Wearable UV Light Exposure Logger
Final Report

Team Members: Tennyson Cheng, Judah Ben-Eliezer

Advisor: Dmitri Donetski

Date: 05/08/2022

Abstract

UV rays due to emission from the sun are everywhere. Although many things on our planet require sunlight, UV rays which can be harmful to the human skin. Excessive exposure to the sun's UV rays can cause sunburns, premature aging of skin and worst of all, skin cancer. There are solutions to combat harmful UV rays such as sunscreen and clothing that covers a lot of skin. The most effective solution is to stay indoors, but we don't expect all of humanity to stay in caves forever. Sunscreen was invented in order to protect our skin from UV rays, while also giving us the ability to go outside for long periods of time. However, sunscreen is not a permanent solution and will need to be reapplied. Our product will measure the amount of UV exposure and notify the user of when to reapply sunscreen, while being completely non-intrusive to everyday life.

Contents

1. Background Research

1.1 UV Rays

UV rays are divided into 3 categories: UVA, UVB and UVC. UVA, UVB and UVC correspond to wavelengths 315-399 nm, 280-314 nm and 100-279 nm [2]. The smaller the wavelength, the more energy the waves contain. But with a smaller wavelength, the penetration power of the wave is decreased. In general, the lower the wavelength, the more harmful the waves are to human skin. The most harmful UV ray, UVC, is completely filtered out, by the ozone layer [2]. The only waves that reach the Earth's surface are UVA and some UVB. UVA is often associated with skin aging, while UVB is associated with skin burning [3]. Unsafe exposure to any of the 3 waves will lead to skin cancer. UV rays penetrate into the skin, damaging the DNA in skin cells and causing genetic defects/mutations [3]. To measure the exposure of UV rays, the UV index measurement standard was created. This index ranges from 0 to 11+, where 0 is the lowest level of exposure and 11 is the highest level that can be produced by the sun. In a more technical definition, 1 level in the UV index equates to 25 mW/m^2 of UV radiation [20].

Sunscreen is the most common defense against UV ray exposure. Due to its composition of certain physical and chemical particles, it is able to reflect UV rays and absorb them to release heat. Sunscreens are labeled with an SPF value, which stands for Sun Protection Factor. This value represents how much longer the skin (as a multiplier) can stay exposed to UV rays [4]. With an SPF of 30, the skin should be able to be exposed to UV rays 30 times longer than without sunscreen. For example, if someone's skin were to take 10 minutes to start burning, then using SPF 30 sunscreen will extend that time to 300 minutes. This rating specifically applies to UVB radiation only. Sunscreens advertised with broad spectrum protection have additional components to protect against UVA [4]. It is important to have sunscreen with broad spectrum protection since UVA accounts for 95% of UV radiation [3], despite being weaker than UVB.

A person's skin type affects their susceptibility to sunburn [5]. Skin types can be divided into 6 categories. Light skin is weaker against UV rays while dark skin is more resistant to UV rays. A more quantifiable way of determining sunburn is by the skin's minimal erythemal dose (MED). The MED is the UV radiation dose required to produce a noticeable reddening of the skin due to the engorgement of capillaries [19]. Each skin type has a different MED, representing their resistance to sunburn. MED is represented in units of J/m^2, which can be converted to (W*sec)/m^2. Determining the time until erythemal dose (sunburn) is simply a matter of dividing the MED by the UVI.

2. Societal Needs and Impact

2.1 Skin Cancer

Skin cancer can develop due to lack of skin care when it comes to UV radiation. According to the American Academy of Dermatology, the most common cancer in Americans is skin cancer and that “one in five Americans will develop skin cancer in their lifetime” [24]. Melanoma, the type of skin cancer caused by excessive UV exposure, causes the death of nearly 20 Americans every day [23].

2.2 Sunburns

One of the ways to reduce the likelihood of developing skin cancer is to prevent sunburns. By simply having 5 or more sunburns in one life, the risk of developing melanoma doubles [23].

2.3 Treatment Cost

Along with skin cancer comes the price of treatment. Around $8.1 billion is spent in the US alone, for treating skin cancer. $3.3 billion of that is used for the treatment of melanoma [24]. By preventing skin cancer, this money can be spent more effectively elsewhere.

2.4 Impact

The UVtech sensor can play an important role in the prevention of sunburn, and thus prevention of skin cancer. While many people apply sunscreen before going outside, they rarely keep track of their UV exposure and often forget to reapply, thus making them susceptible to overexposure. With the monitoring of the UVtech sensor, and its capability of keeping track of sunscreen potency, we foresee a decrease in the number of cases of sunburn and skin cancer in our customers. Due to its affordability and form factor, the UVtech sensor can also be integrated into society easily. Ultimately, this will result in greater quality of life for everyone, as they will live healthier and worry less about the dangers of UV radiation.

3. Constraints and Standards

3.1 Portability

Since the goal of this project is to create a wearable UV sensor, the design needs to be focused on portability. Size and weight are important aspects to the design of the product. Target requirements were set to create a device that will be around the size of a quarter (~20 mm), while also weighing less than 10 g. This should be small and lightweight enough to be embedded onto any piece of clothing without obstructing daily life. For the final product, we will embed the device onto a hat for demonstration.

3.2 Battery

As a consumer product, a long battery life is an important objective. It is undesirable to have a device that needs to be constantly recharged. A set goal of 168 hours (1 week) was set for the battery life. This was determined based on other bluetooth devices such as Fitbit trackers, whose devices last around a week as well. Considering that the final product won't be a device that is always worn, like the Fitbit watches, 1 week is a strong battery life for our device. If needed, battery life requirements can be lowered depending on compromises with portability.

It is also important to implement a standard charging interface with the device. Micro usb is the most common connector for recharging bluetooth devices. The microcontroller should seamlessly switch to USB power when USB is plugged in, while also charging the battery. From a consumer point of view, being able to charge with a universal connector is convenient.

3.3 Waterproofing

Although waterproofing will not be a requirement for the final product, it may be something to be considered in the future. People most commonly use sunscreen when doing outdoor activities in the Spring or Summer. Activities such as swimming should be expected and therefore waterproofing would be a worthwhile addition to the device. For the scope of the project, waterproofing will not be considered.

4. Hardware Design

Hardware should focus on meeting the constraints discussed in section 3: portability, long battery life and bluetooth connectivity. Another focus will be placed on cost. This will help narrow the parts to choose from and result in a more cost effective design that can be more marketable. Considerations of best and worst case design decisions will be discussed.

The primary components needed are a microcontroller with bluetooth connectivity, UV sensors, battery, battery charging IC, and a voltage regulator.

4.1 Microcontroller

The microcontroller should have an RF unit with at least BLE 5.0 support so an extra chip for Bluetooth functionality is not needed. A standalone bluetooth may also consume more power as it is not a part of the microcontroller. BLE 5.0 should be supported as it allows for 2 Mbps data rates, the fastest data rate for Bluetooth.

The microcontroller also needs the appropriate peripherals to support the UV sensors, depending on the interface they use. This could be either an ADC or I2C peripheral. A DMA peripheral is desirable as it allows the movement of data without waking up the microprocessor.

Speed for the microcontroller is not important, we just need to be able to read UV sensor data and send them to a mobile device. Cost and power consumption will be the primary focus for deciding the microcontroller.

4.2 UV Sensor

Four UV sensors will be implemented for the final design. This means a more accurate representation of the UV exposure on the wearer can be determined. If only one sensor were to be used and the sensor is covered, incorrect readings of the UV index will be recorded. More sensors results in more reliable UV index readings.

A PCB will be designed for one UV sensor. The purpose is to be able to add or remove UV sensors as desired. Developing a separate module dedicated to the UV sensor also allows them to be placed anywhere, as long as it is wired to the main microcontroller board.

Preferably, the sensor should be able to detect UVA and UVB rays. But it is fine if it doesn't as long as it can determine the UV index accurately.

4.3 Battery

A Lithium-Ion battery will be used to power the system. Li-Ion batteries can be recharged, unlike a coin cell battery. A non-rechargeable battery may be able to hold more charge and won't need a charging circuit. However, the convenience of a rechargeable battery is more desirable for a consumer wearable device. The voltage level of the battery needs to be compatible with the microcontroller.

4.4 Battery Charging

Battery charging IC should have a programmable charging current and charging cut-off. Programmability for the charging current and charging cut-off is important for the health of the Li-Ion battery. Limiting the charging current to 50 mA and the charging cut-off to 20% will extend the lifespan of the battery [18]. Although disregard for this constraint would not affect functionality of the system, and may even speed up the charging period of our system, longevity of the product is prioritized.

Since the device will be interfaced with micro usb, the charging IC must support a 5V input. The output regulation voltage should also match that of the battery.

An LED will be attached to the 5V input of the micro usb in order to indicate that the micro usb is properly connected and charging the battery.

4.5 Voltage Regulator

The voltage regulator output should be within the voltage range of the microcontroller and UV sensors. But, for a more standard voltage level, we will be targeting an output voltage of 3.3 V. The input voltage must be able to accept the 5 V micro usb input voltage. The voltage of the battery will be lower than 5 V, so the input only needs to support up to ~5 V.

A simple LDO linear voltage regulator should be chosen for this project. Compared to switching regulators, linear regulators produce more heat and have worse efficiency. However, the advantage of using a linear voltage regulator is its ease of implementation and cost. Only a capacitor near the output of the regulator is typically needed for the circuit. Not only will it lower costs, it will also require less space on the PCB, which is important for a wearable device. Also, the voltage drop from 5 V to 3.3 V is only 1.7 V. For a low voltage in-out difference, a LDO linear voltage regulator is more fitting as the efficiency does not have great importance. One more reason to not use a switching regulator is that it can introduce noise into the circuits, which may affect RF performance.

4.6 Power Supply Switching

As mentioned in section 2, the power supply for the system should switch between the battery and micro usb depending on whether the micro usb is plugged in or not. This can be implemented by a simple diode OR circuit. By connecting the 5 V usb input and the battery to the input of the voltage regulator, with 2 diodes separating the power supplies, a current will only be drawn from the power supply with a higher voltage. Another option is to use a dedicated power management IC, but that would be more costly than just using 2 diodes.

Since these diodes will be connected from the power supplies to the regulator, voltage drop is a big concern. Using schottky diodes will help reduce the voltage drop penalty compared to a standard diode. Schottky diodes also have better switching performance, which is ideal when it comes to switching between different power supplies. A low reverse current is also important to prevent overcharging of the battery due to leakage.

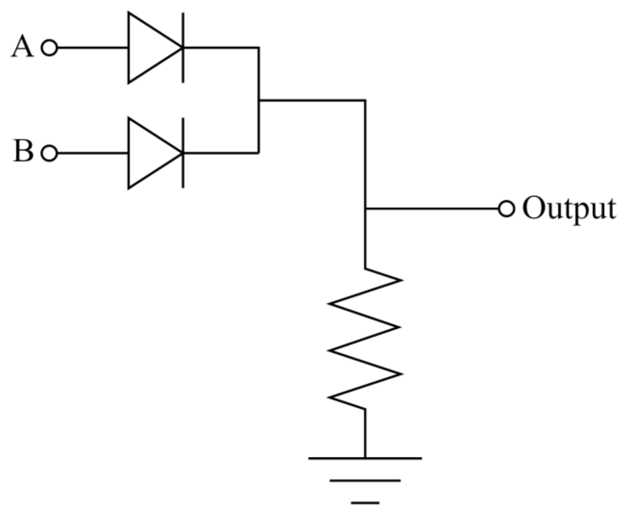


Figure 1. Example of a power supply diode OR

4.7 User Input

A push button connected to the microcontroller will be used for user input. Instead of creating a physical debounce circuit, debouncing can be done in software by the microcontroller. The push button must be connected to a GPIO pin of the microcontroller. An LED for user feedback will also be interfaced with another GPIO pin. This LED can indicate 3 states: advertising, connected or off.

4.8 Solar Panel

Usage of a solar panel for passive charging of the battery was considered but ultimately decided against. Inclusion of a solar panel would increase the cost. It would also require more consideration in the charging circuitry. A solar panel can’t just be simply added to a circuit because there is the case where the solar panel is not able to provide enough power to charge the battery. When that happens, the solar panel will attempt to supply the current required, but fail and crash. Immediately after, the charging IC will be disabled, allowing the solar panel to recover. This only creates a constant loop of the solar panel recovering then crashing. It is also not healthy for the battery to be constantly charging then discharging due to this. A more advanced (costly) charging IC will be needed in order to lower the charge current adaptively, depending on the supply voltage.

5. Software Design

5.1 Embedded Firmware

In order to save space and money on the physical battery, firmware for the microcontroller should be fast and efficient. Reading UV samples via polling should not be considered. Having the microprocessor running at full speed, doing nothing but waiting, is a waste of energy. An interrupt-based approach is the only way that allows us to put the microcontroller to sleep while there are no tasks to do.

In order for an interrupt-based system to be more efficient than a polling-based system, the interrupt service routine must be fast enough that it finishes before the next sample is required. This is easily accomplished by sampling UV data once per second. The routine will only need to command the sensors to get UV data, read the UV data from the sensors, and store them in a buffer. Doing this will ensure that the microcontroller can go back to deep sleep right after reading and storing. To store the UV samples, the DMA should be utilized. The DMA is a common unit on microcontrollers that allows the reading/writing of data without the CPU. This allows the CPU to focus on sending the fetch command to the sensors. Having the fetch command last in the interrupt service routine allows the microcontroller to sleep as the sensors will take time to get new UV data.

Assuming that the microprocessor will be clocked in the MHz range, at least one million clock cycles can occur in one second. The tasks required should be able to be completed in less than a million clock cycles.

On every third second, the microcontroller will also send the UV data buffer to the mobile device through Bluetooth LE (BLE) 2 Mbps. A transmission interval of 3 seconds was chosen because BLE specification states that connection events must be within 7.5 ms to 4 sec. 3 seconds provides enough leeway from the 4 second limit, while also dividing into 60 seconds evenly. BLE 2 Mbps speed will be used since it is the fastest transmission speed of the bluetooth standard. Using 2 Mbps means there is less time needed for the microcontroller to keep the radio peripheral on, resulting in significantly less power consumption. However, using BLE 2 Mbps brings a limitation to the bluetooth devices supported. BLE 2 Mbps was introduced in the Bluetooth 5.0 specification, and would still require that the receiving hardware support 2 Mbps data transfers. On the other hand, Bluetooth 5.0 was released in 2016 and pretty much any modern bluetooth device produced now supports Bluetooth 5.0+ functionality.

Any processing of the raw UV data will be done on the mobile device in order to save processing time for the microcontroller, further conserving battery life.

5.2 Mobile Application

The mobile app is to log UV data received by the wearable. With the users SPF and skin type, the app can determine when to reapply sunscreen. The logged UV data will be saved on the device and can be viewed on a timeline. Additionally, a calendar will be used to access UV data from past dates.

6. Design Selection

Note: Parts were narrowed down based on part availability and price

6.1 Microcontroller

Part Chosen: EFR32BG22

Microcontroller	Bluetooth Version	Maximum TX Radio Power	Current Consumption (TX/RX)	Current Consumption Active	Current Consumption Deep Sleep	Price
STM32WB15	5.2 BLE	5.5 dBm	8.6 mA/7.7 mA	71 uA/mHz	1.9 uA	$5.54
ESP32-SOLO-1	4.2 BLE	21 dBm	130 mA/100 mA	312 uA/mHz	5 uA	$3.74
nRF52832	5.2 BLE	4 dBm	7.1 mA/6.5 mA	58 uA/mHz	1.5 µA	$4.00
EFR32BG22	5.2 BLE	6 dBm	8.5 mA/4.5 mA	22 uA/mHz	1.75 µA	$2.02

Figure 2. Microcontroller comparisons

The EFR32BG22 was decided based on its price/performance ratio. If we looked strictly at performance, the nRF52832 would be the best (ignoring the active power consumption). It has 5.2 BLE, programmable 4 dBm TX power, the lowest current consumption and plenty of support/documentation as it is used in a few Adafruit and Sparkfun breakout boards. However, if we look at the price, the EFR32BG22 offered very competitive performance for half the price of the nRF52832. The well known and trusted STM32 family of microcontrollers consists of the new STM32WB15. The STM32WB15 also has good performance, but was also decided against due to its price. Lastly, the ESP32-SOLO-1 was included because the ESP32 is a very popular microcontroller for low cost bluetooth projects. There are even more resources available for the ESP32 family than there are for the STM32WB and nRF, when it comes to programming bluetooth. Despite their fame, the ESP32 is very power hungry in comparison to the other three.

The EFR32BG22 has an internal DC-DC buck voltage regulator to further lower power consumption of the device. It only needs a single external supply voltage, and it will create all the internal voltage sources necessary for full functionality. The voltage output by the regulator is 1.8 V, and can be used to supply RF, GPIO and ADC voltages independently.

The EFR32BG22 also supports 4 different energy saving modes, EM1, EM2, EM3 and EM4. EM1 is a light sleep mode. The clock to the CPU is disabled, but all peripherals, RAM and flash are available. With DMA, data can be read from the peripherals and written to RAM without CPU intervention. The current consumption of EM1 is 13 µA/MHz. By using the 38 MHz crystal for its main clock, the current consumption of the microcontroller is 494 uA. EM2 is a deep sleep mode, in which only low frequency peripherals are active. There is very limited functionality in this mode, but an RTC can be programmed to wake the CPU. Current consumption in this mode is very small (only 1.94 uA with Full RAM retention and RTC running from LFXO). EM3 represents stop. Not much can be done in EM3 other than a few external interrupts and timer interrupts when certain clock sources are left active. The current consumption is similar to EM2 as it consumes 1.41 uA, with limited RAM retention. EM4 is a shutdown mode in which all peripherals are disabled with no RAM retention.

6.2 UV Sensor

Part Chosen: Si1132

Sensor	Current Consumption Idle	Current Consumption Active	I2C or Analog interface	Price
GUVA-S12SD	N/A	N/A	Analog	$6.60
VEML6070	15 uA	0.25 mA	I2C	$5.95
Si1132	0.5 uA	5.5 mA	I2C	$2.25

Figure 3. UV Sensor Comparisons

The Si1132 has programmable I2C addresses (perfect for four sensors on one bus), and a very low idle current consumption at a cheap price. Although the VEML6070 offers better active current consumption, there is no stock of the VEML6070 and Vishay Semiconductors do not produce any UV sensor alternative. The only way to buy a VEML6070 is through breakout boards made by other companies, increasing their price. Most importantly, the VEML6070 does not have configurable I2C addresses, so having four of these sensors on the same I2C bus would not work without some multiplexing. The GUVA-S12SD is a real UV sensor, and so it outputs an analog voltage. The current is not listed because the current output of the sensor is in the nA range. In order to interpret the voltage from the GUVA-S12SD, an op-amp is needed to amplify the output voltage. The power consumption of the GUVA-S12SD may potentially be better than the Si1132 if the op-amp was only turned on for measurements. Despite that, the cost of the GUVA-S12SD is also 3x that of the Si1132.

Although the Si1132 can only detect UVA rays and not UVB, the chip has been programmed with an algorithm to accurately determine the UV index. In figure 4, we can see that the sensor performs best in sunny conditions compared to cloudy. We can also see that the UV index is more accurate for values larger than 2. However, values less than 2 are insignificant as there is hardly any harmful UV exposure.

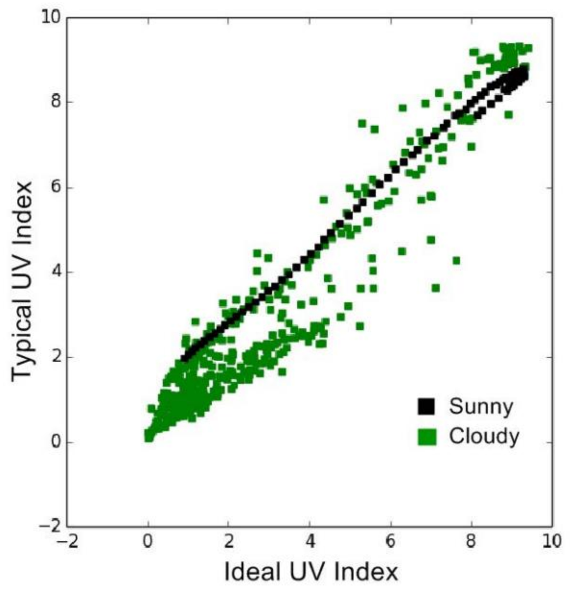


Figure 4. Measured UV Index vs. Actual UV Index

The logic levels for the Si1132 and EFR32BG22 are also the same. Both can operate at 1.71 V to 3.8 V, and use a logic low of 0.3*VDD and logic high of 0.7*VDD.

6.3 Battery

Part Chosen: ASR00003

Battery	Capacity	Weight	Expected Battery Life	Price
LP402025	150 mAh	4.65 g	285.1 h	$6.60
ASR00007	290 mAh	7.0 g	399.1 h	$5.95
ASR00003	150 mAh	3.77 g	285.1 h	$3.95
ASR00011	70 mAh	1.65 g	133.0 h	$3.49

Figure 5. Battery Comparison

The ASR00003 was chosen for the final design for its low cost, low weight and sufficient capacity. The dimensions of all the batteries here are about the size of a quarter, except for the 70 mAh, whose size is nearly 30% of a quarter. The 150 mAh capacity would provide 13.1 days of battery life, while the 70 mAh would only provide 6.1 days. The 290 mAh capacity would have been overkill, providing 25 days. A larger capacity would also increase the charging time however. The form factor of the ASR00011 is incredibly small, but it only yields half the lifetime of the ASR00003 for just a ~$0.50 price reduction. The LP402025 has the same specifications as the ASR00003, but is more expensive.

The ASR00011 has a minimum voltage of 3.0 V when fully discharged and a maximum voltage of 4.2 V when fully charged.

6.4 Battery Charging IC

Part Chosen: MCP73831T-2ATI

MCP73831T-2ATI was chosen for its programmability and price ($0.69). It is the cheapest charging IC that allows a programmable charging cut-off of 20%. Another option is the MC34675AEPR2 ($0.63). The reason it was not chosen was because the charging cut-off is pre-set to 10%. Programmability and charging cut-off is important because we want to extend the lifespan of the battery itself. Other options were more expensive than the MCP73831T-2ATI, so they were not considered as the MCP73831T-2ATI already fulfills the requirements.

The MCP73831T-2ATI is also compatible with the maximum 4.2 V of the battery. It has a regulation voltage of 4.2 V, in which it will stop charging once it detects 4.2 V on its battery pin. It also supports 4.5 V to 6.0 V for its input supply voltage, which covers the 5.0 V supplied by the micro usb.

6.5 Voltage Regulator

Part Chosen: NCP707CMX300TCG

Something cheap and does the job is pretty much the requirement for the voltage regulator. The NCP707 by Onsemi is sufficient. The only caveat is that it outputs 3.0 V instead of 3.3 V, but this is acceptable because the EFR32BG22 and Si1132 can function in voltages as low as 1.8 V. The system will consume less power by using 3.0 V instead of 3.3 V. The reason the 3.0 V variant is used is due to the chip shortage.

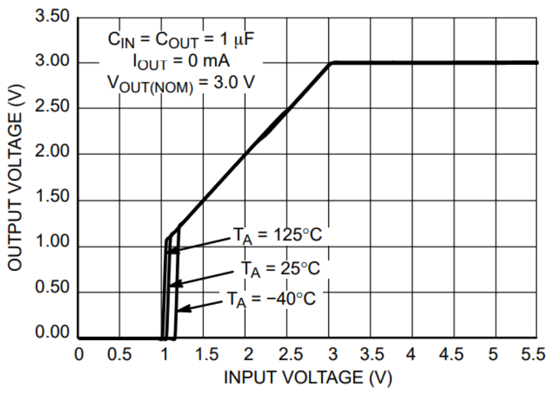


Figure 6. Input vs Output Voltage characteristics of the NCP707CMX300TCG

The NCP707 output voltage scales linearly with the input voltage, and caps at 3.0 V when the input is higher than 3.0 V. In the datasheet, it is stated that the minimum input voltage is 1.8 V, which is low enough to support full discharge of our battery. From figure 6, we can see that for input voltages of less than 3.0 V, the output voltage will be equal to the input until ~1.1 V. Max output current of 500 mA is more than enough for our microcontroller and sensors.

7. Results

7.1 System Schematic

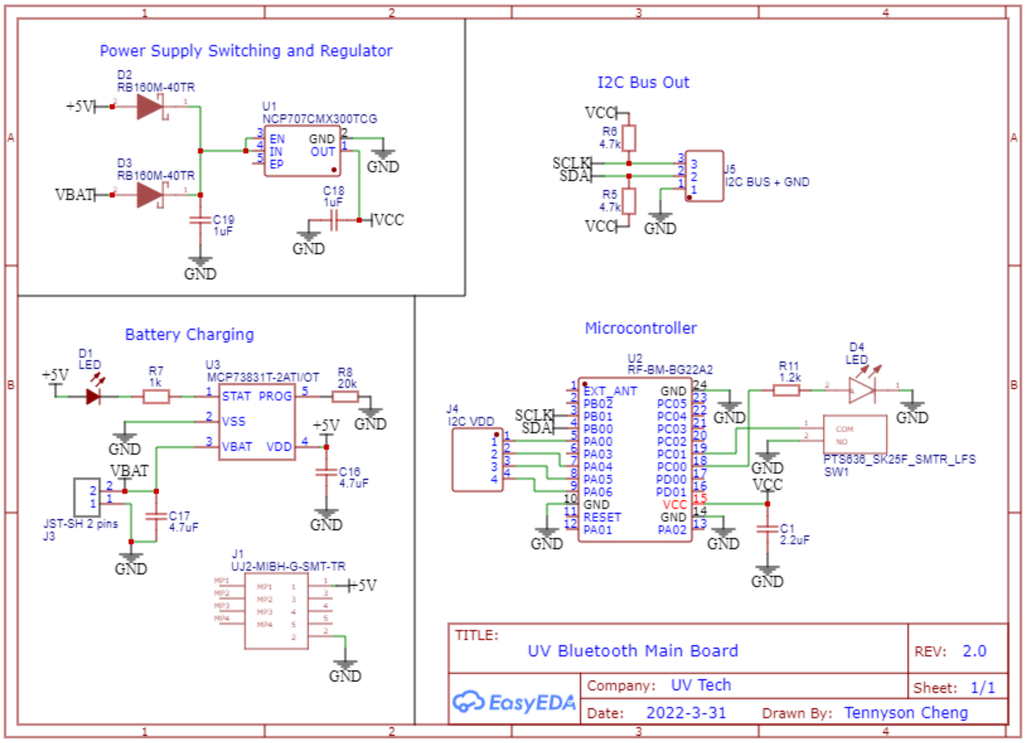


Figure 7. Circuit Schematic for the main board.

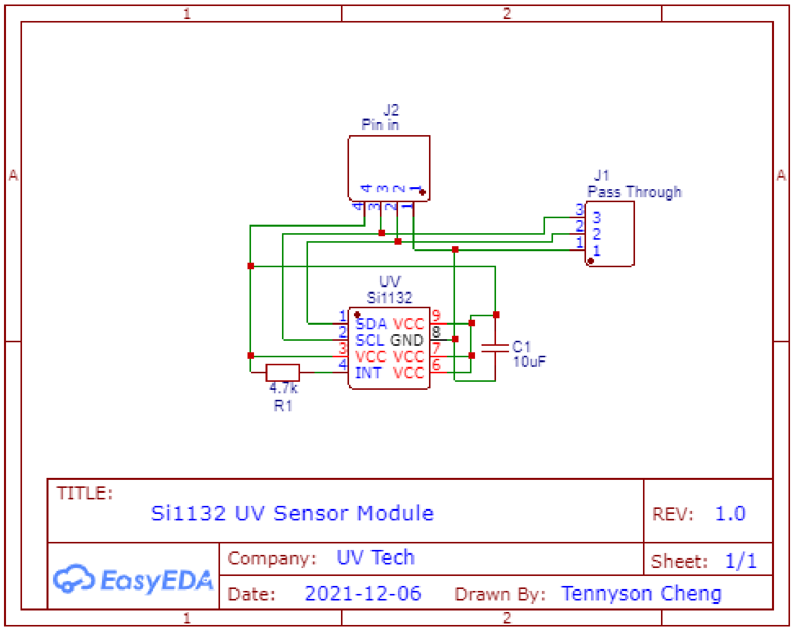


Figure 8. Circuit schematic for a single UV sensor module.

Figure 7 and 8 show the schematic for the system. The main board supplies power and is responsible for any data transfer between the sensors and the mobile device. Decoupling capacitors were determined using the recommended values according to each component's data sheet.

The RF-BM-BG22A2 is a module with the EFR32BG22, PCB antenna, and all necessary components already included. The cost of this module is only \$3.08, a ~\$1.00 difference from buying the baremetal EFR32BG22. Since the PCB antenna is already implemented with the module, we don't need to worry about designing the PCB antenna ourselves. The EFR32BG22 is clocked using the 38.4 MHz crystal, decreasing the current consumption of the microcontroller. The main voltage source VDD is connected to VREGVDD in order to use the internal DC-DC buck regulator for even lower current consumption. The DC-DC regulator produces a voltage Vdcdc on VREGSW when the appropriate inductor and capacitors are connected. This voltage is 1.8 V, and powers the microprocessor and RF peripheral. The GPIO pins are still powered by VDD, meaning logic levels will be scaled with 3.0 V.

LED D1 indicates that the battery is being charged by micro usb. LED D2 is used for user feedback when SW1 is pressed. SW1 is debounced in software, so components for a debounce circuit are not needed.

The EN pin of the voltage regulator is tied to the IN pin because the enable functionality is not needed. The microcontroller is programmed to wake up when SW1 is held for 3 seconds. The microcontroller will constantly be powered as long as there is a power supply connected. However, the microcontroller will be sleeping in EM3 mode, only consuming 1.41 uA typically and 3.7 uA at most.

Since the UV sensors will be chained together, only two pullup resistors (R5 and R6) are needed for the bus. In order to power each sensor separately, J4 along with 4 GPIO pins are used.

7.1.1 I2C Pullup Resistor

In order to calculate the maximum pullup resistor for I2C, the rise time and bus capacitance are needed. The rise time in I2C context is the time it takes for the amplitude in SCL or SDA to reach 70% amplitude from 30% amplitude. For two devices, the smallest maximum rise time is to be used in the calculation. Bus capacitance is the total capacitance on the I2C bus, and is the sum of the capacitances of all devices on the I2C bus plus PCB capacitance.

For the system, I2C communication will be done in fast mode. According to I2C standards, the maximum bus capacitance for fast mode is 400 pF. Since capacitance for the EFR32BG22 and Si1132 are not given, 10 pF will be assumed for each device. 10 pF will also be assumed for the PCB. This gives a total bus capacitance of:

$$C\_bus = C\_EFR32 + 4*C\_Si1132 + C\_PCB = 10 + 40 + 10 = \mathbf{60\ pF}$$

The maximum rise time for the system is **40 ns**, according to the Si1132. The formula for calculating the maximum pullup resistance as a function of maximum rise time is:

$$R_p(\text{max}) = \frac{t_r}{(0.8473 \times C_b)}$$

where t_r is the maximum rise time and C_b is the bus capacitance. With this formula, the theoretical maximum pullup resistor that can be used is:

R_max = t_r / (0.8473*C_bus) = 40 ns / (0.8473*60 pF) = 786.813013887 Ω

However, the max rise time specified in the datasheet seems to be incorrect. The Si1145 sensor has the same timing characteristics as the Si1132, but there exists a breakout board by Adafruit which uses 10 kΩ pull-ups. The 10 kΩ far exceeds the calculated 786 Ω resistor based on the 40 ns rise time specified in the datasheet. Based on this, 300 ns will be assumed for the maximum rise time. By assuming the I2C standard maximum of 300 ns for the rise time:

R_max = (300 * 10^-9)/(0.8473*(60 * 10^-12)) = 5901.09760415 Ω

With 80% margin: 5901.09760415 * 80% = **4.7 kΩ**

7.1.2 Sensor INT Pin

Since the interrupts from the Si1132 sensors won't be used, they are to be tied to their respective VDD through a 4.7 kΩ resistor. The reason they can not be left floating is because they must be at a logic level high during power-up to enable low power operation of the sensors.

7.1.3 Battery Charger PROG Resistor

In order to get a charging current of 50 uA, a programming resistor of 20 kΩ needs to be used. This is derived from the following equation:

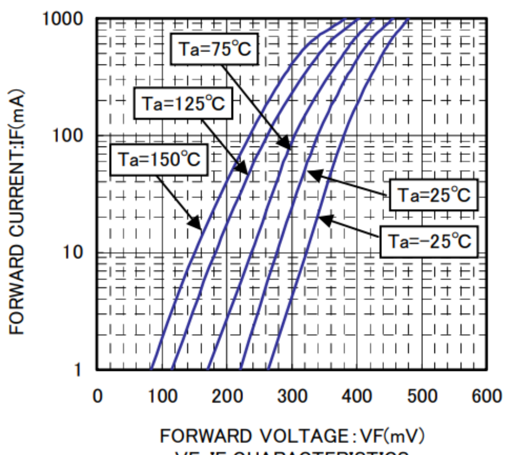
$$I_{REG} = \frac{1000V}{R_{PROG}}$$

Where:

R_{PROG}	=	kOhms
I_{REG}	=	milliampere

7.1.4 Schottky Diode Forward Voltage

The forward voltage of the RB160M-40TR schottky diode is 460 mV. With the lowest battery voltage being 3.0 V, we can expect to see at minimum, 2.54 V at the LDO input. In actuality, the forward voltage is lower due to the current used by the system (around 3 mA when the MCU is awake). According to the plot on the right, we see that the forward voltage is around 280 mV.



VF−IF CHARACTERISTICS

7.2 I2C Speed

The Si1132 has a maximum I2C clock support of 3400 kHz, which qualifies as high-speed mode under the I2C standard. The EFR32BG22 can only support fast mode, which is 1000 kHz. The fastest that the microcontroller can communicate with the UV sensor through I2C is 1000 kHz. However, only I2C fast mode (400 kHz) will be used. The reason being that fast mode plus uses stronger current drivers to achieve its speed. Limiting the I2C clock speed to 400 kHz uses less power. However, upon real testing, we were unable to communicate with 4 sensors using I2C fast mode. This could've been caused by parasitic capacitances in the microchips or PCB. A trade-off is done here between increasing the drive current by lowering the I2C pull-up resistors (R5 and R6), or lowering the I2C clock speed. We opted to lower the clock speed to 100 kHz, I2C standard mode. Increasing the drive current would require more experimentation with resistor values, a much more time consuming process than simply modify the firmware configuration.

7.3 Embedded Firmware

In order to allow the microcontroller to sleep when it is not transmitting data to a mobile device or reading UV data from the sensors, all peripherals used are interrupt triggered. The device has 3 states, OFF, ADVERTISING and CONNECTED. The OFF state means that the device is disconnected and is in EM3 deep sleep mode. ADVERTISING means that the device is advertising itself as a bluetooth peripheral. This state consumes the most energy as it needs to transmit advertising packets. The CONNECTED state means it is paired with a mobile device and will periodically measure UV data and transmit it to the mobile device. The device is in EM3 deep sleep in between periods, and in EM0 active mode when not.

7.3.1 Button Debounce

Before we implement any functionality for the button, it must be debounced to eliminate false triggers. To do this, GPIO interrupts and the RTC peripheral is used. The button connected to PC01 is set as a pull-up input, and will trigger an interrupt on a rising or falling edge. In its initial state, the voltage on PC01 will be a logic high, so when it is pressed, it will produce a falling edge. The IRQ will pick up the initial falling edge, then start a timer that will interrupt 15.6 ms later. The state of the pin is recorded as "PRESSED", and interrupts for PC01 are disabled. By disabling interrupts, we ignore all edges produced on PC01 due to bouncing.

For the duration of 15.6 ms, the microcontroller is asleep, as long as it does not need to read sensor data or transmit data. After 15.6 ms, the RTC timer will wake the microcontroller to check the state of PC01. If it sees that the state of PC01 is not still "PRESSED" (logic low), it will interpret the button press as invalid. However, if the state of PC01 is still "PRESSED" it is registered as a valid button press, and the recorded state will now be "DEBOUNCED". In both cases, interrupts are re-enabled. This effectively debounces the button push and release.

7.3.2 Button Functionality

In addition to the RTC timer interrupt routine from the debounce section, another timer is also set for a button hold. This will cause a timer interrupt 2 seconds later. If this interrupt occurs while the recorded state is still “DEBOUNCED”, then the state will be updated to “HELD” and a signal is sent for button hold functionality.

To avoid this button hold state, the user just has to let go of the button before the 2 second timer interrupt, and while the recorded state is still “DEBOUNCED”. When this happens, the GPIO PC01 interrupt will update the state to “RELEASED” and send a signal for button push functionality. When the 2 second timer interrupt inevitably occurs, it will see that the state is “RELEASED” and not “DEBOUNCED”, thus not triggering a button hold signal.

7.3.2.1 Button Push

A button push will reinitialize all the sensors and buffers when the system is “OFF”. It simply clears its UV data buffers and executes its sensor initialization routine. The LED is turned on for the time it takes to initialize all sensors, which is around one second. In any other state, a button push is ignored and has no functionality.

7.3.2.2 Button Hold

A button hold is used to enter “ADVERTISING” mode. This is only triggered when the device is either “OFF” or “CONNECTED”. When connected, the device will unpair from the mobile device and start advertising.

7.3.3 UV Sensor Initialization

For each sensor, there is a struct that holds the following information: active, VDD port, VDD port pin, and the I2C address it has been assigned to.

One thing to consider is that the Si1132 addressing resets to the default address 0x60. To access each sensor separately, different addresses for each sensor are written during startup. But since the sensors are all on one bus and have the same address initially, changing the address of one sensor will update the other ones as well. To get around this, each sensor is powered separately, so that the address can be set while the other sensors are off. This was done by connecting each sensor’s VDD pin to a different GPIO pin of the microcontroller. The initialization routine turns on PA00, PA04, PA05 and PA06 in sequential order, initializing each sensor separately.

In the case that a sensor is unable to be initialized for whatever reason, the routine will record that sensor as inactive in the sensor array.

7.3.4 UV Data and Buffer

For every sensor that is active, a command is sent to measure the UV data. Another command is then sent to read the UV data from each active sensor. This action is done every 1 second, timed by using the RTC timer. During the time that the device is not reading UV data or transmitting the UV buffer to the mobile device, it is sleeping.

The command used to start a UV measurement in forced mode is to send 0x06 to the sensors. After the command is sent, 300 us is needed for the sensor to sample the UV index. To read the UV data, the MCU needs to read 2 bytes from the sensor's UVINDEX0 register, 0x2C.

The UV data returned from the sensors is 16 bits long, representing 100 times the UV index. This data is stored in a buffer that can hold 12 samples. As a result, the buffer is 24 bytes long. With 4 sensors, this buffer can effectively hold 3 representative samples of the UV index for each of the 3 seconds UV data is sampled.

For inactive sensors, the data is recorded in the buffer as 0xFFFF.

7.3.5 Bluetooth Advertisement

7.3.5.1 Advertising

Advertisement packets are sent for around 30 seconds in ADVERTISING mode. If a BLE connection is not established within this period, the device will stop advertising and enter OFF state. Advertisement packets were programmed to be sent in 1 second intervals. Usually advertising packets are sent in 100 ms to 500 ms intervals, but this is not a requirement. By using 1 second intervals, we are able to save more power in exchange for a slightly longer device detection time. 30 packets are configured to be sent, resulting in a ~30 second advertising time.

7.3.5.2 Advertising LED

The blue LED to indicate advertising is controlled using the RTC timer. The timer is configured to interrupt every 62.5 ms, in which it will toggle the LED diode, D4. This behavior will continue until the advertising period stops, which is 30 seconds long.

7.3.6 BLE GATT

The Generic ATTribute Profile (GATT) defines the way data is transferred between BLE devices. For transferring UV data, a "Sensor Data" characteristic inside the "UV Tech Sense" service is defined in the GATT profile for the device.

7.3.7 UV Data Transmission

The "Sensor Data" characteristic is configured for notifications only. For GATT devices, there are 2 ways for a GATT server (sensors) to send data to the GATT client (mobile device) without a request from the client. They are called notifications and indications. The only difference between the two is that indications require an acknowledgement from the GATT client. By using notifications, the GATT server does not need to be awake in order to detect an acknowledgement packet. This allows for more power to be saved due to the RF peripheral not needing to be active.

The device will send UV data via notifications. When the UV data buffer on the device is full, the device will send its UV buffer to the mobile device. The total length of the data is 24 bytes, configured by the “Sensor Data” characteristic and based on the buffer size.

7.3.8 OTA Updates

The EFR32BG22 can be updated over the air (OTA), via bluetooth. By connecting the device to the EFR Connect mobile app, we can flash new firmware to the device without connecting a debugger.

7.4 Mobile App

7.4.1 Bluetooth Searching and Pairing

Upon startup, the app will launch an intent to connect to the wearable. This intent is just a screen that lists all valid UVtech devices that the mobile device can find. A RecyclerView is used to list all the found devices. Every device will have their name and MAC address displayed, along with a button to pair. The search period lasts for 60 seconds, but can be restarted by swiping down.

In order to filter the list to only show UVtech devices, the ScanFilter class is used to look for devices with a GATT service UUID of “23e490ae-dbed-41a6-8fda-b4a13d4cc679” and name “UV Sense”. The GATT service UUID corresponds to the “UV Tech Sense” service defined in the GATT profile of the wearable device.

7.4.2 Bluetooth Data

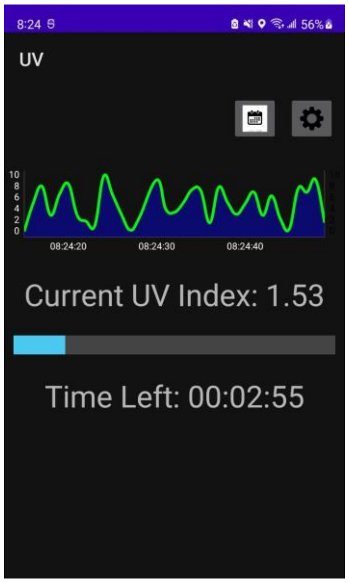
In order to get UV data from the device, GATT notifications need to be enabled for the “Sensor Data” GATT characteristic. This is done by first enabling notifications on the mobile device, via the “Client Characteristic Configuration” attribute (UUID: 00002902-0000-1000-8000-00805f9b34fb). Now that notifications are enabled for the GATT client, it now needs to be written to the GATT server (wearable device) to let it know that it can send notifications.

Once notifications are enabled for both devices, the wearable device will send its UV buffer every 3 seconds. The mobile device will receive this notification, in which it processes it and stores it into a 2D integer array. To convert the raw UV data to the UV index, we only need to divide the raw data by 100. Note that this division is done on the mobile device to save processing power/time on the wearable device. The 2D array is sorted with the 1st array representing the seconds and the 2nd array representing each sensor. For example, array[2][3] will give the UV index for sensor 3, at the 2nd second. The 2nd second represents a more recent time than the 0th second.

7.4.3 Data Storage

Data for each day is stored in an array of doubles, representing the UV index at each second during the day. The UV index is taken to be the average across all functioning sensors at a particular second. When the mobile app is paused, the array is written out to a text file. The text file can be read by the Calendar activity to retrieve the data for previous days, as well as by the Main activity to read data from the current day.

7.4.4 Line Chart



The main screen of the app features a line chart displaying to the user the UV index for each second. The line chart is updated every three seconds as new data arrives from the microcontroller. The chart also contains buttons to change the range of the line chart x axis, or how far back in time the user wishes to see data from. The default range is 1 hour, but the other buttons enable a range of 30 minutes or 15 minutes. The user can also modify the range manually by zooming in/out using two fingers. If the device has not been active for 1 hour/ 30 minutes/ 15 minutes, the line chart will only display the data from when the device became active. Below the line chart on the main screen is a text display of the current UV index. This is also updated every three seconds, and shows the most recent UV index.

7.4.5 Calendar

A Calendar view is used to display data from previous days as well as the current day. To get to the Calendar view, the user simply clicks a calendar icon on the main screen. When the user clicks on a day on the calendar, that day's timeline of UV exposure is displayed in a line chart below the calendar. Below the line chart, the day's total irradiation is shown to the user in units of Joules per square meter.

7.4.6 MED Timer

Below the line chart and current UV index on the main screen is a timer showing the estimated time until their minimal erythemal dose (MED) is reached. To do the estimation, we first take the sum of the UV index at each second and convert it to Watts per square meter. This value is then divided by the spf value to account for the UV blocking of the sunscreen. Subtracting the accrued radiation from the MED gives us the remaining amount of tolerable UV radiation. Dividing this value by the average UV radiation yields the estimated time until the MED threshold is reached. For the average UV radiation, we just take the range of UV index data from when the device was awakened, and convert this to Watts per square meter, accounting for the spf of the sunscreen. When the timer reaches zero, the app sends the user a notification letting them know that they have exceeded their MED. Above the time estimation, the app also shows a progress bar, giving the user a visual representation of their UV exposure with respect to their MED.

7.4.7 Settings Menu

The settings menu gives the user the ability to enter their skin type and the spf they are using into the app. Skin type and spf are chosen from separate Spinner widgets. The skin type is chosen from 6 options representing the Fitzpatrick scale, and the value entered is used to calculate the MED for use by the timer. Spf is chosen from 4 options representing <15, 15-29, 30-50, and >50, as these are the typical classifications of sunscreen. Changing either of these values immediately updates the timer.

7.5 PCB Design

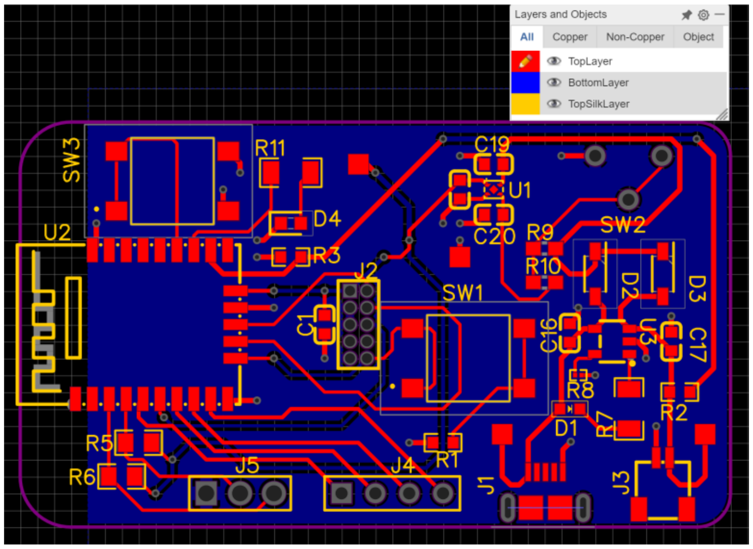


Figure 9. PCB of the prototype for the main board

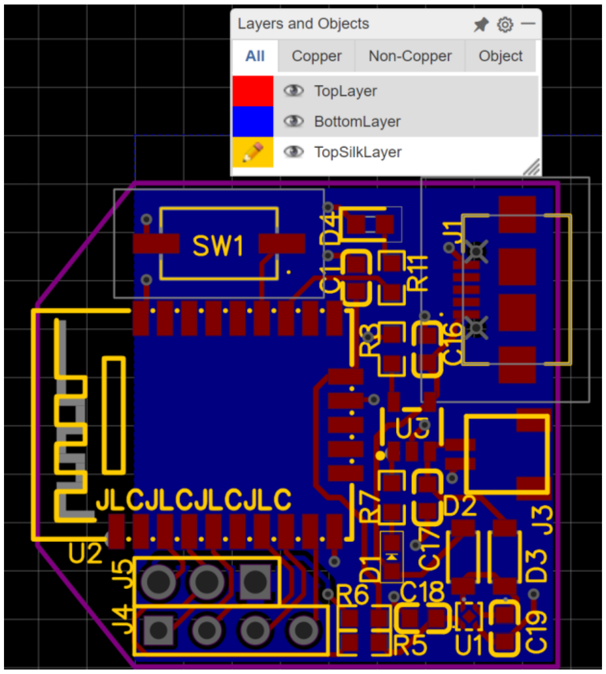


Figure 10. PCB for the final main board

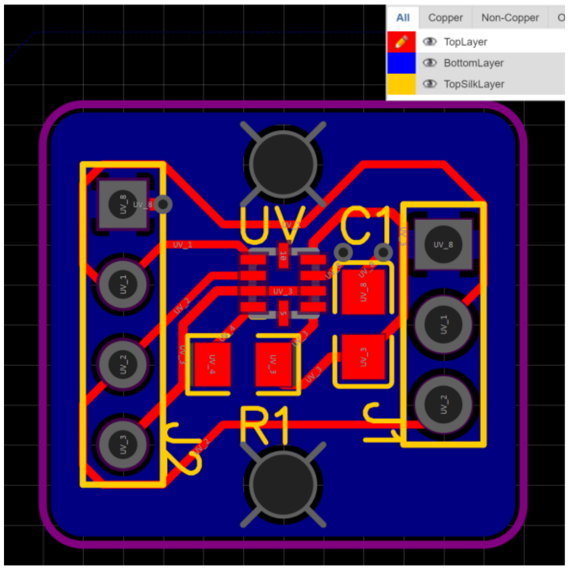


Figure 11. PCB for the UV sensor module

For the prototype main board PCB design, considerations were made to separate the microcontroller from the power supply. The left half was reserved for the microcontroller, while the right half was reserved for power. For the PCB traces, main power traces were given a larger width in comparison to other traces. The traces were also designed in order to avoid sharp angles, which can be important for PCB manufacturing and the EMI of high frequency lines. The only angles used in the PCB are 45 degree angles. Attention was placed to avoid overlap of traces on the bottom and top layer as they can cause parasitic capacitances, affecting the I2C bus. The last consideration is a ground plane on the bottom layer. The ground plane surrounds the entire PCB except for the antenna, which could cause RF interference.

The final main board PCB also took these considerations, but was designed to be more compact and remove unnecessary features. The debugging header J2, the reset button SW1 (on the prototype, not the final), power switch SW2, and the voltage divider for the battery R2 and R3, were removed. These were deemed not necessary anymore and were only in the prototype for testing purposes. In addition, the test pads were removed from the final. Lastly, smaller resistors, capacitors and the push buttons were used in the final. To keep the final PCB more neat looking, a uniform resistor and capacitor size of 0603 (inches) were used. The final PCB also avoided the need to route lines in the bottom layer, except for one located near J5.

The dimensions for the prototype PCB are 53.34 mm x 30.48mm. The dimensions for the final PCB are 27.305 mm x 25.400 mm. The dimensions for the UV sensor module PCB are 15.24 mm x 13.97mm. The goal of the PCB was to make it small enough so it can be embedded into nearly all pieces of clothing.

7.6 Physical Product

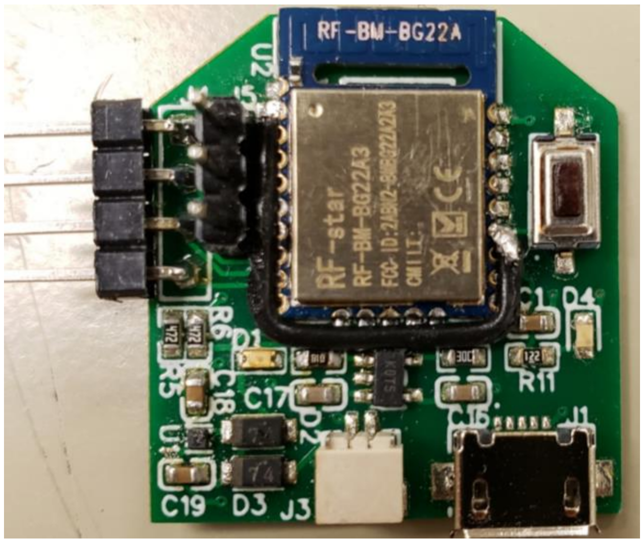


Figure 12. Assembled main board

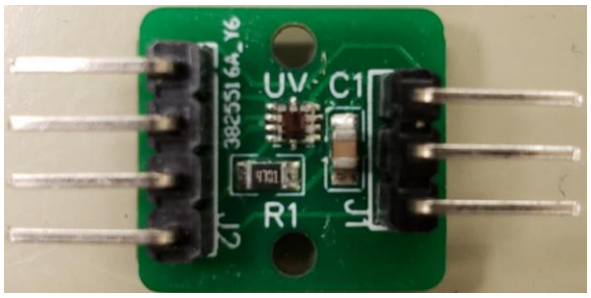


Figure 13. Assembled UV sensor module

Figure 14. Demo hat with the mainboard and 4 sensors embedded around the hat

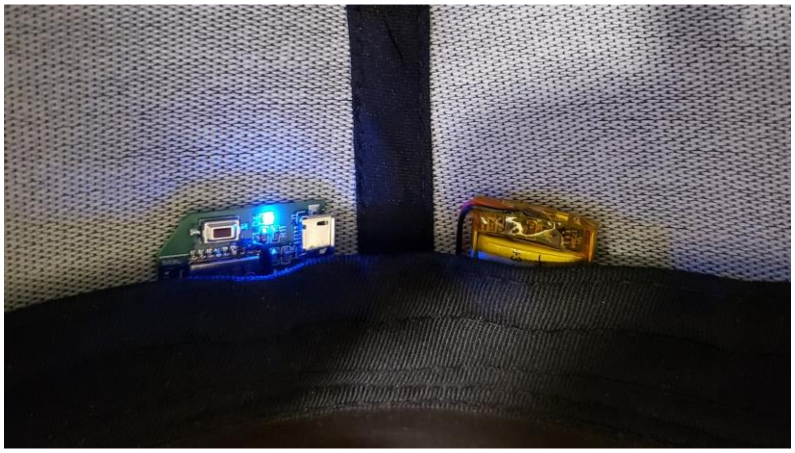


Figure 15. Inside the front of the demo hat, with the mainboard and battery hidden

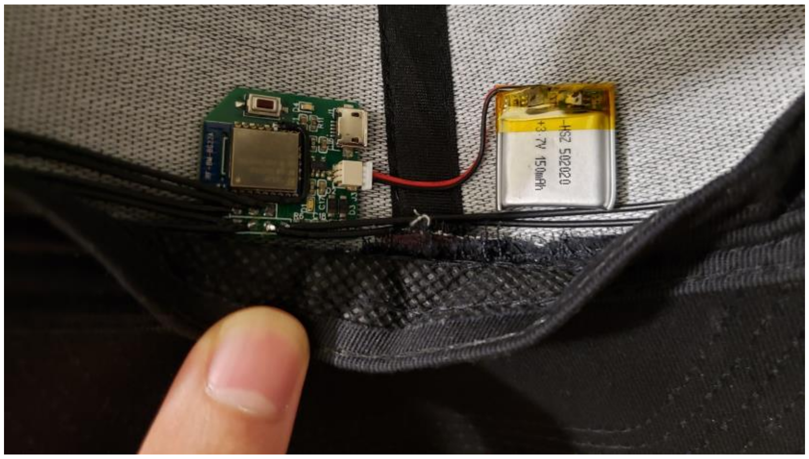


Figure 16. Inside the front of the demo hat, with the mainboard and battery exposed.

The system was embedded into a hat to demonstrate how it can be embedded into clothing. The hat features 4 sensors, located around the hat behind the mesh. The mesh design was chosen on purpose because that way, we can have the sensors exposed to the outside. This doesn't mean that the sensor can't be embedded in non-mesh materials. As long as there is a tiny hole that the sensor can peek out from, then UV data can be recorded.

Each sensor is linked together on the same I2C bus. Because of the I2C pass-through pins in the sensor module PCB, we only needed to connect the power pins from each sensor to the main board. Only one of the sensors needs to be connected to the I2C pins on the main board. This design improves the embeddability of the system.

Another neat design feature was having all the user interfacing on one side of the main board. In figure 15, we can see that the button, LED and charging port is available without having to expose the entire board.

With the addition of OTA firmware updates, the system is able to be upgraded and can even be interfaced with different types of sensors in the future.

7.6.1 Current Consumption and Battery Life

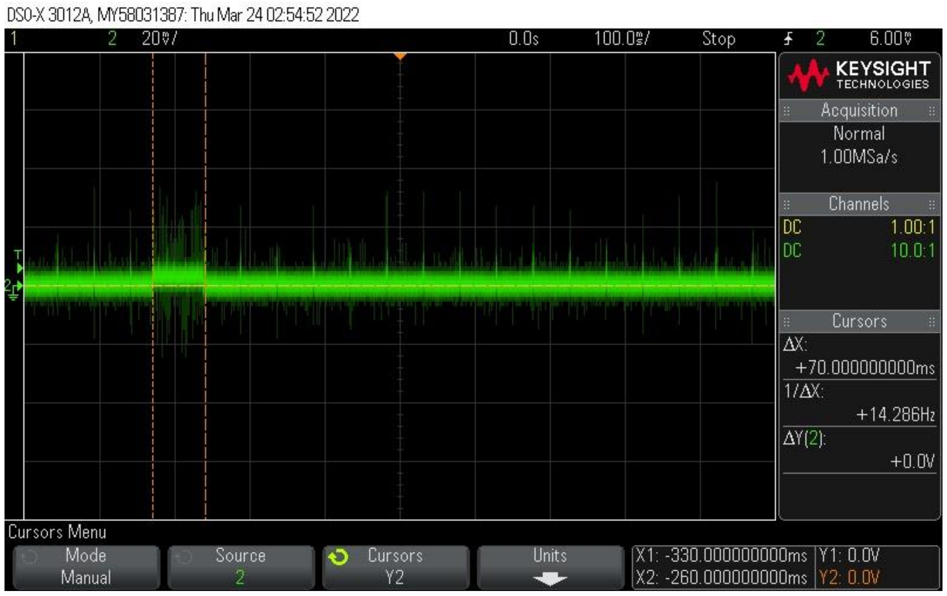


Figure 17. One second oscilloscope reading of the device operating

In order to measure the current consumption of the device, a shunt resistor of 1 Ω was connected in series, right after the load and before the ground of the battery. Since the current consumption of the device is very low, there is a lot of noise picked up, but a clear shift in voltage level can be seen. What we see in Figure 17 is the duration that the microcontroller is woken up for. We can also see that it is woken up every 1 second.

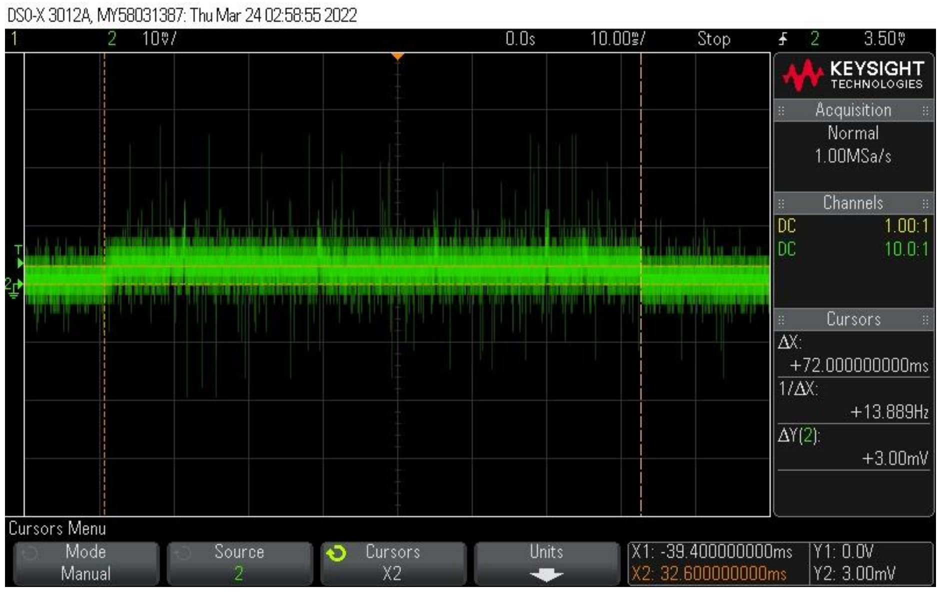


Figure 18. Zoomed in oscilloscope reading of the device while it is awake

Upon closer inspection, we can see that the microcontroller is awake for 72 ms. To determine the voltage level, y-cursors were placed in the center of all the noise. We can see that the voltage across the shunt resistor is around 3 mV when the device is awake. Because the shunt resistor is only 1 **Ω**, the device uses around 3 mA, 7.2% of the time (72 ms/1000 ms = 7.2%). In order to calculate the expected battery life, we divide its capacity with the device's current consumption:

150 mAh / (3 mA * 7.2%) = 694.444444444 hours = 28.9 days

Assuming batteries charged to 80% of full capacity:

(150 mAh * 80%) / (3 mA * 7.2%) = 555.55555556 hours = 23.1 days

The estimated battery life, based on the actual measured current consumption, exceeds the initial specification of 7 days.

7.6.2 Push Button GPIO Oversight

In figure 12, we can see that there is a black wire that has been manually soldered to the MCU. This wire connects pins PB02 to PC01. The reason for this being that ports C and D can not wake up the MCU from EM2+ deep sleep, but ports A and B can. This was not found during testing because testing of the buttons was done with the debugger plugged in. By having the debugger plugged in, the MCU was kept from going into EM2+ deep sleep. To resolve this problem, a wire was connected to PB02 to PC01, connecting the push button to PB02. The firmware was then updated using OTA bluetooth updates, reconfiguring the push button pin to PB02.

8. Testing and Verification

One of the tests that needed to be conducted was whether or not UV data was being measured from each of the sensors. To verify this, every sensor was covered except for one. This test was conducted outdoors on days with clear sun. With every 3 second period, the UV data should be sent to the mobile device. All sensors but one should return a UV index of 0. By doing this test for every sensor, we were able to record the UV index measured by each sensor. If all 4 sensors returned the same UV index individually, then we can say that each sensor is properly interfaced with the main board.

To test the modularity of the sensors, sensors were added to the main board one by one. For each sensor that was not connected, the mobile device should receive 0xFFFF for the UV data. We were able to verify that the sensors could be added and removed with ease. We were also testing the reinitializing function of the push button with this test.

Various voltage tests were performed throughout the system to see if there were any shorts or opens within the device. Also functionality of the schottky OR circuit was tested by plugging in both power sources. We were able to verify that the source with a higher voltage supplies the system and that +5V was not being fed to the battery directly.

Testing for the mobile app involved leaving the system out in the sun for one hour on separate days. Doing this allowed us to verify if the timeline and calendar was working. To verify the sun screen timer, hardcoded values were fed to the app. This way, we can calculate the theoretical time and compare it with the time that our app returned.

9. Conclusion and Discussion

As a result of the research, many design considerations and hardwork, the final device has been created. The device presented in this report is cheap, energy efficient and small. Future proofing of this system has been considered, with the use of OTA firmware updates and modular I2C sensors. The embedded firmware has also been thought up with efficiency in mind, to save battery power and improve the lifespan of the system. The app was designed to be simple and introduce the data in a friendly manner.

One ethical concern was battery safety. Since this is intended to be a consumer product, battery safety is important. The risk of an exploding lithium battery was reduced by keeping the battery healthy. A small charge current of 50 uA and an 80% charging limit was implemented in the design to ensure a healthy battery. The battery also contains overcharging and undercharging, which would damage the battery.

Another ethical concern was the possibility of electronic waste. That is why the product was also designed with future proofing in mind. The device firmware can be updated via OTA over bluetooth. And the implementation of modular sensors means that different and better types of sensors can be used in the future. With a device that won’t go obsolete quickly, we can expect limited electronic waste with our product.

References

[1] How much sun is too much?, 29-Nov-2018. [Online]. Available: https://www.ncbi.nlm.nih.gov/books/NBK321117/. [Accessed: 11-Sep-2021].

[2] “Ultraviolet (UV) radiation,” Ultraviolet (UV) Radiation. [Online]. Available: https://www.cancer.org/cancer/cancer-causes/radiation-exposure/uv-radiation.html. [Accessed: 10-Sep-2021].

[3] UV Radiation, 28-Jun-2021. [Online]. Available: https://www.cdc.gov/nceh/features/uv-radiation-safety/index.html. [Accessed: 12-Sep-2021].

[4] “UV Radiation,” The Skin Cancer Foundation. [Online]. Available: https://www.skincancer.org/risk-factors/uv-radiation/. [Accessed: 10-Sep-2021].

[5] What does the SPF rating really mean? [Online]. Available: https://www.science.org.au/curious/people-medicine/what-does-spf-rating-really-mean. [Accessed: 11-Sep-2021].

[6] Silicon Labs, “EFR32BG22 Wireless Gecko SoC Family Data Sheet,” EFR32BG22C224F512GM32 datasheet, Apr. 2019 [Revised June. 2021].

[7] Silicon Labs, “EFR32xG22 Wireless Gecko Reference Manual,” EFR32BG22C224F512GM32 manual, Jul. 2019 [Revised Aug. 2020].

[8] STMicroelectronics, “STM32WB15CC Datasheet - production data,” STM32WB15CC datasheet, Feb. 2021 [Revised Aug. 2021].

[9] Espressif Systems, “ESP32-SOLO-1 Datasheet,” ESP32-SOLO-1 datasheet, Jun. 2018 [Revised Feb. 2021].

[10] Espressif Systems, “ESP32 Series Datasheet,” ESP32 family datasheet, Aug. 2016 [Revised Oct. 2021].

[11] Nordic Semiconductor, “nRF52832 Product Specification,”, nRF52832 datasheet, Feb. 2016 [Revised Oct. 2017].

[12] Silicon Labs, “UV INDEX AND AMBIENT LIGHT SENSOR IC WITH I2C INTERFACE,” Si1132 datasheet, 2014.

[13] Vishay Semiconductors, “UV A Light Sensor with I2C Interface,” VEML6070 datasheet, Oct. 2012 [Revised Jul. 2015].

[14] GENUV, “UV-A Sensor,” GUVA-S12SD datasheet, Jul. 2011 [Revised 2018].

[15] Microchip, “Miniature Single-Cell, Fully Integrated Li-Ion, Li-Polymer Charge Management Controllers,” MCP73831T-2ATI datasheet, Nov. 2005 [Revised June. 2020].

[16] Rochester Electronics, “28V-Input-Voltage Single-CellLi-Ion Battery Charger with 10mA Regulator,” MC34675AEPR2 datasheet, Apr. 2008 [Revised Apr. 2008]

[17] Onsemi, “200 mA, Very-Low Quiescent Current, IQ 25 A, Low Noise, Low Dropout Regulator,” NCP707CMX300TCG datasheet, 2015 [Revised Apr. 2017]

[18] I. Buchmann, “BU-808: How to prolong lithium-based batteries,” How to prolong lithium-based batteries, 2019. [Online]. Available: https://batteryuniversity.com/article/bu-808-how-to-prolong-lithium-based-batteries. [Accessed: 01-Oct-2021].

[19] de Paula Correa, Marcelo & Godin-Beekmann, Sophie & Haeffelin, Martial & Bekki, Slimane & Saiag, Philippe & Badosa, Jordi & Jegou, F. & Pazmiño, Andrea & Mahé, Emmanuel. (2013). Projected changes in clear-sky erythemal and vitamin D effective UV doses for Europe over the period 2006 to 2100. Photochemical & photobiological sciences : Official journal of the European Photochemistry Association and the European Society for Photobiology. 12. 10.1039/c3pp50024a.

[20] Downs, N., Parisi, A.V., Galligan, L., Turner, J., Amar, A., King, R., Ultra, F., & Butler, H. (2016). Solar radiation and the UV index: An application of numerical integration, trigonometric functions, online education and the modeling process. International Journal of Research in Education and Science (IJRES), 2(1), 179-189.

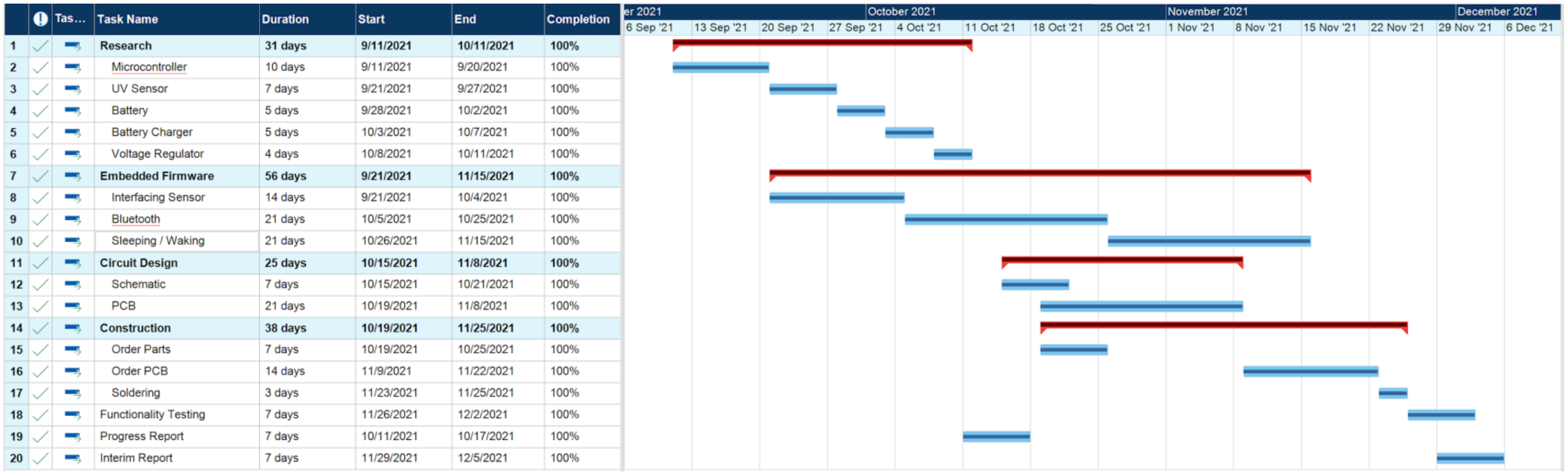
[21] Sayre RM, Desrochers DL, Wilson CJ, Marlowe E. Skin type, minimal erythema dose (MED), and sunlight acclimatization. J Am Acad Dermatol. 1981 Oct;5(4):439-43. doi: 10.1016/s0190-9622(81)70106-3. PMID: 7287960.

[22] F. Massen, “The UV-index UVI,” *The UV-Index (UVI)*. [Online]. Available: https://meteo.lcd.lu/papers/uv/uvi/uvi_01.html. [Accessed: 18-Apr-2022].

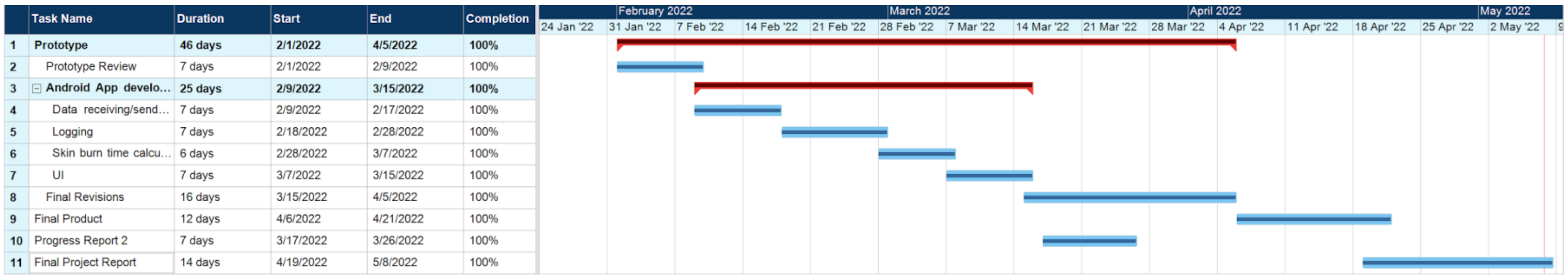
[23] “Skin cancer facts & statistics,” *Skin Cancer Facts & Statistics -What You Need to Know*, 08-Jan-2022. [Online]. Available: https://www.skincancer.org/skin-cancer-information/skin-cancer-facts/. [Accessed: 09-May-2022].

[24] “Skin cancer,” *Skin Cancer*, 22-Apr-2022. [Online]. Available: https://www.aad.org/media/stats-skin-cancer. [Accessed: 09-May-2022].

Appendix A - Program Management



		Tas…	Task Name	Duration	Start	End	Completion
1	✓		**Research**	**31 days**	**9/11/2021**	**10/11/2021**	**100%**
2	✓		Microcontroller	10 days	9/11/2021	9/20/2021	100%
3	✓		UV Sensor	7 days	9/21/2021	9/27/2021	100%
4	✓		Battery	5 days	9/28/2021	10/2/2021	100%
5	✓		Battery Charger	5 days	10/3/2021	10/7/2021	100%
6	✓		Voltage Regulator	4 days	10/8/2021	10/11/2021	100%
7	✓		**Embedded Firmware**	**56 days**	**9/21/2021**	**11/15/2021**	**100%**
8	✓		Interfacing Sensor	14 days	9/21/2021	10/4/2021	100%
9	✓		Bluetooth	21 days	10/5/2021	10/25/2021	100%
10	✓		Sleeping / Waking	21 days	10/26/2021	11/15/2021	100%
11	✓		**Circuit Design**	**25 days**	**10/15/2021**	**11/8/2021**	**100%**
12	✓		Schematic	7 days	10/15/2021	10/21/2021	100%
13	✓		PCB	21 days	10/19/2021	11/8/2021	100%
14	✓		**Construction**	**38 days**	**10/19/2021**	**11/25/2021**	**100%**
15	✓		Order Parts	7 days	10/19/2021	10/25/2021	100%
16	✓		Order PCB	14 days	11/9/2021	11/22/2021	100%
17	✓		Soldering	3 days	11/23/2021	11/25/2021	100%
18	✓		Functionality Testing	7 days	11/26/2021	12/2/2021	100%
19	✓		Progress Report	7 days	10/11/2021	10/17/2021	100%
20	✓		Interim Report	7 days	11/29/2021	12/5/2021	100%



	Task Name	Duration	Start	End	Completion
1	**Prototype**	**46 days**	**2/1/2022**	**4/5/2022**	**100%**
2	Prototype Review	7 days	2/1/2022	2/9/2022	100%
3	**Android App develo…**	**25 days**	**2/9/2022**	**3/15/2022**	**100%**
4	Data receiving/send…	7 days	2/9/2022	2/17/2022	100%
5	Logging	7 days	2/18/2022	2/28/2022	100%
6	Skin burn time calcu…	6 days	2/28/2022	3/7/2022	100%
7	UI	7 days	3/7/2022	3/15/2022	100%
8	Final Revisions	16 days	3/15/2022	4/5/2022	100%
9	Final Product	12 days	4/6/2022	4/21/2022	100%
10	Progress Report 2	7 days	3/17/2022	3/26/2022	100%
11	Final Project Report	14 days	4/19/2022	5/8/2022	100%

Tennyson Cheng: Researched the components for the device, designed the schematic and PCB. Wrote embedded firmware and created the final hardware.

Judah Ben-Eliezer: Designed and wrote the mobile application for android. Implementation of the SPF timer, timeline and calendar.

Appendix B - Meeting Records

Meeting Date	Duration	Attendance	Topic
September 15, 2021	15 minutes	Tennyson Cheng, Judah Ben-Eliezer, Dmitri Donetski	Components research and overall design
September 20, 2021	20 minutes	Tennyson Cheng, Judah Ben-Eliezer	Components Research
September 27, 2021	5 minutes	Tennyson Cheng, Judah Ben-Eliezer	Interfacing sensors with software
October 4, 2021	15 minutes	Tennyson Cheng, Judah Ben-Eliezer	Addressing multiple I2C devices with the same address on the same I2C bus
October 11, 2021	10 minutes	Tennyson Cheng, Judah Ben-Eliezer	Schematic and PCB considerations
November 1, 2021	5 minutes	Tennyson Cheng, Judah Ben-Eliezer	Schematic and PCB considerations
November 8, 2021	15 minutes	Tennyson Cheng, Judah Ben-Eliezer	Analog components selection
November 29, 2021	5 minutes	Tennyson Cheng, Dmitri Donetski	Schematic review and changes
February 7, 2022	15 minutes	Tennyson Cheng, Judah Ben-Eliezer	Changes for the final design and the mobile app implementation
February 28, 2022	10 minutes	Tennyson Cheng, Judah Ben-Eliezer	Mobile app progress
March 15, 2022	10 minutes	Tennyson Cheng, Judah Ben-Eliezer	Mobile app progress
April 10, 2022	5 minutes	Tennyson Cheng, Judah Ben-Eliezer	Mobile app progress and final device testing

Appendix C - Cost of Components

Product Mfr # - (case code in.)	Qty	Unit Price	Total	Schematic Part Label
RF-BM-BG22A2	2	$3.08	$6.16	U2
ASR00003	2	$2.48	$4.96	BT1
NCP707CMX300TCG	2	$0.46	$0.92	U1
MCP73831T-2ATI/OT	2	$0.69	$1.38	U3
UJ2-MIBH-G-SMT-TR	2	$0.45	$0.90	J1
LB Q39E-N1OO-35-1 (0603)	2	$0.41	$0.82	D4
LY Q396-P1Q2-36 (0603)	2	$0.38	$0.76	D1
CL10A225KQ8NNNC (0603)	2	$0.10	$0.20	C1 [Main board]
CL10B105KP8NNNC (0603)	6	$0.10	$0.60	C18, C19, C20
CL10A475KP8NNND (0603)	4	$0.10	$0.40	C16, C17
RC0603JR-134K7L (0603)	4	$0.10	$0.40	R5, R6
RE0603FRE0720KL (0603)	2	$0.10	$0.20	R8
AC0603FR-101KL (0603)	2	$0.10	$0.20	R7
AC0603FR-131K2L (0603)	2	$0.10	$0.20	R11
RB160M-40TR	4	$0.27	$1.08	D2, D3
PTS636 SK25J SMTR LFS	2	$0.18	$0.36	SW2
SI1132-A10-GMR	8	$4.04	$32.32	UV [UV module]
CL21A106KQFNNNE (0805)	8	$0.12	$0.96	C1 [UV module]
CRCW08054K70FKEA (0805)	8	$0.13	$1.04	R1 [UV module]

Note: Enough parts were purchased to build 2 devices.

Note: Price of analog components like resistors and capacitors skew the price of a single unit since they are usually sold in bulk.

Grand Total: **$53.86**

Total cost to build entire system (1 main board and 4 sensors): **$26.93**

Appendix D - Embedded Firmware Source Code

app.h

```c
/***************************************************************************//**
 * @file
 * @brief Application interface provided to main().
 *******************************************************************************
 * # License
 * <b>Copyright 2020 Silicon Laboratories Inc. www.silabs.com</b>
 *******************************************************************************
 *
 * SPDX-License-Identifier: Zlib
 *
 * The licensor of this software is Silicon Laboratories Inc.
 *
 * This software is provided 'as-is', without any express or implied
 * warranty. In no event will the authors be held liable for any damages
 * arising from the use of this software.
 *
 * Permission is granted to anyone to use this software for any purpose,
 * including commercial applications, and to alter it and redistribute it
 * freely, subject to the following restrictions:
 *
 * 1. The origin of this software must not be misrepresented; you must not
 *    claim that you wrote the original software. If you use this software
 *    in a product, an acknowledgment in the product documentation would be
 *    appreciated but is not required.
 * 2. Altered source versions must be plainly marked as such, and must not be
 *    misrepresented as being the original software.
 * 3. This notice may not be removed or altered from any source distribution.
 *
 ******************************************************************************/

#ifndef APP_H
#define APP_H

#include "em_common.h"

#define MAX_SENSORS 0x04
#define MAX_BUFFER 0x03                  //must be greater than 0

typedef struct Sensor {
        bool active;
        uint8_t power_port;
        uint8_t power_pin;
        uint8_t addr;
}Sensor;

typedef enum DEVICE_STATE {
        OFF, ADVERTISING, CONNECTED
}DEVICE_STATE;

void sensor_init();
void measure_uv();

/**************************************************************************//**
 * Application Init.
 ******************************************************************************/
void app_init(void);

/**************************************************************************//**
 * Application Process Action.
 *****************************************************************************/
void app_process_action(void);

#endif // APP_H
```

app.c

```
/***************************************************************************//**
 * @file
 * @brief Core application logic.
 *******************************************************************************
 * # License
 * <b>Copyright 2020 Silicon Laboratories Inc. www.silabs.com</b>
 *******************************************************************************
 *
 * SPDX-License-Identifier: Zlib
 *
 * The licensor of this software is Silicon Laboratories Inc.
 *
 * This software is provided 'as-is', without any express or implied
 * warranty. In no event will the authors be held liable for any damages
 * arising from the use of this software.
 *
 * Permission is granted to anyone to use this software for any purpose,
 * including commercial applications, and to alter it and redistribute it
 * freely, subject to the following restrictions:
 *
 * 1. The origin of this software must not be misrepresented; you must not
 *    claim that you wrote the original software. If you use this software
 *    in a product, an acknowledgment in the product documentation would be
 *    appreciated but is not required.
 * 2. Altered source versions must be plainly marked as such, and must not be
 *    misrepresented as being the original software.
 * 3. This notice may not be removed or altered from any source distribution.
 *
 ******************************************************************************/
#include "em_common.h"
#include "app_assert.h"
#include "sl_bluetooth.h"
#include "gatt_db.h"
#include "app.h"

#include "em_cmu.h"
#include "int_enum.h"
#include "gpio.h"
#include "timer.h"
#include "i2c.h"
#include "si1132.h"

//The advertising set handle allocated from Bluetooth stack.
static uint8_t advertising_set_handle = 0xff;
static uint8_t connection_id = 0xFF;
static Sensor sensors[MAX_SENSORS] = {{false, gpioPortA, 0, DEFAULT_ADDR},
                                      {false, gpioPortA, 4, DEFAULT_ADDR},
                                      {false, gpioPortA, 5, DEFAULT_ADDR},
                                      {false, gpioPortA, 6, DEFAULT_ADDR}};
static uint16_t UV_data[MAX_SENSORS*MAX_BUFFER];
static uint8_t buffer_index = 0;
static uint8_t buffer_size = 0;
static bool notify_en = false;
static DEVICE_STATE state = OFF;

/**************************************************************************//**
 * Application Init.
 *****************************************************************************/
void app_init(void)
{
        //INITIALIZE GPIO
        gpio_init();

        //INITIALIZE I2C
        i2c_init();

        //INITIALIZE TIMER
        timer_init_();

        //INITIALIZE SENSORS
        sensor_init();
}

/**************************************************************************//**
 * Application Process Action.
 *****************************************************************************/
SL_WEAK void app_process_action(void)
{
        /////////////////////////////////////////////////////////////////////////////
        // Put your additional application code here!                              //
        // This is called infinitely.                                              //
        // Do not call blocking functions from here!                               //
        /////////////////////////////////////////////////////////////////////////////
}

/**************************************************************************//**
 * Bluetooth stack event handler.
 * This overrides the dummy weak implementation.
 *
 * @param[in] evt Event coming from the Bluetooth stack.
 *****************************************************************************/
void sl_bt_on_event(sl_bt_msg_t *evt)
{
  sl_status_t sc;
  bd_addr address;
  uint8_t address_type;
  uint8_t system_id[8];
  switch (SL_BT_MSG_ID(evt->header)) {
    // -------------------------------
    // This event indicates the device has started and the radio is ready.
    // Do not call any stack command before receiving this boot event!
        case sl_bt_evt_system_boot_id:
                // Extract unique ID from BT Address.
```

```
            sc = sl_bt_system_get_identity_address(&address, &address_type);
            app_assert_status(sc);

            // Pad and reverse unique ID to get System ID.
            system_id[0] = address.addr[5];
            system_id[1] = address.addr[4];
            system_id[2] = address.addr[3];
            system_id[3] = 0xFF;
            system_id[4] = 0xFE;
            system_id[5] = address.addr[2];
            system_id[6] = address.addr[1];
            system_id[7] = address.addr[0];

            sc = sl_bt_gatt_server_write_attribute_value(gattdb_system_id,

      0,

      sizeof(system_id),

      system_id);
            app_assert_status(sc);

            // Define default connection parameters.
            sc = sl_bt_connection_set_default_parameters(
                    800,    //min. connection interval: 1000 ms
                    880,    //max. connection interval: 1100 ms
                    1,              //latency [maximum missed connection events]: 1 event
                    550,    //timeout: 5500 ms
                    0,              //min. connection event length: 0 ms
                    1);             //max. connection event length: 0.675 ms
            app_assert_status(sc);

            // Set default preferred PHY to 2M PHY.
            // Note: All PHYs are accepted, just that 2M PHY is preferred
            sl_bt_connection_set_default_preferred_phy(0x02, 0xFF);

            // Create an advertising set.
            sc = sl_bt_advertiser_create_set(&advertising_set_handle);
            app_assert_status(sc);

            // Define advertising set timings.
            sc = sl_bt_advertiser_set_timing(
                    advertising_set_handle,
                    1600,   // min. adv. interval (milliseconds * 1.6): 1000 ms
                    1600,   // max. adv. interval (milliseconds * 1.6): 1000 ms
                    0,              // adv. duration: 0 ms = no time limit
                    30);    // max. num. adv. events: 30 events (60 seconds of advertising)
            app_assert_status(sc);
            break;

    // -------------------------------
    // This event indicates that a new connection was opened.
    case sl_bt_evt_connection_opened_id:
        // Update state to CONNECTED
        state = CONNECTED;
        // Store connection id
        connection_id = evt->data.evt_connection_opened.connection;
            // Reset buffer index and size;
            buffer_index = 0;
            buffer_size = 0;
            // Stop advertising LED.
            timer_led_advertise_stop();
    break;

// -------------------------------
// This event indicates that a connection was closed.
    case sl_bt_evt_connection_closed_id:
            // Update state to ADVERTISING
            state = ADVERTISING;
            // Stop the 1 second timer for UV measurements.
            if (notify_en) {
                    sc = timer_sensor_stop();
                    app_assert_status(sc);
                    notify_en = false;
            }
            // Restart advertising after client has disconnected.
            sc = sl_bt_advertiser_start(
                    advertising_set_handle,
                    sl_bt_advertiser_general_discoverable,
                    sl_bt_advertiser_connectable_scannable);
            app_assert_status(sc);
            // Start advertising LED.
            timer_led_advertise_start();
            break;

///////////////////////////////////////////////////////////////////////////
// Add additional event handlers here as your application requires!      //
///////////////////////////////////////////////////////////////////////////

    case sl_bt_evt_advertiser_timeout_id:
            // Update state to OFF
            state = OFF;
            // Stop advertising LED.
            timer_led_advertise_stop();
            break;

    //NOTE: This event handler also triggers when an indication acknowledge
    //is received. We do not need to check for that acknowledge since we
    //we are using notifications.
    case sl_bt_evt_gatt_server_characteristic_status_id:
            if (evt->data.evt_gatt_server_characteristic_status.characteristic
                    != gattdb_sensor_data) {
                    // Exit if event was not caused by a change to sensor_data.
                    break;
            }

            if (evt->data.evt_gatt_server_characteristic_status.client_config_flags
                    == gatt_notification) {
                    // If gatt notifications were enabled,
                    // Reset buffer index and size,
                    // Start a 1 second timer for UV measurements.
                    notify_en = true;
                    buffer_index = 0;
                    buffer_size = 0;
                    sc = timer_sensor_start();
                    app_assert_status(sc);
            }
            else if (evt->data.evt_gatt_server_characteristic_status.client_config_flags
                    == gatt_disable) {
                    // If gatt notifications were disabled,
                    // Stop the 1 second timer for UV measurements.
                    notify_en = false;
                    sc = timer_sensor_stop();
                    app_assert_status(sc);
            }
            break;

    case sl_bt_evt_system_external_signal_id:
            // External signal triggered from sensor timer.
            if (evt->data.evt_system_external_signal.extsignals == INT_TIMER_SENSOR) {
                    CMU_ClockDivSet(cmuClock_HCLK, 8);
                    gpio_led_on();

                    measure_uv();
                    if (buffer_size == MAX_BUFFER) {
                            sl_bt_gatt_server_notify_all(
                                    gattdb_sensor_data,
                                    MAX_SENSORS*MAX_BUFFER*sizeof(uint16_t),
                                    (uint8_t *)UV_data);
                            buffer_index = 0;
                            buffer_size = 0;
                    }

                    gpio_led_off();
                    CMU_ClockDivSet(cmuClock_HCLK, 1);
            }
            else if (evt->data.evt_system_external_signal.extsignals == INT_BUTTON_PRESS) {
                    if (state == OFF) {
                            gpio_led_on();
                            sensor_init();
                            gpio_led_off();
                    }
            }
            else if (evt->data.evt_system_external_signal.extsignals == INT_BUTTON_HOLD) {
                    if (state == OFF) {
                            // Start advertising after button hold
                            state = ADVERTISING;

                            // Start general advertising and enable connections.
                            sc = sl_bt_advertiser_start(
                                    advertising_set_handle,
                                    sl_bt_advertiser_general_discoverable,
                                    sl_bt_advertiser_connectable_scannable);
                            app_assert_status(sc);

                            // Start advertising LED.
                            timer_led_advertise_start();
                    }
                    else if (state == CONNECTED) {
                            // Disconnect from bluetooth device and start advertising
                            sc = sl_bt_connection_close(connection_id);
                            app_assert_status(sc);
                            state = ADVERTISING;
                    }
                    else {
                            __NOP();
                    }
                }
                else if (evt->data.evt_system_external_signal.extsignals == INT_LED_OFF) {
                        gpio_led_off();
                }
                else if (evt->data.evt_system_external_signal.extsignals == INT_LED_ON) {
                        gpio_led_on();
                }
                break;

    // -------------------------------
    // Default event handler.
        default:
                break;
  }
}

void sensor_init() {
        for (uint8_t i = 0; i < MAX_SENSORS; i++) {
                if (si1132_init(sensors[i].power_port, sensors[i].power_pin, I2C0, DEFAULT_ADDR+i+1)) {
                        sensors[i].addr = DEFAULT_ADDR+i+1;
                        sensors[i].active = true;
                        for (uint8_t o = 0; o < MAX_SENSORS*MAX_BUFFER; o += MAX_SENSORS) {
                                UV_data[o+i] = 0x0000;
                        }
                }
                else {
                        GPIO_PinOutClear(sensors[i].power_port, sensors[i].power_pin);
                        sensors[i].active = false;
                        for (uint8_t o = 0; o < MAX_SENSORS*MAX_BUFFER; o += MAX_SENSORS) {
                                UV_data[o+i] = 0xFFFF;
                        }
                }
        }
}

void measure_uv() {
        for (uint8_t i = 0; i < MAX_SENSORS; i++) {
                if (sensors[i].active) {
                        si1132_UV_start_measurement(I2C0, sensors[i].addr);
                        si1132_UV_read_measurement(I2C0, sensors[i].addr, &UV_data[buffer_index]);
                }
                buffer_index++;
        }
        buffer_size++;
}
```

i2c.h

```
/*
 * i2c.h
 *
 *  Created on: Jan 26, 2022
 *      Author: Judah Ben-Eliezer, Tennyson Cheng
 */

#ifndef I2C_H_
#define I2C_H_

#include "em_i2c.h"
```

```
/**************************************************************************//**
 * @brief
 *   Initializes I2C0 for fast mode.
 *   Uses PB0 for SDA and PB1 for SCLK.
 *****************************************************************************/
void i2c_init(void);

/**************************************************************************//**
 * @brief
 *   Writes 1 byte to a register of slave.
 *
 * @param[in] i2c
 *   Pointer to I2C peripheral register block.
 *
 * @param[in] slave_addr.
 *   Address of slave.
 *
 * @param[in] reg_addr
 *   Internal register address to write to.
 *
 * @param[in] data
 *   Byte to write.
 *****************************************************************************/
I2C_TransferReturn_TypeDef i2c_write_single(I2C_TypeDef *i2c, uint8_t slave_addr, uint8_t reg_addr,
uint8_t data);

/**************************************************************************//**
 * @brief
 *   Writes byte array to slave.
 *
 * @param[in] i2c
 *   Pointer to I2C peripheral register block.
 *
 * @param[in] slave_addr.
 *   Address of slave.
 *
 * @param[in] reg_addr
 *   Internal register address to begin write.
 *
 * @param[in] data
 *   Pointer to array of data to write.
 *
 * @param[in] num_bytes
 *   Number of bytes to write.
 *****************************************************************************/
I2C_TransferReturn_TypeDef i2c_write_burst(I2C_TypeDef *i2c, uint8_t slave_addr, uint8_t reg_addr,
uint8_t* data, uint16_t num_bytes);

/**************************************************************************//**
 * @brief
 *   Reads 1 byte from a register of slave.
 *
 * @param[in] i2c
 *   Pointer to I2C peripheral register block.
 *
 * @param[in] slave_addr.
 *   Address of slave.
 *
 * @param[in] reg_addr
 *   Internal register address to read from.
 *
 * @param[in] buffer
 *   Buffer to read byte into.
 ******************************************************************************/
I2C_TransferReturn_TypeDef i2c_read_single(I2C_TypeDef *i2c, uint8_t slave_addr, uint8_t reg_addr,
uint8_t *buffer);

/***************************************************************************//**
 * @brief
 *   Reads byte array from slave.
 *
 * @param[in] i2c
 *   Pointer to I2C peripheral register block.
 *
 * @param[in] slave_addr.
 *   Address of slave.
 *
 * @param[in] reg_addr
 *   Internal register address to begin read.
 *
 * @param[in] buffer
 *   Buffer to store bytes in.
 *
 * @param[in] num_bytes
 *   Number of bytes to read.
 ******************************************************************************/
I2C_TransferReturn_TypeDef i2c_read_burst(I2C_TypeDef *i2c, uint8_t slave_addr, uint8_t reg_addr,
uint8_t *buffer, uint16_t num_bytes);

#endif /* I2C_H_ */
```

<u>i2c.c</u>

```
/*
 * i2c.c
 *
 *  Created on: Jan 26, 2022
 *      Author: Judah Ben-Eliezer, Tennyson Cheng
 */

#include "em_gpio.h"
#include "em_i2c.h"
#include "em_cmu.h"
#include "i2c.h"

void i2c_init(void) {
        //Enable I2C Clock
        CMU_ClockEnable(cmuClock_I2C0, true);

        //Initialize I2C configuration
        I2C_Init_TypeDef i2cInit = I2C_INIT_DEFAULT;             //base initial config from the
default, enabled in master mode
        //i2cInit.freq = I2C_FREQ_FAST_MAX;
        //i2cInit.clhr = i2cClockHLRAsymetric;

        //Both I2C pins configured for open-drain pull up operation
        GPIO_PinModeSet(gpioPortB, 0, gpioModeWiredAndPullUpFilter, 1);           //PB0 = SDA
        GPIO_PinModeSet(gpioPortB, 1, gpioModeWiredAndPullUpFilter, 1);          //PB1 = SCLK

        // Route I2C pins to GPIO
        GPIO->I2CROUTE[0].SDAROUTE = (GPIO->I2CROUTE[0].SDAROUTE & ~_GPIO_I2C_SDAROUTE_MASK)
                                                                | (gpioPortB <<
_GPIO_I2C_SDAROUTE_PORT_SHIFT
                                                                | (0 <<
_GPIO_I2C_SDAROUTE_PIN_SHIFT));
        GPIO->I2CROUTE[0].SCLROUTE = (GPIO->I2CROUTE[0].SCLROUTE & ~_GPIO_I2C_SCLROUTE_MASK)
                                                                | (gpioPortB <<
_GPIO_I2C_SCLROUTE_PORT_SHIFT
                                                                | (1 <<
_GPIO_I2C_SCLROUTE_PIN_SHIFT));
        GPIO->I2CROUTE[0].ROUTEEN = GPIO_I2C_ROUTEEN_SDAPEN | GPIO_I2C_ROUTEEN_SCLPEN;

        //Complete I2C initialization
        I2C_Init(I2C0, &i2cInit);                                        //initialize i2c peripheral
}

I2C_TransferReturn_TypeDef i2c_write_single(I2C_TypeDef *i2c, uint8_t slave_addr, uint8_t reg_addr,
uint8_t data)
{
        //Initialize I2C transfer
        I2C_TransferSeq_TypeDef i2cTransfer;
        i2cTransfer.addr = (slave_addr << 1);
        i2cTransfer.flags = I2C_FLAG_WRITE_WRITE;
        i2cTransfer.buf[0].data = &reg_addr;
        i2cTransfer.buf[0].len = 1;
        i2cTransfer.buf[1].data = &data;
        i2cTransfer.buf[1].len = 1;

        //Initialize I2C transfer return status
        I2C_TransferReturn_TypeDef status = I2C_TransferInit(i2c, &i2cTransfer);

        //Start I2C transfer
        while (status == i2cTransferInProgress) {                //loop until i2c transfer is finished
                status = I2C_Transfer(i2c);
        }

        return status;
}

I2C_TransferReturn_TypeDef i2c_write_burst(I2C_TypeDef *i2c, uint8_t slave_addr, uint8_t reg_addr,
uint8_t *data, uint16_t num_bytes)
{
        //Initialize I2C transfer
        I2C_TransferSeq_TypeDef i2cTransfer;
        i2cTransfer.addr = (slave_addr << 1);
        i2cTransfer.flags = I2C_FLAG_WRITE_WRITE;
        i2cTransfer.buf[0].data = &reg_addr;
        i2cTransfer.buf[0].len = 1;
        i2cTransfer.buf[1].data = data;
        i2cTransfer.buf[1].len = num_bytes;

        //Initialize I2C transfer return status
        I2C_TransferReturn_TypeDef status = I2C_TransferInit(i2c, &i2cTransfer);

        //Start I2C transfer
        while (status == i2cTransferInProgress) {                //loop until i2c transfer is finished
```

```
		status = I2C_Transfer(i2c);
	}

	return status;
}

I2C_TransferReturn_TypeDef i2c_read_single(I2C_TypeDef *i2c, uint8_t slave_addr, uint8_t reg_addr,
uint8_t *buffer)
{
	//Initialize I2C transfer
	I2C_TransferSeq_TypeDef i2cTransfer;
	i2cTransfer.addr = (slave_addr << 1);
	i2cTransfer.flags = I2C_FLAG_WRITE_READ;
	i2cTransfer.buf[0].data = &reg_addr;
	i2cTransfer.buf[0].len = 1;
	i2cTransfer.buf[1].data = buffer;
	i2cTransfer.buf[1].len = 1;

	//Initialize I2C transfer return status
	I2C_TransferReturn_TypeDef status = I2C_TransferInit(i2c, &i2cTransfer);

	//Start I2C transfer
	while (status == i2cTransferInProgress) {		//loop until i2c transfer is finished
		status = I2C_Transfer(i2c);
	}

	return status;
}

I2C_TransferReturn_TypeDef i2c_read_burst(I2C_TypeDef *i2c, uint8_t slave_addr, uint8_t reg_addr,
uint8_t* buffer, uint16_t num_bytes)
{
	//Initialize I2C transfer
	I2C_TransferSeq_TypeDef i2cTransfer;
	i2cTransfer.addr = (slave_addr << 1);
	i2cTransfer.flags = I2C_FLAG_WRITE_READ;
	i2cTransfer.buf[0].data = &reg_addr;
	i2cTransfer.buf[0].len = 1;
	i2cTransfer.buf[1].data = buffer;
	i2cTransfer.buf[1].len = num_bytes;

	//Initialize I2C transfer return status
	I2C_TransferReturn_TypeDef status = I2C_TransferInit(i2c, &i2cTransfer);

	//Start I2C transfer
	while (status == i2cTransferInProgress) {		//loop until i2c transfer is finished
		status = I2C_Transfer(i2c);
	}

	return status;
}
```

<u>si1132.h</u>

```
/*
 * si1132.h
 *
 *  Created on: Oct 13, 2021
 *  Last Modified: Jan 26, 2022
 *      Author: Judah Ben-Eliezer, Tennyson Cheng
 *
 *  This header file contains the declarations for the macros, types, and functions
 *  needed to initialize and interface with the Si1132 UV sensors.
 *
 */

#ifndef SI1132_H_
#define SI1132_H_

/* Modes */
typedef enum
{
        FORCED, AUTO
} Mode_t;

/* Clock limits */
#define MAX_FREQUENCY 34000000U  //max clock frequency (Hz)
#define MIN_FREQUENCY 95000U     //min clock frequency (Hz)

/* More timing macros */
#define STARTUP_TIME 25000U      //startup time (us)
#define CONVERSION_TIME 300U     //time needed for uv data to be ready (us)

#define DEFAULT_ADDR 0x60        //default slave i2c address

/* Bit masks for CHLIST */
#define EN_UV (1<<7)
#define EN_AUX (1<<6)
#define EN_ALS_IR (1<<5)
#define EN_ALS_VIS (1<<4)

/* ADC masks */
#define VIS_RANGE (1<<5)
#define IR_RANGE (1<<5)

/* Interrupt Masks */
#define INT_OE (1<<0)
#define ALS_IE (1<<0)

/* Si1132 Register Addresses */
#define PART_ID 0x00
#define REV_ID 0x01
#define SEQ_ID 0x02
#define INT_CFG 0x03
#define IRQ_ENABLE 0x04
#define HW_KEY 0x07
#define MEAS_RATE0 0x08
#define MEAS_RATE1 0x09
#define UCOEF0 0x13
#define UCOEF1 0x14
#define UCOEF2 0x15
#define UCOEF3 0x16
#define PARAM_WR 0x17
#define COMMAND 0x18
#define RESPONSE 0x20
#define IRQ_STATUS 0x21
#define ALS_VIS_DATA0 0x22
#define ALS_VIS_DATA1 0x23
```

```
#define ALS_IR_DATA0 0x24
#define ALS_IR_DATA1 0x25
#define AUX_DATA0 0x2C
#define UVINDEX0 0x2C
#define AUX_DATA1 0x2D
#define UVINDEX1 0x2D
#define PARAM_RD 0x2E
#define CHIP_STAT 0x30
#define ANA_IN_KEYHH 0x3B
#define ANA_IN_KEYH 0x3C
#define ANA_IN_KEYL 0x3D
#define ANA_IN_KEYLL 0x3E

/* Parameter Ram Addresses */
#define I2C_ADDR 0x00
#define CHLIST 0x01
#define ALS_ENCODING 0x06
#define ALS_IR_ADCMUX 0x0E
#define AUX_ADCMUX 0x0F
#define ALS_VIS_ADC_COUNTER 0x10
#define ALS_VIS_ADC_GAIN 0x11
#define ALS_VIS_ADC_MISC 0x12
#define ALS_IR_ADC_COUNTER 0x1D
#define ALS_IR_ADC_GAIN 0x1E
#define ALS_IR_ADC_MISC 0x1F

/* Command Register Codes */
#define PARAM_QUERY 0b10000000
#define PARAM_SET 0b10100000
#define NOP 0b00000
#define RESET 0b00000001
#define BUSADDR 0b00000010
#define GET_CAL 0b00010010
#define ALS_FORCE 0b00000110
#define ALS_PAUSE 0b00001010
#define ALS_AUTO 0b00001110

/* Response Errors */
#define NOERROR 0x00
#define INVALID_SETTINGS 0x80
#define ALS_VIS_ADC_OVERFLOW 0x8C
#define ALS_IR_ADC_OVERFLOW 0x8D
#define AUX_ADC_OVERFLOW 0x8E
#define I2C_TRANSFER_ERROR 0x8F

/***************************************************************************//**
 * @brief
 *   Initializes si1132 slave for UV detection.
 *
 * @param[in] power_port
 *       GPIO output port for slave VDD.
 *
 * @param[in] power_pin
 *   GPIO output pin for slave VDD.
 *
 * @param[in] i2c
 *   Pointer to I2C peripheral register block.
 *
 * @param[in] slave_addr.
 *   Address to assign to slave.
 ******************************************************************************/
bool si1132_init(GPIO_Port_TypeDef power_port, uint8_t power_pin, I2C_TypeDef *i2c, uint8_t slave_addr);

bool si1132_calibrate(I2C_TypeDef *i2c, uint8_t slave_addr);

/***************************************************************************//**
 * @brief
 *   Sends command to si1132 slave.
 *   Returns the RESPONSE.
 *
 * @param[in] i2c
 *   Pointer to I2C peripheral register block.
 *
 * @param[in] slave_addr.
 *   Address of slave.
 *
 * @param[in] command
 *   Command to send.
 ******************************************************************************/
uint8_t si1132_send_command(I2C_TypeDef *i2c, uint8_t slave_addr, uint8_t command);

/***************************************************************************//**
 * @brief
 *   Sends BUSADDR command to si1132 slave.
 *   Since the slave address is changed for the final RESPONSE read,
 *   this function is seperated from the normal si1132_send_command.
 *   Returns the RESPONSE.
 *
 * @param[in] i2c
 *   Pointer to I2C peripheral register block.
 *
 * @param[in] slave_addr.
 *   Address of slave.
 *
 * @param[in] slave_addr_new
 *   New address of the slave.
 ******************************************************************************/
uint8_t si1132_send_command_BUSADDR(I2C_TypeDef *i2c, uint8_t slave_addr, uint8_t slave_addr_new);

/***************************************************************************//**
 * @brief
 *   Writes 1 byte to ram of si1132 slave.
 *
 * @param[in] i2c
 *   Pointer to I2C peripheral register block.
 *
 * @param[in] slave_addr.
 *   Address of slave.
 *
 * @param[in] addr
 *   Ram address to write to.
 *
 * @param[in] data
 *   Byte to write.
 ******************************************************************************/
bool si1132_write_ram(I2C_TypeDef *i2c, uint8_t slave_addr, uint8_t addr, uint8_t data);

/***************************************************************************//**
```

```
 * @brief
 *   Reads 1 byte from ram of si1132 slave.
 *
 * @param[in] i2c
 *   Pointer to I2C peripheral register block.
 *
 * @param[in] slave_addr.
 *   Address of slave.
 *
 * @param[in] addr
 *   Ram address to read from.
 *
 * @param[in] buffer
 *   Buffer to read data into.
 ******************************************************************************/
bool si1132_read_ram(I2C_TypeDef *i2c, uint8_t slave_addr, uint8_t addr, uint8_t buffer);

/***************************************************************************//**
 * @brief
 *   Sends ALS_FORCE command to start UV measurement in forced conversion mode
 *
 * @param[in] i2c
 *   Pointer to I2C peripheral register block.
 *
 * @param[in] slave_addr.
 *   Address of slave.
 ******************************************************************************/
uint8_t si1132_UV_start_measurement(I2C_TypeDef *i2c, uint8_t slave_addr);

/***************************************************************************//**
 * @brief
 *   Reads word from UVINDEX1|UVINDEX0 and stores into buffer.
 *
 * @param[in] i2c
 *   Pointer to I2C peripheral register block.
 *
 * @param[in] slave_addr.
 *   Address of slave.
 *
 * @param[in] buffer.
 *   Buffer to read word into.
 ******************************************************************************/
uint8_t si1132_UV_read_measurement(I2C_TypeDef *i2c, uint8_t slave_addr, uint16_t *buffer);

/***************************************************************************//**
 * CODE BELOW HAS BEEN PROVIDED BY SILICON LABS
 * "si114x_function.c"
 *
 * Slightly modified for calibration of Si1132 chips only
 ******************************************************************************/
#define FLT_TO_FX20(x)       ((int32_t)((x*1048576)+.5))
#define FX20_ONE             FLT_TO_FX20( 1.000000)
#define FX20_BAD_VALUE       0xffffffff

#define SIRPD_ADCHI_IRLED    (collect(buffer, 0x23, 0x22,  0))
#define SIRPD_ADCLO_IRLED    (collect(buffer, 0x22, 0x25,  1))
#define SIRPD_ADCLO_WHLED    (collect(buffer, 0x24, 0x26,  0))
#define VISPD_ADCHI_WHLED    (collect(buffer, 0x26, 0x27,  1))
#define VISPD_ADCLO_WHLED    (collect(buffer, 0x28, 0x29,  0))
```

```c
#define LIRPD_ADCHI_IRLED    (collect(buffer, 0x29, 0x2a,  1))
#define LED_DRV65            (collect(buffer, 0x2b, 0x2c,  0))

#define ALIGN_LEFT   1
#define ALIGN_RIGHT -1

struct operand_t
{
  uint32_t op1;
  uint32_t op2;
};
struct cal_ref_t
{
        uint32_t sirpd_adchi_irled;
        uint32_t sirpd_adclo_irled;
        uint32_t sirpd_adclo_whled;
        uint32_t vispd_adchi_whled;
        uint32_t vispd_adclo_whled;
        uint32_t lirpd_adchi_irled;
        uint32_t ledi_65ma;
        uint8_t  ucoef[4];
};

uint32_t decode(uint32_t input);
uint32_t collect(uint8_t* buffer, uint8_t msb_addr, uint8_t lsb_addr, uint8_t alignment);
void shift_left(uint32_t* value_p, int8_t shift);
int8_t align( uint32_t* value_p, int8_t direction );
void fx20_round(uint32_t *value_p);
uint32_t fx20_divide( struct operand_t* operand_p );
uint32_t fx20_multiply( struct operand_t* operand_p );
uint32_t vispd_correction(uint8_t *buffer);
uint32_t irpd_correction(uint8_t *buffer);
#endif /* SI1132_H_ */
```

<u>si1132.c</u>

```c
/*
 * si1132.c
 *
 *  Created on: Oct 13, 2021
 *  Last Modified: Jan 26, 2022
 *      Author: Judah Ben-Eliezer, Tennyson Cheng
 *
 *  All code related to the Si1132 i2c interface.
 *      Intended for UV data measurement uses.
 *
 */

#include "sl_udelay.h"
#include "em_gpio.h"
#include "em_i2c.h"
#include "i2c.h"
#include "si1132.h"

struct cal_ref_t calref =
{
        FLT_TO_FX20( 4.021290),  // sirpd_adchi_irled
        FLT_TO_FX20(57.528500),  // sirpd_adclo_irled
        FLT_TO_FX20( 2.690010),  // sirpd_adclo_whled
        FLT_TO_FX20( 0.042903),  // vispd_adchi_whled
        FLT_TO_FX20( 0.633435),  // vispd_adclo_whled
        FLT_TO_FX20(23.902900),  // lirpd_adchi_irled
        FLT_TO_FX20(56.889300),  // ledi_65ma
        {0x7B, 0x6B, 0x01, 0x00} // default ucoef
};

bool si1132_init(GPIO_Port_TypeDef power_port, uint8_t power_pin, I2C_TypeDef *i2c, uint8_t slave_addr)
{
        GPIO_PinOutSet(power_port, power_pin);           //power off i2c

        sl_udelay_wait(STARTUP_TIME);

        GPIO_PinOutSet(power_port, power_pin);           //power on i2c

        sl_udelay_wait(STARTUP_TIME);                    //allow time for startup
                                                         //Note: recommended by the data sheet before
any I2C activity

        /* write 0x17 to HW_KEY for proper operation */
        if (i2c_write_single(i2c, DEFAULT_ADDR, HW_KEY, 0x17) != i2cTransferDone) return false;

        /* set device I2C address */
        if (!si1132_write_ram(i2c, DEFAULT_ADDR, I2C_ADDR, slave_addr)) return false;
        if ((si1132_send_command_BUSADDR(i2c, DEFAULT_ADDR, slave_addr) & 0xF0) != NOERROR ) return
false;

        /* disable interrupts */
        if (i2c_write_single(i2c, slave_addr, INT_CFG, 0x00) != i2cTransferDone) return false;
        if (i2c_write_single(i2c, slave_addr, IRQ_ENABLE, 0x00) != i2cTransferDone) return false;

        /* configure for forced measurement mode */
        if (i2c_write_single(i2c, slave_addr, MEAS_RATE0, 0x00) != i2cTransferDone) return false;
        if (i2c_write_single(i2c, slave_addr, MEAS_RATE1, 0x00) != i2cTransferDone) return false;

        /* enable vis_range and ir_range */
        if (!si1132_write_ram(i2c, slave_addr, ALS_VIS_ADC_MISC, VIS_RANGE) != i2cTransferDone) return
false;
        if (!si1132_write_ram(i2c, slave_addr, ALS_IR_ADC_MISC, IR_RANGE) != i2cTransferDone) return
false;

        /* calibrate the sensor */
        if (!si1132_calibrate(i2c, slave_addr)) return false;

        /* enable UV detection only */
        if (!si1132_write_ram(i2c, slave_addr, CHLIST, EN_UV)) return false;

        return true;
}

bool si1132_calibrate(I2C_TypeDef *i2c, uint8_t slave_addr) {
        //Send command to fetch calibration data
        if ((si1132_send_command(i2c, slave_addr, GET_CAL) & 0xF0) != NOERROR ) return false;

        //Calibration data is located in register addresses 0x22-0x2D
        uint8_t cal[12];
        if (i2c_read_burst(i2c, slave_addr, ALS_VIS_DATA0, cal, 12) != i2cTransferDone) return false;

        //Processing of calibration data for UCOEF values
        uint32_t         long_temp;
        struct operand_t op;
        uint8_t          out_ucoef[4];

        op.op1 = calref.ucoef[0] + ((calref.ucoef[1])<<8);
        op.op2 = vispd_correction(cal);
        long_temp   = fx20_multiply( &op );
        out_ucoef[0] = (long_temp & 0x00ff);
        out_ucoef[1] = (long_temp & 0xff00)>>8;

        op.op1 = calref.ucoef[2] + (calref.ucoef[3]<<8);
        op.op2 = irpd_correction(cal);
        long_temp   = fx20_multiply( &op );
        out_ucoef[2] = (long_temp & 0x00ff);
        out_ucoef[3] = (long_temp & 0xff00)>>8;

        //Write back calibrated UCOEF values back to Si1132
        if (i2c_write_burst(i2c, slave_addr, UCOEF0, out_ucoef, 4) != i2cTransferDone) return false;
        return true;
}

uint8_t si1132_send_command(I2C_TypeDef *i2c, uint8_t slave_addr, uint8_t command)
{
        uint8_t response = 0;

        /* clear response register */
        if (i2c_write_single(i2c, slave_addr, COMMAND, NOP) != i2cTransferDone) return response;
        if (i2c_read_single(i2c, slave_addr, RESPONSE, &response) != i2cTransferDone) return response;

        /* send command to command register */
        if (i2c_write_single(i2c, slave_addr, COMMAND, command) != i2cTransferDone) return response;
        if (i2c_read_single(i2c, slave_addr, RESPONSE, &response) != i2cTransferDone) return response;

        return response;
}

uint8_t si1132_send_command_BUSADDR(I2C_TypeDef *i2c, uint8_t slave_addr, uint8_t slave_addr_new)
{
        uint8_t response = 0;

        /* clear response register */
        if (i2c_write_single(i2c, slave_addr, COMMAND, NOP) != i2cTransferDone) return response;
        if (i2c_read_single(i2c, slave_addr, RESPONSE, &response) != i2cTransferDone) return response;

        /* send BUSADDR to command register */
        if (i2c_write_single(i2c, slave_addr, COMMAND, BUSADDR) != i2cTransferDone) return response;
        if (i2c_read_single(i2c, slave_addr_new, RESPONSE, &response) != i2cTransferDone) return
response;

        return response;
}

bool si1132_write_ram(I2C_TypeDef *i2c, uint8_t slave_addr, uint8_t addr, uint8_t data)
{
        /* Write data to PARAM_WR */
        if ((i2c_write_single(i2c, slave_addr, PARAM_WR, data)) != i2cTransferDone) return false;

        uint8_t ram_addr = (addr & 0x1F) | PARAM_SET;
        /* Write PARAM_WR to ram */
        if ((si1132_send_command(i2c, slave_addr, ram_addr) & 0xF0) != NOERROR) return false;
        return true;
}

bool si1132_read_ram(I2C_TypeDef *i2c, uint8_t slave_addr, uint8_t addr, uint8_t buffer)
{
        uint8_t ram_addr = (addr & 0x1F) | PARAM_QUERY;
        if ((si1132_send_command(i2c, slave_addr, ram_addr) & 0xF0) != NOERROR) return false;
        return (i2c_read_single(i2c, slave_addr, PARAM_RD, &buffer) == i2cTransferDone);
}

uint8_t si1132_UV_start_measurement(I2C_TypeDef *i2c, uint8_t slave_addr)
{
        uint8_t status;

        /* send conversion command to I2C */
        if ((status = (si1132_send_command(i2c, slave_addr, ALS_FORCE)) & 0xF0) != NOERROR) return
status;

        /* allow time for conversion */
        sl_udelay_wait(CONVERSION_TIME);

        return status;
}

uint8_t si1132_UV_read_measurement(I2C_TypeDef *i2c, uint8_t slave_addr, uint16_t *buffer)
{
        uint8_t data[2], status;

        /* read word into buffer */
        if (i2c_read_burst(i2c, slave_addr, UVINDEX0, data, 2) != i2cTransferDone) return
I2C_TRANSFER_ERROR;

        *buffer = (data[1] << 8) | data[0];

        return status;
}

/***************************************************************************//**
 * CODE BELOW HAS BEEN PROVIDED BY SILICON LABS
 * "si114x_function.c"
 *
 * Slightly modified for calibration of Si1132 chips only
 ******************************************************************************/
uint32_t decode(uint32_t input)
{
        int32_t  exponent, exponent_bias9;
        uint32_t mantissa;

        if(input==0) return 0.0;

        exponent_bias9 = (input & 0x0f00) >> 8;
        exponent       = exponent_bias9 - 9;

        mantissa       = input & 0x00ff; // fraction
        mantissa       |=        0x0100; // add in integer

        // representation in 12 bit integer, 20 bit fraction
        mantissa       = mantissa << (12+exponent);
        return mantissa;
}
/**************************************************************************/
uint32_t collect(uint8_t* buffer,
                                                  uint8_t msb_addr,
                                                  uint8_t lsb_addr,
                                                  uint8_t alignment)
{
        uint16_t value;
        uint8_t  msb_ind = msb_addr - 0x22;
        uint8_t  lsb_ind = lsb_addr - 0x22;

        if(alignment == 0)
        {
                value =  buffer[msb_ind]<<4;
                value += buffer[lsb_ind]>>4;
        }
        else
        {
                value =  buffer[msb_ind]<<8;
                value += buffer[lsb_ind];
                value &= 0x0fff;
        }

        if(     ( value == 0x0fff )
                || ( value == 0x0000 ) ) return FX20_BAD_VALUE;
        else return decode( value );
}
/**************************************************************************/
void shift_left(uint32_t* value_p, int8_t shift)
{
        if(shift > 0)
                *value_p = *value_p<<shift ;
        else
                *value_p = *value_p>>(-shift) ;
}
/**************************************************************************/
int8_t align( uint32_t* value_p, int8_t direction )
{
        int8_t   local_shift, shift ;
        uint32_t mask;

        // Check invalid value_p and *value_p, return without shifting if bad.
//      if( value_p  == NULL )  return 0;
        if( *value_p == 0 )     return 0;

        // Make sure direction is valid
        switch( direction )
        {
                case ALIGN_LEFT:
                        local_shift =  1 ;
                        mask  = 0x80000000L;
                        break;

                case ALIGN_RIGHT:
                        local_shift = -1 ;
                        mask  = 0x00000001L;
                        break;
                default:
                        // Invalid direction, return without shifting
                        return 0;
        }

        shift = 0;
        while(1)
        {
                if(*value_p & mask ) break;
                shift++;
                shift_left( value_p, local_shift );
        }
        return shift;
}
/***************************************************************************/
void fx20_round(uint32_t *value_p)
{
        int8_t  shift;

        uint32_t mask1  = 0xffff8000;
        uint32_t mask2  = 0xffff0000;
        uint32_t lsb    = 0x00008000;

        shift = align( value_p, ALIGN_LEFT );
        if( ( (*value_p)&mask1 ) == mask1 )
        {
                *value_p = 0x80000000;
                shift -= 1;
        }
        else
        {
                *value_p += lsb;
                *value_p &= mask2;
        }

        shift_left( value_p, -shift );
}
/***************************************************************************/
uint32_t fx20_divide( struct operand_t* operand_p )
{
        int8_t    numerator_sh=0, denominator_sh=0;
        uint32_t  result;
        uint32_t* numerator_p;
        uint32_t* denominator_p;

//      if( operand_p == NULL ) return FX20_BAD_VALUE;

        numerator_p   = &operand_p->op1;
        denominator_p = &operand_p->op2;

        if(   (*numerator_p   == FX20_BAD_VALUE)
           || (*denominator_p == FX20_BAD_VALUE)
           || (*denominator_p == 0             ) ) return FX20_BAD_VALUE;

        fx20_round  ( numerator_p   );
        fx20_round  ( denominator_p );
        numerator_sh   = align ( numerator_p,   ALIGN_LEFT  );
        denominator_sh = align ( denominator_p, ALIGN_RIGHT );
        result = *numerator_p / ( (uint16_t)(*denominator_p) );
        shift_left( &result , 20-numerator_sh-denominator_sh );

        return result;
}
/****************************************************************************/
uint32_t fx20_multiply( struct operand_t* operand_p )
{
        uint32_t  result;
        int8_t    val1_sh, val2_sh;
        uint32_t* val1_p;
        uint32_t* val2_p;

//      if( operand_p == NULL ) return FX20_BAD_VALUE;

        val1_p = &(operand_p->op1);
        val2_p = &(operand_p->op2);

        fx20_round( val1_p );
        fx20_round( val2_p );

        val1_sh = align( val1_p, ALIGN_RIGHT );
        val2_sh = align( val2_p, ALIGN_RIGHT );


        result = (uint32_t)( ( (uint32_t)(*val1_p) ) * ( (uint32_t)(*val2_p) ) );
        shift_left( &result, -20+val1_sh+val2_sh );

        return result;
}
/****************************************************************************/
uint32_t vispd_correction(uint8_t *buffer)
{
        struct operand_t op;
        uint32_t         result;

        op.op1 = calref.vispd_adclo_whled;
        op.op2 = VISPD_ADCLO_WHLED;
        result = fx20_divide( &op );

        if( result == FX20_BAD_VALUE ) result = FX20_ONE;

        return result;
}
/****************************************************************************/
uint32_t irpd_correction(uint8_t *buffer)
{
        struct operand_t op;
        uint32_t         result;

        // op.op1 = SIRPD_ADCLO_IRLED_REF; op.op2 = SIRPD_ADCLO_IRLED;
        op.op1 = calref.sirpd_adclo_irled;
        op.op2 = SIRPD_ADCLO_IRLED;
        result = fx20_divide( &op );

        if( result == FX20_BAD_VALUE ) result = FX20_ONE;

        return result;
```

```
}
```

gpio.h

```
/*
 * gpio.h
 *
 *  Created on: Feb 14, 2022
 *      Author: Tennyson Cheng
 */

#ifndef GPIO_H_
#define GPIO_H_

#include "em_gpio.h"

#define GPIO_BUTTON_PORT gpioPortB
#define GPIO_BUTTON_PIN 2
#define GPIO_LED_PORT gpioPortC
#define GPIO_LED_PIN 0

typedef enum BUTTON_STATE {
        PRESSED = 0,
        RELEASED = 1,
        HELD,                   //physical state is the same as PRESSED, but logically different
        DEBOUNCED               //physical state is the same as PRESSED, but logically different
}BUTTON_STATE;

void gpio_init();
void gpio_led_off();
void gpio_led_on();

#endif /* GPIO_H_ */
```

gpio.c

```
/*
 * gpio.c
 *
 *  Created on: Feb 14, 2022
 *      Author: Tennyson Cheng
 */

#include "sl_bluetooth.h"
#include "em_gpio.h"
#include "int_enum.h"
#include "timer.h"
#include "gpio.h"

static uint8_t button_state;

void gpio_init() {
        button_state = GPIO_PinInGet(GPIO_BUTTON_PORT,   //store initial button state
                                                 GPIO_BUTTON_PIN);
        GPIO_ExtIntConfig(GPIO_BUTTON_PORT,              //falling and rising edge interrupts enabled
                          GPIO_BUTTON_PIN,
                          GPIO_BUTTON_PIN,
                          true,
                          true,
                          true);
        NVIC_EnableIRQ(GPIO_EVEN_IRQn);                  //enable even pin interrupts
}

inline void gpio_led_off() {
        GPIO_PinOutClear(GPIO_LED_PORT, GPIO_LED_PIN);
}

inline void gpio_led_on() {
        GPIO_PinOutSet(GPIO_LED_PORT, GPIO_LED_PIN);
}

//NOTE: This IRQ routine assumes that the button state had switched.
//If button was pressed, then in order to enter this IRQ, the button had to have been released.
//If button was released, then in order to enter this IRQ, the button had to have been pressed.
//The implemented debounce state logic will account for bounces.
void GPIO_EVEN_IRQHandler(void)
{
        uint32_t int_pins = GPIO_IntGet();               //clear all pin interrupt flags
        GPIO_IntClear(int_pins);

        if (int_pins & (1 << GPIO_BUTTON_PIN)) {         //check if pin interrupt was from button
                if (button_state == RELEASED) {          //if button was not pressed, debounce first
                        NVIC_DisableIRQ(GPIO_ODD_IRQn);  //disable odd pin interrupts
                        button_state = PRESSED;          //update button state to PRESSED
                        timer_button_debounce_start(&button_state);//button debounce using sleeptimer
                }
                else if (button_state == DEBOUNCED){     //if button had been debounced using timer
                        //NOTE: This conditional is only entered
                        //if the button was not held long enough
                        //to be considered HELD
                        timer_button_hold_stop();        //stop the hold detect timer
                        button_state = RELEASED;         //update button state to RELEASED
                        sl_bt_external_signal(INT_BUTTON_PRESS);         //signal button press
                }
                else {                                   //if button was PRESSED or HELD
                        button_state = RELEASED;         //update button state to RELEASED
                }
        }
}
```

timer.h

```
/*
 * timer.h
 *
 *  Created on: Feb 14, 2022
 *      Author: Tennyson Cheng
 */

#ifndef TIMER_H_
#define TIMER_H_

#include "sl_sleeptimer.h"

#define TIMER_SENSOR_PRIORITY 0
#define TIMER_BUTTON_PRIORITY 1
#define TIMER_LED_PRIORITY 2
```

```c
void timer_init_();
sl_status_t timer_sensor_start();
sl_status_t timer_sensor_stop();
sl_status_t timer_button_debounce_start(uint8_t* button);
sl_status_t timer_button_hold_stop();
sl_status_t timer_led_advertise_start();
sl_status_t timer_led_advertise_stop();
void timer_sensor_callback(sl_sleeptimer_timer_handle_t *handle, void *data);
void timer_button_debounce_callback(sl_sleeptimer_timer_handle_t *handle, void *data);
void timer_button_hold_callback(sl_sleeptimer_timer_handle_t *handle, void *data);
void timer_sensor_callback(sl_sleeptimer_timer_handle_t *handle, void *data);
void timer_led_on_callback(sl_sleeptimer_timer_handle_t *handle, void *data);
void timer_led_off_callback(sl_sleeptimer_timer_handle_t *handle, void *data);

#endif /* TIMER_H_ */
```

<u>timer.c</u>

```c
/*
 * timer.c
 *
 *  Created on: Feb 14, 2022
 *      Author: Tennyson Cheng
 */

#include "sl_bluetooth.h"
#include "sl_sleeptimer.h"
#include "int_enum.h"
#include "gpio.h"
#include "timer.h"

static uint32_t timer_freq = 32768;
static sl_sleeptimer_timer_handle_t TIMER_SENSOR;
static sl_sleeptimer_timer_handle_t TIMER_BUTTON;
static sl_sleeptimer_timer_handle_t TIMER_LED;
static uint8_t* button_state;

void timer_init_() {
        //STORE SLEEPTIMER CLOCK FREQUENCY FOR SOFT TIMERS
        timer_freq = sl_sleeptimer_get_timer_frequency();
}

sl_status_t timer_sensor_start() {
        return sl_sleeptimer_start_periodic_timer(&TIMER_SENSOR,
                                                  timer_freq,            //1 second timer period
                                                  timer_sensor_callback,
                                                  NULL,
                                                  TIMER_SENSOR_PRIORITY,
                                         SL_SLEEPTIMER_NO_HIGH_PRECISION_HF_CLOCKS_REQUIRED_FLAG);
}

sl_status_t timer_sensor_stop() {
        return sl_sleeptimer_stop_timer(&TIMER_SENSOR);
}

sl_status_t timer_button_debounce_start(uint8_t* button) {
        button_state = button;                          //store pointer to button state
        return sl_sleeptimer_start_timer(&TIMER_BUTTON,
                                         timer_freq >> 6,        //15.6 ms timer period
                                         timer_button_debounce_callback,
                                         NULL,
                                         TIMER_BUTTON_PRIORITY,
                                        SL_SLEEPTIMER_NO_HIGH_PRECISION_HF_CLOCKS_REQUIRED_FLAG);
}

sl_status_t timer_button_hold_stop() {
        return sl_sleeptimer_stop_timer(&TIMER_BUTTON);
}

sl_status_t timer_led_advertise_start() {
        sl_bt_external_signal(INT_LED_ON);
        return sl_sleeptimer_start_timer(&TIMER_LED,
                                        timer_freq >> 4,         //(1/16) second timer period
                                        timer_led_on_callback,
                                        NULL,
                                        TIMER_LED_PRIORITY,
                                        SL_SLEEPTIMER_NO_HIGH_PRECISION_HF_CLOCKS_REQUIRED_FLAG);
}

sl_status_t timer_led_advertise_stop() {
        sl_bt_external_signal(INT_LED_OFF);
        return sl_sleeptimer_stop_timer(&TIMER_LED);
}

void timer_sensor_callback(sl_sleeptimer_timer_handle_t *handle, void *data) {
        (void)handle;
        (void)data;
        sl_bt_external_signal(INT_TIMER_SENSOR);
}

void timer_button_debounce_callback(sl_sleeptimer_timer_handle_t *handle, void *data) {
        (void)handle;
        (void)data;
        *button_state = (uint8_t)GPIO_PinInGet(GPIO_BUTTON_PORT, //get the current state of the button
                                               GPIO_BUTTON_PIN);  //can only be either (PRESSED or
RELEASED)
        if (*button_state == PRESSED) {          //if the button is still being pressed
                *button_state = DEBOUNCED;               //update button state to DEBOUNCED
                sl_sleeptimer_start_timer(&TIMER_BUTTON, //start timer to detect for button hold
                                         timer_freq << 1,        //2 second timer period
                                         timer_button_hold_callback,
                                         NULL,
                                         TIMER_BUTTON_PRIORITY,
                                         SL_SLEEPTIMER_NO_HIGH_PRECISION_HF_CLOCKS_REQUIRED_FLAG);
        }
        uint32_t int_pins = GPIO_IntGet();       //clear all interrupt flags
        GPIO_IntClear(int_pins);
        NVIC_EnableIRQ(GPIO_ODD_IRQn);           //re-enable odd pin interrupts
}

void timer_button_hold_callback(sl_sleeptimer_timer_handle_t *handle, void *data) {
        (void)handle;
        (void)data;
        if (*button_state == DEBOUNCED) {        //if the button had been debounced
                //Note: It shouldn't be possible to enter this callback
                //with a button_state other than DEBOUNCED
                *button_state = HELD;            //update the button state to HELD
                sl_bt_external_signal(INT_BUTTON_HOLD);          //signal button hold
        }
}

void timer_led_on_callback(sl_sleeptimer_timer_handle_t *handle, void *data) {
        (void)handle;
        (void)data;
        sl_bt_external_signal(INT_LED_OFF);
        sl_sleeptimer_start_timer(&TIMER_LED,
                                  (timer_freq >> 4) +     //(1/16 +
                                  (timer_freq >> 3),      //(1/8) second timer period
                                  timer_led_off_callback,
                                  NULL,
                                  TIMER_LED_PRIORITY,
                                  SL_SLEEPTIMER_NO_HIGH_PRECISION_HF_CLOCKS_REQUIRED_FLAG);
}

void timer_led_off_callback(sl_sleeptimer_timer_handle_t *handle, void *data) {
        (void)handle;
        (void)data;
        sl_bt_external_signal(INT_LED_ON);
        sl_sleeptimer_start_timer(&TIMER_LED,
                                  timer_freq >> 4,        //(1/16) second timer period
                                  timer_led_on_callback,
                                  NULL,
                                  TIMER_LED_PRIORITY,
                                  SL_SLEEPTIMER_NO_HIGH_PRECISION_HF_CLOCKS_REQUIRED_FLAG);
}
```