Over the Air Firmware Update

For embedded Microcontrollers

Ver. 1.0

William Fornaciari

AA 2020-2021

POLITECNICO MILANO 1863

Introduction

Why over-the-air device firmware updates (aka OTA DFU, FOTA or OTAF)?

- The increasing demand by end-users for new functionality
- To address bugs and security vulnerabilities (critical and non-critical)
- To ship products to market faster and have the option of delaying lower priority features and being able to roll them out to devices in the field
- … Because it is required by the customers, even if it will never be actually used !

DFU in a single sentence

- DFU is an operation used to, partially or fully, update the firmware on a device

DFU relies on the existence of a bootloader, that is responsible for

- Launching the main firmware or operating system (OS) in a device
- Providing the capability of updating the device's main firmware or OS

DFU sometimes involves multiple parts of the system that can be all, or partially, updated with the firmware update process

- e.g. bootloader, RTOS/stack, and applications

POLITECNICO MILANO 1863

Introduction

The bootloader is optimized and kept to a minimum

- To ensure minimum impact on boot times
- To ensure bugs are kept to a minimum in this critical part of the device's firmware. Bootloaders are rarely updated, so they need to be robust (more lines of code usually increase the probability of bugs)
- The size of the bootloader impacts how much ROM is left for the application firmware

The main operations of a DFU process include

- Updating the application, the stack/OS (and sometimes even the bootloader)
- Verifying DFU package authenticity
- Downgrade prevention
- Verifying hardware compatibility
- Verifying data integrity
- Decrypting encrypted data
- Support for updating over different transport mediums

POLITECNICO MILANO 1863

Introduction

Main benefits of DFU

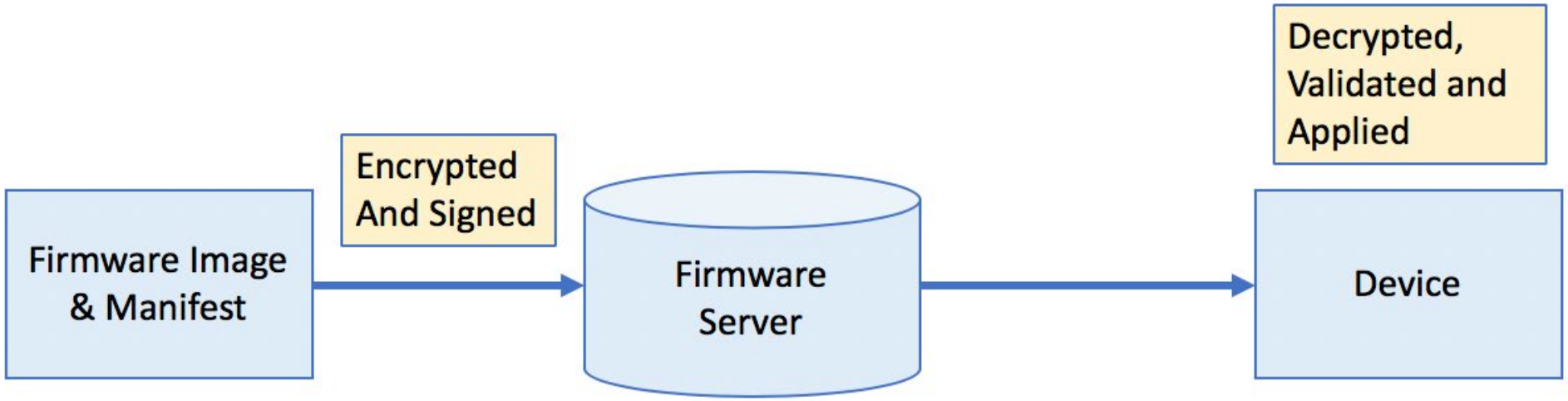
- Adding new features to the end product
- Fixing critical bugs and addressing security vulnerabilities
- Cutting costs: a product recall is usually way more costly than implementing OTA DFU, especially in large product deployments
- but… it is not a good reason to produce not accurately verified/qualified systems!!!!!

Ensuring a Robust and Secure OTA DFU Process

- **Secure**: encryption + identity verification (via digital signatures)
- **Reliable**: verifying integrity + failure recovery
- **Version management**: rollback prevention + versioning system

POLITECNICO MILANO 18

How OTA DFU works: basic steps



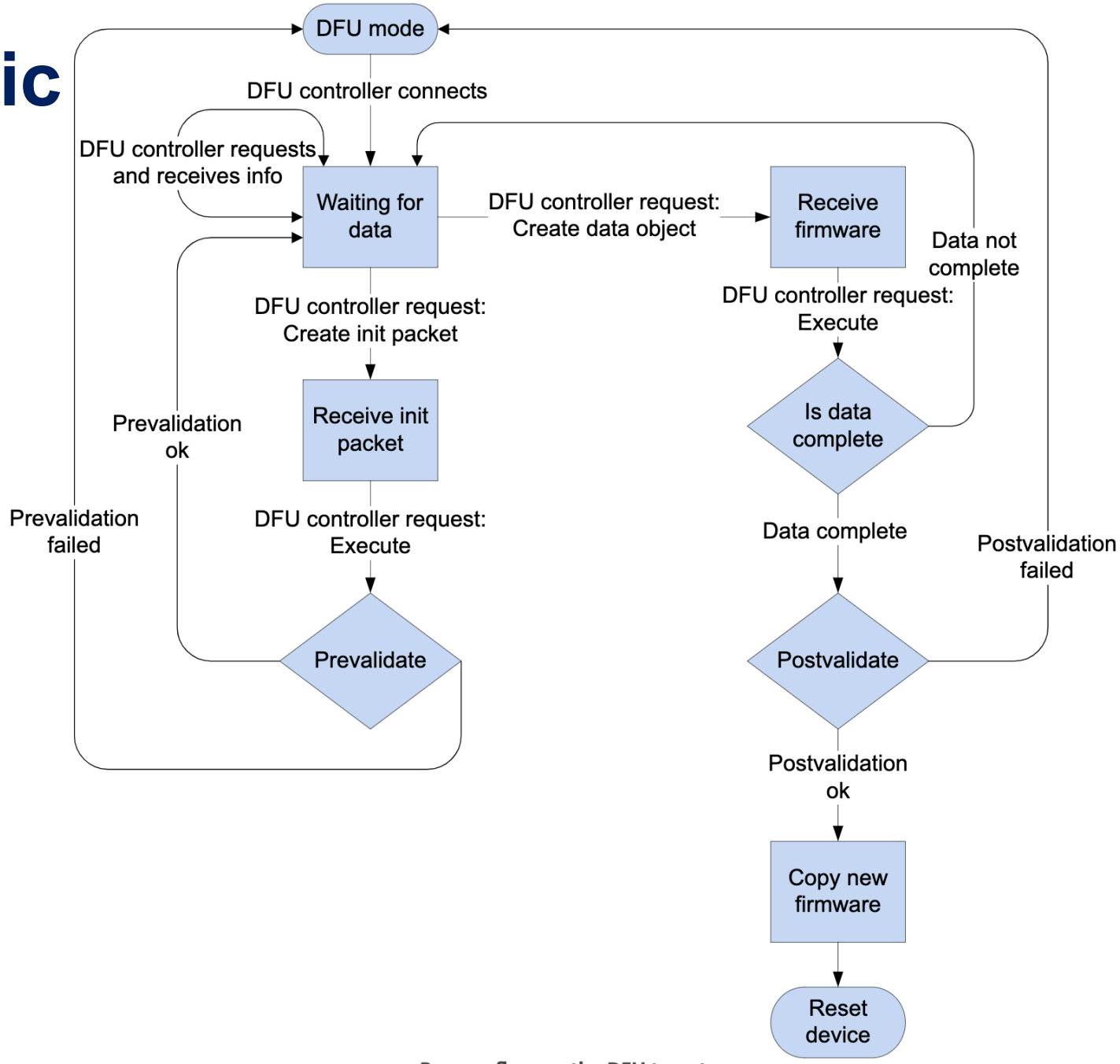
Generic steps (actually depending on the chipset used and system design)

- Firmware Image and manifest are encrypted, signed, and uploaded to the firmware update server
- End-device queries the firmware update server and fetches new firmware image and manifest
- The package will be decrypted, validated and applied
- Be careful…the server itself can be attached, not only the OTA process can be weak

POLITECNICO MILANO 1863

Example: Nordic

- "DFU controller" refers to the device that connects to the target BLE device that needs to be updated and transfers the firmware image to. This could be a mobile phone running an app or nRF Connect on desktop (via an nRF52 DK)
- The target device may be running the bootloader in DFU mode or running the application with DFU running in the background
- The DFU controller connects to the target device and initiates the transfer of the DFU image
- The target device will first receive an init packet which it validates (pre-validation phase)
- The init packet contains important information such as the type of image, hash of the image, firmware version, hardware version, allowed versions of the SoftDevice, and others
- If the init packet is validated, the controller will then initiate the transfer of the DFU image to the target
- The target will receive the full DFU image and validate it
- Once the DFU image is validated, the target will reset and the bootloader will activate the new DFU image to replace the original image

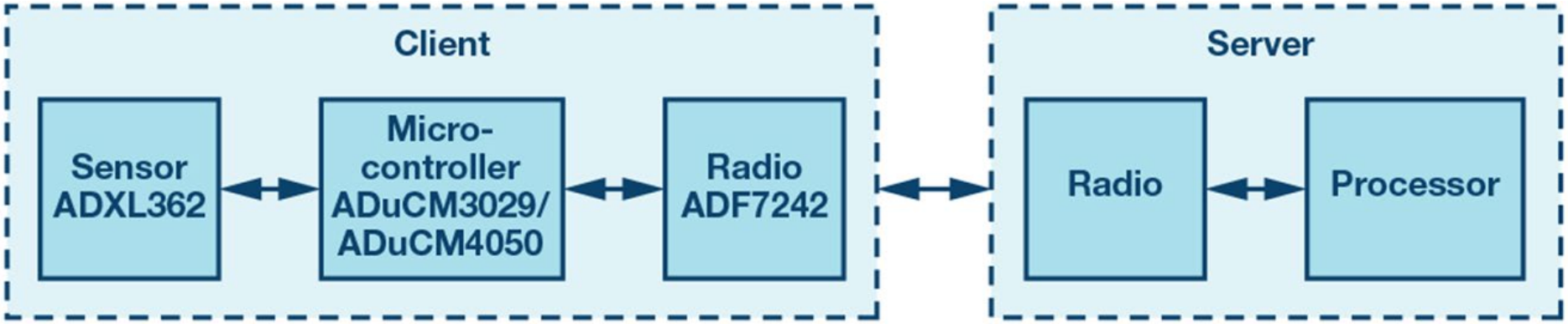


Process flow on the DFU target

POLITECNICO
MILANO 1863

More in general…

Server/client architecture of an emb. sys.



Edge node or client node

- Target of the OTA firmware update
- It contains a microcontroller with limited memory, speed, and power consumption
  - Ultra low power microcontrollers that typically consume 30 µA/MHz to 40 µA/MHz in active mode are ideal for this type of applications
  - Be careful to the status of the battery, if any, before to trigger any OTA process

Cloud or server

- Provider of the new software
- Server and client communicate over a wireless connection using radio transceivers

Server/client architecture of an emb. sys.

OTA update, the data is the new software in binary format

- In many cases, the binary file will be too large to send in a single transfer from the server to the client, meaning that the binary file will need to be placed into separate packets, in a process called *packetizing*
- At a high level, these file formats contain a sequence of bytes that belong at a specific address of memory in the microcontroller

Example

- Each packet contains 8 bytes of data, with the first 4 bytes representing the address in the client's memory to store the next 4 bytes

Major challenges - 1

Memory

- The software solution must organize the new software application into volatile or nonvolatile memory of the client device so that it can be executed when the update process completes
- The solution must ensure that a previous version of the software is kept as a fallback application in case the new software has problems (not infrequent!)
- Must retain the state of the client device between resets and power cycles, such as the version of the software we are currently running, and where it is in memory

Communication

- The new software must be sent from the server to the client in discrete packets, each targeting a specific address in the client's memory
- The scheme for packetizing, the packet structure, and the protocol used to transfer the data must all be accounted for in the software design

Major challenges - 2

Security

- With the new software being sent wirelessly from the server to the client, we must ensure that the server is a trusted party. This security challenge is known as **authentication**
- We also must ensure that the new software is obfuscated to any observers, since it may contain sensitive information. This security challenge is known as **confidentiality**
- The final element of security is **integrity**, ensuring that the new software is not corrupted when it is sent over the air

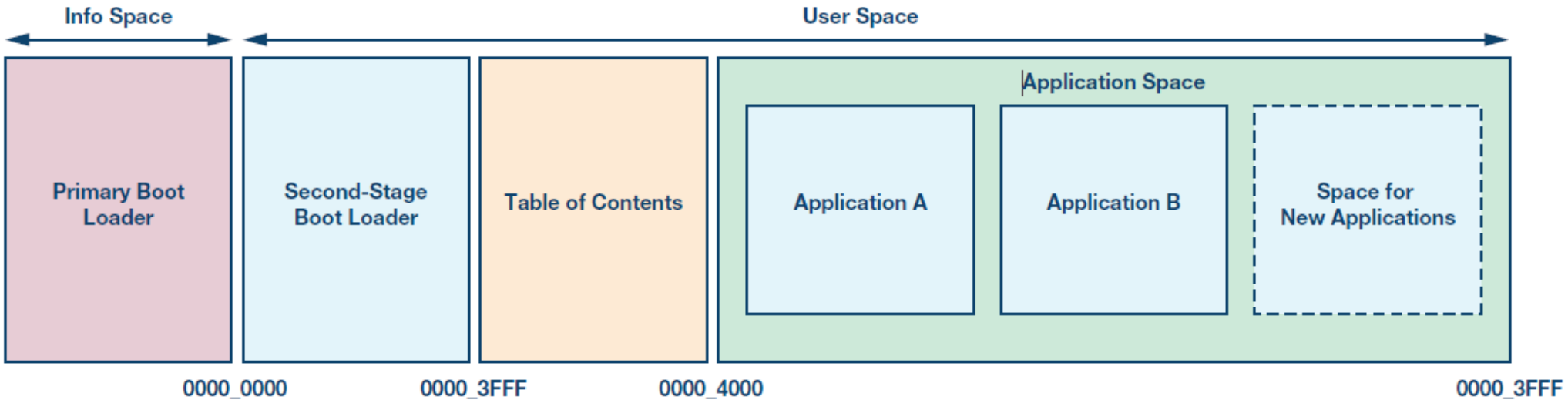
The Second-Stage Boot Loader (SSBL)

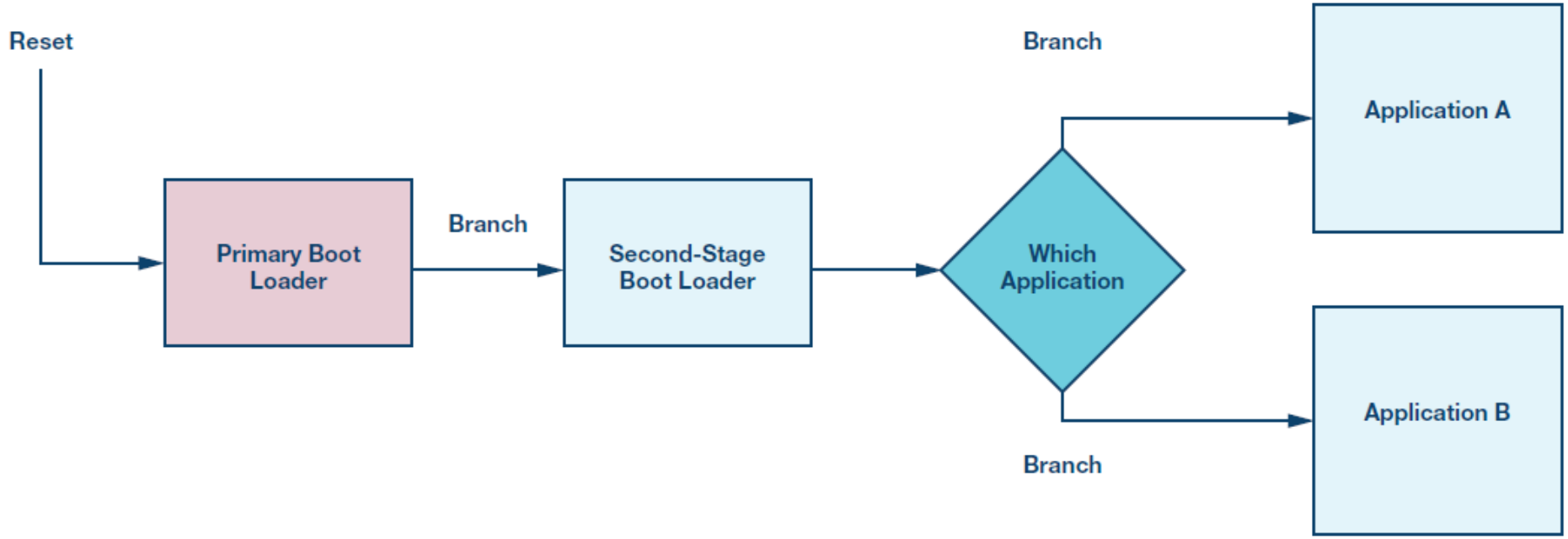
Understanding the Boot Sequence

- The **primary boot loader** is a software application that permanently resides on the microcontroller in read-only memory
  - The region of memory the primary boot loader resides in is known as info space and is sometimes not accessible to users
  - This application executes every time a reset occurs, generally performing some essential hardware initializations, and may load user software into memory. However, if the microcontroller contains on-chip nonvolatile memory, like flash memory, the boot loader does not need to do any loading and simply transfers control to the program in flash
- If the primary boot loader does not have any support for OTA updates, it is necessary to have a **second-stage boot loader**
  - Like the primary boot loader, the SSBL will run every time a reset occurs, but will implement a portion of the OTA update process (see next picture describing the boot sequence)

POLITECNICO MILANO 1863

Typical boot sequence with SSBL

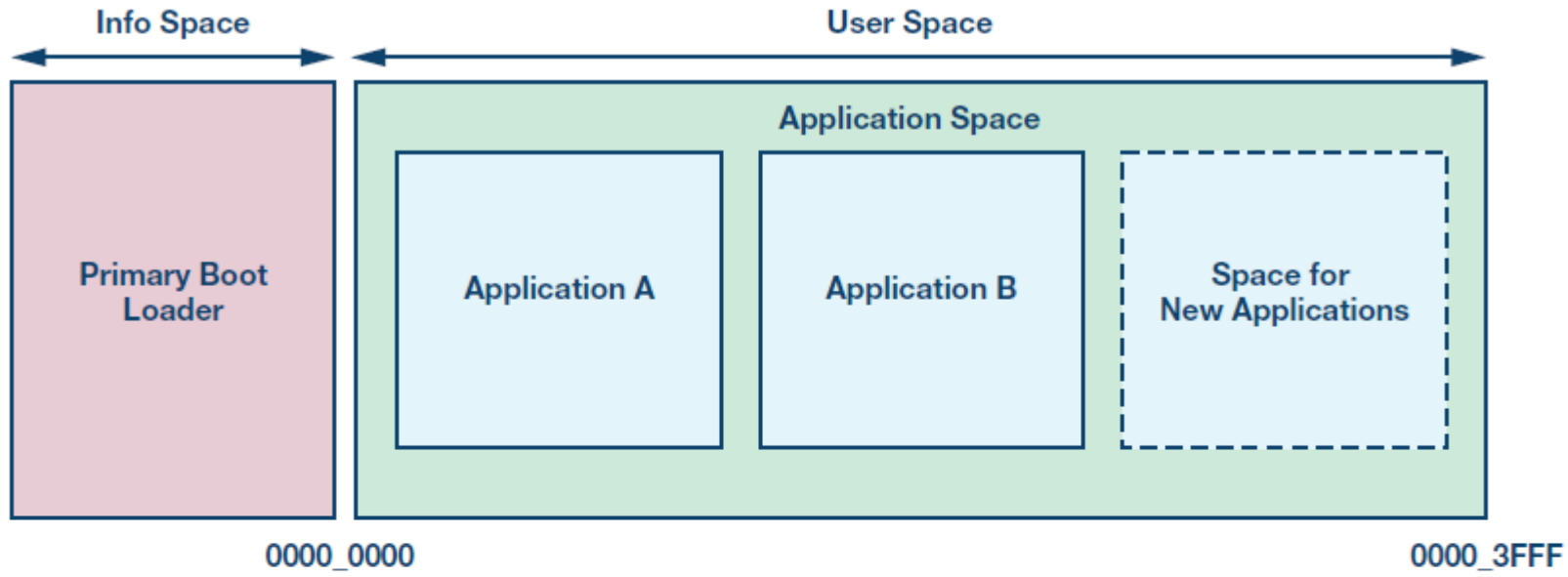


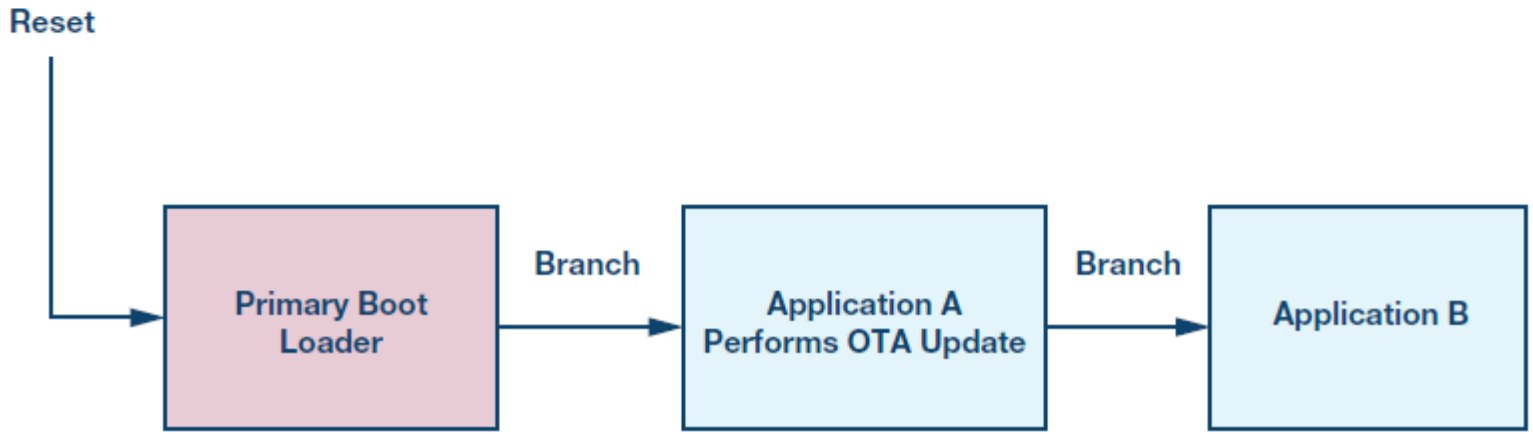


Lesson Learned: Always Have an SSBL

- Conceptually, it may seem simpler to omit the SSBL and place all the OTA update functionality into the user application, as it would allow an existing software framework, operating system, and device drivers to be seamlessly leveraged for the OTA process

- The memory map and boot sequence of a system that chose this approach is illustrated in Figure





Lesson Learned: Always Have a SSBL

Application A is the original application that is deployed on the microcontroller in the field

- This application contains the OTA update-related software, which is leveraged to download Application B when requested by the server
- After this download is complete and Application B has been verified, Application A will transfer control to Application B by performing a branch instruction to the reset handler of Application B
- The reset handler is a small piece of code that is the entry point of the software application and runs on reset. In this case, the reset is mimicked by performing a branch, which is equivalent to a function call

There are two major issues with this approach

POLITECNICO MILANO 1863

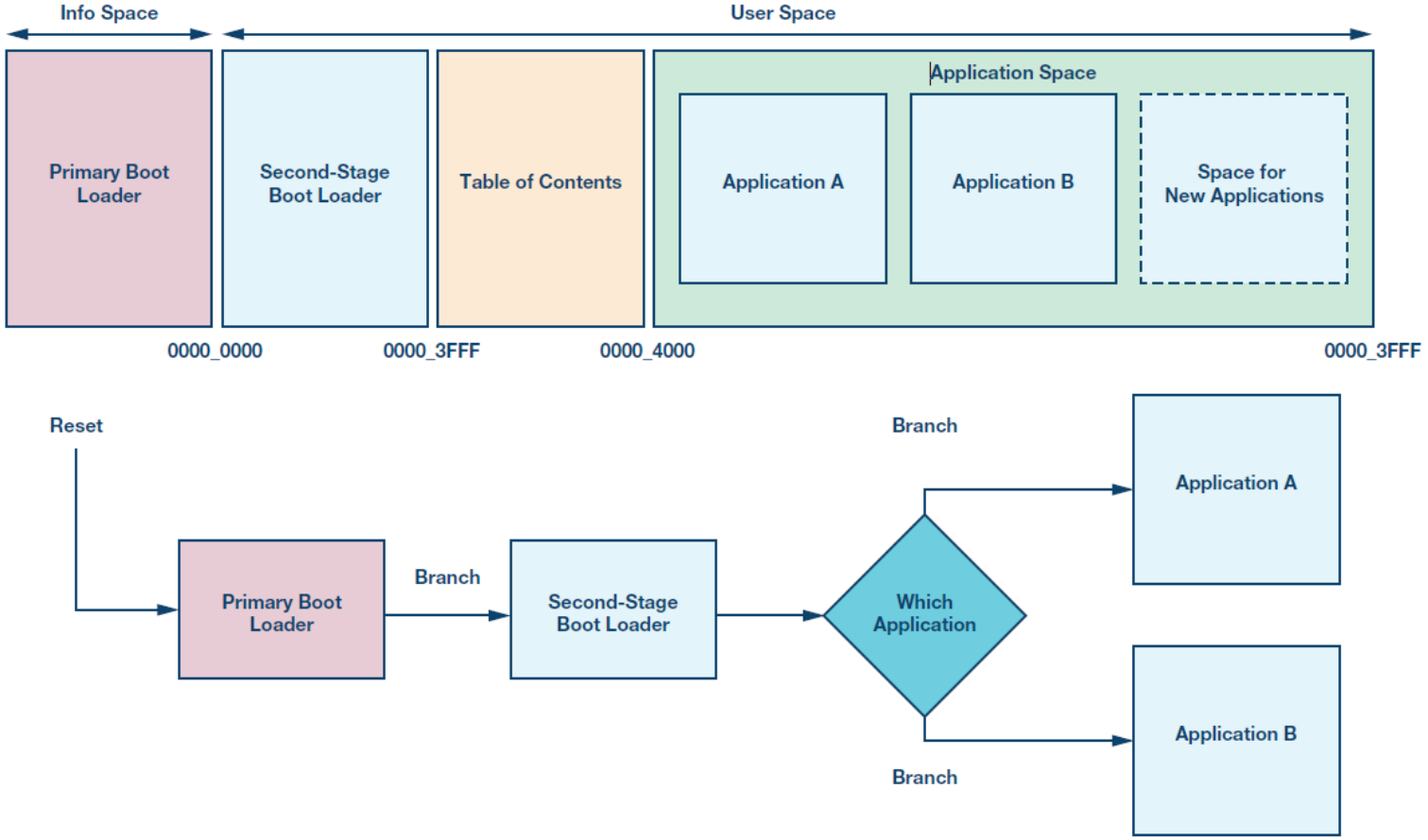
Major issues of not having a SSBL

- Many embedded applications employ a RTOS, which allows the software to be split into concurrent tasks, each with different responsibilities in the system
  - For instance, the application may have RTOS tasks for reading the sensor, running an algorithm on the sensor data, and interfacing with the radio. The RTOS itself is always active and is responsible for switching between these tasks based on asynchronous events or specific time-based delays
  - As a result, it is not safe to branch to a new program from an RTOS task, since other tasks will remain running in the background. The only safe way to terminate a program with a real-time operating system is through a reset
- Based on previous figure, a solution to the previous issue would be to have the primary boot loader branch to Application B instead of Application A
  - However, on some microcontrollers, the primary boot loader always runs the program that has its interrupt vector table (IVT), a key portion of the application that describes interrupt handling functions, located at address 0. This means that some form of IVT relocation is necessary to have a reset map to Application B
  - If a power cycle occurs during this IVT relocation, it could leave the system in a permanently broken state

Major issues of not having a SSBL

These issues are mitigated by having an SSBL fixed at address 0, as illustrated in Figure

- Since the SSBL is a non-RTOS program, it can safely branch to a new application
- There is no concern of a power cycle placing the system in a catastrophic state since the IVT of the SSBL at address 0 is never relocated



Design Trade-Off: The Role of the SSBL

What does this SSBL program do?

- At the bare minimum, the program must determine what the current application is (where it begins) and then branch to that address
- The location of the various applications in the microcontroller memory is generally kept in a table of contents (ToC). This is a shared region of persistent memory that both the SSBL and application software use to communicate with each other
- When the OTA update process completes, the ToC is updated with the new application information
- Portions of the OTA update functionality can also be pushed to the SSBL. Deciding what portions is an important design decision when developing OTA update software

POLITECNICO MILANO 1863

Design Trade-Off: The Role of the SSBL

The minimal SSBL described before is extremely simple, easy to verify, and most likely will not require modifications during the life of the application

- However, this means that each application must be responsible for downloading and verifying the next application. This can lead to code duplication in terms of the radio stack, device firmware, and OTA update software
- On the other hand, we can choose to push the entire OTA update process to the SSBL. In this scenario, applications simply set a flag in the ToC to request an update and then perform a reset. The SSBL then performs the download sequence and verification process
  - This will minimize code duplication and simplify the application specific software. However, this introduces a new challenge of potentially having to update the SSBL itself (that is, updating the update code)
- In the end, deciding what functionality to place in the SSBL will depend on the memory constraints of the client device, the similarity between downloaded applications, and the portability of the OTA update software

Design Trade-Off: Caching and Compression

Organization of the incoming application in memory during the OTA update

- The two types of memory that are typically found on a microcontroller are nonvolatile memory (for example, flash memory) and volatile memory (for example, SRAM)
- The flash memory will be used to store the program code and read-only data of an application, along with other system-level data such as the ToC and an event log
- The SRAM will be used to store modifiable portions of the software application, such as nonconstant global variables and the stack
- The software application binary illustrated in figure only contains the portion of the program that lives in nonvolatile memory. The application will initialize the portions that belong in volatile memory during a startup routine

POLITECNICO MILANO 1863

Design Trade-Off: Caching and Compression

- During the OTA update process, every time the client device receives a packet from the server containing a portion of the binary it will be stored in SRAM
- This packet could be either compressed or uncompressed
  - The benefit of compressing the application binary is that it will be smaller in size, allowing for fewer packets to be sent and less space needed in SRAM to store them during the download procedure
  - Sending data to a number of devices in the field can be costly on metered networks
  - The disadvantage of this approach is the extra processing time that the compression and decompression add to the update process, along with having to bundle compression related code in the OTA update software
- Since the new application software belongs in flash memory but arrives into SRAM during the update process, the OTA update software will need to perform a write to flash memory at some point during the update process
- Temporarily storing the new application in SRAM is called **caching**

POLITECNICO MILANO 1863

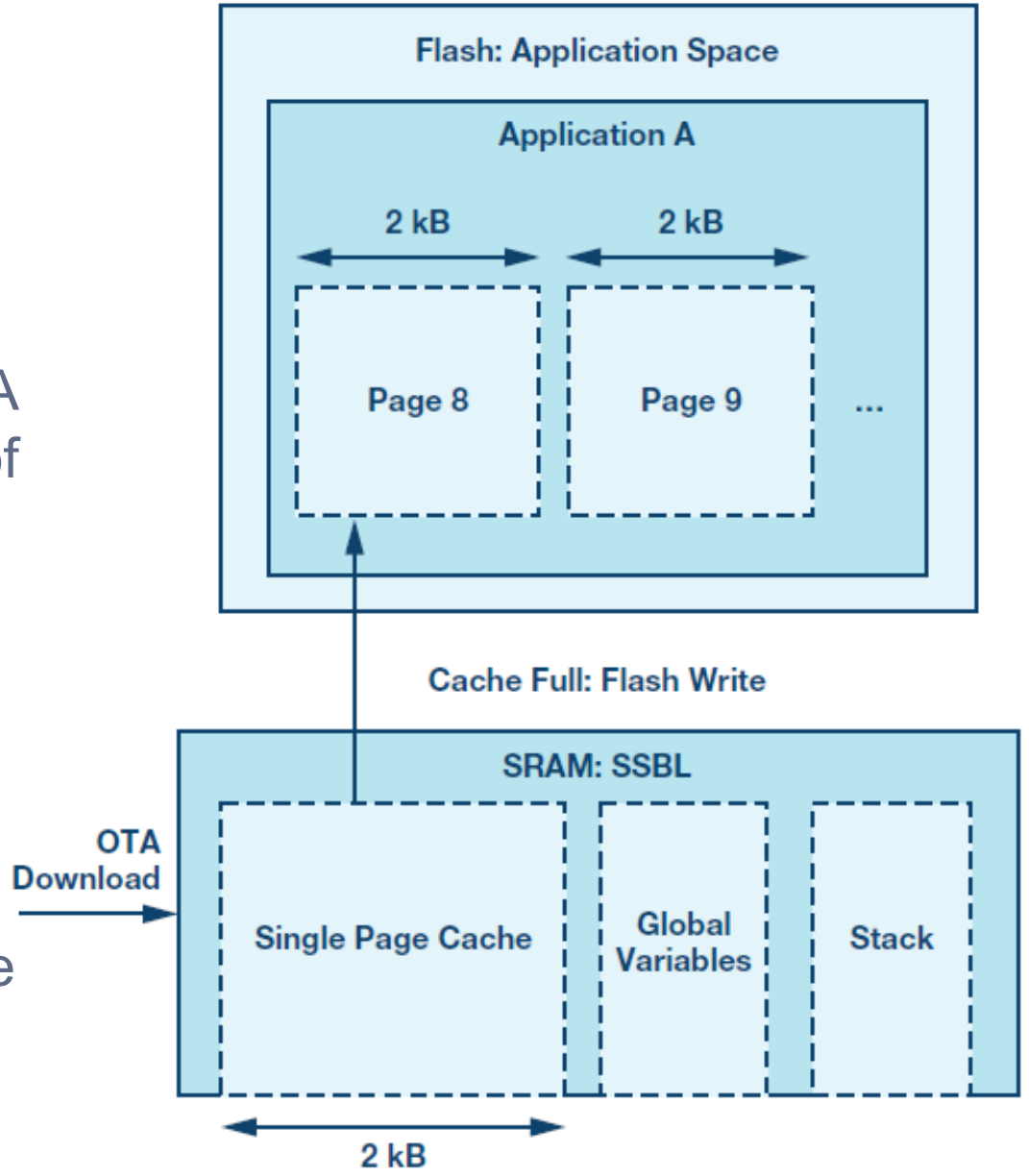
3 approaches of OTA update for caching

- **No caching**: Every time a packet arrives containing a portion of the new application, write it to its destination in flash memory
  - This scheme is extremely simple and will minimize the amount of logic in the OTA update software, but it requires that the region of flash memory for the new application is fully erased. This method wears down the flash memory and adds overhead
- **Partial caching**: Reserve a region of SRAM for caching, and when new packets arrive store them in that region
  - When the region fills up, empty it by writing the data to flash memory. This can get complex if packets arrive out of order or there are gaps in the new application binary, since a method of mapping SRAM addresses to flash addresses is required
  - One strategy is to have the cache act as a mirror of a portion of flash memory. Flash memory is divided into small regions known as pages, which are the smallest division for erasing. Because of this natural division, a good approach is to cache one page of flash memory in SRAM and when it fills up or the next packet belongs in a different page, flush the cache by writing that page flash memory

3 approaches of OTA update for caching

- The second scheme of **partial caching** during an OTA update is illustrated in figure
- The portion of flash memory for Application A is magnified and a functional memory map of the SRAM for the SSBL is illustrated
  - An example flash page size of 2 kB is shown

- Ultimately this design decision will be determined based on the size of the new application and the allowed complexity of the OTA update software



POLITECNICO MILANO 1863

3 approaches of OTA update for caching

- **Full caching**: Store the entire new application in SRAM during the OTA update process and only write it to flash memory when it has been fully downloaded from the server
  - This approach overcomes the shortcomings of the previous approaches by minimizing the number of writes to flash memory and avoiding complex caching logic in the OTA update software
  - However, this will place a limit on the size of the new application being downloaded, since the amount of available SRAM on the system is typically much smaller than the amount of available flash memory

POLITECNICO MILANO 1863

Design Trade-Off: Software vs. Protocol

The OTA update solution must also address security and communication

- Many systems have a communication protocol implemented in hw and sw for normal (non-OTA update related) system behavior like exchanging sensor data
  - This means that there is a method of (possibly secure) wireless communication already established between the server and the client
  - Communication protocols that an embedded system uses (e.g., Bluetooth Low Energy BLE or 6LoWPAN) could have support for security and data exchange that the OTA update software may be able to leverage
- How much abstraction is provided by the existing communication protocol?
  - The existing communication protocol has facilities for sending and receiving files between the server and client that the OTA update software can simply leverage for the download process -- OK
  - If the communication protocol is more primitive and only has facilities for sending raw data, the OTA update software may need to perform packetizing and provide metadata along with the new application binary. This also applies to the security challenges
  - The onus may be on the OTA update software to decrypt the bytes being sent over the air for confidentiality if the communication protocol does not support this

Design Trade-Off: Software vs. Protocol

In summary

- Building facilities like custom packet structure, server/client synchronization, encryption, and key exchange into the OTA update software will be determined based on what the system's communication protocol provides and what the requirements are for security and robustness
- Hard to provide a general solution, each system providers will propose a specific solution
  - See some links and material on BEEP

Security challenges

Needs

- Keep the new application sent over-the-air confidential, detect any corruption in the new application, and verify that the new application was sent from a trusted server as opposed to a malicious party

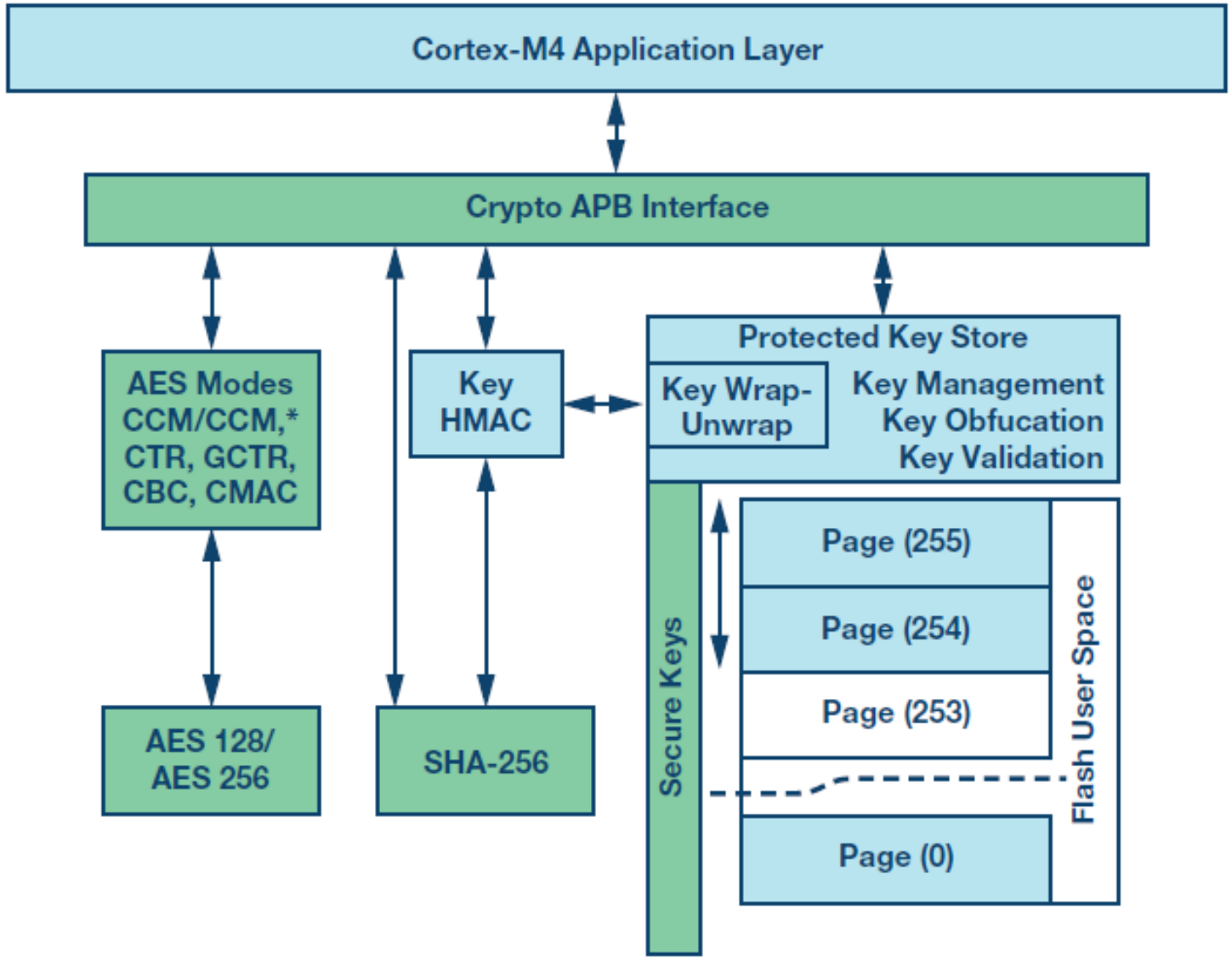
Solution: use of cryptographic (crypto) operations: encryption and hashing

- Encryption will use a shared key (password) between the client and server to obfuscate the data being sent wirelessly. A specific type of encryption that the microcontroller's crypto hardware accelerator may support is called AES-128 or AES-256, depending on the key size
- Along with the encrypted data, the server can send a digest to ensure that there is no corruption. The digest is generated by hashing the data packet—an irreversible mathematical function that generates a unique code
- If any part of the message or digest is modified after the server creates them, like a bit being flipped during wireless communication, the client will notice this modification when it performs the same hash function on the data packet and compares the digests

POLITECNICO MILANO 1863

Security challenges - example

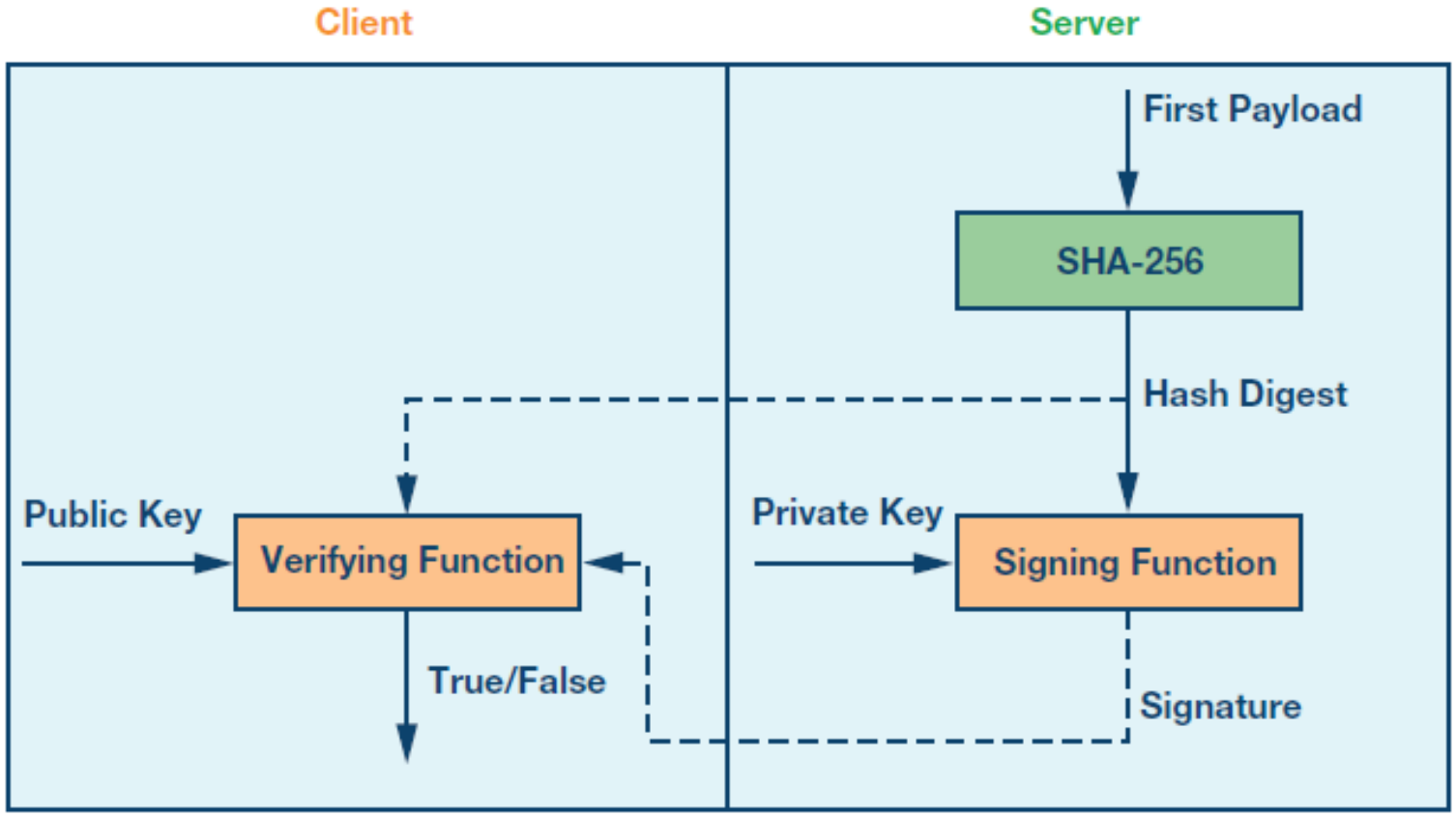
- A specific type of hashing that the microcontroller's crypto hardware accelerator may support is SHA-256
- Figure shows a block diagram of a crypto hardware peripheral in a microcontroller, with the OTA update software residing in the Cortex®-M4 application layer
- This figure also shows the support for protected key storage in the peripheral, which can be leveraged in the OTA update software solution to safely store the client's keys



Hardware block diagram of the crypto accelerator on the ADuCM4050

Use of use asymmetric encryption

- The server generates a public-private key pair
- The private key is known only by the server and the public key is known by the client
  - Using the private key, the server can generate a signature of a given block of data —like the digest of the packet that will be sent over the air. The signature is sent to the client, who can verify the signature using the public key
  - This enables the client to confirm the message was sent from the server and not a rogue third-party



Solid arrows are function input/output and dashed arrows are the information that is sent over the air

POLITECNICO MILANO 1863

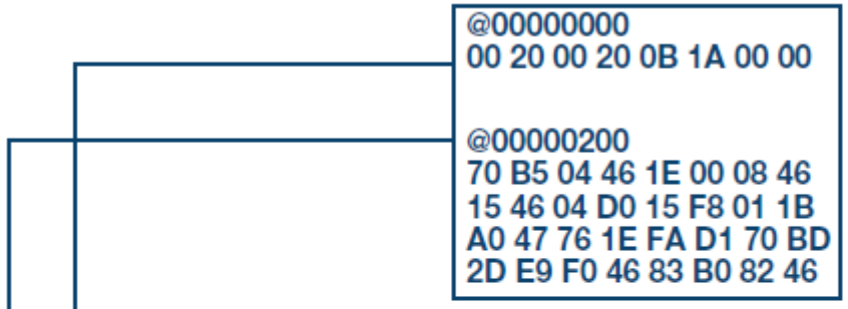
Asymmetric encryption vs microcontrollers

- Most microcontrollers do not have hardware accelerators for these asymmetric encryption operations, but they can be implemented using software libraries such as Micro-ECC, which specifically targets resource constrained devices
- The library requires a user-defined random number generating function, which can be implemented via the true random number generator hw peripheral on the micro
- While these asymmetric encryption operations solve the trust challenge during an OTA update, they are costly in terms of processing time and require a signature to be sent with the data, which increases packet sizes
- We could perform this check once at the end of the download, using a digest of the final packet or the digest of the entire new software application, but that would allow third-parties to download untrusted software to the client, which is not ideal
- Ideally, we want to verify every packet that we receive is from our trusted server without the overhead of a signature each time. This can be achieved using a hash chain

Security challenges

- A hash chain incorporates the cryptographic concepts into a series of packets to tie them together mathematically
  - The first packet (number 0) contains the digest of the next packet. Instead of the actual software application data, the payload of the first packet is the signature
  - The second packet (number 1) payload contains a portion of the binary, and the digest of the third packet (number 2)
  - The client verifies the signature in the first packet and caches the digest, H0, for later use. When the second packet arrives, the client hashes the payload and compares it to H0. If they match, the client can be sure that this subsequent packet was from the trusted server without all the overhead of doing a signature check

The expensive task of generating this chain is left to the server, and the client must simply cache and hash as each packet arrives to ensure packets arrive uncorrupted, with integrity and authentication



Number	Payload	Hash
0	P0 = Signature(H0, Public Key)	H0 = H(P1 + H1)
1	P1 = 00 20 00 20 0B 1A 00 00	H1 = H(P2 + H2)
2	P2 = 70 B5 04 46 1E 00 08 46	H2 = H(P3 + H3)
n	Pn = ...	Hn = H(Pn+1 + Hn+1)

To move deeper into the OTA world

More considerations for Updating the Bootloader Over-the-Air (OTA)

- https://www.embedded-computing.com/guest-blogs/considerations-for-updating-the-over-the-air-bootloader
- Discussion mainly regarding systems running Linux (on BEEP I left the printed version)

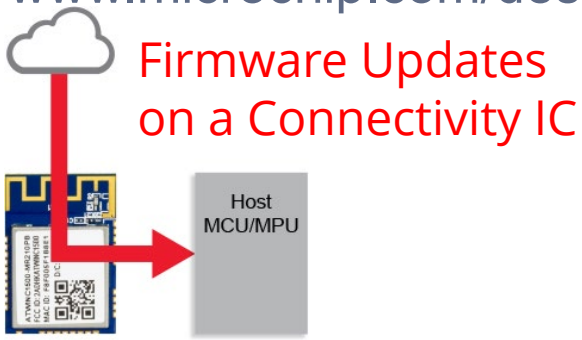
Analog Device ADiCM3029 or ADuCM4050 microcontrollers

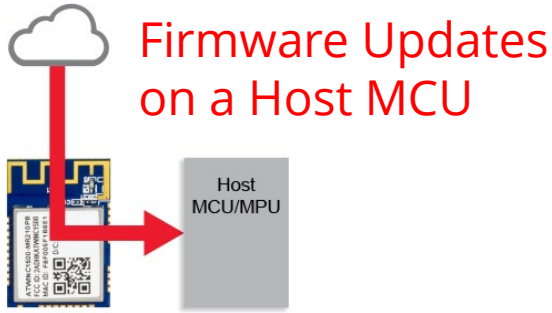
- https://www.analog.com/en/products/ADuCM3029.html
- These microcontrollers contain the hardware peripherals for OTA updates such as flash memory, SRAM, crypto accelerator, and a true random number generator

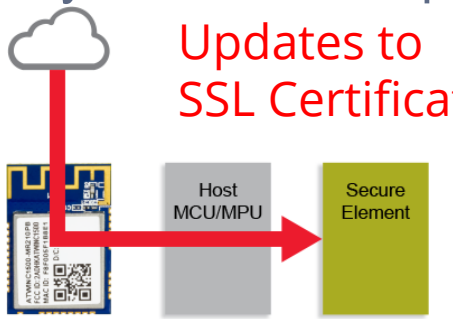
AMTEL Programming Guide for ATWINC1500 (on BEEP)

Microchip solutions – wide range for several scenarios

- https://www.microchip.com/design-centers/wireless-connectivity/over-the-air-updates







POLITECNICO
MILANO 1863

Summary of Best Practices

Best practices

Digital Signing

- Digital signatures ensure the authenticity of the image and integrity of the data in the image. Without a signature, there's no way to verify that the image came from an authentic source, and it could've come from a malicious third-party. A digital signature also ensures that the data within the image has not been modified (preserving integrity) and is intact/complete as it was generated at the source/author

Encryption

- Encrypting the image ensures the confidentiality of the data. This makes that no unauthorized parties are able to peek at the contents of the image and make sense of it. Only the end-device should be able to decrypt the image. This is especially true for wireless over-the-air updates since the data will be susceptible to third parties sniffing the communications channel used to transfer the image. Security measures also have to be in place to protect any decrypted version of the image that's stored locally on the end-device

Best practices

Communications channel protection

- In addition to taking the necessary security measures to protect the DFU image itself (at the source and destination), measures need to be taken to secure the communications channel over which the DFU image is transferred. Depending on the communication technology used, there are different configurations and implementations that would make sure the communications channel is secure. Examples include utilizing Bluetooth LE Secure Connections (which requires version 4.2 or later), TLS, etc.
- Keep in mind that communications channels are not always wireless and may involve intra-system transfers such as from an SoC to external flash. This could pose a security threat if a malicious party gets physical access to the end-device.
- Also the firmware servers can be attacked

Best practices

Versioning

- One of the security measures that need to be put in place is to make sure that a DFU update cannot be performed with an older version of the firmware. A versioning system must be put in place allowing only newer versions to be installed

Recoverability

- A process needs to be put in place to recover from failures in different stages of the DFU process. This includes handling scenarios such as loss of power, data corruption, manipulated DFU image, etc, and being able to roll back the firmware to the original image

Logging and status reporting

- It’s good practice to log operations and their statuses during DFU processes over the lifetime of a product. These logs could be reported to other devices within the system (such as a gateway, cloud server, etc.) for remote monitoring purposes. They could be useful for debugging purposes or for a better understanding of the causes of failures that may have occurred during a DFU

Best practices

Timely updates

- Ensuring timely updates can be critical for certain systems or in specific scenarios such as when a security vulnerability has been discovered. In this case, the manufacturer would want to make sure all devices in the field are updated as soon as possible

Minimize downtime

- Depending on the application, some system functionality may be critical and need to be available even during a DFU operation

User awareness

- It is good practice to make the user aware of an available update. This helps to make sure the user performs the update in close-to-optimal circumstances (e.g. connected to power, sufficient battery levels, minimal disruptions to functionality and usability of the end-device, etc.). It could also help to accelerate initiating the update process in cases where the update addresses vulnerabilities or software bugs (user is more incentivized to perform/allow the update if they're aware of issues that will be fixed by the update)

Best practices

Utilize a hardware-accelerated cryptoprocessor

- to speed up cryptographic operations (e.g. in time-critical applications, memory-constrained applications)

Integrate a hardware-accelerated cryptoprocessor

- into your system when feasible (e.g. in safety-critical applications, time-critical applications, memory-constrained applications)

Do not re-invent the wheel

- Many solutions and IDE already exists, select in a critical way what is part of the sw/hw companion of your microcontroller family
- Perform small-scale test before to trigger a widespread firmware update
  - It is not just a matter of updating the software, maybe that the new functionality is buggy