EC544 Networking the Physical World
2022 Spring

Technical Report

A Data Encoded Neck Healthcare System Based on FRDM Sensor

Team05
Xiaopeng Huang (xphuang@bu.edu)
Tianyi Xu (tyx@bu.edu)

Introduction:

Nowadays people are facing all kinds of circumstances where they need to be sitting for a long time, either for studying or working in the office. As time passes, this would easily lead to some physical illness like neck issues, unconsciously. With the modules we have learned in this class, we would like to develop a healthcare embedded system aiming to solve this problem based on some hardwares we have. FRDM development board becomes a good choice for its small size, powerful characteristics and Arm-based toolchain.

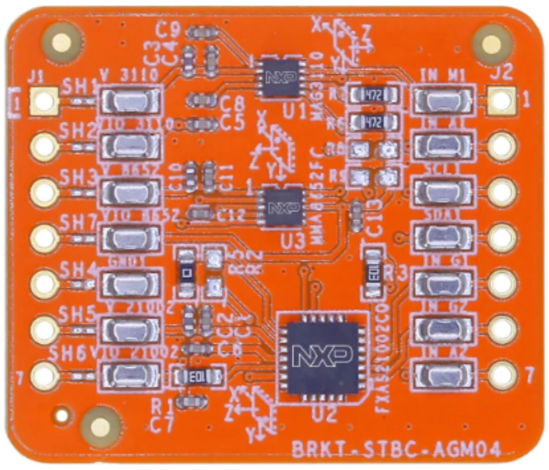
The FRDM-K64F-AGM04 is a 9-axis sensor toolbox Demonstration Kit including the FRDM-K64F MCU board and the FRDM-STBC-AGM04 Sensor Shield, as shown on Figure 1. “AGM” stands for Accelerometer, Magnetometer and Gyroscope, which serves as a stabilized compass that can sense the 3-D movements (rotation, acceleration, etc.) of the board, and generate an accurate orientation estimate.Inspired by this powerful sensor system, we want to develop the FRDM board into a wearable healthcare device to track people’s body position, for example, checks how long one has been sitting or not moving his neck. In the end, information will be received by another terminal device – Raspberry Pi, representing the user end, for visualization.

We regard these output data as valuable personal information, which cannot be reproduced again. Meanwhile, gyroscope and accelerometer information is streaming in realtime at high frequency. Therefore, one efficient way to secure the integrity of the data is of our main concern. We would like to introduce error correction code to encode this personal information, which would increase the failure tolerance of the data storage system. Finally, on the server side, one distributed data storage structure would make full use of the parity bits we computed in the error correction codec, and be able to decode the original data even if one small portion of the data storage devices are dysfunctional.

As thus we can divide the whole project into several main tasks:

a. Plan the movements. Use FRDM-K64F-AGM04 sensors to detect movements and generate data with Freedom Sensor Algorithm.
b. Encode the data before transmission.
c. Store the data and the parity bits to one distributed storage system

d. Fetch the texts and the parity bits from the distributed storage and decode the original data for analysis or other functions.



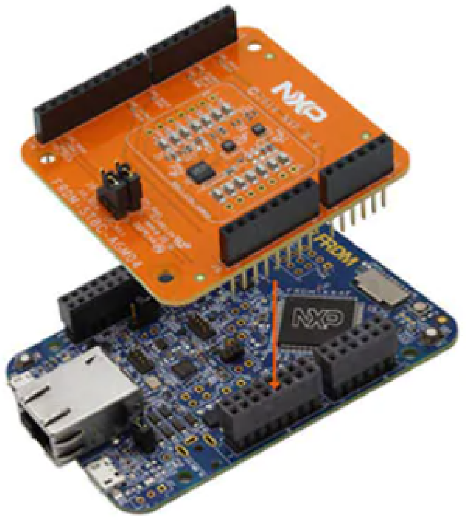


Figure 1. BRKT-STBC-AGM04 Shield and FRDM-K64F Board

Resources:

Hardware:

- FRDM-K64F board
- FRDM-K64F-AGM04 Sensor Shield (Including Gyroscope / Accelerometer / Magnetometer)
- Raspberry PI 4
- Cat-6 Ethernet Patch Internet Cable
- USB cables
- Routers / Switch
- Micro SD cards

Software:

- Freedom Sensor Fusion Toolbox GUI
- Tiltmeter Sensing Algorithm
- Intelligent Sensing Framework
- Sensor Fusion Library

- Arm Mbed OS
- Raspberry Pi OS

Discussion of Failure

1. Misunderstanding of the Reed-Solomon Code
   As we stated in the technical proposal, we would like to implement a “Encryption and Distributed System” to store the sensor value we collected from the Freedom board’s onboard sensor shield. We stated that we would investigate methods like the Reed-Solomon algorithm and Fontain Code for the distributed storage system. However, as we investigated the implementation and theory of the Reed-Solomon Code, we found that this code is only one method to implement the error-correction code instead of “encryption then distributed code” as we expected in the first time. As a result, we haven’t successfully implemented the “encryption” part as we stated in the proposal. What we achieved is the on-board implementation of encoding part of Reed-Solomon Algorithm. The decoding workload is implemented in the data server side, to work as an error correction mechanism.

2. 3-D Movement Detection of Neck
   Originally we were trying to utilize the FRDM sensor to detect 3-D movement of the user's neck, and the gyroscope can indeed sense the board movement in all of the 3 axes. However, we found that the gyroscope sensor (FXAS21002C on the FRDM-K64F) measures only angular rotation rate and it has no ability to sense orientation but only changes in orientation. Considering that the main focus of our application is to get the angles of the neck’s movement , we turned to using the accelerometer (MMA8652FC) on the FRDM board to measure the earth's gravitational field rotated according to its orientation. As a result, the movements happening in the horizontal plane can not be detected, which means that there will be no output changes if the user turns around his/her head horizontally. It is still satisfying that we can get the data of 2-D movement on two

vertical planes, where the user moves his/her neck most frequently, as shown on Figure 2 below.

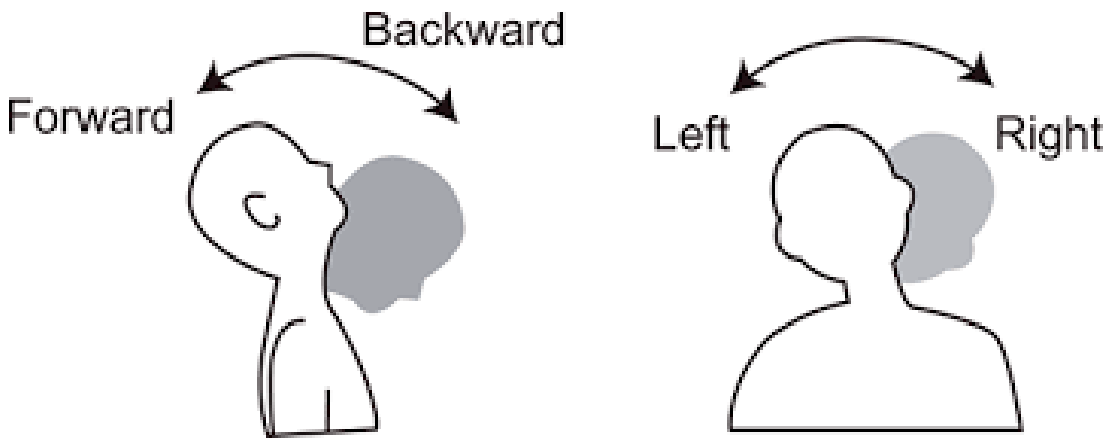


Figure 2. Detected 2-D Movement of Neck

Deployment Details

This project is divided into the following four modules for development.

1. Sensor and Positioning Algorithm for Verification
2. Message passing and cross device transmission
3. Error Correction Code and Distributed Storage System
4. Server Command Line Feedback

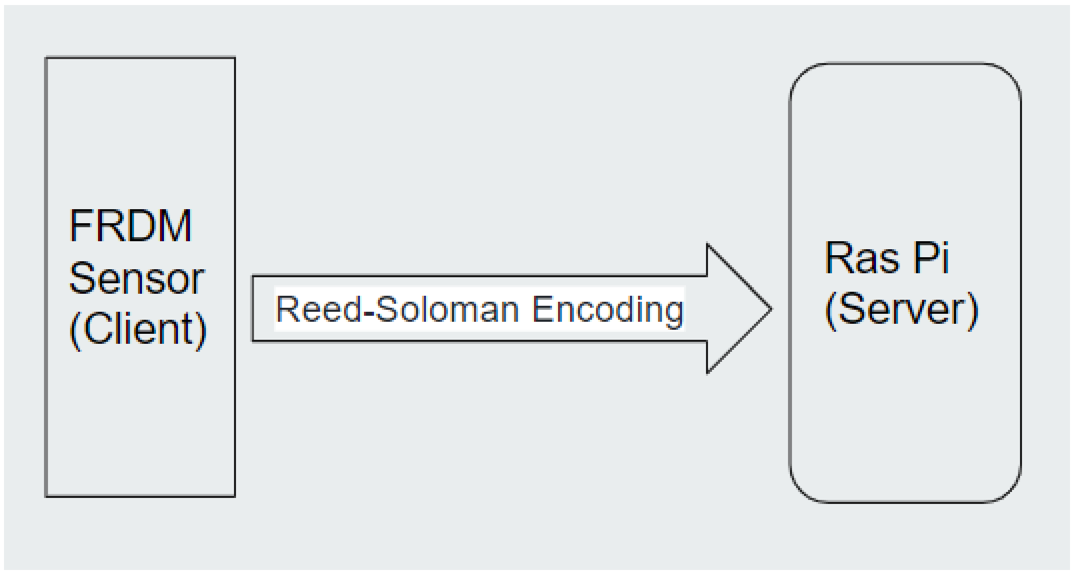
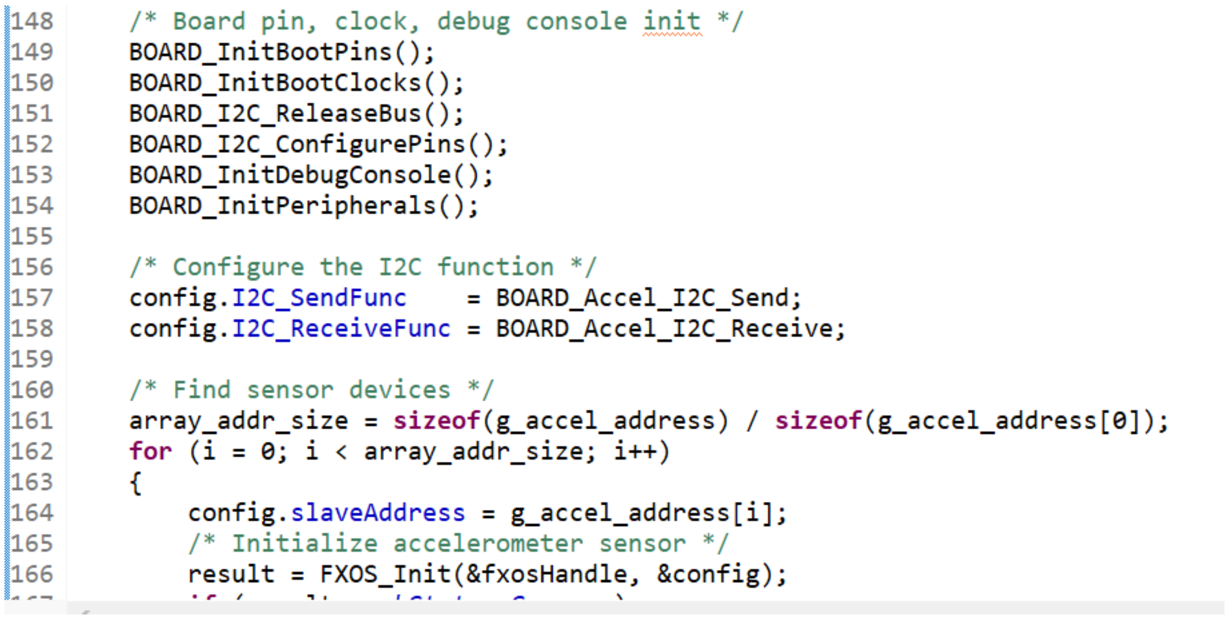


Figure 3. M2M Data Transmission

Sensor and Positioning Algorithm for Verification:

This module can be further divided into two submodules: Sensing data collection and positioning algorithm. We need to enable the sensor to detect movements and generate data. Then the generated data should be performed as the input to our positioning algorithm and information could be output to reflect the position of the user's neck. The positioning algorithm implemented on the client end is mainly for verification of the valid sensing data.

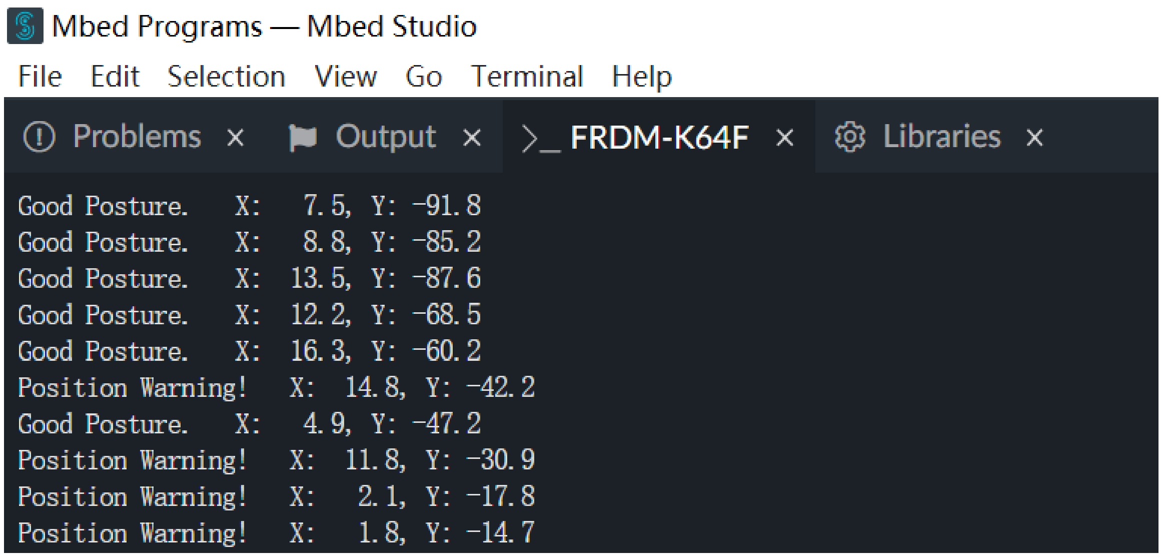


```
    /* Board pin, clock, debug console init */
    BOARD_InitBootPins();
    BOARD_InitBootClocks();
    BOARD_I2C_ReleaseBus();
    BOARD_I2C_ConfigurePins();
    BOARD_InitDebugConsole();
    BOARD_InitPeripherals();

    /* Configure the I2C function */
    config.I2C_SendFunc    = BOARD_Accel_I2C_Send;
    config.I2C_ReceiveFunc = BOARD_Accel_I2C_Receive;

    /* Find sensor devices */
    array_addr_size = sizeof(g_accel_address) / sizeof(g_accel_address[0]);
    for (i = 0; i < array_addr_size; i++)
    {
        config.slaveAddress = g_accel_address[i];
        /* Initialize accelerometer sensor */
        result = FXOS_Init(&fxosHandle, &config);
```

Figure 4. Sensor utilizes I2C to send and receive data

For sensing and data collection, FRDM-K64F-AGM04 provides a powerful GUI called Freedom Sensor Fusion Toolbox. Basically we need to set up the sensing environment inside the development board and select a sensor algorithm. To set up the hardware part, we use the USB cable to connect the development board to the PC. Then the FRDM-STBC-AGM04 sensor shield needs to be plugged onto the development board and device driver needs to be installed. There are several default algorithms aiming at different applications, e.g. 2D Automotive Compass, Rotation, Gaming Handset and so on. Here we want to implement the Tiltmeter Sensing Algorithm, which makes use of MMA8652FC, a 12-Bit Digital Accelerometer to provide an accurate orientation estimate even in the presence of high linear acceleration and the presence of an external magnetic disturbance. Therefore, the accelerometer is able to detect orientation on both

vertical planes. For the communication inside the board, the sensor utilizes an I2C bus to send and receive the data (see Figure 4) at an extreme rate. Also, the sensor algorithm utilizes a BOARD_TIMER_IRQHANDLER to provide a precise timer. These characteristics are pre-configured in the sensor library to be utilized. After the environment is set up, we fix the development board on the user's neck, and start collecting data. During orientation sensing, data will be collected from the two gyroscope axes measurements in units of degrees ranging from -90 to 90. It is displayed through the serial port at the baud rate of 9600.
Now we have real time streaming data shown on the terminal, we need to further convert it to intuitional information that reflects the people's positioning, and hopefully, gives feedback when people are keeping an unhealthy posture, e.g. a very small angle between the neck and horizontal axis. The positioning algorithm takes the real time data as input and sets up an angle threshold = 45 . It has a power loop which can be halted as needed to check the angle all the time. For example, by calling a wait_ns(10000) function inside the loop, we can set the data collection duration as 10000ns between every two times. Without beeper, the simplest way of showing some feedback is to output some texts. Therefore, corresponding messages will be printed when the angle on either x-axis or y-axis is below ("good posture") or beyond ("position warning") the threshold. We can verify if the sensor is working correctly by observing the angle data printed out inside Mbed Studio(see Figure 5).


Mbed Programs — Mbed Studio
File Edit Selection View Go Terminal Help
Problems ×
Output ×
>_ FRDM-K64F ×
Libraries ×

```
Good Posture.    X:   7.5,  Y: -91.8
Good Posture.    X:   8.8,  Y: -85.2
Good Posture.    X:  13.5,  Y: -87.6
Good Posture.    X:  12.2,  Y: -68.5
Good Posture.    X:  16.3,  Y: -60.2
Position Warning!    X:  14.8,  Y: -42.2
Good Posture.    X:   4.9,  Y: -47.2
Position Warning!    X:  11.8,  Y: -30.9
Position Warning!    X:   2.1,  Y: -17.8
Position Warning!    X:   1.8,  Y: -14.7
```

Figure 5. Sensing Verification on Mbed Studio Terminal

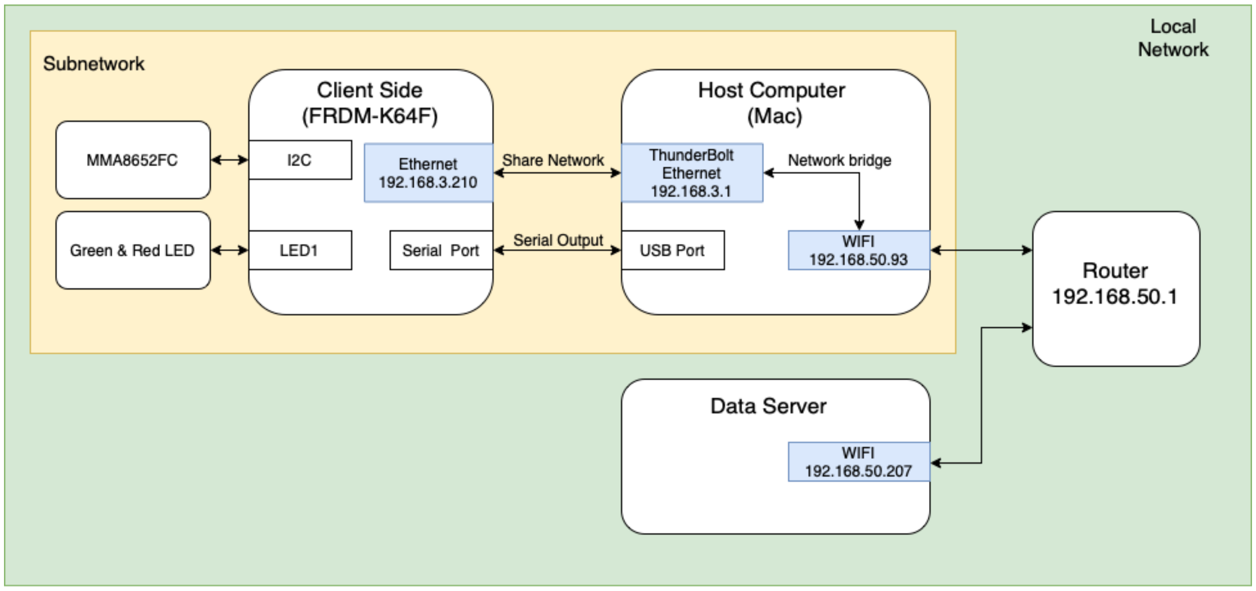


Figure 6. Demo Hardware Configuration

Message passing and cross device transmission

Since the sensor has been successfully configured, and position related data can be streamed out of the sensor, read by the main thread of the development board, it's time to consider the transmission of this data from the FRDM-K64F board to other devices. (like the Raspberry Pi, which has been configured as the data server in our design) FRDM-K64F board equipped with a RJ45 interface, and onboard MAC module to handle Ethernet connections. With the help of MBed OS libraries, we could easily initialize the Ethernet port, and establish the ethernet connection object. In this process, we need to provide 3 parameters to configure the Ethernet interface, namely, the IP address of this device, the netmask and the IP address of the host. In our Demo configuration, we choose to share the host computer's network (basically it's setting up a network bridge to Thunderbolt Interface) to hold the connection to the serial port for serial communication and the network connection for Network Socket at the same time. This configuration can be broken down to what shows in Figure 6, that the FRDM-K64F board is connected directly to the router. In this case, the configuration file in the MBed OS project should be modified according to the router's subnetwork environment.

Next step is to establish the TCPSocket for communication between two TCP/IP capable devices. In the demo, in order to verify the capabilities of our Reed-Solomon Error Correction Code, we established two different TCP Sockets with two different servers, as illustrated in Figure 7. Note that the number of these TCP Sockets can vary in the real use case. We've tested setting up 7 different TCP Sockets at the same time, and the board handles the connection smoothly.

Once we connect the FRDM board and the Servers on Raspberry Pi using the TCP Socket, our data can be transmitted through these sockets. On the client side, we push 256 bytes in one chunk to the socket, while the server side listens to the corresponding socket port and receives these 256 bytes.

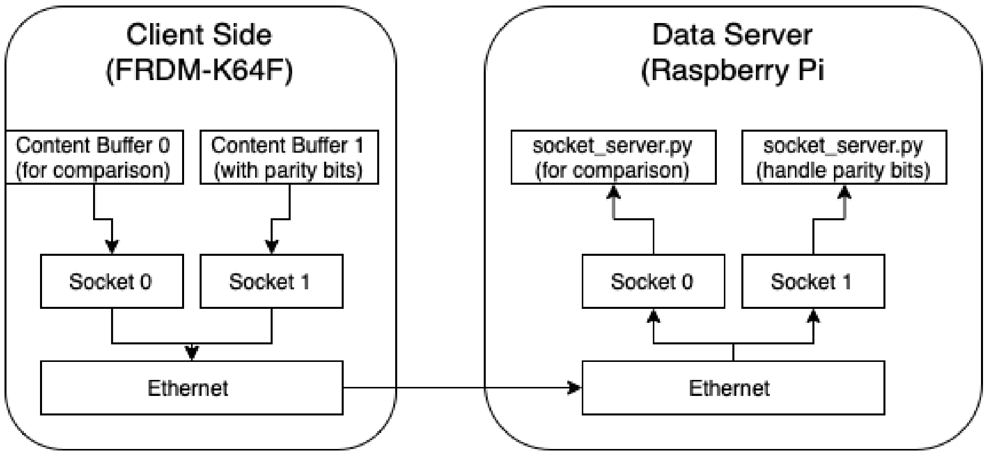


Figure 7. Inter-device Data Flow through the Network

Next problem would be the communication between the sensor and the socket transmitter. What makes the problem more complicated is that we add one Reed-Solomon encoder in between these two modules. For this specific encoding module, the most efficient length of input string should be less than, and also close to $Nmax - Nparity$ bytes. (We would explain this in detail in the next section)

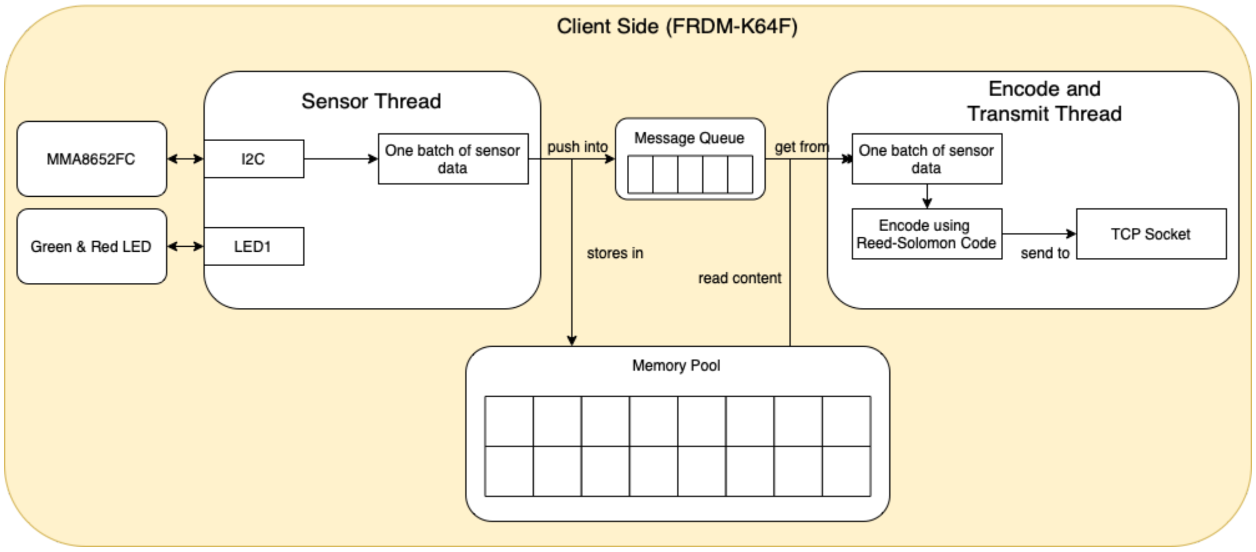


Figure 8. Inter-thread Communication

To solve this problem, we introduce the one message queue, along with a data structure to pass messages in between different threads in a thread-safe mechanism.

As the dataflow in Figure 8 shows, the streaming data can be collected through the I2C interface in a separate thread, then formatted into batches. Here one batch means an array of 14 strings, each string consists of 16 characters. This specific number of characters is designed to fully utilize the computational cycles of the Reed-Solomon Encoder. These batches are stored in a shared memory pool, and then would be inserted to the message queue and waiting for the consumption of the main thread. To increase the reliability of this message passing mechanism, and better handle extreme cases like TCP Socket Blocking, we increase the size of this memory pool to 16 times of one single batch. Meaning that in the worst case, this memory pool could tolerate multiple failures or major delays happening in the main thread while still securely saving the data collected from the sensor. On the other hand, the main thread would continuously listen on the message queue and retrieve data from the queue. It then concatenates all of these 14 strings to one big string, and passes it to the encoder to compute the parity bits. The encoder would generate 16 parity bytes and tail to the input string to form a total message with length of 240 bytes.

This message is regarded as the final string before transmitting to the server. One additional component is added as the header of the transmission message is what we called the “Control bits”. It’s simply the index of the message, counting from 0 when the embedded program is initialized. It’s been allocated to the leading 4 character space of the TCP Socket message. We’ll talk about its function in the next section.

Error Correction Code and Distributed Storage System

One distributed storage system should be defined as a storage system which contains multiple storage nodes or storage tanks for data storage. We introduce this concept to the project to illustrate the ability of one error correction code to retrieve the original data from partially provided data segments, in another word, in the case that one message has been evenly distributed to multiple data tanks, the existence of the parity bits could contribute to the recovery of the original data if some data tanks are dysfunctional. In our implementation, we fully utilized the ability of the Reed-Solomon algorithm. In the client side, as we introduced in the previous section, a 16-bytes-long code is computed based on the original 224-byte-long sensor data. This Reed-Solomon code is one error correction code, based on the finite field arithmetic. The code is “regarding one message as a polynomial following the very well-defined rules of finite field arithmetic”, instead of just a message as a series of numbers or ASCII character. As a result, the extra bits generated by the encoder would contribute to the recovery process of the missing bits, as long as the encoder and the decoder are sharing the same generator matrices.

In our implementation, we imported one well-tested c library as the foundation to build both the client-side encoder and the server-side decoder. We first generated the matrices used in the process of taking max 256 bytes as input and computing 16 bytes parity bits as output. These matrices were distributed to both client and server code as the essential index of the encoding and decoding process. Once the parity bytes were generated, it would follow the data flow explained in the previous section and be transmitted to the server side.

Next we discuss the implementation of the distributed system in detail. The distributed system we designed is the very entry-level system, which should be regarded as one poor simulation of distributed systems, instead of a well-defined storage infrastructure.

To simulate the nature of a distributed system, we use a separated data structure to store the input data of the server-side TCP Socket, namely, we distribute every 16 bytes to one table for storage, as implied in Figure 9, these tables are regarded as the "Content Tanks". One advantage of these distributions is that the order (or the index) of these strings are easily preserved, which is essential in the process of utilizing the parity bits generated by the Reed-Solomon algorithm. The parity bit is stored in a separate storage tank (in a table of a SQLite database), called the "Parity Tank". Since we only implemented a 16-byte-long parity segment, this distributed storage system could only tolerante one dysfunctional of the content tank. In another word, as long as the parity tank along with other 15 content tanks are still functioning, the original data is secured to the retrieved. Notice that this error correction system is using one extra 16-byte space to secure the data integrity of a total number of 16 character loss in the original data, which the combination of these characters can be random, either continus(as we introduced in this simulation) or sparsely distributed.

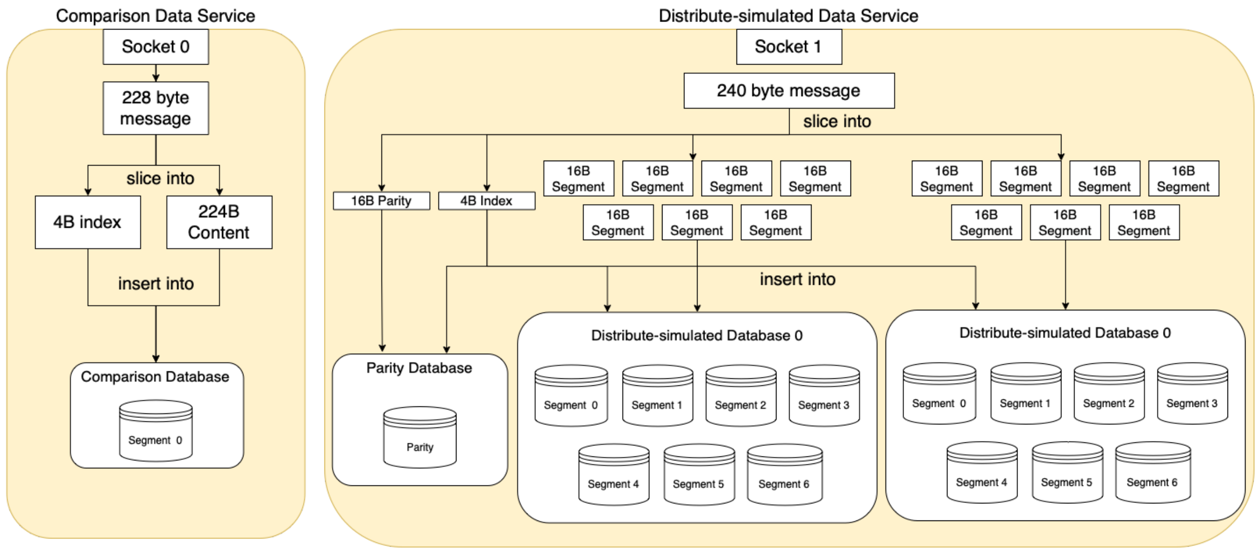
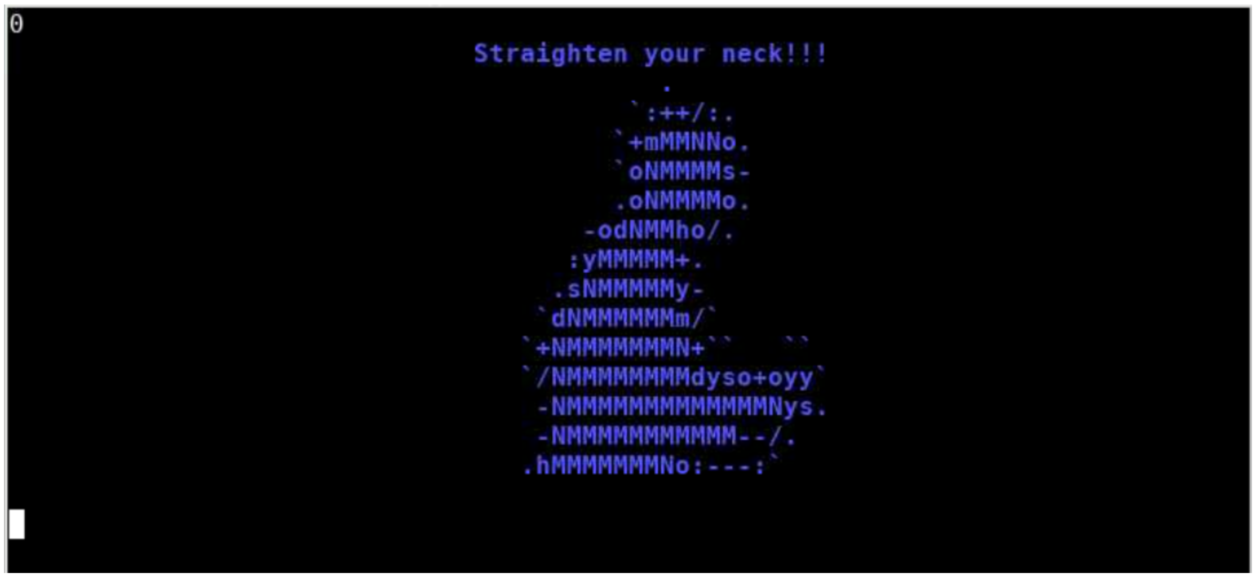


Figure 9 DataFlow inside Data Server (Raspberry Pi)

Server Command Line Feedback

After we have proved the secure and successful data transmission between the client (FRDM sensor) and the user end(Raspberry Pi), it is time to provide feedback based on the collected data. Here we consider server command line animation as a good and intuitive reminder to the user. Still we set a threshold=45 (which can be customized by the user) on the server end before the sensor is in use. If the neck's angle is larger than the threshold for a continuous 5 seconds, there will be an animation pop up on the command line interface indicating that the user is keeping a bad posture for too long that he/she needs to straighten up the neck. The animation consists of a series of frames that are predefined in a list, and will play consecutively when being invoked. There is a permanent loop that keeps examining the live streaming data. An example is shown on Figure 10, the complete visualization can be found in the demo.mp4.



```
0
                              Straighten your neck!!!
                                          .
                                        `:++/:.
                                       `+mMMNNo.
                                       `oNMMMMs-
                                       .oNMMMMo.
                                     -odNMMho/.
                                    :yMMMMM+.
                                   .sNMMMMMy-
                                  `dNMMMMMMm/`
                                 `+NMMMMMMMN+``    ``
                                 `/NMMMMMMMMdyso+oyy`
                                  -NMMMMMMMMMMMMMNys.
                                  -NMMMMMMMMMMM--/.
                                 .hMMMMMMMNo:---:`
```

Figure 10. User End Command Line Feedback

Summary:

Learning:

In this project, we have gained a better understanding of Embedded System Sensors and M2M communication. By implementing what we have learned from course modules, we successfully built a rudimentary healthcare embedded system and made a feasible encode & decode solution during M2M communication. It turns out that the sensor utilizes the I2C bus and interrupt handler for timer, and we can tune the streaming frequency by adding the delay timer without changing the sampling rate

inside the library. Also, mbed-os provides a powerful sensor library so that we can easily enable sensors and get data along each axis printed through the serial port.
For the communication part, we went through the whole process of establishing TCP Socket between clients and servers. We also investigated the inter-thread communication method, the message queue for coordinated, memory-sharing message passing, which is efficient and thread safe.
Additionally, one important use case of the error correction code has been discussed and implemented in this project. Light-weight databases are also one major component of this project.

Future Work:

From a product design perspective, we think the user interface has a lot of space to be improved. One simple example is that we would like to add an easy-perceived feedback function on the client (FRDM-K64F) by using a beeper. This requires an additional component since FRDM-K64F does not integrate any on-board speaker or sound generator. The beeper can serve as a warning sign on this wearable device that reminds the user to adjust his/her posture for health.
Also, on the server side, we are interested in providing other visualization instead of command line animation. For example, we will try to utilize the AWS platform to send warning messages to users that he/she can visualize on mobile devices.
During the M2M communication, we would also like to research and try crypto methods to secure the sensitive information transmission. In this way data will be transmitted as cipher texts and be decrypted on the server end.

References and Open Source Libraries:

FXOS8700CQ Library:
https://os.mbed.com/teams/NXP/code/FXOS8700Q/

Mbed Ethernet Sample:
https://github.com/ARMmbed/mbed-ethernet-sample-kone

Reed-Solomon Encoder Decoder:
https://github.com/relwin/Reed-Solomon-Encoder-Decoder-

Reed-Solomon codes for coders:
https://en.wikiversity.org/wiki/Reed–Solomon_codes_for_coders