# AN4767
# Application note

# On-the-fly firmware update for dual bank STM32 microcontrollers

## Introduction

Dual bank functionality is a feature common to several STM32 microcontrollers. The goal of this document is to describe how to take advantage of this feature in customer applications.

The main topic of this application note is the live upgrade, covered by the X-CUBE-DBFU STM32Cube Expansion Package.

The main advantage of the live update is the minimization of the downtime during the transition phase, even demanding real time tasks can be executed during the update.

Cat.5 devices of the STM32L0 Series, access line and USB OTG devices of the STM32L4 Series and Cat.3 devices of the STM32G4 Series are directly addressed in this document, but other STM32 microcontrollers with two semi-independent banks of memory share some of the described properties and can be used in a similar way.

The following documents, all available on *www.st.com*, are considered as reference:

- Reference manual RM0367: “Ultra-low-power STM32L0x3 advanced ARM®-based 32-bit MCUs”
- Reference manual RM0351: “STM32L4x5 and STM32L4x6 advanced Arm®-based 32-bit MCUs””
- Reference manual RM0440: “STM32G4xx advanced Arm®-based 32-bit MCUs”
- Application note AN2606: “STM32 microcontroller system memory boot mode”
- Application note AN4024: “STM32 secure firmware upgrade (SFU)”
- Application note AN4657: “STM32L0 in-application programming using UART”
- Application note AN4894: "EEPROM emulation techniques and software”

# Contents

# List of tables

# List of figures

# 1 Definitions

**Table 1. List of acronyms**

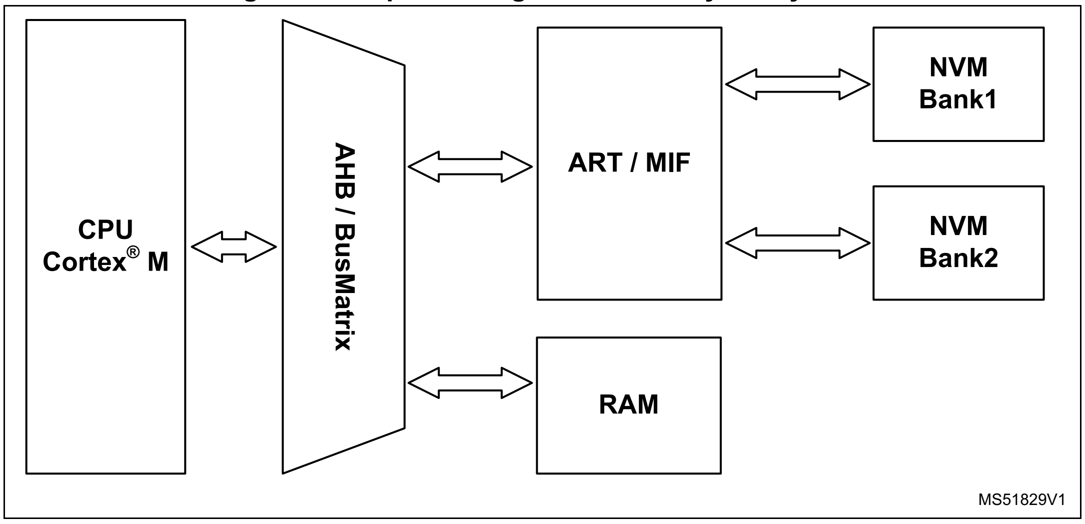
| Term | Description |
|---|---|
| CPU | Central processing unit (part of the MCU) |
| EEPROM | Electrically erasable programmable read only memory |
| ER | Execution region (linker definition) |
| IAP | In-application programming |
| ISR | Interrupt service request |
| IVT | Interrupt vector table |
| LR | Load region |
| MCU | Microcontroller |
| NMI | Non-maskable interrupt |
| NVIC | Nested vector interrupt controller |
| NVM | Non-volatile memory (EEPROM or Flash) |
| UART | Universal asynchronous receiver transmitter |
| USART | Universal synchronous and asynchronous receiver transmitter |
| VTOR | Vector table offset register |

# 2 Memory subsystem summary

STM32 microcontrollers are based on Arm®[a] cores.

The reference manuals describe in detail the memory subsystem. In the extremely simplified diagram of *Figure 1* only common points are visible.

**Figure 1. Simplified diagram of memory subsystem**



STM32L0 Series uses Cortex® M0+, while STM32L4 and STM32G4 Series use Cortex® M4 core with separate data and instruction buses. This fact translates to more complexity within the AHB BusMatrix in latter devices. Another difference is that STM32L0 are based on EEPROM technology. When designing an application, it is important to keep in mind that the STM32L0 data EEPROM shares the bank interface with the program Flash memory.

All the MCUs covered in this document feature some sort of cache within the NVM interface, simpler for the STM32L0 Series, with the more advanced ART Accelerator™ on those based on Cortex M4.

The dual bank memory can be configured and used as a single large NVM block with continuous addressing (with few exceptions, not covered in this document). There are significant advantages when the NVM is configured to serve as two parallel blocks, the most important is the possibility to write on one bank without interrupting reading (and fetching instructions) from the other bank. This is the most important prerequisite to perform the updates without breaking the execution of the code from the program NVM.

When designing an application that uses a dual bank device, there are several choices to make on how to utilize the second half of the program memory.

a. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

# 3 On-the-fly update

Also referred to as live field upgrade, this is a process that allows the user to modify the code and configuration without disturbing the normal operation of the device. There are more advantages compared to the simple IAP solution:

- the loader code can be updated when using the dual bank
- if the loading fails, the original code is still functional (the operation can be “atomic”)
- there is no need to define a loader state, the device is always capable of loading the code.

In the examples detailed in this document the goal is to go from full operation based on the original code to full operation based on the updated code within microseconds. This documents also describes the case when the ISR is executed from RAM using a relocated vector table. This is a common solution to get low interrupt latency, but tricky to execute with field upgrade.

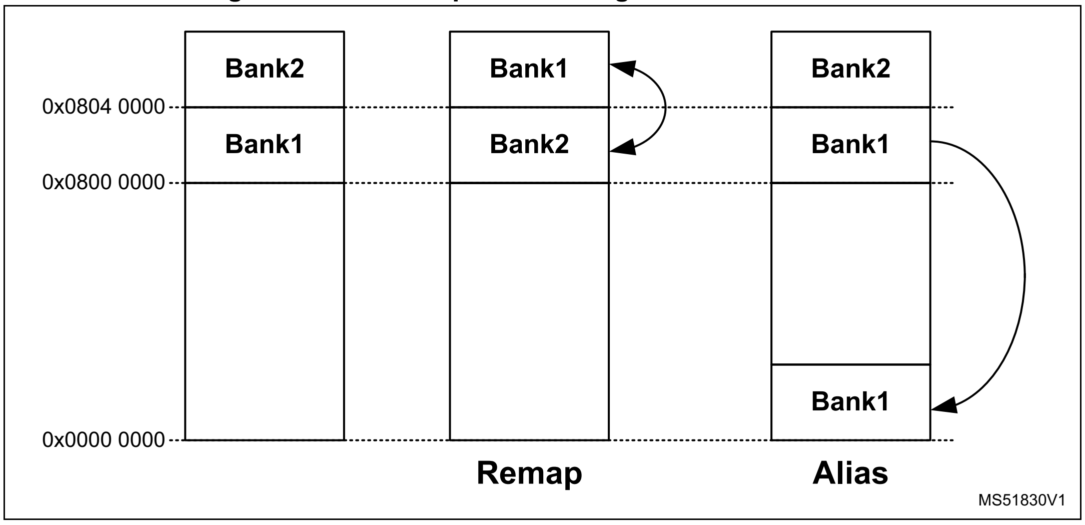
## 3.1 Supporting features of the MCU

There are several mechanisms implemented in the microcontroller to ensure that the firmware performs, among them the most important is the ability to write one non-volatile part while executing code from the other.

### 3.1.1 Memory remapping switch

A control bit labeled as FB_MODE for STM32L4 and STM32G4, and UFB for STM32L0, is accessible to the user code. This bit in the system configuration register controls the memory mapping and aliasing. It also used by the dual bank boot mechanism, and, with enough care, it can be used for the live field upgrade as well.

**Figure 2. NVM remap and aliasing on STM32G4 MCUs**



Depending on the flag setting, either Bank1 or Bank2 is mapped to start at address 0x0800 0000 and aliased on address 0x0000 0000. Because the operation does not affect the PC and other CPU registers, the CPU will simply fetch the next instruction from the other bank when the bit is flipped. The example code does not link to the alias address range, because if ST bootloader in System memory is involved, it will remain aliased to address 0x0000 0000.

Codes in both banks are normally linked to start at address 0x0800 0000.

### 3.1.2 Relocatable interrupt vector table

The CPU features configurable offset for the IVT. It allows the software to define more IVT and switch between them as necessary.

Default value of vector table offset is 0x0000 0000, which usually points to the alias of the program memory.

It is important to reserve enough space for the vector table, 4 bytes for each interrupt vector. There are also limitations on the offset address value, which must be aligned to multiples of 512 bytes.

### 3.1.3 BFB2 flag in user option bytes

The flag essentially triggers the attempt to boot from Bank2 on reset.

It is important to keep BFB2 flag set when there is no code in Bank1, thus being safe in case of unexpected power failures. With BFB2 set, the system bootloader is activated to evaluate if a code is present in Bank2 and then puts it in control, if possible (details are given in reference manuals or in AN2606). Then the firmware has to detect that the code in Bank1 must be replaced, and that the process is running from Bank2.

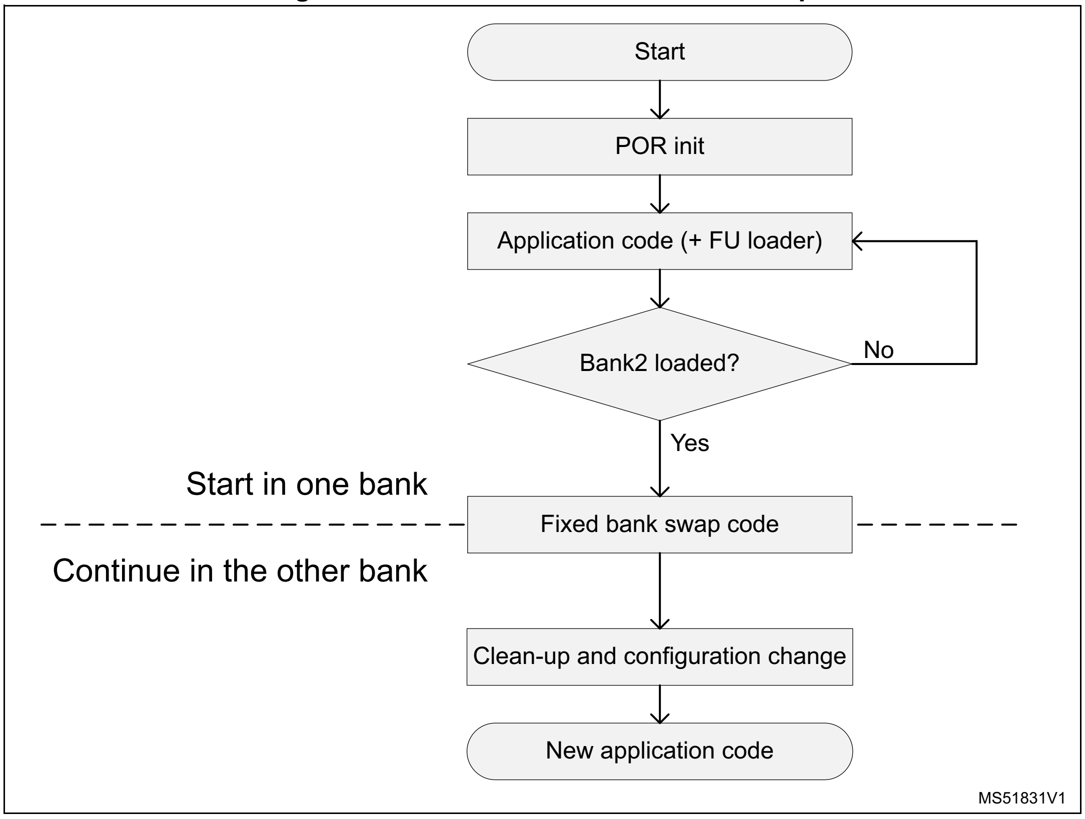
In this application the BFB2 is used only as fail-safe mechanism in the case of unexpected power cut when the code is executing from Bank2. The BFB2 must be kept active when Bank1 contains no valid code, it is cleared when new code is copied from Bank2 to Bank 1.

# 4 Unmodifiable code

As stated earlier, during the bank swap the CPU will fetch the next instruction from the other bank, due to the bank remap. The recommended approach is to create a section of code, preferably within the main() function, where the stack is at its lowest, and make this section the unmodifiable code.

This unmodifiable and fixed code will then take care of the transition itself. If the code in both banks is not consistent near the transition point, the execution can be unpredictable and even crash.

**Figure 3. Unmodifiable code for bank swap**



This code can actually be a very small part of the total project (the smaller, the better). As it is not normally modifiable by the live update it has to be bug free.

This fixed bank swap code performs the following actions:

1. Disables interrupts
2. Disables and flushes cache
3. Modifies the interrupt vector table offset
4. Flips the bank
5. Restores cache configuration
6. Enables interrupts

Remember that having identical source code does not implicate that identical binary codes will be generated. It is better to generate a library to be linked to all future versions, or preserve an object file, or (perhaps) even carefully edit the resulting binary.

# 5 Code in RAM

If the solution with unmodifiable code is deemed too risky for the intended application, it is possible to have a function executing in RAM for the bank swap, optionally manipulate the stack pointer and then jump to a safe entry point within the Flash memory code.

In the provided project example the RAM is used only for ISR code. The rationale is that the applications that require on-the-fly field upgrades typically do so because they need to respond to signals in deterministic time, with minimal latency. The Flash memory is slower than the RAM, hence the time-critical ISRs are put in the volatile memory, where the new instructions after branch can be fetched without latency (at least until the ISR function makes a call to a function in Flash memory).

However, while there are typically different SRAM areas, they can not be remapped like the Flash memory banks. To replace the code in volatile memory normally involves disabling the interrupts, copying replacement code and then re-enabling interrupts, with the risk of fatal crash in case of NMI. Alternatively, a temporary interrupt code placed in the Flash memory can be used while the RAM contents are updated, with a penalty due to the involved latency.

A third option is used in our example. The solution is to define two sections in RAM, each dedicated to ISR code for one bank. The binary is linked with two identical copies of each function in RAM. The project is capable of working with either of them, depending on the Flash memory bank where it is executed. The unused copy can be updated without disruption of the normal functionality, just like when programming the other bank in the Flash memory. But instead of remapping, the vector table offset is used to switch between the two vector tables, each containing addresses of different copy of the ISR function. This way the NMI risk is mitigated, but the NMI handler has to be kept unchanged and without calls to other memory.

The vector table offset is stored in Arm register defined in CMSIS as SCB → VTOR. It is user accessible.

For RAM functions that are not ISR, it is possible to define a similar reference table (as long as the complexity of branching still makes placing code in RAM advantageous). No such code is used in this example.

# 6 Volatile data

The goal of the live field upgrade is to make the transition to a new code version without going through the system reset. On reset, the startup code (usually by default coming from the development tool provider) initializes the RAM before giving control to the user code. This typically involves wiping addresses of zero-initialized RAM area and copying default values from NVM to the areas called constant-initialized.

Most of this usually happens “behind the scenes”, only a long startup time (in case of excessive amount of initialized global variables) is noticed.

When using the on-the-fly update the developer must manage volatile data structures placement and contents. As a bare minimum it is necessary to check the map file and see the changes in the RAM address range.

## 6.1 Avoiding change in volatile data structure

If a new map file shows significant undesirable and unpredicted changes in the placement of data in RAM, there are several ways to put them into the original location. Unfortunately there is no universal way, nothing that works regardless of the used toolchain. Linker decides placement of data in the memory, and all linker controls are product specific.

Some modifications can be prevented by avoiding changes in the compiler and linker settings, but more reliable solution is to remove the sources altogether and replace them with object files generated with the original version.

Newly added data can be moved into a separate, previously unused memory section. If a variable is removed, resulting in address shift, it is sufficient to replace it with a dummy, unused variable of the same size.

In any case refer to the linker documentation to see what aid it can provide to ease the task of patch development.

## 6.2 Managing modifications of volatile data structure

It is possible to define five basic levels of modification severity, as detailed in *Table 2*.

**Table 2. Modification severity levels**

| Severity | Case | Action after update |
|---|---|---|
| 0 | Map file of RAM is identical | No action |
| 1 | Variables removed | No action (but double-check addresses) |
| 2 | New variable added | Initialize as needed |
| 3 | Address modified | Move data |
| 4 | Structure modified | Code must convert old data to the new format |

The recommended solution is to have a dedicated function in the new code, an initializer that executes only once after the update was loaded. This run-once code will take care of the RAM contents before the actual functional code needs to access the variables.

This document does not cover the case of dynamically allocated heap memory. There is no guaranteed functionality of RAM heap in case of on-the-fly upgrade, and it is better to avoid it. Specific implementations may be usable with limitations.

## 6.3 Timing

It is not necessary to perform the bank swap immediately after the loading process finished. The loaded data will remain available in the other bank until the FW decides the time is right.

Depending on the actual application, there may be different tasks with different periodicity. In some cases it is possible to determine a time slot in which it is less likely to receive a time-critical interrupt. The highest priority interrupt is usually the most important, for example a control loop. If it is possible to predict that it will not reappear for a sufficient time window, this is a good moment to initiate the bank swap.

A flag within the system running on the MCU may be the signal to proceed with upgrade process. It may be a simple bit or include a timestamp, depending on software system architecture.

To minimize the time needed for the swap, set the maximum system clock.

## 6.4 The firmware example

The FW package called X-CUBE-DBFU contains a simple example solution intended for usage with selected STM32 EVAL boards. It is closely related to an IAP application, with important difference. There is only one firmware, identical for both banks, serving as both the loader and the application.

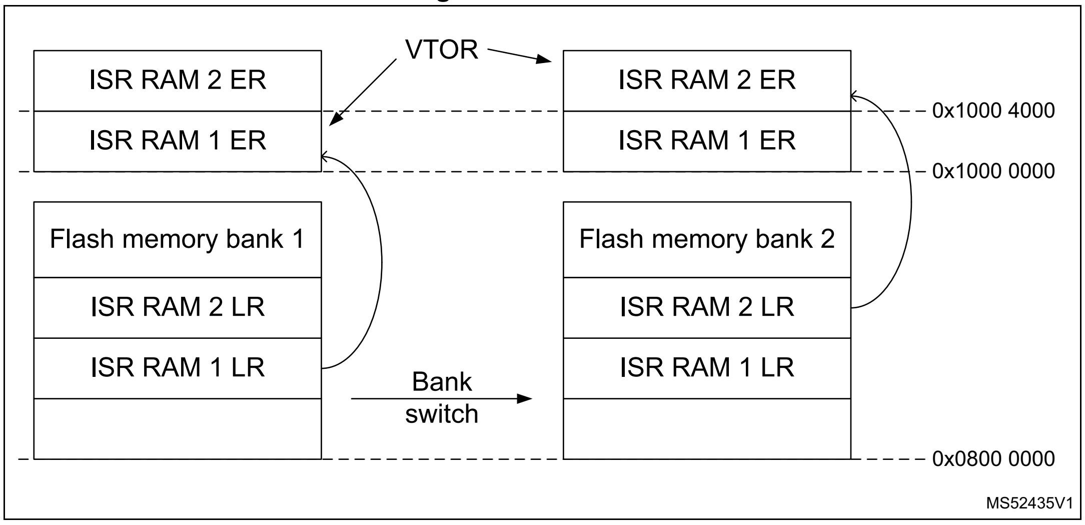
### 6.4.1 The example solution architecture

One of the challenges addressed by the example is the execution of the interrupt service routines from RAM. Each supported product is unique regarding the RAM (a single RAM block for STM32L0, two distinct RAM areas for STM32L4 and three for STM32G4). This does not affect the examples functionality, as the RAMs are not dual bank capable. This means that two RAM address ranges cannot be swapped by a hardware remap.

Instead, the solution described *Section 5: Code in RAM* is used. The other table is declared in stm32xxxx_it2.c source file for each particular project. When using the example as a starting point for further development, it is important to keep both copies of the interrupt routine code updated. Unlike the unmodifiable code section they do not need to have identical binaries, only the functionality should be the same (unless there is an intention to make the product behave differently while running from Bank2).

Setting the vector table is the first thing done in the code. In the example it takes place in main() function but it is possible to modify the SystemInit() in the system_stm32xxxx.c alternatively. Of course first the actual table must be initialized because it is held in the volatile memory, then the VTOR value is modified accordingly.

**Figure 4. ISR in RAM**



## 6.4.2 HW setup

Connect USART2 port to a PC using an RS-232 serial cable (the use of a generic USB adapter acting as virtual COM port is recommended). Use a terminal application that supports YMODEM protocol. Tera Term is offering a solid, cost-free alternative to Windows® HyperTerminal.

Example functionality has been tested with Tera Term version 4.84.

Configure communication speed at 115200 Bd, 8 data bits, no parity, 1 stop bit and then power the board.

### 6.4.3 Example operation

With user interface provided by the terminal SW the operation is easy. The example firmware first displays a header and a menu. Pressing numbers on PC keyboard selects choices.

1. Download a file to the other bank

   First menu choice initiates download of a binary file into the other bank. The former content of the other bank is erased and overwritten. The file size is constrained by the memory size. Select the file to download, it will be decrypted and written. Communication proceeds using the YMODEM protocol.
2. Erase the contents of the other bank

   This option invalidates the content of the other bank.
3. Rewrite the content of the other bank

   Code from the active bank is copied to the opposite memory bank.
4. Check the other bank integrity

   This option initiates check of the other bank content. With L0 Series the example FW stores the code CRC integrity value in the EEPROM Data memory. In case of other devices the check is limited to the simple presence.
5. Switch bank selection

   If there appears to be a functional code in the other bank, the vector table is rewritten and the banks are switched (see *Section 3.1.1: Memory remapping switch*).
6. Toggle the system bank

   Programming the option bytes to modify the value of the BFB2 bit (boot from Bank2, as described in *Section 3.1.3: BFB2 flag in user option bytes*).

The proposed sequence of on-the fly upgrade can be described by the sequence *1*, *4*, *5*, *6* (BFB2 ON), *3*, *4*, *5*, *6* (BFB2 OFF). Option *2* (erase) is included to simulate failure state.

### 6.4.4 Encryption option

It is understood that IP contained in the firmware upgrade file may be precious to the owner and there may be concerns regarding the code confidentiality. Combined with integrity protection, even symmetric encryption provides a basic defense against introducing counterfeit or fraudulent firmware.

Several ST microcontrollers include optional AES HW accelerator peripheral option. In this case, it is possible to use STM32HG484 instead of STM32G474, STM32L486 instead of STM32L476 and STM32L083 instead of STM32L073 in the Eval board to use the encryption functionality.

### 6.4.5 Encrypting the binary

The encryption scheme used in this example is a plain AES used in counter (CTR) mode.

The advantage of this method is that no padding is necessary, only the payload is encrypted. While it is not relevant to field upgrade case, the counter mode is also known for not propagating the communication errors (only the bytes that get corrupted during communication are corrupted in the decrypted message).

Furthermore, there is no need to develop dedicated encryption utility, any publicly available encryption library is capable of generating counter mode scratchpad and make XOR between the scratchpad and the data.

If not sure on how to prepare the inputs, follow the steps listed below:

1. download the open source library Crypto++® (www.cryptopp.com)
2. build the cryptest.exe using your preferred compile.
3. test the library using the./cryptest.exe v (v option: the Validation Suite)
4. if all tests are passed, the message “All tests passed!” will be displayed
5. generate a binary file and place it under Cryptopp library
6. call the cryptest.exe with parameters cryptest.exe ae <HexKey> <HexIV> Project.bin Firmware.aes.

A batch file is included along with the project to remove any doubts about key format.

The cryptest.exe will generate an .aes file that will be used on devices with AES peripheral.

### 6.4.6 Configuring for decryption

Add or uncomment #define ENCRYPT in the project. It will only work on devices with AES peripheral (STM32L083 / STM32L486 / STM32G484 in this case).

Replace the placeholder key and initialization vector with any other value in main.c to personalize the project.

### 6.4.7 Limits of the simple solution

The basic symmetric encryption solution used in this example has some limitations.

It uses the same key for all the devices in the field. Widespread use of single key makes the cryptographic system more vulnerable. Not only it is giving more opportunities to the attacker to obtain the secret key, but once the key is exposed, all the devices are open. This effect can be partially reduced by deriving the key using some property of the code, for example version.

Also, the authentication relies only on the fact of knowing the secret key, not completely mitigating the possibility to provide not genuine code.

The fact that first several bytes of the code (consisting of initial stack pointer and vector table) are mostly predictable (the “known plaintext attack”) can lead to weakening of the cryptographic properties. This weakness can be easily addressed by adding a random sequence (“salt”) to the “plaintext” before encryption. Then the “salt” is ignored after decryption.

### 6.4.8 Other cryptography options

Protection of the code in not only matter of the encryption algorithm, but rather a complex cryptographic system, encompassing security requirements implemented in all product life phases, from design and development, through manufacturing and servicing to deactivation and recycling.

Authentication roles and integrity protection must also be seriously considered.

STMicroelectronics provides top-tier embedded security solutions.

## 6.5 Other implementation options

This section describes various techniques often used when implementing field upgrade mechanisms, but not used in the X-CUBE-DBFU example to keep it simple and generic.

### 6.5.1 The field upgrade file

In our example, the file loaded to the system is a plain binary, encrypted at most. This is normally not recommended. The file should be at least tagged with version and product identification, to prevent loading incompatible code to the product, avoid accidental downgrade or just for added convenience.

The field upgrade package usually includes not only code, but also data. There may be a section containing a new, updated configuration and a section containing initialization values for newly added RAM variables.

In more complex systems, the single package can also contain data that should be sent to other chips on the PCB after a successful update.

The live update process may also identify section of code that should be immediately copied to RAM and the jump point for code with post update actions, which will execute only once and take care of the transition and clean-up.

### 6.5.2 Data in NVM

Other popular reason to use the dual bank is to use the other bank for data, often using an EEPROM emulation algorithm. It is possible to combine on-the-fly upgrade mechanism described in this document with an EEPROM emulation as described in AN4894.

# 7 Conclusion

The dual bank feature, when used properly, has several advantages for a broad range of applications.

The example and description provided in this application note cover only the basic aspects, but serve as functional building blocks for practical projects.

It is very important to test thoroughly the resulting solution, and have a robust and reliable code capable of live updates.

# 8 Revision history

Table 3. Document revision history

| Date | Revision | Description of changes |
|---|---|---|
| 29-Sep-2016 | 1 | Initial release. |
| 31-Oct-2016 | 2 | Updated document title. |
| 15-May-2019 | 3 | Introduced STM32G4 Series.<br>Completely rearranged document contents, with changes in all sections, including the title. |