FIRMWARE MANAGEMENT FOR MCUS: THE QUARK BOOTLOADER APPROACH

Daniele Alessandrelli

OpenIoT Summit North America 2017

Goals of this talk

- Sharing our experience in developing a bootloader and a firmware management mechanism for MCUs
- Pointing other developers to open-source code they can reuse
- Collecting feedback and stimulating discussion

Outline

- Quark Bootloader Overview
- Firmware Management (FM) protocol stack
- Secure extension: authenticated firmware upgrades
- Internals: managing Bootloader Data (BL-Data)
- Concluding remarks

QUARK BOOTLOADER: OVERVIEW

The Quark Bootloader (aka QM-Bootloader)

- Reference bootloader for the Intel® Quark™ microcontroller family
  - Intel® Quark™ D2000 Microcontroller (D2000)
  - Intel® Quark™ SE Microcontroller C1000 (SE C1000)
- Developed as part of the Intel® Quark™ MCUs Software Stack
  - https://github.com/quark-mcu/
  - Originally integrated with the Intel® Quark™ Microcontroller Software Interface (**QMSI**)

QM-Bootloader: Features

- Bootstrap features
  - System initialization
  - Trim code computation
  - Restore context from sleep
- Security hardening features
  - Root of Trust (RoT) setup
- **Firmware Management functionality**
  - More details later...

Quark MCUs: Quick Overview

Quark D2000

- 1 processor core:
  - x86 (Lakemont) @ 32MHz
- SRAM
  - 8 kB
- Flash
  - 32kB + **8kB OTP** + 4kB data only
- Peripherals
  - **UART**, I2C, SPI, GPIOs, ADC, etc.

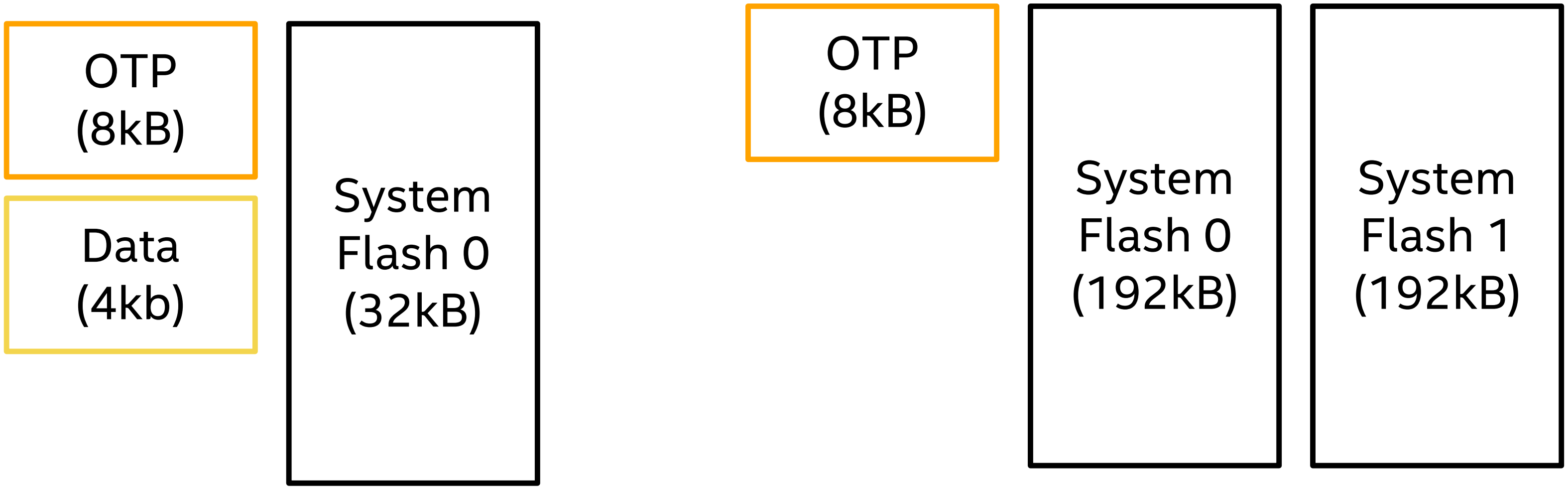
Quark SE C1000

- 2 processor cores:
  - x86 (Lakemont) @ 32 MHz
  - Sensor Subsystem (ARC) @ 32MHz
- SRAM
  - 80 kB
- Flash
  - 384 kB + **8 kB OTP**
- Peripherals
  - **UART**, I2C, SPI, **USB1.1**, GPIOs, ADC, etc.

Quark MCUs: Flash Layout

Quark D2000

Quark SE C1000



Firmware Management (FM) module

FM Features

- Multiple transports
  - UART and USB
- Firmware upgrades
  - Support for signed images
- Other FM functionality
  - Key management
  - System Information retrieval
  - Application erase

FM design goals

- Flash constraints
  - Secure FM over UART must fit in OTP (8kB)
- Modular design / code reuse
  - Both for target code and host tools
- Extensibility
  - Allow for other transport to be easily supported

FIRMWARE MANAGEMENT (FM): PROTOCOL STACK

FM Protocol Stack: Overview

DFU-based Firmware Management

- DFU is used for sending images and commands to the device
- The QDA protocol has been defined to enable DFU-over-UART

QFU image format, block-wise format designed to

- Work with generic DFU tools (e.g., dfu-util)
- Support firmware authentication

QFM protocol, enabling DFU to be used also for FM operations other than firmware upgrades

- Application erase
- System/Firmware information retrieval
- Key provisioning

Layer	USB mode	UART mode
DFU payload	Quark Firmware Management (QFM) Protocol / Quark Firmware Update (QFU) Format	
DFU flavor	USB/DFU	Quark DFU Adaptation (QDA) Protocol
Transport	USB	XMODEM-CRC
Driver	USB device driver	UART driver

(USB) DFU: Quick Introduction

- DFU: Device Firmware Upgrade
- Standard for performing firmware upgrades over USB
- **DFU does not define any specific image format**
  - (but it specifies a DFU file suffix, useful only to the DFU host tool, which strips it off before downloading the image to the device)

- DFU provides two main functions:
  - **DFU_DNLOAD**: to transfer (download) data to the device
    - Used for FW upgrades
  - DFU_UPLOAD: to transfer (upload) data from the device
    - Used for FW extractions
- Both transfers are **block-based**
  - (all blocks, except the last one, must use the same block size)

Why DFU?

- Open, well-documented standard
  - Already used by many embedded devices
- Designed for resource-constrained devices
  - Block-wise transfer/flashing
  - Transmission flow controlled by the device
- Reusing existing host tools
  - dfu-util (GPLv2)
- No constrains on image format
  - We wanted to add our own metadata and authentication mechanism

FM Protocol Stack: DFU over UART

DFU is extended to UART by means of:

- The **Quark DFU Adaptation Protocol (QDA)**
  - Makes DFU functionality available over message-oriented transports (other than USB)
- The **XMODEM-CRC** protocol
  - Old file transfer protocol
  - Used to transport QDA packets
  - Chosen for its simplicity

Layer	USB mode	UART mode
DFU payload	Quark Firmware Management (QFM) Protocol / Quark Firmware Update (QFU) Format	
DFU flavor	USB/DFU	Quark DFU Adaptation (QDA) Protocol
Transport	USB	XMODEM-CRC
Driver	USB device driver	UART driver

QDA: Quark DFU Adaptation layer

Provide all DFU request/response messages

- DFU_DETACH
- **DFU_DNLOAD**
- **DFU_UPLOAD**
- DFU_GETSTATUS (used for flow control during downloads)
- DFU_CLRSTATUS (exit from error)
- DFU_ABORT (abort download/upload)
- DFU_GETSTATE

Mimic (some) generic USB functionality on which DFU relies

- Get device/configuration/interface descriptors
- Set active **alternate settings**

QDA usage is not limited to XMODEM/UART, but it could be used with any message-oriented protocol (e.g., UDP)

QDA: qm-dfu-util (aka dfu-util-qda)

- QDA/UART support on the host side was also needed
- dfu-util is a well-known host-side tool for USB/DFU
    - Open-source (GPLv2)
        - http://dfu-util.sourceforge.net/
    - Multi-platform (Windows/Linux)
- We forked it, creating qm-dfu-util
    - The USB layer (libusb) is replaced with a QDA/UART layer

FM Protocol Stack: Our DFU payload

Thanks to USB/DFU and QDA we have a common (DFU-based) communication layer.

On top of it we can transfer:

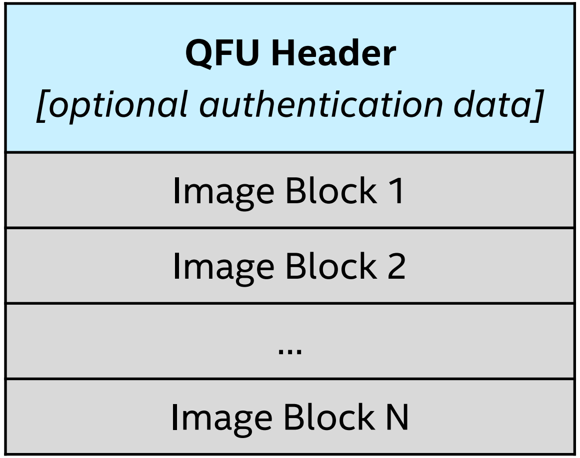
- Upgrade images
  - In the QFU format
- Other FM requests
  - Using the QFM protocol

Layer	USB mode	UART mode
DFU payload	Quark Firmware Management (QFM) Protocol / Quark Firmware Update (QFU) Format	
DFU flavor	USB/DFU	Quark DFU Adaptation (QDA) Protocol
Transport	USB	XMODEM-CRC
Driver	USB device driver	UART driver

QFU Image Format: Overview

Block-wise format:

- QFU images are divided in blocks of the same size (with the exception of the last one)
  - 1st block: header
  - Following blocks: raw firmware image (binary)
- Each block must be transferred in a single DFU DNLOAD request
  - i.e., DFU tools must use the same block-size of the image (specified in the header)



The DFU suffix is not shown since it is not processed by the device.

QFU Image Format: Header

First-level header

- Containing common information for processing the image

Can be followed by an extended header

- Containing information for image verification / authentication

Block size is fixed to 2kB / 4kB (multiple of page size) in the current implementation

- For code / footprint optimization reasons

"QFUH" *(Magic; 4 bytes)*	
vid *(Vendor ID; hex16)*	**pid** *(Product ID; hex16)*
pid_dfu *(Product ID DFU; hex16)*	**part_num** *(Partition number; uint16)*
app_version *(Application version; hex32 – vendor specific)*	
blk_size = 2kB/4kB *(Block size; uint16)*	**blk_cnt** *(Total block number, incl. header; uint16)*
ext_hdr *(Extended header type; 2 bytes)*	**rsvd** *(reserved; 2 bytes)*
<Extended header content or zeroed padding>	

QFU Image Format: Partitions

Flash divided in partitions

- No explicit memory addresses

Current partition scheme

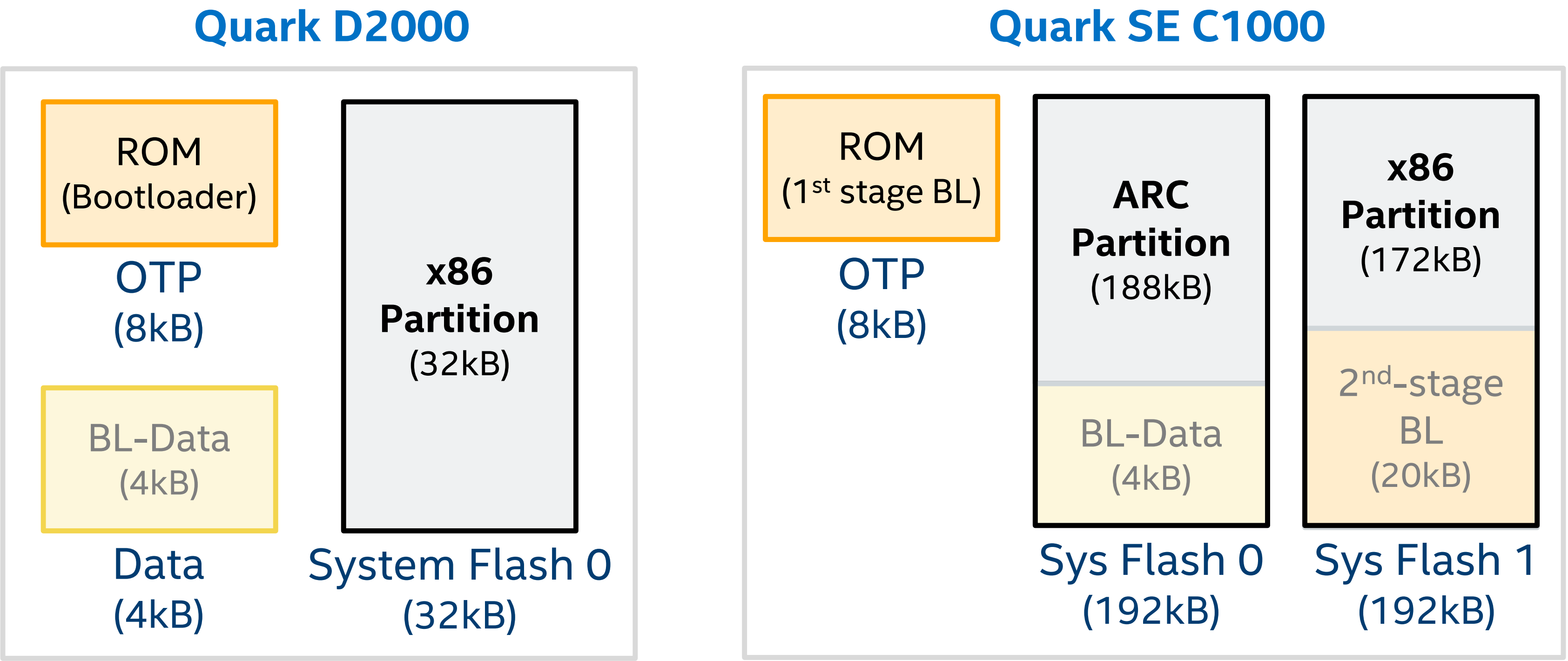
- Quark D2000
  - 1 partition (for x86)
- Quark SE C1000
  - 2 partitions (one for x86, one for ARC)

Other partition scheme are possible

- Including multiple partitions per core

"QFUH" *(Magic; 4 bytes)*	
vid *(Vendor ID; hex16)*	**pid** *(Product ID; hex16)*
pid_dfu *(Product ID DFU; hex16)*	**part_num** *(Partition number; uint16)*
app_version *(Application version; hex32 – vendor specific)*	
blk_size = 2kB/4kB *(Block size; uint16)*	**blk_cnt** *(Total block number, incl. header; uint16)*
ext_hdr *(Extended header type; 2 bytes)*	**rsvd** *(reserved; 2 bytes)*
<Extended header content or zeroed padding>	

Flash layout: Application partitions



QFM Protocol: Overview

Providing extended-FM over DFU:

- **Application erase**
  - Delete all application code
    - (from every partition)
- **Information retrieval**
  - Provide info about device's HW, SW, and configuration
    - E.g., available partitions, bootloader version, application version, etc.
- **Key provisioning**
  - (it will be available in QMSI 1.4)

Layer	USB mode	UART mode
DFU payload	**Quark Firmware Management (QFM) Protocol /** Quark Firmware Update (QFU) Format	
DFU flavor	USB/DFU	Quark DFU Adaptation (QDA) Protocol
Transport	USB	XMODEM-CRC
Driver	USB device driver	UART driver

QFM Protocol: Packets

Requests:

1. QFM_APP_ERASE
2. QFM_SYS_INFO_REQ
3. QFM_UPDATE_KEY

Requests are sent using ***DFU_DNLOAD transactions***

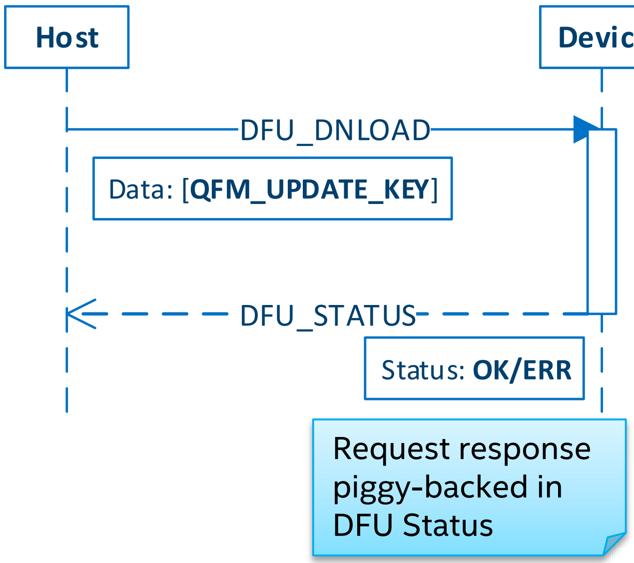
Responses:

1. QFM_SYS_INFO_RESP

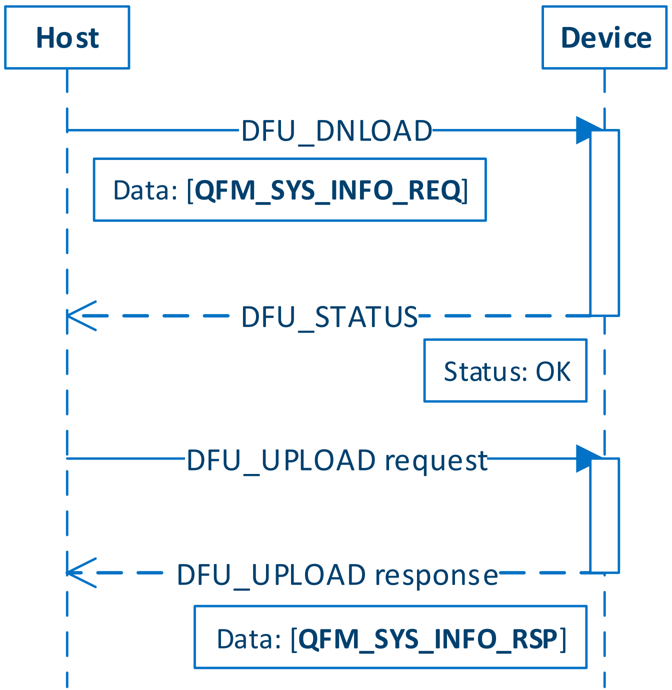
Responses are sent using ***DFU_UPLOAD transactions***

QFM Protocol: Examples

Key Update



System Information Retrieval



FM Protocol Stack: QFM / QFU selection

Different **DFU alternate settings** used to switch between QFM and QFU:

- Alt-Setting 0 is for extended-FM
  - DFU used to exchange QFM packets
- Alt-Settings 1+ are for FW upgrades
  - DFU used to transfer QFU images
  - Each alt-setting identifies a specific partition

Layer	USB mode	UART mode
DFU payload	Quark Firmware Management (QFM) Protocol / Quark Firmware Update (QFU) Format	
DFU flavor	USB/DFU	Quark DFU Adaptation (QDA) Protocol
Transport	USB	XMODEM-CRC
Driver	USB device driver	UART driver

```
$ dfu-util -l
Found DFU: [8086:c100] ver=0100, [...], alt=2, name="Partition2 (ARC)", serial="00.01"
Found DFU: [8086:c100] ver=0100, [...], alt=1, name="Partition1 (LMT)", serial="00.01"
Found DFU: [8086:c100] ver=0100, [...], alt=0, name="QFM", serial="00.01"
```

QFU / QFM: Host Tools

QFU image creator

- `qm_make_dfu.py`
- Converts a raw binary into a QFU/DFU image
  - Adds the QFU header
  - Adds the DFU suffix
- The image must be flashed separately
  - Using a DFU tool
    - (dfu-util / qm-dfu-util)

QFM utility

- qm_manage.py
- Enables QFM functionality
  - Info retrieval
  - Application erase
  - Key provisioning
- DFU tools are called directly
  - To send QFM requests and collect QFM responses

Example: Create QFU image and perform upgrade

1. Build the binary
2. Create a QFU image
   - Using the qm_make_dfu.py python script

```
$ qm_make_dfu.py release/quark_se/x86/bin/blinky.bin -p 1 --app-version 42
```

3. Enter FM mode
   - Ground FM pin and reset the board
     - (not needed if a USB/DFU application is running)
4. Flash via dfu-util
   - Using either the original dfu-util (for USB) or qm-dfu-util (for UART)

```
$ dfu-util -D release/quark_se/x86/bin/blinky.bin.dfu -a 1
```

Example: Using QFM services

Info retrieval

```
$ qm_manage.py info -d <vid>:<pid>
```

```
Version   : 1.4.0
SoC Type  : Quark SE
Auth.     : NONE
Target 00 : x86 (running application on partition 0)
Target 01 : sensor (running application on partition 1)
Part.  00 : App Version 42
Part.  01 : No application installed
```

Application erase

```
$ qm_manage.py erase -d <vid>:<pid>
```

Key provisioning

```
$ qm_manage.py [set-rv-key | set-fw-key] <key-file> -d <vid>:<pid>
```

SECURE FIRMWARE UPGRADE

Secure FW Upgrade Feature: Overview

What is provided
(in forthcoming 1.4 release)

- Authenticated firmware upgrades
  - Symmetric-key scheme
    - HMAC256 authentication
- Key management
  - First-time provisioning and subsequent updates
  - Relaying on an additional key

What is not provided

- Encryption
  - Of the image
  - Of key update request
- Image verification at boot
  - Not difficult to implement though
  - Excluded to minimize boot time

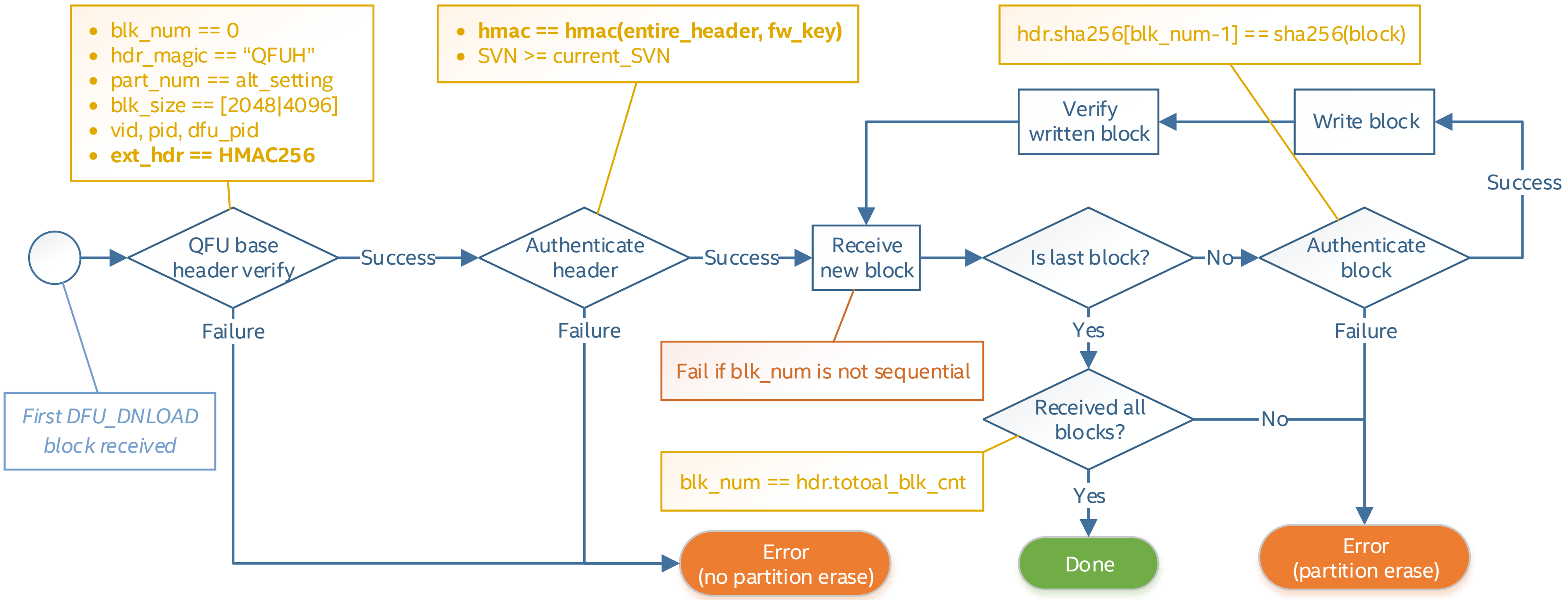
Secure FW Upgrade: QFU extension

The QFU header is extended with an **HMAC extended header**

- Containing all the information needed to authenticate the image
  - A list of block hashes
    - One for blocks
  - An HMAC digest authenticating the entire header
    - Including all the hashes
- (also containing a Security Version Number – SVN)

"QFUH"	
vid	pid
pid_dfu	part_num
app_version	
blk_size = 2kB	**total_blk_cnt**
ext_hdr = HMAC256	rsvd
svn *(security version number; 4 bytes)*	
blk_sha256[0] … **blk_sha256**[*data_blk_cnt* - 1] *(per-block SHA256 hashes; 32*data_blk_cnt bytes)*	
hmac256 *(HMAC256 of the whole header; 32 bytes)*	

Secure FW Upgrade: Upgrade flow



Secure FW Upgrade: Ensuring partition consistency

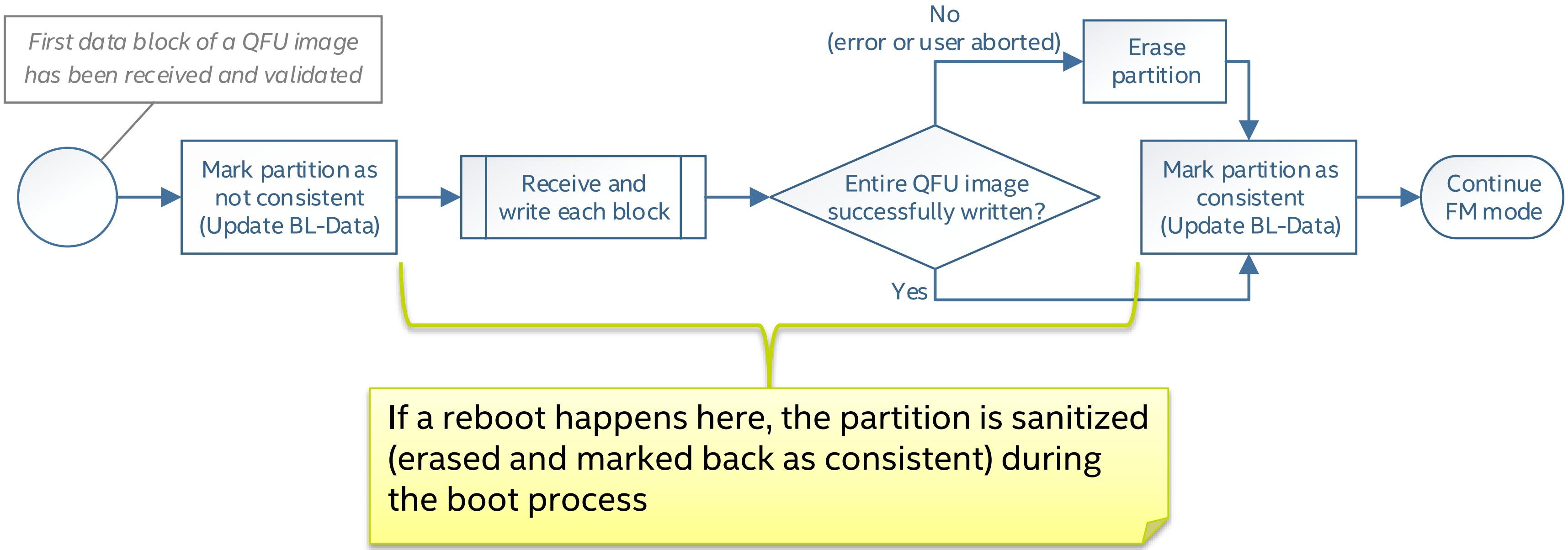
Problem:

- Unhandled failures (e.g., resets) can leave partitions in an inconsistent state

Solution:

- Associate a **consistency flag** to every partition
  - Stored in persistent Bootloader Data (BL-Data)
- Change consistency flag during FW upgrades
  - Before starting the upgrade, mark partition as inconsistent
  - When upgrade is complete, mark partition as consistent
- Sanitize partitions at every boot
  - Look for inconsistent partitions and delete them

Secure FW Upgrade: Consistency flag and upgrade flow



Key Management: Provisioning/update mechanism

- Both first time provisioning and subsequent updates are supported
- The key is sent to the device with a special key-update request
  - Extension of the QFM protocol
- The request and the new key are authenticated with two keys (**double signing**):
  - the old **firmware key** and
  - an additional key, the **revocation key**
- The key-update request is not encrypted
  - Since at the moment only wired and point-to-point transport (i.e., UART and USB) are supported

Key Management: Revocation and firmware keys

The **firmware key** is used for authenticating both key-update request and upgrade images.

The **revocation key** is used only for authenticating key-update requests.

The revocation key can be updated too:

- The same key update request is used
- The request is authenticated with the current firmware key and the old provisioning key

Key Management: First-time provisioning

In un-provisioned devices both keys have the same default (‘magic’) value.

First-time provisioning sequence:

1. Provide the revocation key
    - Signing it with the magic key twice
2. Provide the firmware key
    - Signing it with the magic key (in place of the old firmware key) and the revocation key

Key provisioning enforcement:

- Firmware upgrades are enabled only if the firmware key is set
- The firmware key can be set only after the revocation key

Key Management: Key-update QFM packets

Revocation key update

qfm_pkt_type = QFM_UPDATE_RV_KEY [QFM packet type, 4 bytes]
key *[256-bit new revocation key, 32 bytes]*
hmac256 *[HMAC256 signature of all the previous, done with the FW key and the old revocation key]*

Firmware key update

qfm_pkt_type = QFM_UPDATE_FW_KEY [QFM packet type, 4 bytes]
key *[256-bit new firmware key, 32 bytes]*
hmac256 *[HMAC256 signature of all the previous, done with the old FW key and the revocation key]*

Same algorithm:
HMAC(HMAC(packet, *current_fw_key*), *current_rv_key*)

BOOTLOADER DATA (BL-DATA)

Persistent Bootloader Data (BL-Data)

In order to enable Firmware Management (FM), the bootloader needs to store and maintain some (meta-)data

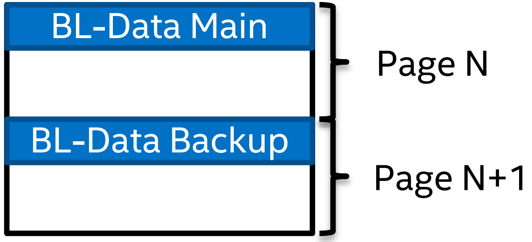
- Application version, partition consistency, etc.
- **Authentication keys**

BL-Data management must be **resilient to update failures** and possible attacks.

Resilience is achieved with:

- BL-Data duplication (backup copy)
- Verification at each boot (sanitization)

BL-Data: Duplication



Two identical copies of BL-Data are maintained:

- BL-Data Main
- BL-Data Backup

Each copy has a **CRC to verify** its **integrity**

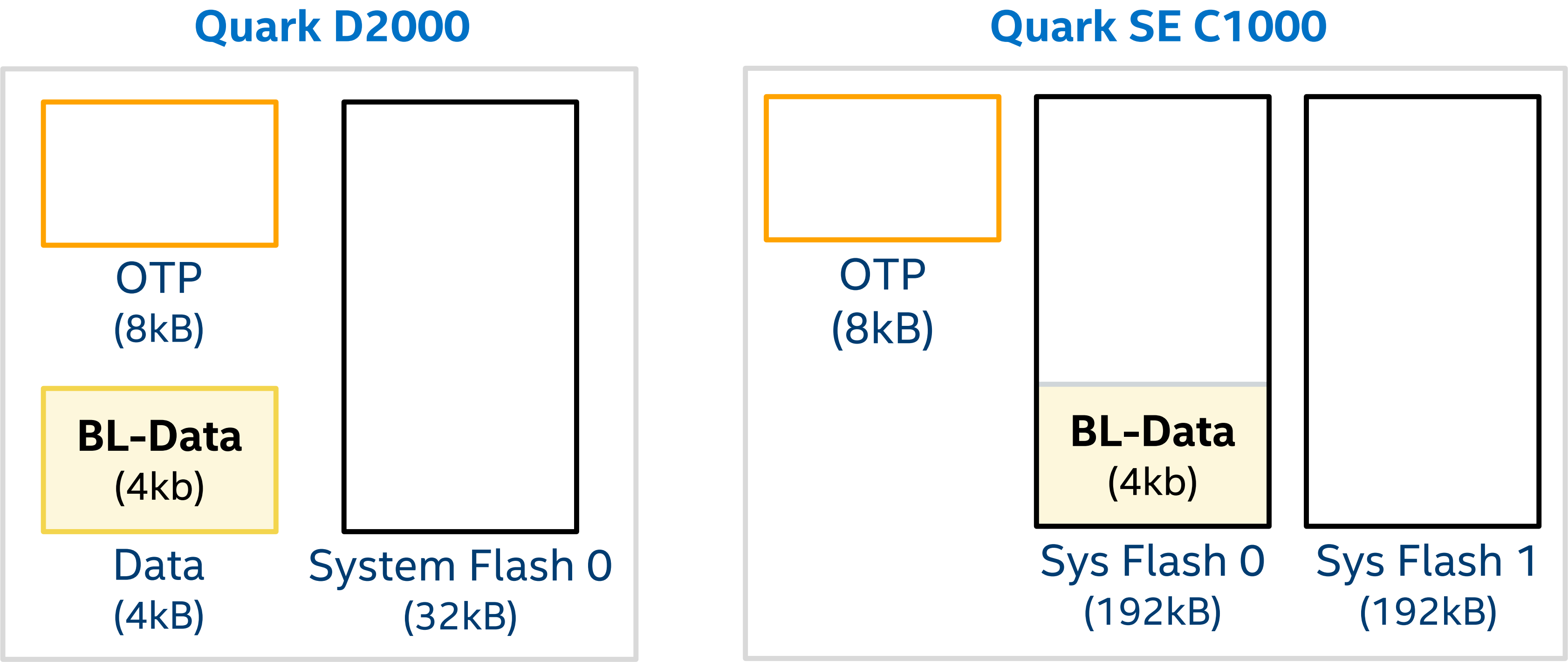
Copies are stored in **different flash pages**

- Since a flash update requires the entire page to be deleted and then rewritten

When BL-Data is changed, copies are **updated always in the same order**

- First BL-Data Main, then BL-Data Backup

BL-Data: Flash location

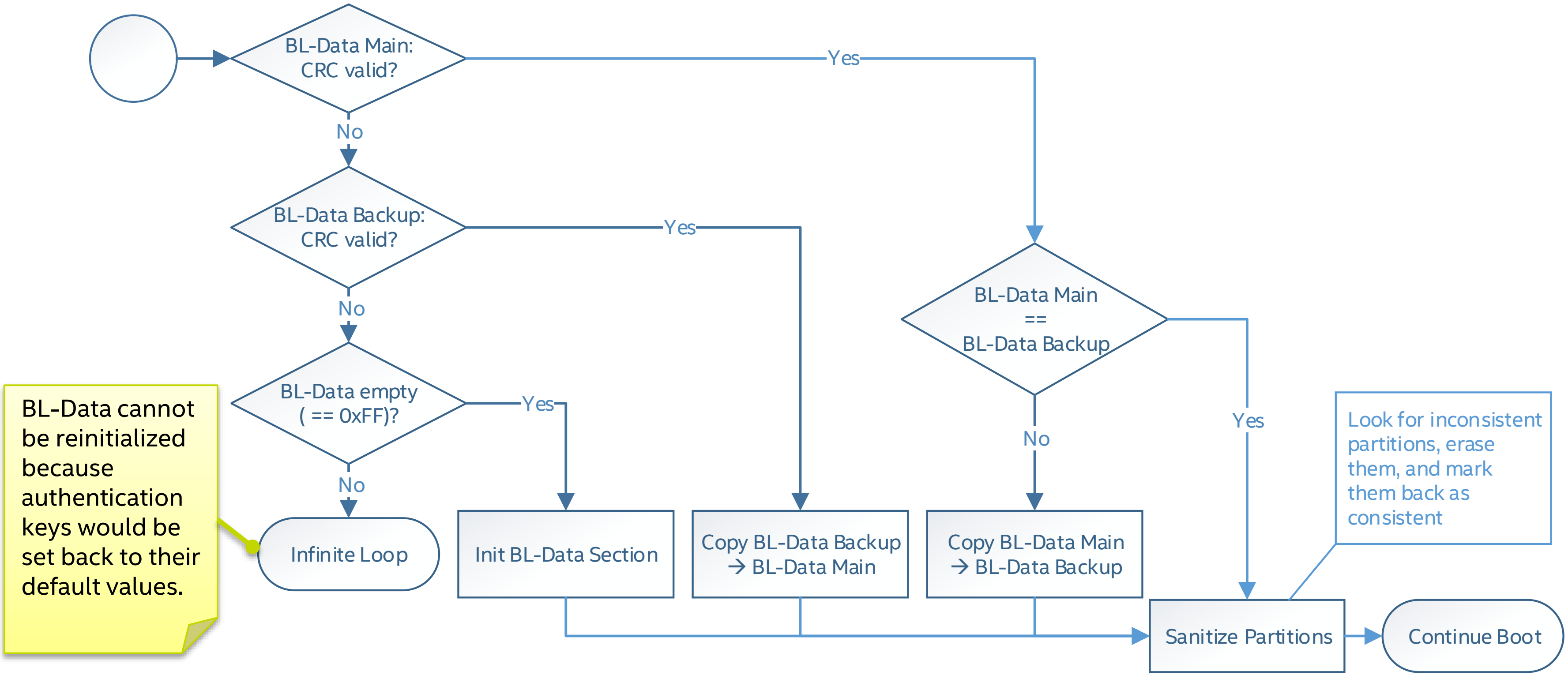


BL-Data: Verification flow

At every boot, BL-Data is verified to detect special conditions requiring fixing:

- **Lack of initialization.**
  - BL-Data Flash Section is blank and BL-Data (both copies) need to be initialized
- **Single BL-Data Copy corrupted or missing**
  - An unhandled failure (e.g., a power loss) has happened during an update
  - The other BL-Data copy contains the latest valid information and must be copied over the corrupted one
- **Both BL-Data copies corrupted**
  - Some critical error has happened (hardware fault or security attack)
  - This is an unrecoverable situation: enter infinite loop (customer return needed)

BL-Data: Verification flow



BL-Data: Content

Bootloader data

trim_codes	*Shadowed trim codes*
partitions[N_PARTS]	*Partition descriptors*
targets[N_TARGETS]	*Target descriptors*
fw_key	*Firmware key*
rv_key	*Revocation key*
crc	*CRC of all of the above*

Partition descriptor

target_idx	*The index of target (core) associated with the partition*
is_consistent	*Consistency flag*
app_version	*The version of the application installed in the partition*
<other>	*Misc information about the structure of the partition (starting address, size, etc.)*

Target descriptor

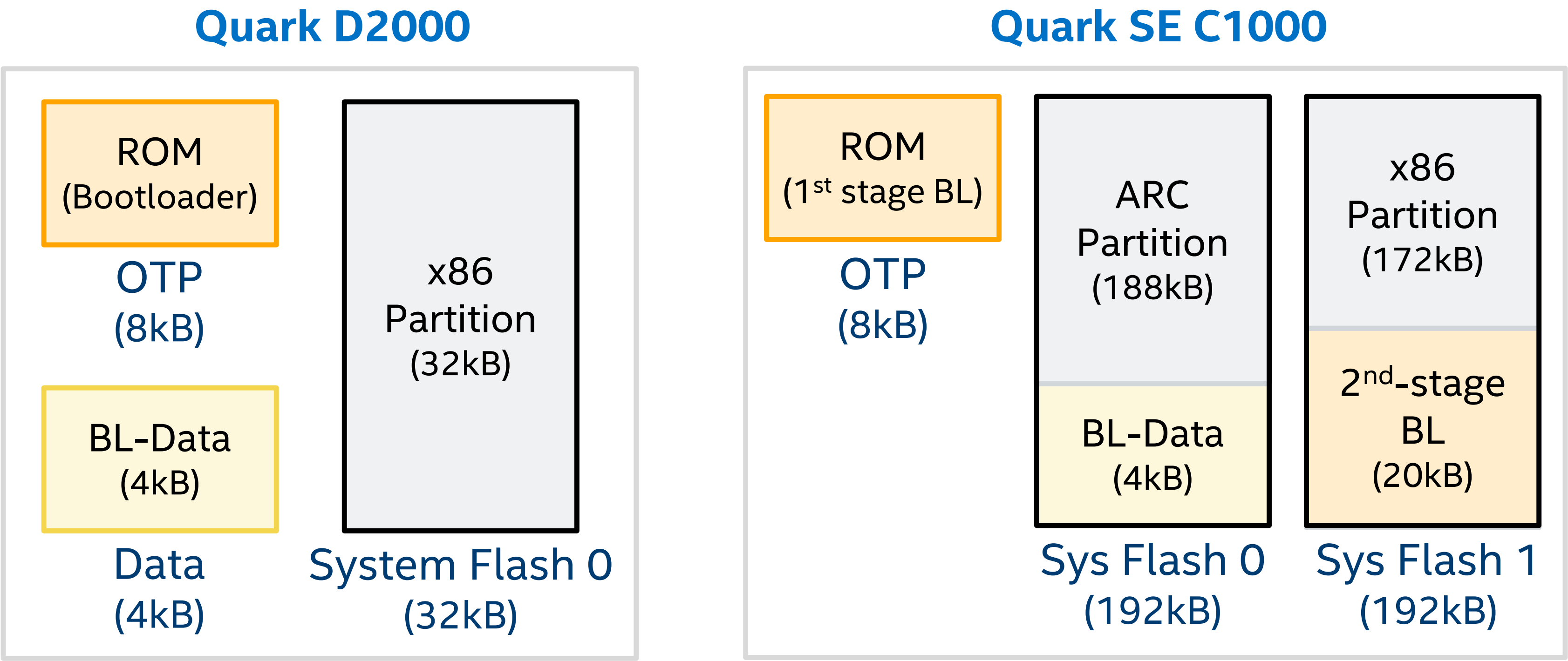
active_part_idx	*The index of the active partition for this target*
svn	*The SVN associated with this target*

BL-Data: Partitions and targets

A **partition** is a portion of flash designed to host an application. A partition is associated with a **target**, i.e., the core that will run the hosted application.

- We currently support only one partition per target
  - On Quark D2000 we have only one target / partition (x86 partition)
  - On Quark SE C1000 we have two targets / partitions (x86 and ARC partitions)
- But the **design allows for multiple partitions per target**
  - Possible use case: fallback partition in case of failed OTA updates
- **External targets / partitions** are also envisioned
  - Associated with board peripherals such as a BLE module

Flash layout



CONCLUDING REMARKS

Reusable software components

- DFU state machine
  - Completely independent from the lower-level communication stack
- QDA (DFU over UART)
  - Not just `qm-dfu-util`, but the device-side code as well
- XMODEM
  - You just have to define your own `getc`/`putc` functions
- QFM/QFU host tools
  - (device-side components are more dependent on QMSI API)

```
fw-manager/
├── dfu
│   ├── core
│   │   ├── dfu_core.c
│   │   └── dfu_core.h
│   ├── dfu.h
│   ├── qda
│   │   ├── qda.c
│   │   ├── qda.h
│   │   ├── qda_packets.h
│   │   ├── xmodem.c
│   │   ├── xmodem.h
│   │   ├── xmodem_io.h
│   │   ├── xmodem_io_uart.c
│   │   └── xmodem_io_uart.h
│   └── usb-dfu
[...]
```

https://github.com/quark-mcu/qm-bootloader

Some lessons learnt

- Modular approach pays back in embedded as well
  - Easier to adapt to changing requirements
  - Some code reused also for host-tools (XMODEM/QDA)
  - Code better validated (e.g., DFU state machine used twice)
- Reuse existing open-source code
  - dfu-util, TinyCrypt, etc.
- LTO offsets most of the overhead of the modular approach
  - 15%-20% flash saving
  - But it complicates debugging
- When dealing with flash layouts use linker script symbols
  - Especially if they are logical layouts
    - App partitions, bl-data section, 2nd-stage
  - And don't be afraid of using the INCLUDE directive

Thank you!

Any questions?