AN4968
Application note

Proprietary code read out protection (PCROP) on STM32F72xxx and STM32F73xxx microcontrollers

Introduction

The software providers are developing complex middleware solutions (Intellectual Propriety code, or IP-code). Protecting them is an issue of high importance for the microcontrollers.

In order to respond to this important requirement, the STM32F72xxx and STM32F73xxx microcontrollers feature:

- The Read Protection (RDP): the protection against read operations
- The Write Protection: the protection against undesired write or erase operations
- The Proprietary Code Read Out Protection (PCROP): the protection against read and write operations.

This application note provides a description of these Flash memory protection techniques, focusing on the Proprietary Code Read Out Protection (PCROP) on STM32F72xxx and STM32F73xxx microcontrollers and providing a basic example of the PCROP protection.

The X-CUBE-PCROP embedded software package is delivered with this document containing the source code of the PCROP example with all the embedded software modules required to run the example.

This application note has to be read in conjunction with *STM32F72xxx and STM32F73xxx advanced ARM®-based 32-bit MCUs* reference manual (RM0431) and the corresponding datasheets available on www.st.com.

Contents

List of tables

List of figures

1 Memory protection description

1.1 Read out protection (RDP)

The Read out Protection is a global Flash memory read protection allowing the embedded software code to be protected against copy, reverse engineering, dumping using debug tools or other means of an intrusive attack. This protection must be set by the user after the binary code is loaded to the embedded Flash memory.

Three RDP levels (0, 1 and 2) are defined and described in the following sections.

1.1.1 Read protection level 0

Level 0 is the default one, the Flash memory is fully open and all the memory operations are possible in all the boot configurations (debug features, boot from the RAM, boot from the system memory bootloader or from the Flash memory). In this mode there is no protection, this mode is for the development and the debug.

1.1.2 Read protection level 1

When the read protection level 1 is activated, no access (read, erase, and program) to the Flash memory or the backup SRAM can be performed via the debug features such as the Serial Wire or the JTAG even while booting from the SRAM or the system memory bootloader.

However when booting from the Flash memory, accesses to the Flash memory and the backup SRAM from the user code are allowed.

Disabling the RDP level 1 protection by re-programming the RDP option byte to level 0 leads to a mass erase.

1.1.3 Read protection level 2

When the RDP level 2 is activated, all the protections provided in level 1 are active and the chip is fully protected. The RDP option byte and all other option bytes are frozen and can no longer be modified. The JTAG, SWV (single-wire viewer), ETM, and boundary scan are disabled.

When booting from the Flash memory, the memory content is accessible to the user code. However, booting from the SRAM or the system memory bootloader is no more possible.

This protection is irreversible (JTAG fuse), so it is impossible to go back to the protection level 1 or 0.

Table 1 describes the different accesses to the Flash memory, backup SRAM, option bytes and One Time Programmable bytes (OTP) when booting from the internal Flash memory, or in debug or boot from the SRAM or the system memory bootloader.

Table 1. Acces status versus protection level and execution modes

Memory area	Protection Level	Debug features, boot from RAM or from System memory bootloader			Booting from Flash memory		
		Read	Write	Erase	Read	Write	Erase
Main Flash Memory and backup SRAM	Level 1	NO		NO[(1)]	YES		
	Level 2	NO			YES		
Option bytes	Level 1	YES			YES		
	Level 2	NO			NO		
OTP	Level 1	NO		NA	YES		NA
	Level 2	NO		NA	YES		NA

1. The main Flash memory and backup SRAM are only erased when the RDP changes from level 1 to 0. The OTP area remains unchanged.

1.1.4 Internal Flash memory content updated on STM32F72xxx and STM32F73xxx RDP protected

When the RDP protection is activated (level 1 or level 2), the internal Flash memory content cannot be updated through the debug or when booting from the SRAM or the system memory bootloader.

An important requirement for the end product is the ability to upgrade the embedded software in the internal Flash memory with new software versions, adding new features and correcting potential issues.

This requirement can be resolved by implementing an user-specific embedded software to perform the In-Application Programming (IAP) of the internal Flash memory. The IAP uses a communication protocol such as the USART for the reprogramming process.

For more details about the IAP, refer to *STM32 in-application programming using the USART* application note (AN3965), available on www.st.com.

1.2 Write protection

The write protection is used to protect the content of specified sectors against a code update or erase. This protection can be applied by sector.

Any write request generates a Write Protection Error. The WRPERR flag is set by hardware when an address to be erased/programmed belongs to a write-protected part of the Flash memory.

For example the mass erase of the Flash memory, where at least one sector is write protected, is not possible and the WRPERR flag is set.

To activate the write protection for each Flash memory sector i, one option bit nWRPi is used. When the write protection is set for the sector i (option bit nWRPi = 0), this sector cannot be erased or programmed.

Enabling or disabling the write protection can be managed either by an embedded user code or by using the STM32 ST-Link Utility software and debug interfaces.

1.3 Proprietary Code Read Out Protection (PCROP)

1.3.1 PCROP protection overview

The PCROP is a read and write protection of an IP-code in the Flash memory. The PCROP is applied by sector (0 to 7) to protect the proprietary code from a possible modification or read out by the end user code, the debugger tools or the RAM Trojan code.

Any read access (other than fetch) to the PCROP-ed sectors, performed through the ITCM or the AXI bus, triggers:

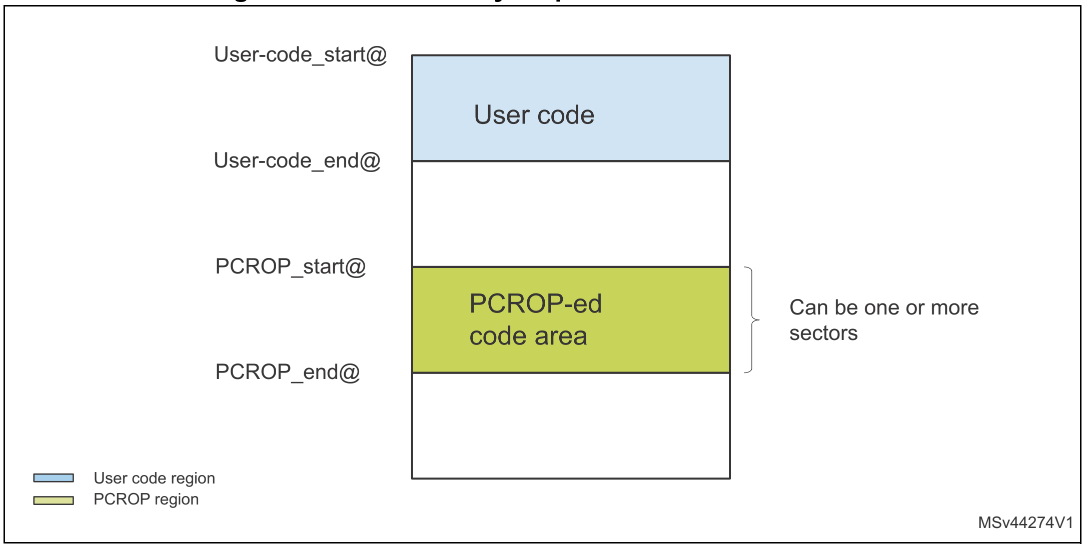
- A bus error on the given bus
- The RDERR flag to be set in the FLASH_SR status register. An interrupt is also generated if the Read Error Interrupt Enable bit (RDERRIE) is set in the FLASH_CR register.

Any program/erase operation on a PCROP-ed sector, triggers a WRPERR flag error.

The protected IP-code can be easily called by the end user application and can still be protected against a direct access to the IP-code itself. Then the PCROP does not prevent the protected codes from being executed.

Caution: While the read access to the PCROP-ed sectors generates a bus error, the Read Error Interrupt can be masked by a Hard Fault Error Interrupt or a Bus Fault Error Interrupt. To use the read error interrupt, the Bus Fault Error Interrupt must be enabled with a priority lower than the priority of the Read Error Interrupt.

Figure 1. Flash memory map with PCROP-ed sectors



1.3.2 How to enable the PCROP protection

The PCROP protection is activated sector by sector, so each sector can independently be a PCROP-ed sector and the protection of additional sectors is possible (when RDP is set to level 0 or level 1).

The PCROP protection is activated through the option bit PCROP[i] in the FLASH_OPTCR2 register:

- PCROP[i] = 0: PCROP protection not active on the sector i (i = 0..7)
- PCROP[i] = 1: PCROP protection active on the sector i (i = 0..7)

To enhance the security level of the PCROP-ed sectors all the debug events are masked while executing the code in the PCROP-ed sectors.

An additional option bit (PCROP_RDP = PCROP1ER [15]) allows to select if the PCROP area is erased or not when the RDP protection is changed from level 1 to level 0.

The bits used to activate the PCROP protection (PCROPi) and the others bits used to activate the write protection nWRPi are independent, thus it is possible to have at the same time, one write protected sector and another PCROP-ed sector. *Table 2* shows what type of protection is set on a sector i according to the WRPi and PCROPi bits values:

Table 2. Protections on sector i

nWRPi	PCROPi	Protection on sector i
1	0	No protection
0	0	Write protection
X	1	PCROP protection

Note: For more details on the PCROP enabling implementation, the user must refer to the PCROP_Enable() function described in the provided FW package (STEP1-ST_Customer_level_n project main.c file).

1.3.3 How to disable the PCROP protection

Depending on the RDP level, the PCROP protection can be disabled if the RDP level is 1 or 0 but when the RDP is set to level 2 the PCROP can no more be disabled. When the RDP is set to level 2, all the option bytes are frozen and cannot be modified. As a result, the PCROP-ed sectors can never be erased or modified and the protection becomes permanent.

The only way to disable the PCROP on a protected sector is:

- To clear the PCROPi bit of the corresponding sectors (multiple sectors could be done at the same time)
- To do a regression level from level 1 to level 0
- To have the PCROP_RDP bit already set

If the PCROP_RDP bit is set, a mass erase of the main Flash memory is performed and depending on the PCROPi value, the PCROP protection is kept enabled or disabled for the corresponding sector.

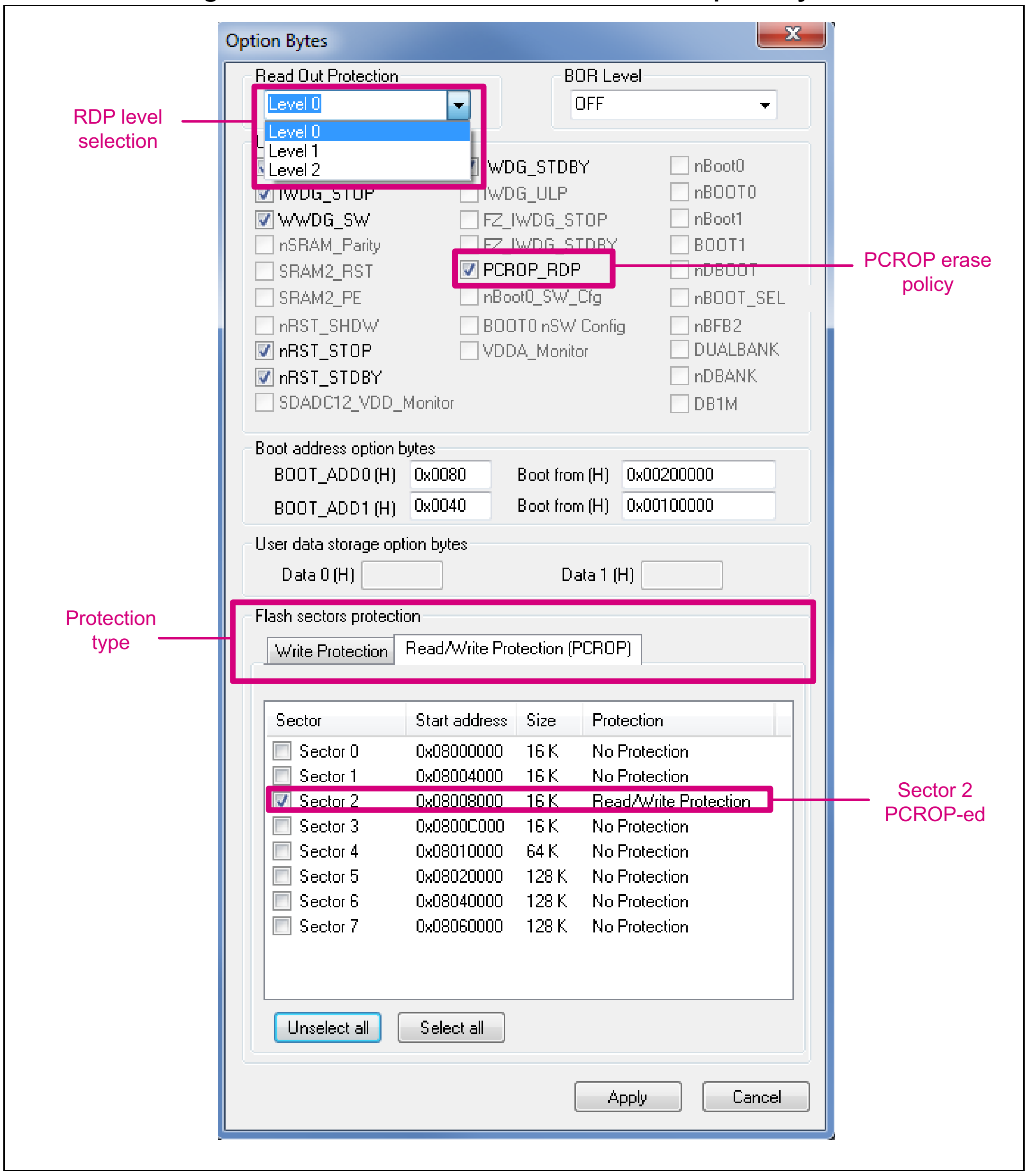
If the PCROP_RDP bit is cleared, the full mass erase is replaced by a partial mass erase that is successive page erases in the bank where PCROP is active, except for the pages protected by PCROP. This is done in order to keep the PCROP code.

Note: *Doing a regression level from level 1 to level 0 at the same time than modifying the PCROP_RDP bit is allowed. As the full or partial mass erased is launched before an option modification, the current PCROP_RDP bit is used and not the new PCROP_RDP bit being programmed.*

Use STM32-Link Utility

During the application development the user may need to disable the PCROP or the global RDP protection without spending time in developing and disabling the protection functions. The STM32 ST-LINK Utility tool can be a very simple way for disabling or enabling protection using debug interfaces as the JTAG or the SWD without the need for developing dedicated functions. *Figure 2* shows how to modify option bytes.

Figure 2. User interface for modification of option bytes



Read Out Protection: Level 0

BOR Level: OFF

	✓ WDG_STDBY	nBoot0
✓ IWDG_STOP	WDG_ULP	nBOOT0
✓ WWDG_SW	FZ_IWDG_STOP	nBoot1
nSRAM_Parity	FZ_IWDG_STDBY	BOOT1
SRAM2_RST	✓ PCROP_RDP	nDBOOT
SRAM2_PE	nBoot0_SW_Cfg	nBOOT_SEL
nRST_SHDW	BOOT0 nSW Config	nBFB2
✓ nRST_STOP	VDDA_Monitor	DUALBANK
✓ nRST_STDBY		nDBANK
SDADC12_VDD_Monitor		DB1M

Boot address option bytes

BOOT_ADD0 (H)	0x0080	Boot from (H)	0x00200000
BOOT_ADD1 (H)	0x0040	Boot from (H)	0x00100000

User data storage option bytes

Data 0 (H) Data 1 (H)

Flash sectors protection

Write Protection | Read/Write Protection (PCROP)

Sector	Start address	Size	Protection
Sector 0	0x08000000	16 K	No Protection
Sector 1	0x08004000	16 K	No Protection
✓ Sector 2	0x08008000	16 K	Read/Write Protection
Sector 3	0x0800C000	16 K	No Protection
Sector 4	0x08010000	64 K	No Protection
Sector 5	0x08020000	128 K	No Protection
Sector 6	0x08040000	128 K	No Protection
Sector 7	0x08060000	128 K	No Protection

For more details on how to use the STM32 ST-LINK Utility software, the user must refer to *STM32 ST-LINK utility software description* user manual (UM0892) available on *www.st.com*.

1.3.4 Placing and executing PCROP-ed IP-code

As previously mentioned, the PCROP does not prevent a protected IP-code from being executed and its functions can be easily called by the user code.

The PCROP-ed sectors are protected against D-code bus read accesses, it is important to mention here that only the code execution is allowed (instruction fetch through IP-code bus) while the data reading is forbidden. Therefore the protected IP-code is unable to access the associated data values stored in the same area (such as literal pools, branch tables or constants which are fetched from the Flash memory through the D-code bus during the execution).

The PCROP-ed code must be an execute-only code and must not contain any data.

The user must configure the compiler to generate an execute-only IP-code avoiding any data read, the necessary compiler configurations are detailed in *Section 2*.

Do not activate PCROP protection for the vector table sector

The interrupt vector table contains entry point addresses of each interrupt handler, which are read by the CPU through the D-code bus. In general the interrupt vector table is located in the first sector at the first address 0x08000000 (except in some cases where it is relocated in other region, as the SRAM).

When placing the code to be protected in the Flash memory, the following rules must be respected:

- The first Flash memory sector where the vector table is located must not be a PCROP-ed sector
- The PCROP-ed code must not be placed in the first sector.

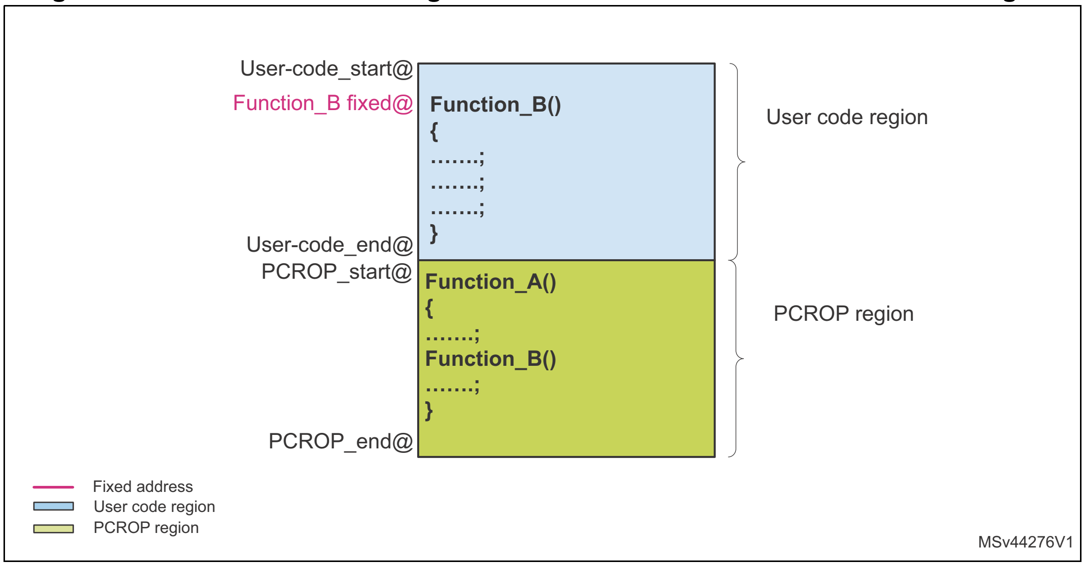
PCROP-ed IP-code dependency

The protected IP-code can call functions from libraries located in the user code region and outside of the PCROP-ed area. In this case the IP-code contains the related functions addresses allowing the PC (Program Counter) to jump to these functions when executing the IP-code. These addresses are unchangeable once the IP-code is a PCROP-ed code. Consequently each called function must be located (outside of PCROP-ed region) at its corresponding fix address written in the PCROP-ed IP-code else, the PC jumps to an invalid address and the IP-code does not work correctly.

To be fully independent, the protected IP-code must be placed together with all its related functions.

Figure 3 shows an example where the PCROP-ed Function_A() is calling a Function_B() located at a fixed address outside the PCROP-ed region.

Figure 3. PCROP-ed code calling a function located outside the PCROP-ed region



2 PCROP example

The X-CUBE-PCROP embedded software example provided with this application note illustrates a use case of the PCROP protection. All the steps required to develop this embedded software are detailed in this section.

2.1 Example requirements

2.1.1 Hardware requirements

The hardware required to run this example is the following:

- STM32F722ZE-Nucleo board RevB
- A mini-USB cable to power the board and to connect the discovery embedded STLINK for debugging and programming

2.1.2 Software requirements

The following software tools are required:

- IAR Embedded Workbench® or Keil® µvision IDE or SW4STM32
- STM32 STLink Utility is required mainly for enabling or disabling the PCROP protection.

2.2 Example overview

2.2.1 Scenario overview

This example describes a use case where an ST Customer level n provides preprogrammed STM32F722ZE MCUs with a critical IP-code to an ST Customer level n+1.

The IP-code must be protected by activating the PCROP.

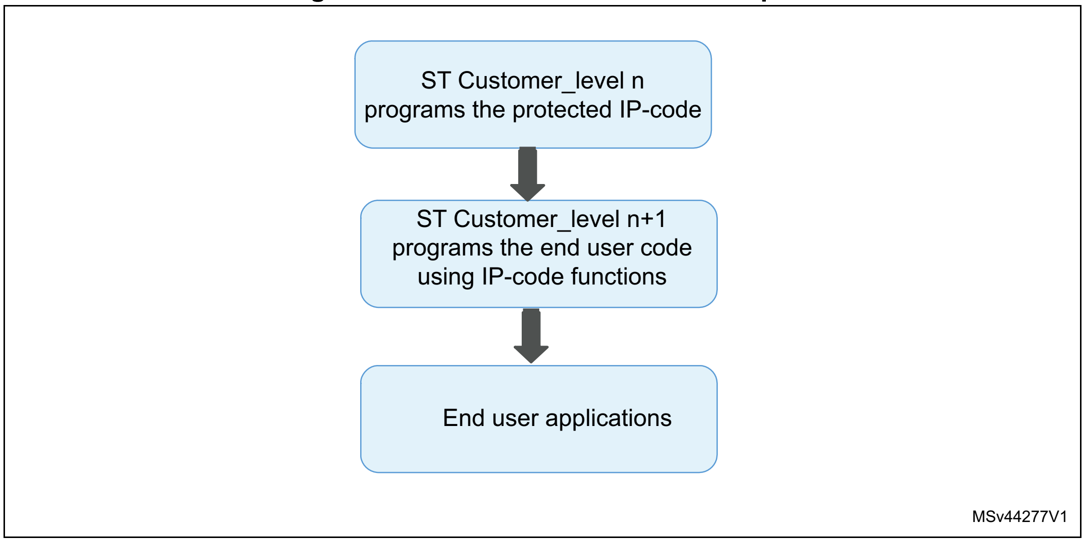
It allows the ST Customer level n+1 to use the IP-code functions (without the ability to read or modify them) and to program the End user application.

The ST Customer level n must then provides, together with a preloaded STM32 MCU, the following inputs:

- The Flash and RAM memories maps, defining the exact PCROP-ed IP-Code location and the available sectors for programming.
- The header file that has to be included in the ST Customer level n+1 project containing the IP-Code function definition to be called in the End user code.
- The symbol definition file containing the PCROP-ed IP-Code function symbols.

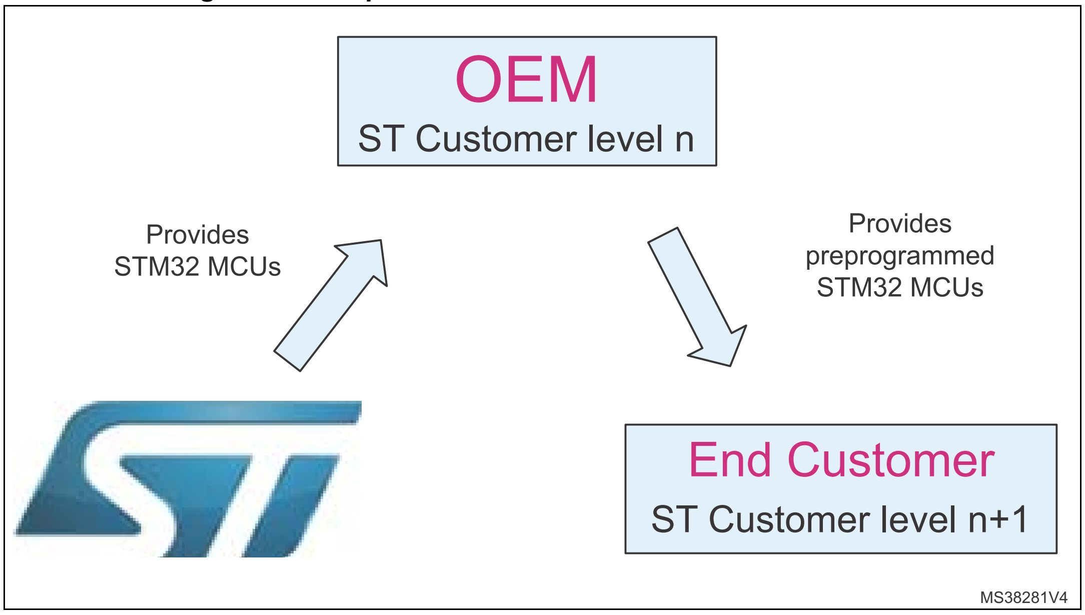
The described use case is schematized in *Figure 4*.

Figure 4. STM32F7 PCROP flow example


ST Customer_level n programs the protected IP-code
ST Customer_level n+1 programs the end user code using IP-code functions
End user applications
MSv44277V1

An OEM (Original Equipment Manufacturer) can be the ST Customer level n using STM32F7 microcontrollers. The OEM provides preprogrammed MCUs to the ST Customer level n+1 that can be the End Customer making the end user product, as illustrated in *Figure 5*.

Figure 5. Example of an ST customer level n and level n+1



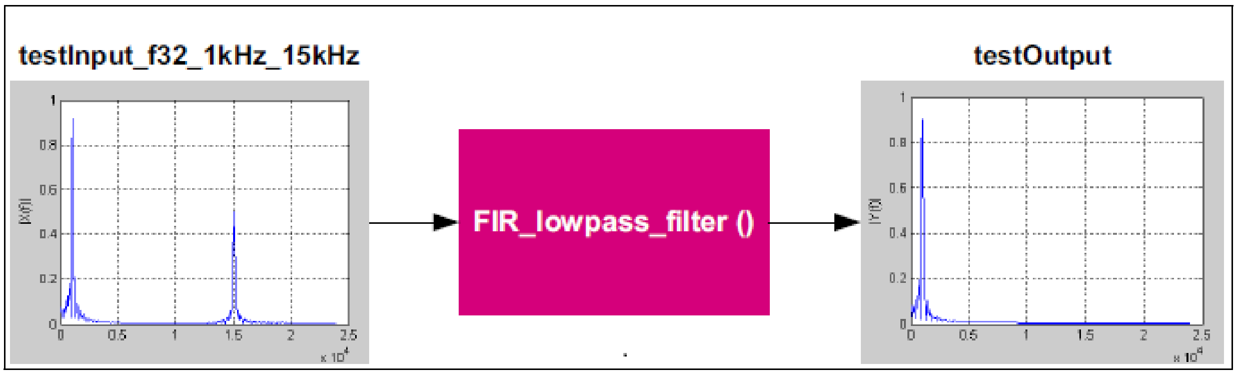
2.2.2 PCROP-ed IP-code: FIR lowpass filter

As example of an IP-code to be protected, a FIR lowpass filter algorithm from the CMSIS-DSP is described, focusing on how to protect and call this IP-code function, without providing details on the functions themselves.

The FIR lowpass filter removes high frequency signal components from the input.

The input signal is a sum of two sine waves having frequencies of 1 KHz and 15 KHz. The lowpass filter (with its preconfigured cutoff frequency set at 6 KHz) eliminates the 15 KHz signal leaving the 1 KHz sine wave at the output.

Figure 6. FIR lowpass filter function block diagram



The CMSIS DSP software library functions used are:

- arm_fir_init_f32(): the initialization function to configure the filter, described in arm_fir_init_f32.c file;
- arm_fir_f32(): the elementary function representing the FIR Filter, described in arm_fir_f32.c file.

The following function is created using the CMSIS DSP functions described above:

- FIR_lowpass_filter(): the global function representing the FIR Filter, described in the fir_filter.c file.

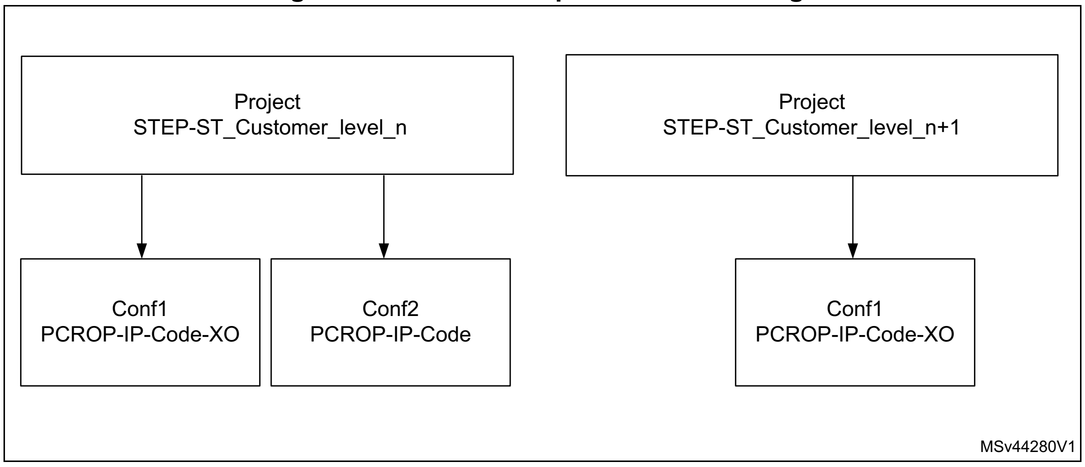
The FPU and DSP embedded in the STM32F72xxx and STM32F73xxx microcontrollers are used for signal processing and floating point calculation to output the correct signal.

For more details on FIR functions the user must refer to the CMSIS documentation in "Drivers/CMSIS/Documentation/DSP" directory included in the associated software package.

2.2.3 Software settings

This application note is provided with two projects:

Figure 7. PCROP example software settings



- **Project 1: STEP1-ST_Customer_level_ n**
  This project shows an example of how an ST Customer level n can place, protect and execute its IP-code and how to generate the IP-code related files as header and symbol definition files to be provided to ST customer level n+1. This project includes two different project configurations:
  - PCROP-IP-code-XO: in this configuration the compiler is configured to generate an execute-only IP-code avoiding any data read from it.
  - PCROP-IP-code: in this configuration the IP-code is compiled without avoiding data (literal pools) generation. This configuration is dedicated for testing purpose, in order to show that the PCROP-ed IP-code must be an execute-only code.
- **Project 2: STEP2-ST_Customer_level_n+1**
  This project shows an example of how an ST Customer level n+1 having a preprogrammed STM32F722ZE MCU with a PCROP-ed IP-code, can create its own end user application using these protected IP-code functions.

2.3 STEP1: ST Customer level n

At this stage the ST Customer level n:

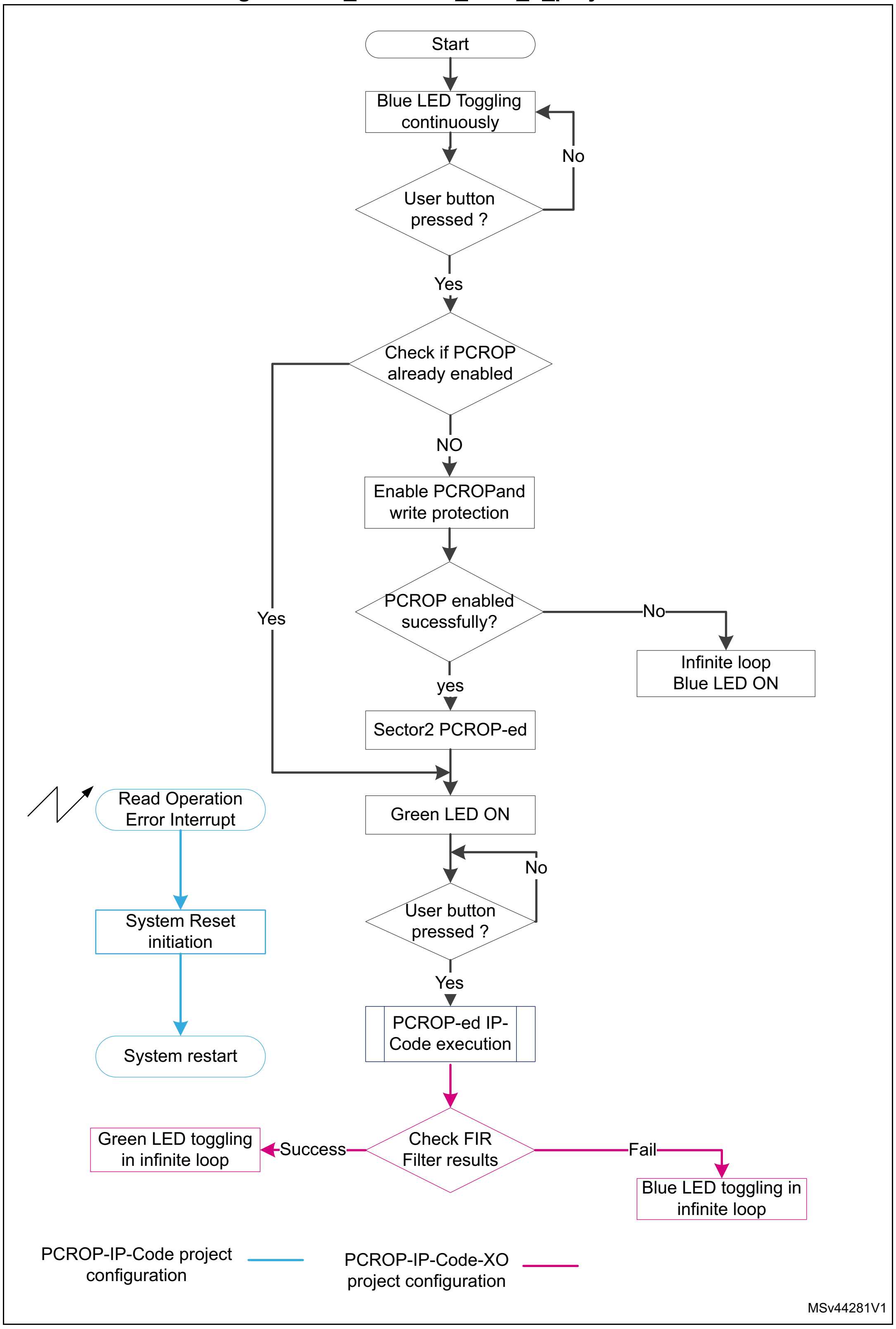
- Generates an execute-only IP-code
- Places the IP-code and data segments at a specified location in the Flash memory
- Protects the IP-code area with PCROP
- Write-protects data segment
- Executes the IP-code by calling its functions in main code
- Creates a header file and generates the symbol definition file to be used in the STEP2-ST_Customer_level_n+1 project.

2.3.1 Project flow

Figure 8 illustrates the STEP1-ST_Customer_level_n project flow with both project configurations:

- PCROP-IP-code-XO (in pink): the PCROP-ed IP-code does not contain any literal pool, then the FIR-Filter algorithm runs successfully noting that a Read Operation Error interrupt is generated and a system reset is initiated if any read/write request occurs from/to the PCROP-ed region.
- PCROP-IP-code (in blue): the IP-code contains literal pools, when starting the PCROP-ed IP-code execution, a Read Operation Error interrupt is generated, then a system reset is initiated and the system is restarted.

Figure 8. ST_customer_level_n_project flow



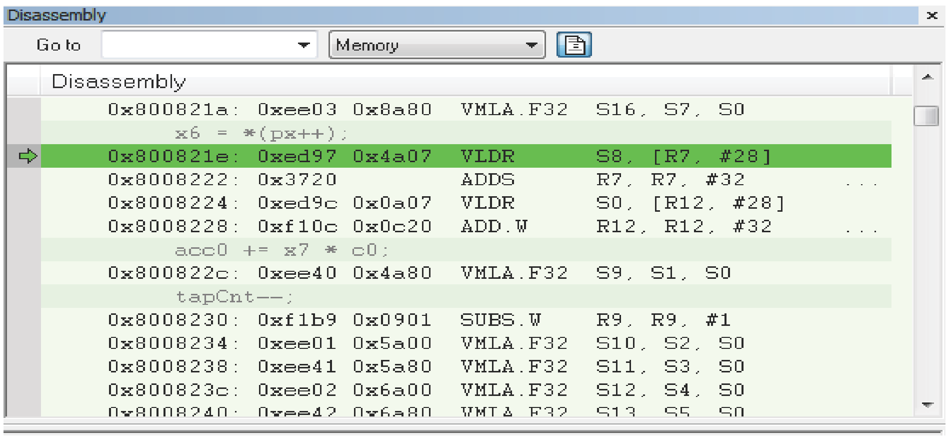
Start
Blue LED Toggling continuously
No
User button pressed ?
Yes
Check if PCROP already enabled
NO
Yes
Enable PCROPand write protection
PCROP enabled sucessfully?
No
Infinite loop Blue LED ON
yes
Sector2 PCROP-ed
Read Operation Error Interrupt
Green LED ON
No
System Reset initiation
User button pressed ?
Yes
PCROP-ed IP-Code execution
System restart
Green LED toggling in infinite loop
Success
Check FIR Filter results
Fail
Blue LED toggling in infinite loop
PCROP-IP-Code project configuration
PCROP-IP-Code-XO project configuration
MSv44281V1

2.3.2 Generating an execute-only IP-code

Each toolchain has its own options to prevent the compiler from generating literal pools and branch tables. For instance the Keil® has the “Execute-only code” option while the IAR and SW4STM32 have the “No data reads in code memory” option.

Figure 9 shows an assembler code containing literal pools, where the instruction has the form of VLDR <variable>, [PC + <offset>

Figure 9. Example of assembler code containing literal pools



```
Disassembly
    0x800821a: 0xee03 0x8a80  VMLA.F32   S16, S7, S0
         x6 = *(px++);
    0x800821e: 0xed97 0x4a07  VLDR       S8, [R7, #28]
    0x8008222: 0x3720         ADDS       R7, R7, #32        ...
    0x8008224: 0xed9c 0x0a07  VLDR       S0, [R12, #28]
    0x8008228: 0xf10c 0x0c20  ADD.W      R12, R12, #32      ...
         acc0 += x7 * c0;
    0x800822c: 0xee40 0x4a80  VMLA.F32   S9, S1, S0
         tapCnt--;
    0x8008230: 0xf1b9 0x0901  SUBS.W     R9, R9, #1
    0x8008234: 0xee01 0x5a00  VMLA.F32   S10, S2, S0
    0x8008238: 0xee41 0x5a80  VMLA.F32   S11, S3, S0
    0x800823c: 0xee02 0x6a00  VMLA.F32   S12, S4, S0
    0x8008240: 0xee42 0x6a80  VMLA.F32   S13, S5, S0
```

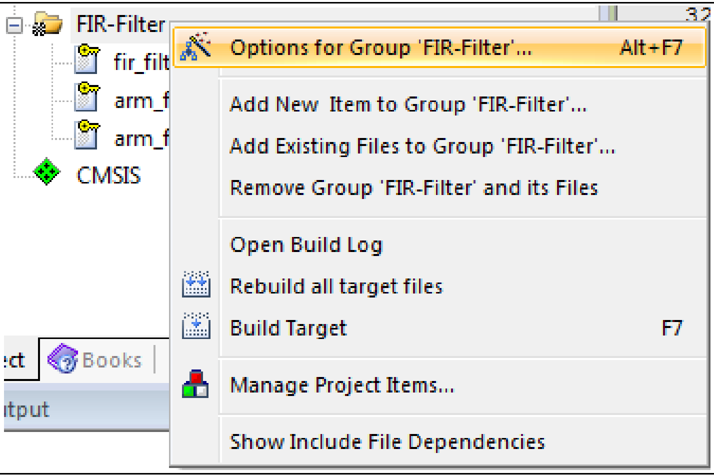
Keil®: Using Execute-only command

The compilation option “Execute-only” must be used to generate a code without literal pools, preventing the compiler from generating any data accesses to code sections.

The option has to be used for all the IP-code files containing literal pools.

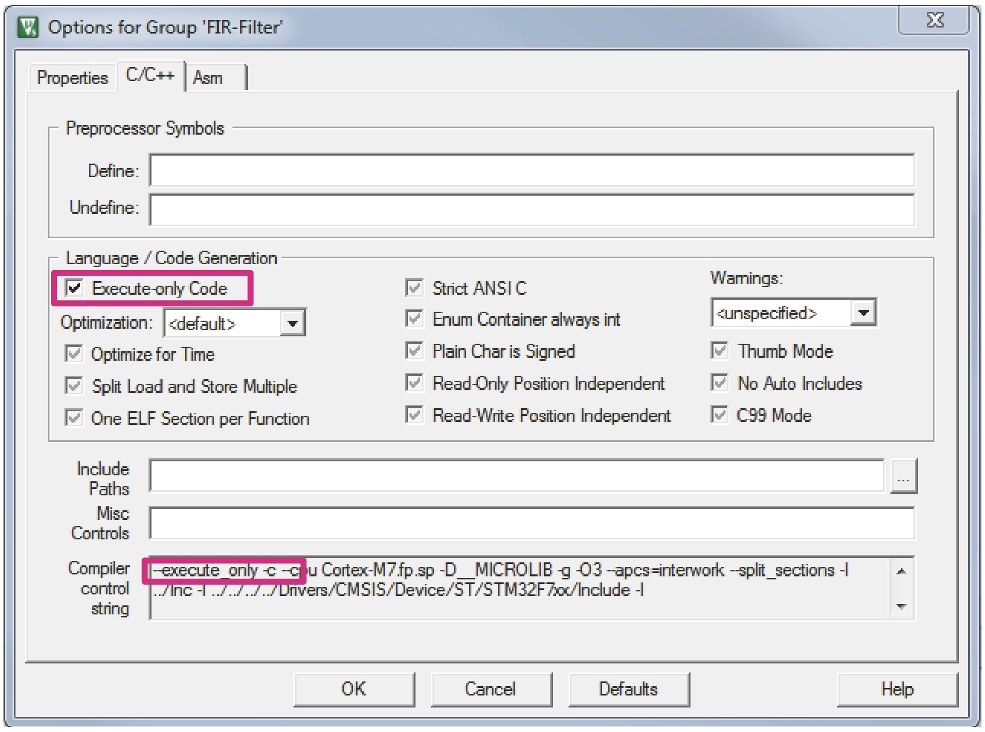
To set this command right click on the group file or the IP-code and (see *Figure 10*) choose “Options for Group 'User/Fir'”:

Figure 10. Accessing the FIR filter options


FIR-Filter
fir_filt
arm_f
arm_f
CMSIS
Options for Group 'FIR-Filter'... Alt+F7
Add New Item to Group 'FIR-Filter'...
Add Existing Files to Group 'FIR-Filter'...
Remove Group 'FIR-Filter' and its Files
Open Build Log
Rebuild all target files
Build Target F7
Manage Project Items...
Show Include File Dependencies
Books
tput

In the following window (see *Figure 11*) check the “Execute-only Code” option, then the execute_only command is added in the compiler control string field.

Figure 11. Setting the execute-only code option



Preprocessor Symbols

Define:

Undefine:

Language / Code Generation

- [✓] Execute-only Code
- Optimization: <default>
- [✓] Optimize for Time
- [✓] Split Load and Store Multiple
- [✓] One ELF Section per Function
- [✓] Strict ANSI C
- [✓] Enum Container always int
- [✓] Plain Char is Signed
- [✓] Read-Only Position Independent
- [✓] Read-Write Position Independent
- Warnings: <unspecified>
- [✓] Thumb Mode
- [✓] No Auto Includes
- [✓] C99 Mode

Include Paths

Misc Controls

Compiler control string:

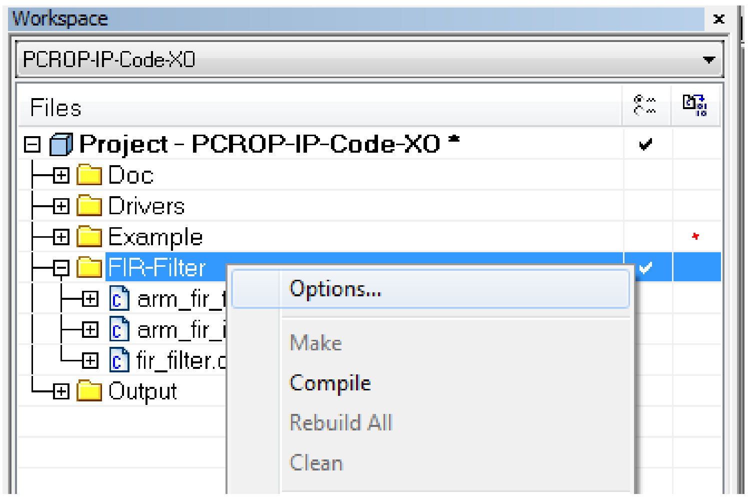
```
--execute_only -c --cpu Cortex-M7.fp.sp -D__MICROLIB -g -O3 --apcs=interwork --split_sections -I
../Inc -I ../../../../Drivers/CMSIS/Device/ST/STM32F7xx/Include -I
```

Then the scatter file that defines the memory region placement of the IP-code sets the access type attribute to +XO. Refer to *Section 2.3.3: Placing IP-code and data segments in Flash and RAM memories*.

IAR: No data reads in code memory

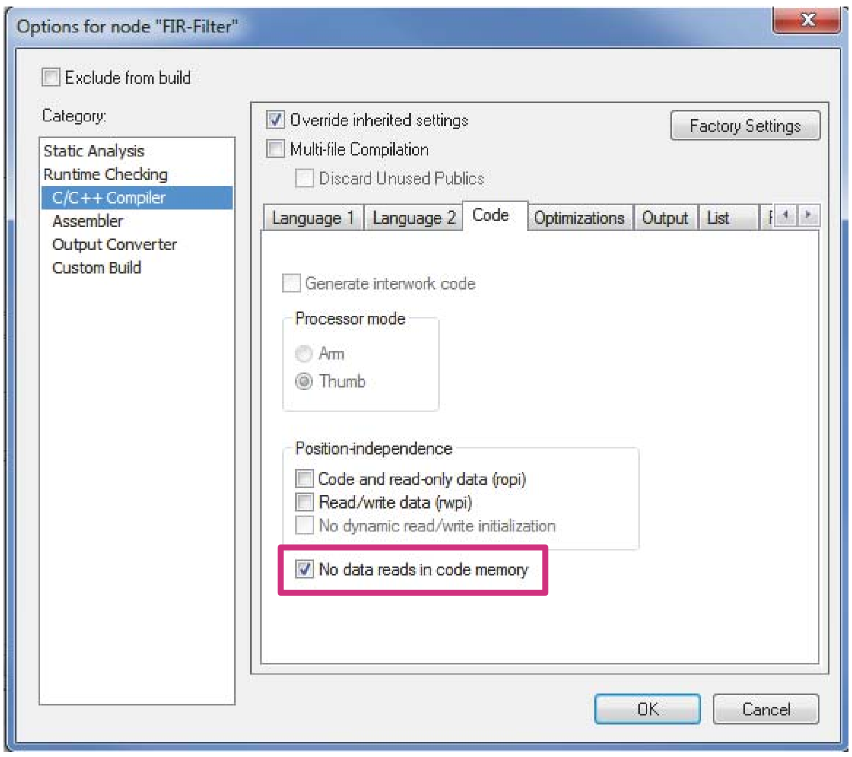
For the IAR the "No data reads in code memory" option must be used to prevent the compiler from generating any data accesses to code sections. To activate this option, right click on the file group “FIR-Filter” containing IP-code source files then (see *Figure 12*) select “Options”.

Figure 12. Accessing to FIR-filter options



Then, in the following window check the option "No data reads in code memory", as indicated in *Figure 13*.

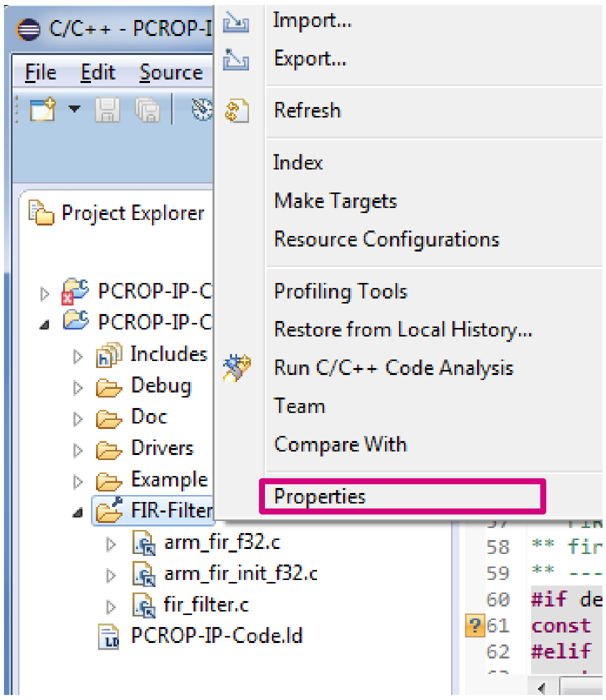
Figure 13. Setting option “no data reads in code memory”



SW4STM32: No data reads in code memory

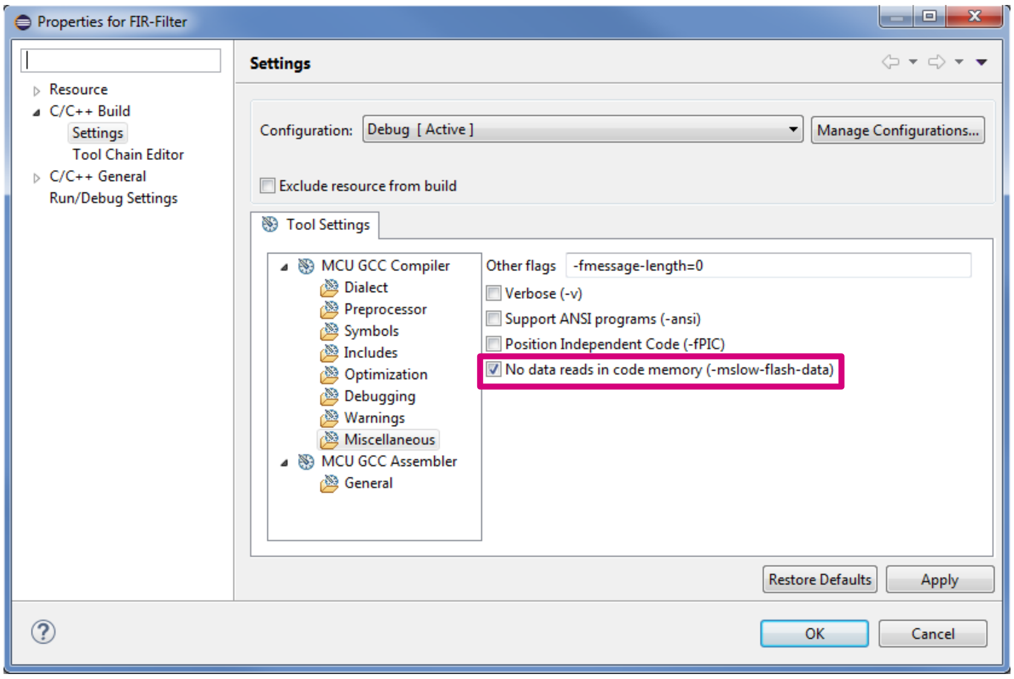
The same option "No data reads in code memory" must be used with SW4STM32 to prevent the compiler from generating any data accesses to code sections. To activate this option, right click on the file group “FIR-Filter” containing IP-code source files then (see *Figure 14*) select “Properties”.

Figure 14. Accessing to FIR-filter option”



Then, in the following window check the option "No data reads in code memory", as indicated in *Figure 15*.

Figure 15. Setting option “no data reads in code memory”



Configuration: Debug [Active]

☐ Exclude resource from build

Other flags -fmessage-length=0

☐ Verbose (-v)

☐ Support ANSI programs (-ansi)

☐ Position Independent Code (-fPIC)

☑ No data reads in code memory (-mslow-flash-data)

2.3.3 Placing IP-code and data segments in Flash and RAM memories

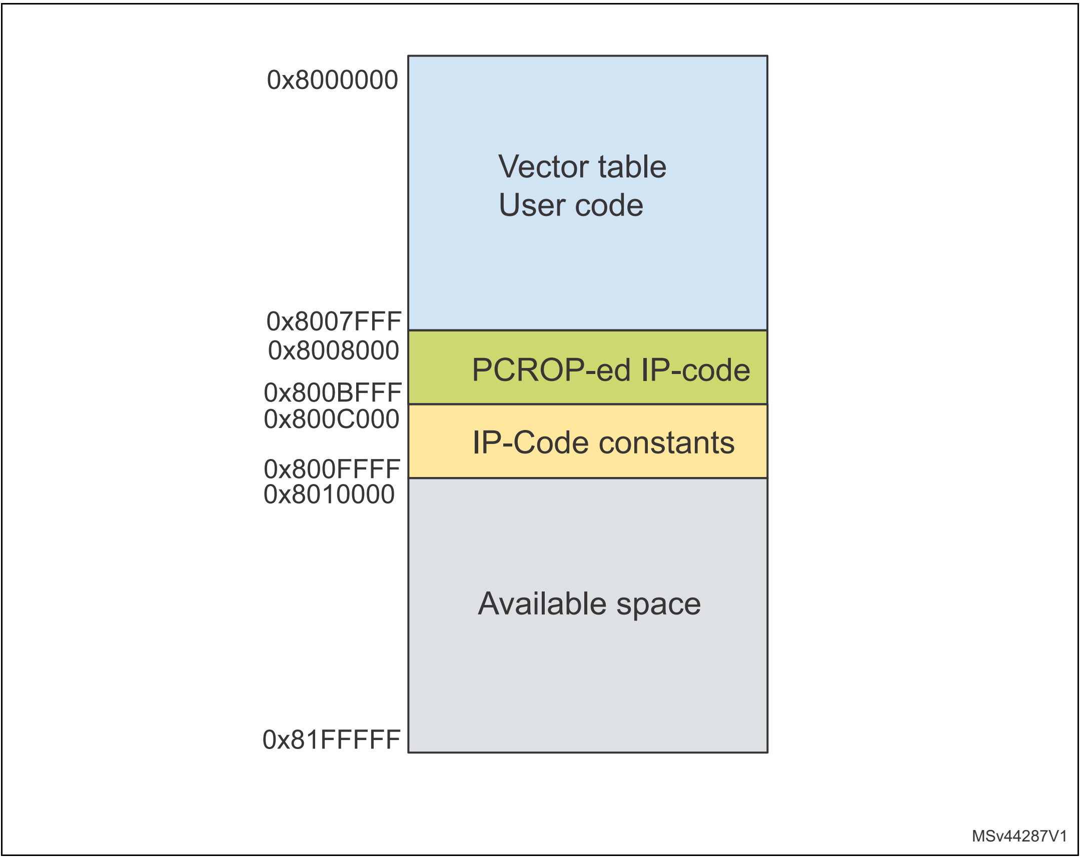
In this example, two dedicated memory sections are defined on the Flash memory:

- IP-code segment located in the Flash memory bank 1 at address 0x0800_8000
- Constant data (FIR filter coefficients) located just after the code section at address 0x0800_C000.

The main memory region for the user code and the vector table are located at the beginning

The resulting Flash memory map is shown in *Figure 16*.

Figure 16. STM32F722ZET6 internal Flash memory map

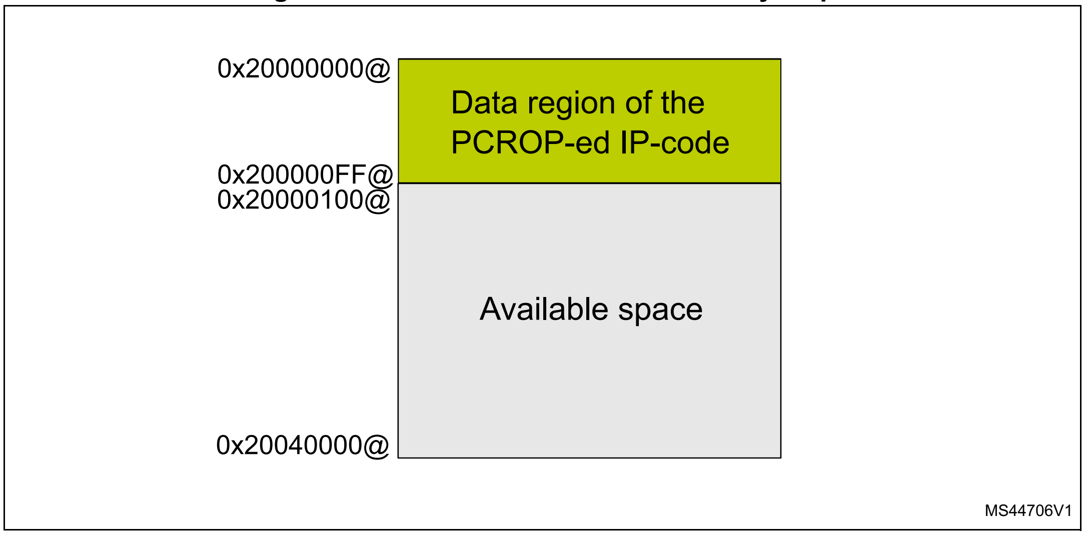


The memory placement is handled by the scatter file for Keil® IDE, the linker configuration file (ICF) in IAR and the linker file in SW4STM32.

Another memory region must be created on the RAM memory dedicated for RW and zero initialized data of the PCROP-ed IP-Code. These data have the same addresses on level n and level n+1, so this same memory region must be reserved for them on the two levels to avoid overwriting data of the main program of level n+1.

The resulting memory map is shown in *Figure 17*.

Figure 17. STM32F722ZET6 RAM memory map

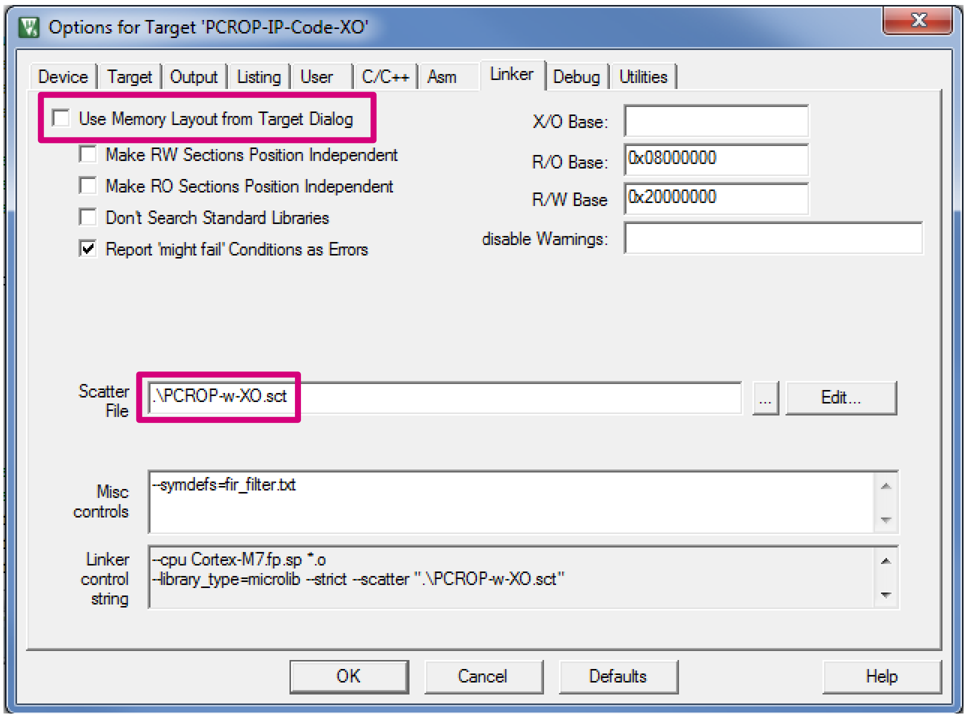


Keil® scatter file

To place the IP-code in Sector 2 follow the sequence below:

Go to Project → Options for Target, select the Linker tab, uncheck “Use Memory Layout from Target Dialog” as shown in *Figure 18*, then click on Edit button to modify the PCROP-w-XO.sct scatter file.

Figure 18. Scatter file modification



- [ ] Use Memory Layout from Target Dialog
- [ ] Make RW Sections Position Independent
- [ ] Make RO Sections Position Independent
- [ ] Don't Search Standard Libraries
- [✓] Report 'might fail' Conditions as Errors

X/O Base:
R/O Base: 0x08000000
R/W Base 0x20000000
disable Warnings:

Scatter File: .\PCROP-w-XO.sct

Misc controls: --symdefs=fir_filter.txt

Linker control string:
```
--cpu Cortex-M7.fp.sp *.o
--library_type=microlib --strict --scatter ".\PCROP-w-XO.sct"
```

The scatter file must be updated with the two new regions:

- LR_PCROP located in the Flash memory with execute-only access attribute for the PCROP-ed IP-code
- LR_DATA located in the Flash memory for the constant data memory region used by the IP-code. A new section is defined, namely "fir_coeff_section".
- FirBSSRAM located in the RAM memory for the data coming from the PCROP-ed section

```
; *************************************************************
; *** Scatter-Loading Description File generated by uVision ***
; *************************************************************
LR_IROM1 0x08000000 0x00004000  {    ; load region size_region
  ER_IROM1 0x08000000 0x00004000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
  }
   FirBSSRAM 0x20000000 0x100 ; RW and Zero Initialized data for fir filter
  {
   fir_filter.o (+RW +ZI)
  }
  RW_IRAM1 0x20000100 0x0003FF00  {  ; RW data of the main program
   .ANY (+RW +ZI)
  }
}
; ********************
; *** PCROP-ed area ***
; ********************
LR_PCROP 0x08008000 0x00004000  {
  ER_PCROP 0x08008000 0x00004000  {  ; load address = execution address
   arm_fir_init_f32.o (+XO)
   arm_fir_f32.o (+XO)
   fir_filter.o (+XO)
  }
}
; ************************************************
; *** Write protected area for FIR coefficients ***
; **** ********************************************
LR_DATA 0x0800C000 0x00004000  {
  fir_coef_section 0x0800C000 0x00004000  {
   FIR_Filter.o (+RO)
```

IAR ICF file

Three memory regions are defined

- LR_PCROP located in the Flash memory with execute-only access attribute for the PCROP-ed IP-code
- LR_DATA located in the Flash memory for the constant data memory region used by the IP-code. A new section is defined, namely "fir_coeff_section".
- FirBSSRAM located in the RAM memory for the data coming from the PCROP-ed section

Note that the compiled object (fir_filter.o) contains the code and global constant data. A split between the code and data is explicit in the ICF file.

```
/* define FirBSSRAM area start and end address */
define symbol __ICFEDIT_region_FirBSSRAM_start__  = 0x20000000;
define symbol __ICFEDIT_region_FirBSSRAM_end__   = 0x200000FF;
define region FirBSSRAM_region    = mem:[from
__ICFEDIT_region_FirBSSRAM_start__   to __ICFEDIT_region_FirBSSRAM_end__];

/* define PCROP area start and end address */
define symbol __ICFEDIT_region_PCROP_start__    = 0x08008000;
define symbol __ICFEDIT_region_PCROP_end__      = 0x0800BFFF;
define region PCROP_region    = mem:[from __ICFEDIT_region_PCROP_start__
to __ICFEDIT_region_PCROP_end__];

/* define write protected area of constant data for fir coefficients */
define symbol __ICFEDIT_region_CONST_start__    = 0x0800C000;
define symbol __ICFEDIT_region_CONST_end__      = 0x0800C800;
define region fir_coef_region   = mem:[from __ICFEDIT_region_CONST_start__
to __ICFEDIT_region_CONST_end__];

/* Place IP-code in sector 2 which will be PCROP-ed */
place in PCROP_region   { ro object arm_fir_f32.o,
                          ro object arm_fir_init_f32.o,
                          ro object FIR_Filter.o};

/* Place constant data used by the IP-code in sector 3 which will be WP-ed
*/
place in fir_coef_region { ro data section fir_coef_section object
FIR_Filter.o };

/* Place RW and ZI data of PCROP-ed section of FirBSSRAM memory region*/
place in FirBSSRAM_region   {
rw object FIR_Filter.o
zi  object FIR_Filter.o};
```

SW4STM32 linker file

Three memory regions are defined

- LR_PCROP located in the Flash memory with execute-only access attribute for the PCROP-ed IP-code
- LR_DATA located in the Flash memory for the constant data memory region used by the IP-code. A new section is defined, namely "fir_coeff_section".
- FirBSSRAM located in the RAM memory and dedicated for the data coming from the PCROP-ed section

```
/* specify the memory areas */
MEMORY
{
RAM (xrw)       : ORIGIN = 0x20000100, LENGTH = 256K - 0x100
FirBSSRAM (xrw): ORIGIN = 0x20000000, LENGTH = 0x100
FLASH (rx)      : ORIGIN = 0x08000000, LENGTH = 16K
PCROP (x)       : ORIGIN = 0x08008000, LENGTH = 16K
CONST (rx)      : ORIGIN = 0x0800C000, LENGTH = 16K
}
  /* Place IP-code in sector 2 which will be PCROP-ed */
.PCROPedCode :
{
  . = ALIGN(4);
   *arm_fir_f32.o  (.text .text*)
   *arm_fir_init_f32.o (.text .text*)
   *fir_filter.o (.text .text*)
  . = ALIGN(4);
  } >PCROP
    /* Place constant data used by the IP-code in sector 3 which will be WP-
ed */
  .WPedData :
  {
  . = ALIGN(4);
   *fir_filter.o (.fir_coef_section .rodata*)
  . = ALIGN(4);
  } >CONST
/* Place RW and ZI data of PCROP-ed section of FirBSSRAM memory region*/
.FirBss :
  {    _sFirbss = .;          /*define a global symbol at FirrBss start */
   *fir_filter.o (.bss .bss*)
    . = ALIGN(4);
    _eFirbss = .;          /*define a global symbol at FirBss end */
  } >FirBSSRAM
```

Explicit data placement

The constant data are mapped on this section thanks to the next compilation attribute. The following code extract shows the syntax for IAR, SW4STM32 and Keil®.

```
/* IAR IDE */
/* Placement with specific address: */
const float32_t firCoeffs32[NUM_TAPS] @ 0x0800C000 = { …};
/* Placement with specific section defined in ICF file: */
const float32_t firCoeffs32[NUM_TAPS] @ "fir_coef_section" = { ….};
/* KEIL IDE */
/* Placement with specific address: */
const float32_t firCoeffs32[NUM_TAPS] __attribute__((at(0x0800C000))) ={…};
/* Placement with specific section defined in scatter file: */
const float32_t firCoeffs32[NUM_TAPS]
__attribute__((section("fir_coef_section"))) = {…};
/* SW4STM32 IDE */
/* Placement with specific address: */
const float32_t firCoeffs32[NUM_TAPS]
__attribute__((section(".0x0800C000"))) ={…};
/* Placement with specific section defined in Id file: */
const float32_t firCoeffs32[NUM_TAPS]
__attribute__((section(".fir_coef_section"))) = {…};
```

2.3.4 Write protection of constants

All the IP-code constants (for example: integer, float or strings) have to be placed outside of the PCROP-ed region as they cannot be accessed by the D-code bus.

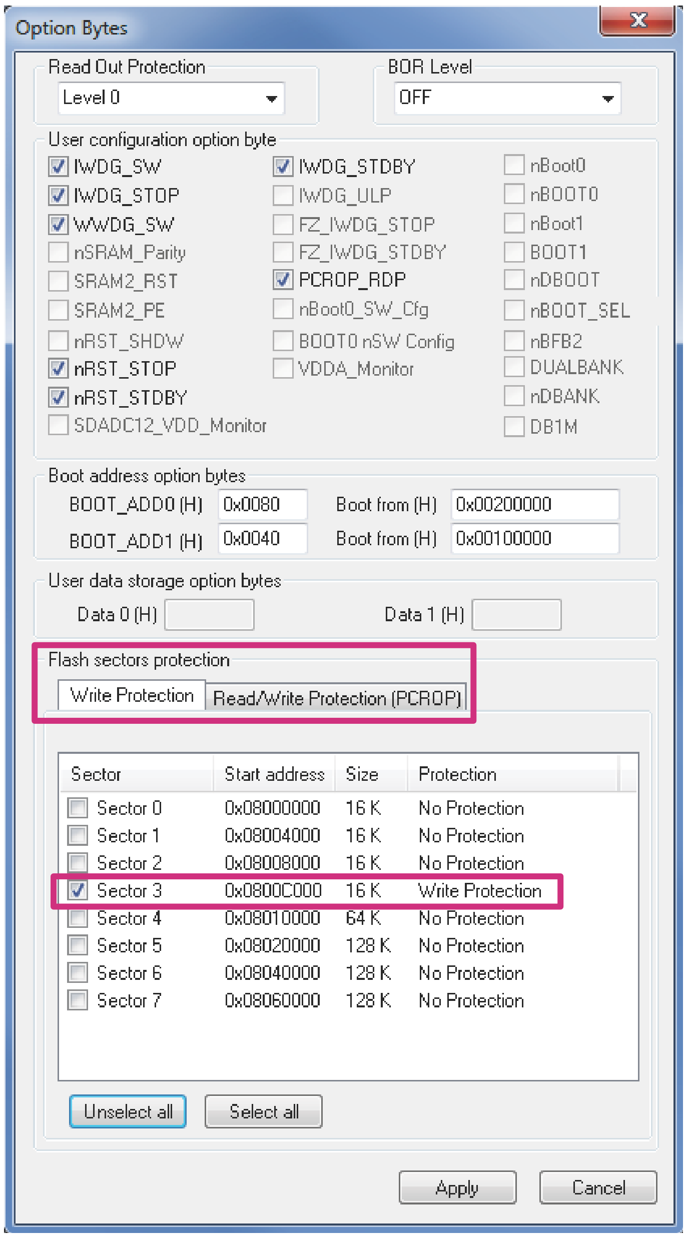
The user must also take into account that the IP-code's constants can be deleted or modified by the ST Customer level n + 1 and the functions can become useless, so it is recommended to protect these constants from unwanted write/erase.

The write protection area is set on the sector i by clearing the bit nWRPi on the Flash option control register (FLASH_OPTCR)

The IP-code of the example uses 116 bytes as constant coefficients. The minimum page size is 16 Kbytes so that a single page is protected (between addresses 0x0800C000 and 0x08010000).

The write protection can be set either by a dedicated Firmware code by setting the right values in dedicated registers (see the PCROP_Enable() function) or it can be set by using STLink-Utility as shown in *Figure 19*.

Figure 19. Enabling write protection with STM32 STlink Utility



Option Bytes

Read Out Protection: Level 0

BOR Level: OFF

User configuration option byte

☑ IWDG_SW	☑ IWDG_STDBY	☐ nBoot0
☑ IWDG_STOP	☐ IWDG_ULP	☐ nBOOT0
☑ WWDG_SW	☐ FZ_IWDG_STOP	☐ nBoot1
☐ nSRAM_Parity	☐ FZ_IWDG_STDBY	☐ BOOT1
☐ SRAM2_RST	☑ PCROP_RDP	☐ nDBOOT
☐ SRAM2_PE	☐ nBoot0_SW_Cfg	☐ nBOOT_SEL
☐ nRST_SHDW	☐ BOOT0 nSW Config	☐ nBFB2
☑ nRST_STOP	☐ VDDA_Monitor	☐ DUALBANK
☑ nRST_STDBY		☐ nDBANK
☐ SDADC12_VDD_Monitor		☐ DB1M

Boot address option bytes

BOOT_ADD0 (H)	0x0080	Boot from (H)	0x00200000
BOOT_ADD1 (H)	0x0040	Boot from (H)	0x00100000

User data storage option bytes

Data 0 (H) Data 1 (H)

Flash sectors protection

Write Protection | Read/Write Protection (PCROP)

Sector	Start address	Size	Protection
☐ Sector 0	0x08000000	16 K	No Protection
☐ Sector 1	0x08004000	16 K	No Protection
☐ Sector 2	0x08008000	16 K	No Protection
☑ Sector 3	0x0800C000	16 K	Write Protection
☐ Sector 4	0x08010000	64 K	No Protection
☐ Sector 5	0x08020000	128 K	No Protection
☐ Sector 6	0x08040000	128 K	No Protection
☐ Sector 7	0x08060000	128 K	No Protection

2.3.5 Protecting the IP-code

Activating PCROP using PCROP_Enable() function

Protecting the IP-code is done by activating the PCROP on Sector 2 using the PCROP_Enable() function defined in the STEP1-ST_Customer_level_n project main.c file.

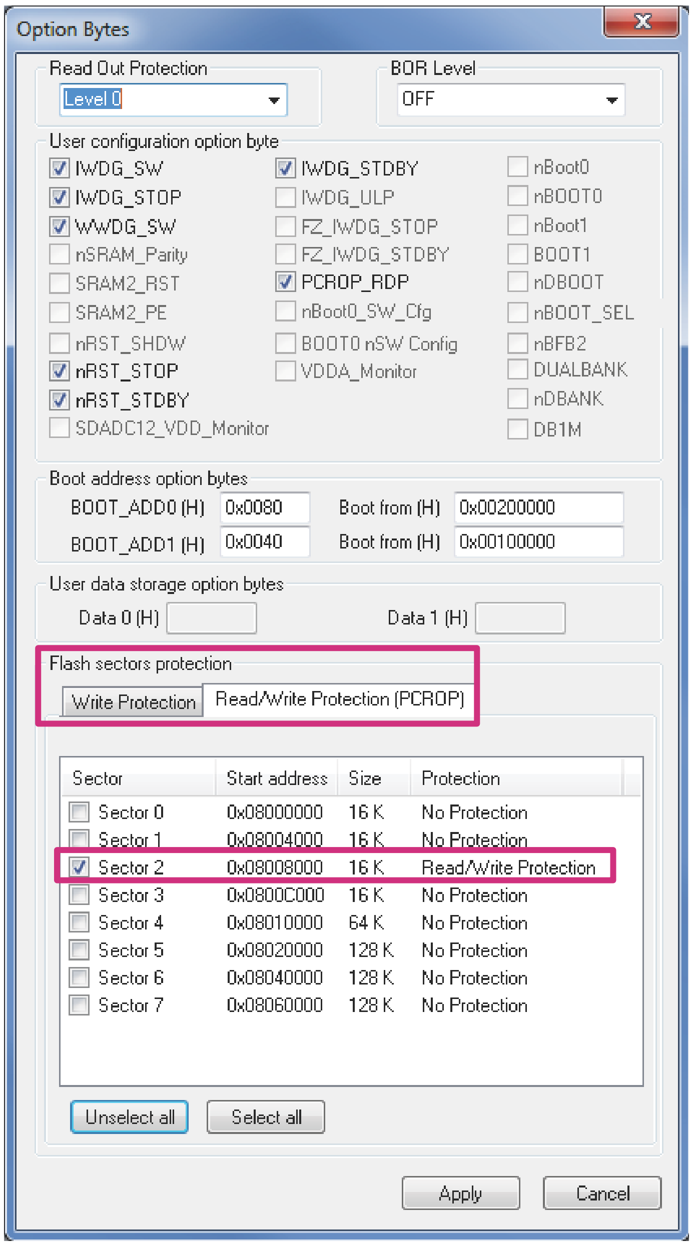
Once this function is executed at least one time, the Sector 2 becomes Read/Write protected and any IP-code modification requires disabling the PCROP then the full Flash memory content is erased.

Activating PCROP using STM32 STLink Utility

To activate PCROP on the Sector 2 using STM32 STLink Utility the user must follow the sequence shown below:

- Power on the board, then in the STLink Utility interface go to Target → Option bytes.
- On the following window set the Flash memory protection mode to Read/Write protection and enable protection on Sector 2, as shown in *Figure 20*
- Click on Apply button.

Figure 20. Enabling PCROP with STM32 STlink Utility



Read Out Protection: Level 0

BOR Level: OFF

User configuration option byte

✓ IWDG_SW	✓ IWDG_STDBY	☐ nBoot0
✓ IWDG_STOP	☐ IWDG_ULP	☐ nBOOT0
✓ WWDG_SW	☐ FZ_IWDG_STOP	☐ nBoot1
☐ nSRAM_Parity	☐ FZ_IWDG_STDBY	☐ BOOT1
☐ SRAM2_RST	✓ PCROP_RDP	☐ nDBOOT
☐ SRAM2_PE	☐ nBoot0_SW_Cfg	☐ nBOOT_SEL
☐ nRST_SHDW	☐ BOOT0 nSW Config	☐ nBFB2
✓ nRST_STOP	☐ VDDA_Monitor	☐ DUALBANK
✓ nRST_STDBY		☐ nDBANK
☐ SDADC12_VDD_Monitor		☐ DB1M

Boot address option bytes

BOOT_ADD0 (H) 0x0080 Boot from (H) 0x00200000

BOOT_ADD1 (H) 0x0040 Boot from (H) 0x00100000

User data storage option bytes

Data 0 (H) Data 1 (H)

Flash sectors protection

Write Protection | Read/Write Protection (PCROP)

Sector	Start address	Size	Protection
☐ Sector 0	0x08000000	16 K	No Protection
☐ Sector 1	0x08004000	16 K	No Protection
✓ Sector 2	0x08008000	16 K	Read/Write Protection
☐ Sector 3	0x0800C000	16 K	No Protection
☐ Sector 4	0x08010000	64 K	No Protection
☐ Sector 5	0x08020000	128 K	No Protection
☐ Sector 6	0x08040000	128 K	No Protection
☐ Sector 7	0x08060000	128 K	No Protection

2.3.6 Executing PCROP-ed IP-code

Once the IP-code has been programmed on the Sector 2 and protected, the IP-code must be tested by calling its functions in the user code.

This section describes how to execute the PCROP-ed IP-code in the STEP1-ST_Customer_level_n project with both configurations, noting that the PCROP-IP-code configuration is only for testing purposes and must not be used for STEP2.
The PCROP-IP-code-XO is the right configuration where the compiler is set to generate an execute-only IP-code. This is the configuration to use to run the STEP2-ST_Customer_level_n+1 project.

PCROP-IP-code-XO (must be used before STEP2)

The compiler is configured to generate an execute-only IP-code avoiding any data read from it (avoiding literal pools).

1. Open project located in the STEP1-ST_Customer_level_n directory and choose your preferred toolchain
2. Select the PCROP-IP-code-XO configuration
3. Rebuild all files.
4. Run the example following the sequence below:
   a) Power on the board and before loading the code, check if there is any PCROP-ed or write protected sector. If yes disable the protection using STM32 STLink Utility then load the code. Once the program has been loaded, the blue LED should toggle continuously
   b) Press the user button key to activate the PCROP protection, once done the green LED is ON and the Sector 2 is a PCROP-ed sector, else the blue LED is ON and PCROP activation failed
   c) Press the user button key to execute the PCROP-ed IP-code called in the main.c file, the green LED should toggle continuously.

PCROP-IP-code (for test only and must not be used for STEP2)

No special compiler option used, just for testing purposes to show that avoiding data in code (as literal pools and branch tables) is mandatory for the PCROP-ed codes.

1. In the same project located in the STEP1-ST_Customer_level_n directory select the PCROP-IP-code configuration
2. Rebuild all files.
3. Run the example following the sequence below:
   a) Power on the board and before loading the code, check if there is any PCROP-ed or write protected sector. If yes disable the protection using STM32 STLink Utility, then load the code; Once the program has been loaded, the blue LED should toggle continuously
   b) Press the user button key to activate PCROP protection, once done the green LED is ON and the Sector 2 is a PCROP-ed sector, else the blue LED is ON and PCROP activation failed
   c) Press the user button key to execute the PCROP-ed IP-code called in main.c file, an Error Operation Interrupt is generated, the system reset is initiated and the blue LED toggles continuously.

Interpretation

The lowpass filter function computes the testInput_f32_1kHz_15kHz input signal and must output a 1 KHz sine wave. The output data (testOutput) is then compared to the reference (refOutput) already calculated with MATLAB, if it matches, the green LED toggles continuously, else the blue LED toggles continuously.

For the PCROP-IP-code configuration where the PCROP-ed IP-code contains literal pools: when executing the IP-code (FIR_lowpass_filter() function), the literal pools can not be accessed through the D-code bus then the RDERR flag is set. The OPERR flag is set as well and a read operation error interrupt is generated, then a system reset is initiated in HAL_FLASH_OperationErrorCallback() function and the blue LED toggles continuously.

However for the PCROP-IP-code-XO configuration, the IP-code is executed correctly and the green LED must toggle continuously.

Note: For more details, refer to the readme.txt inside the embedded software package.

2.3.7 Creating header file and generating symbol definition file

The header file to be provided to the ST Customer level n+1 contains the definitions of IP-code functions to be used. In this example it is the fir_filter.h file included in the main.c file.

Generating symbol definition file with IAR

In the IDE, to export the PCROP-ed IP-code symbols, choose Project→Options→Build Actions and specify the following command line in the Post-build command line text field, as shown in *Figure 21*:

$TOOLKIT_DIR$\bin\isymexport.exe --edit "$PROJ_DIR$\steering_file.txt" "$TARGET_PATH$" "$PROJ_DIR$\fir_filter.o"

Where steering_file.txt is created, to be used as option to keep only the IP-code function symbols in the symbol file to be generated.

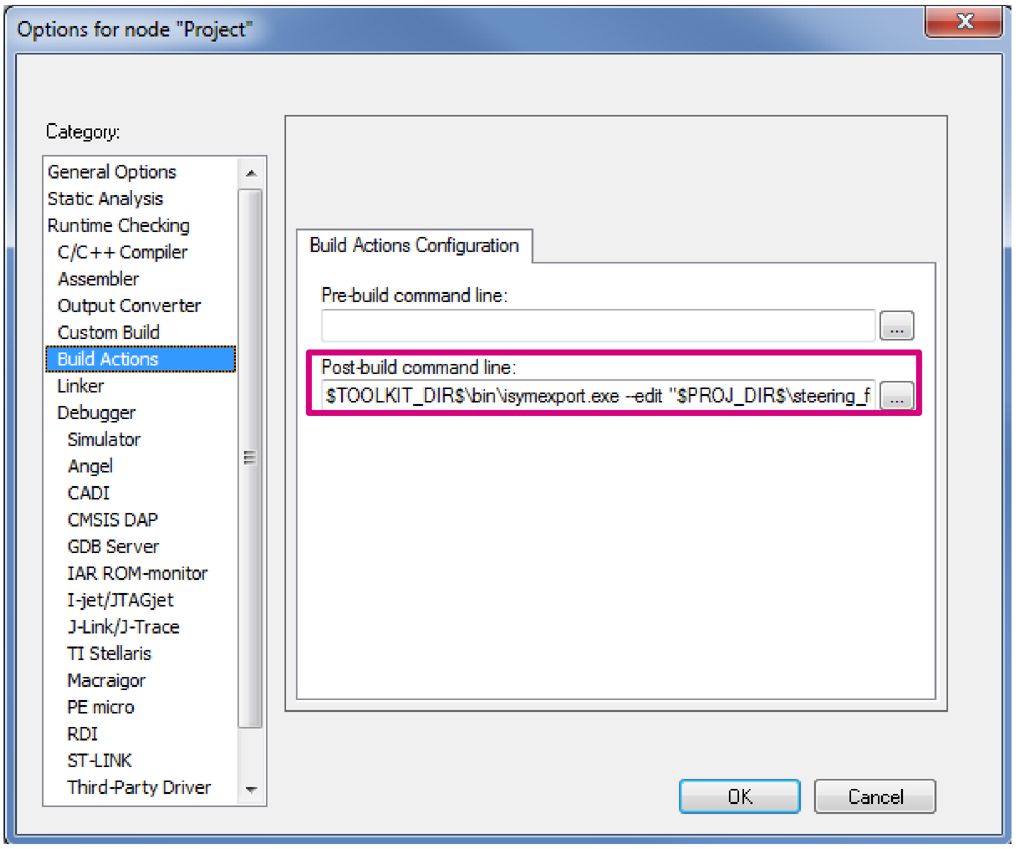
To keep only the IP-code functions in the symbol file, the steering_file.txt must be created as shown below:

```
show arm_fir_f32
show arm_fir_init_f32
show FIR_lowpass_filter
```

Where "show" command is used to keep the preferred functions in the symbol definition file to be generated.

The generated symbol definition file (fir_filter.o) is used by adding it to the STEP2-ST_Customer_level_n+1 IAR project.

Figure 21. Generating symbol definition file with IAR



Pre-build command line:

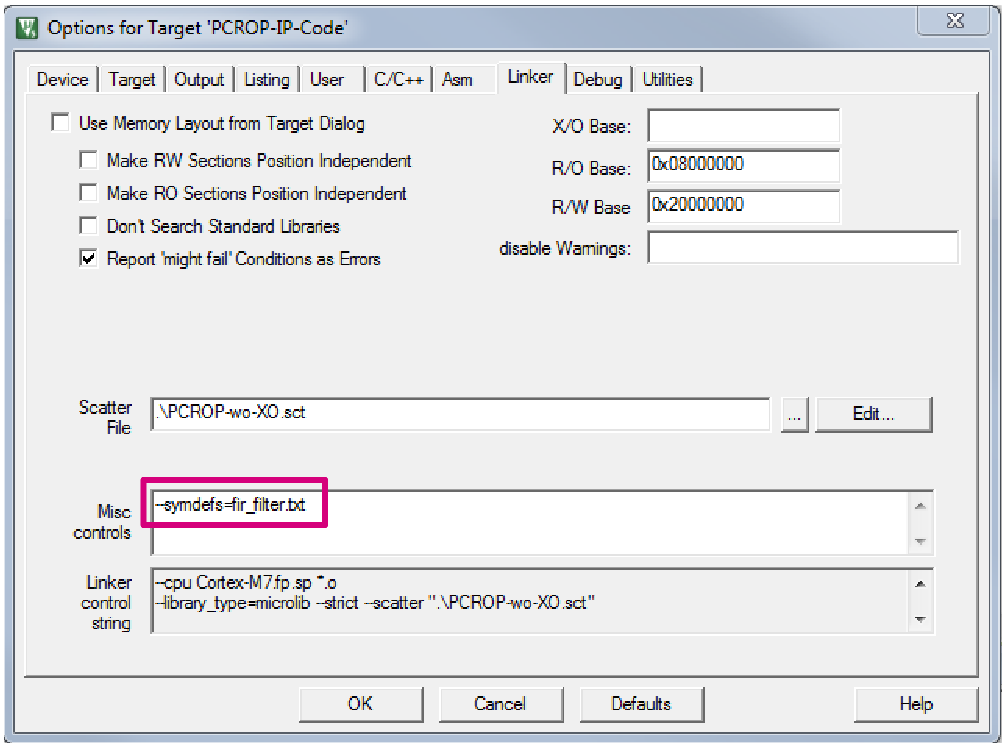
Post-build command line:

```
$TOOLKIT_DIR$\bin\isymexport.exe --edit "$PROJ_DIR$\steering_f
```

Generating symbol definition file with Keil®

To generate the symbol definition file with Keil® go to “Project options” in the Linker tab add the command “--symdefs=fir_filter.txt” and rebuild the project (see *Figure 22*).

Figure 22. Generating symbol definition file with Keil



☐ Use Memory Layout from Target Dialog

☐ Make RW Sections Position Independent

☐ Make RO Sections Position Independent

☐ Don't Search Standard Libraries

☑ Report 'might fail' Conditions as Errors

X/O Base:

R/O Base: 0x08000000

R/W Base 0x20000000

disable Warnings:

Scatter File: .\PCROP-wo-XO.sct

Misc controls: --symdefs=fir_filter.txt

Linker control string:

```
--cpu Cortex-M7.fp.sp *.o
--library_type=microlib --strict --scatter ".\PCROP-wo-XO.sct"
```

The symbol definition file named fir_filter.txt is then created in the STEP1-ST_Customer_level_n\MDK-ARM\PCROP-IP-code-XO directory.

The file contains all the symbols of the project, so only those of the IP-code functions to be called by the end user must be kept. All other ones must be removed, the resulting symbol definition file is:

Symbol definition file fir_filter.txt:

```
#<SYMDEFS># ARM Linker, 5060300: Last Updated: Wed Dec 07 11:38:09 2016
0x08008001 T FIR_lowpass_filter
0x08008049 T arm_fir_f32
0x0800836d T arm_fir_init_f32
```

Generating symbol definition file with SW4STM32

In the IDE, to export the PCROP-ed IP-code symbols, choose Project→Propities→C/C++ Build→Settings→Build Steps and specify the following command line in the Post-build steps command line text field, as shown in *Figure 23*:

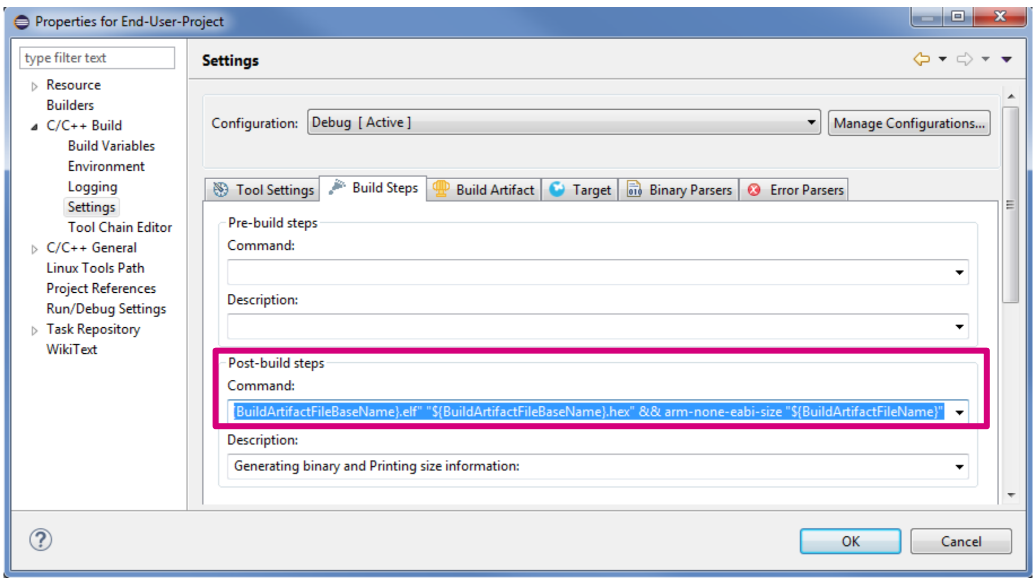
```
arm-none-eabi-objcopy -O ihex "${BuildArtifactFileBaseName}.elf"
"${BuildArtifactFileBaseName}.hex" && arm-none-eabi-size
"${BuildArtifactFileName}" && arm-none-eabi-objcopy  -K arm_fir_f32 -K
FIR_lowpass_filter -K arm_fir_init_f32 --only-section=.PCROPedCode --only-
section=.WPedData --only-section=.FirBss
"${BuildArtifactFileBaseName}.elf" fir_filter.o
```

This command keeps only the followings IP-code functions in the symbol file

```
arm_fir_f32
arm_fir_init_f32
FIR_lowpass_filter
```

The generated symbol definition file (fir_filter.o) is used by adding it to the STEP2-ST_Customer_level_n+1 SW4STM32 project.

Figure 23. Generating symbol definition file with SW4STM32



Configuration: Debug [Active]

Pre-build steps
Command:
Description:

Post-build steps
Command: BuildArtifactFileBaseName}.elf" "${BuildArtifactFileBaseName}.hex" && arm-none-eabi-size "${BuildArtifactFileName}"
Description: Generating binary and Printing size information:

2.4 STEP2: ST Customer level n+1

The ST Customer level n+1 has the preloaded STM32F722ZE MCU with the PCROP-ed IP-code, the provided symbol definition file and the header file from the ST Customer level n.

Then referring to the Flash and the RAM memory maps provided by the ST Customer level n, the ST Customer level n+1 must:

- Create the end project,
- Include the header file and add the symbol definition file provided by the ST Customer level n to its project,
- Call the PCROP-ed IP-code functions,
- Execute and debug the end-user application.

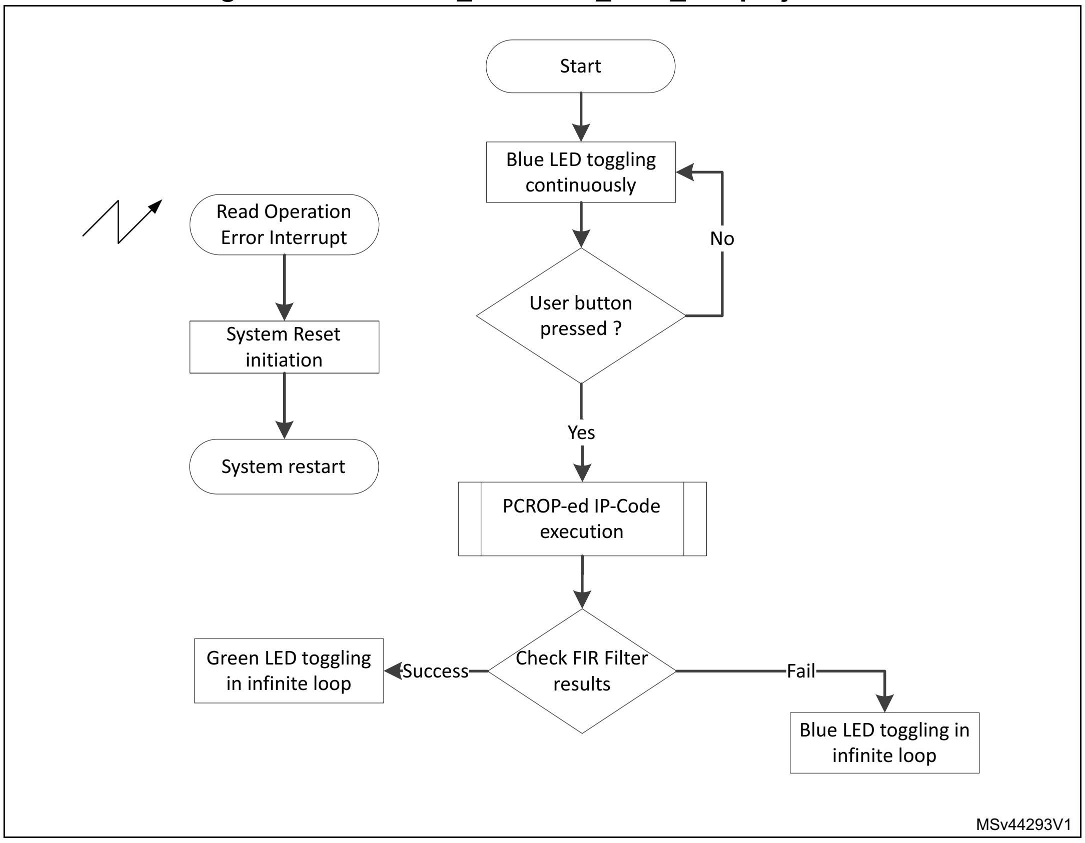
Caution: The ST Customer level n+1 must use exactly the same toolchain and compiler version used by the ST Customer level n to develop and program the IP-code, else the ST Customer level n+1 can never use the IP-code.

As in the provided example, the user must use the same toolchain and compiler version for both projects STEP1-ST_Customer_level_n and STEP2-ST_Customer_level_n+1. For example if the MDK-ARM toolchain V5.21a is used to run the STEP1-ST_Customer_level_n project, it must be used to run the STEP2-ST_Customer_level_n+1 project as well.

2.4.1 Project flow

Figure 24 illustrates the STEP2-ST_Customer_level_n+1 project flow. If any read/write operation request from/to the PCROP-ed IP-code occurs, a Read Operation Error Interrupt is generated, a system reset is initiated then the system restarts and the blue LED toggles continuously

Figure 24. STEP2-ST_Customer_level_n+1 project flow


Start
Blue LED toggling continuously
No
User button pressed ?
Yes
PCROP-ed IP-Code execution
Check FIR Filter results
Success
Green LED toggling in infinite loop
Fail
Blue LED toggling in infinite loop
Read Operation Error Interrupt
System Reset initiation
System restart
MSv44293V1

2.4.2 Creating an end user project

The protected code and write-protected data segments and the RAM region reserved for data coming from the PCROP section are located at a known position of the memories maps (see *Figure 16* and *Figure 17*), hence the user must take care when implementing the linker file to avoid placing any code in these regions.

The PCROP-ed IP-code functions are then used to create the end user application. The project located in the STEP2-ST_Customer_level_n+1 directory is an example where the PCROP-ed FIR filter functions are called in the main.c file.

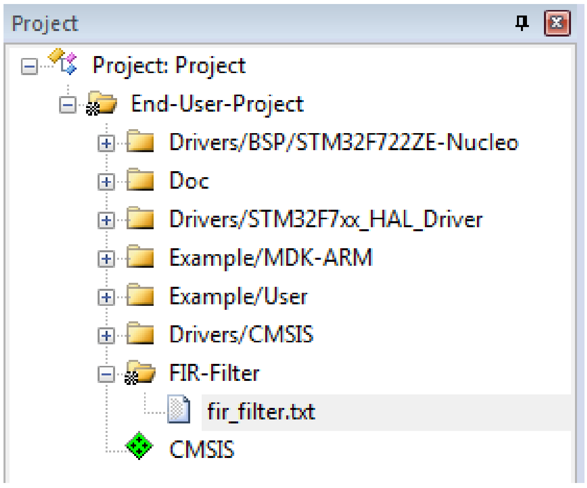
2.4.3 Including header file and adding symbol definition file

The symbol definition file provided from the ST Customer level n must be added as a classic source file which is necessary to link the PCROP-ed IP-code in the STEP2-ST_Customer_level_n+1 project.

Adding symbol definition file in Keil® project

The symbol definition file provided from the ST Customer level n described in *Section 2.3.7: Creating header file and generating symbol definition file* and named fir_filter.txt in this example must be added to the FIR-Filter group as described in *Figure 25*.

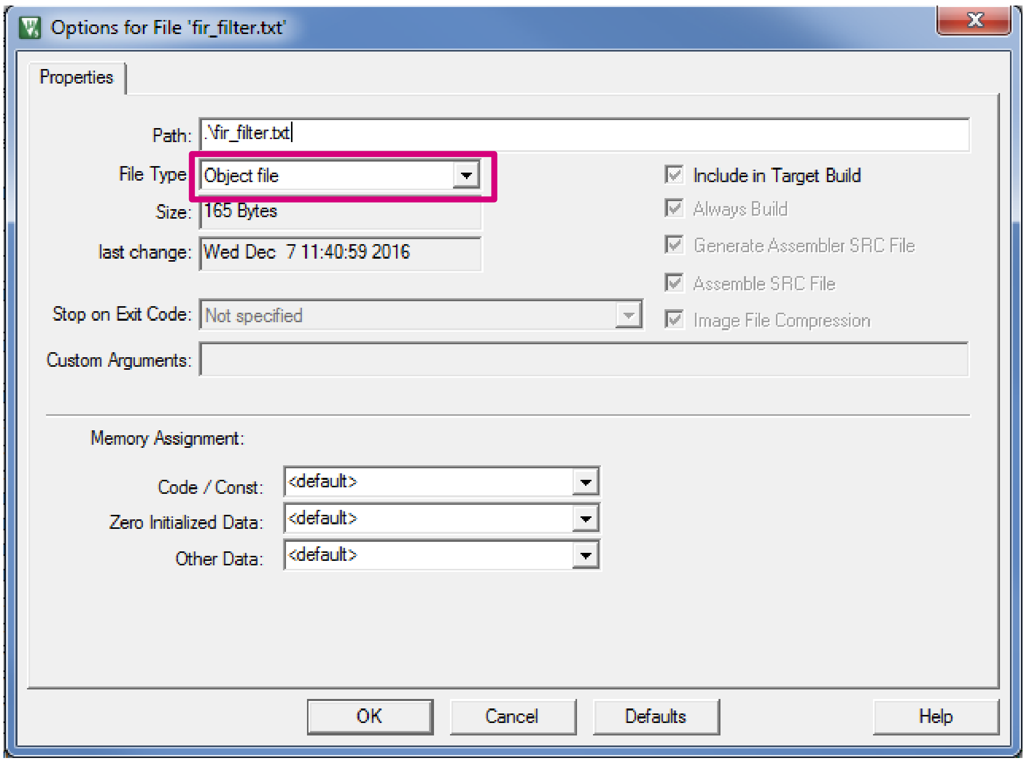
Figure 25. Adding symbol definition file to Keil project



Project
- Project: Project
  - End-User-Project
    - Drivers/BSP/STM32F722ZE-Nucleo
    - Doc
    - Drivers/STM32F7xx_HAL_Driver
    - Example/MDK-ARM
    - Example/User
    - Drivers/CMSIS
    - FIR-Filter
      - fir_filter.txt
    - CMSIS

The added fir_filter.txt file type must be changed to “Object file” instead of text document file, as indicated in *Figure 26*.

Figure 26. Setting symbol definition fil type to object file



Options for File 'fir_filter.txt'

Properties

Path: .\fir_filter.txt
File Type: Object file
Size: 165 Bytes
last change: Wed Dec 7 11:40:59 2016
Stop on Exit Code: Not specified
Custom Arguments:

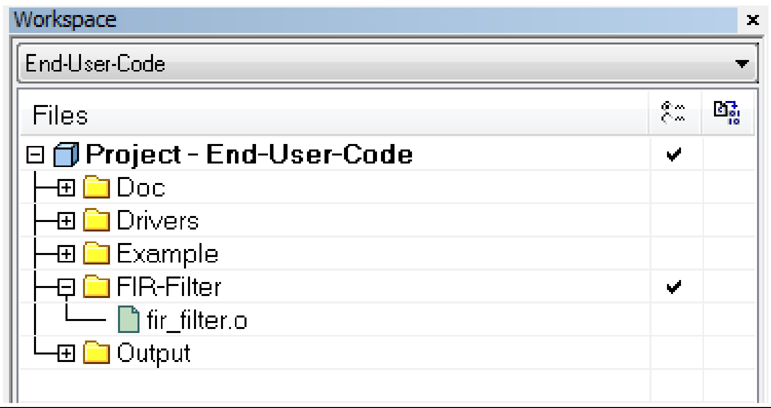
☑ Include in Target Build
☑ Always Build
☑ Generate Assembler SRC File
☑ Assemble SRC File
☑ Image File Compression

Memory Assignment:
Code / Const: <default>
Zero Initialized Data: <default>
Other Data: <default>

Adding symbol definition file in IAR project

The fir_filter.o file must be added as an object file to the group FIR_Filter, as shown in *Figure 27*.

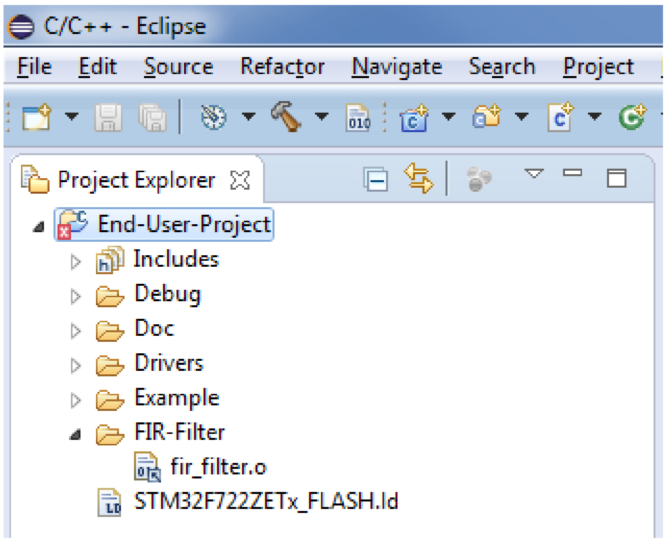
Figure 27. Adding symbol definition file to IAR project



Adding symbol definition file in SW4STM32 project

The fir_filter.o file must be added as an object file to the group FIR_Filter, as shown in *Figure 28*.

Figure 28. Adding symbol definition file to SW4STM32 project



Including header file

The header file (fir_filter.h) is included in the main.c file then the user can call FIR Filter functions.

Reserving RAM memory region for the data coming from PCROP-ed section

While the data coming from the PCROP section have the same addresses on the RAM memory on the projects level n and n+1, so that the region created on the RAM memory for these data on level n, must be also reserved for them on level n+1.

Keil® scatter file

The scatter file must be updated with the FirBSSRAM memory region:

```
; *************************************************************
; *** Scatter-Loading Description File generated by uVision ***
; *************************************************************
LR_IROM1 0x08000000 0x00004000  {    ; load region size_region
  ER_IROM1 0x08000000 0x00004000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
  }
    FirBSSRAM  0x20000000 EMPTY 0x100  {  ; Reserved for RW and Zero
initialized data of fir filter
  }
  RW_IRAM1 0x20000100 0x0003FF00  {  ; RW data of main program
   .ANY (+RW +ZI)
  }
}
```

IAR ICF file

The icf file must be updated with the FirBSSRAM memory region:

```
/* define FirBSSRAM area start and end address */
define symbol __ICFEDIT_region_FirBSSRAM_start__  = 0x20000000;
define symbol __ICFEDIT_region_FirBSSRAM_end__   = 0x200000FF;
define region FirBSSRAM_region    = mem:[from
__ICFEDIT_region_FirBSSRAM_start__   to __ICFEDIT_region_FirBSSRAM_end__];

/* Place RW and ZI data of PCROP-ed section of FirBSSRAM memory region*/
place in FirBSSRAM_region   {
rw object FIR_Filter.o
zi  object FIR_Filter.o};
```

SW4STM32 linker file

With STM4STM32 the linker file must be updated with the FirBSSRAM memory region and it is better to specify also the PCROP-ed and write protection sector:

```
/* specify the memory areas */
MEMORY
{
RAM (xrw)       : ORIGIN = 0x20000100, LENGTH = 256K - 0x100
FirBSSRAM (xrw): ORIGIN = 0x20000000, LENGTH = 0x100
FLASH (rx)      : ORIGIN = 0x08000000, LENGTH = 16K
PCROP (x)       : ORIGIN = 0x08008000, LENGTH = 16K
CONST (rx)      : ORIGIN = 0x0800C000, LENGTH = 16K
}
.PCROPedCode (NOLOAD): { *(.PCROPedCode)    } >PCROP
```

```
.WPedData (NOLOAD):     { KEEP(*(.WPedData)) } >CONST
.FirBss :               { KEEP(*(.FirBss))   } >FirBSSRAM
```

2.4.4 Calling PCROP-ed IP-code functions

Once the fir_filter.h header file is included in the main.c file and the symbol definition file added to the project, the PCROP-ed IP-code functions can be called.

The FIR_lowpass_filter() function (described in fir_filter.c file) is called in the main file as shown below:

FIR_lowpass_filter(inputF32, outputF32, TEST_LENGTH_SAMPLES)

Where:

- inputF32: is the pointer to the table containing input signal data;
- outputF32: is the pointer to the table to be filled with processed output signal data; TEST_LENGTH_SAMPLES: is the number of samples in the input to be processed.

2.4.5 Running the end user application

At this stage the IP-code must be already preloaded and PCROP-ed in the STM32F722ZE MCU, the STEP1-ST_Customer_level_n project (PCROP-IP-code-XO configuration) must be loaded and executed before running the project.

To make the program work, follow the steps described below:

1. Open project located in the STEP2-ST_Customer_level_n+1 directory and choose the same toolchain used in STEP1.
2. Rebuild all the files.
3. Run the example following the sequence below
   a) Power on the board and load the code (here only the user code is loaded)
   b) Press the user button key to execute the PCROP-ed IP-code called in the main.c file, the green LED must toggle continuously.

Note: For more details, refer to the readme.txt inside the embedded software package.

2.4.6 PCROP protection in debug mode

When developing its end user application, the ST Customer level n+1 needs to debug its code. Therefore, when debugging the end user application, the PCROP protection must prevent any read access to the IP-code developed by ST Customer level n.

This section presents how the PCROP protects the preprogrammed IP-code against reading when debugging the user code.

The debugging example described below is done on Keil® project.

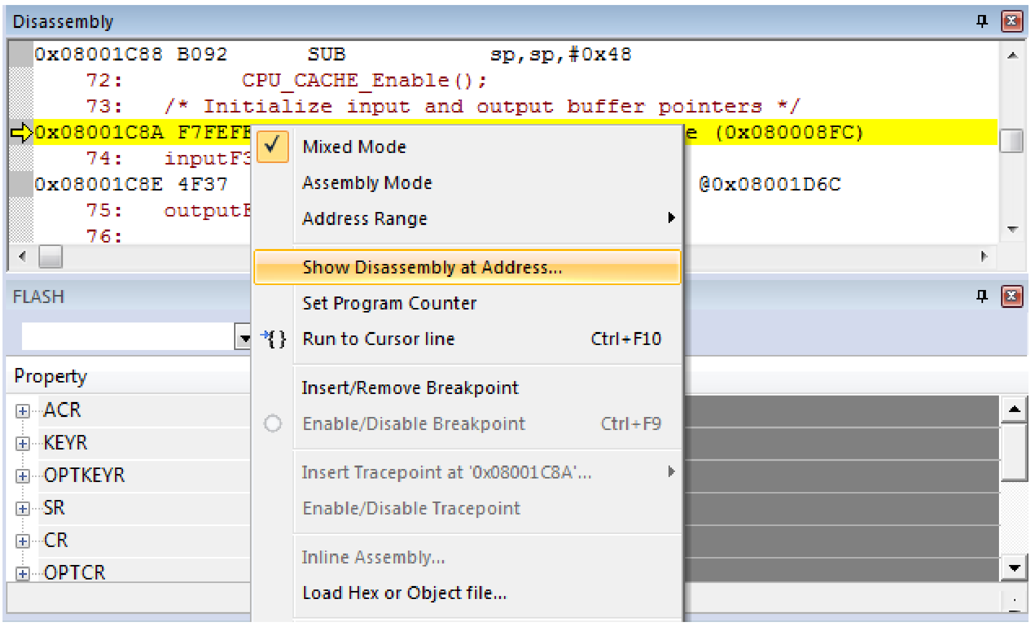
Debugging the end user application

Before debugging the end user project, the STEP1-ST_Customer_level_n project (PCROP-IP-code-XO workspace) must be loaded and executed in order to get the STM32F722ZE MCU with the preprogrammed and PCROP-ed IP-code.

To debug the STEP2-ST_Customer_level_n+1 project, follow the steps described below:

1. Open the project located in the STEP2-ST_Customer_level_n+1 directory and choose the same toolchain used in STEP1.
2. Rebuild all the files.
3. Click on Start/Stop Debug session to start debug.
4. Run the program by clicking on Run, the blue LED should toggle continuously.
5. Press User button to execute the IP-code, the green LED should toggle continuously.
6. To access to the PCROP-ed IP-code, right click on the Disassembly window then select “Show Disassembly at Address” as shown in *Figure 29*.

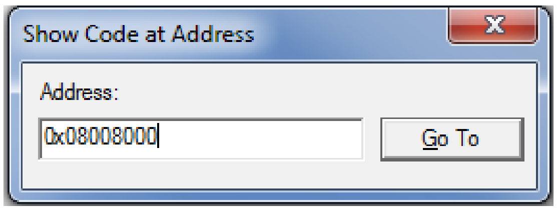
Figure 29. PCROP-ed IP-code assembly reading


Disassembly
Mixed Mode
Assembly Mode
Address Range
Show Disassembly at Address...
Set Program Counter
Run to Cursor line Ctrl+F10
Insert/Remove Breakpoint
Enable/Disable Breakpoint Ctrl+F9
Insert Tracepoint at '0x08001C8A'...
Enable/Disable Tracepoint
Inline Assembly...
Load Hex or Object file...
FLASH
Property
ACR
KEYR
OPTKEYR
SR
CR
OPTCR

```
0x08001C88 B092      SUB         sp,sp,#0x48
    72:          CPU_CACHE_Enable();
    73:    /* Initialize input and output buffer pointers */
0x08001C8A F7FEFE                          e (0x080008FC)
    74:    inputF3
0x08001C8E 4F37                             @0x08001D6C
    75:    outputB
    76:
```

7. Fill in the address field the PCROP-ed sector starting address and click on “Go To button” as shown in *Figure 30*.

Figure 30. Filling PCROP-ed sector starting address



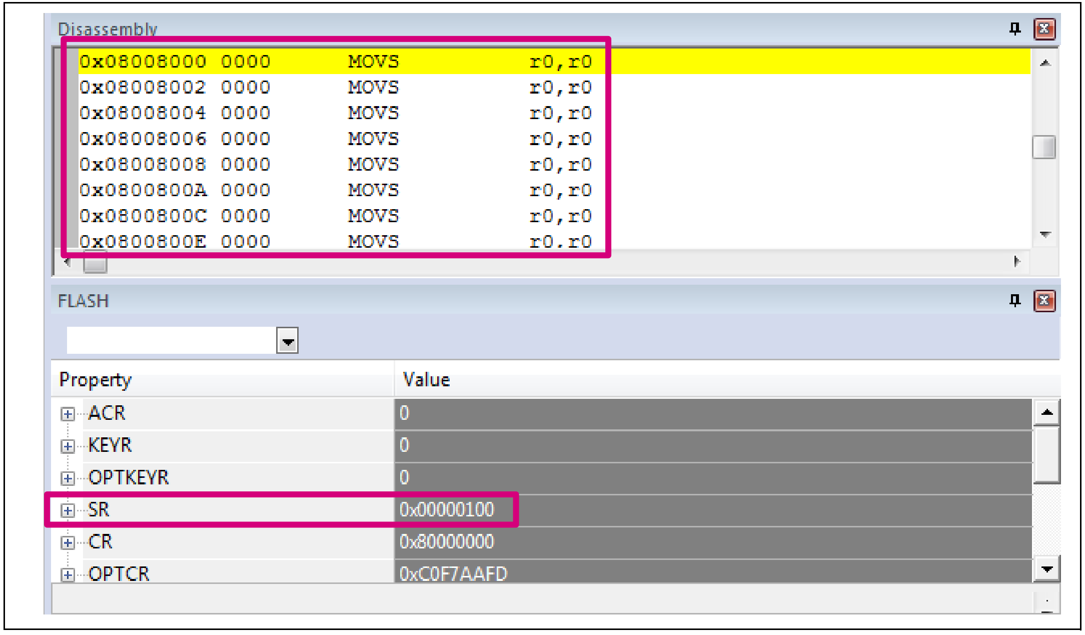
Address: 0x08008000

As shown in *Figure 31*, the PCROP-ed IP-code loaded in the Sector 2 is unreadable while the code located just before 0x08008000 address can be read. Reading the PCROP-ed sector sets the RDERR and OPERR flags in the FLASH_SR register.

A Flash operation error interrupt is generated due to the Flash memory read operation request through the D-code bus when debugging.

Then a software reset is initiated in the HAL_FLASH_OperationErrorCallback() function (described in main.c file) and the blue LED toggles continuously.

Figure 31. PCROP-ed IP-code assembly reading



```
0x08008000 0000       MOVS          r0,r0
0x08008002 0000       MOVS          r0,r0
0x08008004 0000       MOVS          r0,r0
0x08008006 0000       MOVS          r0,r0
0x08008008 0000       MOVS          r0,r0
0x0800800A 0000       MOVS          r0,r0
0x0800800C 0000       MOVS          r0,r0
0x0800800E 0000       MOVS          r0,r0
```

Property	Value
ACR	0
KEYR	0
OPTKEYR	0
SR	0x00000100
CR	0x80000000
OPTCR	0xC0F7AAFD

3 Conclusion

The STM32F72xxx and STM32F73xxx microcontrollers provide very flexible and useful read and/or write protection features that can be used in the applications where the protection is required. This application note describes how to use these read, write and PCROP protection features.

4 Revision history

Table 3. Document revision history

Date	Revision	Changes
15-Feb-2017	1	Initial release.