase for this system.

1.1 Inherited Codebase

The codebase inherited from last year's project is heavily based on the TinyScreen SmartWatch Tutorial by TinyCircuits[20], but uses the Tinyscreen+[5] for the microcontroller and display. The firmware is written in Arduino, a simple platform for writing C++ code for embedded devices. The inherited project includes 41 files inside a single directory. Of these files, 14 files involve C++ code for a Kalman filter and were autogenerated from MATLAB, 24 files are C++ code files from an Arduino Bluetooth library included with the TinyScreen SmartWatch Tutorial, 2 files are C++ files defining a custom frequency analysis function, and the last file is the .ino Arduino source file for the project. These files and their relationships are graphed in Figure 1.1. It is made clear from

the depicted dependency graph that 17 of the files have no impact on the smartwatch firmware and can be safely removed. Of the remaining 24 files, 23 come from the external Bluetooth library. This means that the remaining .ino file represents the majority of the firmware code.

Having the majority of the firmware localized to a single file creates non-modular code[1] . The logical sections of this file should be determined and used to create distinct program files that would increase organization and readability of the codebase. The logical sections of this file are determined to be as follows:

- Code related to reading the current battery level.
- Code related to handling Bluetooth messages and the Bluetooth library.
- Code related to handling external button presses.
- Code related to managing the current time and timers.
- Code related to writing to the display.
- Code related to taking a measurement from the implantable biosensor.
- Code responsible for interconnecting the above into a single project.

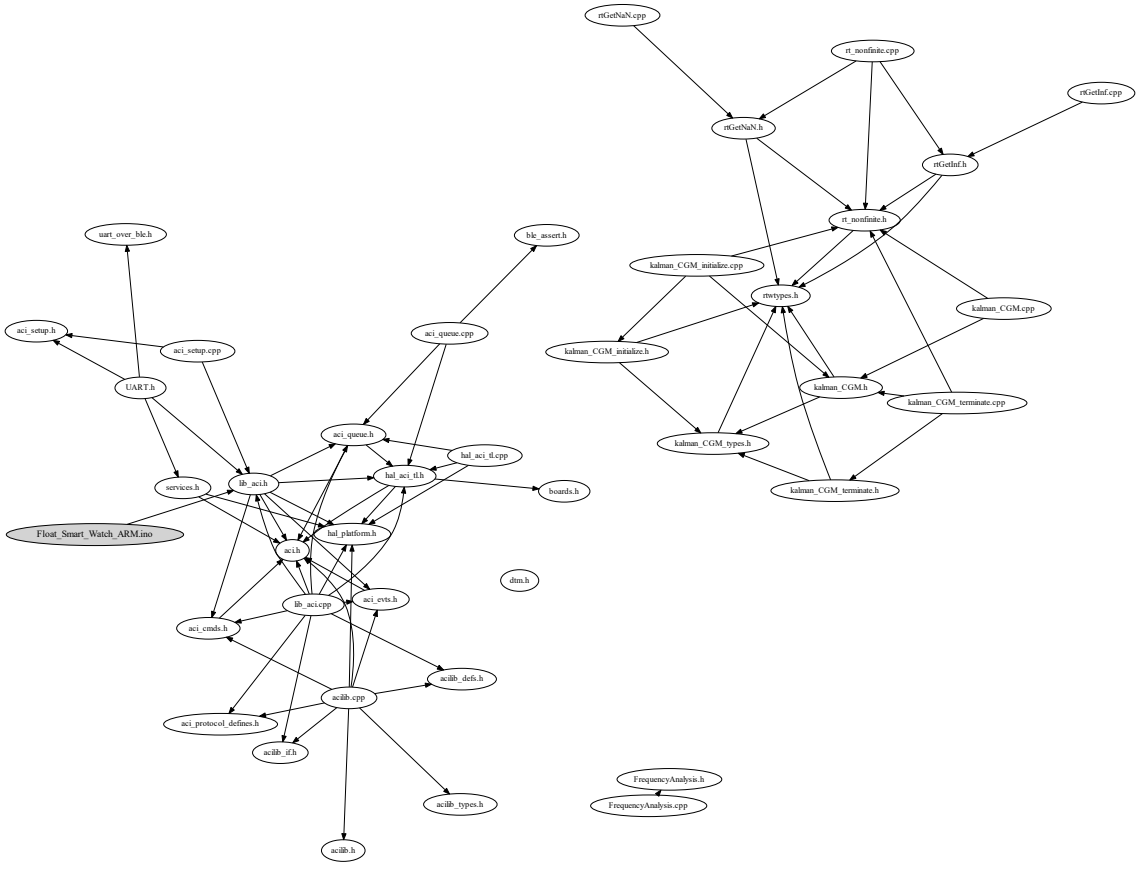
The new codebase will implement these logical sections in order to rewrite the firmware of the smartwatch in a more modular fashion.

1.2 Atmel Advanced Software Framework

The dependency on the Tinyscreen+ board and associated Arduino platform needs to be removed from the inherited smartwatch design, to give the sponsor company more control over the hardware of their product and produce devices at a cheaper price. In replacement of Arduino, the Atmel Advanced Software Framework (ASF)[11] is used as a platform for the new codebase. The decision to use ASF libraries rather than the Arduino platform comes partially from the fact that Arduino hides a majority of control from the programmer. For instance, in an Arduino project on a Windows machine, all of the header files that directly control the embedded device are located in

[1] In computer science, there is a concept called "separation of concerns" that says a computer program should be separated into distinct sections that each handle smaller portions of the code. Such a well-organized program is called a modular program[4].

Figure 1.1: A graph of the dependencies within the inherited codebase shows that 17 files are completely separated from the main Arduino source file, Float_Smart_Watch_ARM.ino. As a result, these files may be removed from the project without consequence.



the AppData folder, which is hidden from the user and not accessible by normal means. In contrast, a project created using ASF libraries would place all related header files into subdirectories that can be easily located from the project directory. A decision to use ASF thus results in a more complete and localized codebase. Additionally, Arduino is written in C++, whereas the C programming language used by ASF libraries is better suited for embedded applications[2] .

The Atmel Advanced Software Framework offers easier control of a target microcontroller (MCU[3]) by providing an abstraction to the MCU hardware. In order to reduce the size of the codebase, certain hardware abstractions can be enabled or disabled through ASF. For simplicity, the MCU used in the new codebase is the same as the MCU used in the inherited codebase: the ATSAMD21G18A[12]. The following components of this MCU are made available through ASF to the codebase:

- Sleep manager (service)
- Delay routines (service) [systick]
- ADC - Analog-to-Digital Converter (driver) [callback]
- EVSYS - Event System (driver) [polled]
- EXTINT - External Interrupt (driver) [callback]
- PORT - GPIO Pin Control (driver)
- RTC - Real Time Counter Driver (driver) [calendar_callback]
- SERCOM SPI - Serial Peripheral Interface (driver) [callback]
- SYSTEM - Core System Driver (driver)
- TC - Timer Counter (driver) [callback]
- TCC - Timer Counter for Control Applications (driver) [callback]

In order to increase parallelism in the smartwatch, the majority of the MCU hardware components use callbacks, which are called in response to hardware interrupts.

[2] This claim is made by personal preference.
[3] Microcontroller unit.

1.3 Smartphone Application

A smartphone app is also provided by last year's project, which is written in Android Studio and is modified from the Android application included with the TinyScreen SmartWatch Tutorial. The smartphone app is not heavily modified from last year's version for the new codebase. The only improvements made to it are additions to the permissions and setup of the application, in order to allow it to run on more modern android devices[4] .

The basic functions of the smartphone app include displaying the glucose measurements received from the smartwatch, sending notifications to the smartwatch, and sending commands to the smartwatch that set the current time and measurement period. The smartphone app stores received data in a .csv file that it creates within the phone memory. The smartphone app requests from the Android phone that it has permission to discover and pair with Bluetooth devices, connect to paired Bluetooth devices, write to an external storage, access the approximate location of the smartphone, and bind to a notification listener service. The minimum Android version configured for the smartphone app is Android Jelly Bean 4.3.x, API level 18. The target Android version configured for the smartphone app is Android Nougat 7.1, API level 25.

[4] The inherited Android application was based on Android application code from 2014, and desperately needed updating for support in modern phones.

Chapter 2

Developing The Code

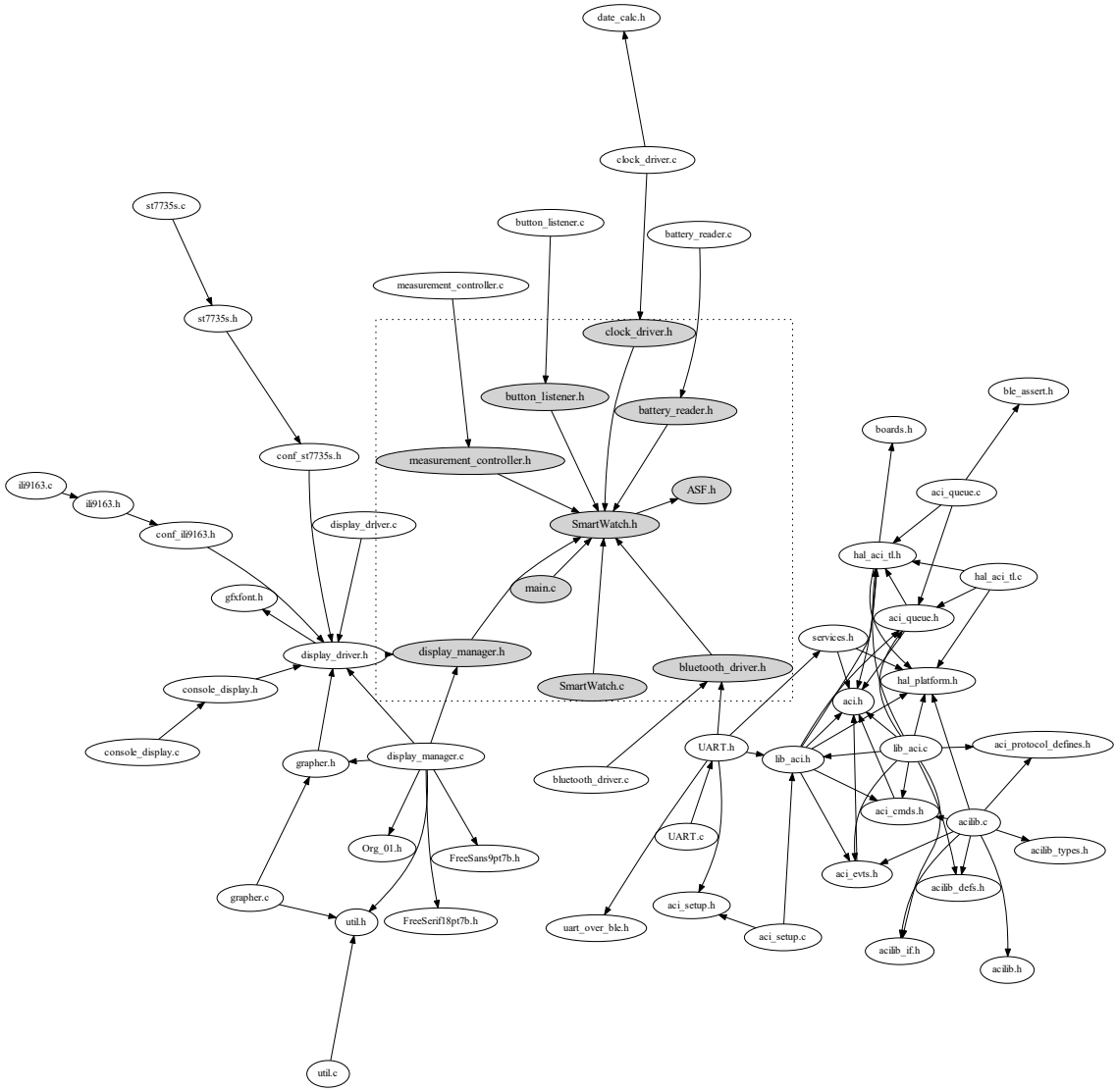
The new codebase is built in C from the bottom up and follows the basic operation of the old codebase while improving on its performance, organization, and functionality. This newly built codebase splits the tasks of the inherited codebase into a number of logical components. These components are grouped as follows:

- Smartwatch Core - Manages all other components.
- Display Manager - Deals with writing to the display.
- Clock Driver - Controls RTC[1] and system timers.
- Bluetooth Driver - Controls Bluetooth interface and protocols.
- Battery Reader - Checks the battery level of the system.
- Measurement Controller - Handles the glucose measurement sequence.
- Button Listener - Records user button presses.

The relationships between all files and components in the new codebase are shown in Figure 2.1. In this graph of relationships, it is clear to see that each component branches out from the Smartwatch.h header file, representing the center of the Smartwatch Core. Notice also that modified versions of the Bluetooth library files from the inherited codebase remain, with an additional UART.c file created to provide a better interface to the library.

[1] Real time clock.

Figure 2.1: A graph of the dependencies within the newly created codebase. All components branch out from the Smartwatch.h header file. On the bottom right, a web of dependencies is seen created from the Bluetooth driver code and associated libraries. On the bottom left, a similar web is shown from the display manager code. The other, simpler components branch linearly from the center.



2.1 Smartwatch Core

The smartwatch core is responsible for bringing together all other components of the smartwatch. This core is contained in these files: main.c, Smartwatch.h, and Smartwatch.c. The main function of the newly developed firmware, shown in S.1 of Appendix A, follows these logical steps:

(1) Initialize the system.
(2) Check if the smartwatch is active:
    (a) If it is active:
        (i) Run the smartwatch tasks.
    (b) If it is not active:
        (i) Enter deep sleep mode.
        (ii) Wake up from deep sleep mode on system interrupt.
(3) Repeat from number 2.

When checking whether the smartwatch is active or not, the core checks that a measurement sequence is not in progress, that a Bluetooth message is not being transmitted or received, that the battery level is not currently being read, and that a button input was not received recently. When all these conditions are met, the system is determined to be inactive and deep sleep is entered to conserve battery. Deep sleep will not be exited until a button interrupt or RTC alarm interrupt is detected. A button interrupt indicates that a user wants to view data on the screen, so the display will be woken up as well. On the other hand, when an RTC alarm interrupt occurs a new measurement will be taken by the device, but the display will not be woken up. If any of the conditions for deep sleep are not met then the system must still be active and the associated smartwatch tasks are performed.

The smartwatch task sequence code, as shown in S.2 of Appendix A, always calls the Bluetooth and measurement tasks, but will only call the battery and display tasks when the display is activated. The display will be activated when there has been a recent button press and will be deactivated after a preset time, as long as no button presses are read during that time. The battery task only needs called when the display is activated because the battery level is only used for updating the battery level indicator on the screen. A future improvement could be to use the battery

level to enter deep sleep mode and prevent wake-up while the battery is at a critical level. Such an improvement would require that the battery task be called at every iteration of the smartwatch task, slightly increasing power usage as the ADC[2] used by the battery reader would need activated more often.

2.2 Display Manager

The display manager is responsible for writing to the display and managing the state of the smartwatch visual interface. It is a larger library consisting of a display driver that provides an API[3] for writing to a given display, a grapher that manages an auto-scaling bar graph of the incoming data, and an edited version of the GFX font library[9] that makes it easier to draw text on the screen. The display manager also relies an a utility file package, consisting of the files util.h and util.c, that implements useful functions for converting numbers into strings[4] .

2.2.1 Display Task

The central files of the display manager are: display_manager.h and display_manager.c. The display manager task function, which is called in the main smartwatch task and is shown in S.3 of Appendix A, begins by drawing the header. The header is composed of the current time, battery level, Bluetooth connection status, and date. These portions of the header are designed to be reorderable. This is accomplished by having each header section write to the screen[5] from the left of a given offset position, then return a new offset from which the next header section should begin drawing. After the header is drawn, the window associated with the current state is drawn.

The display manager uses a basic state machine for keeping the state of the current smart-

[2] Analog to digital converter.
[3] Application programming interface.
[4] The functions ftoa[17] and itoa[14].
[5] I have been using 'display' to reference the entirety of the display hardware and 'screen' to reference the graphics of the display.

watch window, as shown in Figure 2.2. At the bottom of each window, text is drawn at either side of the screen that describes the action that would be performed when a user presses the button at the associated side of the smartwatch. The three windows that are implemented are a main window, a notification window, and a graph window. The main window displays the most recent glucose reading and a message about whether there are any notifications. The notification window displays the most recent notifications received by the Bluetooth driver, if there are any. The graph window displays a bar graph representing the most recent measurement data.

2.2.2 Grapher

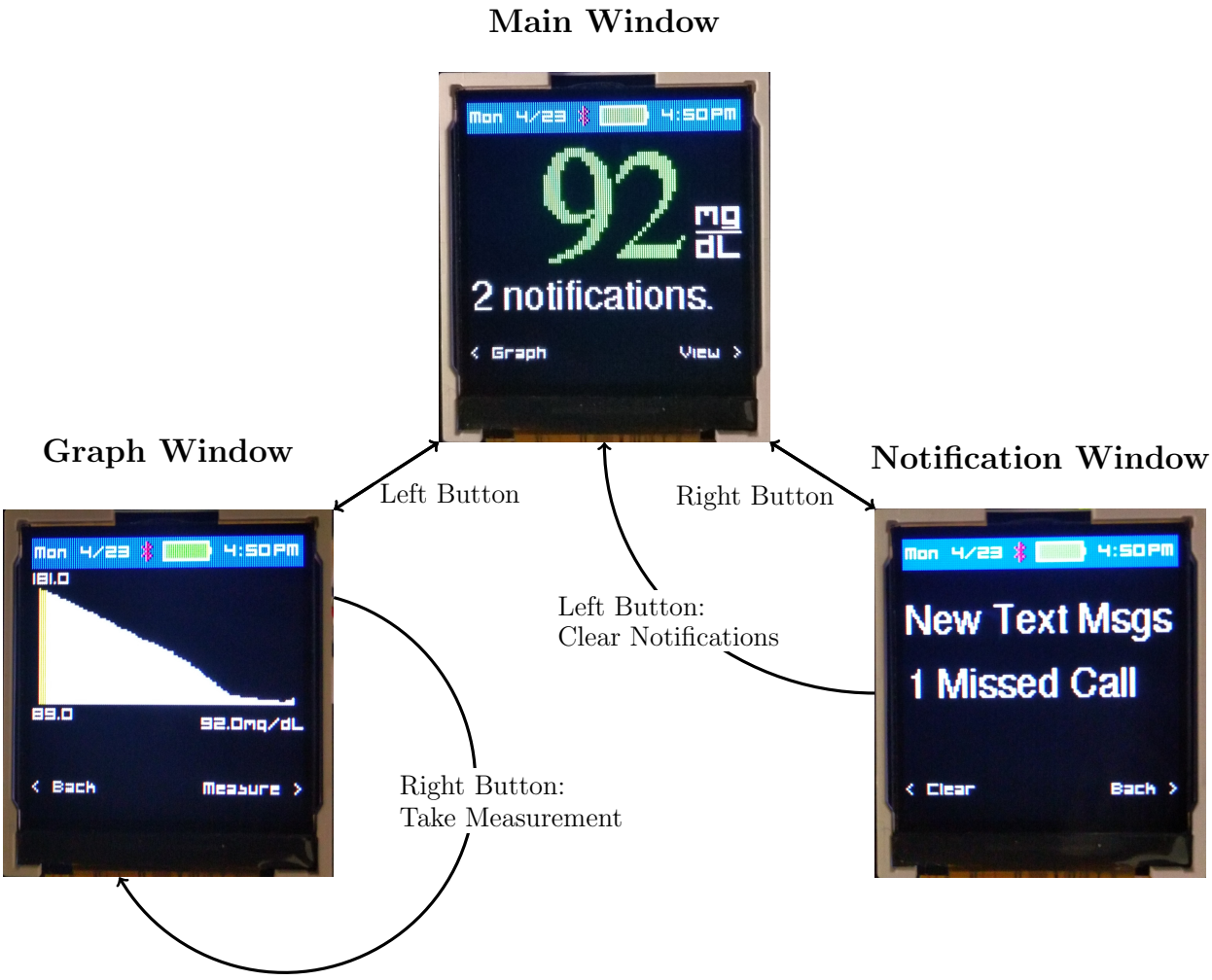
The files that make up the grapher, which manages the bar graph displayed in the graph window, are: grapher.h and grapher.c. The grapher keeps a set of data points and an associated set of precomputed bar heights in two circular buffers[6] . When a new data point is received by the grapher, the grapher will determine if the minimum or maximum values of the graph have changed and then precompute the bar heights for the data points that need updated. The formula for computing the bar height of a given data point is given in the following equation.

$$\text{bar\_height} = \frac{\text{data\_point} - \text{data\_point}_{\text{min}}}{\text{data\_point}_{\text{max}} - \text{data\_point}_{\text{min}}}(\text{bar\_height}_{\text{max}} - \text{bar\_height}_{\text{min}}) + \text{bar\_height}_{\text{min}} \quad (2.1)$$

The grapher provides two functions for drawing the graph. The first function assumes that it must clear the area of the screen itself and so is unused by the firmware, as writing to every pixel within the bounds of the graph will slow down the display writing process. This first, naïve function is shown in S.4 of Appendix A. In order to speed up the writing process, a second function is provided, which is shown in S.5 of Appendix A. The second function assumes that pixels do not need to be cleared by the grapher, rather it is the responsibility of the caller to clear the screen before the graph is drawn. This function provides a hint to the caller about where in the screen

[6] A circular buffer is an array that, when overfilled, will begin overwriting itself from its beginning.[3]

Figure 2.2: A state diagram showing the actions performed by user button presses in each of the viewing windows. At the bottom of each window is text at either side of the screen indicating the action to be performed by the button on the associated side. The images of the display look dithered and faded, due to the quality of the camera and reflections on the screen while the physical display was pictured. The data used to generate the content of the pictured display are simulated.



the graph was drawn by returning a bounding box of the drawn graph. Every pixel within the bounds of the graph would still need drawn for a full set of data, as the bounding box returned from the function will be the same as the bounds of the drawable graph area; however, the total performance improvement seen by allowing the caller to manage the erasure of the graph is still enough to make this second function better than the first.

2.2.3 Display Driver

The display driver provides an API for writing graphics to a given display. Its main files are: display_driver.h and display_driver.c. The display driver provides simple functions that draw pixels, lines, rectangles, and text in up to 4096 different colors to the screen. A buffering strategy is used to improve speed, meaning that when graphics data is to be written to the display, it is first placed in a buffer whose size is equal to the dimensions of the target screen. When the graphics data is ready to be written, this buffer is then copied to the target screen. This buffering strategy provides visually smoother drawing than if each pixel was individually drawn to the screen. Individually drawing each pixel to the screen would create a slight delay between the calculation and drawing of each pixel. By using a buffer, the pixels to be drawn can be calculated beforehand and no visual delay would be seen resulting from these calculations.

A problem with the buffering strategy is that a large region of memory is needed for storing the buffer. For instance, if a single pixel is represented using 18-bits and the dimensions of the screen were 200×200 pixels, then $200 \times 200 \times 18 = 720000$-bits would be needed to store all the information. Additionally, since data is usually stored in multiples of 8-bits, a naïve structure for storing this example data could use $200 \times 200 \times 32 = 1280000$-bits – which is about 156KB. Since the MCU only has 32KB total SRAM[7] , such a structure would be impossible to store in SRAM alone. One solution for reducing the size of the buffer is to store only a couple bits to represent a pixel. If it is known that 4 colors are used by a display, then one can simply store each pixel as a number within 0 to 3 – that is, as a 2-bit number. When it is time to draw the pixel, it can

[7] Static random-access memory.

be converted to the necessary number of bits before being written to the display. Continuing with the example above, the resulting storage would need at least $200 \times 200 \times 2 = 80000$-bits, about 10KB of data. Another solution may be to store the buffer in flash memory, of which the MCU provides 256KB, or to use external memory. All of these solutions share the disadvantage that they would result in a slightly slower display writing process. Due to this disadvantage and the fact that the screen used by the smartwatch is sufficiently small, none of these solutions are used by the display driver. These solutions are left as possible future improvements however, should future modifications of the smartwatch judge them necessary.

The solution used by the display driver for reducing the size of the buffer is to use the minimum pixel resolution of the target display and to store data in the buffer contiguously. With a 128×128-pixel target display screen at a minimum resolution of 12-bits per pixel, the resulting buffer would be $128 \times 128 \times 12 = 196608$-bits, or 24KB. Due to the relatively small size of the rest of the codebase, this large block of memory is able to fit into SRAM and no other solution is needed. Still, remember that data is usually stored in multiples of 8-bits, so the problem of storing each of these 12-bits remains. To store the pixel data, the display driver splits two pixels across three groups of 8-bits. The resulting structure[8] is an array of 8-bit elements with $\frac{3}{2} \times 128 \times 128$ elements, where a simple mapping of pixel positions to array positions is used for determining where a pixel is to be written into the array. This mapping is given in the following equations:

$$\text{array\_position} = \text{array\_x} + (\frac{3}{2} \times \text{pixel\_y} \times \text{display\_width}) \tag{2.2}$$

$$\text{array\_x} = \begin{cases} \frac{3}{2} \times \text{pixel\_x} & ; \quad \text{pixel\_x is even} \\ \frac{3}{2}(\text{pixel\_x} - 1) + 1 & ; \quad \text{pixel\_x is odd} \end{cases} \tag{2.3}$$

In addition to a buffering strategy, the display driver uses partial screen rendering to increase speed. This means that as the display driver draws pixels, it keeps track of the furthest pixels it has drawn. Using a bounding box containing these pixels, the display driver can update the screen

[8] An alternative structure could be created using the C language concept of bit fields[2].

with only the pixels within the box. This significantly increases the speed of the display write process when the updated pixels are localized to certain portions of the screen, as time will not be spent on drawing pixels that were not updated. The code that achieves writes to the buffer and updates the bounds for partial screen rendering is shown in S.6 of Appendix A.

The display driver also uses a concept called graphics grouping[9] , which complements the partial screen rendering strategy and helps simplify the process for removing and updating graphics. Graphics grouping simply stores the bounding box of graphics to be drawn, so that the driver knows what region needs updating when the graphics update or need removed. In the display driver, each graphics group is given a unique ID. When a graphics group is written to in the display driver, the previous graphics group with the same ID provides a bounding box that is then cleared in the buffer. After the old bounding box has been cleared, a new bounding box is stored for the new graphics that were just written. The result is that the display driver will automatically clear the old data of a graphics group whenever that graphics group is written to. Since the display driver uses a buffering strategy, any delays imposed by this grouping process of removing and writing to the same area of the buffer will not result in visual delays on the screen. The code for writing a string to a graphics group is shown in S.7 of Appendix A.

The display manager uses the graphics grouping capabilities of the display driver extensively, and optimizes the process further by committing the buffer to the physical display following the drawing of most groups. Additionally, the display manager will only call the display driver to draw to the screen when the content of the screen will be updated, opposed to on every main loop iteration. This saves further time as the display driver will only need to overwrite its buffer in cases where the data of the buffer will actually change. Example usage of graphics grouping by the display manager is given in the main window drawing command, shown in S.8 of Appendix A.

Finally, as mentioned, the display driver provides an API for writing to any given display. Implementations for two LCD[10] controllers are written, as well as an implementation for the

[9] This concept is named so by me.
[10] Liquid crystal display.

standard output stream, which is used to aid debugging. These LCD controllers are the ST7735S[19] and ILI9163[15], which are supported for use with 12-bit color resolution only. The controllers are written for an SPI[11] connection, but implementations for parallel connections can be written without changing the display driver code. Support for more color resolutions and larger screen dimensions are left as future improvements that can be made to the display driver.

2.3 Clock Driver

The clock driver is responsible for managing the current time and the various system timers. It is contained in these files: clock_driver.h, clock_driver.c, and date_calc.h. The target MCU provides five 16-bit timers, three 24-bit timers, and a 32-bit RTC. A set of two 16-bit timers can be cascaded by the hardware into a 32-bit timer. The RTC is configured to run at 32.768KHz, in order to generate a precise one second period. The other timers that are used by the smartwatch are configured to run at 8MHz, the same as the program clock speed of the MCU. Higher frequencies can be configured, but are not used as they do not provide enough performance improvement to offset their extra power requirements. The only hardware component where the extra performance of a higher frequency clock should be used is in the display SPI, which runs at 48MHz. The display SPI is configured to use a higher clock frequency than the rest of the smartwatch in order to speed up the display writing process.

2.3.1 Real Time Clock

The clock driver uses the RTC to keep the current time. The default time and date are automatically set to the current time and date during compiling, but can be reset using a Bluetooth command should the time or date need correcting. An RTC alarm interrupt is configured as well, which will wake up the smartwatch if needed and then trigger a new glucose measurement. After

[11] Serial peripheral interface.

each RTC alarm interrupt, a new alarm is configured for the next scheduled reading. This periodic measurement can be disabled, by setting the alarm period to zero. The alarm code for the RTC is shown in S.9 of Appendix A.

The clock driver relies on the date_calc.h utility file to accurately follow the Gregorian calendar. This utility file operates on Gregorian dates using algorithms described by Gary Katch[16]. Whenever the current time updates and a new day begins, the next date is calculated using these algorithms and the current date is updated. Additionally, each time the current second changes, the display manager will blink the time delimiter[12] of the displayed time, similarly to most other digital clocks.

2.3.2 Timers

The clock driver manages four timers to be used by the smartwatch. These timers are a screen timer, pulse timer, battery timer, and button timer. The button timer is configured to trigger an interrupt every millisecond, while the other timers trigger separate interrupts each second. When an interrupt is triggered for a timer, an associated counter is decremented. When that counter reaches zero, the timer is disabled and a timeout flag is raised for that timer. The clock driver provides functions for resetting the timeout of each timer, which will set the counter appropriately and re-enable the timer. Each timer follows the same basic code structure. The code used for running the screen timer is shown in S.10 of Appendix A.

The screen timer is reset by the smartwatch core each time the display is activated, either by a button interrupt or software request. While the counter for this timer has not reached zero, the display will be determined as active and will be powered. When the counter reaches zero, the smartwatch core will deactivate the display and place it into sleep mode.

The pulse timer is used by the measurement controller while it is taking a measurement. It is used for allowing the implant time to power up and settle on a value, as well as for taking multiple data points over a period of time in order to find an average measurement.

[12] The colon between the hour and minute of a written time, i.e. in 12:34.

The battery timer is used by the smartwatch core to periodically read the battery level through the battery reader. Additionally, the smartwatch core will set the battery counter to zero in order to trigger the battery reader each time the display is reactivated, so that the displayed battery level always starts at the most recent value.

The button timer is used by the button listener for debouncing the smartwatch buttons. It is the only timer that counts in milliseconds, which allows the buttons to be debounced at a speed that does not conflict with additional button presses from a user.

2.4 Bluetooth Driver

The Bluetooth driver is responsible for handling messages received over Bluetooth and interacting with the ACI[13] BLE[14] library. The core files of the Bluetooth driver are: bluetooth_driver.h and bluetooth_driver.c. These files parse received Bluetooth messages and perform certain actions based on each message. Additionally, when a device is connected to the smartwatch through Bluetooth, the Bluetooth driver will send to that device a message containing the most recent glucose measurement.

2.4.1 Messaging Protocol

A message received by the Bluetooth driver from a connected device, such as a smartphone running the corresponding app, will trigger an action when the message begins with a certain letter. The subsequent characters of the message are then used as parameters for that action. While it is easy to add additional commands and actions, should they be needed in future editions of the smartwatch, the currently supported actions are as follows:

- '1': Sets the 1st notification to the subsequent text.

[13] Application controller interface.
[14] Bluetooth low energy.

- '2': Sets the 2nd notification to the subsequent text.
- 'R': Sets the period for the periodic measurement to the subsequent number of seconds.
- 'D': Sets the date and time to the subsequent text, formatted: yyyy mm dd hh mm ss. [15]

Notice that it is the responsibility of the connected Bluetooth device to construct notifications to be displayed on the smartwatch. The Bluetooth driver will simply set the associated notification in the smartwatch to the received text, up to a length limit. On receipt of a notification, the Bluetooth driver will also set a flag to indicate that there are notifications to be displayed. The display manager uses this information while displaying the number and content of notifications.

The Bluetooth driver keeps a circular buffer of received messages, and will parse each message in the order received. Should a large number of messages be received in a short time, the Bluetooth driver will overwrite the oldest message in the buffer. A future improvement could be to add a field for priority in the message format, which would be used to determine the order for replacing overcrowding messages.

No special command sequence is used when sending data from the smartwatch over Bluetooth, as the smartwatch supports only one type of outgoing command. The Bluetooth driver simply transmits the glucose data found on each new measurement reading. It is the responsibility of the connected device to interpret the data sent to it as a glucose measurement. Should a Bluetooth device not be connected, the Bluetooth driver will wait until a new connection is made before sending the most recent glucose data. The Bluetooth task code is shown in S.11 of Appendix A.

2.4.2 ACI BLE Library

The smartwatch uses the nRF8001[18] to manage its BLE connection and the Bluetooth driver uses the ACI BLE library to interact with it. The ACI BLE library is modified for use with the target MCU from Adafruit's nRF8001 Arduino drivers[10]. The core interface to the ACI BLE Library consists of these files: UART.h and UART.c. Using the ACI BLE library, the Bluetooth driver waits for the nRF8001 to indicate that an event has occurred, then connects to the nRF8001

[15] Year, month, day, hour, minute, then second.

over SPI to receive the event. Once the event and associated parameters have been received, the event is handled based on its type. The supported event types are as follows:

- Device Started Event: Sent every time the device starts.
- Echo Event: Mirrors the last echo command.
- HW Error Event: Sent when a hardware error occurs.
- Response Event: Generic response to a command.
- Connected Event: Link connected.
- Disconnected Event: Link disconnected.
- Bond Status Event: Result of completed pairing.
- Pipe Status Event: Pipe bitmap for available pipes.
- Timing Event: Sent when the radio state changes.
- Data Credit Event: Sent when transmit credits are available.
- Data ACK Event: Acknowledgement to previous command.
- Data Received Event: Sent when new data is available..
- Pipe Error Event: Sent on software error.
- Display Passkey Event: To communicate passwords.
- Security Key Request Event: Sent on password request.

When the nRF8001 is first started, the ACI BLE library attempts to configure it. The nRF8001 is configured using precompiled setup configuration data, which sets device information such as the connection name, timeout limits, and security details. Once the nRF8001 is setup, the ACI BLE library will trigger an attempt to bond with a device. Until a device is connected, the nRF8001 will continuously retrigger a bonding sequence after each timeout. When a device is connected, the Bluetooth driver is notified and sets an appropriate flag. This flag is used by the display manager to render the Bluetooth connection indicator. When a connected device is disconnected or a hardware error occurs, the connection flag is reset and the ACI BLE library triggers another attempt to bond with a device. The nRF8001 acts as a slave device, meaning that it is the responsibility of a Bluetooth enabled device to initiate pairing with the smartwatch.

Similarly, it is the responsibility of a connected device to reconnect with the smartwatch should the connection be lost.

While a Bluetooth device is connected to the smartwatch, data can be transmitted to or from the paired device. When data is received, the ACI BLE library will send the received data to the message buffer in the Bluetooth driver and then trigger a callback in the Bluetooth driver. This callback simply records that a new message was received and increments the next position of the received data buffer. After data is successfully sent from the smartwatch over Bluetooth, the ACI BLE library will trigger a similar callback in the Bluetooth driver to signal that the transmission has completed.

2.5 Battery Reader

The battery reader is responsible for reading the current battery level and is built from these files: battery_reader.h and battery_reader.c. The battery reader uses the ADC of the MCU to measure battery voltage. Voltage taken directly from the battery – which has a maximum voltage of about 4V – is divided by two using a voltage divider circuit, then is fed into the MCU for use as the ADC input. This input is compared to the converted voltage level powering the MCU, which is 3.3V. The battery reading process works because the actual voltage of the battery is higher than the voltage powering the MCU. This means that the difference between the battery voltage and MCU voltage can be measured in order to estimate the battery level of the smartwatch. When the battery has a full voltage, the battery level is read as maxed. When the battery voltage is close to the voltage powering the MCU, the battery level is read as nearly empty. The actual voltage of the battery is divided by two before it is fed to the ADC, meaning that the calculation for the battery level uses 2V as the maximum battery level and 1.65V as the minimum battery level. The battery level is calculated as a percentage of a maximum value, given to the battery reader from the display manager in order to make it easier to draw the battery level indicator. The code that

calculates the battery level is shown in S.12 of Appendix A.

2.6 Measurement Controller

The measurement controller is responsible for taking glucose measurements and is built from these files: measurement_controller.h and measurement_controller.c. The measurement controller stores the most recent glucose value, while previously-measured glucose information is stored in the display manager by the grapher. Additionally, glucose information will be sent over Bluetooth to any connected device, which may store further copies of the glucose information. It is assumed that in order to store the glucose information of real users, the distributors of a glucose-monitoring smartwatch supplied with this firmware have made sufficient steps to follow HIPAA[16] and any other related regulations or laws. This assumption is made also for regulations or laws in countries other than the United States of America, should they apply. It is important to note that this firmware, on its own and in its current state, may not be satisfactory for commercial use without additional security features, such as the encrypting of stored information. Encryption of data and other modifications necessary to produce a commercial edition of this firmware are left as future improvements.

It is important to note that in order to take measurements, the measurement controller only requires that a signal of a certain frequency be given to it. Due to this, the smartwatch can be repurposed for functions other than measuring glucose, so long as the desired data is encoded as a frequency signal.

2.6.1 Glucose-Measuring Implant

The smartwatch is designed to interface with an implantable, glucose-measuring device developed by the sponsor company. This implant sits under the skin and is powered by a solar cell.

[16] Health Insurance Portability and Accountability Act of 1996.[6]

The implant receives power from an array of LEDs[17] that sit at the bottom of the smartwatch. When powered, the implant generates an electric current based on the amount of glucose it detects in its surroundings. The implant will then blink an LED at an appropriate frequency based on the glucose amount. The smartwatch uses an array of photodiodes to detect the transmitted light signal and determine its frequency. A filter sits between the photodiodes and LEDs on the smartwatch in order to filter out all incoming light except the light generated from the implant. The signal frequency measured by the measurement controller is then converted into a glucose value.

2.6.2 Measurement Sequence

The measurement controller begins a measurement sequence by turning on the LED array for a number of seconds, which allows the implant to power on and settle at a glucose value. The measurement controller then triggers a frequency capture event using an external interrupt on the photodiode array pin. This frequency capture event is used to determine the frequency of the received signal. A large number of frequency samples are taken, and the average of these values is recorded. This process is repeated each second for a number of seconds, then the recorded values are averaged into a final frequency value. The final frequency value is then converted into a glucose value using a conversion algorithm. The measurement task code that accomplishes this is shown in S.13 of Appendix A.

2.7 Button Listener

The button listener is responsible for recording button interrupt events and is built from these files: button_listener.h and button_listener.c. The button listener simply initializes an external interrupt for each of the two smartwatch buttons, then will set a flag for the associated button whenever its interrupt occurs. The smartwatch core periodically polls the button listener for recent

[17] Light emitting diodes.

button presses. On receipt of a button read from the smartwatch core, the button listener will clear its button flags and continue waiting for additional button presses.

The button listener configures its external interrupts to be enabled during deep sleep mode, in order to allow the device to be woken up when a user presses a button. In cases where a button press wakes up the device, the button flags will be cleared, so as to not trigger any action usually tied to the buttons while the device is awake.

A timer is used for debouncing the buttons. When a new external interrupt occurs after a button press, a timer will begin counting down from a preset number of milliseconds. Any additional external interrupts that occur before the timer reaches zero are discarded. This prevents a single button press from being read as multiple presses due to the physical behavior of the button. Other implementations for button debouncing are possible, including hardware solutions[13]. These solutions were not used as the timer method for debouncing was found to be adequate during testing. A recommended future improvement to the smartwatch would be the addition of capacitors on the button inputs for a hardware solution to debouncing. Sample code from the button listener is shown in S.14 of Appendix A.

Chapter 3

Conclusions

The new codebase for the 2018 University of Connecticut ECE Senior Design project titled "Smartwatch Device and App for Continuous Glucose Monitoring" successfully realizes the firmware for a custom smartwatch that measures and graphs glucose data as well as performs tasks typically expected of a smartwatch, such as wirelessly communicating with and displaying notifications from a connected Bluetooth device, providing the current time and date, entering a sleep mode during inactivity, and displaying the wireless connection status and current battery level. The resulting codebase is well-structured, as it is separated into a number of logical components with descriptive file names that are organized into subdirectories of a single directory, each component has an interfacing core that provides a single entry point to the component, and each top-level function is written to be as brief and self-descriptive as possible. The codebase is also easily adaptable, as a number of different displays can be implemented without changing the core of the display code, extending the Bluetooth command format is simple, and the measurement capabilities of the smartwatch can be adapted for tasks other than reading glucose.

Future improvements that can be made to the codebase include entering deep sleep mode when the battery is at a critical level, providing display driver options for using different color resolutions and larger screens, implementing priority in the Bluetooth messaging format, encrypting stored data to protect personal user information, and employing a real-time operating system to run the tasks of the smartwatch concurrently.

Bibliography

[1] Biorasis. http://bio-orasis.com/. Accessed: April 10, 2018.

[2] Bit fields. https://en.wikipedia.org/wiki/Bit_field. Accessed: April 17, 2018.

[3] Circular buffer. https://en.wikipedia.org/wiki/Circular_buffer. Accessed: April 17, 2018.

[4] Separation of concerns. https://en.wikipedia.org/wiki/Separation_of_concerns. Accessed: April 10, 2018.

[5] TinyScreen+. https://tinycircuits.com/products/tinyscreenplus. Accessed: April 11, 2018.

[6] Summary of the HIPAA Security Rule. https://www.hhs.gov/hipaa/for-professionals/security/laws-regulations/index.html, 2013. Accessed: April 24, 2018.

[7] ECE Senior Design Team 1714. Smartwatch-like Device and Apps for Continuous Glucose Monitoring. http://ecesd.engr.uconn.edu/ecesd1714/. Accessed: April 10, 2018.

[8] ECE Senior Design Team 1814. Smartwatch Device and App for Continuous Glucose Monitoring. http://ecesd.engr.uconn.edu/ecesd1814/. Accessed: April 10, 2018.

[9] Adafruit. Adafruit GFX Library. https://github.com/adafruit/Adafruit-GFX-Library. Accessed: April 15, 2018.

[10] Adafruit. Drivers for Adafruit's nRF8001 Bluetooth Low Energy Breakout. https://github.com/adafruit/Adafruit_nRF8001. Accessed: April 18, 2018.

[11] Atmel. Advanced Software Framework. http://asf.atmel.com/docs/latest/. Accessed: April 24, 2018.

[12] Atmel. SAM D21E / SAM D21G / SAM D21J Datasheet. https://cdn.sparkfun.com/datasheets/Dev/Arduino/Boards/Atmel-42181-SAM-D21_Datasheet.pdf, 2015. Accessed: April 11, 2018.

[13] Jens Christoffersen. Switch Bounce and How to Deal with It. https://www.allaboutcircuits.com/technical-articles/switch-bounce-how-to-deal-with-it/, 2015. Accessed: April 24, 2018.

[14] GeeksforGeeks. Implement your own itoa(). http://www.geeksforgeeks.org/implement-itoa/. Accessed: April 25, 2018.

[15] Ilitek. a-Si TFT LCD Single Chip Driver 132RGBx162 Resolution and 262K color Datasheet. https://www.rockbox.org/wiki/pub/Main/SonyNWZE370/ILI9163.pdf, 2008. Accessed: April 17, 2018.

[16] Gary Katch. Date calculation algorithms. https://alcor.concordia.ca/~gpkatch/gdate-algorithm.html. Accessed: April 24, 2018.

[17] Stack Overflow. Minimal implementation of sprintf or printf. https://stackoverflow.com/questions/16647278/minimal-implementation-of-sprintf-or-printf. Accessed: April 25, 2018.

[18] Nordic Semiconductor. nRF8001 Single-chip Bluetooth low energy solution Product Specification 1.2. https://cdn.shopify.com/s/files/1/1125/2198/files/nRF8001_PS_v1.2-2797.pdf?14554646480989527344, 2013. Accessed: April 18, 2018.

[19] Sitronix. ST7735 262K Color Single-Chip TFT Controller/Driver. http://www.displayfuture.com/Display/datasheet/controller/ST7735.pdf, 2010. Accessed: April 17, 2018.

[20] TinyCircuits Staff. TinyScreen SmartWatch Tutorial. https://tinycircuits.hackster.io/tbatey_tiny-circuits/tinyscreen-smartwatch-785938, 2016. Accessed: April 10, 2018.

Appendix A

Selected Code Samples

S.1

```
int main(void) {
    init_all();

    for(;;) {
        while (is_active()) {
            smartwatch_task();
        }
        sleep();
        wakeup();
    }
    return 0;
}
```

S.2

```
// Smartwatch core task
void smartwatch_task(void) {
    // Perform smartwatch subtasks:

    // Bluetooth driver task
    bt_task();

    // Measurement controller task
    measurement_task();
    if (is_new_measurement()) {
        updateGraph(get_measurement());
        itoa(get_measurement(), buffer, 4);
        bt_write(buffer, 4);
    }
```

```
    if (is_screen_active()) {
        // (Only need to deal with new battery reads if screen is active)
        // Battery reader task
        battery_task();
        if (!is_battery_active() && is_battery_timeout()) {
            start_battery_read();
            set_battery_timeout(BATTERY_TIMEOUT);
        }

        uint8_t buttons = get_buttons(); // Returns identifier to determine
            ↪ which buttons were pressed
        // Display manager task
        display_ui_task(buttons);

        // We need to set a timeout for screen if it was freshly activated
        if (screen_request || buttons) { // i.e. by explicit request or button
            ↪ press
            screen_request = 0;

            // Wakeup screen if needed
            if (screen_sleep) {
                screen_sleep = 0;
                disp_sleep_disable();
            }

            set_screen_timeout(SCREEN_TIMEOUT);
        }
    } else {
        // Trigger battery timer for next screen activation
        set_battery_timeout(0);

        if (!screen_sleep) {
            display_off();
            screen_sleep = 1;
        }
    }
}
```

S.3

```
// Display manager task
void display_ui_task(uint8_t button) {
    // Display header
    int offset = updateTimeDisplay(DISP_WIDTH);
```

```
    offset = displayBattery(offset);
    offset = updateBLEstatusDisplay(offset);
    updateDateDisplay(-1);

    // Display window
    switch(currentDisplayState) {
        case DISP_STATE_HOME:
            updateMainDisplay(button);
            break;

        case DISP_STATE_GRAPH:
            showGraphView(button);
            break;

        case DISP_STATE_NOTIFICATION:
            viewNotifications(button);
            break;

        default:
            currentDisplayState = DISP_STATE_HOME;
            rewriteMain = 1;
            initScreen();
    }

    if (startup) {
        disp_commit();
        startup = 0;
    }
}
```

S.4

```
void graph(int clear) {
    int i, x, y, index;
    short bar[data_length];

    if (data_start>data_length) return;
    // Fill new structure with precomputed bar heights, in the correct order
    // Rather than to have to worry about the ordering during draw phase
    for (y=0, i=data_start; i<data_length; i++, y++)
        bar[y] = bar_cache[i];
    for (y=data_length-data_start, i=0; i<data_start; i++, y++)
        bar[y] = bar_cache[i];

    if (clear)
```

```
        GRAPH_RESET();
    for (y=MAX_BAR_HEIGHT; y>0; y--) {
        for (x=0; x<GRAPH_WIDTH-(GRAPH_WIDTH%BAR_WIDTH)-data_length*BAR_WIDTH;
            ↪ x++) {
            GRAPH_PIXEL_OFF(x, y);
        }
        for (x=GRAPH_WIDTH-(GRAPH_WIDTH%BAR_WIDTH)-data_length*BAR_WIDTH, i=0;
            ↪ i<GRAPH_WIDTH-(GRAPH_WIDTH%BAR_WIDTH); i++, x++) {
            if ((i/BAR_WIDTH)>=data_length) break;
            index = (data_start+i/BAR_WIDTH)%data_length;
            if (bar[i/BAR_WIDTH]>=y)
                // Color the pixels based on glucose level (whether too high or
                    ↪ too low)
                if (data[index]>=DISP_DANGER_HIGH)
                    GRAPH_PIXEL_DANGER(x, y);
                else if (data[index]>=DISP_WARNING_HIGH)
                    GRAPH_PIXEL_WARNING(x, y);
                else if (data[index]<=DISP_DANGER_LOW)
                    GRAPH_PIXEL_DANGER(x, y);
                else if (data[index]<=DISP_WARNING_LOW)
                    GRAPH_PIXEL_WARNING(x, y);
                else
                    GRAPH_PIXEL_ON(x, y);
            else
                GRAPH_PIXEL_OFF(x, y);
        }
    }
    is_changed = 0;
}
```

S.5

```
void graph_smart_sizing(unsigned short* xret, unsigned short* yret, unsigned
    ↪ short* widthret, unsigned short* heightret) {
    int i, index;
    short x, y;
    short bar[data_length];

    *xret = 0;
    *yret = 0;
    *widthret = 0;
    *heightret = 0;

    if (data_start>data_length) return;
    // Fill new structure with precomputed bar heights, in the correct order
```

```
    // Rather than to have to worry about the ordering during draw phase
    for (y=0, i=data_start; i<data_length; i++, y++)
        bar[y] = bar_cache[i];
    for (y=data_length-data_start, i=0; i<data_start; i++, y++)
        bar[y] = bar_cache[i];

    *xret = DISP_GRAPH_X + GRAPH_WIDTH-(GRAPH_WIDTH%BAR_WIDTH)-data_length*
        ↪ BAR_WIDTH;
    *widthret = GRAPH_WIDTH-(GRAPH_WIDTH%BAR_WIDTH);
    for (x=*xret-DISP_GRAPH_X, i=0; i<*widthret; i++, x++) {
        if ((i/BAR_WIDTH)>=data_length) {
            *widthret = i;
            break;
        }
        for (y=0; y<bar[i/BAR_WIDTH]; y++) {
            index = (data_start+i/BAR_WIDTH)%data_length;
            // Color the pixels based on glucose level (whether too high or too
                ↪ low)
            if (data[index]>=DISP_DANGER_HIGH)
                GRAPH_PIXEL_DANGER(x, y);
            else if (data[index]>=DISP_WARNING_HIGH)
                GRAPH_PIXEL_WARNING(x, y);
            else if (data[index]<=DISP_DANGER_LOW)
                GRAPH_PIXEL_DANGER(x, y);
            else if (data[index]<=DISP_WARNING_LOW)
                GRAPH_PIXEL_WARNING(x, y);
            else
                GRAPH_PIXEL_ON(x, y);
        }
        if (bar[i/BAR_WIDTH] > (*heightret))
            *heightret = bar[i/BAR_WIDTH];
    }

    *yret = DISP_GRAPH_Y+DISP_GRAPH_HEIGHT-1-*heightret;
    *heightret = *heightret + 1;
    is_changed = 0;
}
```

S.6

```
static inline void DISPLAY_DRIVER_WRITE(uint8_t x, uint8_t y, uint16_t color)
    ↪ {
        if (x%2) {
            x = (int)((x-1)*1.5f)+1;
            buffer[x+y*ADJ_WIDTH] = (buffer[x+y*ADJ_WIDTH]&0xf0) | ((color>>8)&0
```

```
            ↪ xf);
            buffer[(x+1)+y*ADJ_WIDTH] = color&0xff;
        } else {
            x = (int)(x*1.5f);
            buffer[x+y*ADJ_WIDTH] = (color>>4)&0xff;
            buffer[(x+1)+y*ADJ_WIDTH] = ((color&0xf)<<4) | (buffer[(x+1)+y*
            ↪ ADJ_WIDTH]&0xf);
        }
}

void disp_write_pixel_at(uint8_t x, uint8_t y, uint16_t color) {
    if (x<DISP_WIDTH && y<DISP_HEIGHT) {
        // Write pixel to the buffer, at given screen position
        DISPLAY_DRIVER_WRITE(x, y, color);

        // Compute bounds for partial screen rendering
        if (new_write) {
            new_write = 0;
            leftx = x;
            rightx = x;
            topy = y;
            bottomy = y;
        } else {
            if (x < leftx)
                leftx = x;
            else if (x > rightx)
                rightx = x;
            if (y < topy)
                topy = y;
            else if (y > bottomy)
                bottomy = y;
        }
    }
}
```

S.7

```
void disp_write_str_group(const char *str, uint8_t group_id) {
    if (group_id>0 && group_id<=MAX_WRITE_ID) {
        uint8_t x, y, w, h;
        disp_get_text_bounds(str, cursor_x, cursor_y, &x, &y, &w, &h);

        // Clear previous group if it exists
        if (!keep_group && lastwidth[group_id-1] > 0) {
            disp_fill_rect(lastx[group_id-1], lasty[group_id-1], lastwidth[
```

```
                ↪ group_id-1], lastheight[group_id-1], textbgcolor);
        }

        // Update bounds of current group
        if (keep_group && lastwidth[group_id-1] > 0) { // Assumes a single line
            if (y < lasty[group_id-1]) {
                if (lasty[group_id-1]+lastheight[group_id-1] >= y+h) {
                    lastheight[group_id-1] = lasty[group_id-1]-y+lastheight[
                        ↪ group_id-1];
                } else {
                    lastheight[group_id-1] = h;
                }
                lasty[group_id-1] = y;
            } else if (lasty[group_id-1]+lastheight[group_id-1] < y+h) {
                lastheight[group_id-1] = y-lasty[group_id-1]+h;
            }
            if (x > lastx[group_id-1]) {
                lastwidth[group_id-1] = x - lastx[group_id-1] + w;
            } else {
                lastwidth[group_id-1] += w;
            }
        } else {
            lastx[group_id-1] = x;
            lasty[group_id-1] = y;
            lastwidth[group_id-1] = w;
            lastheight[group_id-1] = h;
        }
    }
    keep_group = 1;
    disp_write_str(str);
}
```

S.8

```
static void updateMainDisplay(uint8_t button) {
    disp_set_color(DISP_PIXEL_WHITE, DISP_PIXEL_BLACK);
    // Make sure the state of the window is set to the main window state
    if (startup) {
        currentDisplayState = DISP_STATE_HOME;
        rewriteMain = 1;
        initScreen();
    } else if (currentDisplayState != DISP_STATE_HOME) {
        currentDisplayState = DISP_STATE_HOME;
        rewriteMain = 1;
        // Assuming button IDs are only IDs greater than graph ID
```

```
    for (int i = 0; i <= GRAPH_ID; i++) {
        // Remove groups from previous window
        // (But not button text, to optimize performance)
        if (last_drawn[i]) disp_remove_str_group(i);
        last_drawn[i] = 0;
    }
}

// Update state of window on button presses
if (button & VIEW_BUTTON) {
    viewNotifications(0);
} else if (button & GRAPH_BUTTON) {
    showGraphView(0);

} else {
    // Draw the main window, only if it needs redrawing due to an update

    if (rewriteMain || newGlucose) {
        // Draw glucose display
        updateGlucoseDisplay(lastGlucoseVal);
        newGlucose = 0;
    }
    if (rewriteMain || lastAmtNotificationsShown != bt_amt_notifications())
        ↪  {
        lastAmtNotificationsShown = bt_amt_notifications();

        // Draw number of notifications
        disp_set_font(FONT_MEDIUM);
        disp_set_pos(1, (menuTextY[4]+menuTextY[5])/2);
        if (!bt_amt_notifications()) {
            disp_write_str_group("No␣notifications.", NOTIFICATION_NUM_ID);
            disp_end_group();
            last_drawn[NOTIFICATION_NUM_ID] = 1;
        } else {
            disp_write_str_group(itoa(bt_amt_notifications(), buffer, 10),
                ↪ NOTIFICATION_NUM_ID);
            disp_write_str_group("␣notification", NOTIFICATION_NUM_ID);
            if (bt_amt_notifications() > 1)
                disp_write_str_group("s", NOTIFICATION_NUM_ID);
            disp_write_str_group(".", NOTIFICATION_NUM_ID);
            disp_end_group();
            last_drawn[NOTIFICATION_NUM_ID] = 1;
        }
        if (!startup)
            disp_commit();

        // Draw button text at bottom of screen
```

```
            disp_set_font(FONT_SMALL);
            disp_set_pos(1, menuTextY[6]);
            disp_write_str_group("<␣Graph", LEFT_BUTTON_ID);
            disp_end_group();
            if (!startup)
                disp_commit();
            last_drawn[LEFT_BUTTON_ID] = 1;
            uint8_t x, w, y, h;
            disp_get_text_bounds("View␣>", 0, 0, &x, &y, &w, &h);
            disp_set_pos(DISP_WIDTH-w-1, menuTextY[6]);
            disp_write_str_group("View␣>", RIGHT_BUTTON_ID);
            disp_end_group();
            if (!startup)
                disp_commit();
            last_drawn[RIGHT_BUTTON_ID] = 1;
            rewriteMain = 0;
        }
    }
}
```

S.9

```
static inline void next_alarm(void) {
    if (READING_TIMEOUT > 0) {
        alarm.time.second += READING_TIMEOUT;
        while (alarm.time.second >= 60) {
            alarm.time.second -= 60;
            alarm.time.minute++;
            if (alarm.time.minute>=60) {
                alarm.time.minute = 0;
                alarm.time.hour++;
                if (alarm.time.hour > 11) {
                    if (alarm.time.hour == 12) {
                        if (alarm.time.pm) {
                            add_to_date_uchar(1, &(alarm.time.year), &(alarm.time
                                ↪ .month), &(alarm.time.day));
                        }
                        alarm.time.pm = !alarm.time.pm;
                    } else
                    alarm.time.hour = 1;
                }
            }
        }
    }
}
```

```
static void rtc_alarm_callback(void) {
    rtc_alarm_flag = 1;

    // Trigger measurement
    if (READING_TIMEOUT > 0) {
        take_measurement();
    }

    // Set new alarm
    next_alarm();

    rtc_calendar_set_alarm(&rtc_instance, &alarm, RTC_CALENDAR_ALARM_0);
}
```

S.10

```
uint8_t is_screen_timeout(void) {
    return screen_timeout==0;
}

void set_screen_timeout(uint16_t val) {
    tcc_stop_counter(&screen_timer);
    screen_timeout = val;
    if (val != 0) {
        tcc_set_count_value(&screen_timer, 0);
        tcc_restart_counter(&screen_timer);
    }
}

static void screen_timer_callback(struct tcc_module *const module) {
    // Decrement the counter until it reaches zero, then stop the timer
    if (screen_timeout > 0) {
        screen_timeout--;
    } else {
        tcc_stop_counter(&screen_timer);
    }
}
```

S.11

```
// Bluetooth driver task
void bt_task(void) {
```

```
// Run the task for the ACI BLE library interface
aci_loop();

if (rx_buffer_len) { // If there is a received message
    rx_buffer_len--;

    // Set date and time
    if (rx_buffer[cur_rindx][0] == 'D') {
        // expect date/time string- example: DYYYY MM DD HH MM SS (D[Year] [
            ↪ Month] [Day] [Hr] [Min] [Sec])
        struct rtc_calendar_time time;
        rtc_get_time(&time);
        char *ptr;
        time.year = strtol(&rx_buffer[cur_rindx][1],&ptr,10);
        time.month = strtol(ptr,&ptr,10);
        time.day = strtol(ptr,&ptr,10);
        time.hour = strtol(ptr,&ptr,10);
        time.minute = strtol(ptr,&ptr,10);
        time.second = strtol(ptr,&ptr,10);
        rtc_update_time(&time);
        request_screen_on();
    }

    // Set notification 1
    else if (rx_buffer[cur_rindx][0] == '1') {
        bt_set_notification_1(&rx_buffer[cur_rindx][1]);
        request_screen_on();
        new_notifications = 1;
    }

    // Set notification 2
    else if (rx_buffer[cur_rindx][0] == '2') {
        bt_set_notification_2(&rx_buffer[cur_rindx][1]);
        request_screen_on();
        new_notifications = 1;
    }

    // Set measurement reading timeout in seconds
    else if (rx_buffer[cur_rindx][0] == 'R') {
        char *ptr;
        long val = strtol(&rx_buffer[cur_rindx][1],&ptr,10);

        measure_set_reading_timeout(val);
        request_screen_on();
    }

    // Increment circular buffer reading index
```

```
        if (++cur_rindx >= BT_MAX_BUFFER_LENGTH) cur_rindx=0;
    }
}
```

S.12

```
// Battery reader task
void battery_task(void) {
    uint16_t result = 0;
    if (adc_active && adc_read(&adc_instance, &result) == STATUS_OK) { // When
        ↪ the ADC read succeeds
        adc_disable(&adc_instance);

        adc_result = result;
        if (result != 0) {
            // Precompute the level assuming the same max used last time
            battery_level = (int16_t)(((float)result*MAX_V/MAX_ADC - MIN_V)/(
                ↪ MAX_V-MIN_V)*(float)last_max);
            if (battery_level < 1) battery_level = 1;
            if (battery_level > last_max) battery_level = last_max;
        }
        adc_active = 0;
    }
}

int get_battery_level(int max) {
    if (max == last_max) {
        // Return precomputed level if the max has not changed
        return battery_level;
    }
    last_max = max;
    battery_level = (int16_t)(((float)adc_result*MAX_V/MAX_ADC - MIN_V)/(MAX_V-
        ↪ MIN_V)*(float)max);
    if (battery_level < 1) battery_level = 1;
    if (battery_level > max) battery_level = max;
    return battery_level;
}

void start_battery_read(void) {
    if (!adc_active) {
        adc_active = 1;
        adc_enable(&adc_instance);
        adc_start_conversion(&adc_instance);
    }
}
```

S.13

```
void take_measurement(void) {
    measure_busy = 1;
}

// Measurement controller task
void measurement_task(void) {
    if (measure_busy) {
        switch (pulseState) {
            case 0: // Initialize implant
                pulseCounts = 0;
                glucoseTemp = 0;
                numPoints = 0;

                // Enable LEDs to power implant
                port_pin_set_output_level(LED_PIN, true);
                set_pulse_timeout(pulseOne);
                pulseState = 1;
                break;

            case 1: // Enable capturing and wait for reading
                if (is_pulse_timeout()) {
                    set_pulse_timeout(pulseTwo);
                    enable_capture();
                    pulseState = 2;
                }
                break;

            case 2: // Make readings over a specified period, then average them
                if (is_pulse_timeout()) {
                    do_measurement();
                    pulseCounts++;
                    if (pulseCounts < MAX_PULSE_COUNTS) {
                        set_pulse_timeout(pulseTwo);
                        enable_capture();
                    } else {
                        // Disable LEDs to power implant
                        port_pin_set_output_level(LED_PIN, false);
                        if (numPoints > 0) {
                            glucose = (glucoseTemp / numPoints);
                            new_measurement = 1;
                        }
                        measure_busy = 0;
                        pulseState = 0;
                    }
```

```
            }
            break;

        default:
            pulseState = 0;
            break;
    }
  }
}

static void do_measurement(void) {
    disable_capture();

    if (nCap < 1)
        return; // Failure
    numPoints++;
    uint32_t sum = 0;
    // Start at 1, to ignore first read value
    for (uint16_t i = 1; i < nCap; i ++) {
        sum += (uint32_t)periods[i];
    }

    // Assumes 8MHz clock
    freq = 8000000.0f * (float)(nCap-1) / (float)sum;
    nCap = 0;

    glucoseTemp += GLUCOSE_CONVERSION(freq);
}
```

S.14

```
static void button_listener_callback_R(void) {
    if (is_button_timeout()) {
        button_interrupt_flag = 1;
        button_pressed |= BUTTON_R_VAL;
        set_button_timeout(BUTTON_TIMEOUT);
    }
}

// Used to poll the interrupt flag without clearing it
uint8_t is_button_interrupt_soft(void) {
    return button_interrupt_flag;
}

uint8_t is_button_interrupt(void) {
```

```
    if (button_interrupt_flag) {
        button_interrupt_flag = 0;
        return 1;
    }
    return 0;
}

// Used to poll the button values without clearing them
uint8_t get_buttons_soft(void) {
    return button_pressed;
}

uint8_t get_buttons(void) {
    if (button_pressed) {
        uint8_t ret = button_pressed;
        button_pressed = 0;
        return ret;
    }
    return 0;
}
```