AN0164
Application Note

AT32F423 Security Library Application Note

Introduction

This application note is written to help users with a better understanding of the application principles, the use of software resources and example codes relating to the security library of AT32F423 series.

Applicable products:

Part number	AT32F423xx

Contents

List of tables

List of figures

1 Overview

At present, as an increasing number of microcontrollers (known as MCU) require complex algorithms and middleware solutions, how to protect core algorithms and other IP codes of solution providers has emerged as one of the most important concerns in the field of MCU applications.

In response to this demand, AT32F423 series is equipped with a security library, known as sLib, with the aim of preventing important IP codes from being altered or read by end user program, so as to safeguard the rights of solution providers.

Here this document will detail the application logics behind AT3F423 series' security library and its software usage.

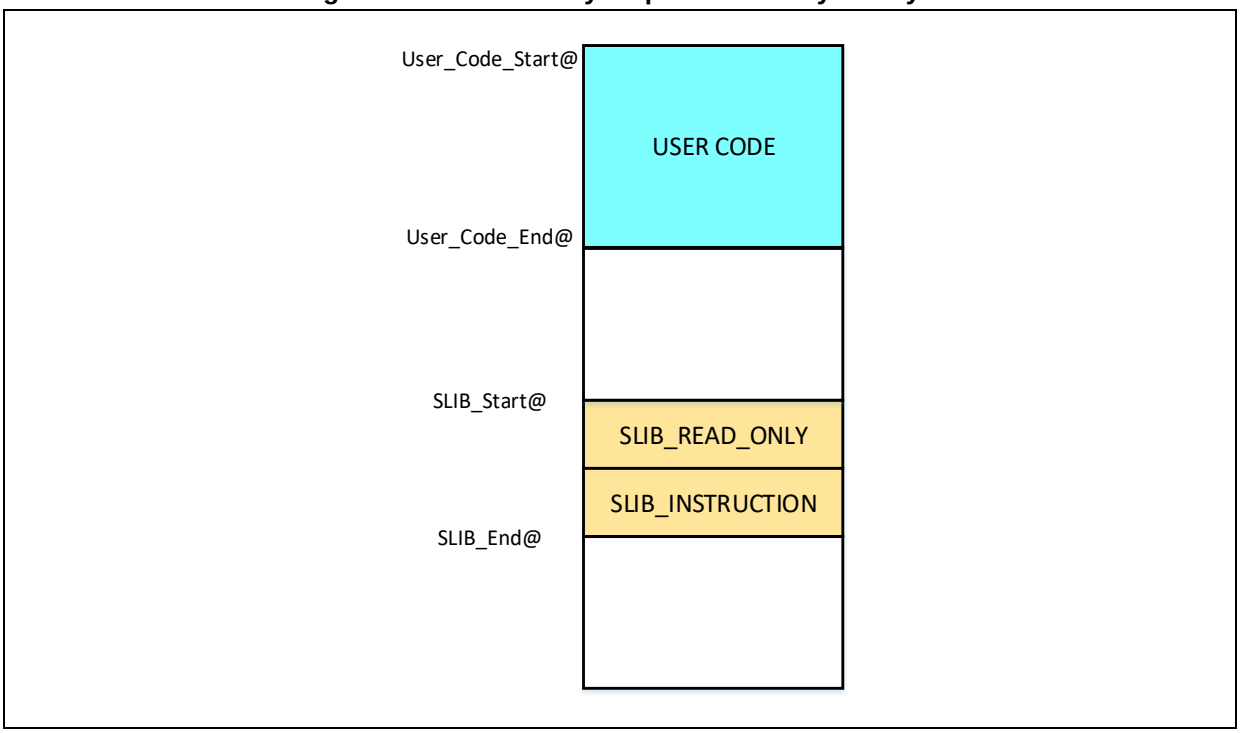
2 Principles

2.1 sLib application principles

- Any part of Flash memory can be designated as a security library (sLib) with password. This sLib is used for storing critical algorithms by solution providers while the remaining memory area can be used for secondary development by end users.
- sLib is divided into a read-only area (SLIB_READ_ONLY) and an instruction area (SLIB_INSTRUCTION). Part of or the entire sLib can be set as read-only area or instruction area
- SLIB_READ_ONLY area can be read through I-Code and D-Code, but it is write-protected
- Program codes in the SLIB_INSTRUCTION area can only be fetched (only executable) by MCU through I-CODE. They cannot be read out by reading access (including ISP/ICP/debug mode or boot from internal RAM) via D-Code, for accessing SLIB_INSTRUCTION by reading operation will return all 0xFF.
- Codes and data within sLib cannot be erased until a correct password is entered. Performing write or erase operation in case of wrong password entry will trigger a warning from EPPERR=1 of the FLASH_STS register
- Mass erase to the main Flash memory by end users will not affect codes and data in the sLib, meaning that programs and data in this secure area will not be erased
- After sLib feature is enabled, users can also unlock this protection through wring a correct password in the SLIB_PWD_CLR register. Once sLib is unlocked, MCU will erase the whole main memory, including sLib. This kind of design is to protect program codes against leakage even if the password set by solution providers is leaked.

Figure 1 below shows a block diagram of main Flash memory with security library. Programs and codes stored in the security library can be called and executed by end users, but they are read-protected.

Figure 1. Flash memory map with security library



The size of sLib area is configured based on the sector level. The sector size is defined by actual MCU series. Table 1 below lists Flash memory capacity, per-sector size and its settable range.

When the 20 KB boot memory is defined as Flash memory extension area, it can also be functioning as a sLib area.

Table 1. AT32F423 series Flash memory capacity

Part number	Internal Flash (Byte)	Per-sector (Byte)	Address range
AT32F423x8	64K	1K	Sector 0 ~ 63 (0x08000000 ~ 0x0800FFFF)
AT32F423xB	128K	1K	Sector 0 ~ 127 (0x08000000 ~ 0x0801FFFF)
AT32F423xC	256K	2K	Sector 0 ~ 127 (0x08000000 ~ 0x0803FFFF)

2.2 How to enable sLib protection

By default, sLib setting register is not readable and write-protected. Before writing to this register, users need first unlock the register by keying in the 0xA35F6D24 value to the SLIB_UNLOCK register, and then check if the unlock operation is successful by checking the SLIB_ULKF bit of the SLIB_MISC_STS register. If successful, sLib setting register can now be written.

Follow the procedures below to enable Flash memory sLib:

- Check the OBF bit in the FLASH_STS register to confirm that there is no other ongoing programming operation;
- Write 0xA35F6D24 to the SLIB_UNLOCK register to unlock security library;
- Check if unlock operation is successful by checking the SLIB_ULKF bit in the SLIB_MISC_STS register;
- Set a would-be-protected area, including sLib start address and end address as well as sLib instruction area start address, through SLIB_SET_RANGE register;
- Wait for the OBF bit to be cleared (“0”);
- Set a sLib password through the SLIB_SET_PWD register;
- Wait for the OBF bit to be cleared (“0”);
- Program codes to be stored into sLib;
- Perform system reset, and reload sLib with setting words;
- Read SLIB_STS0/STS1 register to verify sLib setting results.

Special attention to be paid to the following:

- Either Flash memory or Flash memory extension area can be set as sLib. See *Table 1* for configurable sLib ranges.
- sLib codes must be programmed on a sector level. And sLib start address must be aligned with that of Flash memory or Flash memory extension area.
- Interrupt vector table as a data type is typically placed on the first sector (sector 0) of Flash memory. As a result, sector 0 should not be set as an instruction area of sLib.

For details on sLib setting register, please refer to AT32F423 technical documents.

For the program code on enabling sLib, please refer to “slib_enable()” in the main.c of Project_L0 example case. Besides, it is also possible to set sLib through ICP or ISP programming tool, which will be described in the subsequent sections.

2.3 How to disable sLib protection

After sLib feature is enabled, it is possible for users to unlock it by writing the previously set password in the SLIB_PWD_CLR register.

Once sLib is disabled, the device will perform mass erase on the main Flash memory, including erasing contents in the sLib area.

Follow the procedures below to disable sLib:

- Check the OBF bit in the FLASH_STS register to confirm that there is no other ongoing programming operation;
- Write the previously set password into the SLIB_PWD_CLR register;
- Perform system reset, and reload sLib with setting words;
- Read SLIB_STS0 register to verify sLib setting results.

2.4 Set and run sLib

As described in the previous sections, program codes within the SLIB_INSTRUCTION area can be fetched (only executable) by MCU through I-CODE, but they cannot be read out by means of reading data via D-Code, so as to achieve robust protection. In other words, even the program codes located in the SLIB_INSTRUCTION are forbidden to read data that are placed in the SLIB_INSTRUCTION. Such data, for instance, include the likes of literal pool — compiled C program code, branch table or constants, which will be read through D-code upon instruction execution.

This indicates that only instructions, rather than data, can be placed in SLIB_INSTRUCTION area. As a result, if necessary to store program codes in SLIB_INSTRUCTION area, there is a need for users to generate execute-only code through compiler in order to prevent the generation of above-mentioned types of data.

Figure 2 and *Figure 3* give two examples of frequently-used literal pools and branch tables.

“switch()” is a common jump command in C program. In Figure 2, the “sclk_source” variable reads CRM_CFG register, and “LDR R7, [PC, #288]” is an assembly code. The program counter (known as PC) is used to obtain the address of CRM_CFG register through indirect addressing. The address of CRM_CFG register is stored at a nearby instruction area (also within SLIB_INSTRUCTION) as a constant. At this point, executing “switch()” instruction will trigger data read. And if such program code exist in SLIB_INSTRUCTION area, an error will occur upon program execution.

In Section 3, we give an example detailing how to avoid this problem through setting compiler.

Figure 2. Example of literal pool (1)

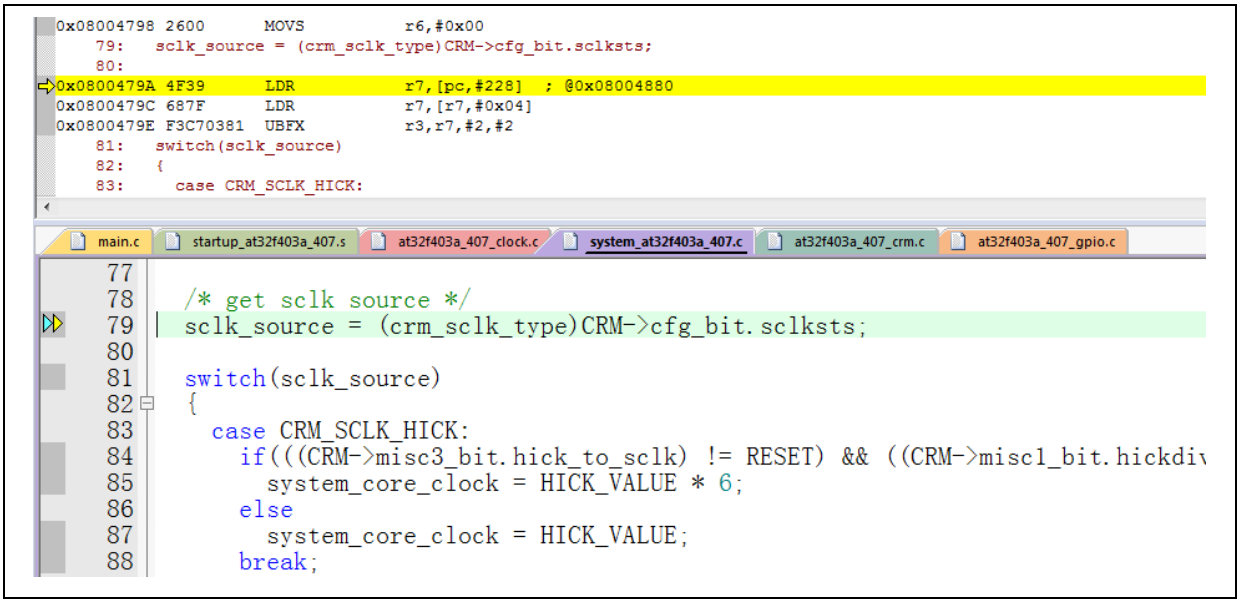
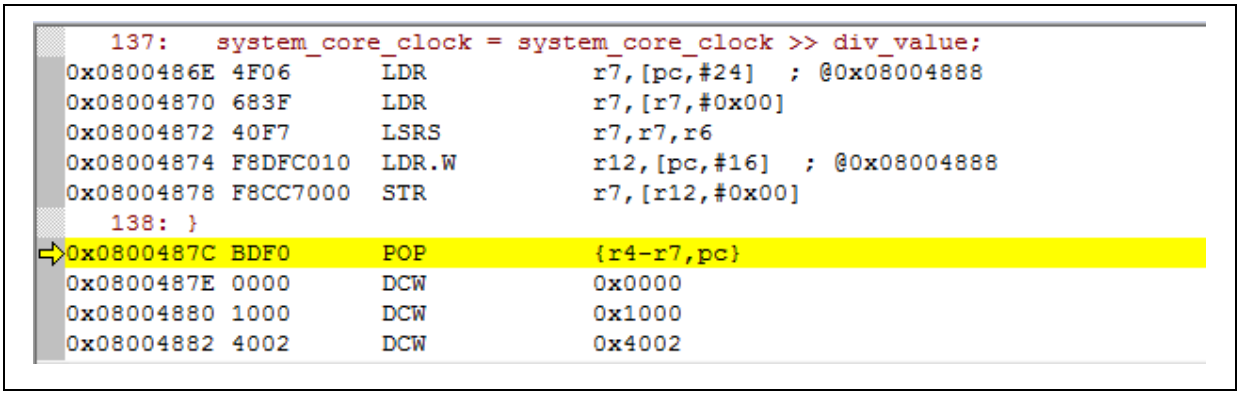


Figure 3. Example of literal pool (2)



2.4.1 Don’t set interrupt vector table as sLib instruction area

Interrupt vector table contains entry addresses of all interrupt handlers which are readable by MCU using D-Code. In most cases, the table is located at sector 0 with start address 0x08000000 in Flash memory. Therefore, the following rule should be respected when designating sLib instruction area.

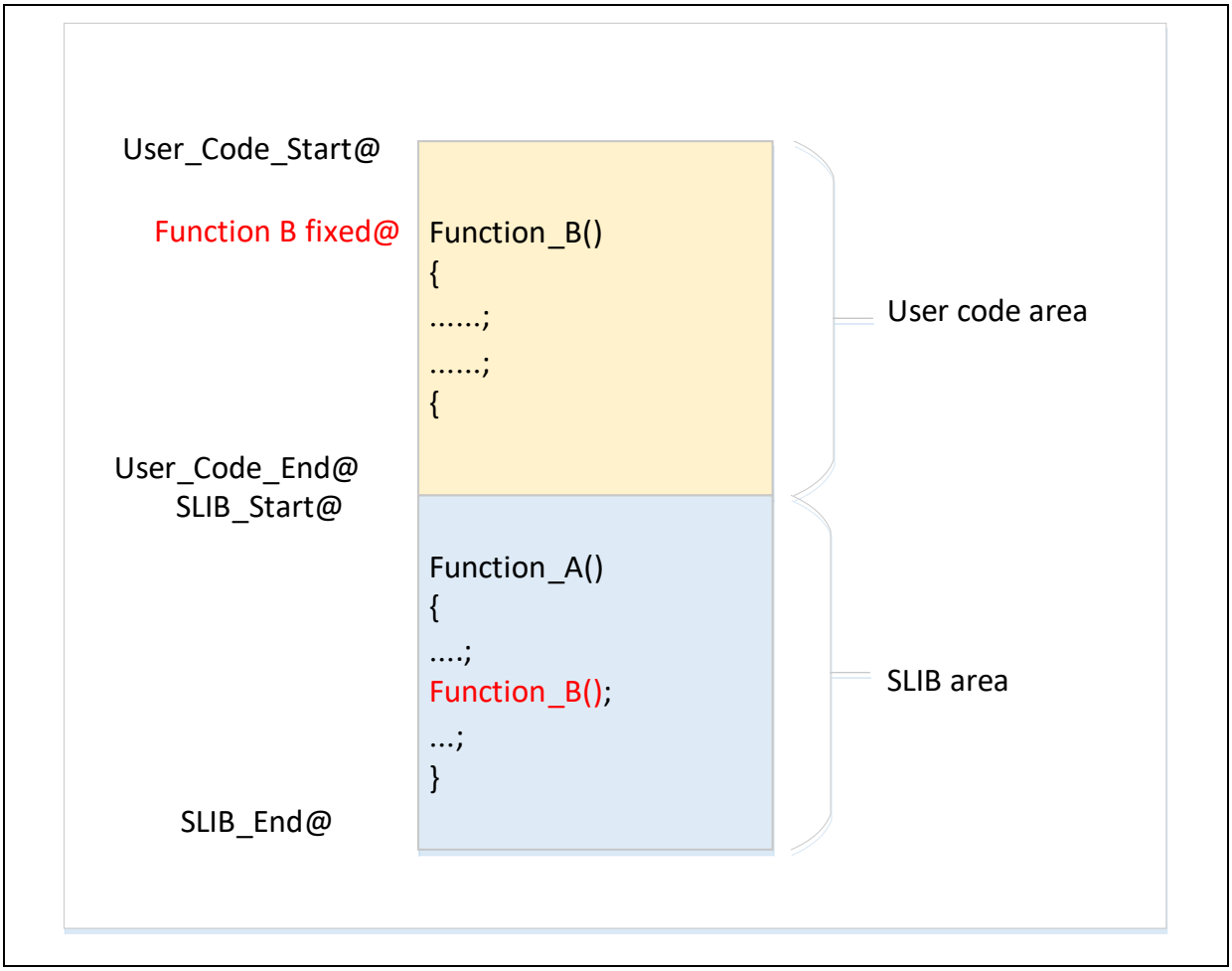
- The first sector of Flash memory should not be set as an instruction area of sLib.

2.4.2 Relevance between sLib code and user code

IP-code protected by sLib is able to call functions from a function library in the user code area. In this scenario, IP-Code will also carry the addresses of such functions, allowing PC (program counter) to jump to them while executing IP-Code. Once sLib is enabled, such functions' addresses are unchangeable. This means that these addresses in the user code area must be fixed or remain unchanged, otherwise, PC will jump to a wrong address and fail to work. Based on this, before setting sLib, it is necessary to place all functions relating to IP-Code in sLib to avoid such problem.

Figure 4 gives an example on how a protected Function_A() calls Function_B() in user code area.

Figure 4. Example of sLib function calls a function in the user code area



Besides, there is another commonly seen scenario in which C language standard function library is used, such as memset() and memcpy(). If both IP-Code and user code call such functions, aforementioned problem may occur. Despite this, here are two ways to resolve this issue.

1) Place such functions in sLib. For more information, please refer to the corresponding Keil or IAR documents.
2) Try not to use C language standard function library in the IP-Code. If there is a need to use them, their names must be changed. In the example below, write a "my_memset()" function to replace the previous "memset()".

Figure 5. Example of user-defined function

```
void* my_memset(void *s, int c, size_t n);

void arm_fir_init_f32(
  arm_fir_instance_f32 * S,
  uint16_t numTaps,
  float32_t * pCoeffs,
  float32_t * pState,
  uint32_t blockSize)
{
  /* Assign filter taps */
  S->numTaps = numTaps;

  /* Assign coefficient pointer */
  S->pCoeffs = pCoeffs;

  /* Clear state buffer and the size of state buffer is (blockSize + numTaps - 1) */
  my_memset(pState, 0, (numTaps + (blockSize - 1u)) * sizeof(float32_t));

  /* Assign state pointer */
  S->pState = pState;

}


void* my_memset(void *s, int c, size_t n)
{
  while (n>0)
      *( (char*)s + n-- -1 ) = (char)c;

  return (s);
}
```

3 Example code in sLib

This chapter offers example codes on the use of sLib alongside detailed operating procedures.

3.1 Requirements

3.1.1 Hardware requirements

- AT-START-F423 evaluation board with embedded AT32F423VCT7 microcontroller
- AT-Link debugger which is used to debug programs

3.1.2 Software requirements

- Keil® µvision IDE (this example here uses µvision V5.36.0.0) or IAR Embedded workbench IDE (this example here uses IAR V8.22.2)
- ARTERY's ICP or ISP programming tool to enable or disable sLib

3.2 Example projects

This application note offers two example projects demonstrating how software provider develops IP-Code to meet end user application requirements.

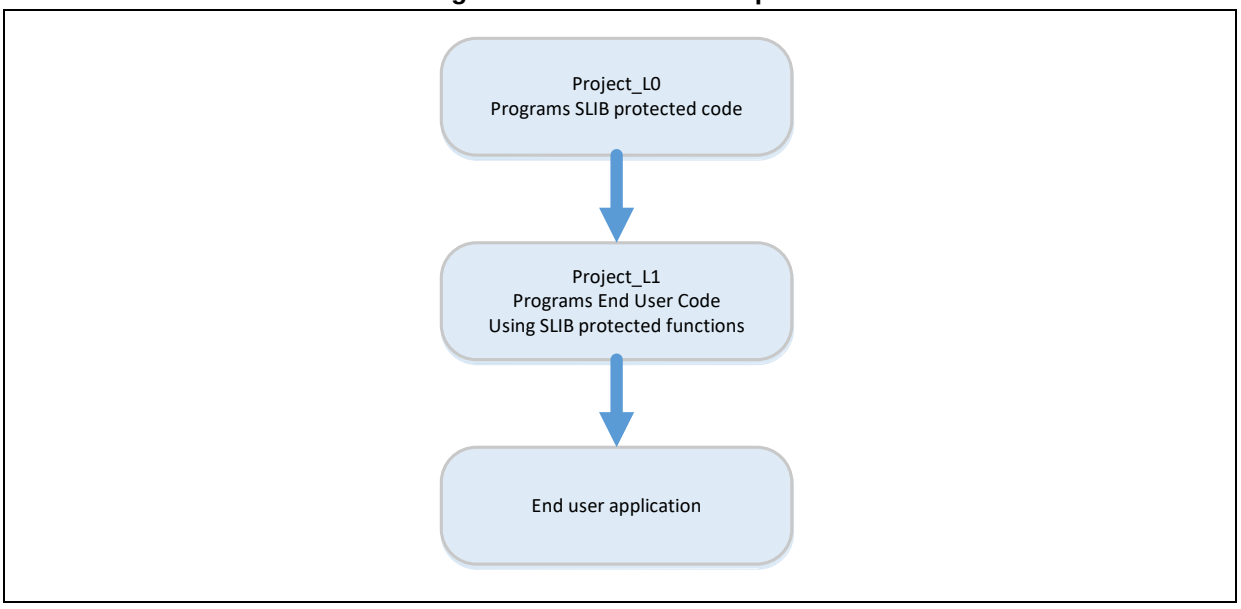
- Project_L0 shows how solution provider develops an algorithm and place it into sLib
- Project_L1 shows how end users apply this algorithm

Algorithms developed in Project_L0 will be downloaded and programmed into AT32F423 device in advance with sLib function being enabled. Meanwhile, the following information are also available to end user programs.

- Main Flash memory map, indicating the area owned by sLib and the area that can be developed by users
- Header files containing algorithm function definitions, for user programs to call
- Symbol definition file, containing the addresses of IP-Code functions, for users to call

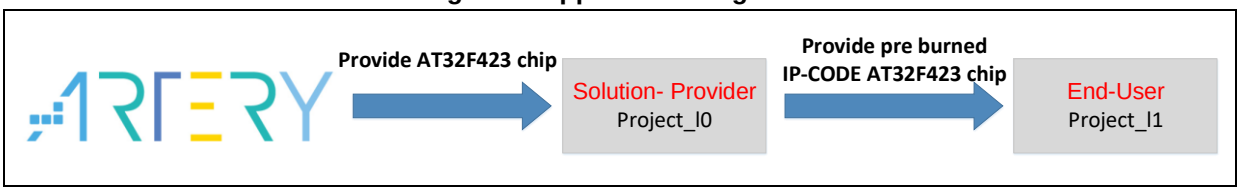
  See Figure 6 below for reference.

Figure 6. Flow chart example



Software provider can refer to *Project_L0* and *Project_L1* to develop algorithm code for end users, see Figure 7.

Figure 7. Application diagram



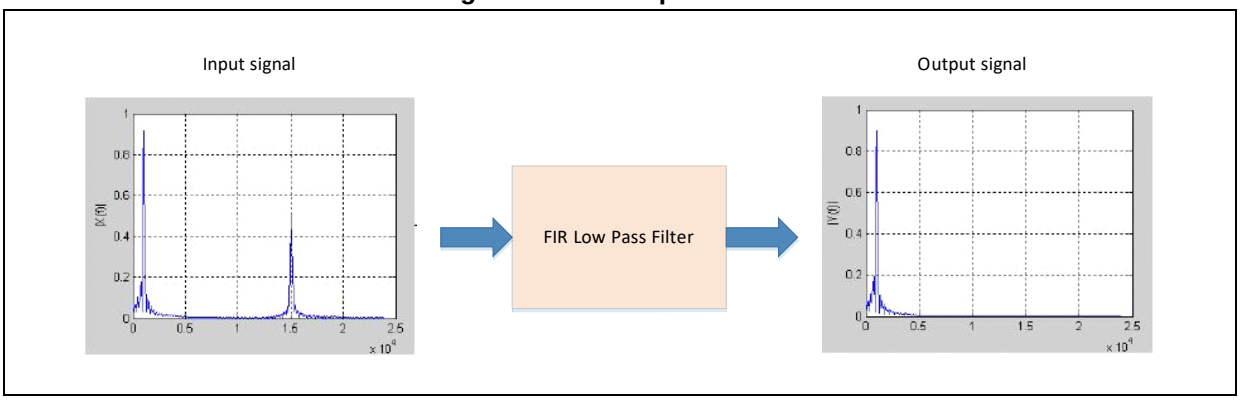
3.3 sLib protected code: FIR low-pass filter

The examples here use FIR lowpass filter algorithm from CMSIS-DSP library and set it as sLib-protected IP-Code. For details on FIR lowpass filter, please refer to the CMSIS-DSP-related documents as the subsequent sections focus only on how to set sLib to protect such algorithm and how to be called by end user programs.

In the example, the input signals of low-pass filter is from two sine wave signals with 1KHz and 15KHz respectively. The cut-off frequency is 6KHz for this low-pass filter. After going through low-pass filter, 15KHz signal is filtered, leaving only 1KHz sine wave output.

Figure 8 shows a diagram of FIR low-pass filter function.

Figure 8. FIR low-pass filter



The following CMSIS DSP functions and files will be used:

- arm_fir_init_f32()

This is used to initialize filter functions, and it is included in the *arm_fir_init_f32.c*.

- arm_fir_f32()

This is a main part of a filter algorithm, and it is included in the *arm_fir_f32.c*.

- FIR_lowpass_filter()

This is a global function of FIR low-pass filter, written on the basis of the two above functions. It is called by end user applications. It is included in the *fir_filter.c*.

- fir_coefficient.c

This .C file contains coefficients used in the FIR filter. These coefficients are read-only constants. In the example, they are placed in the read-only sLib.

In the example, FPU and DSP instructions embedded in the device are used to handle signals and for floating point operation in order to guarantee correct operation and output signals.

3.4 Project_L0 example for solution providers

To begin with, the following procedures need to be operated:

- Compile algorithm-related functions as execute-only ones;
- Place algorithm code in sector 4 of main Flash memory;

- Place coefficients of filter functions in the sector 2 of main Flash memory;
- Execute “FIR_lowpass_filter()” in the main program to verify;
- After successful verification, set sector 4 as a sLib instruction area, and sector 2 as a read-only sLib area. This step can be done by calling “slib_enable()” in the main program of this example, or by using Artery ICP Programmer tool (this option is recommended).
- Generate header files and symbol definition files used for calling low-pass filter functions by end user programs.

3.4.1 Generate execute-only code

Every toolchain offers its own setting options used to avoid the generation of literal pools and branch table by compiler, for they may produce a format of instruction reading data when an instruction is executed, for example, LDR Rn, [PC, #offset], etc.

For more information on literal pools and branch table, please refer to *Section 2.4 Set and run sLib*

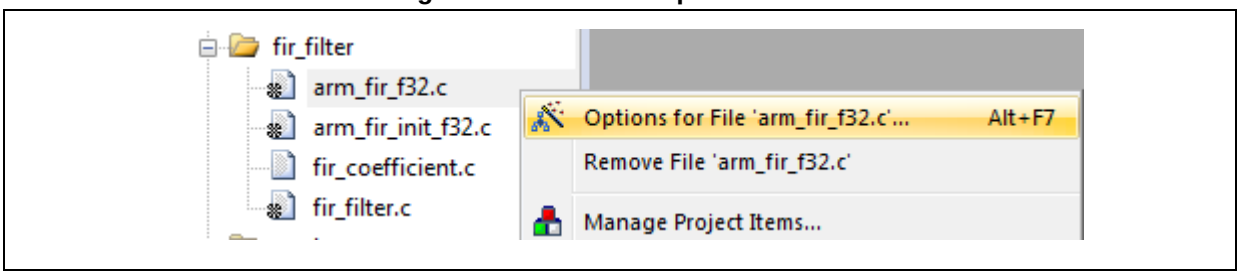
Taking Keil® µvision as an example, Keil® µvision has an option “Execute-only Code” to conduct settings below:

Keil® µvision: use Execute-only Code option

Proceed as follows:

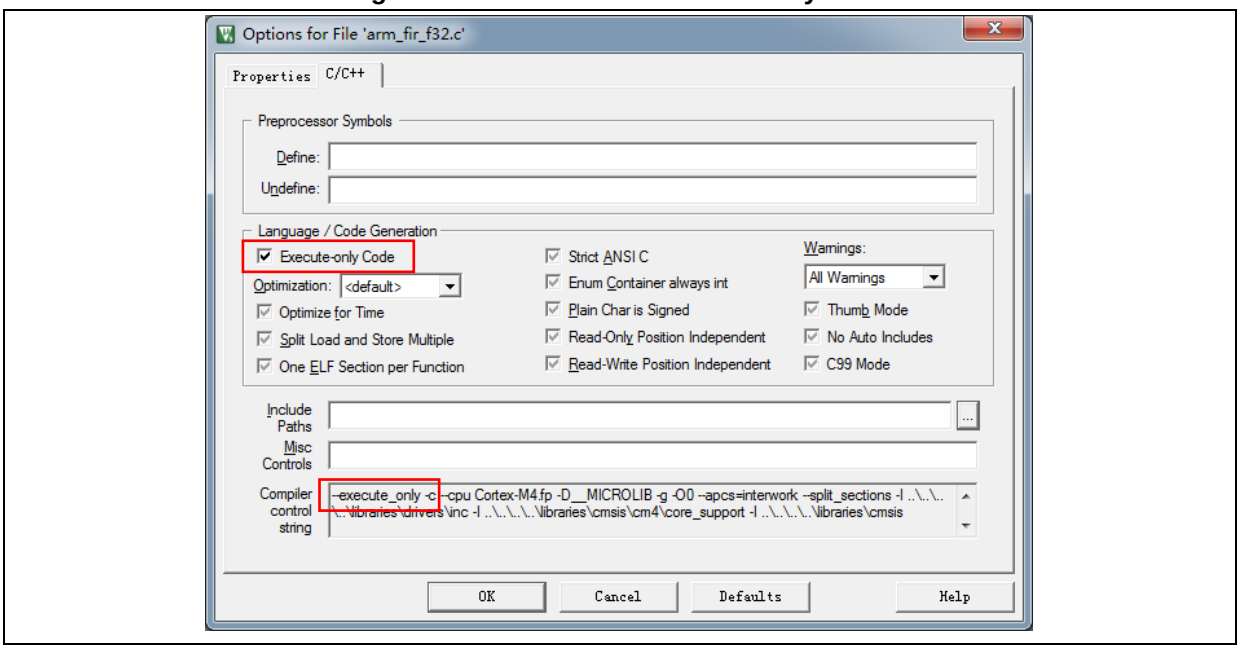
- Choose a C file group or an individual C file. In the example, the would-be protected C files are included in the fir_filter group
- Right click and choose corresponding file, for example, “Option for File ‘arm_fir_f32.c’” as shown in Figure 9.

Figure 9. Keil enters Option window



- In “C/C++” window, check “Execute-only Code” option, then the “--execute_only” command is added to the compiler control string, as shown in Figure 10 below.

Figure 10. Keil chooses Execute-only Code



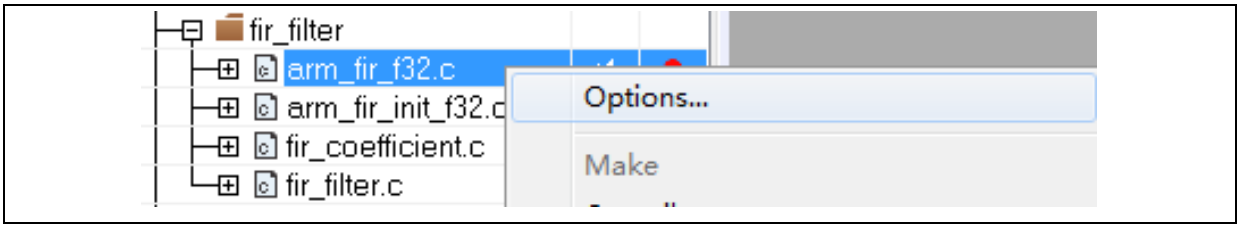
- There three files of arm_fir_f32.c, arm_fir_init_f32.c and fir_filter.c in SLIB_INSTRUCTION area. All of three must be configured as Execute-only code.

IAR: use “No data read in code memory” option

Proceed as follows:

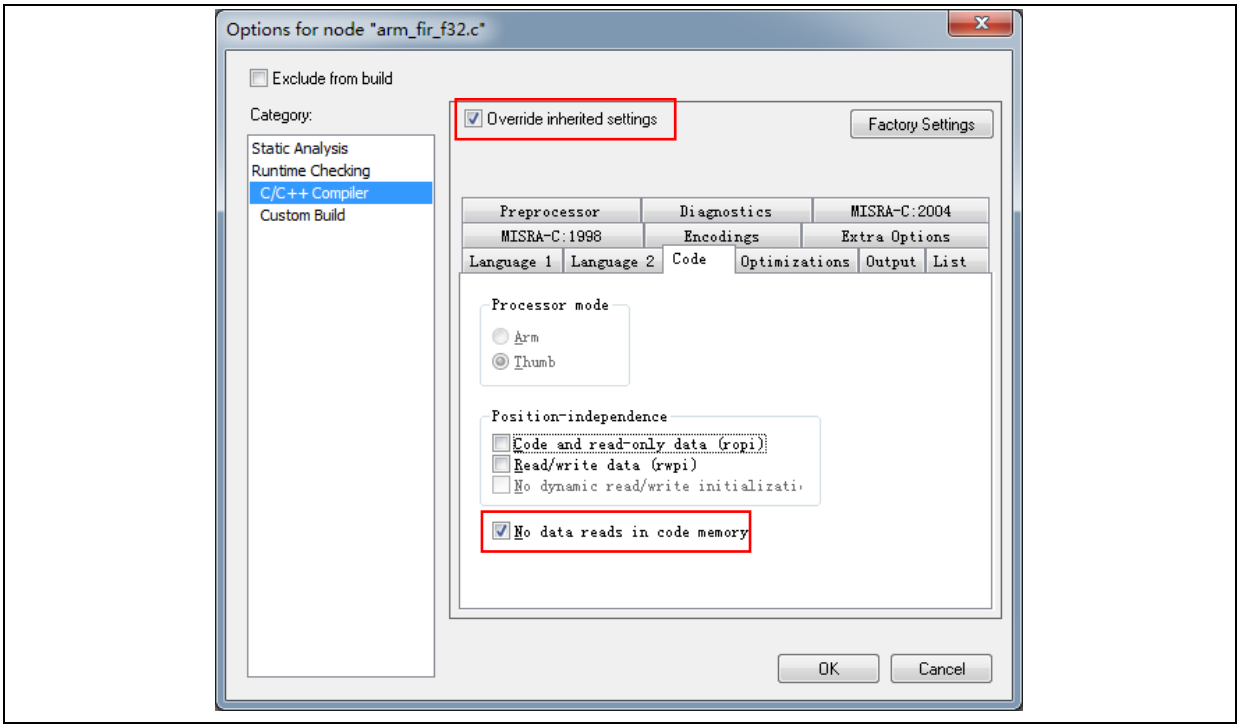
- Choose a particular file from “fir_filter” group, and right click and choose “Options”

Figure 11. IAR enters “Options” window



- In Figure 12 below, in “C/C++” screen, check “Override inherited settings” and “No data read in code memory” options

Figure 12. IAR C/C++ window

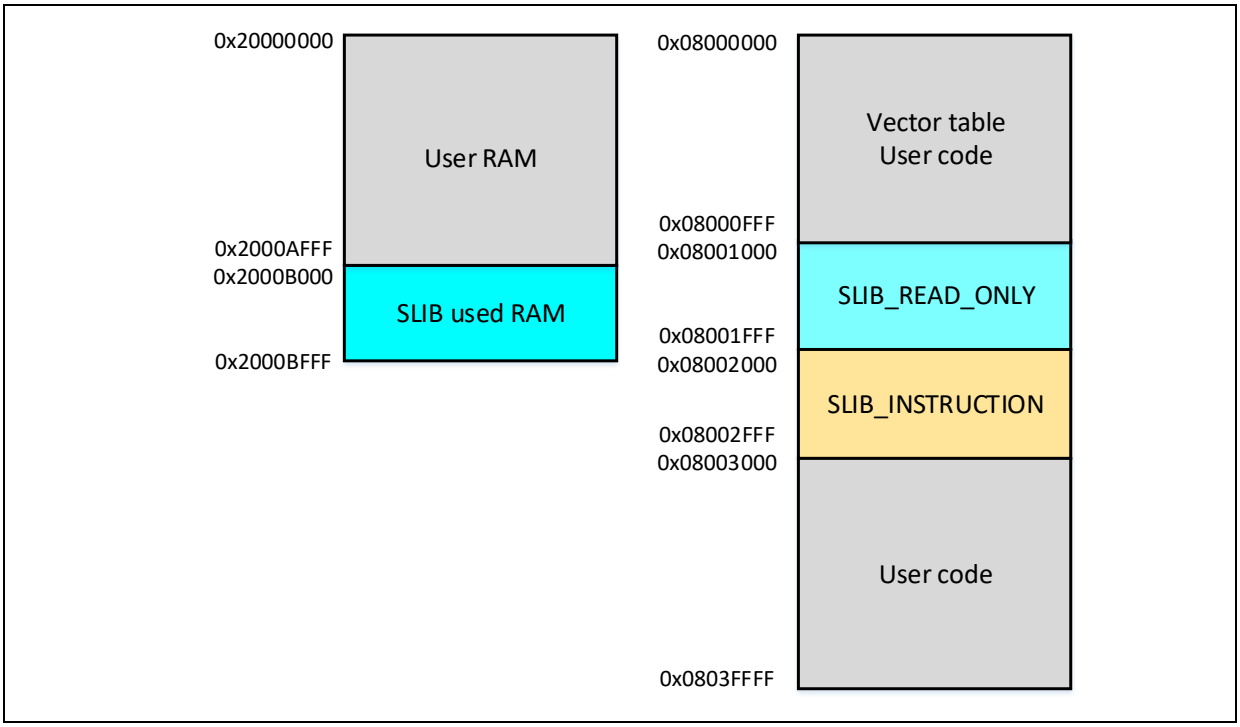


- There are three files of *arm_fir_f32.c*, *arm_fir_init_f32.c* and *fir_filter.c* in the SLIB_INSTRUCTION area. All of three must be configured as execute-only code.

3.4.2 Set sLib addresses

As mentioned in the previous sections, the sector 0 of Flash memory is used to store interrupt vector tables, and thus the example case sets sLib starting from sector 2. Note that sector 4 represents sLib instruction, and sector 2 represents a read-only sLib. Figure 13 below shows Flash memory map and RAM range distribution. The RAM segment is mainly aimed at preventing the use of the same RAM by sLib-protected code and user code.

Figure 13. Flash memory map and RAM segment in the example

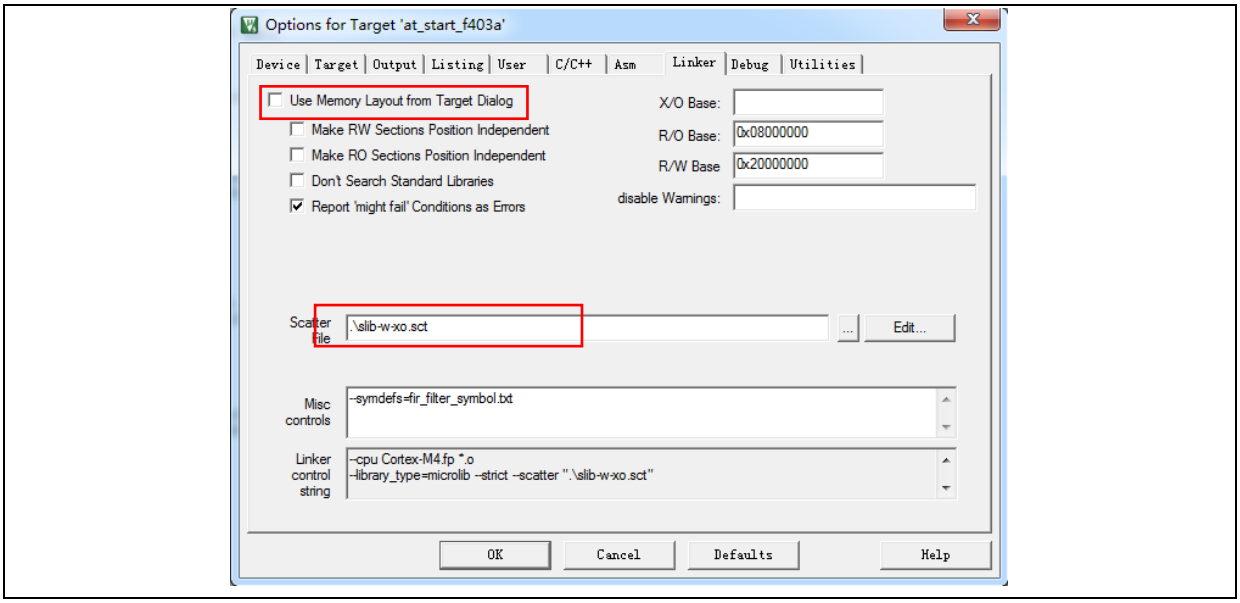


Keil® µvision's scatter file

Follow the procedures below:

- Go to Project → Options for Target→Linker, cancel "Use memory layout from Target Dialog" option, and then "Edit" to open "slib-w-xo.sct" for modification, as shown in Figure 14.

Figure 14. Linker settings in Keil



- After opening "scatter file", place an object file of the code which needs to be put in the SLIB_INSTRUCTION area into a dedicated load area named "LR_SLIB_INSTRUCTION", and change its mark to "execute-only (+XO)". This load area stars with sector 4 with a size of sector. Meanwhile, it is necessary to reserve SLIB_READ_ONLY area and place it into a specialized load area named "LR_SLIB_READ_ONLY". This is to avoid compiler to store other non-IP-Code functions into sLib.

  RW_IRAM2 block ranges from 0x2000B000 to 0x2000BFFF. It is assigned to algorithm functions of sLib, with the aim of preventing end-user projects from using the same RAM block to cause program error.

Figure 15. Keil scatter modification

```
LR_IROM1 0x08000000 0x001000  {    ; load region size_region
  ER_IROM1 0x08000000 0x001000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
  }

  RW_IRAM1 0x20000000 0x0000B000  {  ; user RW data
   .ANY (+RW +ZI)
  }

  RW_IRAM2 0x2000B000 0x00001000  {  ; RAM used for slib code
   fir_filter.o (+RW +ZI)
  }
}

LR_SLIB_READ_ONLY 0x08001000 0x00001000  {    ; sLib read-only area
  ER_SLIB_READ_ONLY 0x08001000 0x00001000  {
   fir_coefficient.o (+RO)
  }
}

LR_SLIB_INSTRUCTION 0x08002000 0x00001000  {    ; slib instruction area
  ER_SLIB_INSTRUCTION 0x08002000 0x00001000  {  ; load address = execution address
   arm_fir_init_f32.o (+XO)
   arm_fir_f32.o (+XO)
   fir_filter.o (+XO)
  }
}

LR_IROM2 0x08003000 0x0003D000  {    ; user code area
  ER_IROM2 0x08003000 0x0003D000  {  ; load address = execution address
   .ANY (+RO)
  }
}
```

- With regard to the use of RAM, in addition to above-mentioned method, it is also possible to use Keil's "__attribute__((at(address)))" descriptor to place variables at a fixed address of 0x2000B000, as shown in Figure 16 below.

Figure 16. RAM address change in Keil

```
#if   defined ( __ICCARM__ )
static float32_t firStateF32[BLOCK_SIZE + NUM_TAPS - 1]  @ 0x2000B000 ;
#elif defined ( __CC_ARM )
static float32_t firStateF32[BLOCK_SIZE + NUM_TAPS - 1] __attribute__((at(0x2000B000)));
#endif
```

- Read-only sLib area starts with sector 2 (0x08001000). Constants used in FIR low-pass filter functions should be placed at this address. In addition to the above "scatter file change" method, it is also possible to use Keil's "__attribute__((at(address)))" descriptor to place constants at a fixed address as shown in Figure 17.

Figure 17. Constant address change in Keil

```
#if   defined ( __ICCARM__ )
const float32_t firCoeffs32[NUM_TAPS] @ 0x08001000 ={
#elif defined ( __CC_ARM )
const float32_t firCoeffs32[NUM_TAPS] __attribute__((at(0x08001000))) = {
#endif
  -0.0018225230f, -0.0015879294f, +0.0000000000f, +0.0036977508f, +0.0080754303f, +0.0085302217f, -0.00(
  -0.0341458607f, -0.0333591565f, +0.0000000000f, +0.0676308395f, +0.1522061835f, +0.2229246956f, +0.25(
  +0.1522061835f, +0.0676308395f, +0.0000000000f, -0.0333591565f, -0.0341458607f, -0.0173976984f, -0.00(
  +0.0080754303f, +0.0036977508f, +0.0000000000f, -0.0015879294f, -0.0018225230f
};
```

IAR's ICF file

Proceed as follows:

- Open "icf" file under project_l0\IAR_V8.2, add three new load area as shown below. Here, SLIB_RAM area from 0x2000B000 to 0x2000BFFF is reserved for algorithm functions.

Figure 18. SLIB address definition in icf file

```
/* SLIB read-only area */
define symbol __ICFEDIT_region_SLIB_READ_ONLY_start__ = 0x08001000;
define symbol __ICFEDIT_region_SLIB_READ_ONLY_end__   = 0x08001FFF;

/* SLIB instruction area */
define symbol __ICFEDIT_region_SLIB_INST_start__ = 0x08002000;
define symbol __ICFEDIT_region_SLIB_INST_end__   = 0x08002FFF;

define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__   = 0x2000BFFF;

/* SLIB RAM region */
define symbol __ICFEDIT_region_SLIB_RAM_start__ = 0x2000B000;
define symbol __ICFEDIT_region_SLIB_RAM_end__   = 0x2000BFFF;
```

- In ICF file, sLib area should also be reserved in order to prevent non-IP-Code functions from being placed into sLib area by compiler. At the same time, the RAM used for IP-Code should be reserved as well.

Figure 19. Address distribution in icf file

```
/* Reserved 0x08001000 ~ 0x08002FFF as SLIB area */
define region ROM_region   = mem:[from __ICFEDIT_region_ROM_start__   to __ICFEDIT_region_ROM_end__]
                           -mem:[from __ICFEDIT_region_SLIB_READ_ONLY_start__ to __ICFEDIT_region_SLIB_READ_ONLY_end__]
                           -mem:[from __ICFEDIT_region_SLIB_INST_start__ to __ICFEDIT_region_SLIB_INST_end__];

define region SLIB_READ_ONLY_region   = mem:[from __ICFEDIT_region_SLIB_READ_ONLY_start__   to __ICFEDIT_region_SLIB_READ_ONLY_end__];

define region SLIB_INST_region   = mem:[from __ICFEDIT_region_SLIB_INST_start__   to __ICFEDIT_region_SLIB_INST_end__];

/* Reserved 0x2000B000 ~ 0x2000BFFF as RAM used for SLIB code */
define region RAM_region   = mem:[from __ICFEDIT_region_RAM_start__   to __ICFEDIT_region_RAM_end__]
                           - mem:[from __ICFEDIT_region_SLIB_RAM_start__   to __ICFEDIT_region_SLIB_RAM_end__];

define region SLIB_RAM_region = mem:[from __ICFEDIT_region_SLIB_RAM_start__   to __ICFEDIT_region_SLIB_RAM_end__];
```

- For RAM used for IP-Code, it is possible to place variables at a fixed address of 0x2000B000 through IAR's @ descriptors, or follow Figure 20 to change .icf file.

Figure 20. Modify RAM in icf file

```
/* Place IP Code in instruction area which will be SLIB protected */
place in SLIB_INST_region   { ro object arm_fir_f32.o,
                              ro object arm_fir_init_f32.o,
                              ro object fir_filter.o};

/* Place SLIB DATA(or CODE) in read-only area  */
place in SLIB_READ_ONLY_region   { ro object fir_coefficient.o };


place in RAM_region   { readwrite,
                        block CSTACK, block HEAP };

/* Place slib used sram */
place in SLIB_RAM_region   { readwrite object fir_filter.o };
```

- The start address for read-only sLib is sector 2 (0x08001000) which is to store constants used for FIR low-pass filter functions. In addition to the above-mentioned ICF file modification, it is also possible to place constants at a fixed address through IAR's @ descriptors.

Figure 21. sLib constant address change in IAR

```
#if   defined ( __ICCARM__ )
static float32_t firStateF32[BLOCK_SIZE + NUM_TAPS - 1] @ 0x2000B000 ;
#elif defined ( __CC_ARM )
static float32_t firStateF32[BLOCK_SIZE + NUM_TAPS - 1] __attribute__((at(0x2000B000)));
#endif
```

3.4.3 How to enable sLib function

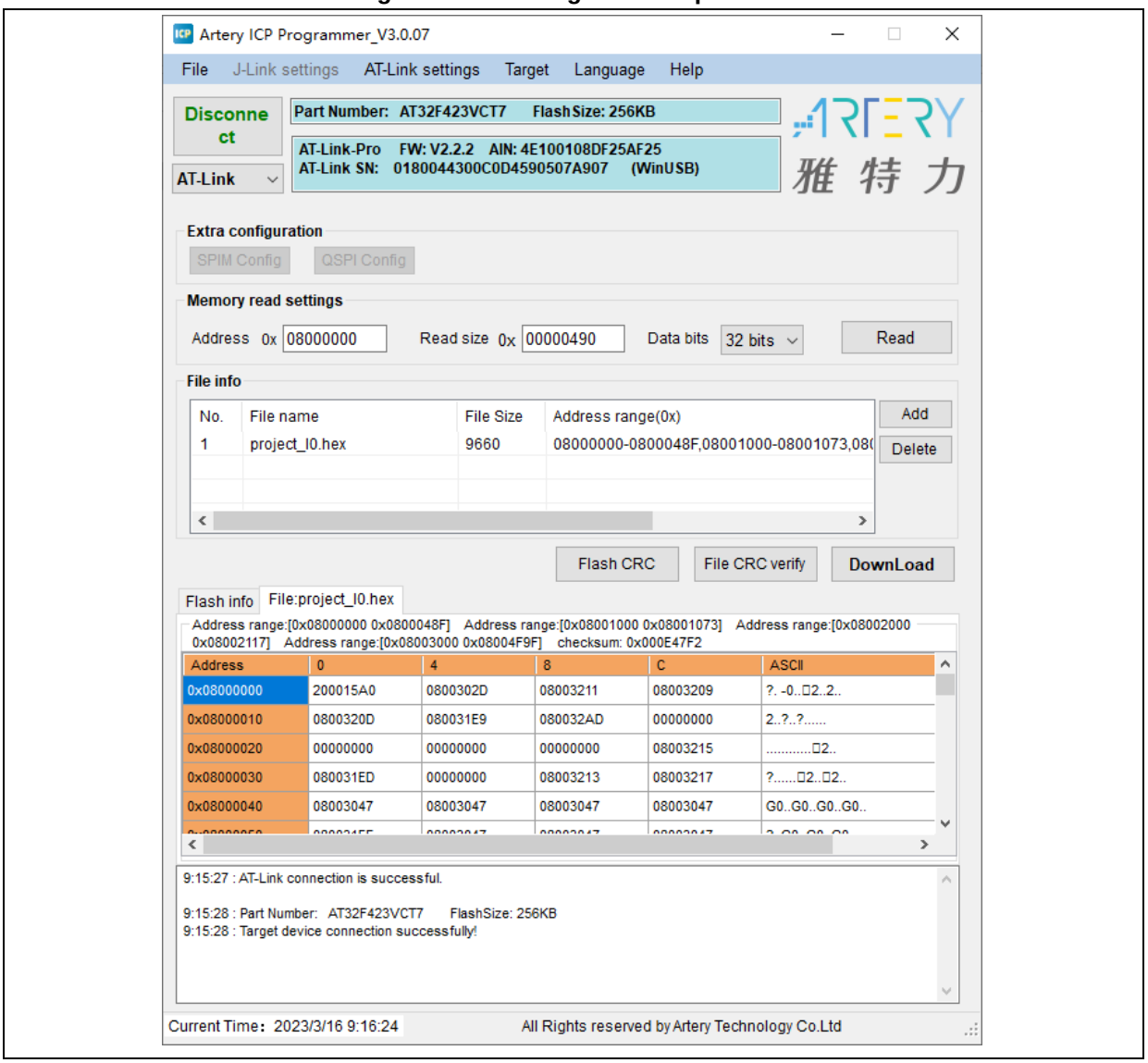
There are two ways to enable sLib function as follows:

(1) Use Artery ICP Programmer (recommended)

If use ICP Programmer, follow the steps below:

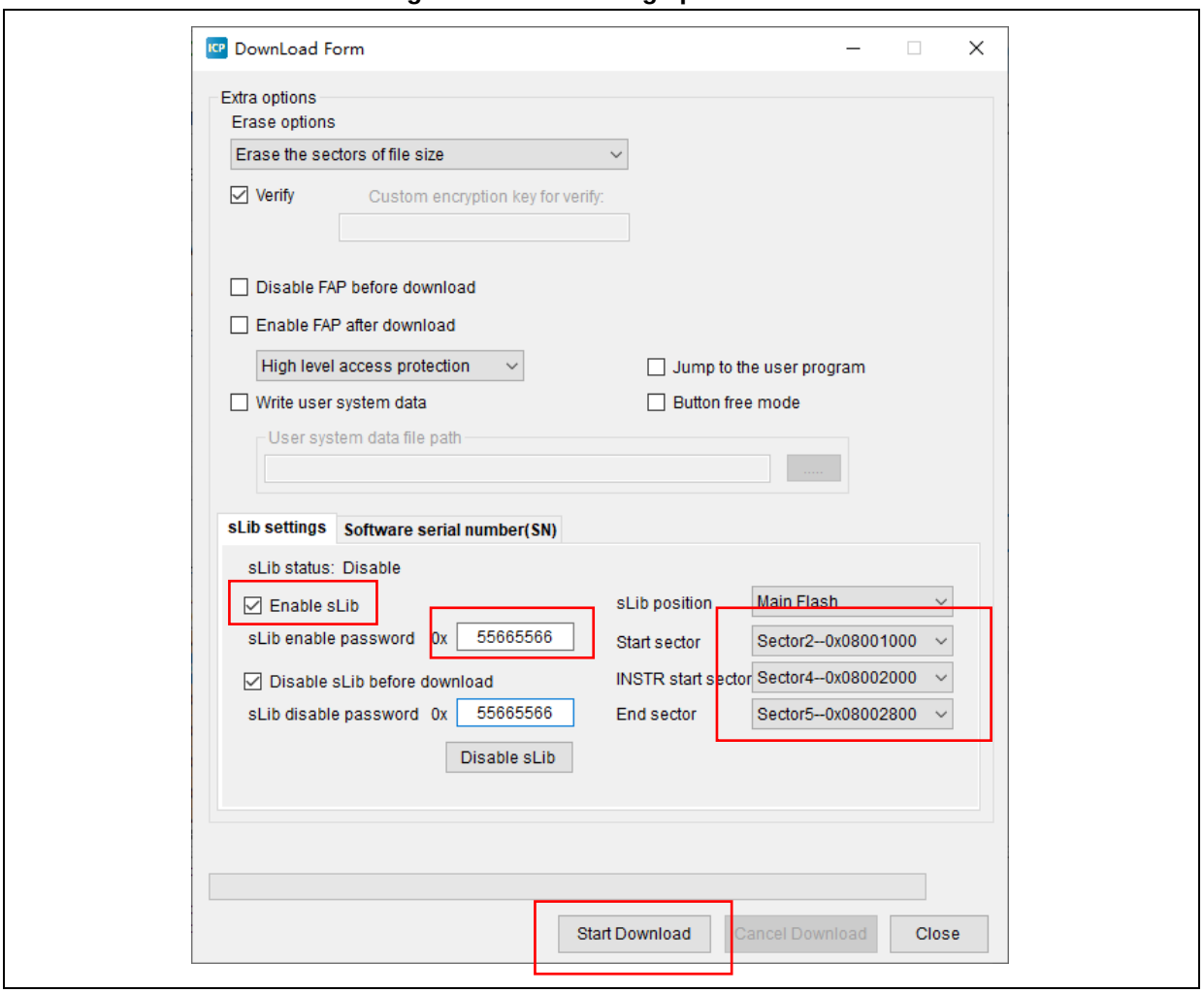
- Connect AT-Link to AT-START-F423 evaluation board and supply power to it
- Open ICP Programmer, select AT-Link connection, add Project_L0 example and generate HEX or BIN files, as shown below:

Figure 22. ICP Programmer operation



- Click “Download”, a “Download Form” will pop out displaying sLib-related settings parameters. Choose “sector 2” as a start sector, “sector 4” as an instruction start sector, and “sector 5” as an end sector. Then set a password 0x55665566 (customizable) for enabling sLib, and check “Enable sLib” option, and click “Start download”. In this way, it is ready for you to start programming and enable sLib, as shown in Figure 23.

Figure 23. sLib settings parameters



For details on the use of ICP Programmer, please refer to ICP Programmer user guide manual.

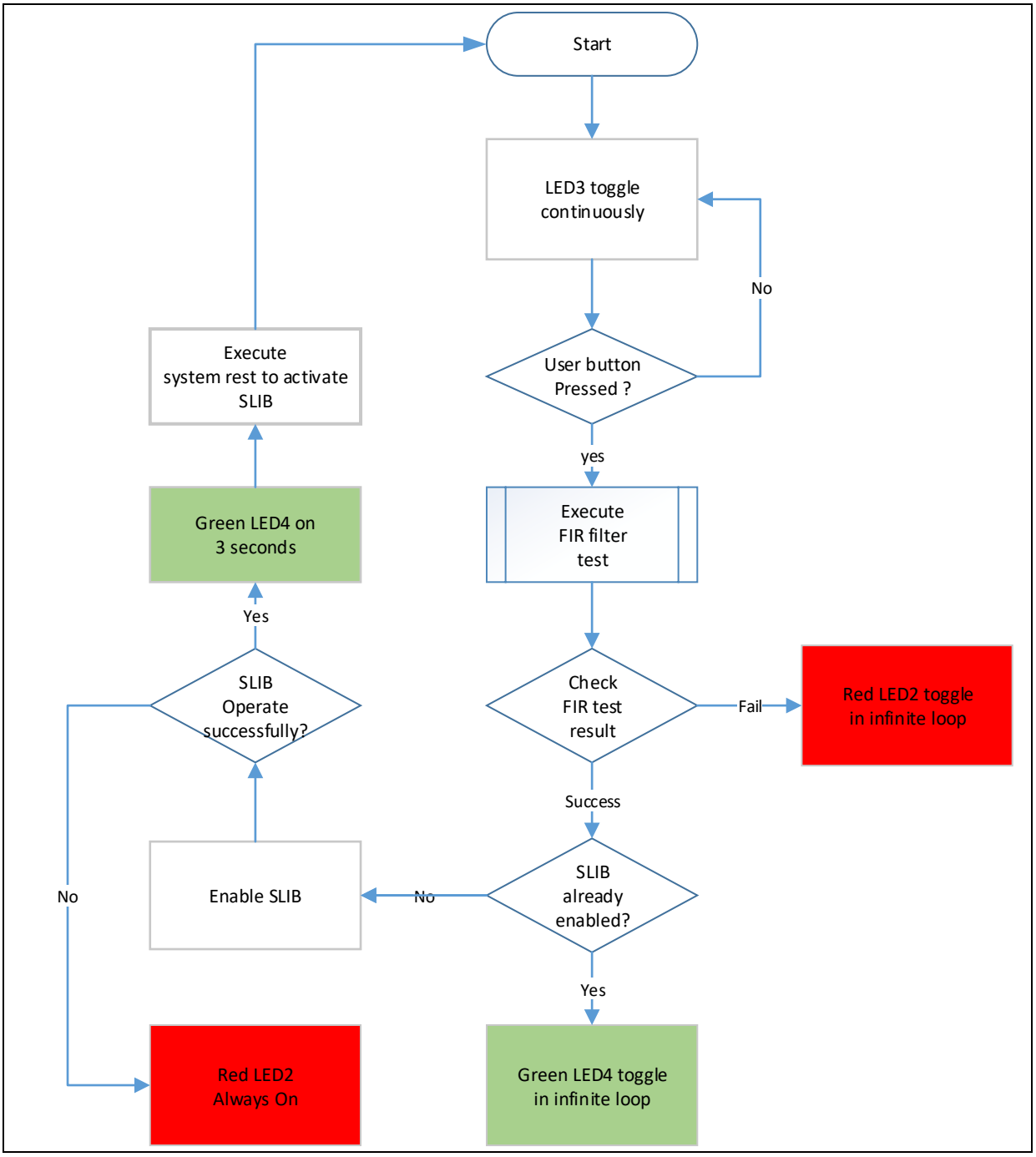
(2) Use “slib_enable()” function in the main.c

Executing “slib_enable()” once after successful low-pass filter function test can allow users to enable sLib feature. The function “slib_enable()” can be executed by simply enabling “#define USE_SLIB_FUNCTION” in main.c.

3.4.4 Project_L0 flow chart

In this example, FIR low-pass filter calculates the input signal “testInput_f32_1kHz_15kHz” — a mixed signal of 1KHz and 15KHz sine waves, and outputs a 1KHz sine-wave data and stores it at testOutput. Then this output data will be compared with MATLAB-calculated data stored at refOutput. If error is lower than expected (signal to noise ratio SNR is greater than pre-defined threshold), a green LED on the evaluation board will start blinking; otherwise, a red LED will start blinking. Figure 24 shows a flow chart of Project_L0.

Figure 24. Project_L0 flow chart



To run this example code, follow the procedures below:

(1) Use Keil® µvision to open Project_L0 under utilities\AT32F423_slib_demo\project_l0\mdk_v5, and start compiling

(2) Prior to download, first check sLib or read/write protection (FAP/EPP) is enabled for the AT-START-F423 evaluation board. If enabled, use ICP tool to unlock this protection before starting download;

(3) After successful download and execution, LED3 on the board will keep blinking;

(4) Press “USER” button on the board to start low-pass filter operation

(5) Compare operation results, if correct, green LED4 will start blinking, otherwise, red LED2 starts flashing;

(6) On the premise that operation results are correct, if USE_SLIB_FUNCTION in main.c is already defined and sLib is not enabled, then the slib_enable() will be executed to set sLib. If sLib settings failed, red LED2 will be always ON; If successful, a green LED4 will flash for around 3s and start to perform system reset to enable sLib. Next, program returns to step (3).

3.4.5 How to generate header files and symbol definition files

Both header file and symbol definition file are required for Project_L1 to call FIR low-pass filter functions. In the example, header file refers to the “fir_filter.h” in the main.c.

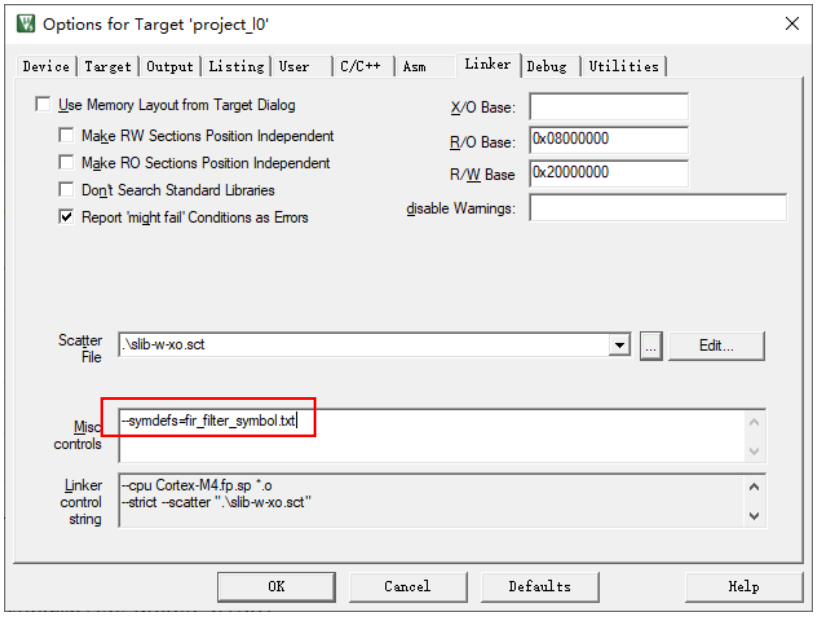
How to generate symbol definition file depends on toolchains used.

Use Keil® µvision to generate a symbol definition file

Follow the procedures as follows:

- Go to Options for Target → Linker window
- In “Misc controls” column (as shown in Figure 25), add the command “--symdefs=fir_filter_symbol.txt”

Figure 25. Keil Misc controls option



- After compiling the whole project, a symbol definition file named “fir_filter_symbol.txt” is created under project_l0\mdk_v5\Objects.

- Such symbol definition file contains all symbol definitions related to the project, and thus some of them should be removed so as to reserve low-pass filter function definitions which will be used by end users. The reduced definition fir_filter_symbol.txt is shown below.

Figure 26. Reduced fir_filter_symbol.txt

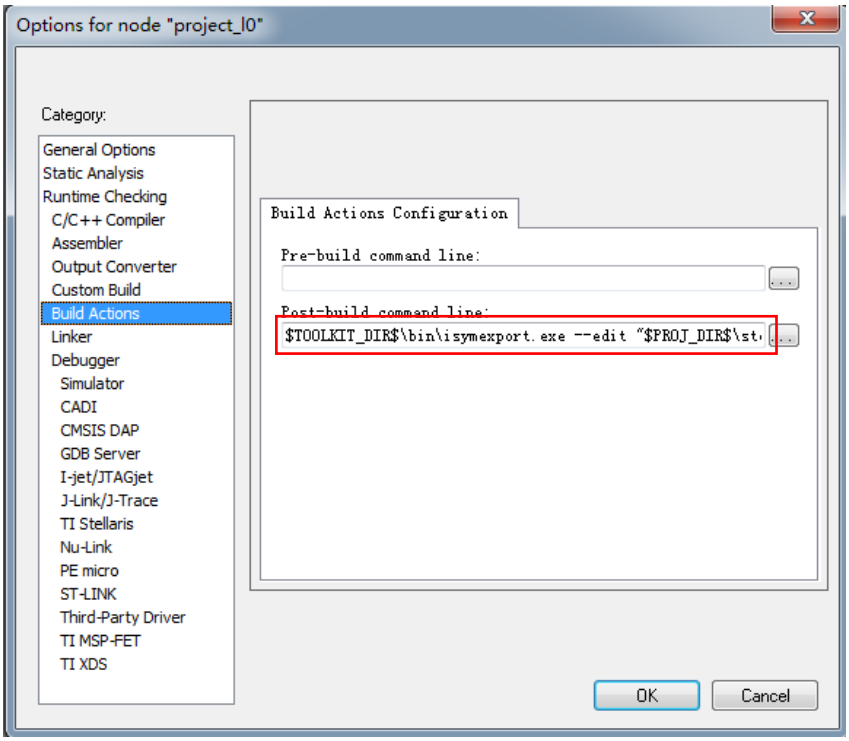
```
0x08002001 T FIR_lowpass_filter
```

Use IAR to generate symbol definition files

Follow the procedures below:

- Go to Project→Option→Build Actions

Figure 27. IAR Build Actions option



- Add the following command in the Post-build command line

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

- The “fir_filter_symbol.o” refers to a symbol definition file. The “steering_file.txt” stored under project_l0\iar_v8.2 is used to select which symbols of functions need to be created. Then edit them according to the contents in the sLib, as shown in Figure 28 in which “show” is used to select a function command.

Figure 28. Edit steering_file.txt

```
show FIR_lowpass_filter
```

3.5 Project_L1 example for end users

Project_L1 example needs to use FIR low-pass filter functions that are debugged in Project_L0 and programmed into AT32F423's Flash memory with sLib enabled.

Based on header file, symbol definition file and Flash memory map defined in Project_L0, end users are able to do the following on the basis of Project_L1 example:

- Create an application project
- Introduce header file and symbol definition file from Project_L0 into its project
- Call FIR low-pass filter functions
- Develop and debug user programs

Cautions:

Project_L1 must use the same toolchains and the same version of compiler as those of Project_L0 as differences between software versions may cause incompatibility issue, which in turn makes it impossible to use codes from Project_L0.

For example, Project_L0 uses Keil® µvision V5.36.0.0, so does Project_L1.

3.5.1 Create a user project

Considering that some Flash memory sectors have been occupied by sLib area enabled in Project_L0, the addresses in which Project_L1 codes are stored must be configured taking into account Flash memory map in Project_L0.

Figure 13 shows Flash memory map used in this example, where sector 5 to sector 5 are owned by sLib. It is necessary for end users to separate such sLib area (sector 2 to sector 5) from other areas through linker control file, so as to prevent codes from being placed into sLib.

Keil® µvision's scatter file

Based on the "end_user_code.sct" file under project_l1\mdk_v5, the users can divide main Flash memory into two segments. The area in between is sLib area. Besides, the space after the address 0x2000B000 is reserved for RAM, as shown in Figure 29.

Figure 29. Modified scatter file

```
LR_IROM1 0x08000000 0x00001000  {    ; load region size_region
  ER_IROM1 0x08000000 0x00001000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
  }
  RW_IRAM1 0x20000000 0x0000B000  {  ; RW data
   .ANY (+RW +ZI)
  }

 ; 0x2000B000 ~ 0x2000BFFF   RAM reserved for SLIB code

}

; 0x08001000 ~ 0x08002FFF is SLIB area

LR_IROM2 0x08003000 0x0003D000  {    ; load region size_region
  ER_IROM2 0x08003000 0x0003D000  {  ; load address = execution address
   .ANY (+RO)
  }
}
```

IAR's ICF file

The users can refer to the following content in the "enduser.icf" file which is stored under project_l1\iar_V8.2.

Figure 30. Modified icf file

```
define region ROM_region      = mem:[from __ICFEDIT_region_ROM_start__   to __ICFEDIT_region_ROM_end__]
                                -mem:[from __ICFEDIT_region_SLIB_start__ to __ICFEDIT_region_SLIB_end__];

define region RAM_region      = mem:[from __ICFEDIT_region_RAM_start__   to __ICFEDIT_region_RAM_end__]
                                - mem:[from __ICFEDIT_region_SLIB_RAM_start__   to __ICFEDIT_region_SLIB_RAM_end__];
```

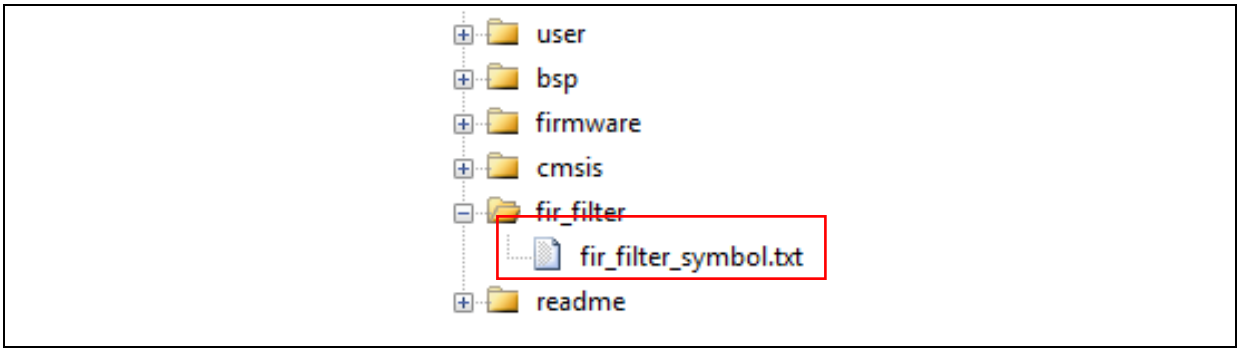
3.5.2 Add symbol definition file into project

The symbol definition file "fir_filter_symbol.txt" which is created in Project_L0 must be added to Project_L1 so that it can be correctly compiled and linked to sLib codes.

Add a symbol definition file in Keil® µvision environment

Add the "fir_filter_symbol.txt" into project, as shown in Figure 31.

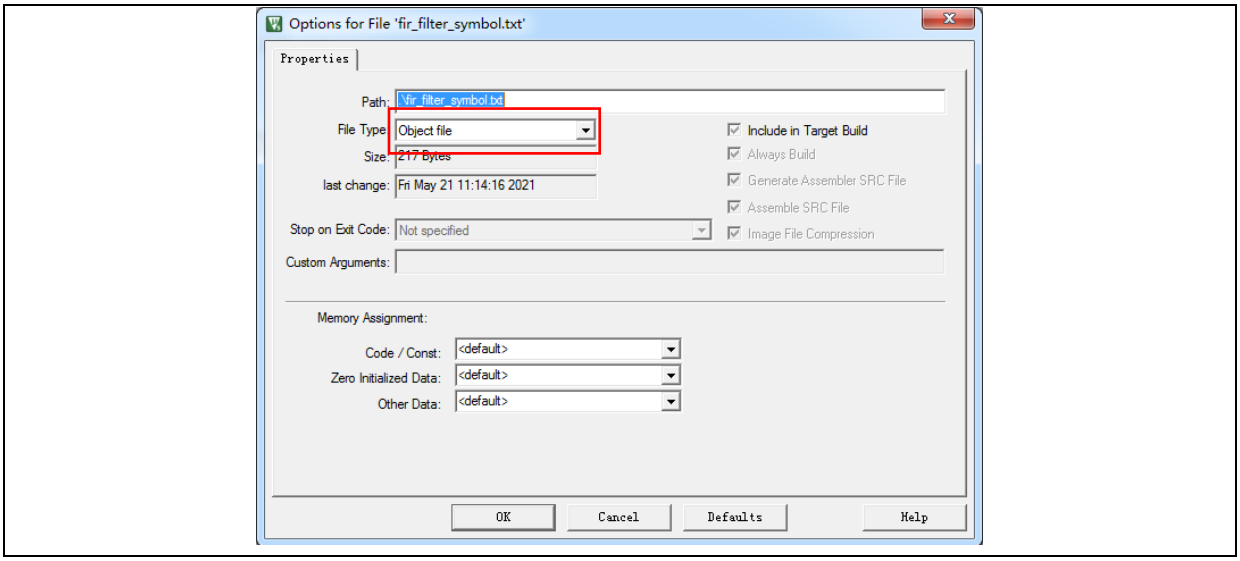
Figure 31. Add symbol definition file in Keil



After adding this file into "fir_filter" group, its file type must be changed into Object file, instead of its original text format.

Change it as follows:

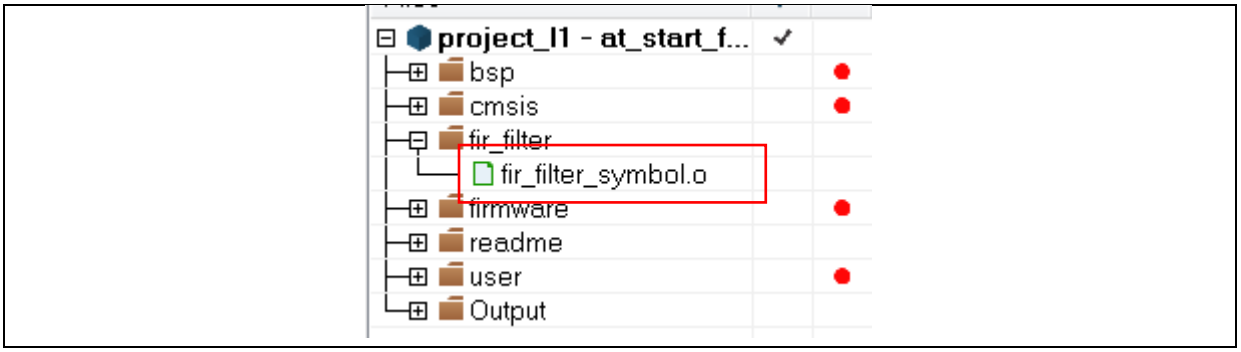
Figure 32. Change symbol definition file to Object file



Add symbol definition file in IAR environment

Add the “fir_filter_symbol.o” file into “fir_filter” group, as shown in Figure 33:

Figure 33. Add symbol definition file in IAR



3.5.3 Call sLib functions

After filter.h file is referenced by main.c and symbol definition file is successfully added into project, it is now ready to call low-pass filter functions from sLib area.

Proceed as follows:

FIR_lowpass_filter(inputF32, outputF32, TEST_LENGTH_SAMPLES);

With:

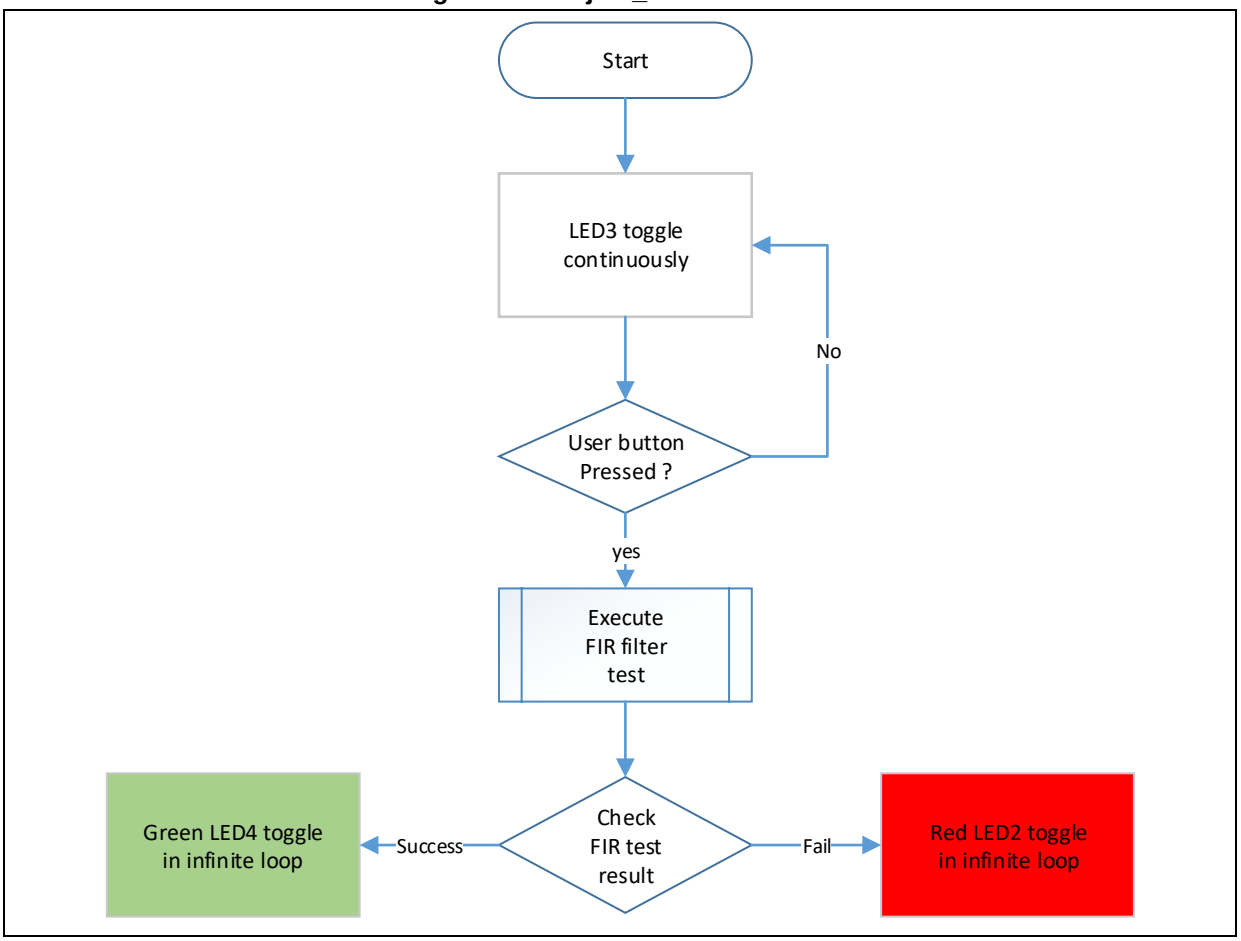
- *inputF32:* pointer to data table storing input signals
- *outputF32* : a pointer to data table storing output signals
- *TEST_LENGTH_SAMPLES*: the size of signal samples to be processed

3.5.4 Project_L1 flow chart

Project_L1 flow chart is shown in Figure 34:

- LED3 will start blinking upon execution;
- Press “USER” button onboard AT-START to start operating FIR_lowpass_filter();
- If operation result is correct, greed LED4 starts flashing, if failed, read LED2 starts flashing.

Figure 34. Project_L1 flow chart

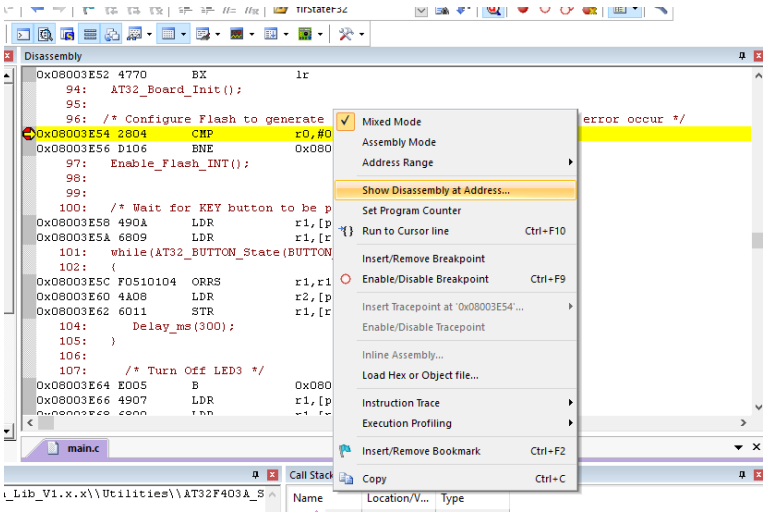


3.5.5 sLib protection in debug mode

Considering the fact that end users need to debug codes during application development, here we use Keil® µvision as an example to demonstrate how to prevent sLib codes from being read in debug mode.

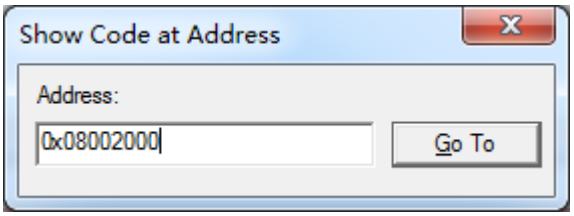
- Open Project_L1 and recompile;
- Click ”Start/Stop Debug Session” to enter debug mode;
- In ”Disassembly” window, right click and choose ”Show Disassembly at Address”, as shown in Figure 35.

Figure 35. "Show Disassembly at Address" window



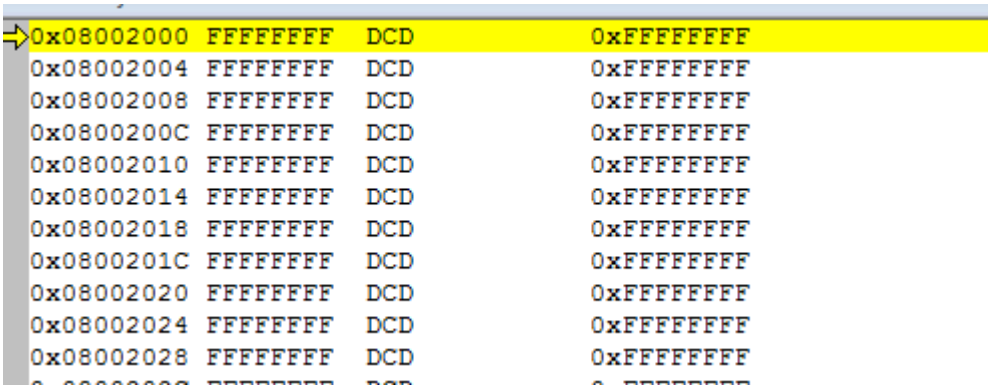
- Enter address 0x08002000, which is the start address (sector 2) of SLIB_INSTRUCTION.

Figure 36. "Show Code at Address" setting



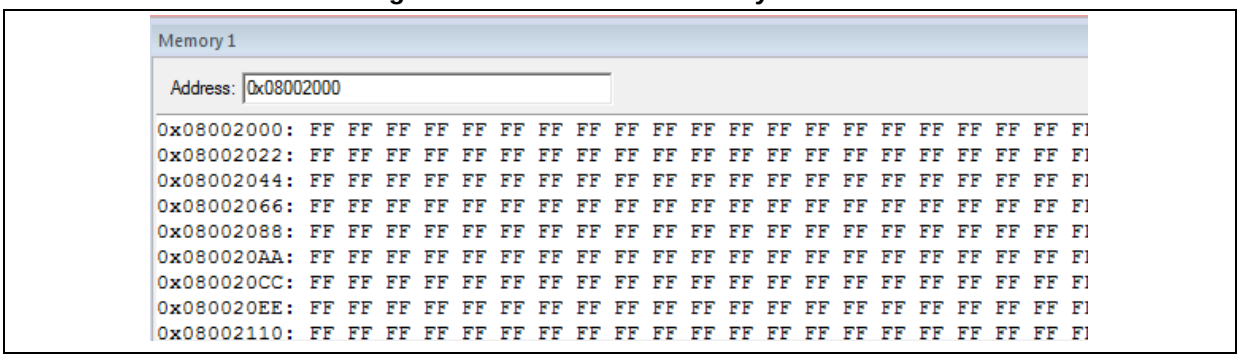
You can see that the address starts with 0x08002000, and all codes are 0xFFFFFFFF.

Figure 37. View code



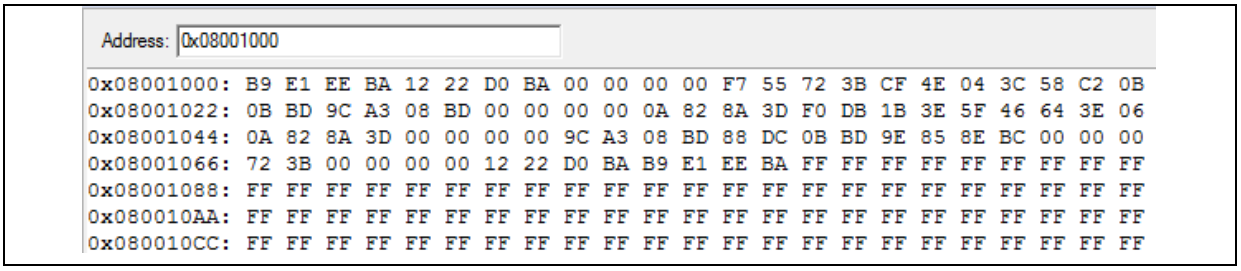
- Similarly, in "Memory" window, enter address 0x08002000 and return all 0xFF.

Figure 38. View code in Memory window



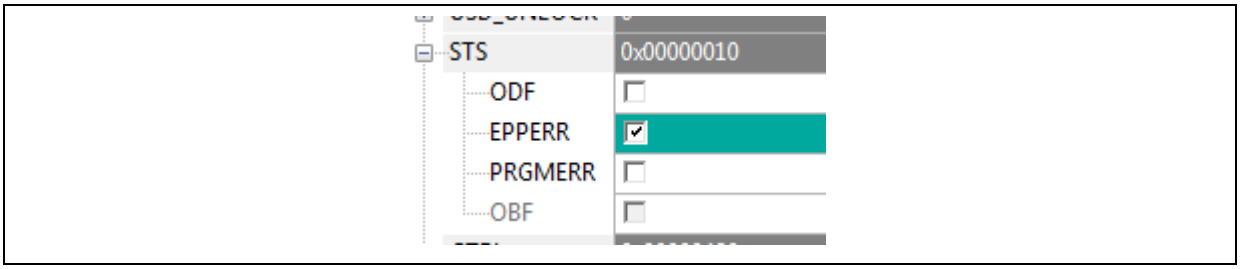
- In “Memory” window, enter 0x08001000 — the start address (sector 2) of SLIB_READ_ONLY. Because this area is readable by D-Code bus, we can see their original data.

Figure 39. View SLIB_READ_ONLY start sector in Memory



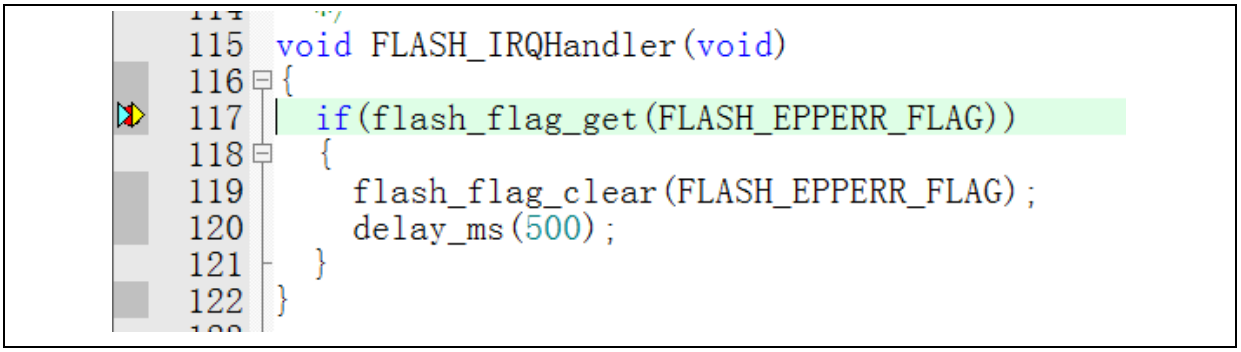
In “Memory” window, when you click data in the 0x08002000 twice to try to modify them, the EPPERR bit in the FLASH_STS register will be set to 1 as a warning, indicating that they are write protected.

Figure 40. SLIB write protection test



If write protection error interrupt is enabled, it will enter interrupt routine.

Figure 41. Write protection error interrupt



```
115  void FLASH_IRQHandler(void)
116  {
117    if(flash_flag_get(FLASH_EPPERR_FLAG))
118    {
119      flash_flag_clear(FLASH_EPPERR_FLAG);
120      delay_ms(500);
121    }
122  }
```

4 Integrate and download codes of solution provider and user

After the completion of code design on both solution providers and end users, these code should be downloaded into the same MCU device. In this scenario, data security issue should be taken into account. In the subsequent sections, two download procedures based on Project_L0 and Project_L1 are recommended as a reference. The procedures involves AT-Link offline download mode, with its details being described in ICP user guide and AT-Link user manual.

4.1 Write code separated on solution provider and end user

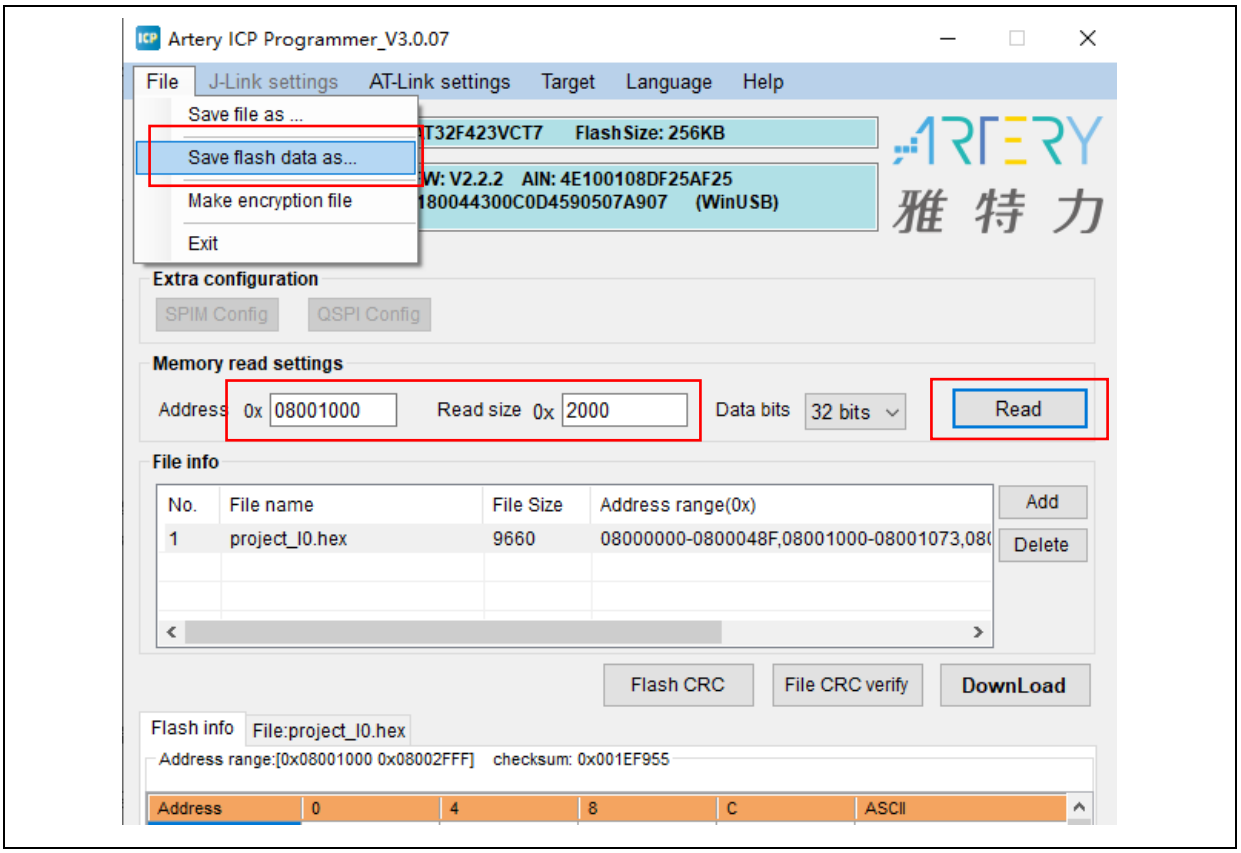
First, solution provider programs sLib codes into MCU, second, end user program application codes into MCU, as shown below:

(1) **Method A:**

The solution provider uses ICP to save the compiled sLib codes as BIN or HEX file:

First download the whole project to MCU (do not configure sLib, FAP, etc at this point), then read sLib code (address from 0x08001000 to 0x08002FFF) through memory read function. Finally, in ICP tool, click “File”, and choose “save flash data as” to save data as BIN or HEX. In Figure 42 below, slib.bin is a BIN file.

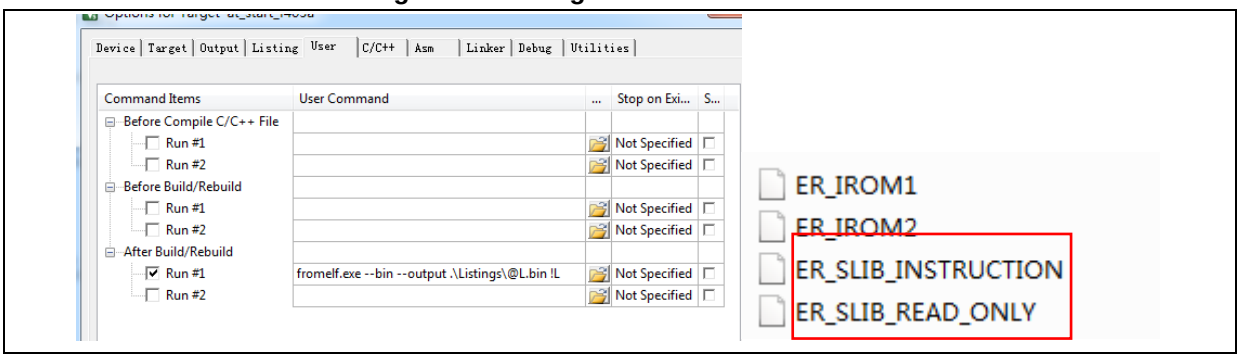
Figure 42. Save SLIB code



Method B: the solution provider uses the compiled project to directly generate BIN file. Choose a section of sLib area, for example, in Keil project, in “User” option, add “fromelf.exe --bin --output .\Listings\@L.bin !L” to produce a corresponding BIN file, that is, add a suffix “.bin” to this sLib file.

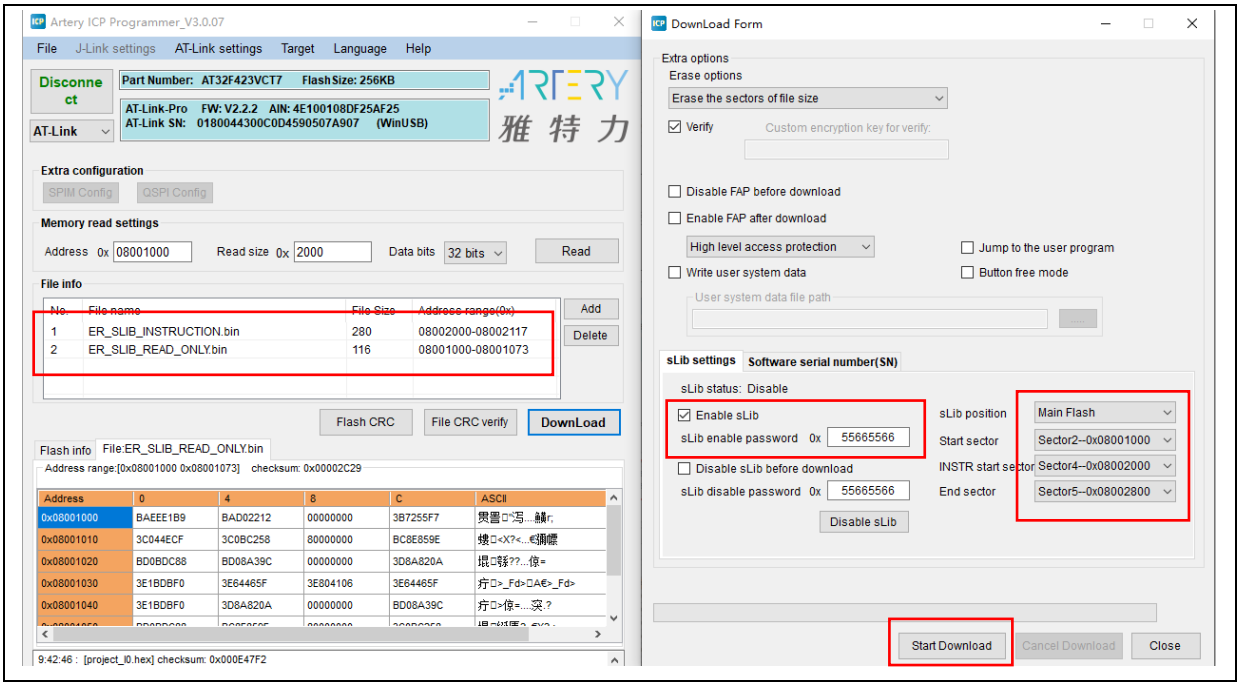
In this example, ER_SLIB_INSTRUCTION.bin and ER_SLIB_READ_ONLY.bin correspond to SLIB-INSTRUCTION file at 0x08002000 and SLIB-READ-ONLY file at 0x08001000 respectively.

Figure 43. Change SLIB code to BIN file



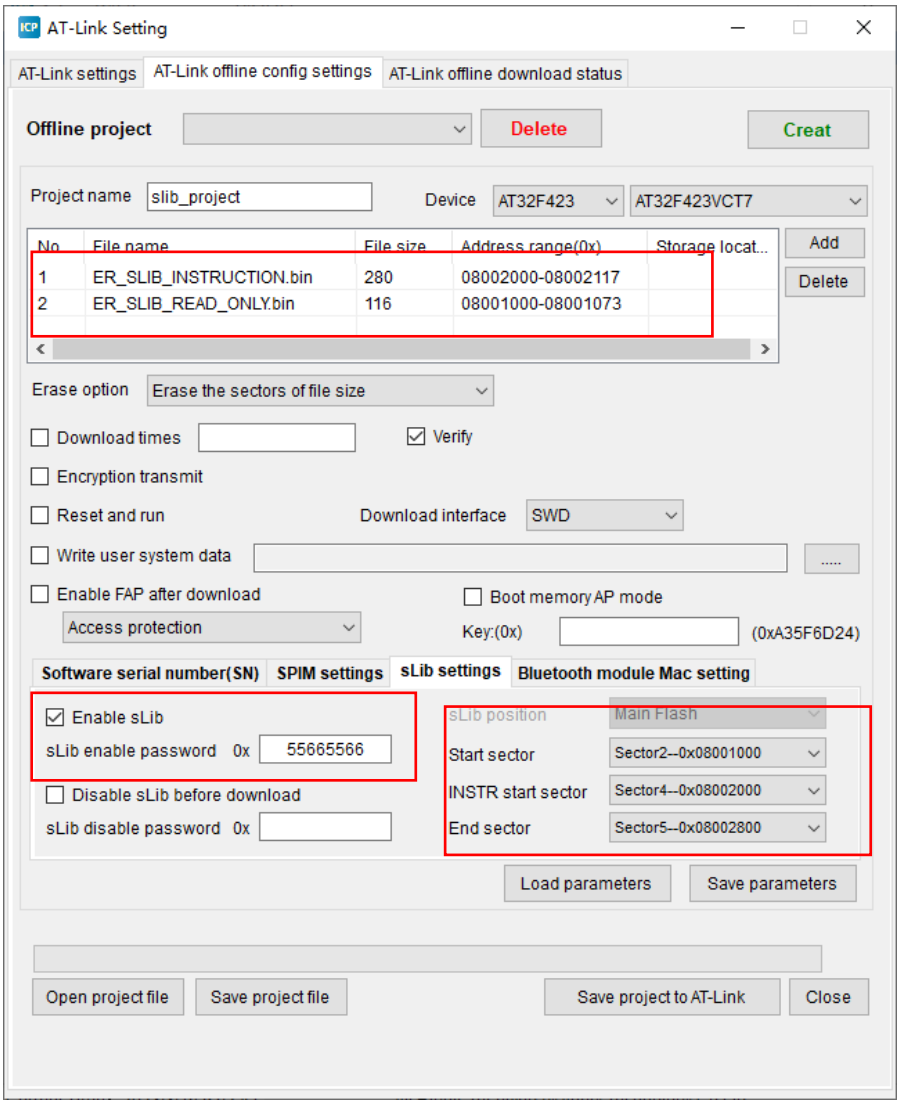
(2) Use ICP tool to program bin file into MCU online

Figure 44. ICP programs MCU online



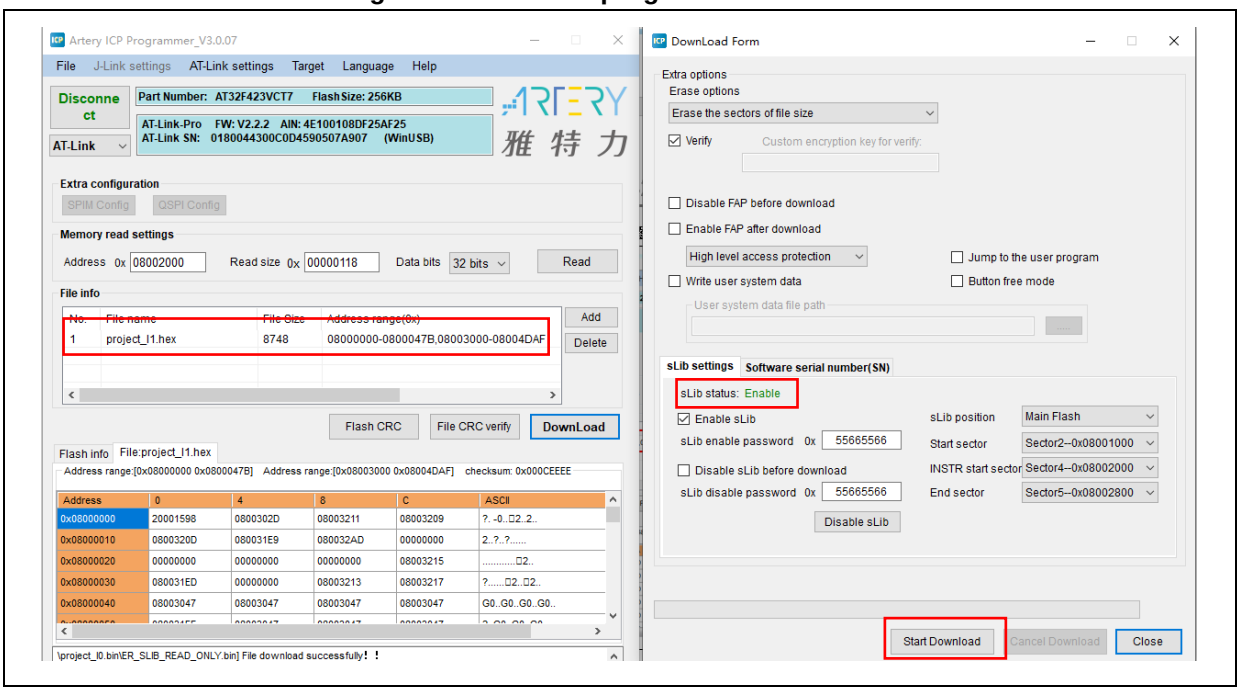
(3) Alternatively, use ICP tool to configure an offline project and save it to AT-Link, then program it into MCU through AT-Link offline mode, and save this offline project, as shown in Figure 45.

Figure 45. AT-Link programs MCU offline



(4) After step (2) or (3), a MCU device with the programmed sLib code is delivered to end user. In this case, sLib is already enabled, and the end user can program application code via online or offline mode to MCU to finish the rest of the process. Figure 46 gives an online programming example.

Figure 46. End user programs code to MCU



4.2 Combine solution provider code with end user code

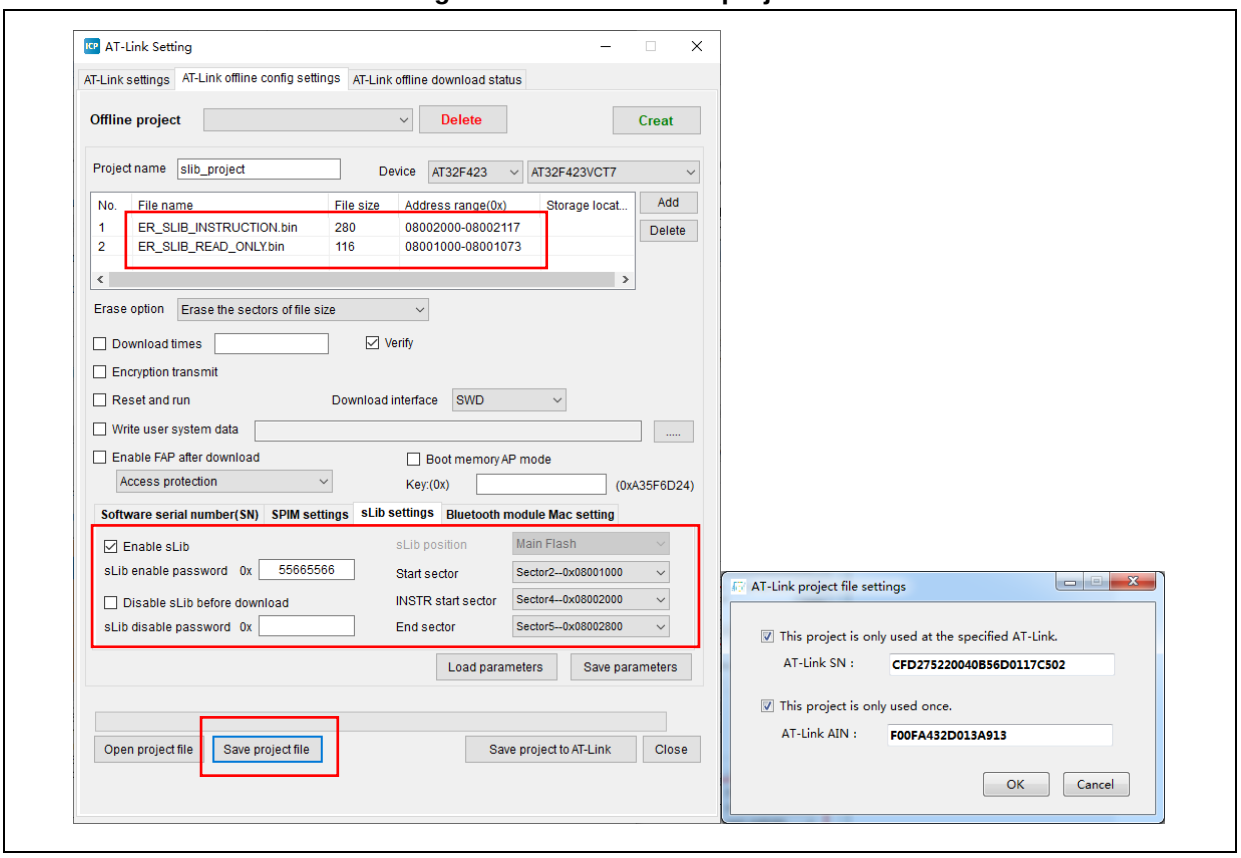
SLIB code from solution provider and end user code are integrated into an offline project, which is then downloaded into MCU via AT-Link offline mode.

(1) The solution provider creates a BIN format of sLib code according to the section 4.1.

(2) The solution provider uses ICP to create an offline project and save it to PC. Multiple parameters such as “download times”, “project bonded to AT-Link”, “enable FAP after download” and others can be configured according to actual needs, as shown in Figure 47.

Note: The offline project itself is encrypted. To enhance data security, the slib.bin can be changed into an encrypted slib.benc file for solution provider before being added to an offline project. But in this case, such offline project can only be accessible to the corresponding AT-Link with passkey.

Figure 47. Create offline project

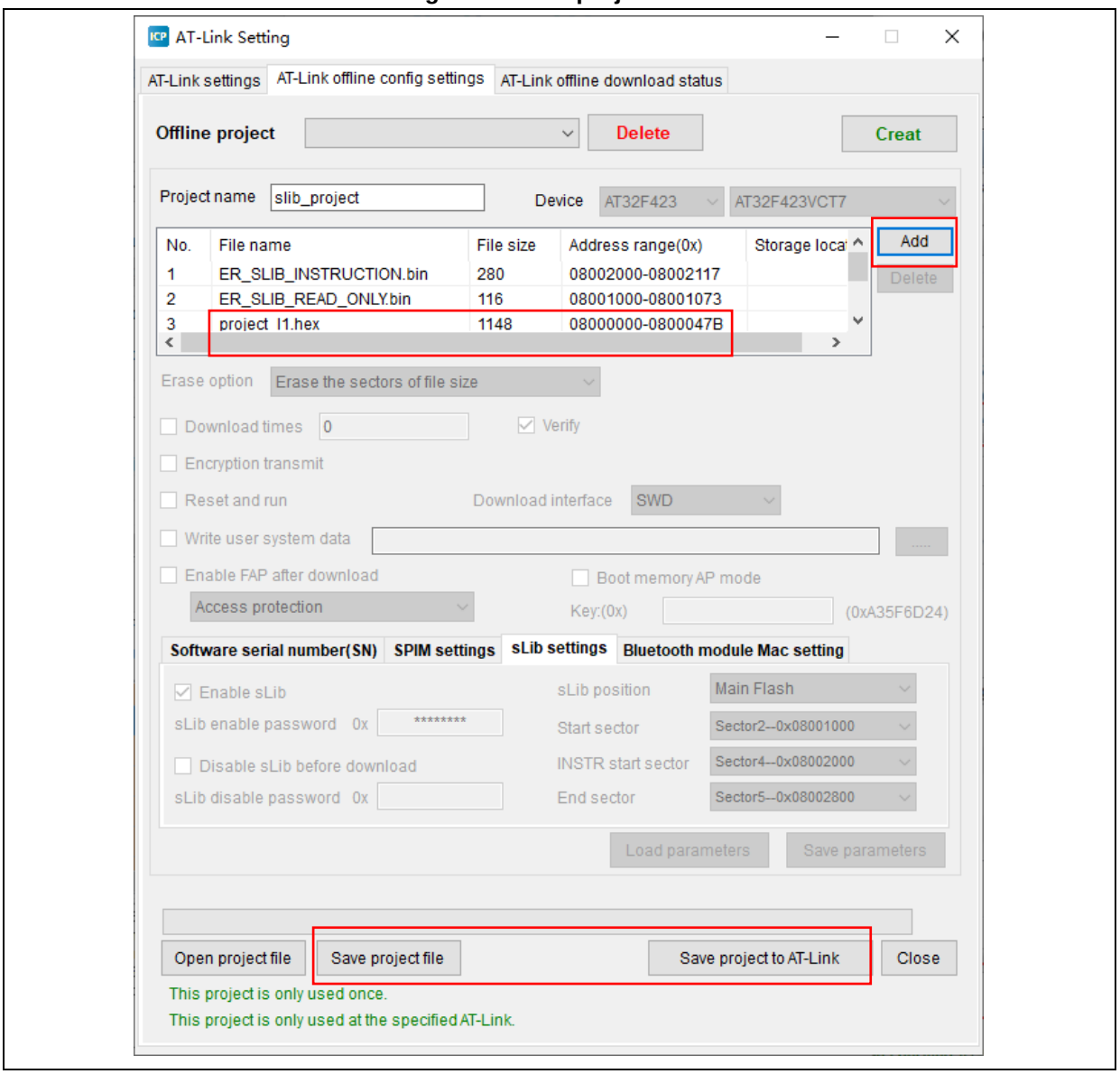


(3) For end users, they can use ICP to open such offline project, and click “Add” to add user application code into such project, and save it to PC or directly to AT-Link, and then perform offline download to finish the whole operation.

Figure 48 shows how to add a project file.

Note: To avoid code disclosure and cracking, it is forbidden to change parameters settings while adding code into an offline project. Based on this consideration, it is necessary for solution providers to configure final settings in advance.

Figure 48. Add project file



5 Revision history

Table 2. Document revision history

Date	Revision	Changes
2023.1.13	2.0.0	Initial release

IMPORTANT NOTICE – PLEASE READ CAREFULLY

Purchasers are solely responsible for the selection and use of ARTERY's products and services; ARTERY assumes no liability for purchasers' selection or use of the products and the relevant services.

No license, express or implied, to any intellectual property right is granted by ARTERY herein regardless of the existence of any previous representation in any forms. If any part of this document involves third party's products or services, it does NOT imply that ARTERY authorizes the use of the third party's products or services, or permits any of the intellectual property, or guarantees any uses of the third party's products or services or intellectual property in any way.

Except as provided in ARTERY's terms and conditions of sale for such products, ARTERY disclaims any express or implied warranty, relating to use and/or sale of the products, including but not restricted to liability or warranties relating to merchantability, fitness for a particular purpose (based on the corresponding legal situation in any unjudicial districts), or infringement of any patent, copyright, or other intellectual property right.

ARTERY's products are not designed for the following purposes, and thus not intended for the following uses: (A) Applications that have specific requirements on safety, for example: life-support applications, active implant devices, or systems that have specific requirements on product function safety; (B) Aviation applications; (C) Aerospace applications or environment; (D) Weapons, and/or (E) Other applications that may cause injuries, deaths or property damages. Since ARTERY products are not intended for the above-mentioned purposes, if purchasers apply ARTERY products to these purposes, purchasers are solely responsible for any consequences or risks caused, even if any written notice is sent to ARTERY by purchasers; in addition, purchasers are solely responsible for the compliance with all statutory and regulatory requirements regarding these uses.

Any inconsistency of the sold ARTERY products with the statement and/or technical features specification described in this document will immediately cause the invalidity of any warranty granted by ARTERY products or services stated in this document by ARTERY, and ARTERY disclaims any responsibility in any form.