AN4730
Application note

Using the FIREWALL embedded in STM32L0/L4/L4+ Series MCUs for secure access to sensitive parts of code and data

Introduction

This application note describes how to use in a secure way the FIREWALL embedded in STM32L0, STM32L4 and STM32L4+ Series microcontrollers. This IP provides a very high security for code and data coming from third party, like an original equipment manufacturer (OEM) developing software code.

Sensitive information (such as cryptographic keys, secured algorithms and associated variables) is specified on identified segments of the memory mapping. The FIREWALL manages the access to these trusted areas and rejects illegal ones by resetting the microcontroller.

This document describes typical use cases in which the protection level requires the FIREWALL, in addition to some other protection features offered by STM32 MCUs.

Specific considerations needed to implement the most adequate software development strategy to build a secure project are highlighted as well. A list of recommendations is then provided, resulting in easier and faster development during the software debug phase.

This document also describes the software/hardware constraints linked to the FIREWALL implementation, to correctly handle the security mechanisms available in microcontrollers of the STM32L0, STM32L4 and STM32L4+ Series.

Contents

List of tables

List of figures

1 FIREWALL description

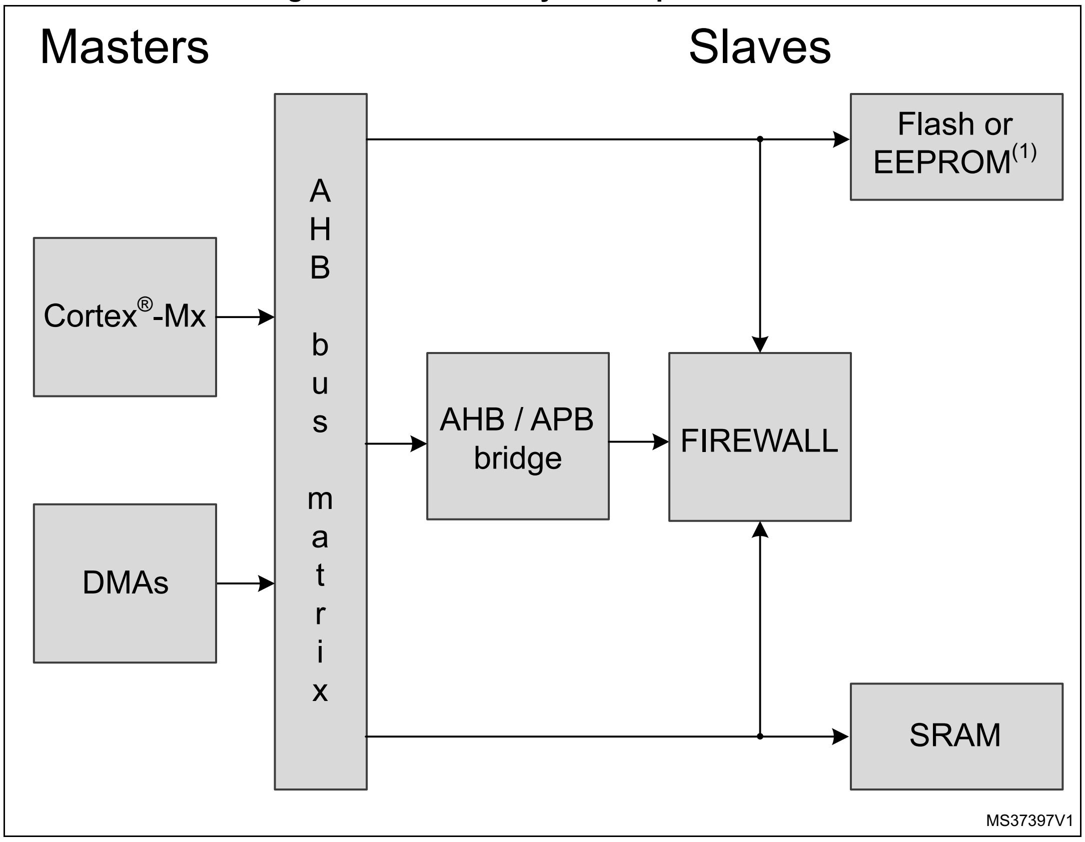
1.1 Introduction

The FIREWALL allows the user to establish trusted areas in the memory mapping where code or data are stored, and monitors access to these areas.

The STM32 microcontrollers featuring the FIREWALL (refer to the associated reference manual or datasheet to check the availability of this IP) are based on Arm®[a] cores.

The user may want to protect some sensitive algorithms using confidential associated data in the memory mapping, and/or manage exactly when the user code accesses the trusted areas, and when this secure area goes back to the non protected user-code execution. The FIREWALL performs the access monitoring (instruction fetches, read, write operations) and generates a reset if unexpected accesses are detected during the code execution, to stop immediately any intrusive action on the protected areas.

Figure 1. FIREWALL system implementation



1. Depending on the targeted STM32 microcontroller.

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

1.2 Trusted area

The FIREWALL monitors accesses to specified locations of the memory mapping identified as trusted areas. Three distinct locations can be defined (see *Figure 2*), each of them is also called segment.

- **Code segment**

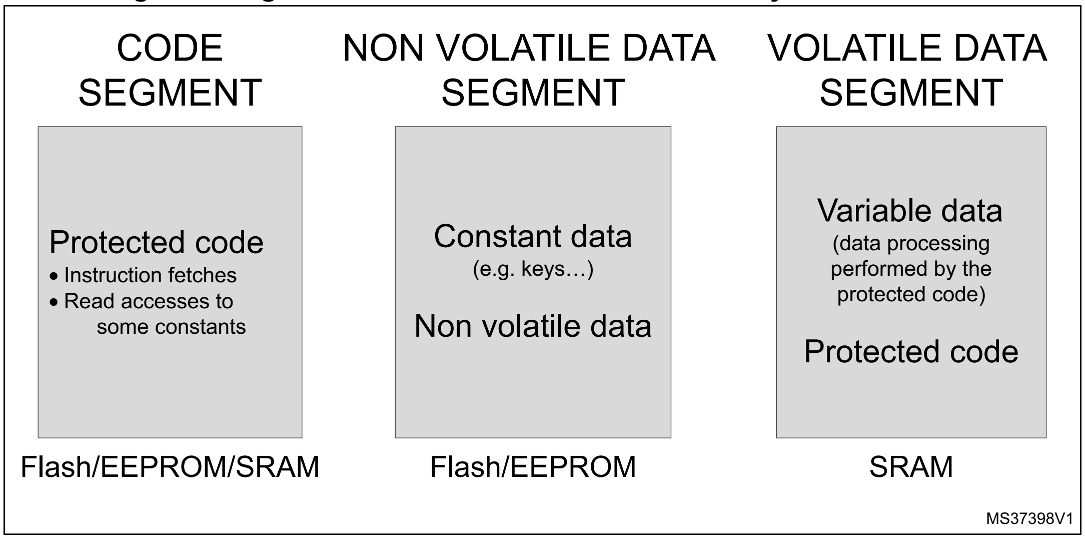
  This area contains the code to be protected by the FIREWALL. It monitors instruction fetches or data read accesses that have to take place at the right time (see *Section 1.4*) otherwise the FIREWALL rejects the accesses with a RESET. The protected code can be located into the Flash memory, or into the SRAM if the volatile data segment has been declared as shared or executable.
- **Non volatile data segment**

  This area contains mainly sensitive constants (like keys), or more or less static sensitive data used by the protected code. The segment is defined in the Flash memory or in the EEPROM (program or data), depending on the microcontroller used by the application.
- **Volatile data segment**

  This trusted area contains potentially variable data used by the protected code. It is located inside the SRAM. Data can be read or written when the accesses are done during protected code execution. In such case, the FIREWALL does not detect intrusive actions and lets the accesses to be performed. This segment can also be used to execute sensitive part of code if it is configured as executable during the FIREWALL configuration (refer to *Section 1.4*).

All these trusted areas (segments) are configurable through their respective registers pair, where the start address of each segment and their respective length (granularity defined in *Table 2*) is defined.

Figure 2. Segment location and content monitored by the FIREWALL



1.3 FIREWALL states

The FIREWALL has three states: *IDLE*, *CLOSE* and *OPEN*. *Figure 3* shows the transition scheme between the different FIREWALL states.

The FIREWALL monitors the AHB bus accesses connected to the Flash memory, or to the EEPROM or to the SRAM, whatever the FIREWALL state except in *IDLE* mode.

The Call gate mechanism is defined in *Figure 4*: it allows the user to put the FIREWALL in *OPEN* state, giving specific accesses to the trusted areas, which is not the case when the FIREWALL is *CLOSE* (refer to *Section 1.4*).

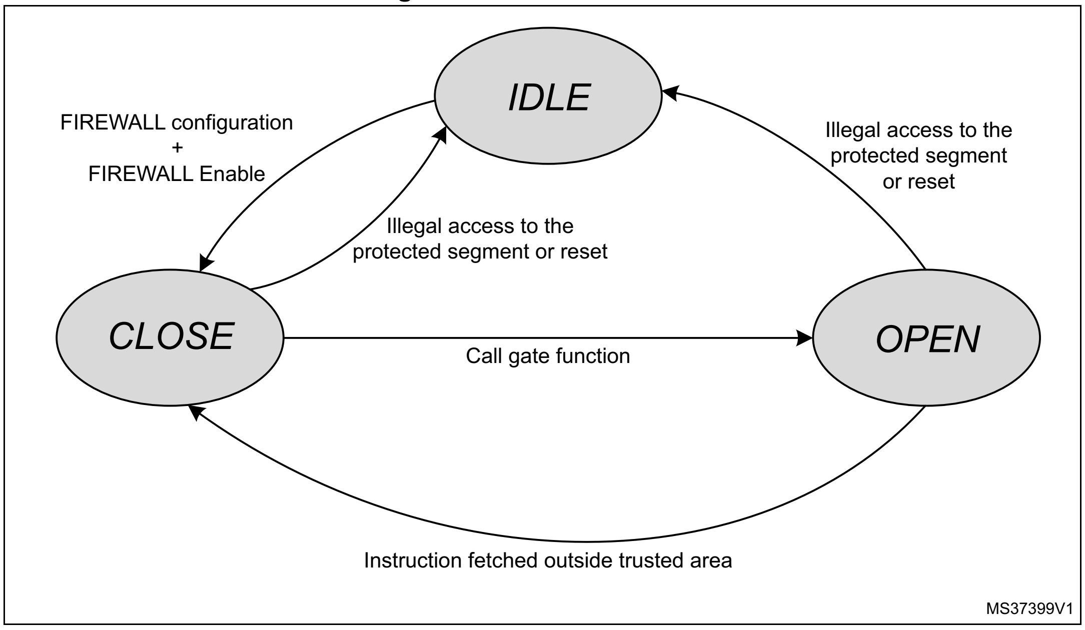
To close the FIREWALL and so to reject any access to the trusted areas (protected segments) from CPU, it is enough to fetch and execute instruction code located outside of the code segment or the volatile data segment if this last is declared as executable.

At release of the reset, the FIREWALL is in *IDLE* state. It reaches the *CLOSE* state as soon as it is configured and enabled.

Note: For consumption purposes, the FIREWALL starts the bus monitoring only when enabled.

There is no FIREWALL memory bus monitoring if it is not enabled, it remains in *IDLE* state, the default FIREWALL state after reset.

Figure 3. FIREWALL states


IDLE
FIREWALL configuration + FIREWALL Enable
Illegal access to the protected segment or reset
Illegal access to the protected segment or reset
CLOSE
Call gate function
OPEN
Instruction fetched outside trusted area
MS37399V1

1.4 Trusted areas access properties

As soon as the FIREWALL is enabled, it starts to monitor all the accesses to the non volatile memory and to the SRAM. Each access requested on the trusted areas is checked and can be flagged as illegal by the FIREWALL, depending on its current state.

Table 1 summarizes the type of access permitted to the trusted areas according to the FIREWALL state.

Table 1. Segment access properties depending on the FIREWALL state

Trusted area		**FIREWALL status**		
		***IDLE* (FIREWALL deactivated)**	***CLOSE***	***OPEN***
Code segment		All accesses allowed[1]	Read/Write/Execute accesses are illegal, except the Call gate function execution	Read and execute accesses allowed Write access is illegal
Non volatile data segment			Read/Write/Execute accesses are illegal	Read and Write accesses allowed Execution access is illegal
Volatile data segment[2]	Not shared Not executable			
	Not shared Executable		Read/Write/Execute accesses are illegal, except the Call gate function execution	Read/Write/Execute accesses allowed
	Shared Not executable		Read/Write/Execute accesses allowed	
	Shared Executable			

1. The accesses may be restrained if some specific protection level have been set-up by the user like PCROP, Write protection or some Level 1/Level 2 Flash memory read out protection (please refer to the reference manual of the STM32 targeting the application).
2. The Volatile Data Segment configuration has to be done before enabling the FIREWALL. The bit VDS and VDE present into the FIREWALL_ CR register allows to specify the segment attributes for the Volatile Data Segment.

Each Illegal access when the FIREWALL is closed generates a FIREWALL reset, which generates a global reset of the STM32 microcontroller, thus “killing” the access attempt.

1.5 Call gate function

1.5.1 Introduction

The call to the Call gate function is the single entry point in the user code to jump into and to execute the protected segment switching the FIREWALL from *CLOSE* to *OPEN* state.

1.5.2 Call gate function requirements

It is mandatory for the Call gate function to respect the following criteria to avoid the FIREWALL to flag an intrusive access to the trusted areas when it is called:

- Call gate function must be located into the code segment or the volatile data segment when this last is configured with the attributes “not shared” and “executable”.
- The function must be located at the start address + 4 of the trusted areas that can be executed. It can be:
  - @FIREWALL_ CSSA + 4; or
  - @FIREWALL_VDSSA + 4 if the Volatile data segment is declared to be not shared and executable.
- The instructions fetched at the @FIREWALL_ CSSA + 4 and @FIREWALL_ CSSA + 8, or @FIREWALL_VDSSA + 4 and @FIREWALL_VDSSA + 8 must be executed sequentially to open the FIREWALL.

There are two ways to manage the correct address alignment of the Call gate function:

1. In the scatter file (for instance with MDK-ARM™ toolchain):

   If the Call gate function is defined into a dedicated c-file (for instance callgate.c if the code segment start address is 0x08005000):

   ```
   ER_IROM2 0x08005004 FIXED {
   callgate.o (+RO)
   }
   ```
2. Using an address pointer on the function if, for instance, the protected code is located at the address 0x08005000 as start address for the code segment. In the c-file calling the Call gate function, it may be (for STM32 device based on Cortex®-M0+ core):

   ```
   typedef tAPIReturnCode (*API_CallGate_p) (tAPIArgsPointer cargs);
   API_CallGate_p CallGate;
   CallGate = (API_CallGate_p) 0x08005005;
   .....
   .....
   result = CallGate(cargs);
   ```

   The compiler is going to compile the Call gate function at the address 0x08005004. Actually the jump address is at the address 0x08005005, because the LSB of the address when doing the jump must be equal to 1. In fact, the core masks off the LSB, but it has to be set to execute thumb instruction. In STM32 based on Cortex®-M0+, this bit must be set. For STM32 based on Cortex®-M4, it can be set or cleared depending on the way the function is called (respectively thumb or normal Arm® function).

Note: *In this example, it is supposed that the Call gate function and the protected code are already delivered by the third party to OEM and are not visible. Only the API must be given to the OEM by the third party to define which are the arguments (input/output) passed to the Call gate function. Refer to Section 4.3.*

1.5.3 Call gate function structure

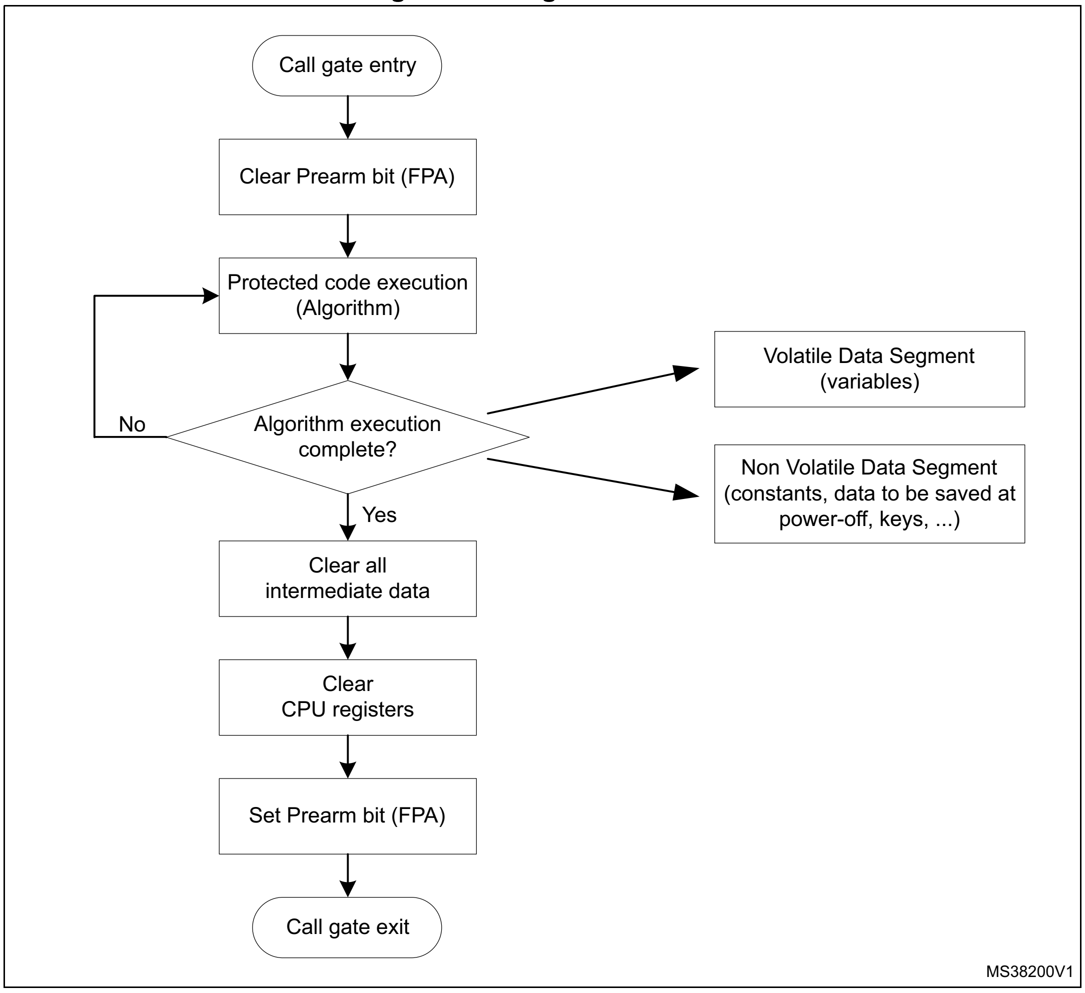
The Call gate function is already identified as protected and sensitive code. This is the single entry point in the non-protected code to open the FIREWALL and to give the privileged accesses to the trusted areas.

It can be seen as a switch case C-function, which enables to execute parts of sensitive protected code depending on the parameters passed to the Call gate function during its call from the non-protected code area.

The Call gate function is not just the entry point, it also manages the exit point from the trusted area to the main user code on the Call gate function return. It closes the FIREWALL removing privileged accesses to the trusted areas.

Figure 4 shows the main Call gate function structure that can be implemented during the development phase of the secure code.

Figure 4. Call gate structure


Call gate entry
Clear Prearm bit (FPA)
Protected code execution (Algorithm)
Algorithm execution complete?
No
Yes
Volatile Data Segment (variables)
Non Volatile Data Segment (constants, data to be saved at power-off, keys, ...)
Clear all intermediate data
Clear CPU registers
Set Prearm bit (FPA)
Call gate exit
MS38200V1

The Call gate function has to manage the context cleaning before jumping back to the non-protected user code, to remove any trace of algorithm execution. The Prearm bit (bit FPA of the FW_CR register) must be set before returning back from the Call gate function, avoiding any reset generated by the FIREWALL. This is the way to define the exit point of the protected code execution.

1.6 FIREWALL configuration

The FIREWALL needs to be configured before enabling it. The configuration is performed thanks to the 3-pair registers (start address + length for each segment), in addition to a configuration register.

Here is the procedure to respect when configuring the FIREWALL:

1. Enable the FIREWALL clock (APB clock) setting the bit FWEN into the RCC_APB2ENR register.
2. Set the start address of the code segment into the FW_CSSA (protected code). Set the length of the segment to protect (FW_CSL register) with the granularity defined in *Table 2*.
3. Set the start address of the Non Volatile Data Segment into the FW_NVDSSA. Set the length of the segment to protect (FW_NVDSL register) with the granularity defined in*Table 2*.
4. Set the start address of the Volatile Data Segment into the FW_VDSSA. Set the length of the segment to protect (FW_VDSL register) with the granularity defined in*Table 2*.
5. Set the configuration register into the FW_CR to describe how the Volatile Data Segment is going to behave (executable or not, shared or not).
6. Enable the FIREWALL (clearing the bit FWDIS into the SYSCFG_CFGR1 register). It is a write-once bit. When set, it remains in this state until next reset occurs.

Table 2. Segment properties

Segment	Granularity	Max area(1)	Memory
Code	256 Bytes	4096 KBytes	Flash / EEPROM
Non volatile data	256 Bytes	4096 KBytes	Flash / EEPROM
Volatile data	64 Bytes	96 KBytes	SRAM

1. Depending on the STM32 product targeted (refer to the corresponding reference manual).

The Volatile data segment or the Non volatile data segment may be not defined (with a length equal to 0). In such case there is no protection on the considered segment.

If the Non volatile data segment has a length equal to 0, the access to the FW_CR register is possible whatever the FIREWALL state. On the contrary, the access to this register is only possible when the FIREWALL is in *OPEN* or *IDLE* state. All accesses when the FIREWALL is in *CLOSE* state generate a reset.

2 Constraints

The use of the FIREWALL protection in an application code induces some constraints, mandatory for correct code execution.

2.1 DMA

DMA accesses into the trusted areas are forbidden as soon as the FIREWALL is enabled. Whatever the FIREWALL state (*OPEN* or *CLOSE*), any access to these areas is detected by the FIREWALL, which immediately resets the microcontroller.

This constraint avoids opening a vulnerability dumping some part of code or even some data if DMA requests are served when the FIREWALL is *OPEN*.

2.2 Interruptions

Note: When the FIREWALL is OPEN, no interrupt must take place during the execution of the protected code.

It is up to the non-protected code to ensure that this constraint is satisfied, otherwise the FIREWALL reacts like described below:

- If the interrupt is generated, and if the Call gate function does not manage the bit FPA (keeping the bit at 1 in the Call gate function), the program counter jumps to execute the interrupt subroutine, having as effect to *CLOSE* the FIREWALL. Returning back from the interrupt subroutine generates a reset by the FIREWALL since the protected code continues from the point where it was interrupted, without re-opening the FIREWALL through a Call gate sequence.
- If the Call gate function coding is managing the FPA bit as in *Figure 4*, the reset is generated by the FIREWALL as soon as the interrupt is served.

Note: If the application software needs to be interrupted before the completion of the protected algorithm, the user has to periodically check some pending bit of the peripherals, which can interrupt the algorithm execution.
In such case the Call gate function has to be coded to enable anticipated exit windows, discarding the on-going processing, to set the FPA bit and to exit in the shortest time without having real interruption feature.

Figure 5. Interrupt generation when the FIREWALL is *OPEN* with FPA = 1

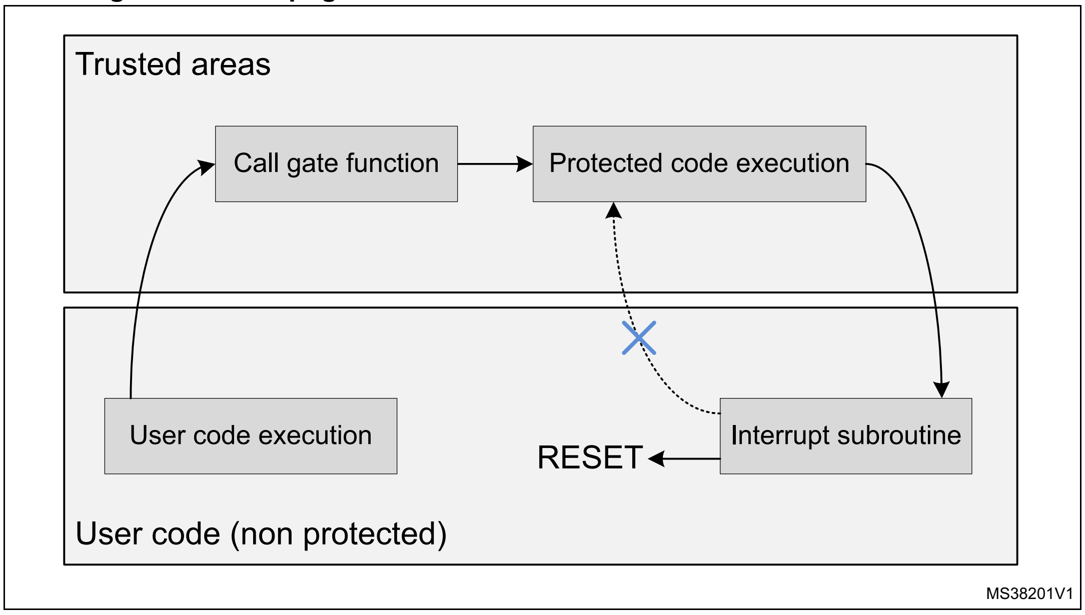
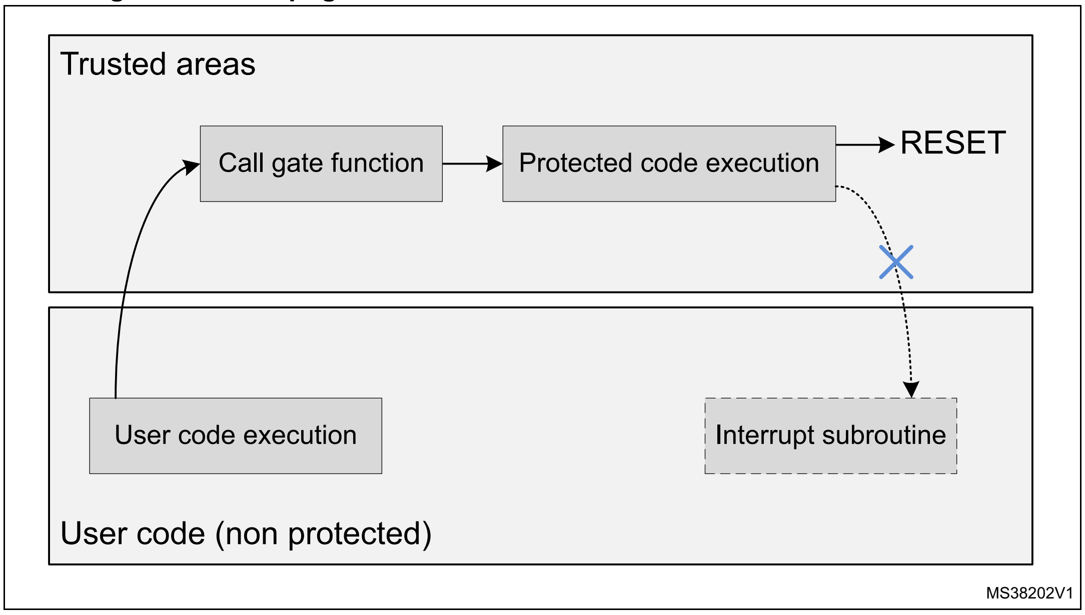


Figure 6. Interrupt generation when the FIREWALL is *OPEN* with FPA = 0



2.3 Return from the Call gate function to close the FIREWALL

As discussed in *Section 1.5*, it is strongly recommended to set the bit FPA into the FW_CR register before leaving the Call gate function and after cleaning the context (CPU registers, intermediate variables) to remove any trace of data processing when returning back to the non-protected code execution.

Specific care must be taken to assure a return from the Call gate function to the non-protected user code without any reset generated by the FIREWALL, even if the FPA bit write condition has been requested.

Indeed, the write instruction to the FW_CR is crossing the AHB/APB bridge. The write is bufferable and so from AHB bus side, the bridge completes the write access whereas it is not yet performed on APB side (Write operation to FW_CR may be shifted on APB). The CPU executes the next instruction after the write operation, and it is the return of the Call gate function. If the FPA bit is not yet effective in the register at this time, a reset is generated by the FIREWALL.

To avoid this side effect, it is recommended after the write of the bit FPA to read it back before returning from the Call gate function to be sure the write operation is completed when the read is performed. The procedure is the following:

1. Clean the context (variables, CPU registers)
2. Set FPA bit into the FW_CR register
3. Read the FW_CR register (dummy read)
4. Return from the Call gate function

2.4 Debug session

When debugging a software having the FIREWALL enabled, there are some points to take care to avoid reset generation by the FIREWALL.

2.4.1 Breakpoints

During the debug session, some breakpoints can be positioned inside the trusted areas. If the PCROP protection has been set, it is forbidden to put breakpoints into the protected code segment, because the protected code must be not visible by the developers. The FIREWALL does not generate a reset as it is *OPEN*, but the PCROP protection reacts if the developer is looking, for instance, to the disassembly code (refer to *Section 3* and *Section 4* for more details about possible protection levels during the debug phase depending on the application context development).

2.4.2 Debugger windows

The software developer has to take some precautions when debugging with the FIREWALL activated:

- Disassembly viewer: when the code is under debug, the developer must monitor which addresses are displayed on the disassembly viewer. If the FIREWALL is *CLOSE* and if the software execution reaches a breakpoint, the disassembly viewer must not be displaying any address located into the protected segment. Otherwise, the reset is generated when restarting the code in run or step by step execution.
- Memory viewers: be careful to not let a memory viewer opened pointing on protected memory map when launching the code in execution (all modes: run, step by step, step over). The effect is to get an immediate reset because the debugger is refreshing the display reading back some data into the protected area while the FIREWALL is *CLOSE*.
- Watch viewers: like memory viewers, displaying in real time protected c-variables triggers the same effect, a reset generated by the FIREWALL.

2.4.3 Step by step operations on protected code

In addition to the points highlighted in *Section 2.4.2*, specific attention is required for the protected code developer when debugging its code with the FIREWALL activated.

Breakpoints must not be put on the start address + 0 or + 4 or + 8 of the code segment or the volatile data segment if defined as executable, otherwise a reset is generated when restarting to run the code from the breakpoint. The FIREWALL is considered *OPEN* if the start address + 4 is fetched first, followed by the start address + 8 sequentially. So having a breakpoint before the start address + 0x0C generates a reset because the disassembly window is going to display data of the protected code whereas the FIREWALL has not yet reached the *OPEN* state.

3 Protection level requirements

In addition to the FIREWALL protection, some other protection levels are required to get the maximum security level on accessing sensitive and protected areas whatever the project phase in development or production.

There are additional protection levels offered by STM32 microcontrollers that can be used in parallel to the FIREWALL protection.

3.1 PCROP protection

This protection avoids a dump of a code placed in the region covered by the PCROP (defined by the user). Only instruction fetch accesses are allowed in this area. Read and write operations are rejected. The user can only remove the PCROP protection by RDP level regression (see *Section 3.3*), but in this case the whole Flash memory is completely erased.

PCROP protection can be used to protect, in parallel with FIREWALL, the sensitive code placed into the code segment allocated during the FIREWALL configuration. It can be used when the software developer is developing the code (non-protected user code) interacting with a third party protected code (for instance crypto algorithm) located into the code segment.

In such case, the developer cannot dump the code in debug mode, if a breakpoint is put somewhere inside the protected code area (FIREWALL is *OPEN*).

There is a drawback: the developer of the protected code must force it to not have literal pool in the code segment or to not have any constant located in this area (using specific compilation). This is to avoid any read operations not allowed on a PCROP region.

3.2 Write protection

Write protection can be applied on the region in which the FIREWALL configuration and activation are performed (also the reset vector). It avoids any unwanted write operation that can corrupt the FIREWALL configuration and/or modify it.

3.3 Read out protection Level 0 (RDP Level 0)

There is no protection on the Flash memory when this read out protection level is set, allowing the user to have full debug features. In an application using the FIREWALL mechanism, the debugger can access the protected code if a breakpoint is positioned inside the code segment and if the PCROP protection has not been set. In such case, the protected code can be read when the breakpoint is reached, the FIREWALL is *OPEN*.

Note: The volatile data segment protected by the FIREWALL may be accessible in this mode by the debugger under specific conditions.

3.4 Read out protection Level 1 (RDP Level 1)

With this protection level 1, the debug of the code in the Flash memory is no longer possible. As soon as a debugger is connected to the device, any access to the Flash memory is impossible.

In such case, it is also recommended to deliver devices for debug with PCROP protection on protected code to the developers coding the non-protected application code, to avoid any possible intrusion during the debug phase in the trusted areas.

Note: The volatile data segment protected by the FIREWALL is accessible in this mode by the debugger under specific conditions. As the execution from Flash memory is no longer possible, the debugger only allows the user to get a snapshot of the RAM content and of the CPU and peripherals registers at the moment when it has been connected.

3.5 Read out protection Level 2 (RDP Level 2)

This is the highest protection level, recommended to use for full protection level on sensitive applications. Access by the JTAG/SWD is forbidden. Debugger cannot be connected since the link is broken. It is the level to be set when the devices are in production mode to reach the end customer, resulting in full protection of the designated areas against external attacks. The FIREWALL is assumed to be fully efficient with read out protection level 2.

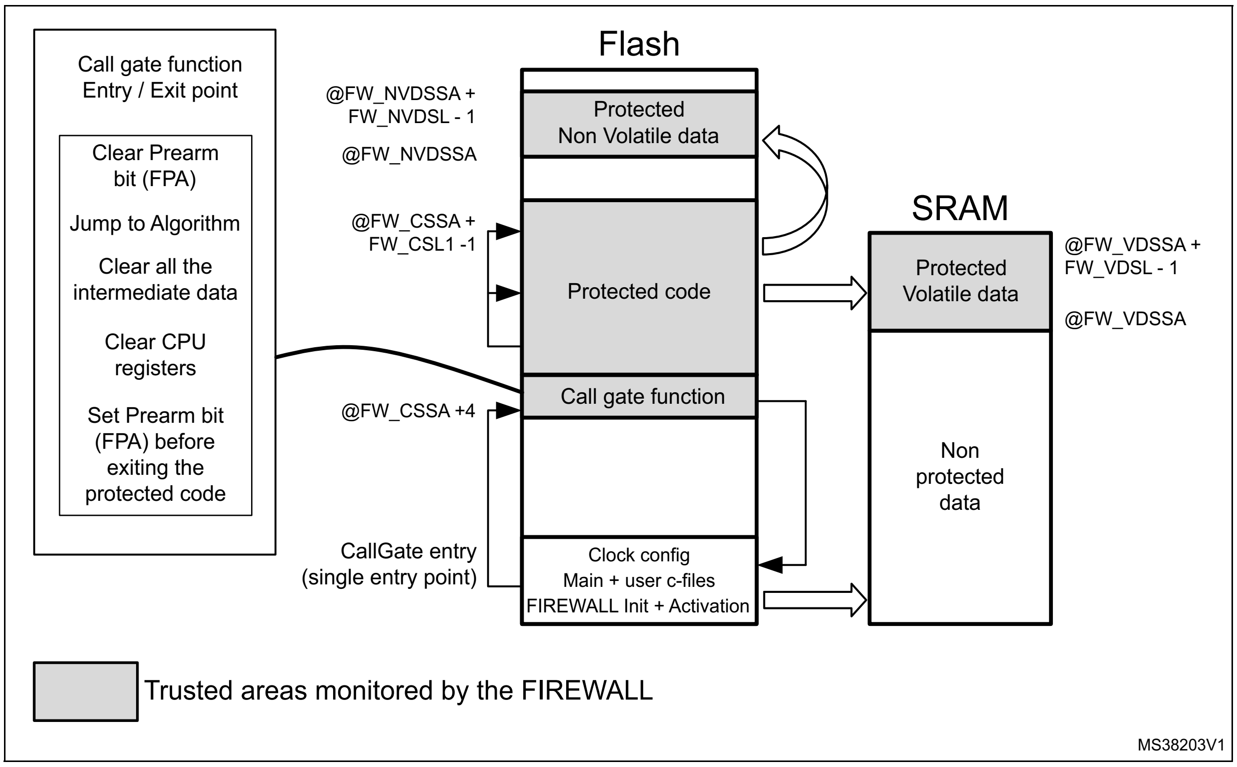
For the debug phase refer to *Section 4.4* if this applicative use case is the one user is targeting in its project.

4 Applicative use cases requesting FIREWALL monitoring

4.1 Software fully developed by a single third party

This section illustrates the applicative scenario in which the protected code and the non-protected code are developed by a single entity, as illustrated in *Figure 7*.

Figure 7. Software developed by a single third party



There is no need here to get RDP level 1 or 2 in such case during the debug phase. The single party has the full right to get visibility of the complete code (protected or not) during the debug, and can define manage the Call gate function. Take care of constraints (refer to *Section 2*) and to set the level 2 after the software validation phase when starting the production phase to get the highest protection level (refer to *Section 3.5*).

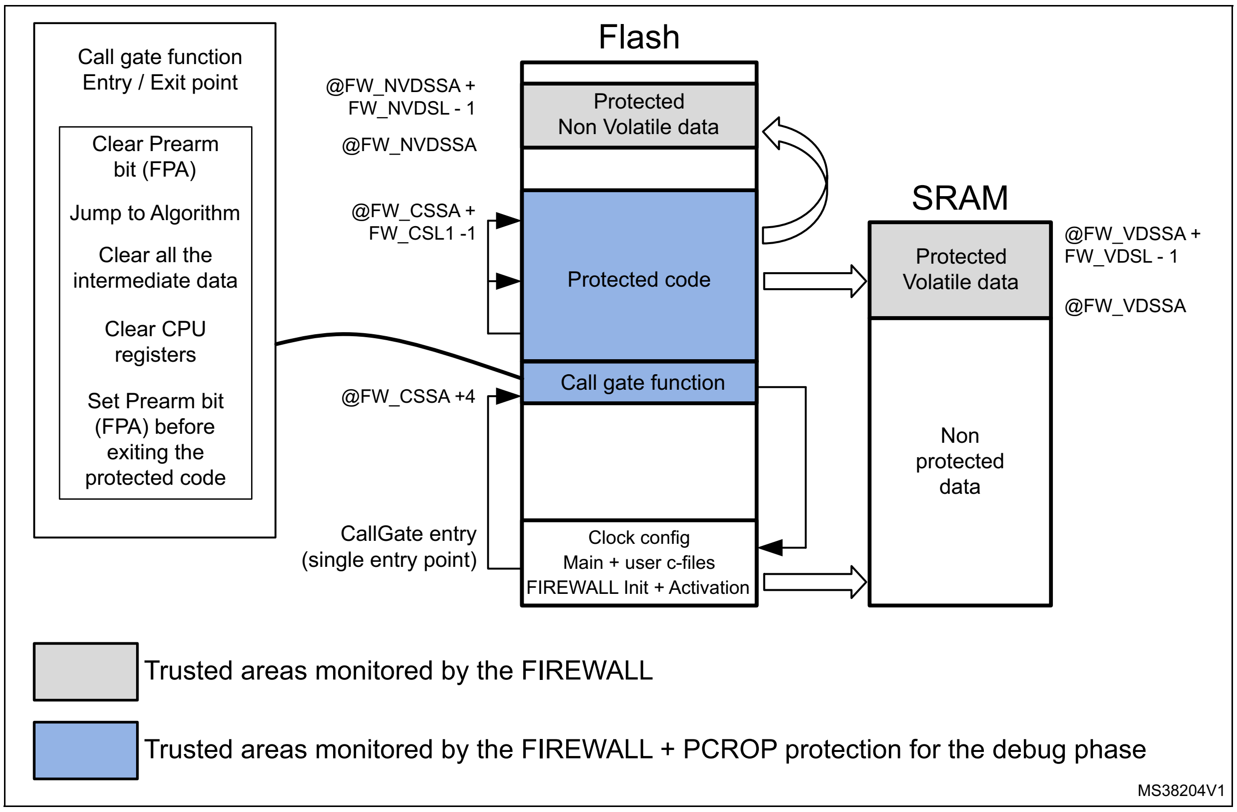
IAP (In application programming) code can be included inside the protected code, taking care about the code update, which must avoid any reset vector change value as well as changes inside the first part of the code (until the FIREWALL is enabled). In such a way, the PCROP protection is not mandatory on the protected code and the Call gate function when the device is in production.

4.2 Software from an OEM with a licensed algorithm (covered by NDA)

This section illustrates the applicative scenario in which the protected code (accesses secured by the FIREWALL) is developed externally and denies any visibility to the developers of the application software (see *Figure 8*).

Some API must be provided to them in order to interact with the protected code through the Call gate function. During the debug, the device is not with a RDP level 2, but rather a RDP Level 0 (mainly). In such case, NDAs have to be signed between the external provider of the protected code and the software developers (non protected application code) to cover the constraints induced by the debugging in RDP level 0 by the third party (refer to *Section 3.3*).

Figure 8. Software by a third party with a licensed algorithm (medium security)


Call gate function
Entry / Exit point
Clear Prearm bit (FPA)
Jump to Algorithm
Clear all the intermediate data
Clear CPU registers
Set Prearm bit (FPA) before exiting the protected code
Flash
@FW_NVDSSA + FW_NVDSL - 1
@FW_NVDSSA
Protected Non Volatile data
@FW_CSSA + FW_CSL1 -1
Protected code
@FW_CSSA +4
Call gate function
CallGate entry (single entry point)
Clock config
Main + user c-files
FIREWALL Init + Activation
SRAM
Protected Volatile data
@FW_VDSSA + FW_VDSL - 1
@FW_VDSSA
Non protected data
Trusted areas monitored by the FIREWALL
Trusted areas monitored by the FIREWALL + PCROP protection for the debug phase
MS38204V1

There are two possibilities:

- The best is to give devices to the software developer of the non-protected code with preloaded embedded libraries into the MCU with PCROP protection set (refer to *Section 3.1*). This can be done by the providers of the protected code (third party) before sending the parts to the OEM. The memory mapping code affected (code and data, including Call gate) has to be communicated to the third party to allow it to compile code and variables at the right place in the memory (trusted areas).
- An NDA can be negotiated between the third party and the OEM in case of potential protected code update during the debug/development phase by the OEM. In such a case, the OEM is in charge to erase the protected code (for instance removing the PCROP) and to embed the protected code without sending back the microcontrollers to the third party setting the PCROP protection before sending them to the developers.

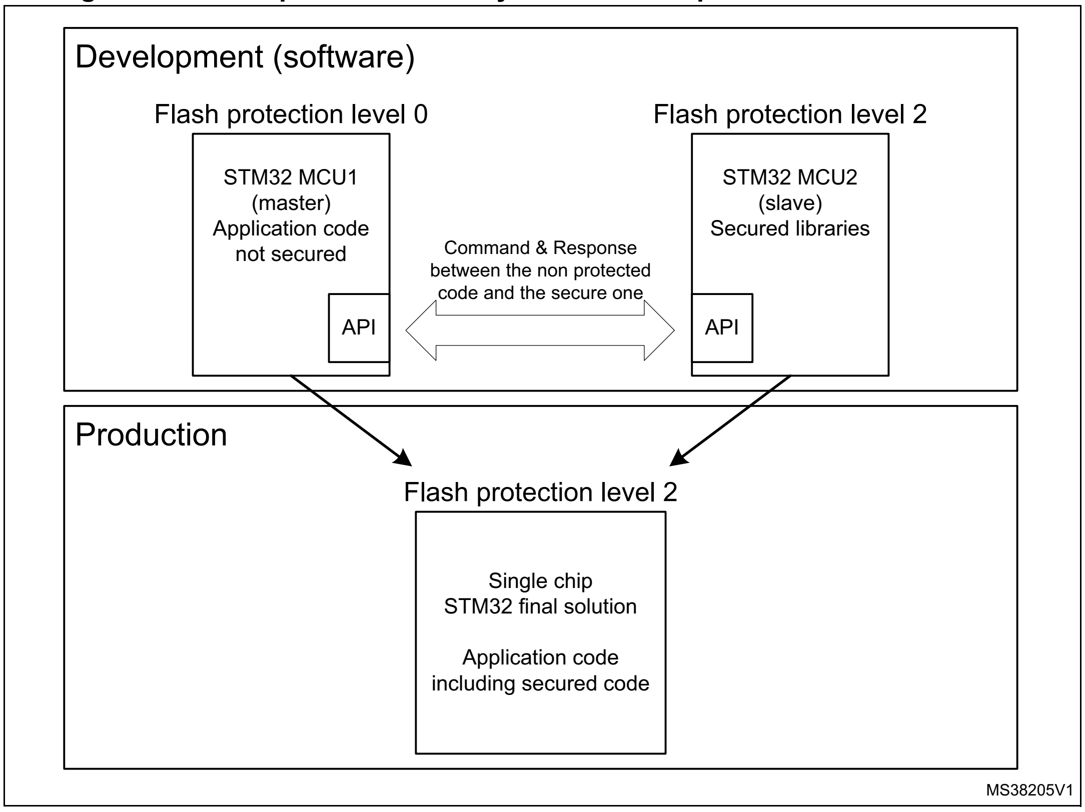
Here again, on production line, it is recommended to put a Flash memory protection level 2 to propose the highest security level of protection for the complete application (refer to *Section 3.5*).

Note: With read out protection level 2 associated to the PCROP, protected code update is no more possible.

4.3 Software from a third party with a licensed algorithm (full protection level)

The section illustrates the applicative scenario in which the protected code (accesses secured by the FIREWALL) is developed externally and should not have any visibility from the application software developers. No NDA has to be signed between the owner of the protected code and the third party developing the non-protected application code interfacing this protected code. In such case to make the application debug of the non-protected code, the best is to build a multichip debug environment until the validation is complete before merging in a single chip the final application (refer to *Figure 9*).

Figure 9. Dual chip solution for fully secure development with the FIREWALL


Development (software)
Flash protection level 0
Flash protection level 2
STM32 MCU1 (master) Application code not secured
API
Command & Response between the non protected code and the secure one
STM32 MCU2 (slave) Secured libraries
API
Production
Flash protection level 2
Single chip STM32 final solution
Application code including secured code
MS38205V1

The goal is to have the protected code in a STM32 co-processor MCU with Read Out Protection Level 2. The non-protected user code is developped in another STM32 MCU in level 0 to have full debug capability. Communication between the two microcontrollers can be done using a communication interface like the SPI for instance. The same API is implemented on both sides to send command from MCU1 and potentially to get back response from the secured MCU2.

The API has to be built so that it is easy to merge in a single chip the complete end user application at the end of the development phase.

Next sections describe what is inside each of STM32 devices in such configuration, having the following memory mapping as example:

- Protected code + Firewall initialization: 0x0800_0000 to 0x0800_2FFF
- SRAM protected area: 0x2000_0000 to 0x2000_2FFF
- User developer code: from 0x0800_3000
- User developer SRAM: from 0x2000_0300.

4.3.1 STM32 secured co-processor

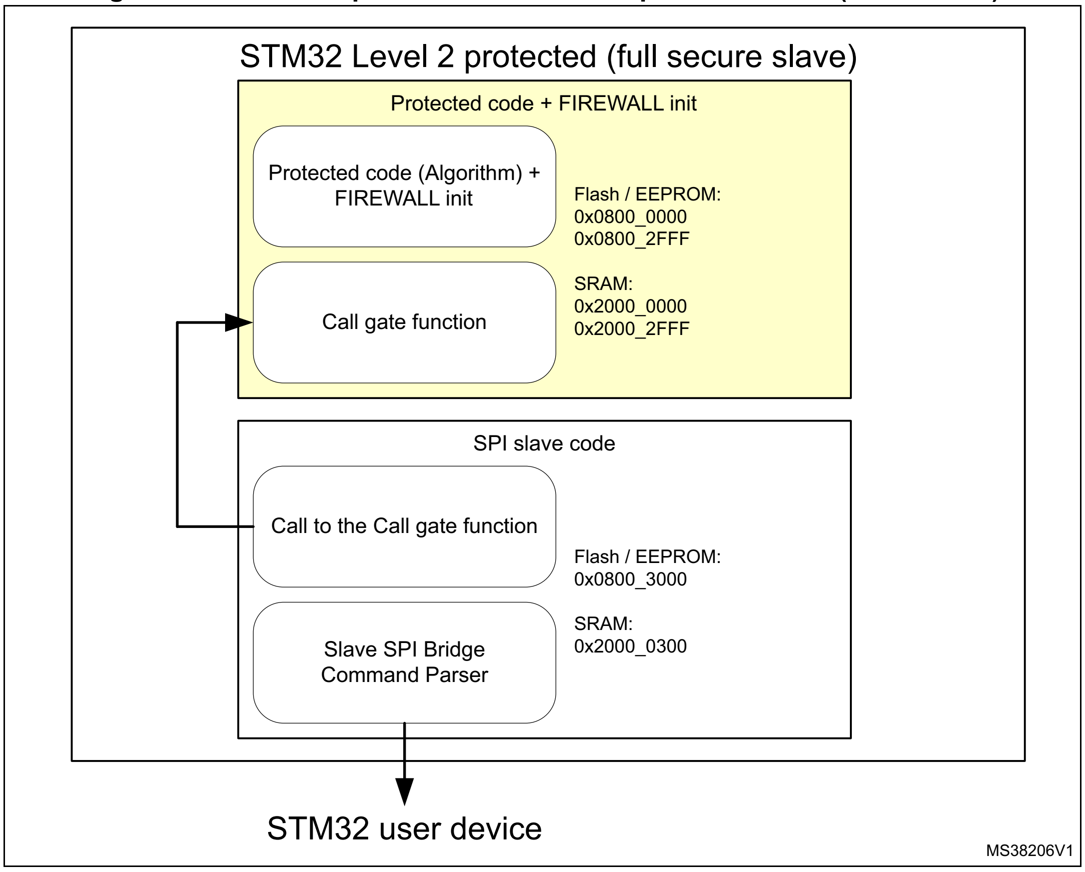
In the full secure STM32 co-processor, as shown in *Figure 10*, there are basically two parts:

- **Protected code and initialization**

  This is where the STM32 co-processor boots and in which the FIREWALL is configured before jumping to the communication SPI layer to exchange information from/to the STM32 co-processor and the STM32 user device. The final Call gate function is located there to manage the FIREWALL states and to interface user functions with the protected code.
- **SPI slave code**

  It contains the command parser, which gets the command from the STM32 user device, to pack it in a way to call the Call gate function, with the appropriate arguments (those used for the final production solution).

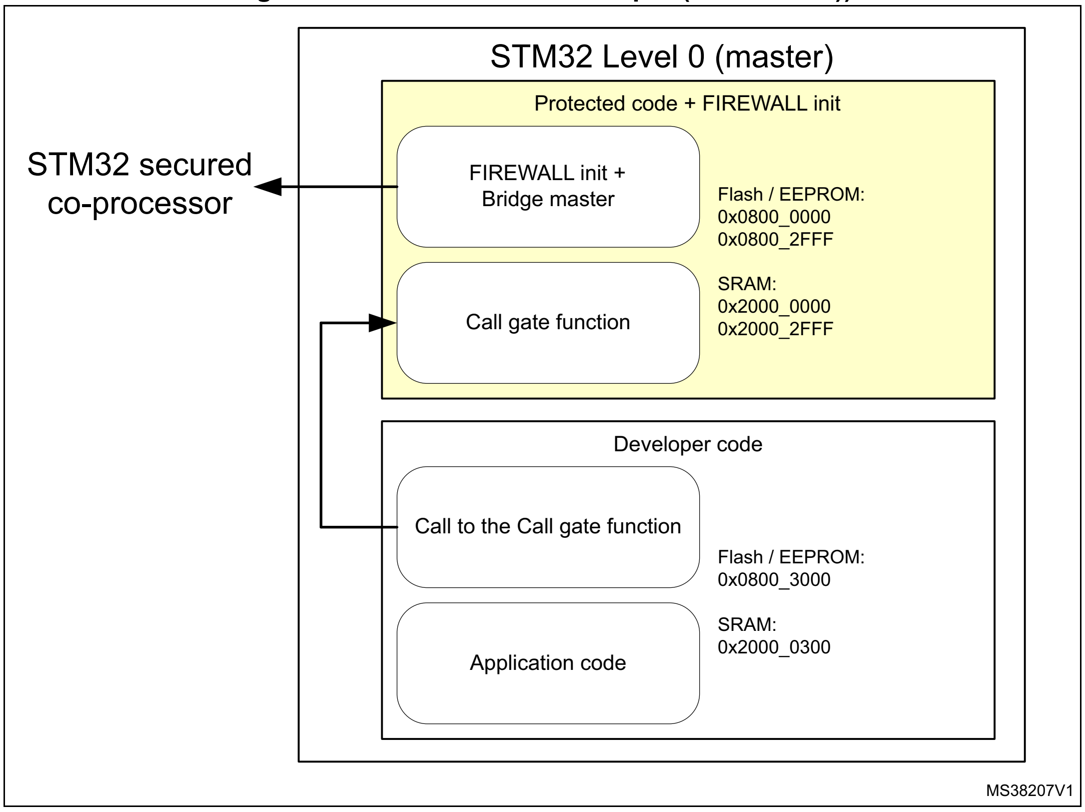
Figure 10. STM32 co-processor device with protected code (RDP level 2)


STM32 Level 2 protected (full secure slave)
Protected code + FIREWALL init
Protected code (Algorithm) + FIREWALL init
Flash / EEPROM: 0x0800_0000 0x0800_2FFF
Call gate function
SRAM: 0x2000_0000 0x2000_2FFF
SPI slave code
Call to the Call gate function
Flash / EEPROM: 0x0800_3000
SRAM: 0x2000_0300
Slave SPI Bridge Command Parser
STM32 user device
MS38206V1

4.3.2 STM32 developer device

The STM32 developer device allows the developer to make his applicative code, sending the command to the co-processor to hit the protected code through the Call gate function. The SPI API calls are identical to the one contained in the co-processor.

Figure 11. STM32 device developer (RDP level 0))


STM32 secured co-processor
STM32 Level 0 (master)
Protected code + FIREWALL init
FIREWALL init + Bridge master
Flash / EEPROM: 0x0800_0000 0x0800_2FFF
SRAM: 0x2000_0000 0x2000_2FFF
Call gate function
Developer code
Call to the Call gate function
Flash / EEPROM: 0x0800_3000
SRAM: 0x2000_0300
Application code
MS38207V1

In such case, the protected code embeds the SPI API to handle communication with the STM32 secured co-processor device to interact with its protected code.

In this example, the interrupt vectors table (except the reset) has to be relocated elsewhere than the segment 0 into the Flash/EEPROM memory (for instance in 0x0800_3000) to hit the developer code area. Indeed, it is completely linked to the application code.

The start-up file startup_stm32xxx.s needs to be modified to create new area for interrupt vectors excepted for RESET and stack pointer.

```
; Vector Table Mapped to Address 0 at Reset
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size

__Vectors       DCD     __initial_sp              ; Top of Stack
   DCD     Reset_Handler             ; Reset Handler
__Vectors_End

; Relocate the vector table outside of the sector 0.
          AREA    FLASHRELOC, DATA, READONLY

__Vectors_FLASH
```

```
   DCD      __initial_sp              ; Top of Stack
                DCD     Reset_Handler              ; Reset Handler
                DCD     NMI_Handler                ; NMI Handler
                DCD     HardFault_Handler          ; Hard Fault Handler
                DCD     MemManage_Handler          ; MPU Fault Handler
                DCD     BusFault_Handler           ; Bus Fault Handler
                DCD     UsageFault_Handler         ; Usage Fault Handler
                DCD     0                          ; Reserved
                DCD     0                          ; Reserved
                DCD     0                          ; Reserved
                DCD     0                          ; Reserved
                DCD     SVC_Handler                ; SVCall Handler
                DCD     DebugMon_Handler           ; Debug Monitor Handler
                DCD     0                          ; Reserved
                DCD     PendSV_Handler             ; PendSV Handler
                DCD     SysTick_Handler            ; SysTick Handler

                ; External Interrupts
                DCD     WWDG_IRQHandler                 ; Window Watchdog
                DCD     PVD_IRQHandler                  ; PVD through EXTI
Line detect
               DCD     RTC_IRQHandler                ; RTC through EXTI Line
                DCD     FLASH_IRQHandler                ; FLASH
                DCD     RCC_CRS_IRQHandler              ; RCC and CRS
                DCD     EXTI0_1_IRQHandler              ; EXTI Line 0 and 1
   ....
   ....
__Vectors_FLASH_End

__Vectors_FLASH_Size  EQU  __Vectors_FLASH_End - __Vectors_FLASH
```

The scatter file has to refer to this new location for the interrupt vector table:

```
LR_ROM2 0x08003100 0x9F {
  ER_ROM2 0x08003100 0x9F {
  main_user.o (+RO)
  }
}
...
...
LR_ROM4 0x08003000 0xFF {
  ER_ROM4 0x08003000 0xFF {
  startup_stm32l0xx.o (FLASHRELOC, +First)
  }
}
```

Then the user needs to adapt the system_stm32lxxx.c to explicitly configure the MCU to relocate the vector table in the Flash memory or in the EEPROM.

```
#define VECT_TAB_OFFSET  0x3000 /*!< Vector Table base offset field.
                                 This value must be a multiple of 0x200. */


....
  /* Relocate the interrupt vector table */
  SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET; /* Vector Table Relocation in
Internal FLASH */
```

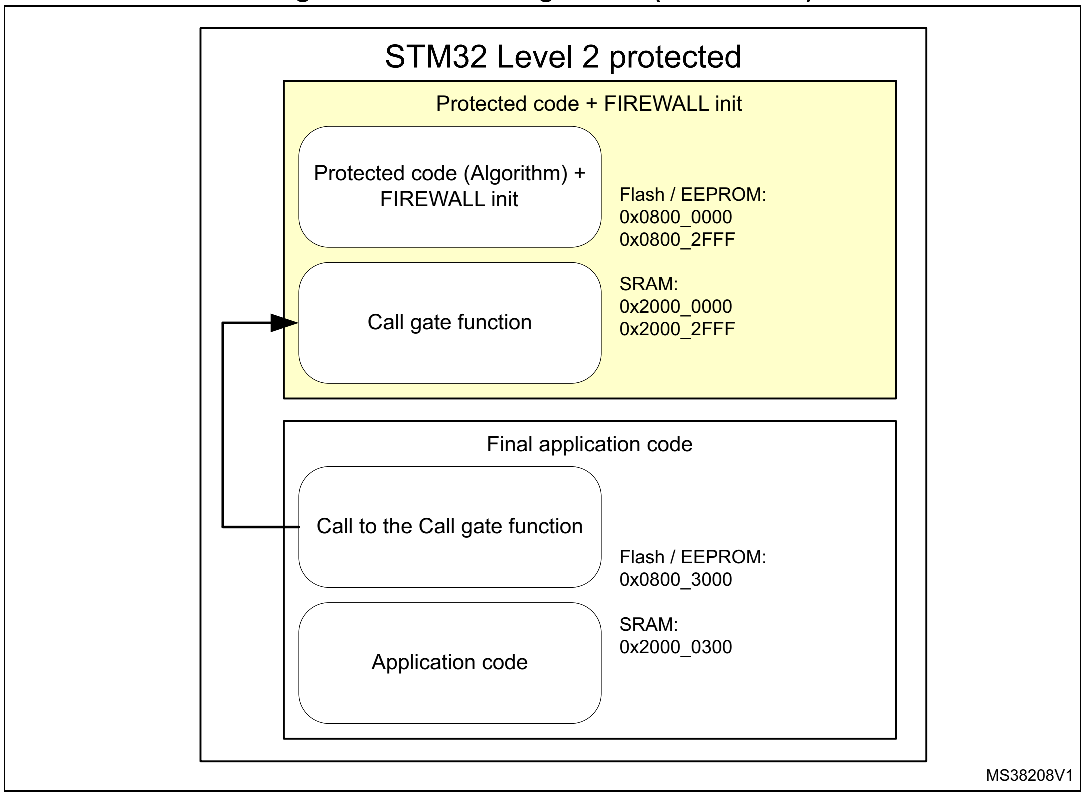
4.3.3 STM32 final production device

When the debug phase is complete, the two devices (secured co-processor + developer device) have to be merged inside a single STM32 MCU for production with Level 2 Read out protection set (RDP level 2), as schematically shown in *Figure 12*.

Having the same mapping constraints for secure code and non-protected one into both MCUs in debug phase makes the merging to a single chip solution easier.

The protected code of the STM32 secured co-processor has to exactly replace the protected code of the STM32 developer device without any need to adapt or change the scatter file in the tool-chain for the compiler.

Figure 12. Final configuration (RDP level 2)



4.3.4 Development process

Here are the main rules to follow to use this working model for debugging sensitive application based on double chips configuration, and to easily merge operation in a single chip for production phase.

- The STM32 co-processor device containing the protected code (level 2 Flash memory protection): decodes the commands sent by the STM32 developer device targeting the protected code and returns the corresponding response. This communication is possible thanks to an interface (e.g. SPI).
- STM32 developer device, which is the device used for full application development containing SPI command parser that sends commands and receives the responses to/from the STM32 co-processor via SPI interface.
- All application code is developed using STM32 co-processor, which launches real calls to the protected code to be made and validated during the development phase in a fully secure way.
- To facilitate ease of development, the third party in charge of the protected code must provide to the developer the memory map, the library API calls and the Call gate mechanism (Call gate function) that is identical for both the STM32 co-processor and the STM32 developer device. The development device also contains the same read, write and FIREWALL protected IP sectors as the STM32 co-processor.
- A 1-to-1 matching of address and API calls allows the final developed code to be placed into parts pre-programmed with the protected code for a single device application.
- The STM32 co-processor approach facilitates the need for the debug of a sensitive application (based on protected code) and prevents exposure of the protected code and of the sensitive data.
- At the end of the development cycle, the final firmware is merged into the targeted STM32 device containing the protected code. The API calls are identical for both STM32 developer device and STM32 co-processor, making the merging easier. Protection code and FIREWALL init code of the STM32 developer device has to be dropped and replaced by the one of the STM32 co-processor, place to place.

Note: *The application developers need to take into consideration latency times created by the transport mechanism between the development device and the co-processor during the debug phase.*

4.4 Software from a third party with dummy protected code during the debug phase

The section illustrates the scenario in which the protected code (accesses secured by the FIREWALL) is developed externally and must not have any visibility from the application software developers. The aim is to avoid delivering the protected code during the debug phase waiting for the last moment to insert it. The OEM then needs to develop a dummy code emulating the final protected code provided by another third party.

When the debug phase is complete, the protection code replaces the dummy code into the targeted STM32 and the Flash memory protection level 2 is set with PCROP protection on the protected code.

In such case, the OEM has to know where the protected code is placed, and how to handle the Call gate function to interact with it. This is an immediate pick and place at the end for final production between the dummy protected code and the targeted one.

5 Specific software / tools recommendations

There are some points to carefully monitor when the developer has to make the protected code under the FIREWALL protection, avoiding unexpected resets during code debug.

The developer choosing to protect the sensitive code with PCROP protection needs to be sure that the code is compiled without any literal pool. If not, the PCROP protection reacts to any read of the protected code, generating a Hard_fault error.

The Call gate function may be a switch case c-statement (in the protected code). If there is no specific option given to the compiler, it may build a switch case code placed in main user area (not protected) if this last is also using switch-case statements. It results in a FIREWALL reset executing the switch case function in the Call gate function. With Arm® compiler, the option *--no_branches_table* may help to avoid that.

Globally, be careful when using functions from the HAL library during the protected code implementation. If these basic functions are called in the protected area while they are also used in the non-protected user code (and compiled here), there is a FIREWALL reset as soon as the function is called into the protected code. It is recommended in such case to use them exclusively into the protected code (and compiling it inside this area). If this is not possible, it is recommended to rewrite the functions directly in the protected code without sharing any code with the non-protected user code.

Sometimes the compiler can create object files from the compiled protected code in the main Flash memory in a non-protected area. It triggers a FIREWALL reset when executing this part of code during the protected code execution (for instance *__aebi_idiv* with associated *uidiv.o* and *idiv.o* object files). In such case, it is possible to declare these object files into the protected code area specifying it explicitly in the scatter file.

There is no possible SFU or firmware upgrade if the FIREWALL is enabled with the read out protection level 2 set associated to the PCROP protection. The non-protected code (without PCROP protection on it) may be updated thanks to a custom embedded IAP (in application protocol).

6 Revision history

Table 3. Document revision history

Date	Revision	Changes
27-Aug-2015	1	Initial release
12-Aug-2019	2	Changed document classification, from ST restricted to Public. Introduced STM32L4+ Series, hence updated document title. Minor text edits across the whole document.