

Authenticated and Secured Over-The-Air File Transfer in Internet of Things

Shaluo Wu



McGill

Department of Electrical & Computer Engineering
McGill University
Montréal, Québec, Canada

August 14, 2024

A thesis presented for the degree of Masters of Electrical Engineering

©2024 Shaluo Wu

Abstract

Over-The-Air (OTA) updates in the Internet of Things (IoT) ecosystem can present significant security risks, as corrupted or malicious firmware can cause device malfunctions or data breaches. To address these risks, we developed a robust and secure OTA Device Firmware Update (DFU) for embedded systems. Our design consists of three key components: the host processor, the wireless module, and the iOS application. The host processor is responsible for handling the heavy data flow from the incoming packets. We implemented interrupt mode to manage this load efficiently, reduce Central Processing Unit (CPU) overhead, and handle errors. We use timeouts and security protocols to address network latency and potential data loss. In the wireless module, we integrated a secured protocol, allowing seamless communication across iOS and other Bluetooth-enabled devices. The iOS application is implemented to perform application-level error and timeout management. We also use cryptographic algorithms to protect the integrity and confidentiality of transmitted data. By incorporating these features, we have created a robust OTA solution with a success rate of 95% and effective error handling techniques.

Abrégé

Les mises à jour Over-The-Air (OTA) dans l'écosystème Internet of Things (IoT) peuvent présenter des risques de sécurité significatifs, car des micrologiciels corrompus ou malveillants peuvent entraîner des dysfonctionnements des appareils ou des violations de données. Pour répondre à ces risques, nous avons développé une solution OTA robuste et sécurisée pour les systèmes embarqués. Notre conception se compose de trois composants clés : le processeur hôte, le module sans fil et l'application iOS. Le processeur hôte est responsable de la gestion du flux de données élevé provenant des paquets entrants. Nous avons mis en œuvre un mode d'interruption pour gérer cette charge efficacement, réduire la surcharge du CPU et gérer les erreurs. Nous utilisons des temporisations et des protocoles de sécurité pour faire face à la latence du réseau et à la perte potentielle de données. Dans le module sans fil, nous avons intégré un protocole sécurisé, permettant une communication fluide entre les appareils iOS et d'autres appareils compatibles Bluetooth. L'application iOS est mise en œuvre pour gérer les erreurs de haut niveau et les temporisations. Nous utilisons également des algorithmes cryptographiques pour protéger l'intégrité et la confidentialité des données transmises. En

intégrant finalement ces fonctionnalités, nous avons créé une solution OTA robuste avec un taux de réussite de 95% grâce aux techniques de gestion des erreurs.

Acknowledgements

I am sincerely grateful to my academic supervisor, Prof. Zeljko Zilic, whose guidance and support have been invaluable throughout my studies and research in the Master's program. His inspiration and technical expertise have played a crucial role in the completion of this thesis.

I would also like to extend my heartfelt thanks to my peers, Guanyi Heng, Yuxiang Ma, and Zice Tang, for their support and companionship during my research journey. Their assistance, encouragement, and moral support have been truly appreciated.

Additionally, I am grateful to my friend, Hang Zhang, for her moral support and understanding during challenging times.

Last but certainly not least, I am deeply thankful to my mother for her boundless love, kindness, and unwavering encouragement over the years.

To all those who have contributed to this thesis, I extend my sincere gratitude. Your support and encouragement have made a significant difference, and I am truly appreciative of your contributions.

Contents

| | | |
|----------|--|----------|
| 1 | Introduction | 1 |
| 1.1 | Motivation | 1 |
| 1.2 | Objectives | 2 |
| 1.3 | Structure of Thesis | 3 |
| 1.4 | Contribution of Authors | 3 |
| 2 | Background and Literature Review | 5 |
| 2.1 | Bluetooth | 5 |
| 2.1.1 | Protocol Stack | 6 |
| 2.1.2 | Bluetooth BR/EDR and Bluetooth LE Comparison | 7 |
| 2.2 | OTA DFU in IoT and Blockchain | 10 |
| 2.2.1 | OTA DFU with Blockchain | 12 |
| 2.3 | Cryptography in Wireless Communication | 14 |
| 2.3.1 | Cryptography in Bluetooth | 16 |

| | | |
|----------|--|-----------|
| 3 | Proposed Methodology | 18 |
| 3.1 | Bluetooth Module Design | 20 |
| 3.1.1 | Security Protocol | 20 |
| 3.1.2 | Implementation | 22 |
| 3.1.3 | UART Streaming Function | 24 |
| 3.1.4 | I/O Configurations | 25 |
| 3.2 | iOS Application Design | 30 |
| 3.2.1 | Choice of Framework | 33 |
| 3.2.2 | Implementation | 33 |
| 3.3 | Host MCU Design | 38 |
| 3.3.1 | I/O Configurations | 40 |
| 3.3.2 | Data Reception | 41 |
| 3.3.3 | OTA Subroutine | 46 |
| 3.3.4 | DFU Design for Bluetooth Module | 48 |
| 3.3.5 | DFU Design for Host MCU | 51 |
| 3.3.6 | Automation and Robustness Design | 52 |
| 4 | Results and Discussion | 55 |
| 4.1 | Results | 55 |
| 4.2 | Discussion | 58 |
| 4.2.1 | Baseline | 59 |

| | | |
|-------|--|----|
| 4.2.2 | Performance and Error Handling | 60 |
| 4.2.3 | Wireless Security | 62 |
| 4.3 | Conclusion | 64 |
| 4.3.1 | Future Work | 65 |

List of Acronyms

| | |
|-------------------------|--|
| AE | Authenticated Encryption. |
| BCoT | Blockchain of Things. |
| BLE | Bluetooth Low Energy. |
| Bluetooth BR/EDR | Bluetooth Enhanced Data Rate/Enhanced Data Rate. |
| CCA | Chosen-Ciphertext Attack. |
| CPU | Central Processing Unit. |
| DFU | Device Firmware Update. |
| DNS | Domain Name System. |
| EA | External Accessory. |
| EEPROM | Electrically Erasable Programmable Read-Only Memory. |
| IC | Integrated Circuit. |
| IoT | Internet of Things. |
| M2M | Machine-to-Machine. |
| MCU | Micro-Controller Unit. |

| | |
|---------------|--------------------------------|
| MITM | Man-In-The-Middle. |
| OSI | Open Systems Interconnection. |
| OTA | Over-The-Air. |
| RAM | Random-Access Memory. |
| RFCOMM | Radio Frequency Communication. |

List of Figures

| | | |
|------|---|----|
| 2.1 | Bluetooth Protocol Stack and OSI Network Model Comparison | 7 |
| 2.2 | Bluetooth Single Mode and Dual Mode | 8 |
| 2.3 | Bluetooth LE Network Topology | 9 |
| 3.1 | Embedded System Connection Overview | 19 |
| 3.2 | Security Protocol Procedure between MCU, mobile device and co-processor . | 21 |
| 3.3 | Bluetooth Module with Security Chip Connection Scheme | 28 |
| 3.4 | MCU with Security Chip Connection Scheme | 30 |
| 3.5 | iOS Application Default View | 31 |
| 3.6 | iOS Application Overview | 32 |
| 3.7 | EASessionController Class Diagram | 34 |
| 3.8 | uploadFile() Sequence Diagram | 36 |
| 3.9 | Host MCU Software Design Overview | 39 |
| 3.10 | A packet that contains two variables | 44 |

| | | |
|------|---|----|
| 3.11 | Interrupt Service Routine and OTA Subroutine | 45 |
| 3.12 | UML Diagram for OTA's Customized UART Buffer | 46 |
| 3.13 | Data Exchange between MCU and iOS device in OTA DFU | 47 |
| 3.14 | DFU Process with BT122 | 49 |
| 3.15 | Captured BGAPI Response Messages | 50 |
| 3.16 | DFU Process with STM32U585 | 52 |
| 4.1 | Screenshot from BGTool Console, Successful Reboot from New Firmware Image | 56 |
| 4.2 | Screenshot from BGTool Console, Reboot Process | 56 |
| 4.3 | Screenshot from iOS device, Update of Device Name | 57 |
| 4.4 | MCU with Red LED Indicating New Firmware Version | 57 |
| 4.5 | Screenshot from MCU Console, Reboot | 58 |

List of Tables

| | | |
|-----|--|----|
| 2.1 | Bluetooth BR/EDR and Bluetooth LE Comparison | 9 |
| 3.1 | BT122 GPIO and Pin Name Mapping | 27 |
| 3.2 | STM32U5A5 and BT122 Connection Details | 40 |
| 4.1 | Overview of Comparison with Baseline | 59 |
| 4.2 | Measured Performance of OTA DFU | 61 |
| 4.3 | Security Features of OTA DFU | 63 |

List of Acronyms

| | |
|-------------------------|--|
| AE | Authenticated Encryption. |
| BCoT | Blockchain of Things. |
| BLE | Bluetooth Low Energy. |
| Bluetooth BR/EDR | Bluetooth Enhanced Data Rate/Enhanced Data Rate. |
| CCA | Chosen-Ciphertext Attack. |
| CPU | Central Processing Unit. |
| DFU | Device Firmware Update. |
| DNS | Domain Name System. |
| EA | External Accessory. |
| EEPROM | Electrically Erasable Programmable Read-Only Memory. |
| IC | Integrated Circuit. |
| IoT | Internet of Things. |
| M2M | Machine-to-Machine. |
| MCU | Micro-Controller Unit. |

| | |
|---------------|--------------------------------|
| MITM | Man-In-The-Middle. |
| OSI | Open Systems Interconnection. |
| OTA | Over-The-Air. |
| RAM | Random-Access Memory. |
| RFCOMM | Radio Frequency Communication. |

Chapter 1

Introduction

1.1 Motivation

The rapid proliferation of IoT devices has created an increasing need for efficient and secure OTA firmware updates. OTA allows the contactless and swiftly updates of end nodes. However, OTA updates can introduce significant security risks, including data breaches and device malfunctions due to tampered or corrupted firmware. The problem is exacerbated by the diverse range of IoT devices and the inherent vulnerabilities in wireless communication.

Designing a robust OTA framework can reduce the risk of firmware-related vulnerabilities. Our work pivots from the host Micro-Controller Unit (MCU) design and expands on the existing Bluetooth Dual Mode modules to build an integrated embedded system for OTA. This research contributes to the development of Blockchain IoT by providing a comprehensive

approach to OTA security, and to the development of embedded systems by exploring data processing under restricted resources. The study's outcomes have the potential to improve user trust in IoT systems and promote the wider adoption of secure OTA update practices.

1.2 Objectives

In this thesis, we present the holistic construction of an embedded system for OTA. To ensure compatibility as well as security in transmission, we implement a transport layer protocol that provides authentication and trust for participating parties through the verification of protocols. To integrate such protocol into our system, we also design an iOS application that only communicates with hardware accessories with matching protocols. The majority of our design and testing effort is devoted to the host MCU, where we handle and process the data flow and perform security encryption and error handling. In the MCU, we implement the transport protocol header managing subroutine, which enables the embedded system to communicate with multiple platforms. There is also the OTA subroutine that processes the firmware payload, passed from the communication protocol subroutine. The OTA subroutine prepares the payload into chunks, saves it into local flash, and later uploads them to the target hardware. We tailor the upgrade process and handle the responses respectively for each component.

The main focus of our proposed solution is on the embedded side implementation of firmware updates. We aim to improve the performance and reliability of the system through

the integrated design among different parts.

1.3 Structure of Thesis

In this thesis, we propose an integrated embedded system design towards achieving an authenticated and secured OTA. We will introduce the background in Bluetooth communication, existing solutions for OTA and potential applications in a blockchain environment in Chapter 2. We will then present the design choices and methodology in Chapter 3. With each subsection being dedicated to one hardware component, these sections are structured based on the organization of the implementation. We discuss both the potential design choices and our solution. Chapter 3.1 covers the Bluetooth module-related design details, and Chapter 3.2 describes the iOS application, as part of our integrated solution. Chapter 3.3 expands on the host MCU designs, including the overall robustness and security design, as well as the interaction among different parts. In Chapter 4, we illustrate the results and performance evaluation of our system compared to existing OTA models. Finally, we conclude and provide future work in Chapter 5.

1.4 Contribution of Authors

This section declares that the work presented in this document was carried out by Shaluo Wu. The author built and integrated the wireless module, host MCU and user application

for this work. The framework of the host MCU program is derived from the work of Christos Cuning. This work also builds on Guanyi Heng's work on the communication protocol subroutine in the host MCU programming. The integration into the communication protocol, data processing, user end device programming, and target upgrade devices are carried out and tested by myself.

Chapter 2

Background and Literature Review

The subsequent chapter offers an in-depth examination of foundational definitions, concepts, and research inquiries essential for comprehending the discussion about our design.

The initial section provides a comparative analysis between Bluetooth Enhanced Data Rate/Enhanced Data Rate (Bluetooth BR/EDR) and Bluetooth Low Energy (BLE). The second section conducts a comprehensive review of extant literature pertaining to OTA DFU solutions in conventional IoT setup and Blockchain applications. Lastly, the third section elucidates cryptographic concepts including encryption schemes and levels of security.

2.1 Bluetooth

Bluetooth constitutes a wireless communication standard operating within the 2.4GHz frequency band. Its widespread adoption spans various consumer electronics and industrial

settings, particularly within the IoT domain. Bluetooth encompasses both the Bluetooth Classic radio, also known as Bluetooth Enhanced Data Rate/Enhanced Data Rate, with origins dating back to the 1990s, and Bluetooth Low Energy. While Bluetooth LE represents a more recent addition to the Bluetooth specification, it diverges from Bluetooth Classic across several dimensions. These differences extend from protocol stack structure and data rate to power consumption and application suitability. Bluetooth LE demonstrates enhanced power efficiency, making it suitable for applications with limited power resources.

2.1.1 Protocol Stack

The comparison between Bluetooth BR/ EDR and Bluetooth LE is best illustrated through an overview of their protocols. Bluetooth SIG has specified a Protocol Stack for developing interactive services and applications across various design procedures, similar to Open Systems Interconnection (OSI) for computer networking. Figure 2.1 shows, from a high level, how the Protocol Stack of Bluetooth BR/ EDR compares to that of Bluetooth LE.

As seen on the bottom of Figure 2.1, a Dual Mode device supports BR/ EDR and BLE connection at the same time. A Bluetooth 5.0 device, for instance, can support Bluetooth BR/EDR and Bluetooth LE connections concurrently, as illustrated in Figure 2.2. Bluetooth ensures backward compatibility, where devices with newer versions of Bluetooth,

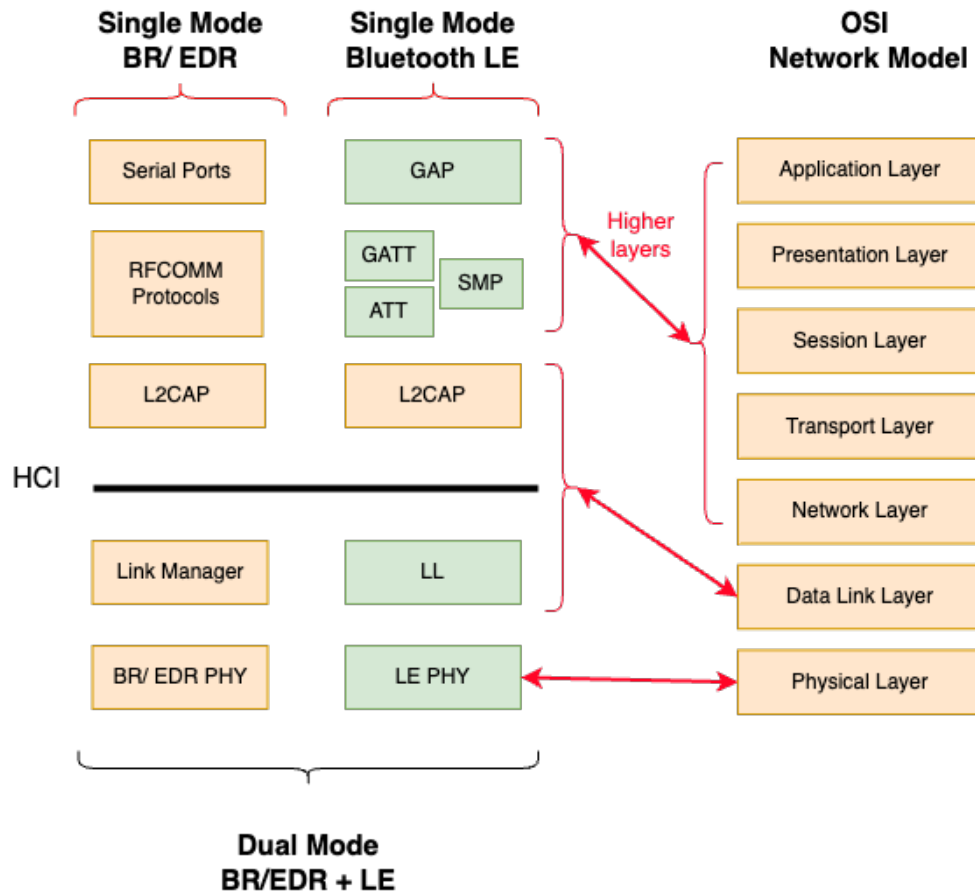


Figure 2.1: Bluetooth Protocol Stack and OSI Network Model Comparison

are compatible with older versions.

2.1.2 Bluetooth BR/EDR and Bluetooth LE Comparison

Bluetooth BR/EDR and Bluetooth LE exhibit significant differences, primarily due to their distinct applications. Bluetooth BR/EDR is tailored for high-throughput data transmission purposes, such as audio streaming, file transfer, and hands-free calling. Conversely,

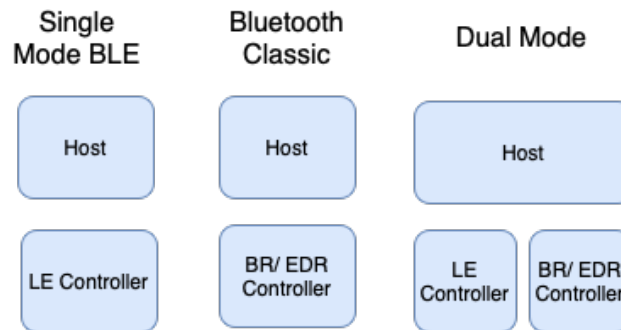


Figure 2.2: Bluetooth Single Mode and Dual Mode

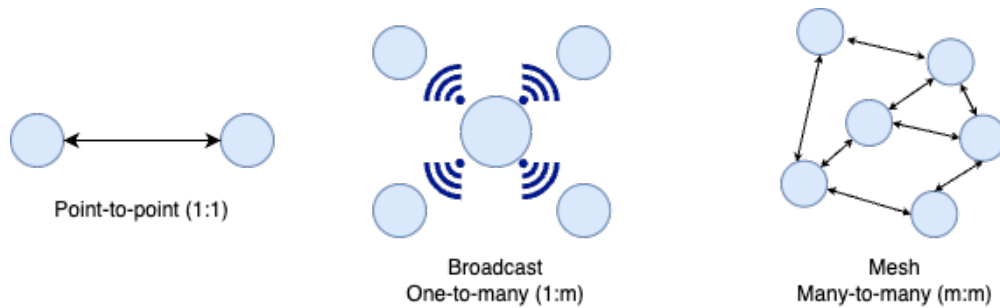
Bluetooth LE operates as a more streamlined protocol, prioritizing low-power consumption and intermittent data transmission applications. Bluetooth LE is widely seen in wearables, IoT devices and sensors. To fulfill different application requirements, they each have their own power consumption level, feasible communication range, connection scheme, and throughput. Table 2.1 contrasts Bluetooth BR/EDR and Bluetooth LE from a high level.

Bluetooth BR/EDR and Bluetooth LE also adopt different communication topologies. Bluetooth Classic supports point-to-point communication. Devices must exchange security keys to pair with each other and establish a secure connection securely. Bluetooth Low Energy, besides supporting connection-oriented communication, also supports broadcast and mesh topologies. That is, a central LE device can broadcast messages in a connection-less manner, as illustrated in Figure 2.3. Data broadcast, also referred to as BLE beacons, employs one-way communication, with sensors openly broadcasting their data to neighboring devices. This technology finds utility in low-accuracy location services, such as simple indoor navigation. On the other hand, mesh networking facilitates

| | Bluetooth BR/ EDR | Bluetooth LE |
|---------------------|---|---|
| Frequency Band | 2.4 GHz ISM Band (2.402 – 2.480 GHz Utilized) | |
| Data Rate | BR PHY: 1 Mb/s EDR PHY: 2~3 Mb/s | 1~2 Mb/s |
| Power Usage | 1 W | ~0.01x W to 0.5x W |
| Connection Topology | Point-to-point (1:1) | Point-to-point (1:1) Broadcast (1:m) Mesh (m:m) |
| Device Discovery | Inquiry or paging | Advertising |
| Application | Steaming audio Large file transfer Hands-free calling | Location and navigation Data transmission, medical and sports equipments Device network, automation and control systems |

Table 2.1: Bluetooth BR/EDR and Bluetooth LE Comparison

many-to-many (m:m) communication within larger-scale device networks. Mesh networking is used in complex large-scale device networks for monitoring, control, and automation systems.

**Figure 2.3:** Bluetooth LE Network Topology

2.2 OTA DFU in IoT and Blockchain

Over-The-Air Device Firmware Upgrade is crucial for maintaining and enhancing IoT devices, ensuring they remain up-to-date and secure. However, OTA DFU poses significant challenges, including ensuring secure data transmission, managing network latency, and handling potential interruptions. Addressing these challenges is vital for the reliability and security of the IoT ecosystem, making robust OTA DFU solutions essential for modern connected devices.

First, from the hardware level, IoT nodes have restricted access to processing power, as discussed in Arakadakis et al.'s survey paper [1], making it hard to apply security mechanisms. To work around that, Mughal et al. [2] propose to design more lightweight software while Banerjee et al.'s work [3] focuses on hardware-based cryptographic acceleration schemes. Low-cost microcontrollers may also struggle with limited memory capacity. The RAM cannot always accommodate the firmware image as well as the bootloader. Therefore, flash or EEPROM is often used for the storage of firmware images in DFU, as seen in Panta's [4] approach. The embedded device downloads the firmware to EEPROM and copies it to flash, such that the device remains operational during the downloading process. Energy consumption is another limiting factor because IoT nodes often operate unattended on batteries or ambient sources, such as solar power. Reijers et al. [5] finds that the transmission of data consumes approximately 1,000 times more energy than executing commands. OTA researchers should be aware of the energy consumption

level and improve the power efficiency of the system.

Moreover, there are application layer challenges arising from the deployment of IoT networks. During firmware upgrades, the embedded device uses bootloader to load the new firmware, reboot, and complete the upgrade. In Zhu's algorithm [6], where nodes function collaboratively for data routing, such overhead from rebooting results in catastrophic disruptions in the system. Zhang et al. [7] approaches this problem by code patching, where they add a jump instruction in the original code and direct the execution to the novel addition.

Also, security is paramount to the operations of IoT systems because an adversarial can compromise the IoT system on a large scale. For example, as mentioned by Antonakakis et al. [8], a large number of IoT devices are overwhelmed with denial-of-service attacks, and technical interventions, such as encryption and digital signature, should be applied to address these risks. In the meantime, Moran et al. [9] points out that, the security of the OTA process cannot rely on link layer, network layer, or transport layer mechanisms. That application layer encryption must be applied to address the security vulnerability. The work in Zhou et al. [10] also suggests that an OTA upgrade should be reliable for the hijacking, tampering, or replacing of the firmware file.

2.2.1 OTA DFU with Blockchain

With the rapid proliferation of IoT, the challenges of poor interoperability, network complexity, privacy and security vulnerabilities arise. IoT systems are by nature decentralized and of complex network structures. Dai et al. [11] point out that Blockchain technology offers promising solutions to these issues, leading to the integration of blockchain and IoT, referred to as Blockchain of Things (BCoT).

Integrating blockchain into the IoT environment offers several benefits. First, it ensures data traceability with historic timestamps for each block of data on the blockchain. It also brings reliability thanks to the cryptographic mechanisms like encryption, hash functions, and digital signatures in blockchains. It also allows the automatic execution of commands, such that no human intervention is required as long as the conditions are met. The implementation of smart contracts in blockchain makes these benefits possible in BCoT. They are contractual clauses that get enforced automatically if a condition is satisfied. Smart contracts containing information about the upgrade target, firmware hash, and scheduling are deployed across the network. Christidis et al. [12] uses smart contracts as a secure automatic firmware upgrading solution. In BCoT, each contract statement, such as a wireless upgrade target, is recorded as immutable transactions stored in the blockchain. Without human intervention, the distributed nodes can perform an upgrade promptly. As a result, the security maintenance cost of the system is saved.

The advantages of BCoT also expand to the interoperability among different IoT

systems, according to Wan et al. [13]. In particular, in the manufacturing industry, by associating each part with a unique ID and immutable timestamp, the identification information is stored in a blockchain, making it tamper-resistant and traceable. Also, blockchain's inherently decentralized nature brings the extra advantage of mitigating the demand for high performance at a central agency. For example, as discussed by Dai et al. [11], the validation in blockchain operates without the need for a trusted third party and saves computation costs.

In the meantime, besides IoT, blockchain is also suggested to be a good complement to the Domain Name System (DNS), as discussed in Li et al.'s work [14]. The DNS is the phonebook of the Internet, mapping alphabetic domain names to numerical IP addresses. The DNS is managed by a root server, which means that it is subject to single-point failure of name servers. If the name server of a critical domain is under attack, the sub-domain services are affected, too. Also, the cache poisoning attack is another threat to the current DNS. The adversarial party can forge a DNS entry to inject it into the cache and clients will be redirected to the phishing website. In response to these challenges, Li et al. [14] and Liu [15] discuss a novel model, combining blockchain and DNS. That is, to immutably store data in the blockchain, which is not managed by a central unit. Once a domain name is registered on the blockchain, it is tamper-proof and resistant to unauthorized changes. Li et al.'s approach [14] uses Proof of Stake as the blockchain consensus to ensure the consistency of DNS records. A registry holds a probability proportional to its stake, in

this case, the number of domains registered, to be elected as the block generator. This design is empirically shown to be present a lower attack success rate and higher attack cost against malicious attacks than legacy DNS. Thus, blockchain-based DNS offers a promising alternative to traditional DNS, with enhanced security, transparency, and decentralization.

From the scope of embedded system realization, the requirements posed by blockchain applications are analogous to those of wireless DFU. They are both in essence, the secure, authenticated, validated delivery of blocks of data. BCoT, just like wireless DFU, demands data integrity and confidentiality from the transport layer, despite application layer security designs. By building a reliable and secure network for scalable file transmission, it can be generalized toward the realization of Blockchain in a resource-limited IoT environment.

2.3 Cryptography in Wireless Communication

In wireless data transmission, cryptographic schemes play a crucial role in ensuring secure and intact delivery. Adversarial parties, with varying levels of capability, access, and objectives, necessitate different cryptographic systems to protect against threats. When evaluating security levels in embedded system engineering, it is essential to consider computing power limitations while safeguarding wireless connections. The following section aims to examine the various commonly-seen cryptographic schemes in the IoT environment.

Adversarial parties can be categorized into eavesdroppers and active malicious attackers. Eavesdroppers passively gather information from communication channels

without initiating attacks. To thwart eavesdropping, encrypting the payload with an encryption key before wireless transmission is imperative. Active attackers employ strategies such as brute-force attacks, Man-In-The-Middle (MITM) attacks, replay attacks, chosen-plaintext attacks, chosen-ciphertext attacks, and collision attacks.

MITM attacks involve adversaries clandestinely altering communication to compromise transmission integrity. Protecting against MITM attacks requires sender identity verification and message integrity checks at each receiver. Digital certificates facilitate mutual authentication, where parties possess certificates signed by trusted certificate authorities. Collision attacks exploit the fact that different plaintexts can produce the same hash output. Employing hash functions with higher computational security mitigates this risk. Brute-force attacks involve adversaries systematically guessing cryptographic keys by trying all possible combinations. Modern encryption schemes are typically computationally infeasible to break by brute force. Chosen-plaintext attacks occur when adversaries decrypt arbitrary plaintext and analyze corresponding ciphertext to infer encryption algorithms. Chosen-ciphertext attacks enable adversaries to submit arbitrary ciphertext for decryption and deduce encryption algorithm structures. In CCA, adversaries possess broader capabilities than in CPA attacks. CCA-secure wireless connections are preferred in IoT environments.

Authenticated Encryption (AE) and public key encryption schemes are two types of algorithms providing CCA security. AES-CCM, combining counter-mode encryption with

CBC-based message authentication, exemplifies a CCA-secure encryption scheme. In public key encryption schemes, a public key, derived from a private key, is used for encryption, while the private key is used for decryption.

2.3.1 Cryptography in Bluetooth

The inherent vulnerabilities of Bluetooth BR/EDR communication expose the system to the risk of interception, eavesdropping, and MITM attacks. To mitigate these risks, Moran et al. [9] argues that it is crucial to implement robust security measures at the application layer.

Firstly, encrypting the payload at the application layer protects the data from being intercepted by unauthorized parties. This can be achieved by using advanced encryption algorithms such as AES. AES encryption ensures that even if the data packets are intercepted during transmission, the contents remain unintelligible to the attacker. The encryption process involves converting the plaintext data into ciphertext using a secret key, which can only be decrypted by authorized parties who possess the corresponding decryption key.

Secondly, ensuring data integrity is equally important to prevent any unauthorized modification of the transmitted data. This can be achieved by using cryptographic hash functions like SHA-256. A hash function generates a fixed-size hash value from the input data, which acts as a digital signature of the data. Before transmission, the sender can compute the hash value of the payload and send it along with the encrypted data. Upon receiving the data, the receiver can recompute the hash value of the decrypted payload and

compare it with the received hash value. If the hash values match, it confirms that the data has not been altered during transmission. A hash function should be computationally collision-resistant, that is, no two distinct plaintexts can produce the same ciphertext, such that the adversarial cannot replace the data with malicious input that has the same hash value.

In summary, by encrypting the payload at the application layer and using cryptographic hash functions to ensure data integrity, we can significantly enhance the security of OTA DFU over Bluetooth. These measures help improve the system's security by encryption, digital signature, and authentication.

Chapter 3

Proposed Methodology

This chapter provides an in-depth examination of the hardware setup and software structure necessary for achieving a secure over-the-air upgrade across various components. This project aims to perform OTA DFU for hardware accessories through a mobile device. The upgrade target can either be the microprocessor or the wireless module.

Figure 3.1 abstracts the system connection and component responsibilities. The central unit of the embedded system is the host MCU. Most functionalities of our design are achieved through there. We also program the wireless module and the security module, connected to the host MCU. The mobile phone application handles user-level interactions and provides the source file for the embedded system to upgrade with.

The microprocessor is connected to the iOS device through Bluetooth BR/ EDR, and it is recognized as an External Accessory. We also employ a security authentication co-processor

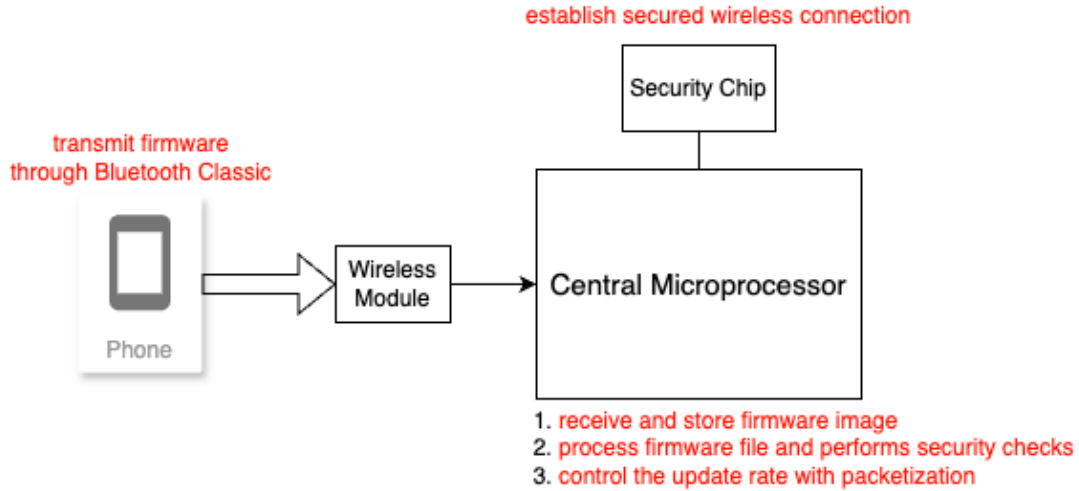


Figure 3.1: Embedded System Connection Overview

that handles the authentication, certification and encryption of data. The co-processor is connected to the host MCU, maintaining the flexibility to communicate with both devices that support the security protocol and those that do not.

We design an iOS application that provides the firmware image and performs version control. The Bluetooth module directs all Bluetooth Classic communication to the main processor. The host MCU processes all the incoming packets, downloads and stores them in the flash, and decides where to direct the firmware file. Towards building an automated and robust OTA DFU system, we design the architecture such that the user can monitor and control the firmware version, update process, and update target while only interacting with the iOS application.

3.1 Bluetooth Module Design

In our experimental setup, we use Bluetooth Classic as our communication band and BT122 is our choice of wireless module. BT122 runs on two modes, data mode and BGAPI mode. Mostly, we use BT122 data mode and define its behaviors in the .bgs file. BGAPI mode is where BT122 is controlled by a series of tailor-made protocols over the UART interface, known as BGAPI commands. They are binary serial protocol commands sent to the Silicon Labs Bluetooth Dual Mode stack over UART interface. BGAPI is designed for users to take advantages of all the features that dual mode offers from an external host in real-time, instead of a pre-compiled software. One can program the BT122 to control which mode it is running in. We set it in data mode by default to perform Bluetooth connection management and streaming. If the OTA DFU target is BT122, we then switch it into BGAPI mode, where behaviors defined in the script are paused and the module only listens to UART for BGAPI commands.

3.1.1 Security Protocol

To enhance the overall security of our IoT system, we have integrated a robust security scheme into our Bluetooth network, building upon Ravdeep's work [16]. This system enables password-free access over a secured link, significantly speeding up connections compared to manual authentication and facilitating secure Machine-to-Machine (M2M) interactions essential for IoT applications.

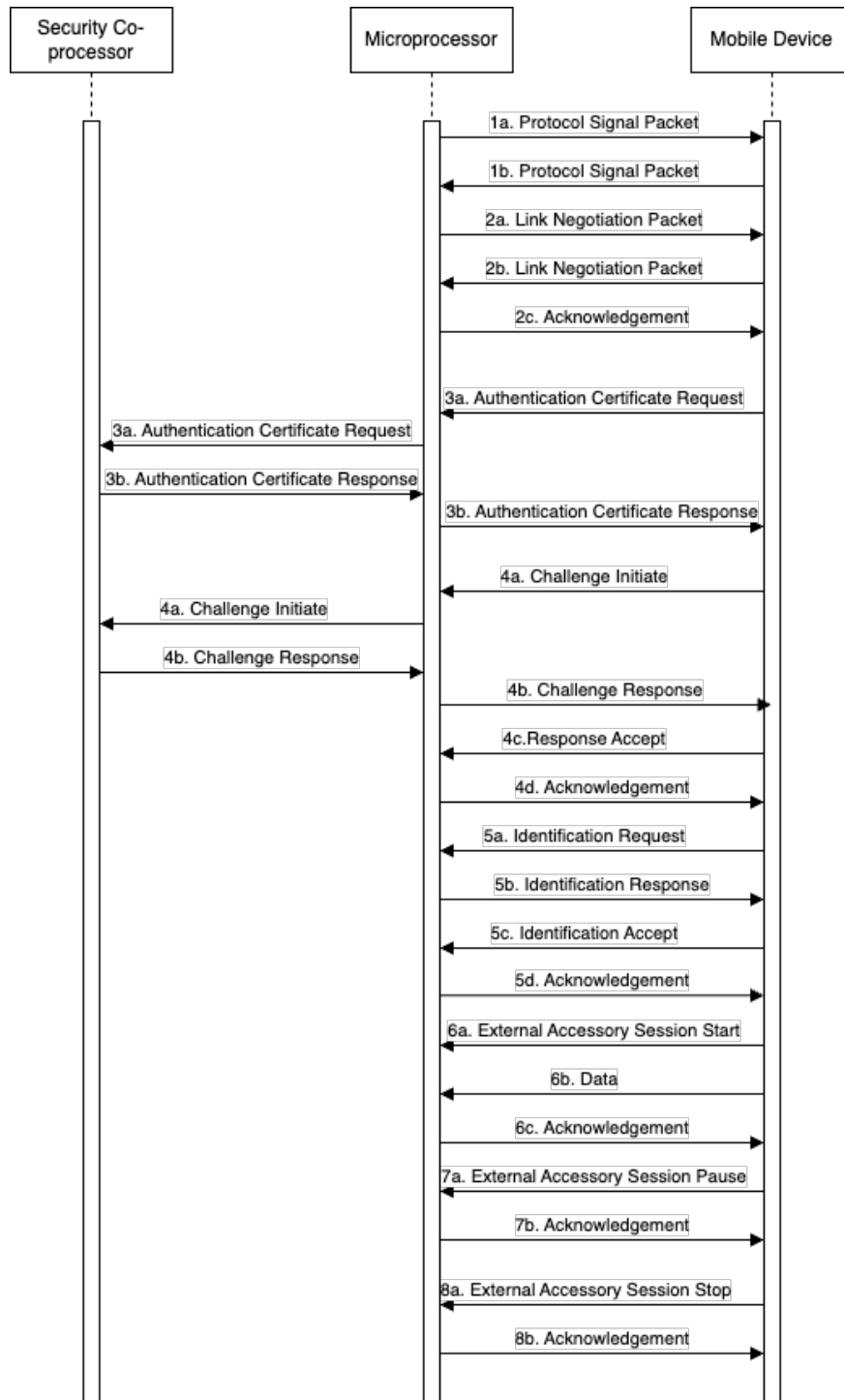


Figure 3.2: Security Protocol Procedure between MCU, mobile device and co-processor

The system enhances security by ensuring that sensitive authentication elements such as keys, certificates, and challenge responses remain invisible to users and are exchanged using strong, irreversible hashing algorithms. Devices equipped with the correct certificate and capable of solving the challenge can securely connect to the network. The system enforces authentication protocols, includes error detection, and handles multiple devices based on operating systems, Android, iOS, or Windows. The authentication process involves exchanging certificates and challenge/response pairs, which are securely stored and computed in an external security coprocessor. If necessary, the system performs encryption to secure communications further.

The authentication and encryption modules are hardened to ensure their invisibility to the user. In the embedded system realization, we connect the security Integrated Circuit (IC) to the embedded system and design the MCU's behaviour accordingly. Flowchart 3.2 illustrates the protocol communication among the co-processor, host MCU, and user-end device. We implement the MCU to handle the link session, identification establishment, and external accessory session. The authentication and challenge requests are forwarded to the co-processor for generation.

3.1.2 Implementation

The Bluetooth module, BT122's application includes .xml files that configure the hardware and communication-level details of the wireless module. The .bgs script defines the module's

behaviors and can be omitted if no complex behavior designs are required. Every BT122 project begins with a `project.xml` file, which instructs the compiler on all the files in the application.

- **.bgs** file uses BGScript, a programming language developed by Silicon Labs, to drive BT122 when it is in data mode. BGScript provides access to the same APIs as BGAPI but it hides the complexity of low level protocol designs to facilitate application development. It is an event-based programming language. Code execution is triggered by event callbacks. An *event* is a callback, and a *procedure* is a custom-defined function that can be invoked in an event. BGScript application can be used simultaneously with BGAPI commands.
- **project.xml** file instructs the compiler to include and identify each `.xml` and `.bgs` file.
- **gatt.xml** contains GATT profiles. GATT refers to Bluetooth Generic Attribute Profile in Bluetooth Low Energy. GATT forms the foundation for most BLE applications and services. It uses the attribute protocol, which contains a lookup table for 16-bit IDs. These IDs define the services that a BLE device can offer.
- **spp.xml** configures Serial Port Profile for Bluetooth BR/ EDR. It includes a list of profiles that enables BT122 to emulate a serial port and facilitate data transfer. A project can have both the GATT profile and SPP, because BT122 supports dual mode.
- **hardware.xml** sets all the default configurations of different interfaces, such as I2C,

UART, and GPIO. This file also defines if the module is in data mode or BGAPI mode by default.

3.1.3 UART Streaming Function

BT122, as a wireless module, is connected to the mobile device through Bluetooth Classic and connected to the STM32 device through UART and GPIO pins. The fundamental task of a wireless module is to stream between endpoints, that is, direct whatever is received in one port to the other.

`endpoint_set_streaming_destination(0,endpoint)` and

`endpoint_set_streaming_destination(endpoint,0)` describes the bidirectional streaming behaviour of the Bluetooth module upon the opening of the RFCOMM port. The 0 input represents the UART port, as defined in BT122 API. The endpoint refers to the RFCOMM endpoint triggering the event, in our case, the mobile device.

In `hardware.xml`, we configure UART with baud rate 115,200. This has to comply with that of the host MCU, STM32U5 setup. Flow control is turned on for the reliability of large file transfer. BGAPI mode is low by default and will be turned on as required.

3.1.4 I/O Configurations

GPIO Configuration

GPIO is an interactive portal that lets us override settings and configurations during the execution of a program. It is also a simplistic way of obtaining transparency for debugging purposes.

To switch BT122's mode in response to the program's requirements, the following function handles the GPIO related features. When performing OTA for BT122, BT122 needs to be in script mode by default to act as a routing Bluetooth module. If we want to update BT122 itself, it should later switch to BGAPI mode to accept BGAPI commands and perform DFU, after the firmware is downloaded and checked.

Algorithm 1 UART Modes Toggling Function

```

1: function HARDWAREINTERRUPTHANDLING(interruptState,timestamp)
2:   if interruptState then
3:     if device is in script mode then
4:       Toggle to BGAPI mode
5:       Turn on LED to indicate BGAPI mode
6:     else
7:       Toggle to script mode
8:       Turn off LED to indicate script mode

```

In the `hardware_interrupt` event, when interrupt is raised, host MCU checks if the hardware is in BGAPI mode. If not, we switch BT122 back to BGAPI mode and turn on LED. If it is in BGAPI mode, then we turn off the LED as a debugging indicator. We use a GPIO input port on BT122 Bluetooth module to detect interrupt signals from STM32U5

and set LED as GPIO output. LED is used in several occasions throughout the project. By toggling LED, the user can easily tell whether the board is in script mode or BGAPI mode. We can also tweak the default LED states to indicate different firmware versions. For example, the older version sets LED0 OFF by default and the newer version sets LED0 ON. By observing the LED, one can track the OTA progress.

In line `hardware_write_gpio(2, 0x0008, 0x0008)`, the first input argument in the port index. According to BT122 API Reference, PA pins are of port 0, PC pins are at port 1, and PF pins are port 2. The push buttons and LED's are all wired to the PF pins, therefore the first input argument should be 2. There are two user push buttons on BT122. They are connected to the two yellow LED's, respectively, via a driver transistor. Table 3.1 shows the mapping of these GPIO pins. The second argument represents the target of the command and the third argument is the command data. Both input arguments follow a bit-mask logic, where 0008 in hexadecimal should be interpreted as 1000 in binary. This means that only the pin represented by the most significant bit, PF3 and LED1, is configured in this command. The third argument 0x0008 sets PF3 in high state. The state of the LED is reversed of the pin state, so LED1 is turned off.

The default states of I/O are configured in `hardware.xml`. The following code depicts the interrupt settings of a GPIO input port. `interrupts_falling` is set to 0x0004. This configures the PF2 pin for falling edge interrupt detection.

```
1 #Configures PF2 (BUTTON1) as inputs with falling interrupts enabled
```

Table 3.1: BT122 GPIO and Pin Name Mapping

| | | | |
|-----|---------|------|--------|
| PF2 | BUTTON0 | LED0 | 0x0004 |
| PF3 | BUTTON1 | LED1 | 0x0008 |

```

2 <port index="2" input="0x0004" pullup="0x0004"
3     interrupts_falling="0x0004"/>
4 #Configures PF3 led0 as output.
5 <port index="2" output="0x0008"/>

```

UART Configuration

For debugging purposes, we also configure a console output for additional transparency. This is implemented in the script file as, `endpoint_send(0, 7, "message")`. 0 is the UART interface and 7 is the length of the message. Besides using UART, it is also possible to use a UART adapter for monitoring. The adapter's RX is connected in parallel with the source of communication. It can be plugged into a USB port to be viewed on a terminal.

Configuration 1: Authentication IC Connecting to Bluetooth Module

During different stages of our design, there are two configurations that we constructed. In the initial setup, the authentication chip is connected to the Bluetooth module and all traffic, regardless of the target device, is encrypted. To enhance the overall customizability, we later redesign the system and move the authentication-related configurations to the host MCU. This current setup brings more flexibility into the system, such that we can distinguish

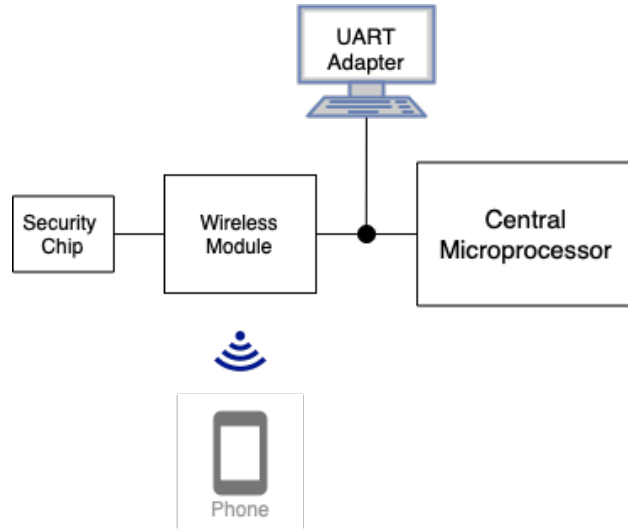


Figure 3.3: Bluetooth Module with Security Chip Connection Scheme

the device and choose to either use or not use the authentication protocol. This section expands on the configuration details of the initial setup. The Bluetooth module handles the encryption, headers, and checksum related to the security protocol.

To configure the Bluetooth module for security link support, we derive from the sample projects provided by Silicon Labs. By adding the protocol-specific UUID in the configuration file, the Bluetooth module queries for the library support and handles the subsequent communication. In the script file, we set all the Bluetooth friendly parameters, namely the device name, supplier name, device version, application name, protocol name, and product plan. While the other fields are rather customizable, the protocol name should agree with that defined in the iOS app. After setting the parameters, the script checks for the connection and prints to the console whether it is successful.

It is also important to utilize the correct version of the compiler software such that it provides library support for the security protocol to build the project. The authentication coprocessor is connected to Bluetooth module through I2C. Figure 3.3 depicts the schematics of system connection from a high level. With this setup, the Bluetooth module is responsible for establishing the connection, trimming headers for incoming packets and prepending those for outgoing communication. From the host MCU point of view, it receives the payload without any metadata. We can use a UART adapter to sniff the communication between the Bluetooth module and the host MCU, for debugging purposes.

Configuration 2: Authentication IC Connecting to MCU

The former setup has compatibility limitations. For example, in the TX configuration, all outgoing packets conform to the security protocol, without distinguishing between devices that support the protocol and those that do not. To improve the generalization, we connect the authentication chip to the host board. From there, we perform data encapsulation for all packets. Figure 3.4 shows the physical connection in this scheme. This setup retains the flexibility to communicate with any mobile device via Bluetooth BR/EDR or Bluetooth LE.

The Bluetooth module performs a slightly different role, as it is not responsible for protocol headers anymore. In its implementation, it performs only one function, streaming data from one port to another. The wireless module is connected to the mobile device through Bluetooth, and connected to the host MCU through physical UART pins. It streams

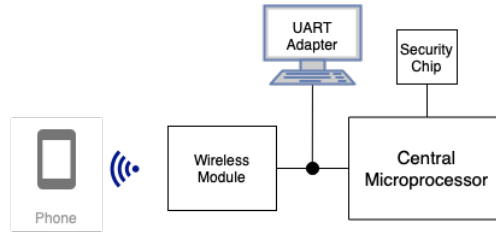


Figure 3.4: MCU with Security Chip Connection Scheme

all communication received in one port and redirects it to another one. In this setting, we centralize all the design and implementation in the host board.

3.2 iOS Application Design

The iOS application is the conduit for provisioning firmware updates to the microprocessors. Programming with Swift, we design an interactive application that interfaces with Bluetooth device under the security protocol. The application has two primary functionalities: session management and file transfer. To manage the connection sessions, we define our own utility module `EASessionController`. This module abstracts away low-level connection operations. We also implemented `OTAManager`, which facilitates efficient and reliable transfer of firmware images over the air.

To initiate OTA updates using the application, users must first ensure that their device is connected to the microprocessor via the System Settings menu. Then, in the app, depicted in Figure 3.5, users can initiate the OTA session by clicking the **Refresh** button. It prompts the display of a drop-down menu containing all active Bluetooth connections. The user then

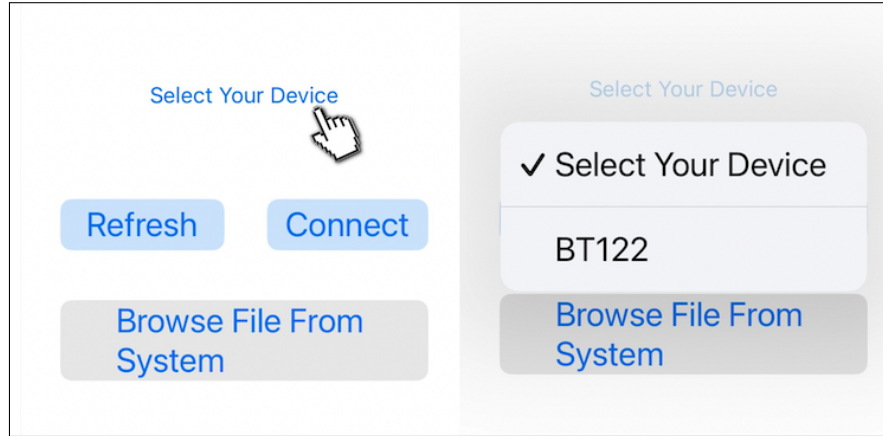


Figure 3.5: iOS Application Default View

selects the relevant hardware from the list and initiates the EASession by clicking **Connect**. After that, if no commands are received within 2 minutes, the EASession automatically terminates. Additionally, minimizing the application also results in the cessation of the EASession. If users wish to resume the session, they can utilize the drop-down menu to re-establish the connection. Upon state transitions, such as connection establishment or termination, the application sends a message, to indicate either the start or termination of a session, to the Bluetooth module. This functionality enables users to monitor session status via the receiving side console output.

Users can proceed to select the firmware image from the System Files. Detailed instructions for generating these firmware images are provided in Chapter 3.3.4 and 3.3.5. Users can only select binary files and other file types are displayed in grey. Upon selecting the firmware image, the OTA process commences, and users have the option to halt it by clicking **Stop** to terminate the session.

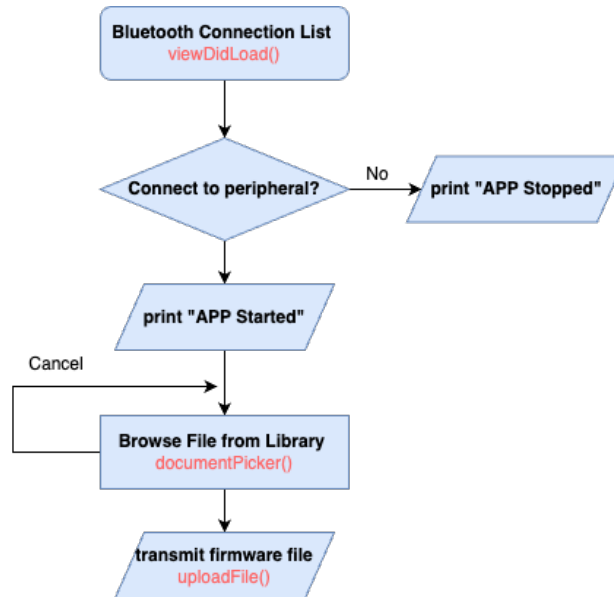


Figure 3.6: iOS Application Overview

Flowchart 3.6 provides a high-level overview of the program. It loads the list of current Bluetooth connections in `viewDidLoad()`. `viewDidLoad()`, which sets up the default user interface display, loads the list of current Bluetooth connections. From there, the program can either connect or disconnect with the accessory depending on user action. After the connection is established, the user clicks on **Browse file from Library** button, which invokes the `documentPicker()` to select valid inputs. The user can choose to send a firmware image or cancel in the file browser window. If the user picks a file, it is sent to the Bluetooth peripheral using `uploadFile()`.

3.2.1 Choice of Framework

In Swift, developers have access to two primary Bluetooth communication frameworks: Core Bluetooth and External Accessory (EA). Core Bluetooth is tailored for facilitating communication between Apple devices and Bluetooth LE peripherals, whereas External Accessory is designed to handle communication with Bluetooth Classic devices. As discussed in Chapter 2.1.2, Bluetooth LE supports different applications than Bluetooth BR/EDR, focusing primarily on intermittent data exchanges. In order to transmit large file data, in our case, a firmware image, we use Bluetooth BR/EDR, hence the EA framework.

The EA framework treats all connected devices as hardware accessories. Such device can communicate with an Apple device either through a physical connection or wirelessly through Bluetooth Classic. It is crucial for the hardware to be configured with the same protocol as the application, as only applications employing matching protocols can communicate with the external accessory. A single mobile device can run multiple applications simultaneously, each using distinct protocols, and uniquely pairing to multiple hardware accessories. For the iOS application, the communication protocol is defined in the `info.plist`.

3.2.2 Implementation

`info.plist`

The `info.plist` is a property list file that contains essential configuration information about the application in Swift development. We define the supported external accessory protocols as

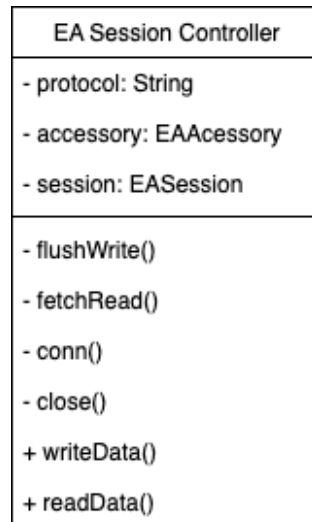


Figure 3.7: EASessionController Class Diagram

`com.bluegiga.iwrap`, following the reverse domain name notation. This protocol is provided by SiliconLabs, the manufacturer of the module. This entry should be identical to the SPP setup in the Bluetooth module, in order to establish a secure connection.

EASession Controller

This class encapsulates our custom helper functions, as illustrated in Class Diagram 3.7. The `EASessionController` Class contains objects such as the TX and RX data, protocol name, accessory, and session. It automates the process of establishing and termination of a connection. The `conn()` and `close()` functions handle input and output buffers and are invoked within the `ViewController` when initiating or concluding message transmission.

Also, `EASessionController` hides the low-level communication tasks from the user interface and provides a set of simplified functions to interact with. The private function

`flushWrite()` ensures the availability of the output stream manages the idleness of the write storage buffer. The user can directly use the public function `writeData()` to transmit target data, not having to worry about the buffer state. This abstraction enhances code readability because we centralize the handling fundamental tasks and ensure consistency across the application.

OTAManager

The OTAManager module oversees the transmission of large firmware files from a high-level. It transmits the firmware images in small chunks, and performs flow control on the transmission. It is designed in accordance to the behaviour of the microcontroller. This class comprises two functions, `uploadFile()` and `writeFileDataToAcc()`.

To transmit a large file, we employ a packetization approach, dividing the file into smaller packets. Each packet is sized to match one page in the STM32U5 flash, 8192 bytes. This design choice aims to facilitate the downloading process on the accessory's end. The host MCU should write to the flash page by page. The `writeFileDataToAcc()` function is responsible for packetizing the payload and awaiting receipts from the accessory after each transmission. This design significantly enhances the reliability performance of the transmission. Despite that Bluetooth Classic offers flow control methods at the transportation layer, the buffer at accessory side can be easily flooded without prompting errors. This function is also implemented with a timeout mechanism in the absence of a

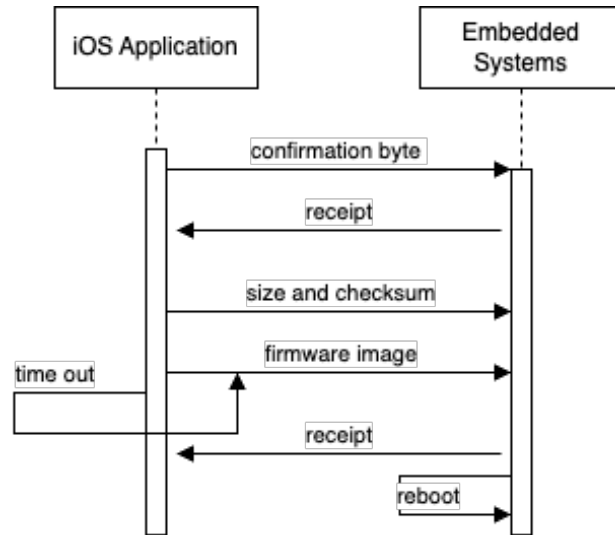


Figure 3.8: `uploadFile()` Sequence Diagram

receipt. This mitigates the risk of the program becoming trapped in a loop iteration indefinitely.

`uploadFile()` is the only public function in `OTAManager`, as illustrated in Figure 3.8. It controls the progress of transmission by processing receipts and invokes `writeFileDataToAcc()` to transmit the large payload. It first transmits a predefined string to the hardware accessory. This string marks the start of the OTA task and the target updated device, prompting the STM32U5 microcontroller to transition into the OTA subroutine. The STM32U5 is programmed with consideration of other functions, such as sensor support and data processing, as part of potential future work. The `uploadFile()` then computes a checksum or digital signature, depending on the setup of the program. This signature is transmitted to the accessory before the firmware transfer begins such that the microcontroller can verify the integrity of the payload before it performs DFU. If a

discrepancy is detected, indicating potential tampering or corruption, the mission is aborted to prevent compromised firmware installation.

Similar to `writeFileDataToAcc()`, the public function `uploadFile()` and can be interrupted by disconnecting from the current `EASession`. This feature provides users with the flexibility to halt the transmission process if they encounter unexpected issues or suspect that the program may be stuck in a loop. This design enhances the robustness of the OTA update process.

Document Browser

To browse files from System Library, we use the `UIDocumentPickerViewController` class. It provides a user interface for selecting and importing documents from sources other than the application's sandbox. The `onClickSendSystemFile()` function is connected to the select firmware button in the main view. It temporarily presents the document picker view controller, which opens up Files, as shown in Figure ???. The `documentMenu()` function exclusively accepts input of type `UTType.Data`, which represents raw binary data type. Files of other types are not selectable. The `documentPicker()` is presented by `documentMenu()` modally. By selecting a document, it reads data from the input URL and passes it to the `OTAManager` function `uploadFile()` for further processing.

3.3 Host MCU Design

The microprocessor is the central processing unit of the embedded system. It oversees the operation of the entire system, coordinating the flow of data. It also interfaces with other peripherals through GPIO and UART interfaces. It is the core of the project, representing the focal point of our design endeavours.

We have conducted experiments with two main microprocessors for the scope of this project: the STM32U585 and the STM32U5A5. The STM32U585 is equipped with 4MB of flash memory, while the STM32U5A5 features 2MB of flash memory. Both microprocessors utilize the same Cortex M33 processor architecture and offer support for dual bank functionality. This feature divides the flash memory into two banks, allowing for the selection of either bank for rebooting purposes. Dual bank functionality is particularly advantageous for the function where we aim to perform firmware upgrades for the STM32U5 board itself. We design our project to accommodate both platforms through differentiation in flash loading addresses.

Throughout various development stages of our project, the scope and behavior of the MCU has evolved. Initially, we designed a standalone OTA DFU project. The MCU only manages incoming data and performs DFU for different components. Later, we combine the OTA function with the location service and security protocol subroutine. The microprocessor also handles geographic positioning data and security protocol establishment. Figure 3.9 depicts an overview of STM32U5's software design. The security

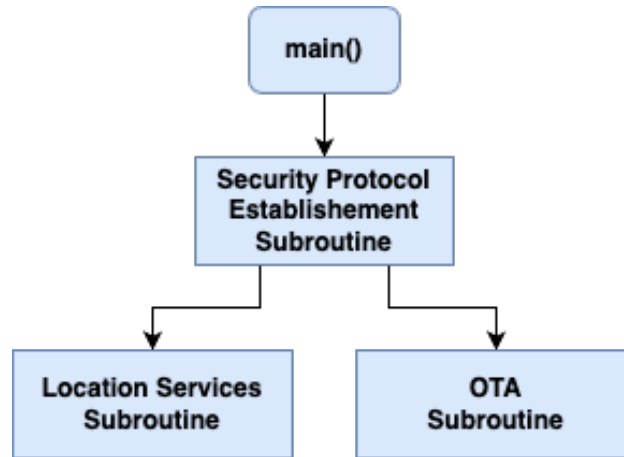


Figure 3.9: Host MCU Software Design Overview

protocol subroutine is responsible for establishing wireless connection to the iOS device. For all subsequent communication, the security protocol function in the host MCU removes the headers from the packets and expose the payload to the rest of the program. From there, we use the headers to distinguish whether this is a location service packet or OTA packet. If it is OTA, then we pause other subroutines and enter OTA. Consequently, we design the OTA DFU subroutine as an automated and well-encapsulated module with error handling mechanisms and version control. It does not interfere with other functions in the program and it is compatible with complicated system designs. The OTA subroutine processes the DFU differently depending on the target and there are component-specific design details. We then incorporate the communication protocol management into the OTA DFU subroutine and adapt our data processing mechanisms accordingly. The following section examines the foundational configurations and functionalities that we implemented to facilitate stable data transfer as well as component-specific DFU designs.

| STM32U5 Pin | RX | TX | RTS | CTS | GPIO Input | GPIO Output |
|-------------|----|----|-----|-----|-------------|-------------|
| BT122 Pin | TX | RX | CTS | RTS | GPIO Output | GPIO Input |

Table 3.2: STM32U5A5 and BT122 Connection Details

3.3.1 I/O Configurations

In this OTA DFU project, we are performing firmware transmission through Bluetooth Classic wireless link. However, from the perspective of the STM32 microcontroller, data transmission does not occur wirelessly. Instead, the Bluetooth module redirects all Bluetooth input to the microprocessor via the UART interface. Table 3.2 depicts the connection details of the STM32 board. The Bluetooth module is connected to UART 2, and flow control is enabled. There are also two GPIO pins connected to the Bluetooth module, as discussed in detail in Chapter 3.1.4, to toggle BT122's script mode or BGAPI mode. The output GPIO pin from the MCU should be set "HIGH" by default and toggle to set "LOW" in order to trigger a falling-edge interrupt on the wireless module.

There are multiple UART ports available in the microprocessor, and the user is free to choose any for the Bluetooth module connection except UART 1. It is physically wired to the USB pin connector. For more debugging clarity, we should reserve UART 1 for console communication with PC.

3.3.2 Data Reception

Background

In firmware upgrades, the system must show robustness to efficiently handle the transmission of large packets within short time frames. As the central component for data processing, the STM32U5 must maintain stability and reliability, regardless of whether the receiving channel experiences sporadic or heavy packet loads. Even the loss of a byte leads to the failure of the entire transmission process. When considering the embedded system as a whole, Bluetooth BR/EDR and security protocols offer reliable flow control mechanisms. The Bluetooth module's streaming performance also exhibits reliability. The primary design challenge lies in the robust implementation of the UART receiving function. There are three UART reception modes available in the STM32 HAL library, interrupt, polling and DMA mode.

In polling mode, the CPU constantly check the receive buffer for incoming data in a loop. We experimented with this approach during early stages of our design. While this method facilitates data retrieval, it also results in high CPU utilization, especially when the channel is idle for a long time. Moreover, during file transfer operations, the periodic polling results in data loss. For example, polling is implemented with

`HAL_UART_Receive(huart, pData, size, timeout)`, where `huart` is the pointer to the UART port receiving the data. `pData` is a pointer to the destination data buffer. The CPU queries the target UART port for the desired number of payload and times out if no

sufficient bytes are found. For this reception function to work, we need to know exactly the amount of bytes we are expecting in each conversation. If we set the target input size to one (`size = 1`), then the CPU reads the first byte from the input buffer and drops the remaining buffer contents. This outcome is not acceptable for file transfer because a tampered firmware file is an invalid firmware file. One can work around this limitation by increasing the polling frequency, coming at the cost of a huge waste of CPU power. In addition, we need to implement the iOS application accordingly. For example, designing more security checksum and receipts to pinpoint the data transmission and to resume as desired. Both the STM32 and iOS devices must monitor the transmission progress. In the event of a tampered or incomplete packet, they can re-transmit the specific packet. However, this approach significantly increases design complexity, leading us to conclude that polling mode is less reliable compared to other methods.

The DMA mode stands out as another viable option for UART receiving mode. Further discussion on this option is presented in Section 4.3.1.

Implementation

The interrupt mode, on the other hand, provides more flexibility towards the unexpected incoming flow of data. It is robust towards both an inconsistent data stream and the rising of expected data packets. Data is processed upon reception such that it utilizes the CPU efficiently and results in low latency. To cope with the unpredictable nature of the data

flow, we want to receive data byte by byte. That is, for every byte received on the UART interface, an interrupt is triggered. As implemented in

```
HAL_UART_IT(huart, pData, size, timeout)
```

the size is set to 1. If not, the challenge is similar to that arising from the polling mode. If ten bytes are received, and the hardware is expected to read six bytes from the UART interface, then the other four bytes are dropped automatically. We always risk losing the remaining data in the channel if the input size cannot be flexible.

We address this challenge by performing byte-to-byte data processing in the callback function. This approach is particularly useful in the secure link establishment stage because we can design the performance in response to the security headers. As discussed in Section 3.1.1, the protocol specifies unique headers that contain information about the packet type, payload length and checksum. With the careful handling of authentication code messages, we can directly use the interrupt mode to distinguish and process the authentication codes and communication headers.

Yet, during the firmware transmission stage, the transportation layer headers do not reveal accurate information about the payload. We can get information on the total size of the payload in this packet, but we cannot know if this packet contains more than one type of data. For instance, the firmware digest and firmware size are transported within the same packet, labelled in red and blue respectively, as shown in Figure 3.10. If the host MCU is expecting to receive a four-byte firmware size, but the transportation layer subroutine

| Raw Bytes (99 bytes) | | | |
|----------------------|-------------------|----------|--|
| 0000 | ff5a0063 407a0502 | .Z.c@z.. | |
| 0008 | 83001800 00000000 | | |
| 0010 | 03e00053 48413235 | ...SHA25 | |
| 0018 | 36206469 67657374 | 6 digest | |
| 0020 | 3a203764 37313030 | : 7d7100 | |
| 0028 | 31336263 38303164 | 13bc801d | |
| 0030 | 65386666 35346266 | e8ff54bf | |
| 0038 | 34393564 63323766 | 495dc27f | |
| 0040 | 62323836 34336166 | b28643af | |
| 0048 | 37643336 34306631 | 7d3640f1 | |
| 0050 | 66353039 32303635 | f5092065 | |
| 0058 | 38346363 63616531 | 84cccae1 | |
| 0060 | 333642 | 36B | |

Figure 3.10: A packet that contains two variables

transmits more than just the desired data, the MCU reads the four bytes from the payload and disposes the rest. Some data is lost unless we create a separate buffer.

To design a plug-and-play modular firmware upgrade subroutine, we choose to create a receiving buffer of our own. Figure 3.11 depicts the UART Reception interrupt service routine and how it serves the OTA program from a high-level point of view. Raw packets are analyzed and processed by the security subroutine first. The payload is then sent to the OTA buffer, where it is stored and later accessed by the OTA subroutine accordingly. We define our custom UART Receive Class to ensure the encapsulation of the OTA subroutine variables. Figure 3.12 is the UML diagram of the UART Receive Class. Privately, there are three functions that interact with the buffer, `put_uart()` writes data into the buffer, `get_uart()` reads data out of the buffer, and `clear_uart()` wipes the buffer. The buffer is defined and allocated with an amount of memory larger than a chunk size upon the reception

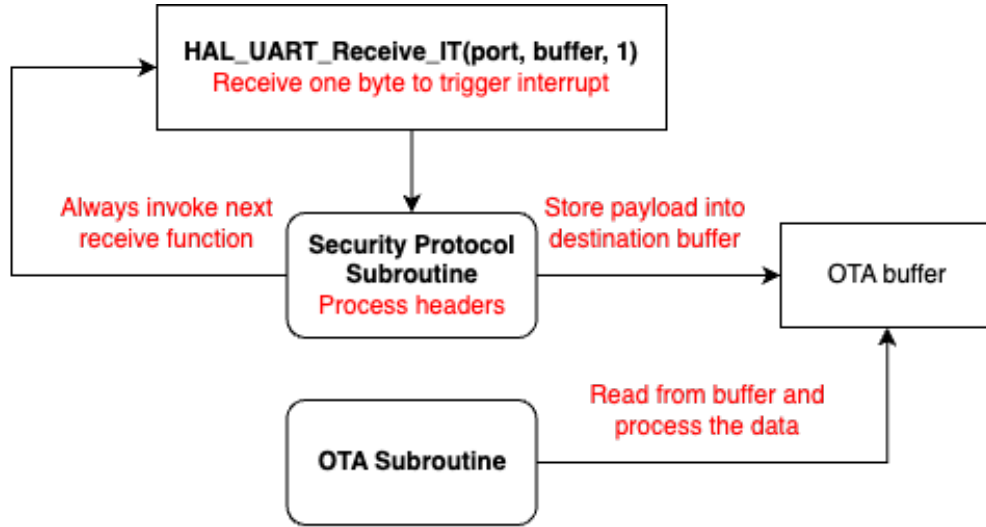


Figure 3.11: Interrupt Service Routine and OTA Subroutine

of the first OTA packet. A chunk size should be the integer multiple of the hardware's flash page size because the firmware image is relatively large and should be stored in flash. We set the chunk size equal to the size of a size page for stability in system performance. Flash memory is written page by page, so the buffer should keep at least a page of data before writing to flash.

To interact with the buffer, the program keeps track of two indices, the read index and the write index. The write index is modified and updated every time the `put_uart()` writes data into the buffer. The read index is updated after `get_uart()` accesses the buffer. These two indices are not correlated and most of the time not equal, due to the unexpected nature of data flow. The two indices are also modified by `clear_uart()`, which resets them to zero, instead of wiping out the buffer.

In the OTA subroutine, we further encapsulate the `get_uart()` function with a public

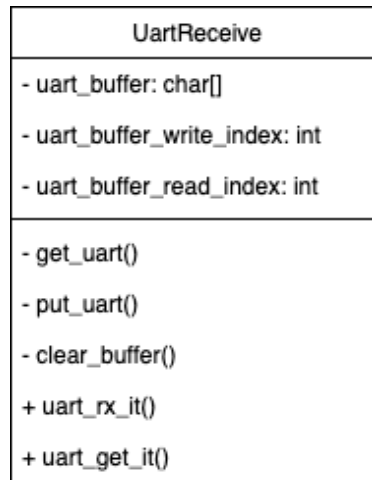


Figure 3.12: UML Diagram for OTA's Customized UART Buffer

function `uart_rx_it()`. When the microprocessor is expecting a string from the Bluetooth module UART port, we specify the target variable and expected size to access the buffer using `uart_rx_it()`.

3.3.3 OTA Subroutine

There are two components that our project is capable of upgrading, the BT122 and STM32U5 main microprocessor. We implement customized functions for different components. Both functions utilize a shared subroutine for connection and data reception. Figure 3.13 shows how data is exchanged between the embedded side and iOS device. The end device transmits a special packet with OTA DFU headers before transmitting the payload. This is for the STM32U5 device to isolate the OTA process from other functions and enter the subroutine. After that, the iOS device transmits the payload size, such that the hardware accessory can

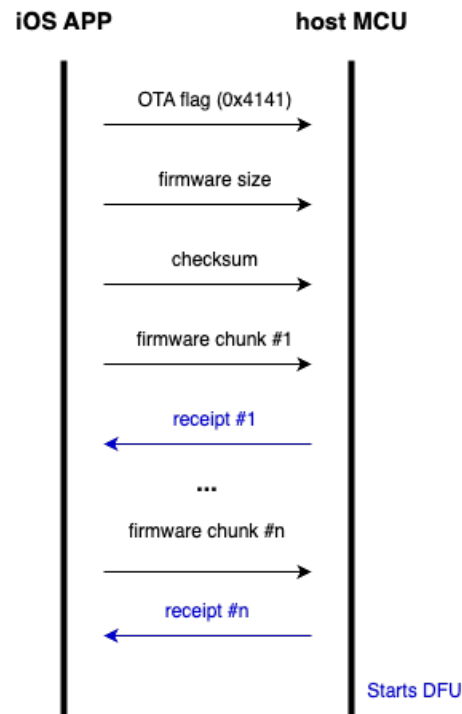


Figure 3.13: Data Exchange between MCU and iOS device in OTA DFU

check whether it has sufficient flash memory for the task. It then transmits a hash digest as its digital signature.

After that, the iOS device transmits the firmware image chunk by chunk. The size of each chunk should be an integer multiple of a flash's page size because the flash must be written page by page. In our case, we pick the chunk size that is equal to the page size. The smaller the chunk size is, the more stable the system's performance is. The MCU transmits a receipt to the iOS device upon successfully receiving a chunk. The iOS device always waits for the receipt before transmitting the next chunk. It is witnessed empirically often that some bytes are lost during the file transmission stage if we do not perform flow control. This

might be a result of a flooded UART reception buffer. The data arrival speed is much larger than the processing and writing speed. Some bytes are lost if we do not manually control the transmission. These transportation layer handling, such as the security protocol subroutine, UART buffer, and flow control, masks lower-level problems from the application level OTA subroutine.

The MCU will also compute the SHA-256 hash digest using its built-in hash module. If the two strings do not match, the MCU will not proceed with OTA and abort the task. If they match, the STM32U5 will proceed to upload the verified firmware image to the target component.

3.3.4 DFU Design for Bluetooth Module

With a verified firmware image stored in flash, the STM32 microcontroller can upgrade itself or BT122. Upgrading BT122, from STM32 side, can be abstracted as data transportation through UART and BGAPI commands manipulation.

STM32 changes BT122 to BGAPI mode and transmits the payload back to BT122. As mentioned in 3.1, BT122 runs in script mode for packet-routing and switches to BGAPI mode to perform DFU. Figure 3.14 illustrates the process. By detecting the GPIO pin connected to BT122, STM32 can get knowledge of BT122's mode and switch it accordingly. When BT122 is in BGAPI mode, host MCU uses the `dfu_reset()` BGAPI command to reboot the Bluetooth module into DFU mode. If the reboot is successful, the BT122 responds

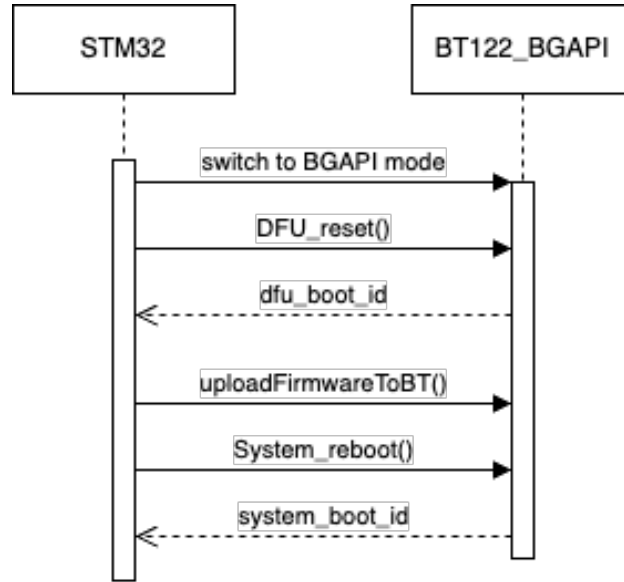


Figure 3.14: DFU Process with BT122

with `evt_dfu_boot`. The STM32 device then sends command sets the flash start address in BT122 and start transmitting the firmware image to BT122. After that, reset the BT122 and its bootloader will verify the firmware and boot it from there.

When BT122 is in BGAPI mode while UART is connected to a source of input, it is not possible to send BGAPI commands over BGTool console. We can, however, for debugging clarity, use BGTool's Interactive console to monitor the BGAPI message ID response in real-time. Figure 3.15 is an example of captured BGLIB communication, where we can see the module's response message. When switching the Bluetooth module to BGAPI mode, it would generate BGAPI messages indicating the detection of interrupt, current endpoint state, and result of BGAPI mode switch, as seen in Figure 3.15. We should ensure that the UART buffer is empty before transmitting DFU booting command to BT122. It is crucial

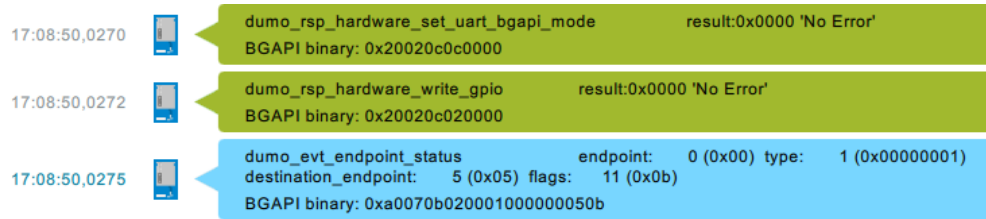


Figure 3.15: Captured BGAPI Response Messages

that the re-initialization stage follows this procedure, as shown in the below code snippet. After setting BT122 into BGAPI mode, we should wait long enough to start re-initialize the UART port buffers. We use `register_UART` to create the custom UART buffer, as discussed in 3.3.2. We then set the UART port as the destination of BGLIB commands and trigger the interrupt to initialize the BGAPI response reception.

```

1 setBT122UARTMode(BGAPI_MODE); // switch the Bluetooth module to BGAPI mode
   for DFU commands
2 uart_rx_it_clear_buffer(get_UART_num(huart_port)); // clear the UART buffer
   and prepare for a reset
3 HAL_UART_DeInit(BT_port);
4 HAL_Delay(1000); // wait for the Bluetooth module to reset to avoid
   receiving irrelevant messages
5 HAL_UART_Init(BT_port); // re-initialize the port
6 register_UART(2,BT_port); // re-register the UART port for our customized
   Receive function
7 initializeBGLIB(BT_port);
8 HAL_UART_Receive_IT(BT_port, (uint8_t *)&IAP_bytereceived_it, 1); //
   receive messages from the port and trigger interrupt. Rest of the

```

communication is handled by the interrupt service routine

One should design the GPIO pins with discretion such that BT122 can reboot successfully into DFU mode. As BT122 is configured has falling edge triggered interrupt, the GPIO output pin on the MCU side should be set to high by default and output a falling edge behaviour as necessary. To generate firmware image for BT122, use command `line image out = \filename.bin` in `project.xml` to configure the output.

3.3.5 DFU Design for Host MCU

With the verified firmware stored in the flash, DFU is the same as using the bootloader to reboot from flash. We use the dual bank functionality offered by the hardware for DFU. We download the firmware image to the second bank such that the program can reboot itself directly from the second bank. Figure 3.16 demonstrates this subroutine in detail. We need to toggle the flash option byte, `OB_SWAP_BANK_ENABLE`, to enable the bank swapping feature. Option bytes are non-volatile memory locations that hold configuration settings that can be programmed. Then, reboot the device from the second bank and DFU is completed.

To generate firmware image for STM32, STM32CubeIDE provides such support by configuring the MCU Post build outputs to produce the binary file.

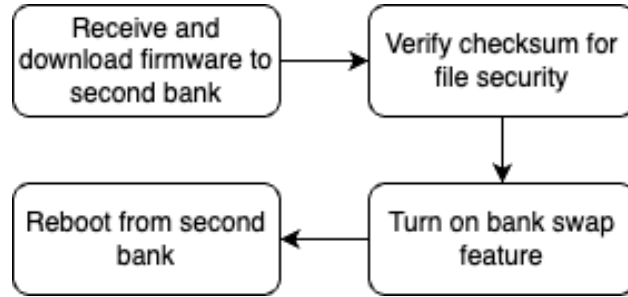


Figure 3.16: DFU Process with STM32U585

3.3.6 Automation and Robustness Design

We find that empirically the performance of the embedded system is not as reliable as expected, especially when the payload size is large. We thus implement a series of tools to improve the system's robustness and stability.

Flow Control

In order to perform flow control, we divide the payload into small chunks and introduce a receipt each time after a chunk of data is transmitted. By transmitting smaller segments of the payload, we process each chunk independently, thus reducing the likelihood of data congestion or overflow. Also, this design improves the responsiveness of the system and enables us design more error handling techniques. We also implement the microprocessor with an acknowledgement mechanism. It replies with a receipt upon having seen sufficient bytes and the iOS does not send the next chunk if the receipt is not received. The STM32U5 can control of the transmission and adjust its rate accordingly.

Timeout Handling

We also implement a timeout mechanism to prevent the program from becoming stuck in an infinite loop. The below code demonstrates our design. For example, when checking the input buffer size using the condition `while(uart_rx_it_get_length(huart) < 15)`, where '15' represents the expected size of the payload, a timeout mechanism becomes essential. If, due to transmission errors or other issues, parts of the payload are missing, the STM32U5 may continue waiting indefinitely, leading to a deadlock situation. Additionally, the iOS application on the other end may also be waiting for the receipt from the microcontroller. In the event of a timeout, we set a public variable, to the iOS device, requesting re-transmission of the missing data chunk. This approach represents a trade-off between transmission latency and reliability, as it ensures that the system remains responsive while taking longer time.

```
1 uint32_t startTime = HAL_GetTick();
2 while (uart_rx_it_get_length(huart)<size && elapsedTime<TIMEOUT){
3     currentTime = HAL_GetTick(); // timer management
4     elapsedTime = currentTime - startTime;
5 } // check if there are sufficient contents in the buffer
6 if (elapsedTime >= TIMEOUT) {
7     OTAFlag = 0;
8     return 1;//handle timeout in OTADFU
9 }
10 uart_rx_it(huart, size, &buffer[0]); //read from buffer
11 return 0; //successful
```

Security Design

Though the communication-level security protocol only allows authenticated source to provide firmware for the embedded system, we still want to improve the system's security towards eavesdropping and man-in-the middle attack. We encrypt the payload with symmetric encryption, AES, and check the integrity with HASH. In AES, a symmetric encryption scheme, the sender and receiver use the same key for encryption and decryption. To establish a shared key without revealing it to any eavesdroppers, we use the Diffie-Hellman key exchange protocol.

Thus, the two parties can securely exchange a mutual key for encryption and decryption. Also, in response to the instability in transmission process, we use hash to verify the integrity of the payload. A hash algorithm takes an input message and produces a fixed-size string of bytes. The output, or the hash digest, is a unique representation of the input data. We use SHA-256 as our hash function. Regardless of the size of the input, 254 kB or 8 kB, it produces a fixed-size output of 32 bytes. Compared to the firmware image payload, a hash digest is not a considerable overhead. Hash is collision resistant, such that it is computationally infeasible to find two inputs with the same digest. It also improves the system's security towards the MITM attack. We can find out with low overhead, whether the received data has been modified.

Chapter 4

Results and Discussion

4.1 Results

We design an embedded system that performs secure over-the-air firmware updates for different components. There are three parts to the project setup, the main microprocessor, the wireless module, and the user end device. We design the transport layer and application layer performance of the components to achieve secure OTA DFU. The user can select the upgrade target as desired.

Figure 4.1 is the console output for the wireless module update, indicating that it has been upgraded with a different firmware image. The Bluetooth module reboots into DFU mode and firmware file is uploaded from the main processor's flash to the Bluetooth module's flash, as seen in Figure 4.2. Upon the reception of 1985 `dfu_flash_upload` response ID's, the

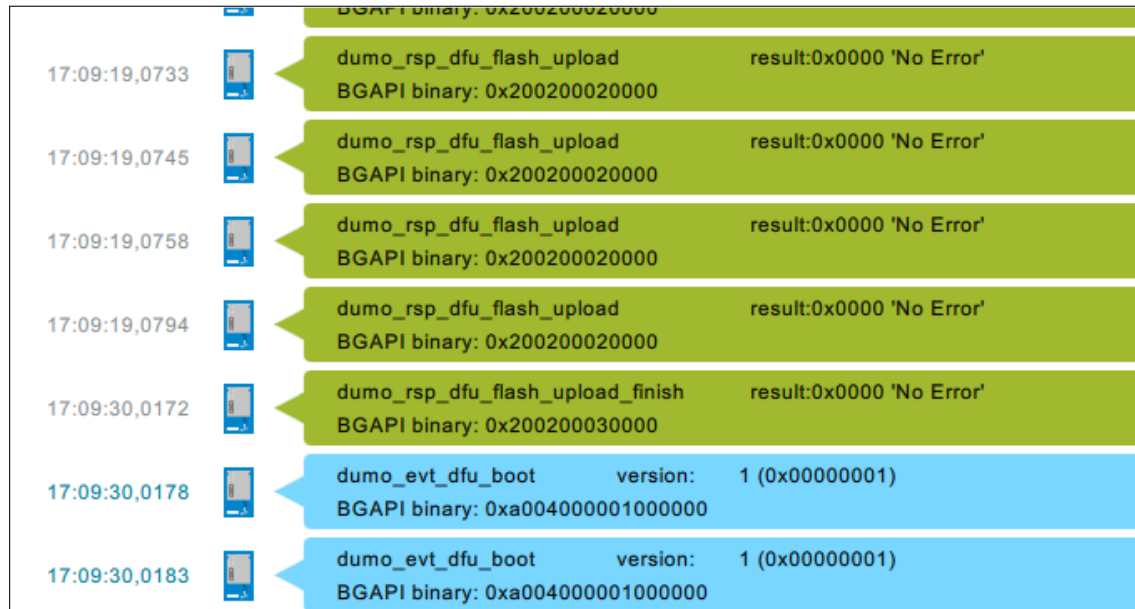


Figure 4.1: Screenshot from BGTool Console, Successful Reboot from New Firmware Image

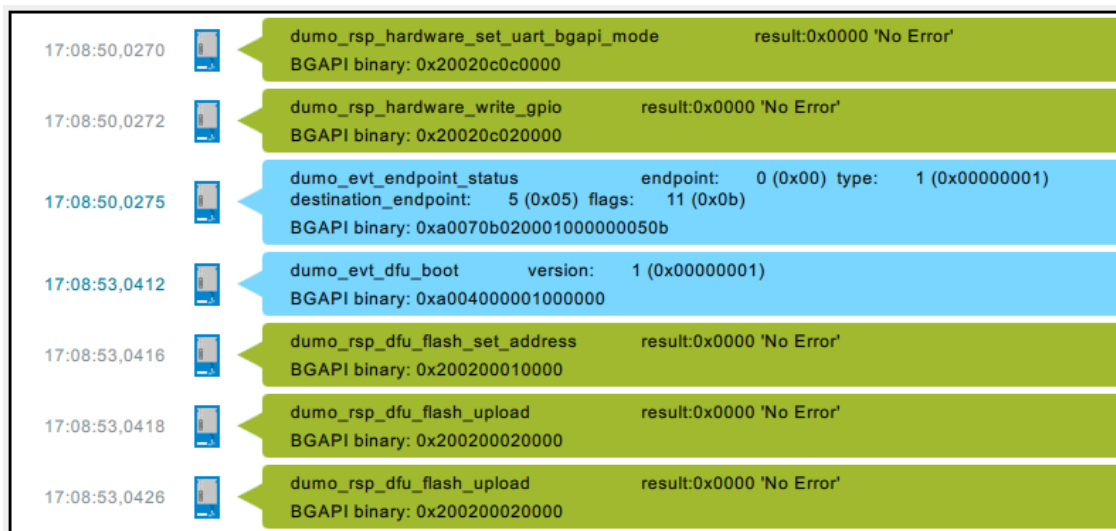


Figure 4.2: Screenshot from BGTool Console, Reboot Process

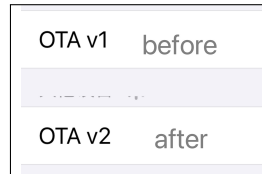


Figure 4.3: Screenshot from iOS device, Update of Device Name

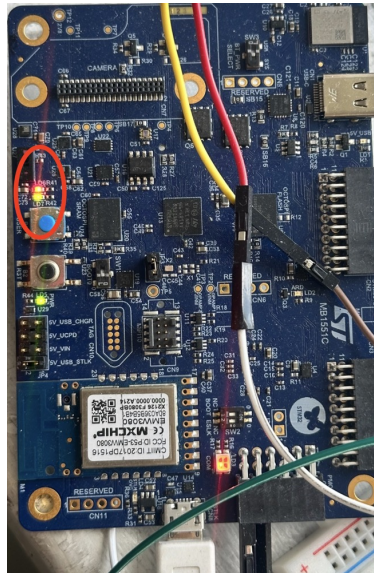
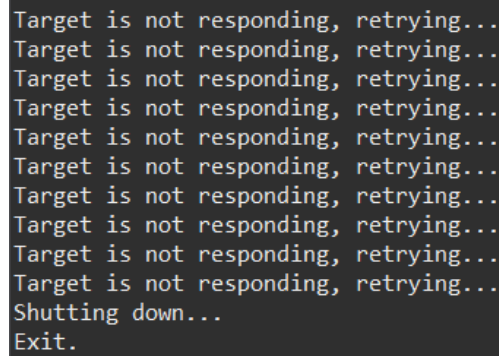


Figure 4.4: MCU with Red LED Indicating New Firmware Version

main processor sends a `dfu_flash_upload_finish` flag to the wireless module and triggers a DFU reboot command. From there, the module reboots on the new firmware version and sends back a response ID, 1, indicating successful OTA DFU procedure. For BT122 in particular, we can update many things during DFU, including the Bluetooth stack, the BGScript application, bootloader, and general project configurations.

Figure 4.3 is another successful example where we upgrade the device's Bluetooth-friendly name. As we boot the device on the firmware version that contains the new Bluetooth

A screenshot of a terminal window with a dark background and light-colored text. The text shows a series of ten lines, each saying "Target is not responding, retrying...", followed by "Shutting down..." and "Exit." on the final line.

```
Target is not responding, retrying...
Target is not responding, retrying...
Target is not responding, retrying...
Target is not responding, retrying...
Target is not responding, retrying...
Target is not responding, retrying...
Target is not responding, retrying...
Target is not responding, retrying...
Target is not responding, retrying...
Target is not responding, retrying...
Shutting down...
Exit.
```

Figure 4.5: Screenshot from MCU Console, Reboot

friendly name, OTA v2. It is recognized by other Bluetooth devices as this name.

Our project supports updating the host MCU as well. By downloading the firmware image into one bank of the flash, the STM32U5 which supports dual-bank functionality can reboot from that bank and perform a DFU. Figure 4.4 is a picture of a successful trial, where we program the new firmware with a blinking red LED. The red LED on the left side of the picture indicates the new firmware version. Also, each time when a firmware upgrade is performed and the microcontroller reboots from another bank, from the debugging console, we witness a non-responsive system prompt, as shown in Figure 4.5, because rebooting disconnects the MCU from the USB link and terminates the console output to the PC.

4.2 Discussion

Firmware updates play a critical part in a hardware accessory's life cycle. Designing an automated and robust over-the-air system enables us to upgrade a microcontroller that is

| | Proposed Model | BT122's built-in project | STM32H5's built-in project |
|----------------------|-----------------|--------------------------|----------------------------|
| Upgrade Target | STM32 and BT122 | BT122 | STM32 |
| Authenticated Source | Yes | Yes | No |
| Wireless Connection | Yes | Yes | No |
| Wireless Security | Yes | No | Not Applicable |

Table 4.1: Overview of Comparison with Baseline

otherwise physically inaccessible. At the same time, being wireless introduces vulnerabilities to the system and requires security designs. By implementing the security protocol, we ensure that only devices with matching protocols can provide firmware images and that it is a trusted source of data. In the meantime, we use asymmetric cryptographic schemes to ensure the integrity and non-repudiation of the payload. We also implement a user-friendly iOS application, such that the user has access to a more interactive and simple way of controlling the microcontroller system.

4.2.1 Baseline

Table 4.1 summarizes the comparison of our project with the baseline projects. First, we compare our model to the BT122's built-in OTA DFU project. In this setup, the Bluetooth device is connected to an external flash, or a separate development board as a data storage unit. The BT122 is programmed with a special OTA bootloader, such that it enables it to reboot from external memory. Also, this project can only upgrade the Bluetooth module alone. It is impossible if the user wants to upgrade other parts in the embedded system. A

measured DFU time for a 254 kB firmware image is similar to our implementation.

Another baseline performance is a wired DFU setup. We compare our model to the STM32H5’s built-in DFU project. While it demonstrates faster transmission and lower error rates, a wired transmission has a lot of physical accessibility constraints. In IoT, over-the-air updates bring significant benefits and flexibility to device deployments. We try to ensure a reliable, efficient and secure file transmission process while maintaining the wireless nature of the setup.

4.2.2 Performance and Error Handling

We measured the percentage of successful OTA firmware updates across multiple trials. Table 4.2 depicts the measured performance in detail.

In our evaluation, a successful update was defined as one where the firmware was successfully booted onto the target device without any errors. We also measure the time taken for the entire firmware transmission process, which only includes wireless communication time, but not firmware loading time or DFU rebooting time, as that is subject to the performance of the upgrade target. To account for discrepancies in network conditions, we perform these tests from arbitrary distances to the Bluetooth module. We use the application ATS to monitor the Bluetooth communication happening on an iOS device and by observing the trace, we get information regarding the transportation latency.

There are three files that we used for testing, with sizes of 5 kB, 37 kB, and 254 kB

| File Size | Success Rate (%) | Transmission Duration (seconds) |
|-----------|------------------|---------------------------------|
| 5 kB | 100 | 1 |
| 37 kB | 100 | 7 |
| 254 kB | 95 | 51 |

Table 4.2: Measured Performance of OTA DFU

respectively. The firmware size for the Bluetooth module varies between 5 kB to 37 kB, and size of the MCU's firmware is fixed at 254 kB. We picked the edge cases for testing. For each test case, we conducted 20 update attempts and calculated the average performance in all metrics.

As shown in Table 4.2, we find that when the file size is small, especially when smaller than the size of a flash page on the host MCU, 8192 bytes, the system demonstrates great stability. For a large file that takes up to 31 pages in Flash, the success rate witnessed a drop to 95%. There was one failure when the host MCU jumped to the file checksum comparison stage without initiating the file reception subroutine.

The success rate for a 254 kB file used to be around 30%, without any error-handling techniques applied. The limiting factor here is the data processing speed of the host MCU. The mobile device and the wireless module both have a high throughput and reliability. However, when a large number of packets arrive at the UART port of the microprocessor, waiting to be processed and saved to Flash, they flood the UART buffer and eventually some data is lost. The host MCU demonstrates less reliability towards a large payload.

To address the error rate, we introduce an acknowledgment and timeout mechanism. The user-end device transmits a chunk of the file and waits for an acknowledgment receipt before transmitting the next chunk. If the MCU does not send the acknowledgment before the system times out, the iOS application re-transmits the chunk. On the embedded system side, we implement a similar timeout mechanism, disposing of packets if the system times out. This approach enhances the success rate of the system at the cost of increased total transmission time.

4.2.3 Wireless Security

In wireless DFU, one of the primary challenges arising from the inherent vulnerabilities of Bluetooth BR/EDR communication, is interception, eavesdropping, and man-in-the-middle attacks. We therefore implement additional cryptography schemes to address these risks and enhance the security of the OTA DFU process. Table 4.3 shows the security features of the OTA DFU system. We have implemented both HASH and AES in the firmware update process. By taking advantage of STM32U5's encryption hardware accelerator, we mitigate the threat of firmware tampering and injection attacks. HASH ensures the integrity of the firmware image, whether it is modified by an adversarial or modified because of a processing. AES encryption protects the firmware from interception and ensures confidentiality of the data. Besides, by using the security protocol, we ensure an authorized source in our communication, because only the iOS application with matching protocol can

communicate with our hardware accessory system.

| Feature | Description |
|----------------|---|
| Encryption | AES-256 encryption for data confidentiality |
| Integrity | SHA-256 hash for firmware integrity |
| Access Control | Security protocol matching for user authorization |

Table 4.3: Security Features of OTA DFU

These securities measures come with performance trade-offs. The processing overhead associated with cryptographic operations, especially AES encryption, may impact the responsiveness and efficiency of the OTA update process. Also, the cryptographic measures pose challenges for processing power of the microcontroller. Empirically we find the output of these security schemes reliable, because they are processed on a separate module other than the CPU. Our cryptographic design poses resource constraints to the MCU. In the absence of a encryption module, it is important to evaluate the trade-offs between security and performance.

4.3 Conclusion

In this thesis, we have developed a secure and robust OTA firmware update system for embedded devices. Our design includes three key components: the host microprocessor, a wireless module, and an iOS application. By integrating security protocols at both the transport and application layers, we ensure the integrity and confidentiality of the firmware updates.

Our results demonstrate the successful implementation of the OTA DFU system, as evidenced by the successful updates of both the wireless module and the host MCU. We achieved a 95% success rate by implementing error-handling mechanisms, significantly improving reliability. The integration of HASH and AES encryption further enhances the security of the OTA process, protecting against common threats such as interception and tampering.

The research significantly advances the understanding and implementation of secure OTA updates in the IoT ecosystem. By addressing both security and performance challenges, we provide a comprehensive solution that balances reliability and efficiency. This project highlights the importance of combining robust security measures with practical performance considerations in the development of embedded systems.

4.3.1 Future Work

During the design phase of the STM32U5 firmware, we utilize interrupt mode to manage incoming packets. This approach involves analyzing security packets and storing payload data in a buffer, which the OTA DFU function subsequently accesses. However, one can only implement the OTA function on a serial execution model. Both the raw incoming data and payload data require real-time processing. For example, the OTA function should verify the firmware size prior to the transmission of the firmware image. We should make the OTA subroutine have access to data as the transmission is taking place. Our approach is to embed the packet analyzing process within the DFU subroutine. While effective, this approach introduced implementation complexity and compromised encapsulation between the transport and application layers.

Multi-threading can be a promising solution addressing these challenges. By implementing separate threads for processing raw input data packets and fetching buffered data, we can design well-encapsulated subroutines and only have public buffers. We will also have the freedom of adding more services to the embedded systems, such as executing OTA DFU and location services in parallel. This approach not only enhances the overall code readability but also maintains the encapsulation of individual routines.

Furthermore, exploring the integration of DMA mode for firmware storage presents a good opportunity for CPU power optimization. DMA transfers data directly between peripherals and memory and offloads the CPU to perform other tasks. Incoming packets can be directly

stored into memory, potentially reducing transmission latency and error rates. Especially when the input stream is of considerable size, for instance, a firmware image, this approach can free the CPU for more time-sensitive processes thus improve overall performance most notably when the CPU power is a scarce resource.

Employing multi-threading and DMA into our DFU design can potentially improve efficiency, reduce complexity, and enhance the performance of the overall system. These advancements not only address existing limitations but also lay the groundwork for future enhancements and optimizations in IoT firmware development.

Another important area of future work deals with preserving privacy, safety and security, while facilitating OTA of the tightly-coupled health closed-loop controllers, especially when they are bound to capture precisely energy and food consumption and provide control for blood glucose management [17].

Finally, regarding the application and generalization of this work, the performance requirements of a BCoT system are very similar to those of OTA DFU. It can be modified to suit the needs of BCoT embedded side realization. For example, the flow control techniques performed in the user end device application, the host MCU application, and the wireless module configurations, can be used towards establishing a BCoT node. The security designs, especially the end validation process, as much as being computationally demanding, are mitigated by the inherently decentralized nature of blockchain. Merging this work into a blockchain environment provides transport layer reliability and robustness.

Bibliography

- [1] K. Arakadakis, P. Charalampidis, A. Makrogiannakis, and A. Fragkiadakis, “Firmware over-the-air programming techniques for iot networks - a survey,” *ACM Comput. Surv.*, vol. 54, oct 2021.
- [2] M. A. Mughal, X. Luo, A. Ullah, S. Ullah, and Z. Mahmood, “A lightweight digital signature based security scheme for human-centered internet of things,” *IEEE access*, vol. 6, pp. 31630–31643, 2018.
- [3] U. Banerjee, A. Wright, C. Juvekar, M. Waller, A. Arvind, and A. Chandrakasan, “An energy-efficient reconfigurable dtls cryptographic engine for securing internet-of-things applications,” *IEEE Journal of Solid-State Circuits*, vol. PP, pp. 1–14, 05 2019.
- [4] R. K. Panta, S. Bagchi, and S. P. Midkiff, “Zephyr: efficient incremental reprogramming of sensor nodes using function call indirections and difference computation,” in *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX’09, (USA), p. 32, USENIX Association, 2009.

-
- [5] N. Reijers and K. Langendoen, “Efficient code distribution in wireless sensor networks,” in *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pp. 60–67, 2003.
- [6] H. Zhu, Z. Zhang, J. Du, S. Luo, and Y. Xin, “Detection of selective forwarding attacks based on adaptive learning automata and communication quality in wireless sensor networks,” *International Journal of Distributed Sensor Networks*, vol. 14, no. 11, p. 1550147718815046, 2018.
- [7] C. Zhang, W. Ahn, Y. Zhang, and B. R. Childers, “Live code update for iot devices in energy harvesting environments,” in *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pp. 1–6, 2016.
- [8] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, “Understanding the mirai botnet,” in *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC’17, (USA), p. 1093–1110, USENIX Association, 2017.
- [9] B. Moran, H. Tschofenig, D. Brown, and M. Meriac, “A Firmware Update Architecture for Internet of Things.” RFC 9019, Apr. 2021.

-
- [10] I. Zhou, I. Makhdoom, N. Shariati, M. A. Raza, R. Keshavarz, J. Lipman, M. Abolhasan, and A. Jamalipour, "Internet of things 2.0: Concepts, applications, and future directions," *IEEE Access*, vol. 9, pp. 70961–71012, 2021.
 - [11] H.-N. Dai, Z. Zheng, and Y. Zhang, "Blockchain for internet of things: A survey," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8076–8094, 2019.
 - [12] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *IEEE Access*, vol. 4, pp. 2292–2303, 2016.
 - [13] J. Wan, J. Li, M. Imran, D. Li, and F. e Amin, "A blockchain-based solution for enhancing security and privacy in smart factory," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 6, pp. 3652–3660, 2019.
 - [14] Z. Li, S. Gao, Z. Peng, S. Guo, Y. Yang, and B. Xiao, "B-dns: A secure and efficient dns based on the blockchain technology," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 1674–1686, 2021.
 - [15] J. Liu, B. Li, L. Chen, M. Hou, F. Xiang, and P. Wang, "A data storage method based on blockchain for decentralization dns," in *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, pp. 189–196, 2018.

-
- [16] R. S. Boparai, A. Alexandridis, and Z. Zilic, “Multi-point security by a multiplatform-compatible multifunctional authentication and encryption board,” *Journal of computing and information technology*, vol. 26, no. 4, pp. 235–250, 2018.
- [17] K. Radecka and Z. Zilic, “Energy and food consumption tracking for weight and blood glucose control,” *US Patent Application 14/570,070*, 2016.