EE 459: Embedded Systems Laboratory

Spring 2024

TEAM 13 Project Report

EMPOWERING AUTOMATED GROCERIES MANAGEMENT: A PERSPECTIVE ON SUSTAINABILITY

TEAM 13: *Justin Liu* * *Richard Chen* * *Sriya Kuruppath* *

{jliu3548, tsungyeu, kuruppat}@usc.edu

University of Southern California, USA

Table of Contents

ABSTRACT

Food waste has become a significant problem in our fast-paced society. This problem largely stems from the non-manageability, inadequate planning, and malfunctioning of refrigerators for food storage. According to the *Natural Resources Defense Council*, it is estimated that one out of four families in the United States wastes approximately $1,500 worth of food each year due to buying duplicate items or not consuming products before they expire. Moreover, about 76% of Americans often find expired food in their fridge, highlighting the need for better food management and storage. This inefficiency leads to financial loss and environmental harm, as wasted food in landfills emits methane. Improving kitchen and fridge organization can reduce waste and environmental impact while saving money. To this end, our project improves the issue by integrating a smart refrigerator system and a user-friendly mobile app that provides extended details of what is inside the refrigerator, leveraging efficient inventory tracking and management. We present two key advancements in efficient food management: (1) An adaptive inventory system that lets users store or remove grocery items on their local fridge interface and mobile app. It synchronizes details such as name, quantity, expiration date, and storage method between the two platforms. (2) Smart Fridge Alerts: Managing Expiry and Temperature for Food Safety. Demonstrating seamless performance, and safety metrics, our approach shows the promising potential for using FridgeHub for sustainability.

1 INTRODUCTION

Embedded systems have become an integral part of home smart devices. Our design of the Fridgehub consisted of key modules: a local fridge-embedded device governed by the Atmega328, a Raspberry Pi as a liaison between the main device and the database, and a mobile app. The proposed methodologies employ a combination of components such as a barcode scanner, LCD display, temperature sensor, keypad, and Raspberry Pi with the Atmega328p at its core. Along with a cloud storage database (Firebase) and a mobile app, we empower multimedia management of the fridge inventory. Through the hardware implementation, the Atmega328p retrieves the barcode number from the scanning mechanism. Subsequently, our local software interprets the message and matches the universal barcode number with its specific product

counterpart. The keypad can then used by the user to enter the expiration date of the product in the format of year/month/day. We also designate the keypad for the user to navigate through the list of menus in the inventory system that includes product name & quantity, expiration date, and storage method. From a safety perspective, the temperature sensor records real-time temperature and alerts the user via flickering the green LEDs if the temperature exceeds a set threshold. All of the information discussed above is illustrated to the user via our LCD display. After all, the information that was stored locally will be sent to the Raspberry Pi from the Atmega328p via SPI(Serial Peripheral Interface). Our FridgeHub features a user-friendly grocery management app developed with SwiftUI, optimizing inventory management. Connected to Firebase's real-time database, it ensures that all household members have instant access to synchronized grocery data. The app's architecture follows the MVVM (Model-View-ViewModel) design pattern. The ViewModel interacts with Firebase for data operations, fetching, and storing, while the View layer handles user interactions and data presentation. Along with the APP, we hope our product can provide the smart home device community with a comprehensive sustainability and safety standpoint to explore the usage of Fridgehub for the betterment of the environment.

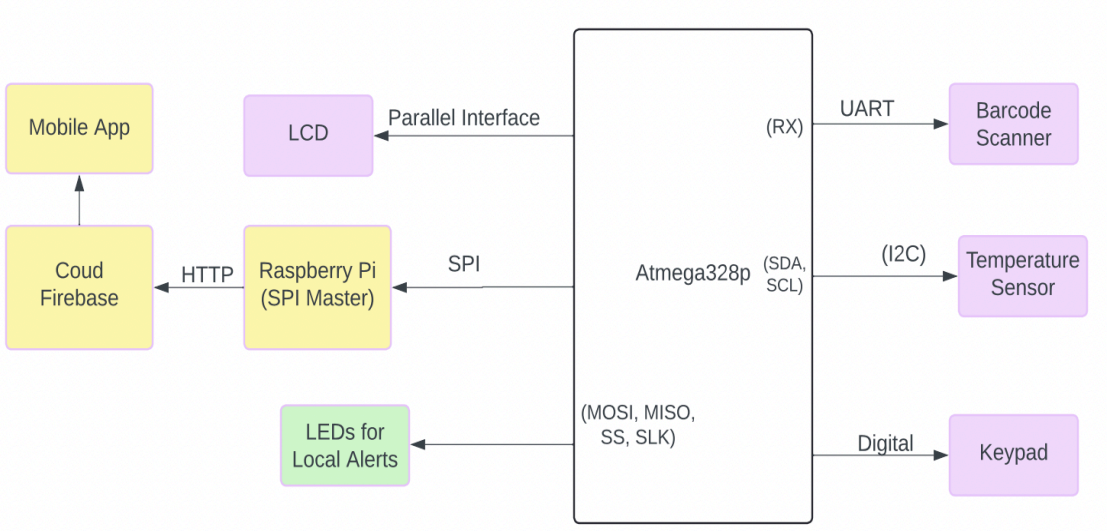


Figure 1: Overview of input-output integration of Fridgehub.

Figure 2: Picture of the actual product.

2 System Components

2.1 Crystalfontz LCD Display–Parallel Communication

In our project, we chose pins PD4-PD7 on the Atmega328p as data pins for parallel communication with the LCD. The higher-order nibble (the most significant 4 bits of the ASCII value) is sent first. Subsequently, the higher-order nibble is followed by the lower-order nibble (the least significant 4 bits). In parallel communication the Register Select (RS) and Enable (E) are being defined to govern the functionality of the LCD. We designated pin PB0 on the Atmega328p as the Register Select and PC0 as the Enable. The Register Select determines the mode of operation for the data sent to the LCD. When RS is low (0), the LCD interprets incoming data as a command (such as clear screen, cursor move), and when RS is high (1), it processes the data as ASCII characters to be displayed.

The Enable pin is used to latch the four-bit data presented on PD4-PD7 into the LCD's internal data registers. The Enable pin is essential in the timing of data transfer. It must be held high and then fiddled to low after presenting data on the data pins. When the Enable pin goes high, the

LCD captures the data present on PD4 to PD7. When the enable bit gets fiddled to low, the LCD starts to process the data.

2.2 Waveshare Barcode Scanner Module–UART

Hardware

The Waveshare Barcode Scanner offers efficient decoding of both 1D and 2D codes, including barcodes and QR codes. The manufacturer of the barcode scanner provides multiple modes of scanning and we chose the continuous scanning mode due to the continuous nature of product scanning tasks.

The barcode scanner features a UART(Universal Asynchronous Receiver/Transmitter) interface for integration with Atmega328p. In our configuration, we use the Tx pin from the barcode scanner for transmitting the barcode number to the receiving Rx pin on the Atmega328p. Additionally, the barcode scanner's Ground pin is connected to the Atmega328p's ground, and its power is supplied by the Atmega328p's 5V power supply based on the specifications in the user manual.

For reliable **UART** communication, we configured the following setup by scanning the setting codes on the user manual:

- Baud Rate: 9600 bps
- Data Bits: Set to 8 bits, which is the size of each data packet sent or received.
- Stop Bits: 1 stop bit, marking the end of each data packet.
- Parity: Set to none

Software

The software for the barcode scanner is written in two separate files called UART.c and UART.h. First, the *serial_init* function initializes the baud rate and configures the frame format (8 data bits, no parity, and 1 stop bit) for UART. This initialization ensures the data transmission rate is synchronized between the scanner and the Atmega328p.

```
24  void serial_init(unsigned short ubrr) {
25      UBRR0 = ubrr;  // Set baud rate
26      // Enable receiver and RX Complete interrupt
27      UCSR0B |= (1 << RXEN0) | (1 << RXCIE0);
28  
29  // Define constants
30  #define F_CPU 7372800
31  #define BAUD 9600
    #define UBRR_VALUE ((F_CPU/16/BAUD)-1)
```

We used the USART Receive Complete (RXC0) interrupt to enable the program to handle incoming data asynchronously. Specifically, the ISR(USART_RX_vect) function is called whenever a new byte is available in the UART data register (UDR0). As the barcode scanner manufacturer indicates, the barcode number will also be sent with a terminating (CARRIAGE_RETURN) symbol. Therefore, as barcode data bytes are received, they are appended to a buffer (barcode) until a carriage return character (CARRIAGE_RETURN) is detected. We set a count variable to keep track of the number of received characters, and barcode_ready is set to signal that a complete barcode is ready for use and processing once the terminating signal is received.

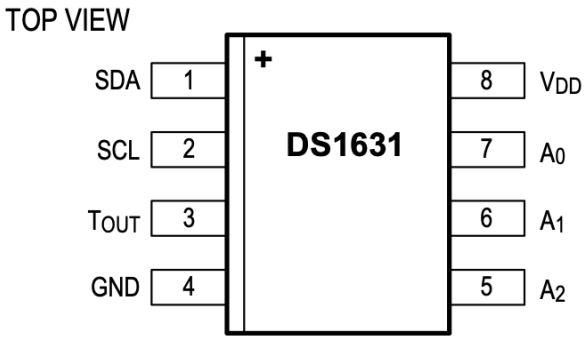
```
ISR(USART_RX_vect)
{
    char rx_char = UDR0; // Read the received data

    if (rx_char == CARRIAGE_RETURN)
    {
        if (count > 0)
        {
            // Null-terminate the string
            barcode[count] = '\0';
            // Set the flag indicating that a full barcode is ready
            barcode_ready = 1;
            // Prepare for the next barcode
            count = 0;
        }
    }
    else if (count < MAX_BARCODE_LEN)
    {
        barcode[count++] = rx_char; // Store the character
    }
}
```

2.3 Temperature Sensor–I2C

Hardware

The temperature sensor that we used in this project is Analog Devices' DS1631. It supports the I2C communication protocol. There are 8 pins for this device as shown in the diagram on the right. The device has a 7-bit slave address which is $1001A_2A_1A_0$. A_2, A_1, and A_0 are pins 7, 6, and 5. We wired pins 7, 6, and 5 to ground in our project. Therefore, the slave address of our temperature sensor is 10010000 (=0x90). A zero is concatenated at the end of the 7-bit address because we did a reading operation. If we wrote something to the temperature sensor, then the last bit would be zero.



Software

To communicate with the I2C devices, we have a function called i2c_io(uint8_t device_addr, uint8_t *wp, uint16_t wn, uint8_t *rp, uint16_t rn). The first input is the slave device address, the second input is the write buffer, the third input is the number of writing

operations, the fourth input is the read buffer, and the last input is the number of reading operations. Before going into the while(1) loop, we first initialized the temperature sensor by writing 0xAC and 0x51 which accesses the configuration register and starts the temperature conversion.

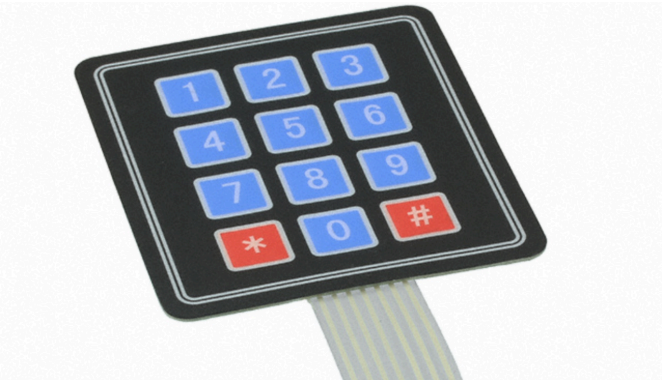
```
317     // Temperature sensor initialization
318     wdata[0] = 0xAC; // Set config for active high = 1 and continuous acquisitions
319     status = i2c_io(Tempsensor_ADDR, wdata, 2, NULL, 0);
320     wdata[0] = 0x51; // Start conversions
321     status = i2c_io(Tempsensor_ADDR, wdata, 1, NULL, 0);
```

In our code, the int getTemperature() function is implemented to get the integer value of the temperature in Fahrenheit as shown below. The 0xAA would read the last converted temperature value from the 2-byte temperature register. The lines from 90 to 94 take the temperature value out from the read buffer and line 95 converts the temperature from Celsius to Fahrenheit.

```
79  int getTemperature() // function that returns the temperature
80  {
81      uint8_t wdata[20];
82      uint8_t rdata[20];
83      uint8_t status;
84      int temperature;
85
86      // Set the sensor to read temperature
87      wdata[0] = 0xAA;                                    // Command to read temperature
88      status = i2c_io(Tempsensor_ADDR, wdata, 1, rdata, 2); // Assuming Tempsensor_ADDR is defined as before
89
90      unsigned int c2 = rdata[0] * 2;
91      if (rdata[1] != 0)
92      {
93          c2++;
94      }
95      temperature = (c2 * 9) / 10 + 32; // Convert to Fahrenheit
96      return temperature;
97  }
```

2.4 Keypad–Digital

Hardware



We used a 4x3 keypad in our project. There are 4 inputs for the rows and 3 inputs for the columns. Each combination of the row and the column can be assigned to a button. Therefore, there are 12 buttons on the device. Each button is like a switch. When a button is pressed, its corresponding switch will be open.

Software

We used a scanning mechanism for the keypad. We first set high to all the columns and rows and detect if any of them are low. Whenever a low is scanned, its corresponding column and row could be further used to locate the pressed button. The big for loop from lines 52 to 85 is for iterating the column. The lines from 55 to 65 make the iterated column low. The for loop from lines 68 to 80 is the scanning mechanism for the row. Therefore, for each iterated column, all the rows would be scanned once. We used a 5 ms delay to avoid button debouncing. 5 ms interval works the best for the users to input their desired character after we tested all the different lengths of delay.

```
52      // Scan columns
53      for (int col = 0; col < 3; col++) {
54          // Set all columns to high before grounding one
55          COLS_PORT |= (1 << COL1) | (1 << COL2);
56          COL3_PORT |= (1 << COL3);
57
58          // Ground the current column
59          if (col == 0) {
60              COLS_PORT &= ~(1 << COL1);
61          } else if (col == 1) {
62              COLS_PORT &= ~(1 << COL2);
63          } else {
64              COL3_PORT &= ~(1 << COL3);
65          }
66
67          // Scan rows
68          for (int row = 0; row < 4; row++) {
69              _delay_ms(5); // Delay for debouncing
70
71              if (row < 3) {
72                  if (!(ROWS_PIN & (1 << (PC1 + row)))) {
73                      return keys[row][col];
74                  }
75              } else {
76                  if (!(ROW4_PIN & (1 << ROW4))) {
77                      return keys[row][col];
78                  }
79              }
80          }
81
82          // Set column pins back to high after scanning
83          COLS_PORT |= (1 << COL1) | (1 << COL2);
84          COL3_PORT |= (1 << COL3);
85      }
```

2.5 Raspberry Pi–Serial Peripheral Interface

Hardware

In this project, we used the Raspberry Pi Model 4 B as a liaison between the Atmega328p and the Firebase. The Atmega328p sends the barcode number followed by the expiration date of the item being added to the inventory to the Raspberry Pi. If an item is being removed from the local machine, we append a “!” in the front of the message sent, signaling the mobile device to remove the item. Similarly, the mobile device sends signals differentiating add and delete to the local device via Raspberry Pi for read-time synchronization. In the SPI framework, we configured the Raspberry Pi to be the master and the Atmega328p to be the slave. The four essential connections between the two devices for SPI protocol are MISO (Master In Slave Out),

MOSI (Master Out Slave In), SCK (Serial Clock), and SS (Slave Select). **Note:** Since the Atmega328p operates at 5V, whereas the Raspberry Pi Model 4 B operates at 3.3V, we need to address the voltage disparity when connecting these devices. By using a voltage logic level converter, we adjust the voltage levels of the connection lines. Namely, (1) signals sent from the Atmega328p (5V) to the Raspberry Pi are stepped down to 3.3V; (2) signals sent from the Raspberry Pi (3.3V) to the Atmega328p are stepped up to 5V.

Software

Master-Slave Interaction: SPI operation empowers the master device to control the communication flow and timing. The master initiates communication by generating a clock signal and enabling the slave select (SS) line. The slave responds to the master based on this clock. Data exchange happens synchronously with the clock pulses provided by the master.

Continuous Polling by Master: Since the master(Raspberry Pi) has no knowledge about when the barcode number or the temperature from the Atmega(328p) will be triggered to be sent, we implement a polling logic from the master. The master continuously sends a dummy byte (0xFF) to generate the necessary SPI clock signals, prompting the slave to respond. If the slave has no barcode or temperature to send, it replies with a dummy byte (0x00).

Barcode or Temperature Available: When an item is being scanned or a temperature reading crosses a threshold, the Atmega328p captures this data and stores it in the SPI message buffer. On the next SPI transaction, instead of sending a dummy byte, the slave sends the actual data from its buffer. The master receives this data instead of the usual dummy byte and sends the message to the server.

Code for polling for exchange from the master:

```
def spi_exchange(data):
    """Exchange data with the MCU via SPI."""
    response = spi.xfer2(data)
    return response

def receive_barcode():
    barcode = ""
    while True:
        response = spi_exchange([0xFF])  # Send 0xFF as the dummy byte to poll the MCU
        if response[0] != 0:  # If response is not null
            while response[0] != 0:
                barcode += chr(response[0])
                response = spi_exchange([0xFF])
            break
    print(f"Received barcode: {barcode}")
    return barcode
```

Code for the slave to send exchange messages:

```
ISR(SPI_STC_vect) {
    // Signal the server that the local stored an item
    if (message_ready) {
        strncpy(send_buffer, Send_message, MAX_BARCODE_LEN - 1);
        send_buffer[MAX_BARCODE_LEN - 1] = '\0'; // Correctly terminate within bounds
        //barcode_ready = 0;
        send_index = 0;
        message_ready = 0;
    }
    // Signal the server has local deleted an item
     else if (barcode_ready && (state == Delete)) {

        send_buffer[0] = '!';
        strncpy(send_buffer + 1, barcode, MAX_BARCODE_LEN - 1);
        send_buffer[13] = '\0';

        barcode_ready = 0;
        send_index = 0;
    }

    if (temp_high && count == 0) {
        strcpy(send_buffer, "   Temperature: "); // Copy the initial part of the message
        int length = strlen(send_buffer); // Get the length of the string already in send_buffer
        sprintf(send_buffer + length, "%d", temperature); // Append the temperature

        send_buffer[20] = '\0'; // Correctly terminate within bounds
        //temp_high = 0;
        count ++;
        send_index = 0;
        barcode_date = 0;
    }
```

Information from Master to Slave: In scenarios where the items being stored or removed from the APP need to be synchronized to the local fridge device, we enable the full duplex communication of SPI. Once the message is prepared, the Raspberry Pi begins sending the data to the MCU where the spi_exchange function is called for each byte in the prepared list. This function sends a byte to the MCU and simultaneously reads a byte from the MCU, allowing for full-duplex communication. Even though the Raspberry Pi might not use the exchanged incoming data during this operation, the MCU could use this opportunity to send back dummy data as acknowledgments.

Code for the master to send a barcode message when an item is being added to the APP:

```
def send_message_to_mcu(message):
    """Send a string message to the MCU, followed by a null byte to indicate end of message."""
    message_bytes = [ord(char) for char in message] + [0]  # Append a null byte at the end
    for byte in message_bytes:
        spi_exchange([byte])
        time.sleep(0.05)  # Short delay to ensure the MCU processes each byte
```

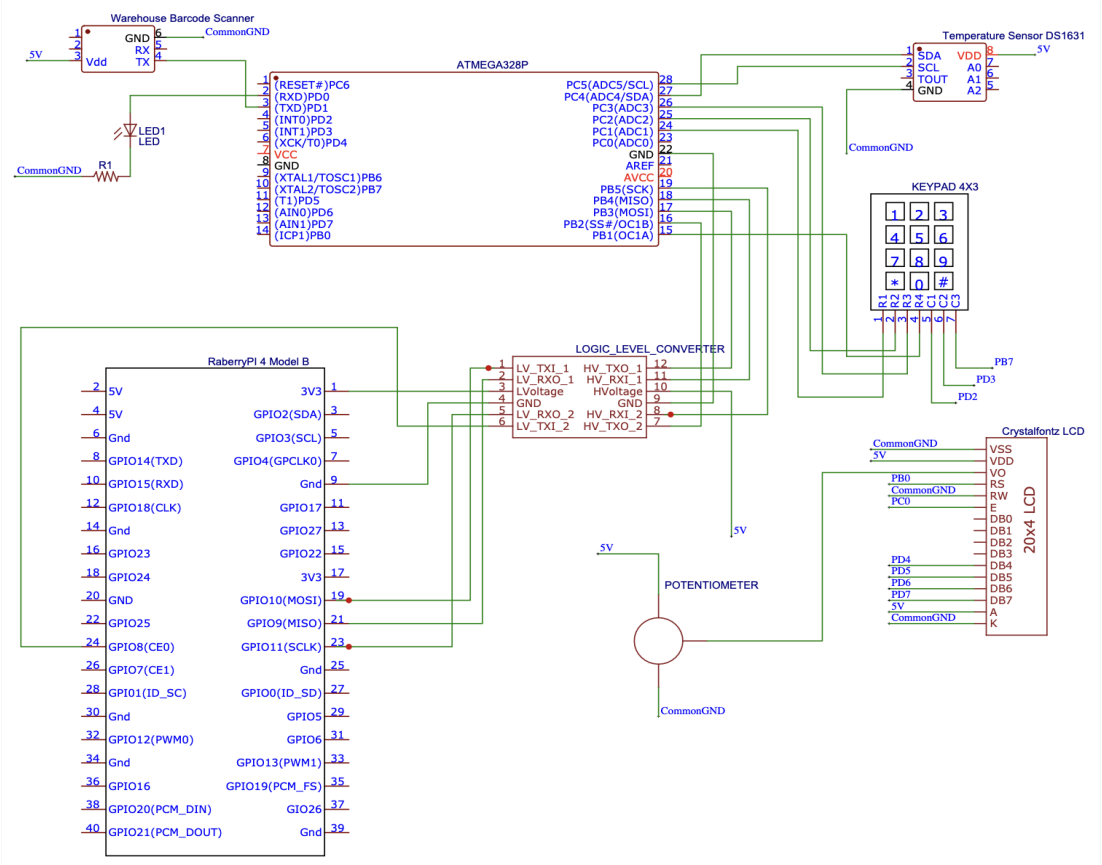
Code for the slave to receive the message into a buffer:

```
ISR(SPI_STC_vect) {
    /////////Receiving the stuff from RPI
    char received = SPDR;  // Read the received data first, crucial for full-duplex SPI

    // Process the received data
    if (received == '\0')
    {
        if(response_index > 0)
        {
            response_buffer[response_index] = '\0'; // Null-terminate the string
            response_complete = 1;
            strcpy(received_message, response_buffer);
            response_index = 0;  // Reset index for next message
        }
    }

    else if(received != 0xFF)
    {
        if (response_index < MAX_BARCODE_BACK)
            response_buffer[response_index++] = received;
    }
```

3 SCHEMATICS DIAGRAM



4 Remote Inventory Management

4.1 Firebase Database:

Firebase serves as the central hub in the Fridgehub, linking the mobile application, Raspberry Pi, and the stored data. As a NoSQL database, Firebase utilizes a JSON-like storage format and offers real-time data synchronization, which is best for applications that require immediate updates across user interfaces and devices like ours.

Through interaction with the mobile app, users can input or update information such as adding new items to the fridge, updating quantities, or setting expiration dates. When these changes are made, the mobile app communicates with Firebase, sending the updated data to the database in real time. Conversely, any updates made elsewhere—via the Fridgehub local device— constantly update Firebase, and those changes are instantly pushed to the mobile app. The real-time characteristic of Firebase ensures instant synchronization of any added, removed, or modified data across all connected devices and interfaces.

This is how data is structured in our database:

- Item Entries: Each item is represented as a key under "groceries", with the key typically being a human-readable identifier that uniquely describes the item.
- Item Attributes–Barcode: Each item has a barcode attribute that stores a unique numerical identifier, which can be used for scanning and quick identification through automated systems like the Raspberry Pi setup.
- Category: This describes the type of the product, helping in categorization and potentially in filtering operations within the app.
- Expiration Date: Monitors product freshness and triggers notifications as the date approaches.
- Location: Indicates where the item is stored within the fridge (either fridge or freezer).
- Quantity: Reflects the number of units available in the fridge.

4.2 Raspberry Pi–Server/Client:

The Raspberry Pi can act both as a server and as a client. When it acts as a server, it listens to updates on Firebase acted by the APP. The corresponding client code (that runs on a local machine) will continuously poll the firebase and detect any changes in inventory or temperature. Once a change is detected, firebase will send the barcode that is followed by an expiration date as its message to the Raspberry Pi to indicate an add action or "!" followed by only the barcode number to indicate a remove.

```python
def process_data(data, conn):
    """Process received data and perform actions based on message format."""
    message = data.decode().strip()
    print(f"Received: {message}")

    if message.startswith("!"):
        # This is a zero quantity update, message should be "!barcode"
        barcode = message[1:]  # Remove the '!' to get the barcode
        print(f"Zero quantity for Barcode: {barcode}")
        send_message_to_mcu(f"!{barcode}")  # Send command to MCU
    elif "," in message:
        # This is a regular item update with barcode and expiration date
        barcode, expirationDate = message.split(',', 1)
        print(f"Barcode: {barcode}, Expiration Date: {expirationDate}")
        send_message_to_mcu(f"{barcode},{expirationDate}")  # Send both barcode and date to MCU
    else:
        print(f"Error processing data: Unexpected format. Received data: {message}")
        conn.sendall("Error: Incorrect data format. Please use ',' as a delimiter.".encode())
```

In its role as a **client**, the Raspberry Pi engages directly with Firebase to fetch the latest data from the Atmega328p and performs real-time updates to cloud-based data. This functionality is essential to promptly update the inventory details in Firebase when the local Fridgehub receives messages via the connected barcode scanner and temperature sensors. For instance, if the temperature is too high, the Atmega328p sends the temperature to the Raspberry Pi. Subsequently, Firebase can retrieve this data from the Raspberry Pi and act on it by notifying users via the app.

```python
def handle_data_update(event, conn):
    parts = event.path.strip('/').split('/')
    item_name = parts[0]
    data = event.data

    print(f"Handling update for {item_name}, Path parts: {parts}, Event type: {event.event_type}, Data: {data}")

    if len(parts) == 1:
        # Single part path, need to check if it's a full item update or a specific 'quantity' update
        if isinstance(data, dict) and 'quantity' in data and len(data) == 1:
            # Specific 'quantity' update
            quantity = int(data['quantity'])
            print(f"Detected quantity-only update via single path for {item_name}: {quantity}")
            handle_quantity_update(conn, item_name, quantity)
        else:
            # Full item update or other type of update
            update_cache(item_name, data)
            if 'quantity' in data:  # Check if quantity was part of a full update
                handle_quantity_update(conn, item_name, int(data['quantity']))
            if 'barcode' in data or 'expirationDate' in data:
                send_item_details_notification(conn, item_name)
                print(f"Sent full item update for {item_name}")
    elif len(parts) == 2:
        property_name, property_value = parts[1], data
        update_cache(item_name, {property_name: property_value})
        if property_name == 'quantity':
            quantity = int(property_value)
            print(f"Quantity update for {item_name} to {quantity}")
            handle_quantity_update(conn, item_name, quantity)
        else:
            send_item_details_notification(conn, item_name)
            print(f"Sent update notification for non-quantity change for {item_name}")
```

4.3 The App:

Fridgehub utilizes a struct to represent a grocery item with properties such as name, barcode, category, quantity, location, and expiration date. This struct conforms to *Identifiable* for easy handling in SwiftUI lists and *Equatable* for equality checks.

```
struct GroceryItem: Identifiable, Equatable {
    let id = UUID()
    var name: String
    var barcode: Int
    var category: String
    var quantity: Int
    var location: Location
    var expirationDate: Date
}
```

GroceryViewModel:

We use a *GroceryViewModel*, a view model class that conforms to *ObservableObject*. This allows SwiftUI views to react to changes in the data it manages, such as updates to the list of grocery items. The view model includes methods for adding and deleting groceries from a Firebase, indicating that the app uses Firebase as its backend for data storage. It also handles barcode lookups, presumably to verify item details when a barcode is scanned. *GroceryViewModel* interacts with Firebase to perform CRUD operations—fetching, adding, and updating grocery data. The *fetchGroceries* method listens for real-time updates from the Firebase database, reflecting changes immediately in the app's UI

```
struct BarcodeLookupView: View {
    @State private var barcode: String = ""
    @State private var lookupResult: GroceryItem?
    @ObservedObject var viewModel: GroceryViewModel // Assuming
        ViewModel is injected or shared

    var body: some View {
        NavigationView {
            VStack {
                TextField("Enter barcode", text: $barcode)
                    .keyboardType(.numberPad)
                    .padding()

                Button("Lookup") {
                    performLookup()
                }
                .padding()

                if let item = lookupResult {
                    Text("Item: \(item.name)")
                    // Display other item details as needed
                }
            }
            .navigationTitle("Barcode Lookup")
            .padding()
        }
    }

    private func performLookup() {
        guard let barcodeInt = Int(barcode) else { return }
        lookupResult = viewModel.itemForBarcode(barcodeInt)
    }
}
```

APP Structuring:

ContentView: The root view that contains a TabView with multiple tabs for listing groceries by location, handling temperature, notifications, and barcode lookup.

AddItemView: A view for adding new grocery items. It presents a form where users can input item details and add the item to the list.

EditableDetailView: A view where users can modify item properties and save changes.

SwiftUI Form Components

Forms: Used in *EditableDetailView* and *AddItemView* for inputting and editing item details. They contain *TextField* components for textual inputs, *Picker* for selecting the location, and *DatePicker* for choosing expiration dates.

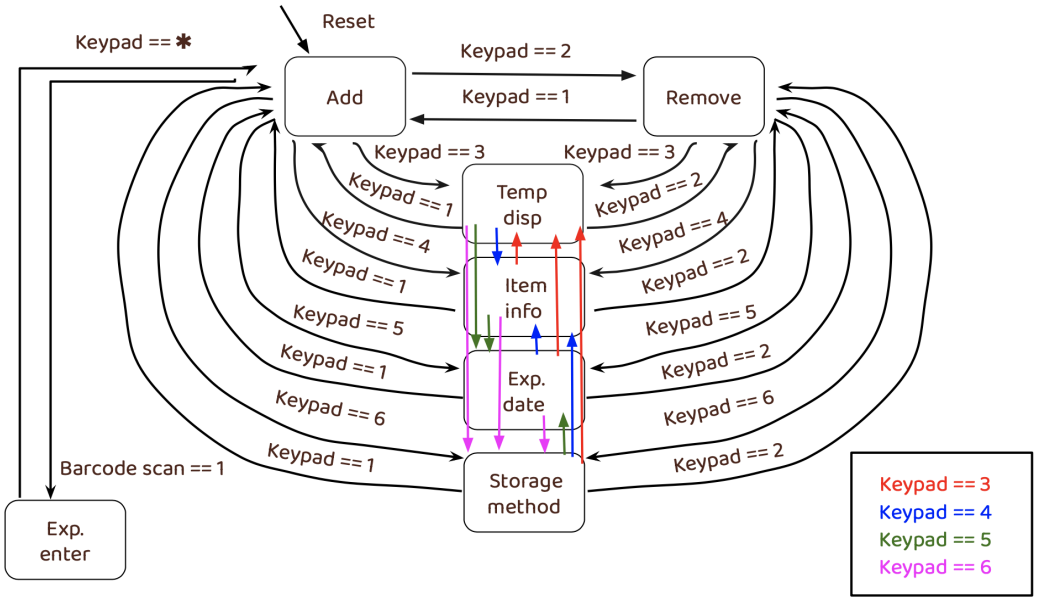
Interaction Components

List Handling: *ForEach* is used within *ListView* to dynamically display items based on their location and other properties. It supports deleting items using swipe gestures.

Asynchronous Data Handling: The app handles data fetching asynchronously using closures to ensure that the user interface remains responsive. Updates to the UI are performed on the main thread.

Error Handling and Data Validation: Before adding or updating items, the app checks the validity of input data such as barcode and quantity being integers.

5 STATE MACHINE



There are seven states in the Fridgehub. On reset, the state would be initialized to *ADD* state. To add an item to the system, the user has to be in the add state and scan the item's barcode. If an item is scanned in the add state, the state machine takes the user to the *Expiration Date Setting*

state. The user can then type in the date by pressing the number button on the keypad. After everything is typed down, press the star button on the keypad to go back to the ADD state which would complete the whole storing operation. To delete an item, the user can press 2 to transition to the REMOVE state and scan the barcode. The corresponding item would then be removed. Each state has its number to transit to except the Expiration Date Setting state. For example, *ADD* is one, *REMOVE* is two, *Temperature Display* is three, *Item Information* is four, *Expiration Date Display* is five, and *Storage Method* is six. The barcode scanning logic would function in *ADD* and *REMOVE* states which means that nothing would happen if the users scan items in states other than add and remove.

6 USER MANUAL

Functions	Conditions	Steps
Add item	At add state	1. Scan the barcode 2. Enter expiration date 3. Press key button
Remove item	At remove state	Scan the barcode
Display temperature	At any of the state	Press number 3 button
Display item information	At any of the state	Press number 4 button
Display expiration date	At any of the state	Press number 5 button
Display storage method	At any of the state	Press number 6 button
Temperature warning	At any of the state	No action is needed. Warning would display automatically if the temperature is higher than the threshold

7 Challenges:

7.1 SPI Contention:

While implementing the SPI protocol for full-duplex communication between the Atmega328p and the Raspberry Pi, we encountered difficulties because the MOSI, MISO, and SLK pins were preoccupied with the use of the programmer. To resolve this conflict, we utilized female jumper wires for connecting to the Atmega328p's pins, allowing us to easily disconnect the SPI during programming. That way, we were able to fit the SPI lines on the pins that were used.

7.2 Hardware Debugging:

In the first few weeks of building the project, our progress was slow because we were not familiar with hardware debugging. We underestimated the importance of using an oscilloscope for debugging. Without checking signals at both ends, it's impossible to determine if issues are hardware or software-related. The Raspberry Pi that we used originally for this project malfunctioned. We struggled to find out that the error was caused by the hardware until we ran the testing code of the Raspberry Pi and analyzed the signal on the oscilloscope. If we had realized the importance of testing hardware components before implementation, we could have saved a lot of time.

8 Engineering Standards:

- **ISO 9241-210**: Requirements and recommendations for human-centered design principles and activities throughout the life cycle of computer-based interactive systems.
- **ISO 14040:** Principles and framework for life cycle assessment (LCA), including goal definition, inventory analysis, impact assessment, and value choice conditions.
- **ISO/IEC 27001**: Requirements for Information Security Management Systems
- **IEC 60335-1**: Safety of Household and Similar Electrical Appliances. Ensures that the sensor and its integration into the fridge meet stringent safety criteria.

Implementation of Standards: The Fridgehub is designed to prevent food waste caused by buying duplicative items and poor planning for consumption. We provide efficient grocery

tracking platforms with optimized local device and mobile interfaces. Our system includes a temperature sensor for power outage alerts. Additionally, reducing food waste with Fridgehub helps combat climate change by decreasing methane emissions from landfill decomposition.

9 Conclusion and Future Improvements:

From conceptualizing to implementing the Fridgehub based on embedded systems, we learned the imperative lesson of planning the scheme of the design with as much detail as possible, including carefully selecting the communication protocol that serves distinctive goals. In this project, our team used all of the communication protocols for hardware including parallel, I2C, UART, and SPI. It may have caused more than the work that's planned but we are happy that we have learned the implementation of all the crucial communications that can potentially be applied to our future engineering endeavors. Additionally, this project introduced a new perspective on integrating ethics and sustainability into our design process. This perspective has made us ponder on the deeper subject of the ultimate purpose of learning. Through learning, lives are enriched and society bettered by engineering designs that bring positive impacts to humanity. We are committed to making positive contributions through our future designs.

Looking ahead, FridgeHub aims to enhance its functionality and user experience through several upgrades. We aim to optimize the product for its aesthetics, transitioning from a utilitarian appearance to a more integrated and seamless design, eliminating exposed wires and circuitry. Furthermore, we plan to add price monitoring features for easier financial management, allowing users to track grocery spending with detailed budget recaps. We also aspire to explore AI enhancements to predict expiration dates more accurately based on local climate and usage patterns. Additionally, a recipe suggestion feature will help minimize waste by proposing meals tailored to use up leftovers and available inventory.

References:

Link to project source code and video of demo:
https://drive.google.com/drive/u/0/folders/1eVc-Aa65MBvGPWCVOR1CrISV-lPuoEKu

Appendix–Signature Sheet

	Justin Liu	Richard Chen	Sriya Kuruppath
System design	33%	33%	33%
Component selection	50%	50%	0%
MCU Installation	50%	50%	0%
Temperature Sensor	0%	100%	0%
LCD Display	50%	50%	0%
Barcode Scanner	100%	0%	0%
Keypad	50%	50%	0%
Raspberry Pi	55%	20%	25%
State Machine	0%	75%	25%
Database	0%	0%	100%
App	0%	0%	100%
Project report (oral)	30%	30%	40%
Project report (written)	50%	30%	20%

Justin Liu:_____

Richard Chen: 

Sriya Kuruppath: