Development of a System that Senses an Environment and Controls a Rotary Fan

Anand Seetharaman

Renesas Electronics | Farmington Hills, Michigan

Field Applications Engineering Intern (Summer 2024)

Table of Contents

Abstract

This document outlines my summer project at Renesas Electronics, where I worked as a Field Applications Engineering intern. The project involved designing and developing a solution for sensing an environment and controlling a rotary fan to enhance air circulation. The specific use case and algorithms were to be defined and developed by me.

This document is not an exhaustive technical report but serves as a resource for engineers to understand the hardware and software components I used. It also aims to assist customers in creating robust solutions for their applications. For a detailed view of the project code and configuration files, please refer to the GitHub link provided in the references. The requirements of this project were to:

- Develop GreenPAK device to sequence buck regulators
- Develop HVPAK to drive HVAC blower (not full size)
- Write code to read sensor data and report over wireless
- Keep track of “alive” time using RTC and VBATT
- Wrap all hardware into a project box

This project was inspired by one of Renesas' “winning combos,” which are integrated solutions offered to customers rather than selling individual components. For example, if a customer wanted to build a security system, Renesas would offer a comprehensive solution including microcontrollers, regulators, and GreenPAKs, rather than selling these components separately. My task was to recreate and enhance an existing winning combo, improving it wherever possible given the time constraints.

In this document, I will cover the development process, the challenges encountered, and suggestions for future improvements.

The components used in this project include:

- RRH62000 All-in-One Environmental Sensor

- DA14531 Bluetooth Low Energy PMOD
- RAA211820 Buck Regulator (4.5 to 75V input, 5V output)
- RAA808013 Buck Regulator (2.7 to 5.5V input, 3.3V output)
- SLG47115V-DIP HVPAK
- SLG46537-DIP GreenPAK with ASM
- SLG4DVKADV GreenPAK Advanced Development Kit
- SLG4SA-DIP GreenPAK DIP Adapter
- EK-RA6M4 Microcontroller
- BOJACK Solderless Breadboard Kit
- Wathai Brushless DC PWM Fan
- BOJACK Electronic Fun Kit

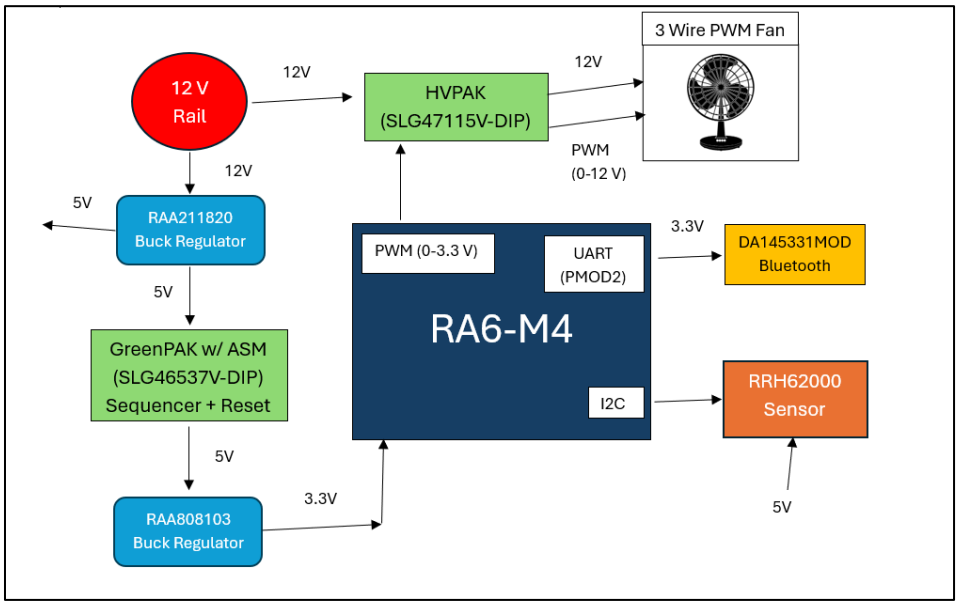

Figure 1: block diagram for my entire system

Steps for Design and Development

In this section, I will outline the key steps of my development process:

1. Establishing Communication with RRH62000
2. Sending and Receiving Data Through Wireless Communication
3. Fan Control Using HVPAK and Microcontroller
4. Sequence Buck Regulators and Provide a Hard Reset with GreenPAK
5. Implementing a Real-Time Clock
6. Integrating the Entire System and Enclosing Hardware

Establishing Communication with RRH62000

Establishing communication with the RRH62000 was one of the more challenging aspects of the project. Limited training materials were available, requiring me to determine the appropriate communication protocol (UART or I2C), select the right FSP API, configure the project, and handle errors and data type conversions.

Initial Steps:

- **Reviewing the Datasheet:** I began by analyzing the sensor's datasheet to understand its slave address, address mode, data transmission methods, and write commands.
- **Pin Configuration:** I looked at the RA6M4 schematic and datasheet to identify I2C-compatible pins, which guided my stack configuration.

Protocol Choice: I opted for I2C communication based on its recommendation in the original winning combo. This choice proved effective given the short-wired connection between the sensor and microcontroller. I2C also offers future flexibility for adding more peripherals to the serial bus.

Coding the Sensor Thread:

- **I2C Setup:** I used the RM_COMMS_I2C API to open the I2C bus, create a blocking semaphore, and establish a recursive mutex at the start of my thread.
- **API Calls:** I implemented the API functions in the following sequence:

```
err = RM_COMMS_I2C_Write(&g_comms_i2c_device0_ctrl, &TX_BUFFER2[0], 1);
```

The RA Flexible Software Package Documentation provided clear guidelines for API formats and parameters. The variable err was used to verify successful execution, with FSP_SUCCESS (or 0) indicating success and other codes indicating errors. I incorporated error handling and a half-second delay between function calls to ensure proper execution.

Callback Function: A callback function was included to handle events from the I2C API. It monitored callback events like RM_COMMS_EVENT_OPERATION_COMPLETE, RM_COMMS_EVENT_TX_OPERATION_COMPLETE, RM_COMMS_EVENT_RX_OPERATION_COMPLETE, and RM_COMMS_EVENT_ERROR, helping to pinpoint issues effectively.

Data Storage and Conversion: To handle the 37-byte sensor message, I:

- Created an array of 37 unsigned 8-bit integers.
- Set the address pointer using the write API and read data into the array.
- Converted sensor data into appropriate formats, such as:

```
uint8_t byte_temperature[2] = {sensor_data[25], sensor_data[24] };
```

For data transfer, I used memcpy to copy values into signed 16-bit integers. The data was then displayed on the terminal or sent wirelessly. I used a while loop to continuously update sensor data

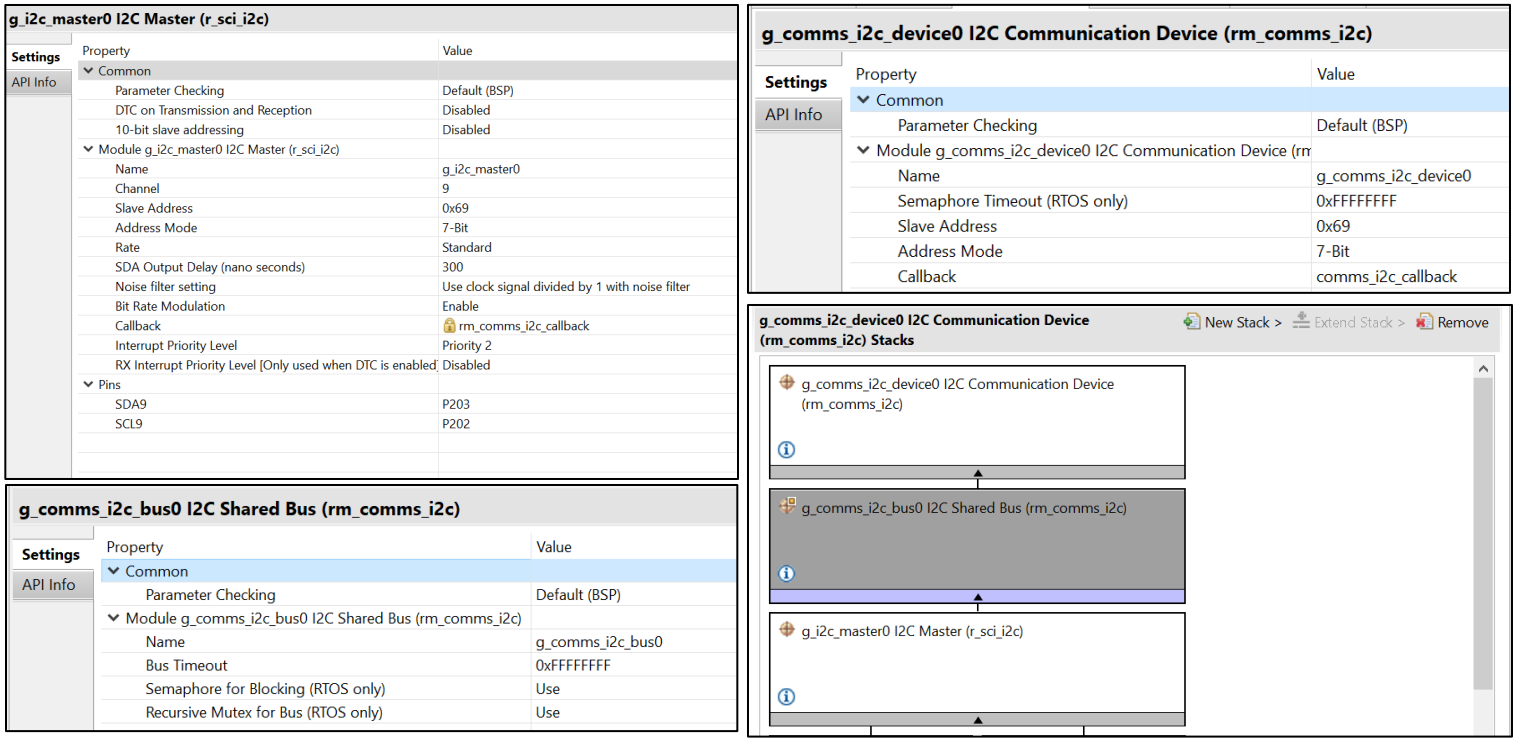

Figure 2: configuration and properties for my e2studio project

Sending and Receiving Data Through Wireless Communication

For wireless communication, I selected Bluetooth Low Energy (BLE) due to its efficiency and compatibility with mobile applications. The initial step involved re-flashing the BLE PMOD (DA14531). I followed the instructions detailed in the Renesas DA14531 Firmware Upgrade document and used the Renesas SmartBond Flash Programmer software for this task

Initial BLE Configuration: Initially, I designed a custom BLE profile using the Bluetooth LE lab document titled Lab A_ Introducing the New DA14531 FSP Bluetooth LE Framework Update. In this framework, I created a dedicated server for reading sensor data, along with individual characteristics within the server for specific data points.

Transition to Quick-Connect Service: As the project evolved, I decided I needed a graphical user interface (GUI) and mobile application. Therefore, I transitioned to using Renesas's Quick-Connect service. This service utilizes a protocol that includes:

- **Request Characteristic:** For sending data to the device.
- **Response Characteristic:** For receiving data from the device.

Details on this protocol can be found in the Renesas Quick-Connect Mobile Sandbox Deep Dive presentation. The Quick-Connect framework simplifies the transmission of requests and reception of responses, which I implemented by adapting code from the QC_Sandbox_Example_RA6M4_FreeRTOS example project.

Creating the Framework:

- **Code Generation:** I generated code for my custom profile and created prototype declarations for user functions.
- **Handler Integration:** I added necessary read/write handlers to the qc_svc_request_handlers_t structure.
- **Variable Declaration and Function Definitions:** I declared variables and wrote function definitions. The format was intuitive due to prewritten code templates, ensuring consistency and ease of integration.

I included functionality to send data for PWM duty cycle (0-100%), set temperature (0-100°C), and set humidity (0-100%). Details on how this data is utilized will be discussed in the next section.

Data Handling with Queues: For intertask communication, I used queues, which provide a thread-safe FIFO (First In, First Out) buffer. The primary functions utilized from the queue API were xQueueSend() and xQueueReceive():

- **Sending Data:** I used xQueueSend() to queue up sensor data and followed this with xEventGroupSetBits() to unblock tasks waiting for specific bits.

- **Receiving Data:** To retrieve data from the queue, I used:
    - `if (pdTRUE == xQueueReceive(g_sensor_queue, &sens_data, 0))`

  Here, pdTRUE confirms that data was successfully received. The sens_data array, which stores signed 16-bit integer values, is then processed and assigned to variables.

GUI Configuration for QC Sandbox Mobile App: To create widgets for the QC Sandbox mobile app, I assigned unique IDs to each attribute and followed a precise format in the gui_cfg.json file within the source folder. Accuracy in this file is crucial, as any formatting error can cause parsing issues when connecting to a mobile device.

- **File Conversion:** I used IDLE Shell 3.12.4 to run the json2array.py script with gui_cfg.json and gui_cfg.h as arguments. This process was iterative, allowing me to refine the interface until it met my requirements.

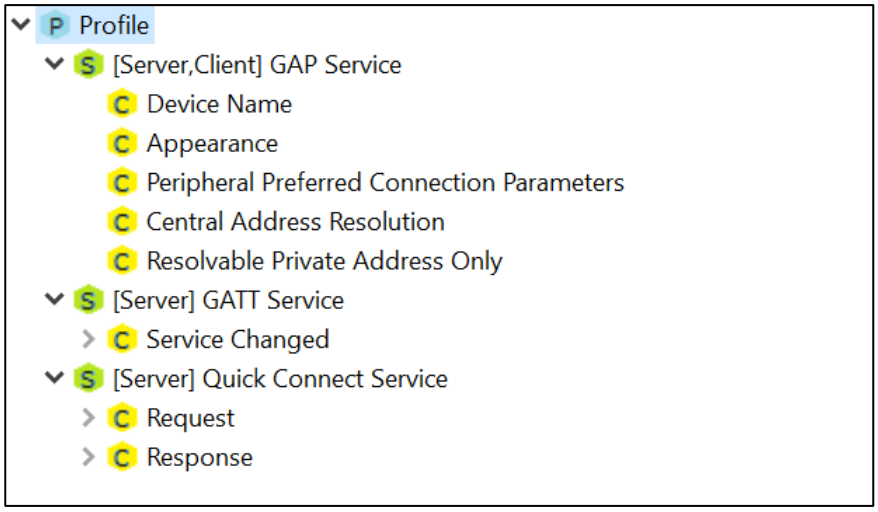

Figure 3: qe gen custom profile tool for BLE configuration

Fan Control Using HVPAK and Microcontroller

For this part of the project, the goal is to control a fan using a microcontroller (MCU) and an HVPAK (SLG47115V-DIP). The MCU generates a PWM signal (0-3.3V), which is then processed by the HVPAK to produce a high voltage PWM signal (0-12V) to drive the fan motor.

Software Development:

On the software side, I utilized the Renesas R_GPT API to generate and manage the PWM signal. The process involved using the following functions:

- R_GPT_Open: Initializes the GPT (General Purpose Timer) module.
- R_GPT_Start: Starts the GPT timer.
- R_GPT_InfoGet: Retrieves information about the GPT configuration.
- R_GPT_DutyCycleSet: Sets the PWM duty cycle.

I referred to the "gpt_ek_ra6m4" example project and sample code from the FSP documentation for guidance. The software implementation involves:

1. Calling R_GPT_Open and R_GPT_Start to initialize and start the PWM generation.
2. In the main loop of the sensor thread, separate queues handle the PWM duty cycle, set temperature, and set humidity.
3. The fan operates based on sensor readings: if the current temperature and humidity exceed the set values, the fan speed adjusts according to the user-defined PWM duty cycle.
4. Visual indicators are provided using LEDs: the blue LED (Port 4 Pin 15) signals when a PWM input is received, while the red LED (Port 4 Pin 0) indicates when the set temperature and humidity are lower than the current values.
5. A GUI for adjusting the PWM duty cycle was created by modifying the .json configuration file in the src folder.

Hardware Design:

The hardware design for the HVPAK was straightforward. The fan used is a 4-wire PWM fan with pins for VDD, PWM, FG (frequency generator), and Ground. The FG pin is not used in this project.

1. **HVPAK Configuration:**
    - Set the HV OUT CTRL block mode to "half bridge."
    - Connect the PWM output signal from the MCU to the IN0 pin and VDD to the OE0 pin of the HV OUT CTRL block. This setup ensures that the PWM pin of the fan receives the required signal.
    - Additionally, connect VDD to the IN1 and OE1 pins, which drives the corresponding output pin to 12V. This 12V is used as the VDD for the PWM fan.

Table 1. HV_GPO0_HD Half Bridge Logic

Sleep0	OE0	IN0	HV_GPO0_HD (Pin 7,8)	Function
1	X	X	Hi-Z	Off
0	0	X	Hi-Z	Off (Coast)
0	1	0	L	Brake
0	1	1	H	Forward

Figure 4: logic table dependent on system inputs

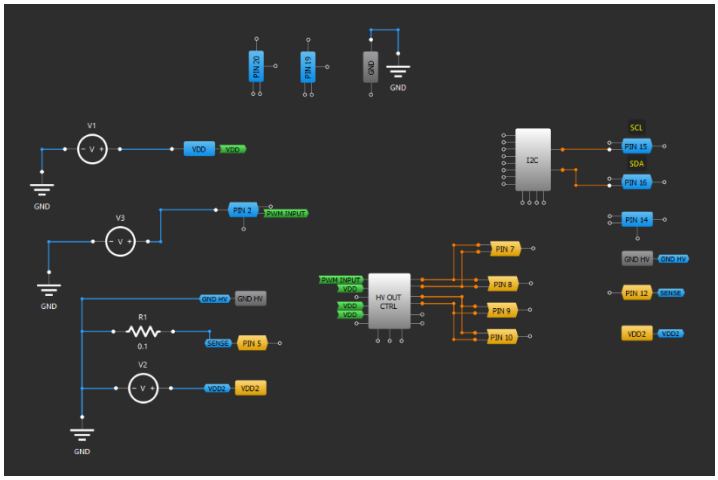

Figure 5: Go Configure HVPAK layout

Sequence Buck Regulators and Provide Hard Reset with GreenPAK

The design of my GreenPAK involved creating a sequencer for controlling Buck regulators, more complex than my HVPAK design. The GreenPAK was intended to manage the RAA808013 Buck Regulator, which activates when a voltage between 4.5V and 5.5V is applied. The RAA211820 Buck Regulator steps down from 12V to 5V.

Initially, I designed a simpler sequencer using only LUTs and Delay blocks for the on, off, and reset logic. However, with additional time at the end of my internship, I decided to develop a more advanced asynchronous state machine sequencer with visual indicators for overvoltage and undervoltage conditions.

Design Steps:

1. **Voltage Sensing:**
    - I used a voltage divider to step down the input voltage, making it suitable for input on Pin 6 of the GreenPAK. This was necessary because the current comparators in my configuration had a maximum comparison value of 1.2V (1200mV).

- The input voltage was tested with three different comparators to check if it was above 50mV, 900mV, or 1200mV.

2. **Asynchronous State Machine:**
    - I designed an asynchronous state machine with four states: Off (default), On, Overvoltage, and Undervoltage.
        - **Off**: Default state or when the voltage is below 50mV.
        - **On**: When the voltage is between 900mV and 1100mV.
        - **Overvoltage**: When the voltage exceeds 1100mV.
        - **Undervoltage**: When the voltage is between 50mV and 900mV.
3. **Logic Implementation:**
    - Using a combination of LUTs, delays, inverters, and combinational logic blocks, I generated output signals corresponding to each state of the asynchronous state machine.
    - Voltage dividers were used to drive LEDs: a blue LED for undervoltage and a red LED for overvoltage conditions.
4. **Reset Functionality:**
    - A reset button was incorporated with an inverted signal that drives low when pressed.
    - The reset functionality was implemented using an AND gate combining the reset signal with the "On" signal. If both signals were high, Pin 13 was driven high.
    - Pressing the reset button triggers the system to turn off for 800 ms and then turn back on.

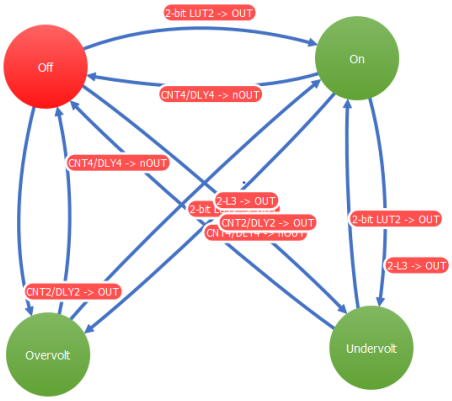

Figure 6: ASM editor diagram with states and transitions

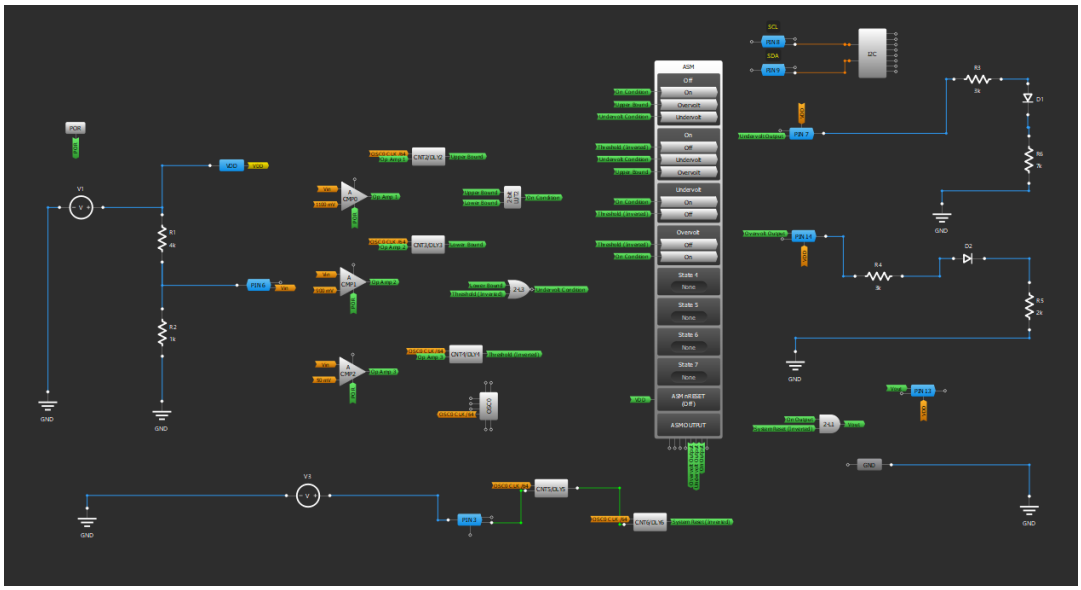

Figure 7: Go Configure AnalogPAK layout

Implementing a Real Time Clock

For tracking the system's uptime, I utilized the R_RTC API to implement a Real-Time Clock (RTC). The goal was to monitor how long the system has been running. The key functions used were:

- R_RTC_Open: Initializes the RTC module.
- R_RTC_CalendarTimeSet: Sets the RTC calendar time.

- R_RTC_CalendarTimeGet: Retrieves the current RTC calendar time.

Implementation Steps:

1. **Initialization:**
    - Defined rtc_time_t structures named g_set_time, g_get_time, and time_diff to manage time data.
    - Initialized each structure's fields (tm_hour, tm_isdst, tm_mday, tm_min, tm_mon, tm_sec, tm_wday, tm_yday, and tm_year) to null or default values.
2. **Setting Initial Time:**
    - Called R_RTC_Open to initialize the RTC.
    - Used R_RTC_CalendarTimeSet to set the initial calendar time based on the g_set_time structure.
3. **Time Tracking in the Thread:**
    - In the main loop of my sensor thread, repeatedly called R_RTC_CalendarTimeGet to update the g_get_time structure with the current time.
    - Calculated the difference between the current time (g_get_time) and the initial time (g_set_time) using the difftime function. Updated the time_diff structure with these differences for seconds, minutes, and hours.
4. **Calculating Uptime:**
    - Computed the total uptime in seconds using the formula:

```
alive_time = (int16_t) ((time_diff.tm_sec) + (time_diff.tm_min * 60) +
(time_diff.tm_hour * 3600));
```

Limitations:

- The RTC is configured for a 24-hour cycle, meaning it resets after 24 hours of operation.

This implementation effectively tracks the system's uptime, allowing for accurate monitoring of how long the system has been running.

Integrating the System and Enclosing the Hardware

Integrating the various components of the system presented several challenges, which I will address in the challenges section. However, the process of assembling and enclosing the hardware was straightforward. Here's how I approached it:

1. **Planning and Organization:**
    - Developed a layout plan for organizing the components within the enclosure.
2. **Drilling and Mounting:**
    - Drilled holes in the enclosure to allow for external wiring connections.
    - Inside the box, I placed the following components:
        - HVPAK
        - Microcontroller
        - BLE PMOD
        - Buck regulators
    - Mounted the following components externally:
        - PWM fan
        - GreenPAK with reset button
        - Sensor
3. **Wiring and Connections:**
    - Soldered jumper wires to the buck regulators and microcontroller to ensure reliable connections.

- Managed additional connections using a breadboard and connectors for flexibility and ease of integration.

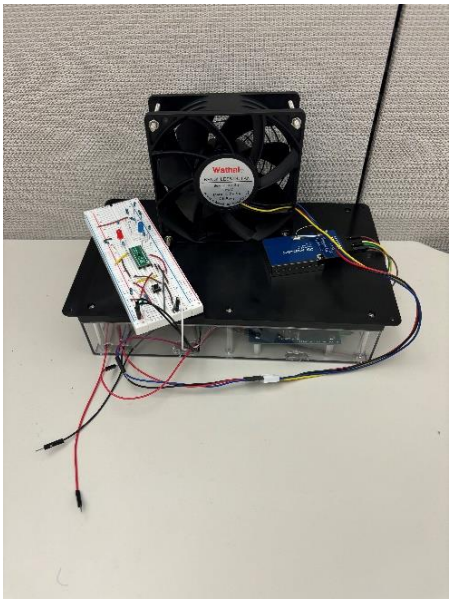

Figure 8: completed project with casing

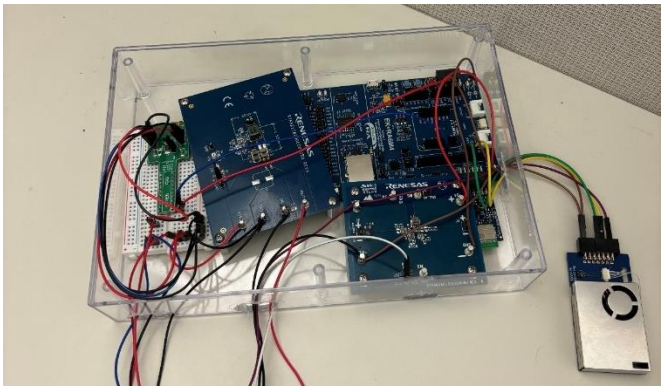

Figure 9: interior of completed project

Challenges Encountered During the Project

Throughout my internship, I faced several challenges while developing and integrating the system. Here's an overview of the issues encountered, and the lessons learned:

1. **Adapting HS3001 Sensor Code:**
    - Initially, I attempted to adapt code from the HS3001 temperature and humidity sensor for the RRH62000 sensor. This approach proved problematic because I did not fully understand the I2C protocol or the specifics of the RRH62000 datasheet. A more effective strategy would have been to thoroughly review the I2C protocol and the datasheet and refer to I2C API examples for basic read and write commands.
2. **Omitting Callback Functions:**
    - I initially overlooked the importance of callback functions. Without them, error handling was difficult. Implementing callbacks facilitated better debugging and error detection. Details on using callback functions are outlined in the section on establishing communication with the RRH62000.
3. **Incorrect Datasheet:**
    - The datasheet I used for the RRH62000 was incorrect, particularly regarding the byte order of variables. Ensuring the datasheet is accurate and up to date is crucial, especially for new sensors or components.
4. **Insufficient Software Delays:**
    - I encountered issues with I2C communication due to inadequate software delays between function calls. Adding appropriate delays often resolved communication problems by allowing time-dependent processes to complete.
5. **Write Then Read for I2C Communication:**

- My initial attempt to retrieve sensor data by directly issuing a read command failed. I later discovered that a write command was necessary before the read command to properly address the sensor. The correct sequence involves writing the address pointer to the sensor before executing the read command. Further details are available in the section on establishing communication with the RRH62000 and on my GitHub page.

6. **Incorrect Firmware on BLE PMOD:**
    - I initially used an incorrect firmware version for the DA14531 BLE PMOD. It is advisable to reflash any PMOD boards upon receipt. Details on this process are provided in the section on sending and receiving data through wireless communication.
7. **Learning Digital Circuits and Logic Design:**
    - With no formal training in digital circuits or logic design, I relied on the GreenPAK cookbook and training materials. Experimenting with configurations and observing the effects in software simulations proved to be the most effective learning method. Small adjustments and multiple simulations were crucial for troubleshooting.
8. **Issues with Asynchronous State Machine (ASM):**
    - My initial ASM design allowed transitioning to the off state only from undervoltage and overvoltage states, which led to issues with the system getting stuck in the undervoltage state even when the input voltage was 5V. This was due to the rise time of the power supply. I resolved this by adding a transition from the undervoltage state to the on state.
9. **Advanced Development Kit Simulation:**
    - Using the advanced development kit presented challenges, such as weak through-hole connections on the SLG4SA-DIP Adapter and pin number mismatches on the HVPAK. For instance, pin 6 of the HVPAK, which is supposed to be VDD2, cannot handle 12V if

connected to the advanced development board. Proper connection and pin function knowledge were essential to avoid damage.

10. **Sequencing the Second Buck Regulator:**
    - I faced difficulties in sequencing the second buck regulator. The output voltage read from the GreenPAK device was 5V, but it dropped to 2.5V when connected to the buck regulator's Vin terminal. This issue arose because a jumper connector was linking the enable and Vin pins, causing voltage splitting. It was necessary to provide separate voltages to the enable and Vin pins to meet the regulator's minimum voltage requirement.
11. **Bluetooth Connectivity Issues:**
    - After disconnecting from the debugger, I faced difficulties with Bluetooth connectivity. Despite successful operation when connected to the debugger, attempts to power the microcontroller through different sources failed to establish a Bluetooth connection. After investigating UART communication with a logic analyzer, I resolved the issue by adding a 1-second delay in the BLE thread before entering the app_main function. This delay allowed for proper initialization of components and helped prevent race conditions.
12. **BSP_SECTION_FLASH_GAP void Default_Handler Error:**
    - This error frequently occurred due to accessing variables outside their defined scope or missing necessary header files. Ensuring proper variable scope and including all required header files resolved these issues.
13. **Configuring gui_cfg.json:**

- Creating the JSON file for Quick Connect and QC Sandbox was challenging due to parsing errors. Careful attention to QC Sandbox sample code and using a diff check between working and new versions of the JSON file helped resolve these issues.

Suggestions for Future Improvements

My suggestions for future improvements are divided into two parts: enhancing the instructional material and training for the project and improving the project I created.

Instruction Improvements:

1. **More BLE Training:**
   - While the presentations from Patrick and Jon on BLE architecture, GAP/GATT frameworks, and the advertising and scanning processes were insightful, I found it challenging to connect these concepts to software implementation. Despite spending a day analyzing the app_main function to understand the process, I believe more targeted training material would be beneficial. Providing interns with in-depth resources on how BLE protocols are implemented in software, beyond the basics of the QE generator tool, would significantly enhance their understanding and ability to work with BLE technology effectively.
2. **Improving QuickConnect**
   - Quick Connect was a great way to help me improve the UI of my design. However, it had limitations. I think finding a way to “debug” .json files would significantly speed up the development process. Moreover, including configurations for a bar graph and improved graphing would help users implement more meaningful visuals. Also, more instructional material for having different object types on the same page would have been useful for my development.

Project Improvements

1. **Code Organization and Structure**
    - I could have organized my code for improved readability and efficiency. Tasks such as initializing I2C communication, implementing my RTC, and sending data over queues could have been broken down into functions. Additionally, creating structs for sending data over queues, reducing the number of queues I created, and streamlining variable declarations could have improved my code.
2. **Exploring Power Management**
    - Reducing power consumption and sending power consumption data over Bluetooth Low Energy would have been two great ways to improve my project.
3. **More Error Handling**
    - There are many possible moments where my project could fail: faulty I2C communication, BLE data not properly transmitted, etc. Although I do have some form of error handling in my code, implementing error codes to be transmitted via BLE could have been a good addition.
4. **Parts Selection**
    - I didn't experiment with using different microcontrollers, wireless frameworks, dc-dc converters, etc. There are likely parts I could have selected to reduce the wiring, reduce power consumption, or improve my project overall.

References

Basics of the I2C Communication Protocol (circuitbasics.com)

RA Flexible Software Package Documentation: Introduction (renesas.github.io)

FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions

Stack Overflow - Where Developers Learn, Share, & Build Careers

Wikipedia