

## AT32F415 Security Library Application Note

### Introduction

This application note introduces the security library (sLib) application principle of AT32F415 MCUs, operation methods and example projects.

Applicable products:

Part number	AT32F415
-------------	----------

## Contents

<b>1</b>	<b>Overview .....</b>	<b>7</b>
<b>2</b>	<b>Application principles .....</b>	<b>8</b>
2.1	Application principle of sLib .....	8
2.2	How to enable sLib protection .....	10
2.3	How to disable sLib protection.....	11
2.4	Compile and execute program in sLib .....	11
2.4.1	Setting interrupt vector table as security library not allowed .....	12
2.4.2	Correlation between sLib area and user code area.....	12
2.4.3	Use and compile software floating-point arithmetic library .....	14
<b>3</b>	<b>Example applications of sLib .....</b>	<b>16</b>
3.1	Example application requirements.....	16
3.1.1	Hardware requirements.....	16
3.1.2	Software requirements .....	16
3.2	Overview .....	16
3.3	SLIB protected code: FIR low-pass filter.....	17
3.4	Project_L0: example for solution providers.....	18
3.4.1	Generate execute-only code.....	18
3.4.2	Compile security library address.....	20
3.4.3	Enable sLib protection.....	24
3.4.4	Project_L0 execution process.....	26
3.4.5	Generate header file and symbol definition file.....	28
3.5	Project_L1: example for end users .....	29
3.5.1	Create user application project .....	30
3.5.2	Add symbol definition file to project .....	31
3.5.3	Call functions in SLIB-protected area .....	32
3.5.4	Project_L1 execution process.....	32
3.5.5	SLIB protection in debug mode.....	33
<b>4</b>	<b>Integrate codes and download .....</b>	<b>37</b>
4.1	Program codes separately.....	37

4.2	Integrate and program codes .....	40
5	Revision history.....	43

## List of tables

Table 1. Flash size of AT32F415 MCUs .....	9
Table 2. Document revision history .....	43

## List of figures

Figure 1. Mapping of main Flash memory featured with sLib .....	9
Figure 2. Literal pool example (1) .....	12
Figure 3. Literal pool example (2) .....	12
Figure 4. Example of function in sLib area calling the function in user code area .....	13
Figure 5. Example of user-defined function .....	14
Figure 6. Assembly codes in division functions .....	14
Figure 7. Example application process .....	16
Figure 8. Example application .....	17
Figure 9. FIR low-pass filter .....	17
Figure 10. Enter Option interface in Keil .....	18
Figure 11. Tick Execute-only Code in Keil .....	19
Figure 12. Enter Option interface in IAR .....	19
Figure 13. Set C/C++ options in IAR .....	20
Figure 14. Main Flash memory mapping and RAM partition .....	21
Figure 15. Set Linker option in Keil .....	21
Figure 16. Modify scatter in Keil .....	22
Figure 17. Modify SLIB RAM address in KEIL .....	22
Figure 18. Modify SLIB constant address in KEIL .....	23
Figure 19. SLIB address definition in icd file .....	23
Figure 20. Address assignment in icf file .....	23
Figure 21. Modify IP-Code RAM in icf file .....	24
Figure 22. SLIB constant address in IAR .....	24
Figure 23. Modify SLIB C_LIB in icf file .....	24
Figure 24. Configure ICP Programmer .....	25
Figure 25. Set parameters in Download Form .....	26
Figure 26. Project_L0 execution process .....	27
Figure 27. Set Misc controls in Keil .....	28
Figure 28. Contents of modified fir_filter_symbol.txt .....	28
Figure 29. Set Build Actions in IAR .....	29
Figure 30. Edit steering_file.txt content .....	29
Figure 31. Modified scatter file .....	30
Figure 32. Modified icf file .....	31
Figure 33. Add symbol definition file in Keil .....	31
Figure 34. Modify symbol definition file type to "Object file" .....	31

Figure 35. Add symbol definition file in IAR .....	32
Figure 36. Project_L1 execution process .....	33
Figure 37. Enter Show Disassembly at Address .....	34
Figure 38. Set Show Code at Address .....	34
Figure 39. View codes .....	34
Figure 40. View codes in Memory .....	35
Figure 41. SLIB_DATA start sector in Memory .....	35
Figure 42. SLIB write test .....	35
Figure 43. Write protection error interrupt .....	36
Figure 44. Save SLIB codes .....	37
Figure 45. Generate bin file of SLIB code .....	38
Figure 46. Online MCU programming in ICP .....	38
Figure 47. Offline programming to MCU via AT-Link .....	39
Figure 48. End users program codes to MCU .....	40
Figure 49. Set offline project .....	41
Figure 50. Add project file .....	42

## 1 Overview

As more and more MCU applications require complex algorithms and middleware solutions, it has become an important issue that how to protect IP-Codes (such as core algorithms) developed by software solution providers.

The AT32F415 series MCUs are designed with a security library (sLib) to protect important IPCodes against being changed or read by the end user's program.

This application note details the sLib application principle and operation methods of AT32F415 MCUs.

## 2 Application principles

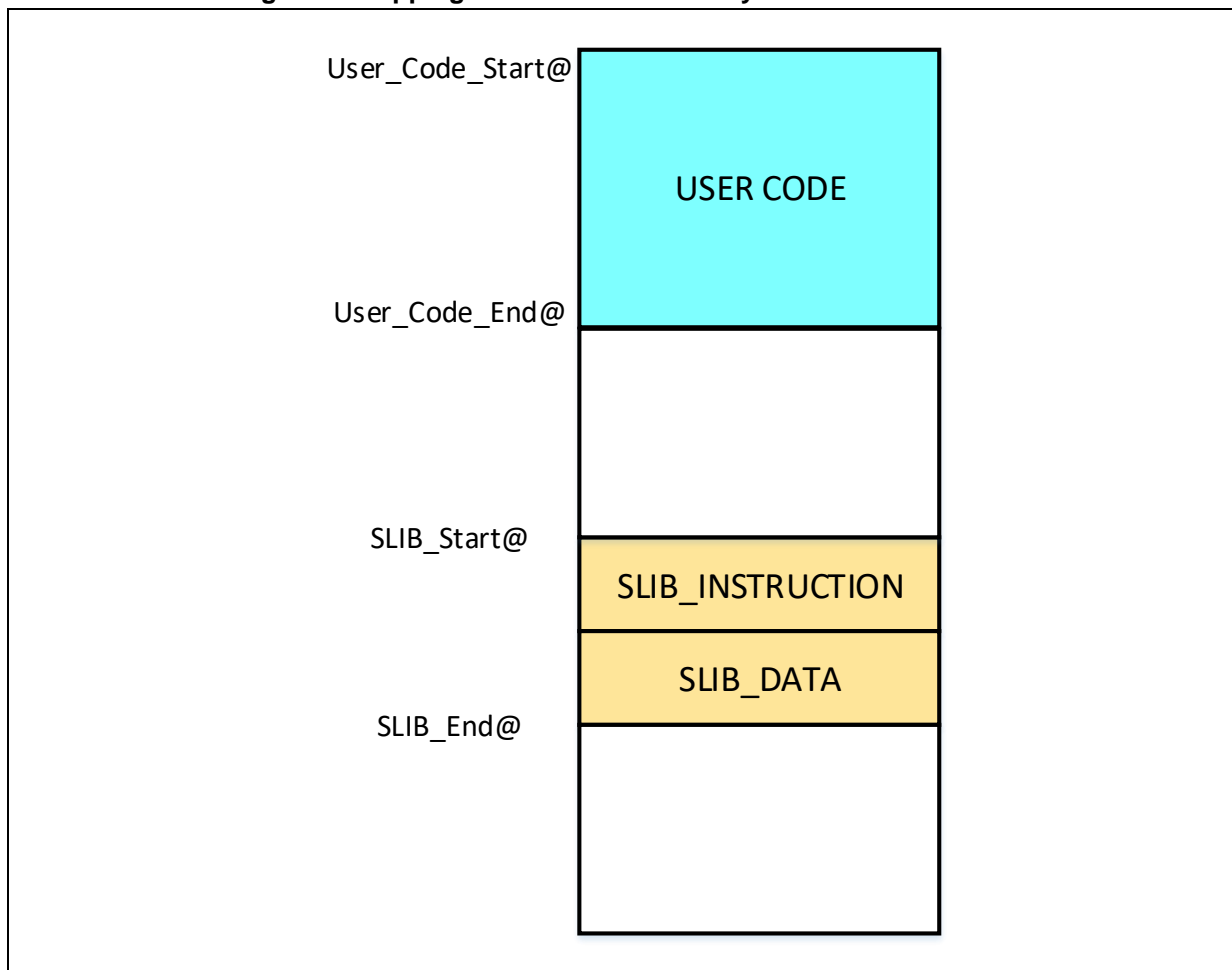
### 2.1 Application principle of sLib

- Security library is a defined area protected by a code in the main memory, so that solution providers can program core algorithm into this area, and the rest of the area can be used for secondary development by end customers.
- Security library includes instruction security library (SLIB\_INSTRUCTION) and data security library (SLIB\_DATA); users can select part of or the whole security library for instruction storage, but using the whole security library for storing data is not supported.
- Program codes in the instruction security library (SLIB\_INSTRUCTION) can only be fetched (can only be executed) by MCU through I-Code bus and cannot be read through D-Code (including ISP/ICP debug mode and programs that boot from internal RAM). When accessing the SLIB\_INSTRUCTION in the manner of reading data, values are all read 0xFF.
- Data in the data security library (SLIB\_DATA) can only be read through D-Code bus and cannot be programmed.
- The program code and data in security library cannot be erased unless the correct code is keyed in. If a wrong code is keyed in, in an attempt of writing or erasing the security library, a warning message will be issued by EPPERR=1 in the FLASH\_STS register.
- The program code and data in security library are not erased when the end users perform a mass erase on the main Flash memory.
- Users can write the previously defined password in the SLIB\_PWD\_CLR register to disable security library protection. When the security library protection is disabled, the chip will perform a mass erase on the main Flash memory (including the contents of security library). Therefore, even if the code defined by the software solution provider is leaked, the program code will not be leaked.

The mapping of main Flash memory featured with sLib is shown in Figure 1. The program codes in security library can be easily called and executed by end users, but cannot be read directly.



Figure 1. Mapping of main Flash memory featured with sLib



The range of sLib is set by sector, and the size of each sector is subject to the specific MCUs.

Table 1 lists the main Flash size, sector size and configurable range of AT32F415 series MCUs.

In addition, once the boot memory extension area is enabled, the entire 18 KB area can be used as security library.

Table 1. Flash size of AT32F415 MCUs

Model	Internal Flash size (byte)	Sector size (byte)	Configurable range
AT32F415x8	64K	1K	Sector 4 ~ 63 (0x08001000 ~ 0x0800FFFF)
AT32F415xB	128K	1K	Sector 4 ~ 127 (0x08001000 ~ 0x0801FFFF)
AT32F415xC	256K	2K	Sector 2 ~ 127 (0x08001000 ~ 0x0803FFFF)

## 2.2 How to enable sLib protection

By default, security library setting register is unreadable and write protected. To enable write access to this register, security library should be unlocked first. Write 0xA35F6D24 to the SLIB\_UNLOCK register, and check the SLIB\_ULKF bit in the SLIB\_MISC\_STS register to verify if it is unlocked successfully, and then write the programmed values into the security library setting register.

The steps to enable security library are as follows:

- 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 the SLIB\_ULKF bit in the SLIB\_MISC\_STS register to verify that it is unlocked successfully;
- Set the sectors to be protected in the SLIB\_SET\_RANGE register, including the address of instruction and data areas;
- Wait until the OBF bit becomes “0”;
- Set a security library password in the SLIB\_SET\_PWD register;
- Wait until the OBF bit becomes “0”;
- Program the code to be saved in security library;
- Perform system reset, and then reload security library setting word;
- Read the SLIB\_STS0/STS1 register to verify the security library settings

### Notes:

- It is allowed to set security library in the main Flash memory only, and refer to [Table 1](#) for the configuration range;
- Set security library in the boot memory (with AP mode enabled), with the configurable range of the whole boot memory;
- It is not permissible to program the main Flash and boot memory as a security library simultaneously;
- The security library code must be programmed by sectors, with its start address aligned with the main Flash memory address;
- The interrupt vector table is in data type and usually placed in the first sector (sector0) of the main Flash memory, which should not be configured as a security library;
- Codes to be protected by the security library should not be placed in the first 4 KB of main Flash memory.

For details of security library setting register, please refer to *AT32F415 Series Reference Manual*.

The program to start security library can be found in the *slib\_enable()* function in *main.c* file of project\_I0. In addition, users can use Artery ICP/ISP tools for configuration.

## 2.3 How to disable sLib protection

The security library protection can be disabled by writing the previously defined password to the SLIB\_PWD\_CLR register. While disabling security library protection, MCU will perform mass erase operation to the main Flash memory (including the contents of security library).

The steps to disable main Flash security library are as follows:

- Check the OBF bit in the FLASH\_STS register to ensure that there is no other ongoing programming operation;
- Write the previously defined password to the SLIB\_PWD\_CLR register;
- Perform system reset, and then reload security library setting word;
- Read the SLIB\_STS0 register to check the security library settings.

## 2.4 Compile and execute program in sLib

As aforementioned, program codes in the instruction security library (SLIB\_INSTRUCTION) can be fetched by MCU via I-Code bus but cannot be read via D-Code bus in the manner of reading data, which means that program codes in SLIB\_INSTRUCTION cannot read the data saved in the same SLIB\_INSTRUCTION. For example, literal pool, branch table or constant compiled from C program code in the SLIB\_INSTRUCTION cannot be read via D-Code bus.

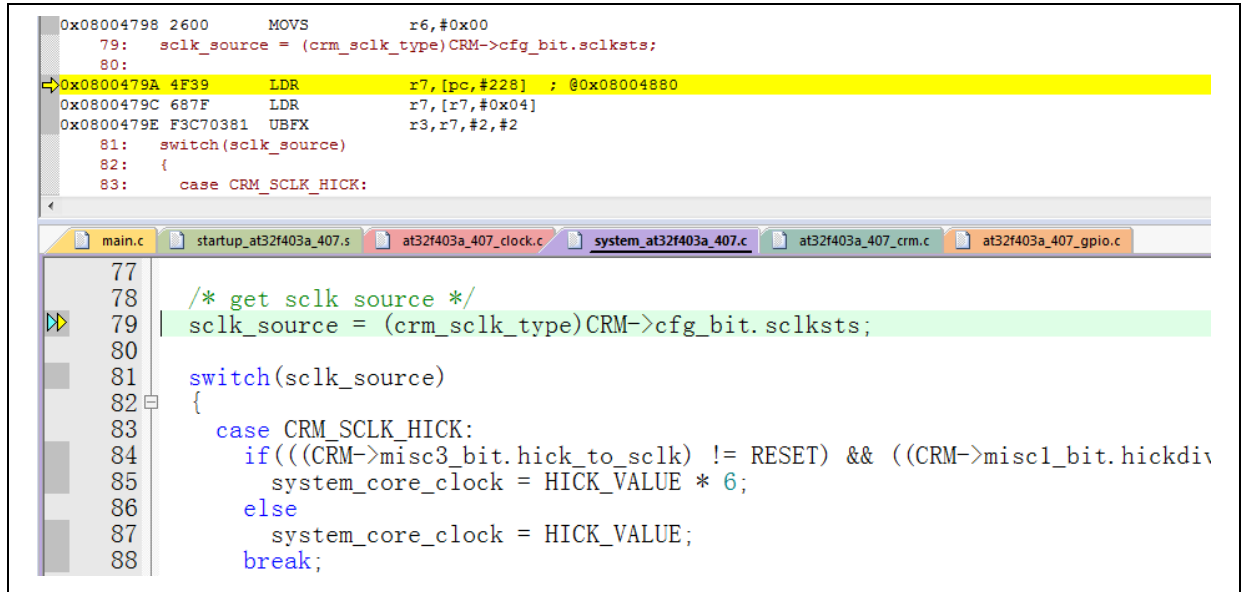
Only instructions rather than data can be placed in the instruction security library. Therefore, when compiling program codes to be placed in the instruction security library, the user must configure the compiler to generate execute-only codes to avoid generating the above mentioned data.

[Figure 2](#) and [Figure 3](#) show the examples of literal pool and branch table.

The “switch()” is a jump instruction in C program, and the “sclk\_source” variable is used to read the CRM\_CFG register. As shown in Figure 2, the compiled assembly code “LDR R7, [PC, #288]” obtains the address of CRM\_CFG register in a PC (program counter) indirect addressing manner, and the address of CRM\_CFG register is saved as a constant in the adjacent instruction area (within the instruction security library); therefore, the data is read when the switch() instruction is executed. An error will occur during execution if there is such program code in the instruction security library.

The example program in Section 3 introduces how to configure compiler settings to avoid error.

Figure 2. Literal pool example (1)

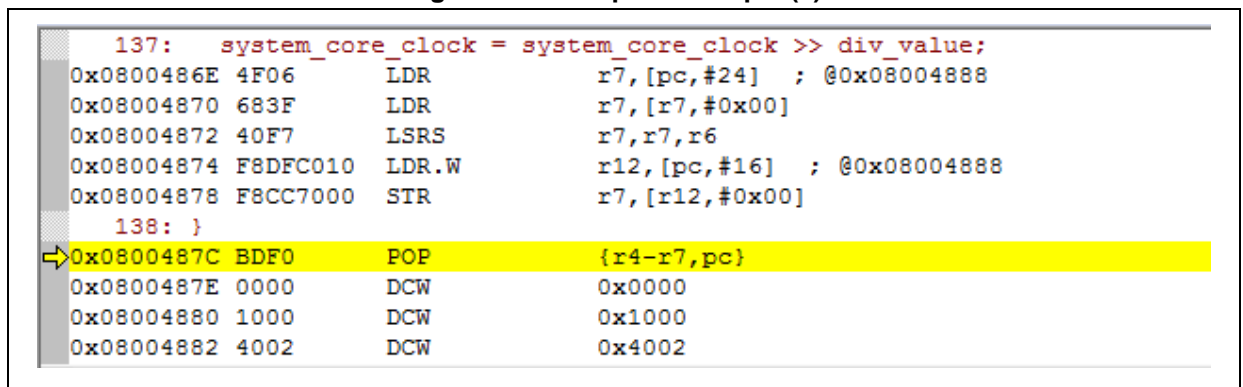


```

0x08004798 2600      MOVs      r6, #0x00
79:      sclk_source = (crm_sclk_type)CRM->cfg_bit.sclksts;
80:
0x0800479A 4F39      LDR       r7, [pc, #228] ; @0x08004880
0x0800479C 687F      LDR       r7, [r7, #0x04]
0x0800479E F3C70381  UBFX     r3, r7, #2, #2
81:      switch(sclk_source)
82:      {
83:      case CRM_SCLK_HICK:
77
78      /* get sclk source */
79      sclk_source = (crm_sclk_type)CRM->cfg_bit.sclksts;
80
81      switch(sclk_source)
82      {
83      case CRM_SCLK_HICK:
84          if(((CRM->misc3_bit.hick_to_sclk) != RESET) && ((CRM->misc1_bit.hickdiv
85              system_core_clock = HICK_VALUE * 6;
86          else
87              system_core_clock = HICK_VALUE;
88          break;

```

Figure 3. Literal pool example (2)



```

137:      system_core_clock = system_core_clock >> div_value;
0x0800486E 4F06      LDR       r7, [pc, #24] ; @0x08004888
0x08004870 683F      LDR       r7, [r7, #0x00]
0x08004872 40F7      LSRS     r7, r7, r6
0x08004874 F8DFC010  LDR.W    r12, [pc, #16] ; @0x08004888
0x08004878 F8CC7000  STR      r7, [r12, #0x00]
138: }
0x0800487C BDF0      POP      {r4-r7, pc}
0x0800487E 0000      DCW      0x0000
0x08004880 1000      DCW      0x1000
0x08004882 4002      DCW      0x4002

```

## 2.4.1 Setting interrupt vector table as security library not allowed

The interrupt vector table contains entry point address of each interrupt handler, which is read by MCU via D-Code bus. Generally, the interrupt vector table is located in the first sector (sector0, starting address: 0x08000000). Therefore, the following rules must be followed when setting the instruction security library:

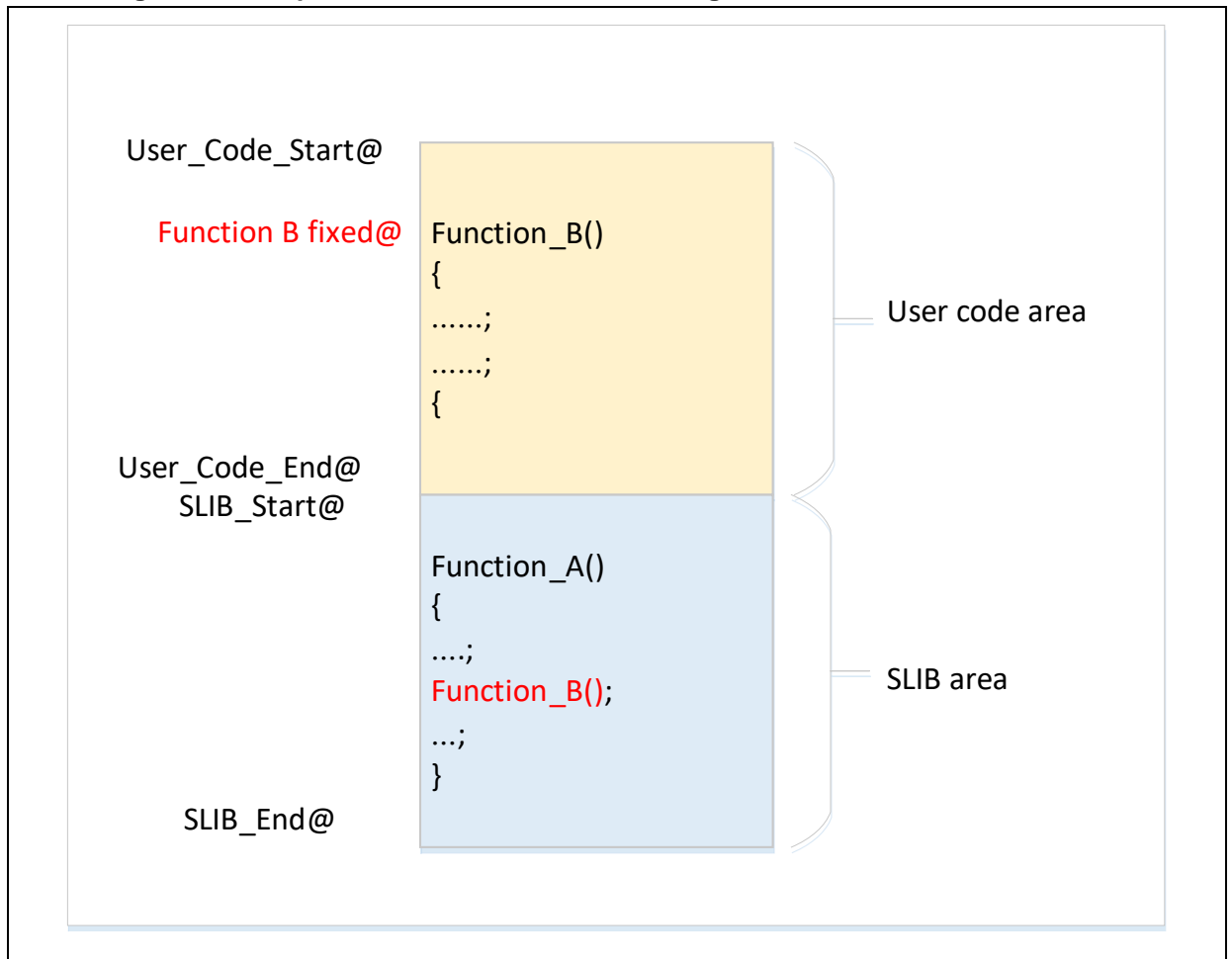
- Do not configure the first sector of the main Flash as security library;
- Program codes to be protected by security library should not be placed in the first sector.

## 2.4.2 Correlation between sLib area and user code area

Program code (IP-code) protected by sLib area can call functions from the function library located in user code area (outside the sLib area). In this case, these function addresses are contained in the IP-Code, allowing PC (program counter) to jump to these functions when IP-Code is executed. Once the sLib area is enabled, function address cannot be changed. At this point, addresses of functions in the user code area must be fixed; otherwise, PC will jump to a wrong address and not work properly. Therefore, when configuring the sLib area, all functions related to IP-Code should be

compiled into the sLib area. Figure 4 gives an example of the protected Function\_A() being called to Function\_B() in the user code area.

**Figure 4. Example of function in sLib area calling the function in user code area**



In addition, the standard function library of C programming language is commonly used, such as `memset()` and `memcpy()` functions. If both IP-Code and user area code call such functions, the above mentioned error may occur.

Recommended solutions:

- 1) Compile into the sLib area (refer to Keil or IAR documents for details about implementation).
- 2) Do not use the standard function library of C programming language in IP-Code. If it is necessary to use in IP-Code, functions to be used must be renamed. Figure 5 shows an example of writing the `my_memset()` function to replace the original `memset()` in IP-Code.

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);
}
```

## 2.4.3 Use and compile software floating-point arithmetic library

Since the AT32F415 series does not have a hardware floating point unit (FPU), the Keil or IAR compiler will use ARM® software floating-point arithmetic library for floating-point operations. The software floating-point arithmetic library functions are compiled codes that cannot be modified, and some of them are in the literal pool format as mentioned before; therefore, these functions cannot be compiled to the SLIB\_CODE but must be placed outside the security library. As shown in Figure 6, the division functions in Keil floating-point arithmetic library have assembly codes in literal pool format.

Figure 6. Assembly codes in division functions

Disassembly			
0x0803C776	F3EF8000	MRS	r0,APSR
0x0803C77A	4770	BX	lr
aeabi_fdiv:			
0x0803C77C	F44F0C7F	MOV	r12,#0xFF0000
0x0803C780	EA1C12D0	ANDS	r2,r12,r0,LSR #7
0x0803C784	BF1E	ITTT	NE
0x0803C786	EA1C13D1	ANDS	r3,r12,r1,LSR #7
0x0803C78A	EA920F0C	TEQ	r2,r12
0x0803C78E	EA930F0C	TEQ	r3,r12
0x0803C792	F0008085	BEQ.W	0x0803C8A0
0x0803C796	EA900F01	TEQ	r0,r1
0x0803C79A	BF48	IT	MI
0x0803C79C	F4427280	ORR	r2,r2,#0x100
0x0803C7A0	F4400C00	ORR	r12,r0,#0x800000
0x0803C7A4	F4410000	ORR	r0,r1,#0x800000
0x0803C7A8	F02C417F	BIC	r1,r12,#0xFF000000
0x0803C7AC	F020407F	BIC	r0,r0,#0xFF000000
0x0803C7B0	B500	PUSH	{lr}
0x0803C7B2	4281	CMP	r1,r0
0x0803C7B4	EBA20203	SUB	r2,r2,r3
0x0803C7B8	F20F1C08	ADR.W	r12,{pc}+0x10C ; 0x0803C8C4
0x0803C7BC	EBA4E50	SUB	lr,r12,r0,LSR #17
0x0803C7C0	F1C00000	RSB	r0,r0,#0x00
0x0803C7C4	F89EE000	LDRB	lr,[lr,#0x00]
0x0803C7C8	EA4F0E4E	LSL	lr,lr,#1
0x0803C7CC	FB00FC0E	MUL	r12,r0,lr

After the sLib is enabled, all contents in the sLib cannot be changed, including the address of floating-point arithmetic library functions called by SLIB\_CODE. Section 3 introduces how to compile the floating-point arithmetic library functions to be used into a fixed area outside the SLIB-CODE, so that programs in the SLIB-CODE can be called properly after the sLib protection is enabled.

For details about Keil floating-point arithmetic library, refer to the ARM DUI0378G *ARM® Compiler v5.06 for μVision® ARM C and C++ Libraries and Floating-Point Support User Guide* under the installation directory.

For details about IAR floating-point arithmetic library, refer to the EWARM\_DevelopmentGuide *IAR C/C++ Development Guide* (PREBUILT RUNTIME LIBRARIES section) under the installation directory.

### 3 Example applications of sLib

This section introduces example applications of sLib and how to complete these applications step by step.

#### 3.1 Example application requirements

##### 3.1.1 Hardware requirements

- AT-START-F415 demo board with AT32F415RCT7 chip
- AT-Link emulator for debugging

##### 3.1.2 Software requirements

- Keil® µvision IDE (µvision V5.18.0.0 is used in this example) or IAR Embedded workbench IDE (IAR V8.22.2 is used in this example)
- Artery ICP/ISP programming tools for enabling and disabling sLib

#### 3.2 Overview

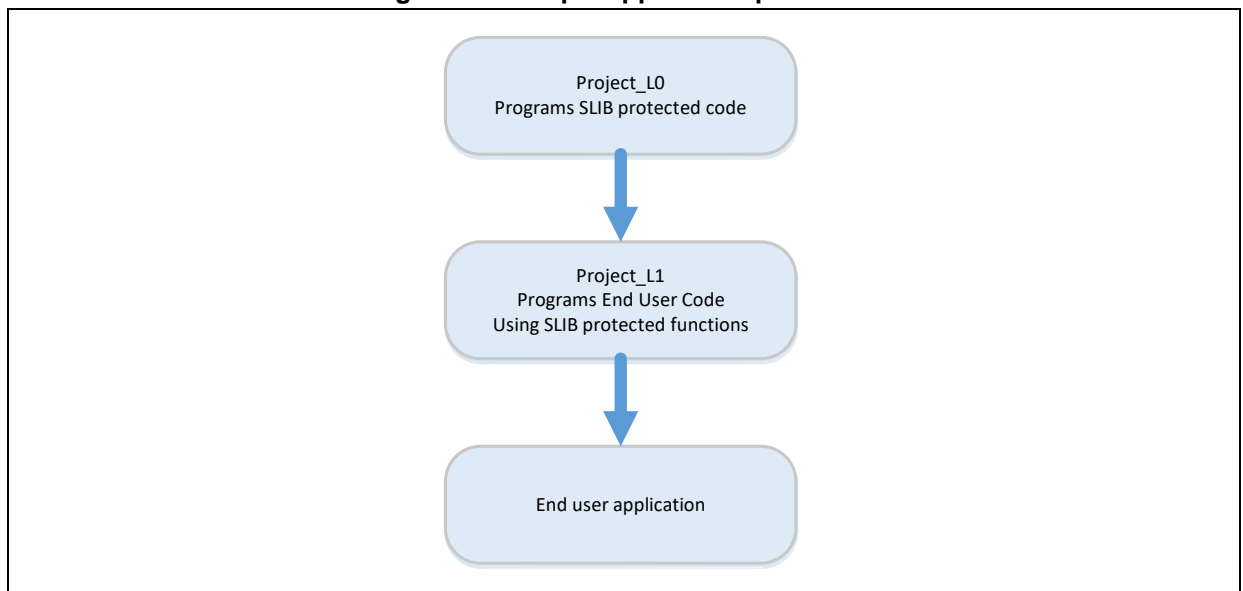
This application note provides two example projects to demonstrate that software developers develop IP-Code for end-user applications.

- Project\_L0: Solution provider develops algorithm and compiles to sLib
- Project\_L1: Apply algorithm by end users

The algorithm completed in Project\_L0 will be pre-downloaded and pre-burned to AT32F415 chip and configured as sLib protected. In addition, the following settings are available for the end-user applications.

- Main Flash memory mapping, showing the area occupied by sLib and the area where users can develop programs;
- Header file that contains algorithm function definitions, allowing end users to call relevant functions;
- Symbol definition file, which contains the actual address of each IP-Code function, so that functions can be called properly by the end-user application.

**Figure 7. Example application process**





Software solution providers can refer to the Project\_L0 to develop algorithm code and refer to Project\_L1 for end-user application.

**Figure 8. Example application**

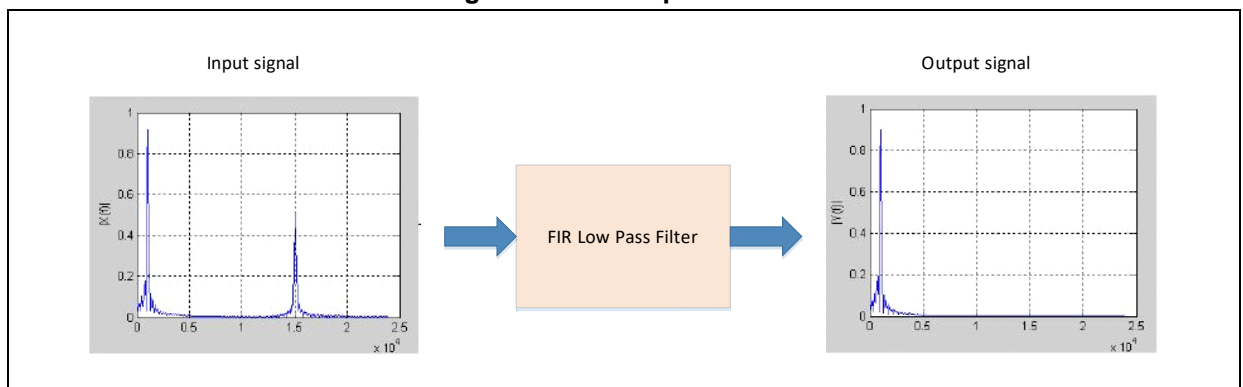


## 3.3 SLIB protected code: FIR low-pass filter

This example uses FIR low-pass filter algorithm provided by CMSIS-DSP library as the sLib protected IP-Code. For details about FIR low-pass filter algorithm, refer to CMSIS-DSP relevant documents. This application note mainly introduces how to configure sLib to protect this algorithm and how it is called by the end-user program code.

The low-pass filter input signal in this example is a combination of two sine waves at frequencies of 1 KHz and 15 KHz, while the low-pass filter cut-off frequency is about 6 KHz. A 15 KHz signal is filtered through the low-pass filter and outputs 1 KHz sine wave. Figure 9 shows the FIR low-pass filter functions.

**Figure 9. FIR low-pass filter**



CMSIS DSP library functions and files to be used are:

- `arm_fir_init_f32()`

It is used for initialization of filter, which is included in “arm\_fir\_init\_f32.c” file.

- `arm_fir_f32()`

It is the main part of filter algorithm, which is included in “arm\_fir\_f32.c” file.

- `FIR_lowpass_filter()`

It is a FIR low-pass filter global function written by using the above two functions for the end user to call, which is included in “fir\_filter.c” file.

- `fir_coefficient.c`

This C file contains coefficients (read-only constants) used by FIR filter functions, and these coefficients are placed in data security library in the example.

In this example, FPU and DSP instructions in the MCU are used for signal processing and floating point arithmetic to realize accurate calculation and correct output signal.

### 3.4 Project\_L0: example for solution providers

The following projects are completed in this level:

- Compile the algorithm-related functions to execute-only code;
- Place the algorithm program code to the main Flash memory sector 2;
- Place the coefficients of filter functions to the main Flash memory sector 4;
- Place the floating-point arithmetic library functions and C library to be called to the main Flash memory sector 6;
- Execute FIR\_lowpass\_filter() in the main program to verify its correctness;
- If correct, set sector 2/3 as the instruction security library and sector 4/5 as the data security library, which can be completed by calling the *slib\_enable()* function in the main program or using Artery ICP Programmer (it is recommended to use ICP tools to complete configurations);
- Generate the header file and symbol definition files that are used by end-user program to call low-pass filter functions.

#### 3.4.1 Generate execute-only code

Each toolchain has specific setting options to prevent the compiler generating literal pools and branch table that can read data while executing instructions, such as “LDR Rn, [PC, #offset]”. Section 2.4 lists examples of literal pools and branch table.

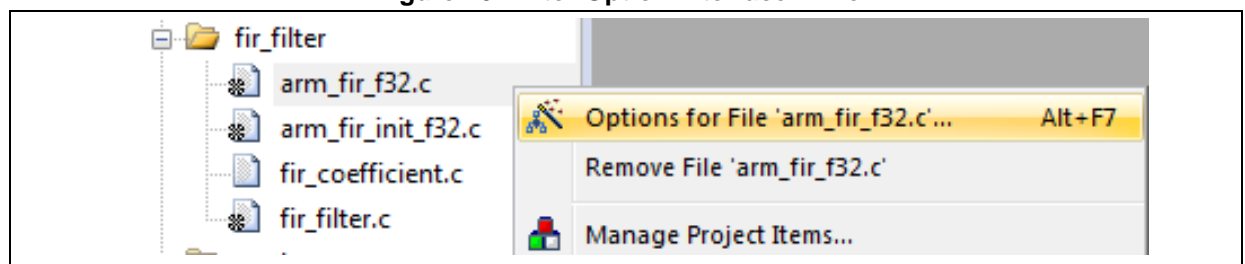
For example, Keil® µvision has the Execute-only Code option, which can be set as follows:

##### Keil® µvision: Set Execute-only Code option

Operate as follows:

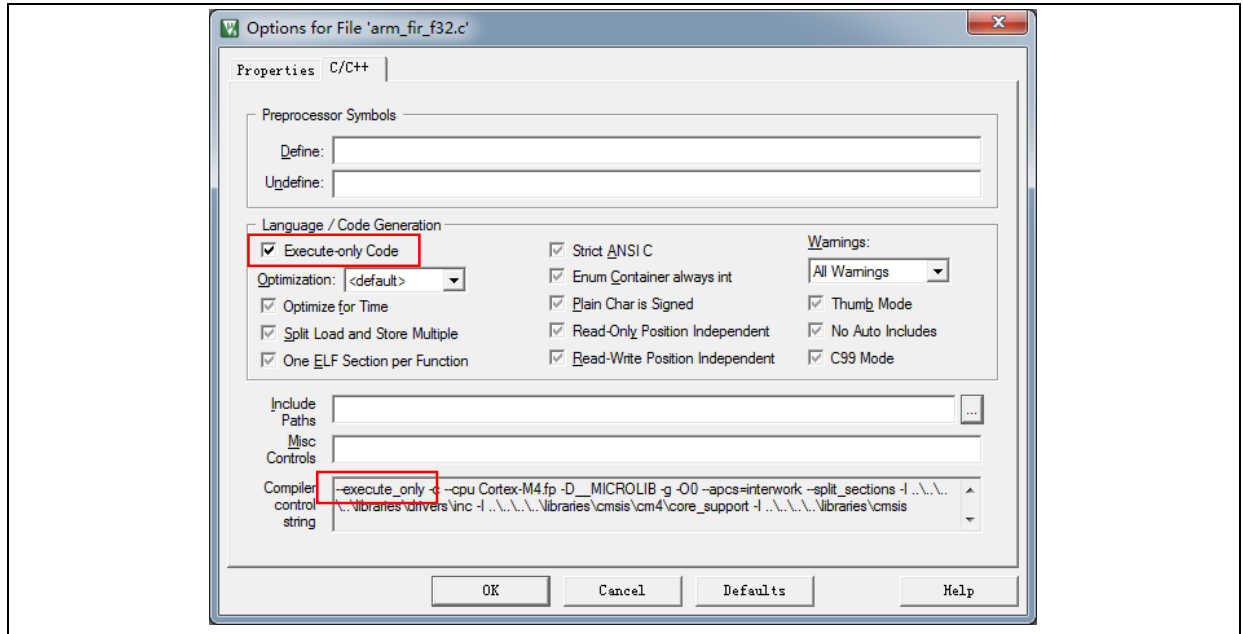
- Select C file group or individual C file (in this example, the C files to be protected are placed in “fir\_filter”);
- Right click and select the corresponding files (for example, the *Option for File ‘arm\_fir\_f32.c’*), as shown in Figure 10;

Figure 10. Enter Option interface in Keil



- Tick “Execute-only Code” in the C/C++ interface, and the “--execute\_only” instruction is added to the compiler control string, as shown in Figure 11;

Figure 11. Tick Execute-only Code in Keil



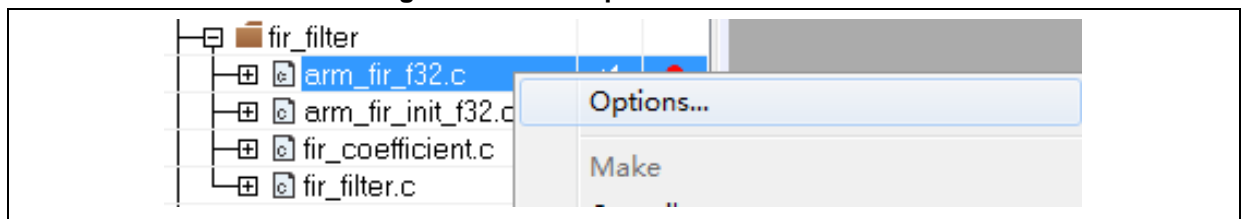
- The *arm\_fir\_f32.c*, *arm\_fir\_init\_f32.c* and *fir\_filter.c* files are in the SLIB\_INSTRUCTION area, and these files need to be set as generating execute-only code.

#### IAR: Set No data read in code memory option

Operate as follows:

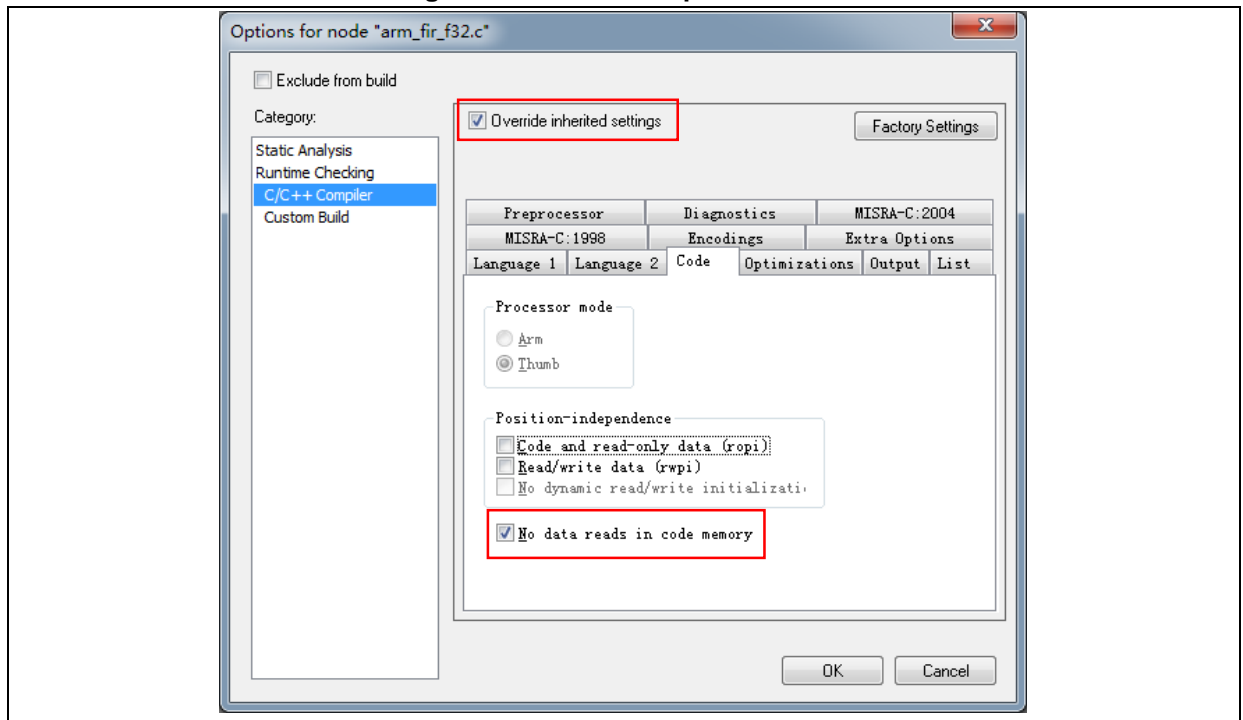
- Select the corresponding file in the *fir\_filter* group; right click and select Option;

Figure 12. Enter Option interface in IAR



- Enter "C/C++" interface and tick "Override inherited settings" and "No data read in code memory", as shown in Figure 13;

Figure 13. Set C/C++ options in IAR

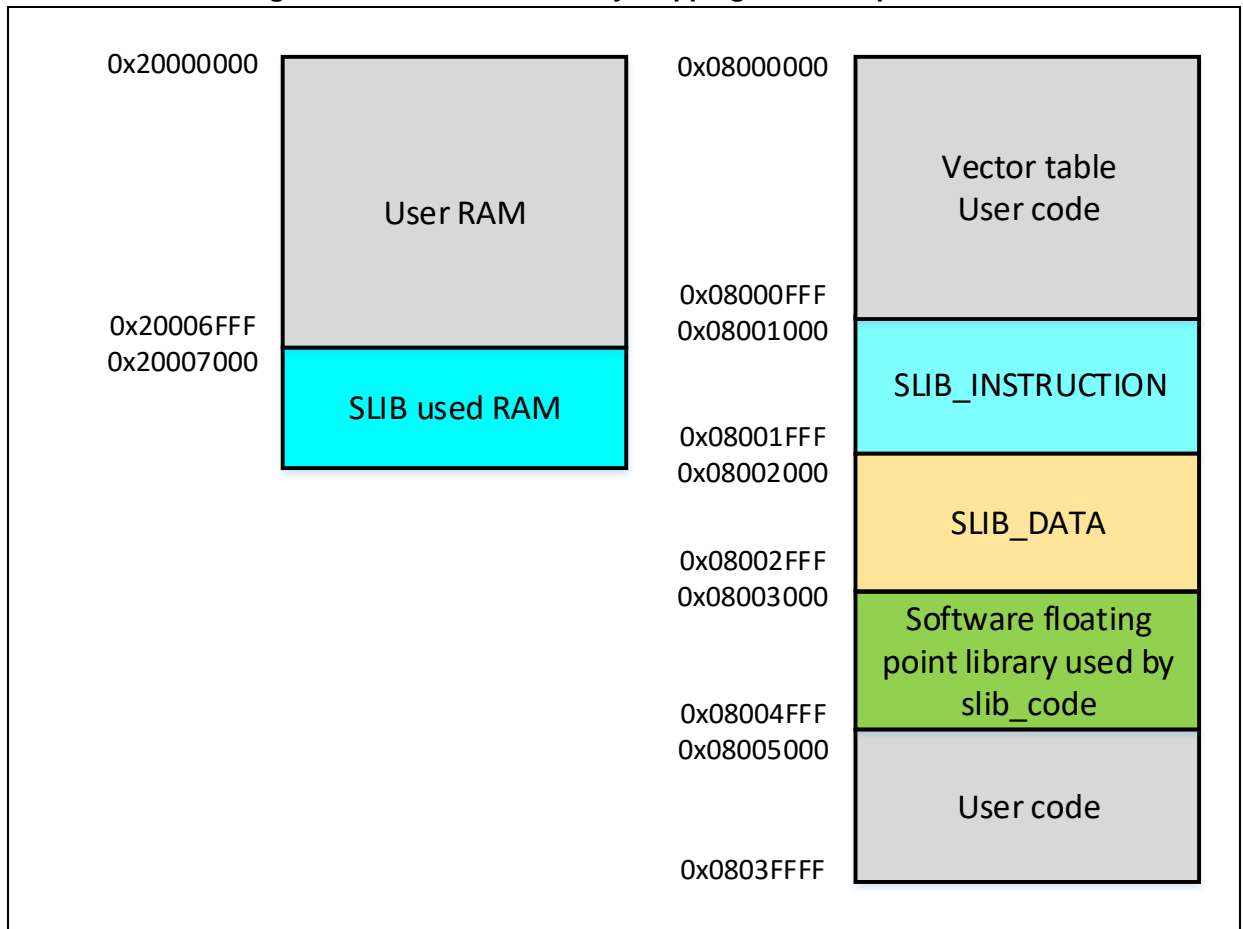


- The *arm\_fir\_f32.c*, *arm\_fir\_init\_f32.c* and *fir\_filter.c* files are in the SLIB\_INSTRUCTION area, and these files need to be set as generating execute-only code

### 3.4.2 Compile security library address

As aforementioned, the first sector (sector0) of the main Flash memory is used to store interrupt vector table. Therefore, the security library is set from sector 2 in this example, with sector 2 and sector 3 being set as instruction security library, sector 4 and sector 5 being set as data security library, and sectors 6-9 being set as the storage area of software floating-point arithmetic library functions and C library. Figure 14 shows the main Flash memory mapping and RAM partition. The main purpose of RAM partitioning is to avoid the same RAM being used by sLib-protected code and end user code.

Figure 14. Main Flash memory mapping and RAM partition

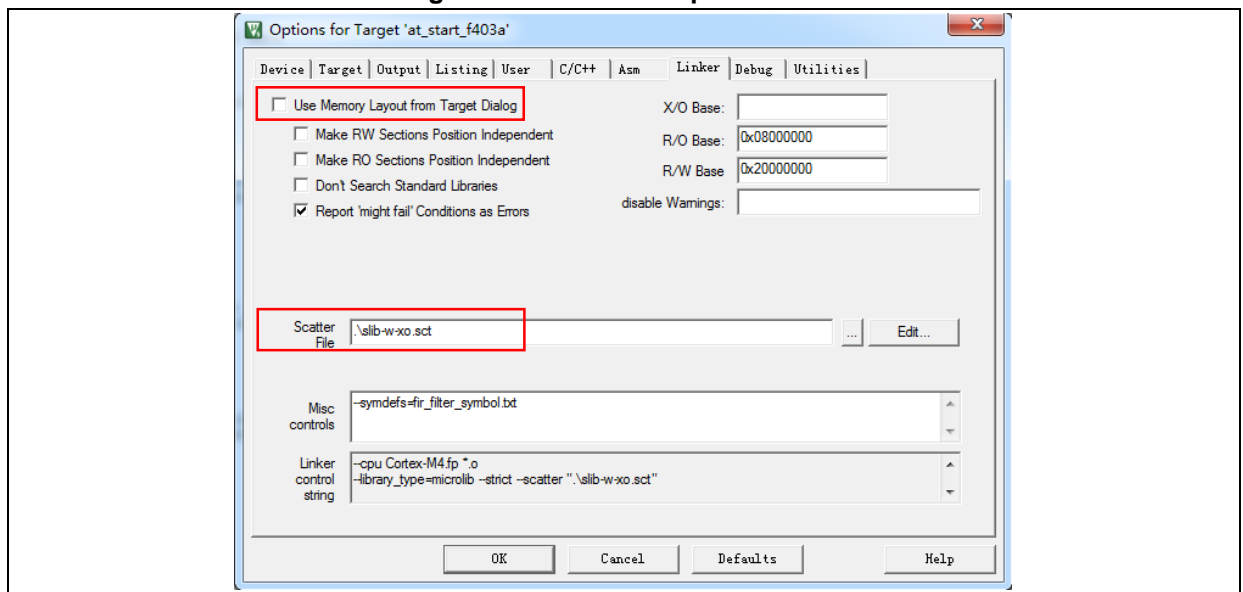


## Keil® µvision: scatter file

Operate as follows:

- Click Project → Options for Target → Linker, untick “Use memory layout from Target Dialog” and click “Edit” to open and modify *slib-w-xo.sct* file, as shown below:

Figure 15. Set Linker option in Keil



- Open scatter file, load the object file of the code to be placed in SLIB\_INSTRUCTION area to

“LR\_SLIB\_INSTRUCTION” (a dedicated loading area that starts from sector 2 and occupies two sectors), and modify the label to “execute-only (+XO)”. Place the area occupied by SLIB\_Data to a dedicated loading area named “LR\_SLIB\_DATA” to avoid the compiler compiling other non-IP-Code functions to the SLIB area. In addition, the LR\_SLIB\_C\_LIB stores the software floating-point arithmetic library and C library to be used. The RW\_IRAM2 assigns the region from 0x20007000 to 0x20007FFF to the algorithm functions to avoid the same RAM region being used by end-user project, causing fault or error in program execution.

**Figure 16. Modify scatter in Keil**

```
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 0x00007000 { ; user RW data
.ANY (+RW +ZI)
}

RW_IRAM2 0x20007000 0x00001000 { ; RAM used for slib code
fir_filter.o (+RW +ZI)
}

LR_SLIB_INST 0x08001000 0x00001000 { ; slib inst area
ER_SLIB_INST 0x08001000 0x00001000 { ; load address = execution address
arm_fir_init_f32.o (+XO)
arm_fir_f32.o (+XO)
fir_filter.o (+XO)
}
}

LR_SLIB_DATA 0x08002000 0x00001000 { ; sLib data area
ER_SLIB_DATA 0x08002000 0x00001000 {
fir_coefficient.o (+RO)
}
}

LR_SLIB_C_LIB 0x08003000 0x00002000 { ; sLib function used c lib area
ER_SLIB_C_LIB 0x08003000 0x00002000 {
*armlib* (+RO)
}
}

LR_IROM2 0x08005000 0x0003B000 { ; user code area
ER_IROM2 0x08005000 0x0003B000 { ; load address = execution address
.ANY (+RO)
}
}
```

Note: Use *\*armlib\** to compile the floating-point arithmetic library and C library to be used into the LR\_SLIB\_C\_LIB area.

- In addition to modifying the scatter file, for the RAM used by IP-Code, users can also use the Keil “\_\_attribute\_\_((at(address)))” descriptor to load variables to 0x20007000, as shown in Figure 17.

**Figure 17. Modify SLIB RAM address in KEIL**

```
#if defined ( __ICCARM__ )
static float32_t firStateF32[BLOCK_SIZE + NUM_TAPS - 1] @ 0x20007000 ;
#elif defined ( __CC_ARM )
static float32_t firStateF32[BLOCK_SIZE + NUM_TAPS - 1] __attribute__((at(0x20007000)));
#endif
```

- The start address of data security library is sector 4 (0x08002000). To compile the constants used by FIR low-pass filter functions to this address, users can modify the scatter file as aforementioned, or use the Keil “\_\_attribute\_\_((at(address)))” descriptor to load the constants to a fixed address, as shown in Figure 18.

Figure 18. Modify SLIB constant address in KEIL

```
#if defined ( __ICCARM__ )
const float32_t firCoeffs32[NUM_TAPS] @ 0x08002000 = {
#elif defined ( __CC_ARM )
const float32_t firCoeffs32[NUM_TAPS] __attribute__((at(0x08002000))) = {
#endif
-0.0018225230f, -0.0015879294f, +0.0000000000f, +0.0036977508f, +0.0080754303f,
-0.0341458607f, -0.0333591565f, +0.0000000000f, +0.0676308395f, +0.1522061835f,
+0.1522061835f, +0.0676308395f, +0.0000000000f, -0.0333591565f, -0.0341458607f,
+0.0080754303f, +0.0036977508f, +0.0000000000f, -0.0015879294f, -0.0018225230f
};
```

## IAR: ICF file

Operate as follows:

- Open the *icf* file in \project\_I0\IAR\_V8.2\, and add new SLIB loading areas as shown in Figure 19. The SLIB\_RAM region reserves the corresponding RAM for the algorithm functions.

Figure 19. SLIB address definition in icf file

```
/* SLIB inst area */
define symbol __ICFEDIT_region_SLIB_INST_start__ = 0x08001000;
define symbol __ICFEDIT_region_SLIB_INST_end__ = 0x08001FFF;

/* SLIB data area */
define symbol __ICFEDIT_region_SLIB_DATA_start__ = 0x08002000;
define symbol __ICFEDIT_region_SLIB_DATA_end__ = 0x08002FFF;

/* SLIB used C LIB area */
define symbol __ICFEDIT_region_SLIB_USED_C_LIB_start__ = 0x08003000;
define symbol __ICFEDIT_region_SLIB_USED_C_LIB_end__ = 0x08004FFF;

define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__ = 0x20007FFF;

/* SLIB RAM region */
define symbol __ICFEDIT_region_SLIB_RAM_start__ = 0x20007000;
define symbol __ICFEDIT_region_SLIB_RAM_end__ = 0x20007FFF;
```

- In the *icf* file, the area occupied by SLIB is reserved to avoid the compiler compiling other non-IP-Code functions to the SLIB area, and the RAM region used by IP-Code is reserved.

Figure 20. Address assignment in icf file

```
define region ROM_region = mem:[from __ICFEDIT_region_ROM_start__ to __ICFEDIT_region_ROM_end__]
- mem:[from __ICFEDIT_region_SLIB_INST_start__ to __ICFEDIT_region_SLIB_INST_end__]
- mem:[from __ICFEDIT_region_SLIB_DATA_start__ to __ICFEDIT_region_SLIB_DATA_end__]
- mem:[from __ICFEDIT_region_SLIB_USED_C_LIB_start__ to __ICFEDIT_region_SLIB_USED_C_LIB_end__];

define region SLIB_INST_region = mem:[from __ICFEDIT_region_SLIB_INST_start__ to __ICFEDIT_region_SLIB_INST_end__];
define region SLIB_DATA_region = mem:[from __ICFEDIT_region_SLIB_DATA_start__ to __ICFEDIT_region_SLIB_DATA_end__];
define region SLIB_USED_C_LIB_region = mem:[from __ICFEDIT_region_SLIB_USED_C_LIB_start__ to __ICFEDIT_region_SLIB_USED_C_LIB_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__];
define region SLIB_RAM_region = mem:[from __ICFEDIT_region_SLIB_RAM_start__ to __ICFEDIT_region_SLIB_RAM_end__];
```

- For the RAM used by IP-Code, users can use the IAR @ descriptor to load variables to a fixed address or modify the *icf* file, as shown in Figure 21.

Figure 21. Modify IP-Code RAM in icf file

```
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 of data security library is sector 4 (0x08002000). To compile the constants used by FIR low-pass filter functions to this address, users can modify the *icf* file as aforementioned, or use the IAR @ descriptor to load the constants to a fixed address, as shown in Figure 22.

Figure 22. SLIB constant address in IAR

```
#if defined ( __ICCARM__ )
const float32_t firCoeffs32[NUM_TAPS] @ 0x08002000 = {
#elif defined ( __CC_ARM )
const float32_t firCoeffs32[NUM_TAPS] __attribute__((at(0x08002000))) = {
#endif
    -0.0018225230f, -0.0015879294f, +0.0000000000f, +0.0036977508f, +0.0080754303f,
    -0.0341458607f, -0.0333591565f, +0.0000000000f, +0.0676308395f, +0.1522061835f,
    +0.1522061835f, +0.0676308395f, +0.0000000000f, -0.0333591565f, -0.0341458607f,
    +0.0080754303f, +0.0036977508f, +0.0000000000f, -0.0015879294f, -0.0018225230f
};
```

- The start address of C\_LIB fixed area is sector 6 (0x08003000). To compile the *m7Mx\_tl.a* (floating-point arithmetic library) and *rt7Mx\_tl.a* (C library) to be used by FIR low-pass filter functions to this address, modify the *icf* file, as shown in Figure 23.

Figure 23. Modify SLIB C\_LIB in icf file

```
place in SLIB_INST_region { ro object arm_fir_f32.o,
                             ro object arm_fir_init_f32.o,
                             ro object fir_filter.o};

place in SLIB_DATA_region { ro object fir_coefficient.o };

place in SLIB_USED_C_LIB_region { ro object m7Mx_tl.a,
                                   ro object rt7Mx_tl.a };
```

### 3.4.3 Enable sLib protection

There are two methods to enable sLib protection:

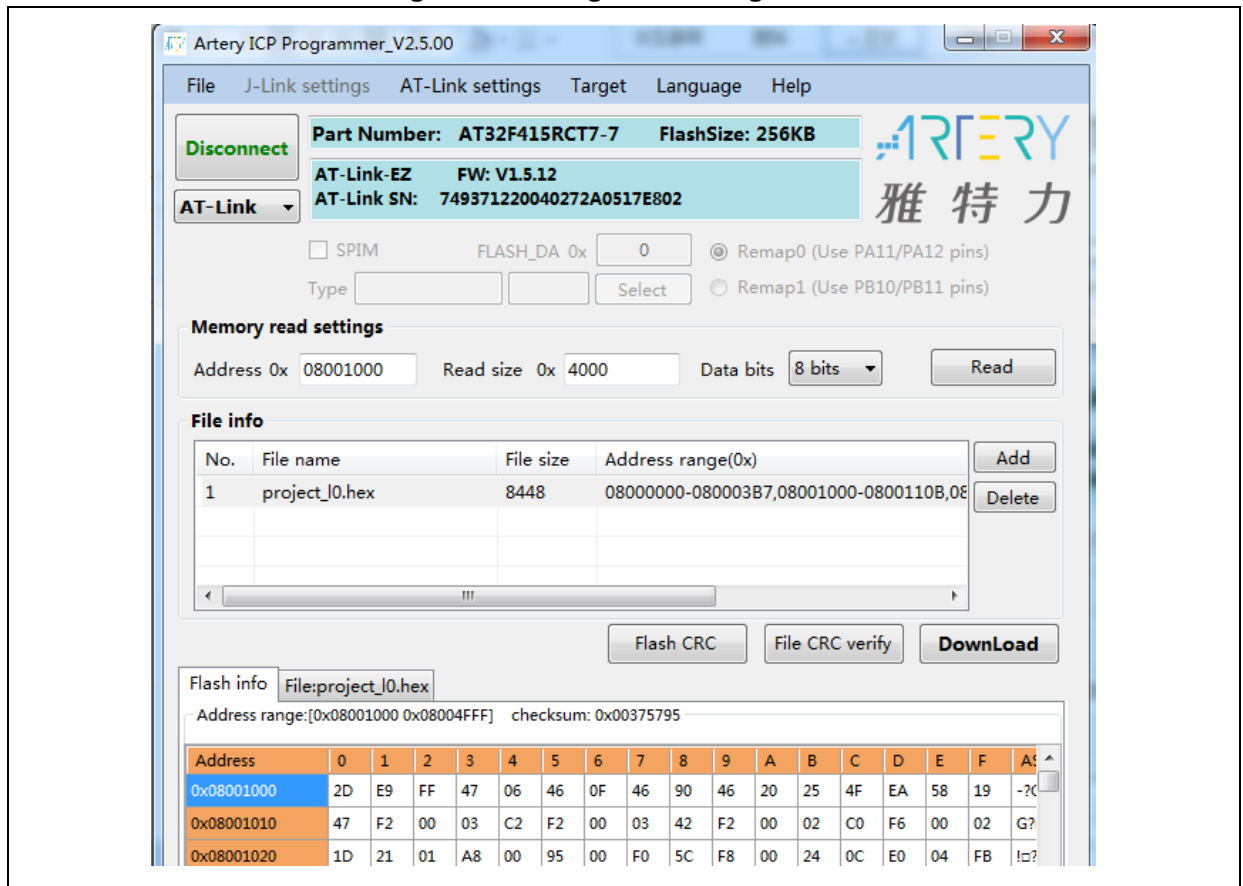
#### (1) Use Artery ICP Programmer (recommended)

It is recommended to use the ICP Programmer as follows:

- Connect AT-Link emulator to AT-START-F415 board and power on;
- Open ICP Programmer, select AT-Link for connection, and add the HEX or BIN file generated by Project\_L0, as shown in Figure 24.



Figure 24. Configure ICP Programmer



- Click “Download” and the “Download Form” pops up, which shows SLIB status and relevant parameters. Set sector 2 as the start sector, sector 4 as the data start sector and sector 5 as the end sector; set the enable password as “0x55665566” (user-defined), tick “Enable SLIB”, and then click “Start Download” to complete programming and enable SLIB, as shown in Figure 25.

Figure 25. Set parameters in Download Form

For details about ICP Programmer, refer to ICP Programmer User Manual.

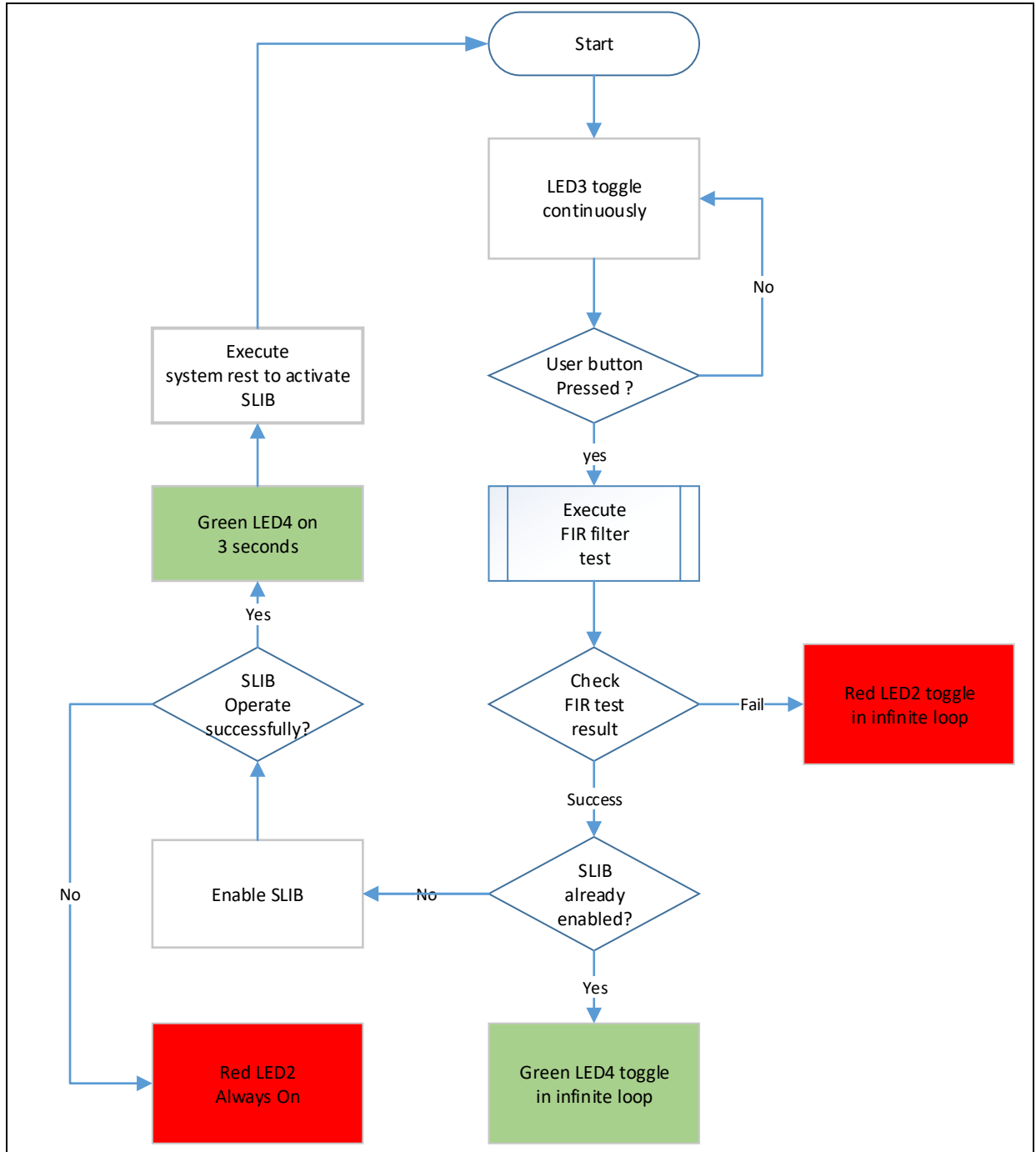
## (2) Use *slib\_enable()* function in main.c

After the *slib\_enable()* function is verified correct by low-pass filter function and then executed, the sLib protection can be enabled. To execute this function, enable the “#define USE\_SLIB\_FUNCTION” in main.c.

## 3.4.4 Project\_L0 execution process

In this example, FIR low-pass filter calculates the input signal (testInput\_f32\_1kHz\_15kHz) mixed with 1 KHz and 15 KHz sine waves, and the output 1 KHz sine wave data is saved in testOutput, which will be compared with the data calculated by MATLAB and saved in refOutput. If the error value is smaller than expected (SNR larger than the preset threshold), the green LED on the board blinks; otherwise, the red LED blinks. Figure 26 shows the Project\_L0 execution process.

Figure 26. Project\_L0 execution process



Go through the following steps to execute this example program:

- (1) Use Keil® µvision to open the Project\_L0 under \utilities\AT32F415\_slib\_demo\project\_l0\mdk\_v5\, and then compile;
- (2) Before downloading the code, check whether the chip on AT-START-F415 board is SLIB-protected or write/read-protected (FAP/EPP). If it is protected, use ICP programmer to disable protection and then download the code;
- (3) After successful download, start to execute the code, and the on-board LED3 keeps blinking rapidly;
- (4) Press the on-board USER button to perform operation of low-pass filter;
- (5) Compare the computation result. If it is correct, the green LED4 keeps blinking; otherwise, the

red LED2 keeps blinking;

- (6) After obtaining the correct result, if the USE\_SLIB\_FUNCTION in main.c is defined and the SLIB is not enabled, the slib\_enable() function will be executed to set SLIB. If SLIB setting fails, the red LED2 will be always ON; if SLIB setting succeeds, the green LED4 will be ON for about three seconds and then perform system reset to enable SLIB; then, go to step (3).

### 3.4.5 Generate header file and symbol definition file

The header file and symbol definition file are used when the Project\_L1 calls FIR low-pass filter functions, which is the fir\_filter.h file in main.c in this example.

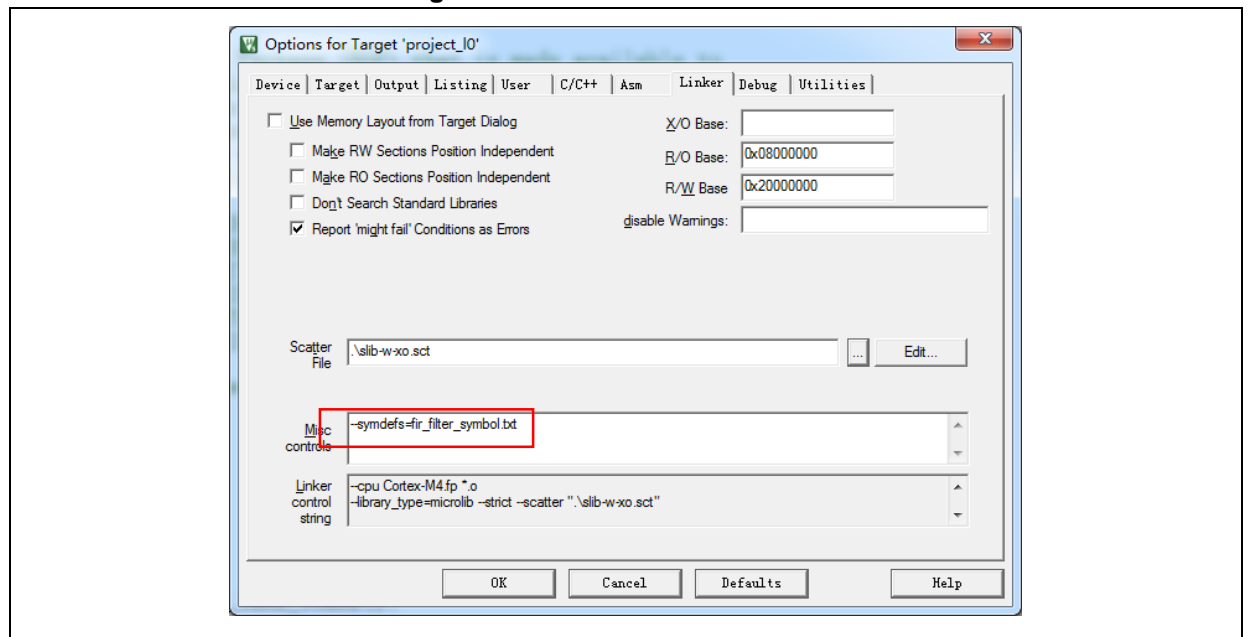
The generation of symbol definition file is related to the specific toolchain being used.

#### Use Keil® µvision to generate symbol definition file

Operate as follows:

- Enter Options for Target → Linker interface;
- Add “--symdefs=fir\_filter\_symbol.txt” command in the Misc controls, as shown in Figure 27;

Figure 27. Set Misc controls in Keil



- After compiling the project, a symbol definition file named “fir\_filter\_symbol.txt” is generated under “project\_I0\mdk\_v5\Objects”;
- This symbol definition file contains all symbol definitions of the project, and it needs to be modified to only remain the definitions of low-pass filter functions to be called by end users. The modified *fir\_filter\_symbol.txt* is shown in Figure 28;

Figure 28. Contents of modified fir\_filter\_symbol.txt

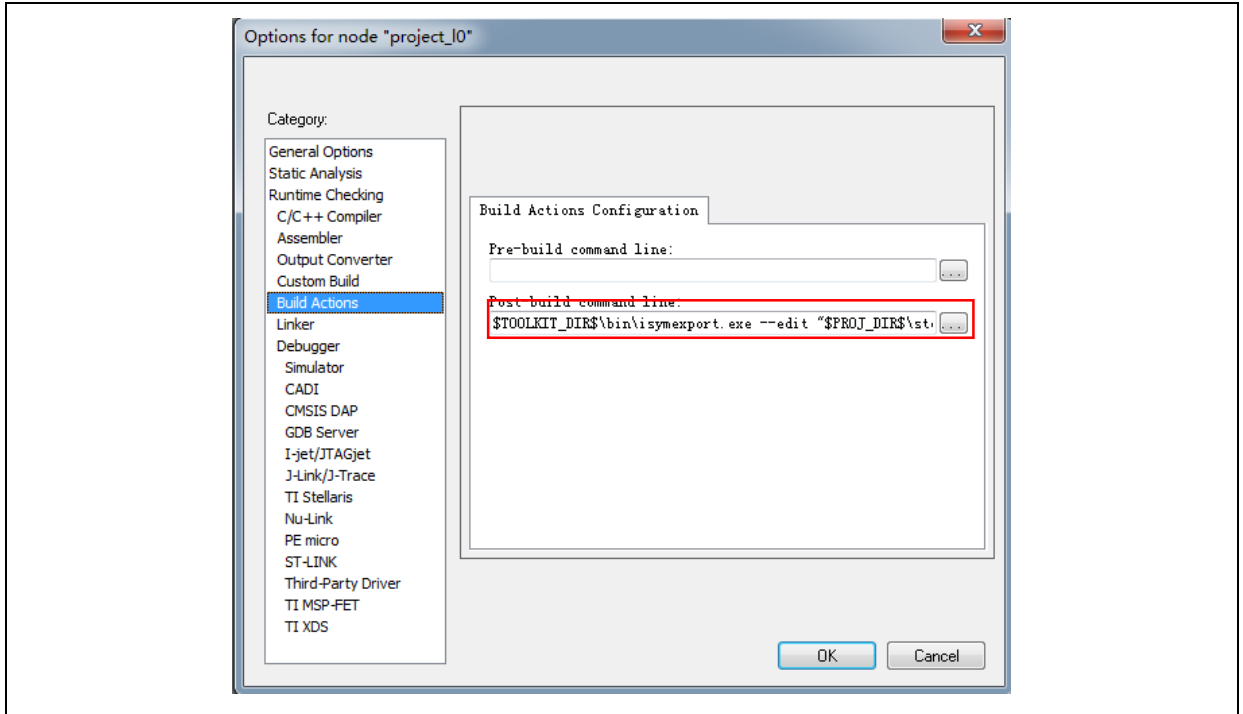
```
0x08001001 T FIR_lowpass_filter
```

## Use IAR to generate symbol definition file

Operate as follows:

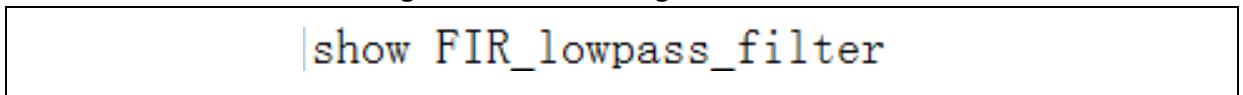
- Select Project→Option→Build Actions

**Figure 29. Set Build Actions in IAR**



- Input the following commands to 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* is the symbol definition file to be generated, and the *steering\_file.txt* is saved under project\_I0\iar\_v8.2, which is used to select function symbols to be generated. Users can manually edit according to the contents called by sLib. As shown in Figure 28, the "show" is the command used to select functions.

**Figure 30. Edit steering\_file.txt content**



## 3.5 Project\_L1: example for end users

Project\_L1 uses the FIR low-pass filter function that is debugged in Project\_L0, programmed to AT32F415 MCU main Flash memory and SLIB-protected. According to the header file, symbol definition file and the main Flash memory mapping of Project\_L0, end users can complete the followings for Project\_L1:

- Create an application project;
- Add the header file and symbol definition file provided by Project\_L0 to the project;
- Call the FIR low-pass filter function;

- Develop and debug user's program.

**Note:**

Project\_L1 must use the same toolchain and the same version of the compiler as that of Project\_L0; otherwise, incompatibility problem may occur and the code provided by Project\_L0 cannot be used properly. For example, Project\_L0 uses Keil® µvision V5.18.0.0; therefore, Project\_L1 need to use the same version.

### 3.5.1 Create user application project

The security library enabled in Project\_L0 occupies some specific main Flash memory sectors; therefore, the address for Project\_L1 code storage should be compiled according to the main Flash memory mapping of Project\_L0, as shown in Figure 14. In the main Flash memory, sector 2 to sector 9 are occupied by the security library and C\_LIB fixed area, and end users need to isolate this region by using linker control file to avoid code being compiled to this region.

**Keil® µvision: scatter file**

Refer to the *end\_user\_code.sct* file under project\_l1\mdk\_v5\, and divide the main Flash memory into two regions, and the middle part is the SLIB-protected area. In addition, the area used by sLib needs to be reserved in the RAM area, as shown in Figure 31.

**Figure 31. 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 (+R0)
  }
  RW_IRAM1 0x20000000 0x00007000 { ; RW data
    .ANY (+RW +ZI)
  }

; 0x20007000 ~ 0x20007FFF RAM reserved for SLIB code
}

; 0x08001000 ~ 0x08002FFF is SLIB area
; 0x08003000 ~ 0x08004FFF is used c lib
LR_IROM2 0x08005000 0x0003B000 { ; load region size_region
  ER_IROM2 0x08005000 0x0003B000 { ; load address = execution address
    .ANY (+R0)
  }
}
```

**IAR: ICF file**

Refer to the *enduser.icf* file under project\_l1\iar\_V8.2\, as shown in Figure 32.

Figure 32. Modified icf file

```

/* Reserved SLIB area */
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__];

```

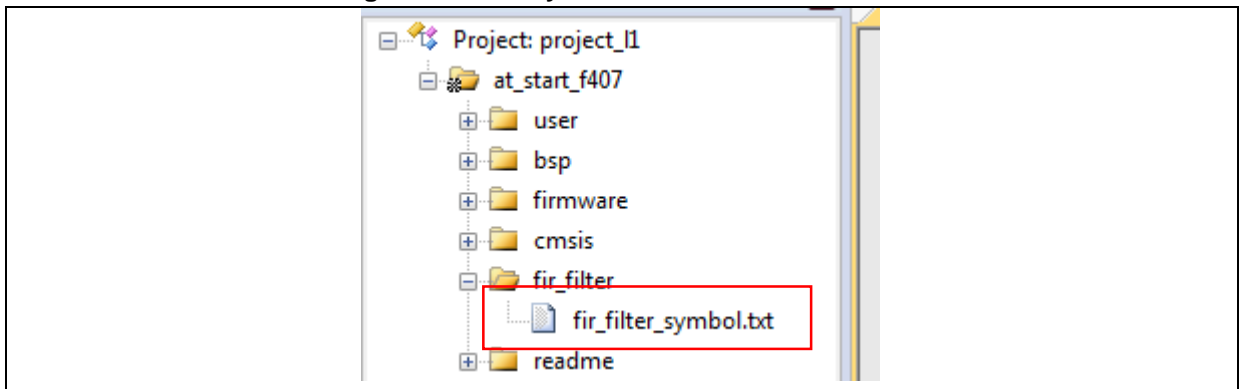
### 3.5.2 Add symbol definition file to project

The symbol definition file *fir\_filter\_symbol.txt* generated in Project\_L0 must be added to Project\_L1, so that it can be correctly compiled and linked to the SLIB-protected area code.

#### Add symbol definition file in Keil® uvision

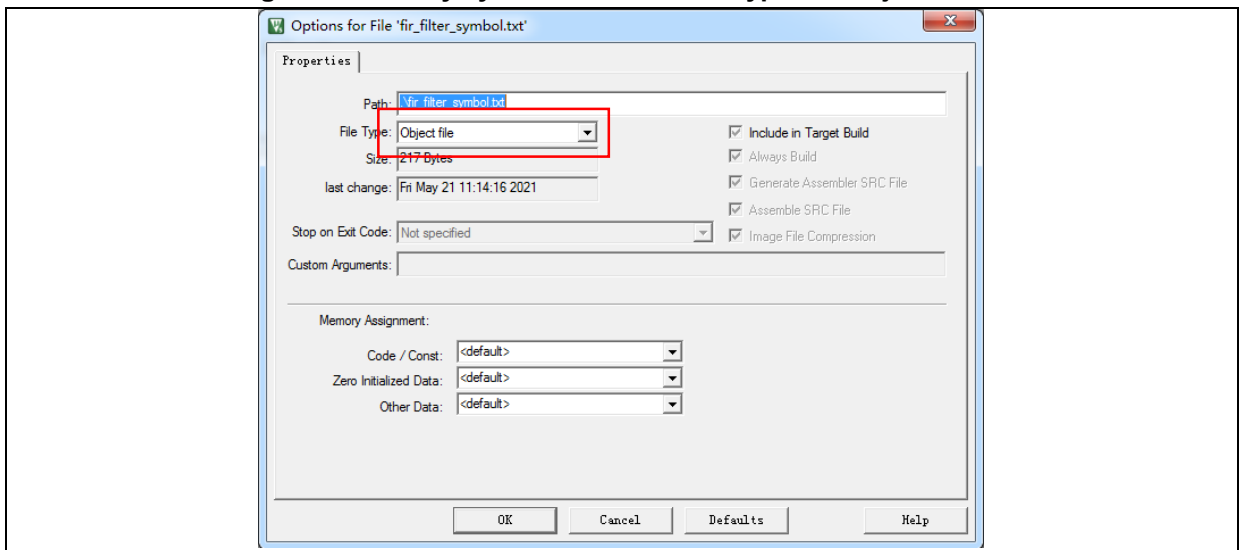
Add *fir\_filter\_symbol.txt* to the project, as shown in Figure 33.

Figure 33. Add symbol definition file in Keil



Add this file to *fir\_filter*, and then modify its file type from “text” to “Object”, as shown in Figure 34.

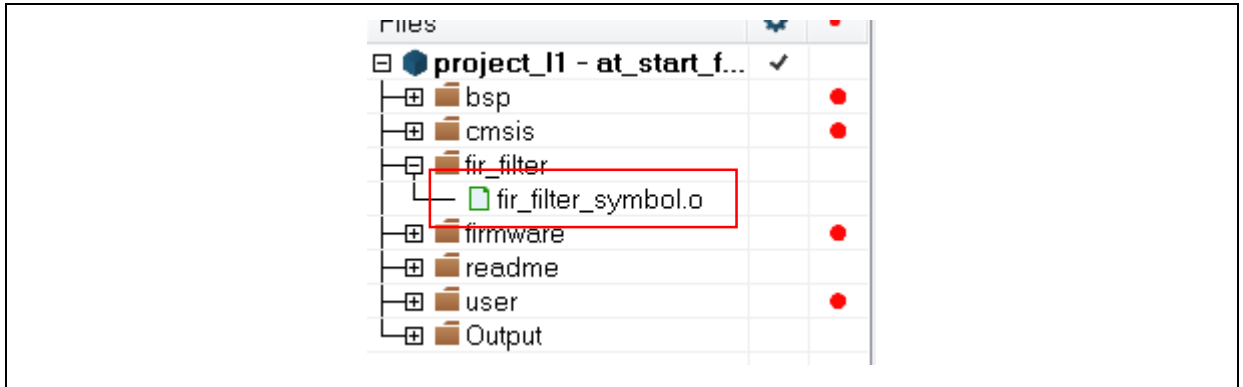
Figure 34. Modify symbol definition file type to “Object file”



#### Add symbol definition file in IAR

Add the *fir\_filter\_symbol.o* (Object) file to *fir\_filter*, as shown in Figure 35.

Figure 35. Add symbol definition file in IAR



### 3.5.3 Call functions in SLIB-protected area

When the *filter.h* header file is referred in *main.c* and the symbol definition file is added to the project, the low-pass filter function in the protection area can be called, as shown below:

```
FIR_lowpass_filter(inputF32, outputF32, TEST_LENGTH_SAMPLES);
```

Where,

- *inputF32*: pointer containing input signal data table;
- *outputF32*: pointer storing output signal data table;
- *TEST\_LENGTH\_SAMPLES*: the number of signal samples to be processed.

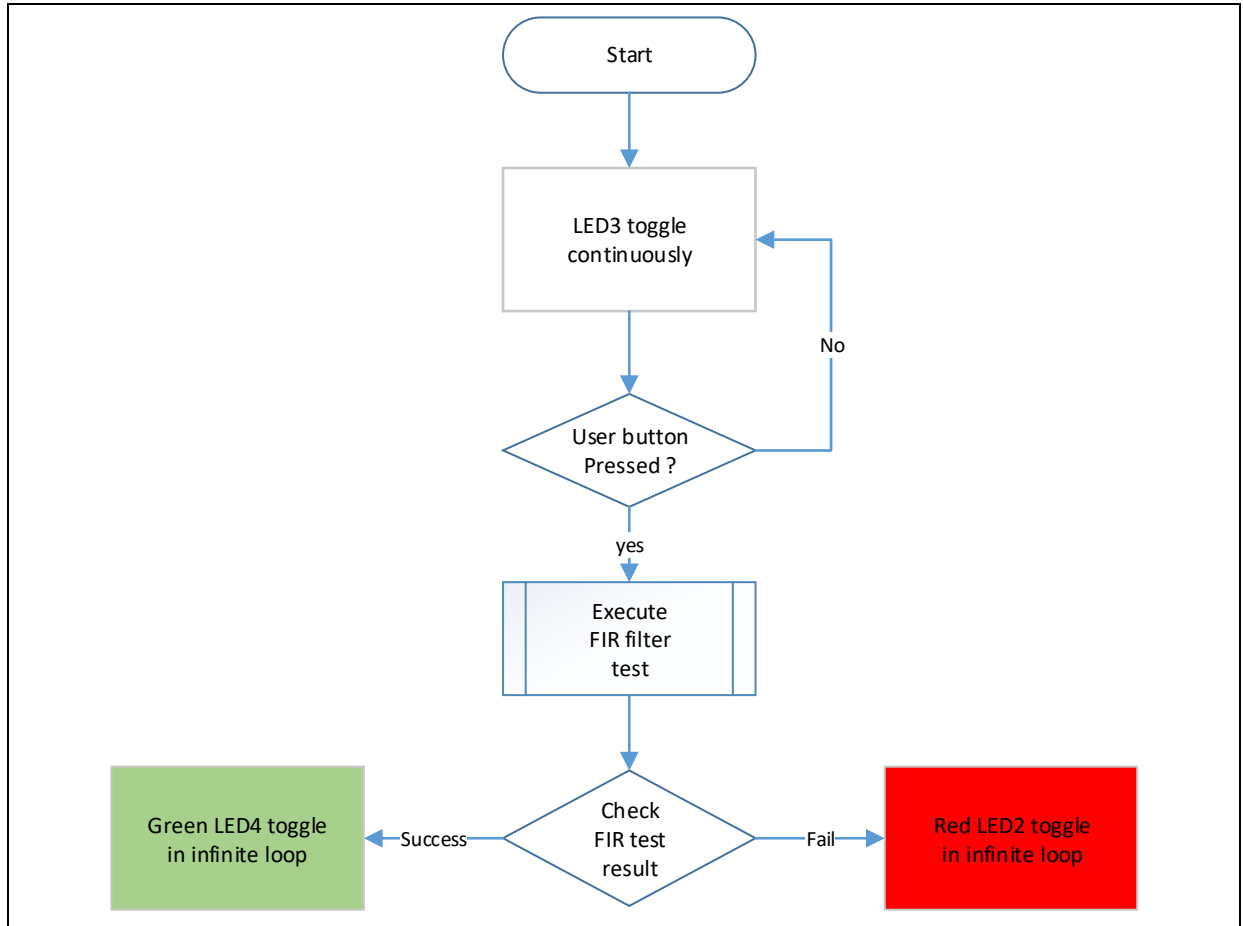
### 3.5.4 Project\_L1 execution process

Figure 36 shows the execution process of Project\_L1:

- Start execution, and LED3 will keep blinking;
- Press the USER button on AT-START board, and the *FIR\_lowpass\_filter()* starts operation;
- If the result is correct, the green LED4 will keep blinking; otherwise, the red LED2 will keep blinking.



Figure 36. Project\_L1 execution process

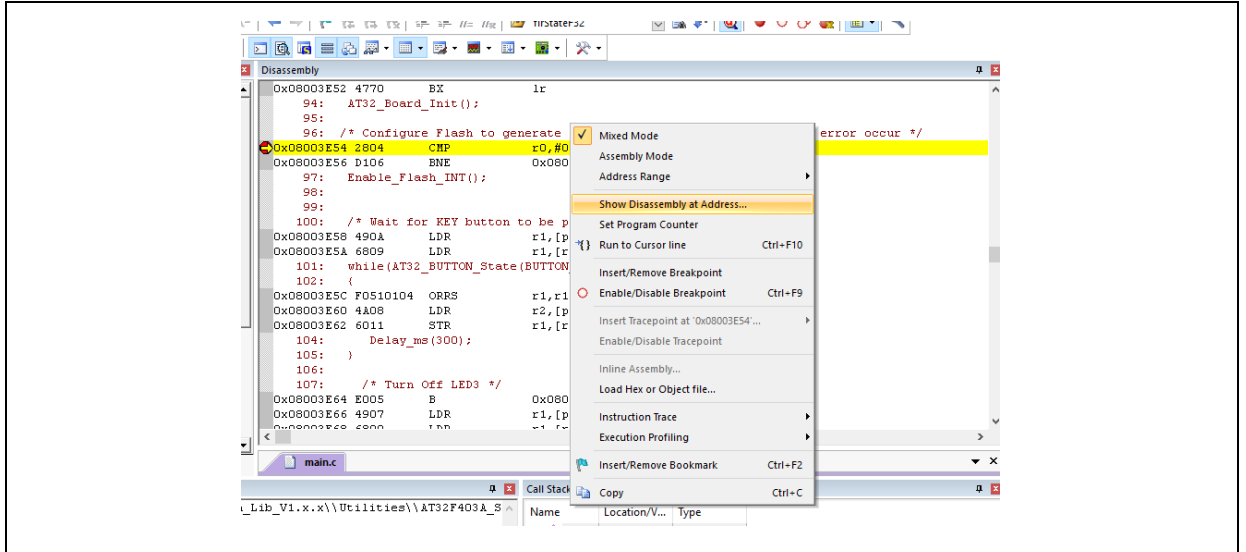


## 3.5.5 SLIB protection in debug mode

Development tools are used by end users to debug codes when developing applications. This section takes Keil® µvision as an example to introduce how to protect codes in the SLIB-protected area from being read as data in debug mode.

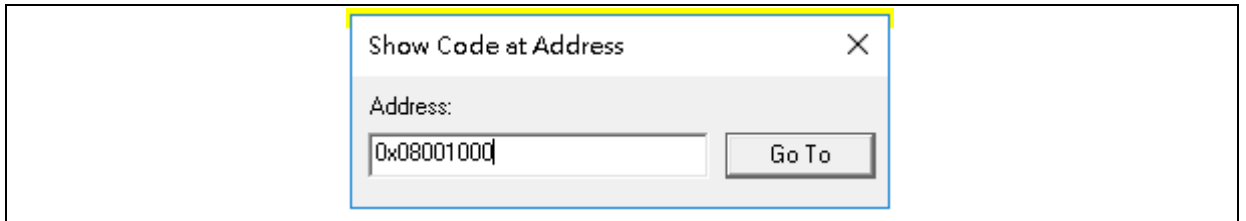
- Open Project\_L1 and compile;
- Click "Start/Stop Debug Session" to enter debug mode;
- Right click in the "Disassembly" interface and select "Show Disassembly at Address", as shown in Figure 37;

**Figure 37. Enter Show Disassembly at Address**



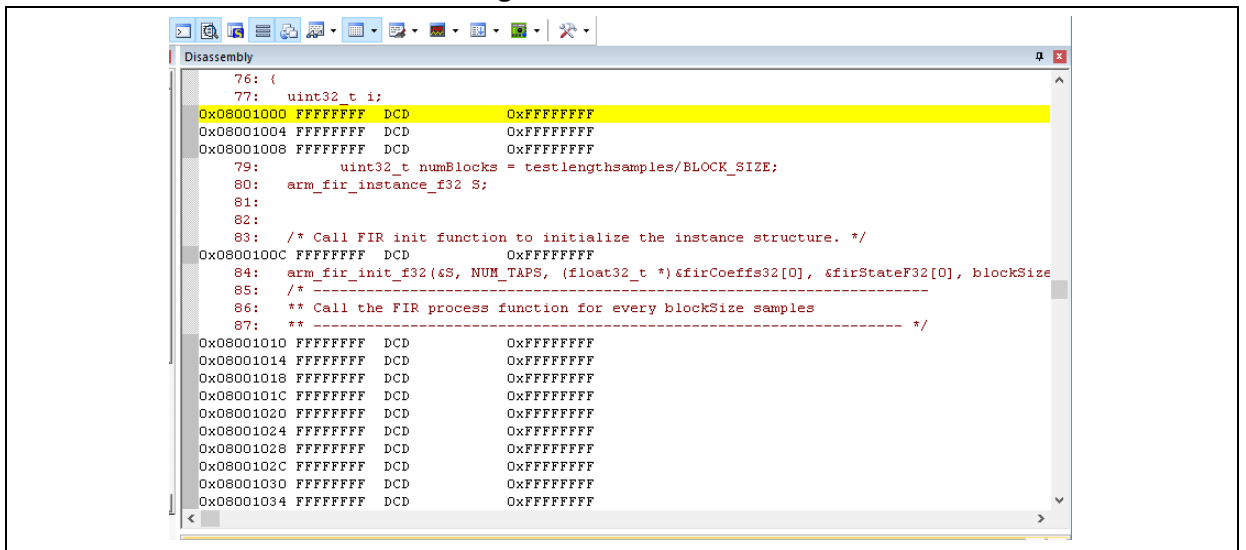
- Enter the address “0x08001000” of SLIB\_INSTRUCTION start sector (sector 2);

**Figure 38. Set Show Code at Address**



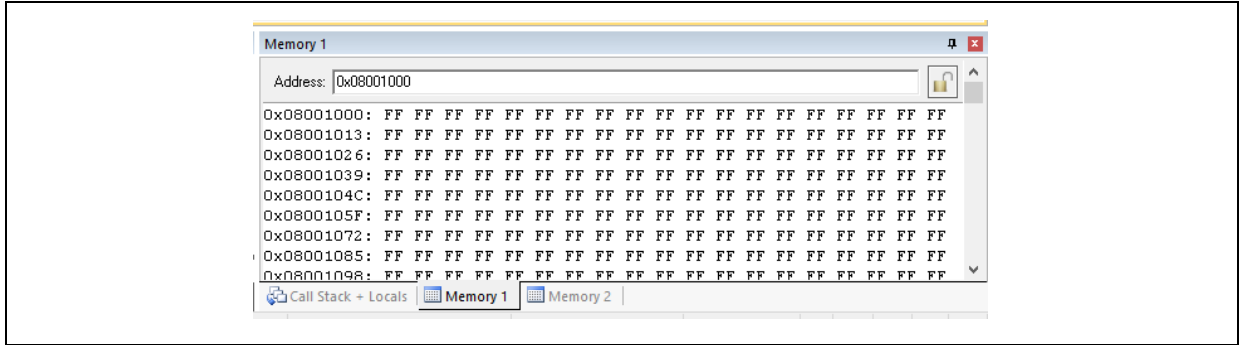
- As shown in Figure 39, codes from 0x08001000 are all 0xFFFFFFFF;

**Figure 39. View codes**



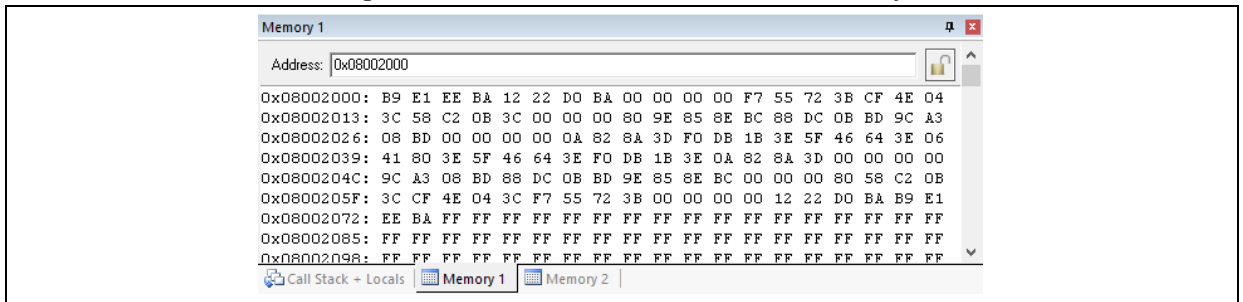
- Similarly, enter address 0x08001000 in “Memory” window, and codes are all 0xFF, as shown in Figure 40;

Figure 40. View codes in Memory



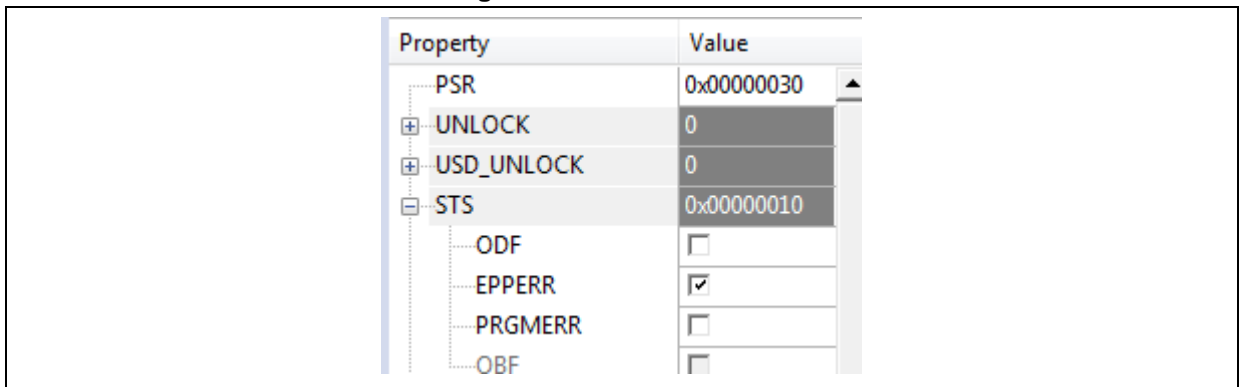
- In the “Memory” window, enter the address 0x08002000 of SLIB\_DATA starting sector (sector 4); this region is allowed to be read through D-Code bus, so original values can be found, as shown in Figure 41;

Figure 41. SLIB\_DATA start sector in Memory



- In the “Memory” window, double click to modify the value of 0x08002000, and a warning pops up by setting EPPERR=1 in the FLASH\_STS register, indicating the protection is enabled;

Figure 42. SLIB write test



- In case of enable erase/program protection error interrupt, continuing execution will enter the interrupt program;

Figure 43. 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 codes and download

After codes of the solution provider and end user are well designed, download to the same MCU on the premise of guaranteeing code security. Project\_L0 and Project\_L1 are used to introduce two downloading methods for reference.

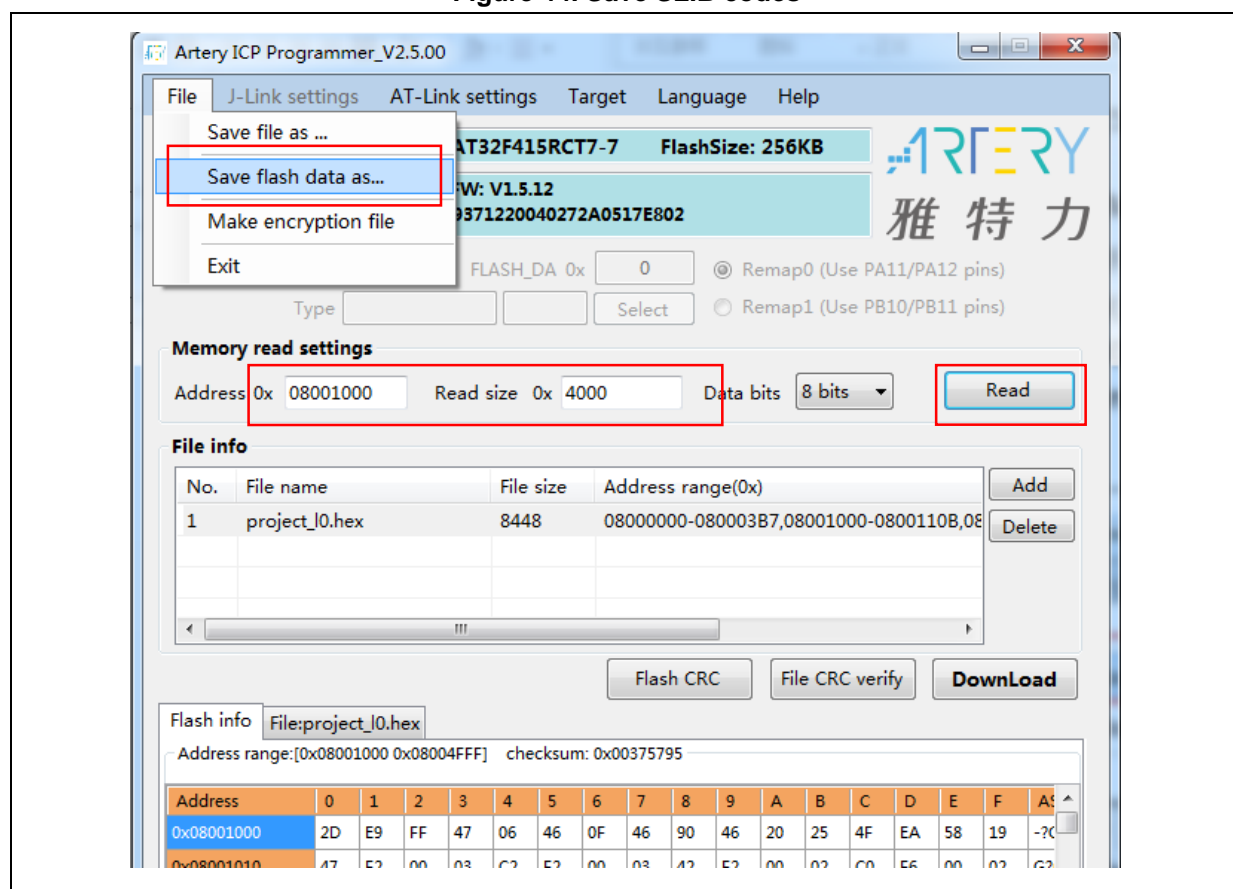
This operation involves offline downloading mode of AT-Link. For details, refer to operation manuals of ICP and AT-Link.

### 4.1 Program codes separately

Firstly, the solution provider programs SLIB codes to MCU; then, the end user programs application codes to MCU. The process is as follows:

- (1) **Method A:** The solution provider uses ICP tool to save the SLIB code in the compiled project as BIN or HEX file: download the complete project to MCU (do not configure SLIB and FAP), read the corresponding SLIB codes and floating-point arithmetic library (0x08001000~0x08004FFF) by using the memory access function, and then click “File-Save Flash data as” to save the codes as BIN or HEX file. In this example, it is named “slib.bin”, as shown in Figure 44.

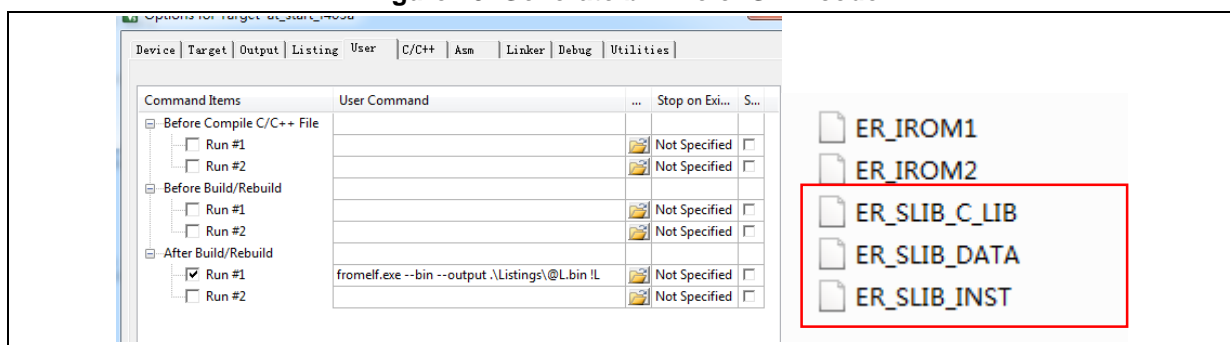
Figure 44. Save SLIB codes



**Method B:** The solution provider uses the compiled project to generate a bin file directly, and take the corresponding section in the SLIB area. For example, in the KEIL project, add “fromelf.exe --bin --output .\Listings\@L.bin !L” in the “user” option to generate a bin file of the corresponding firmware, and add a suffix “.bin” to the SLIB area file. In this example, they are “ER\_SLIB.bin”,

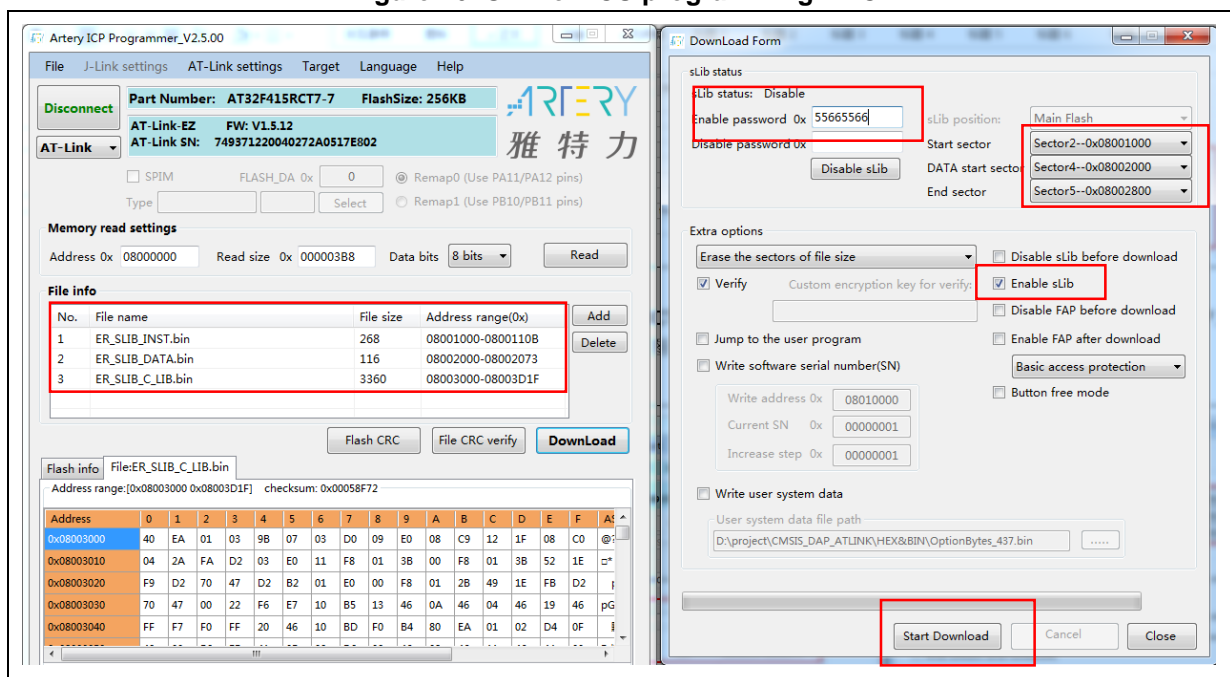
“ER\_SLIB\_DATA.bin” and “ER\_SLIB\_C\_LIB.bin”, corresponding to the SLIB-INSTRUCTION file (0x08001000), SLIB-DATA file (0x08002000), floating-point arithmetic library and C library files, as shown in Figure 45.

Figure 45. Generate bin file of SLIB code



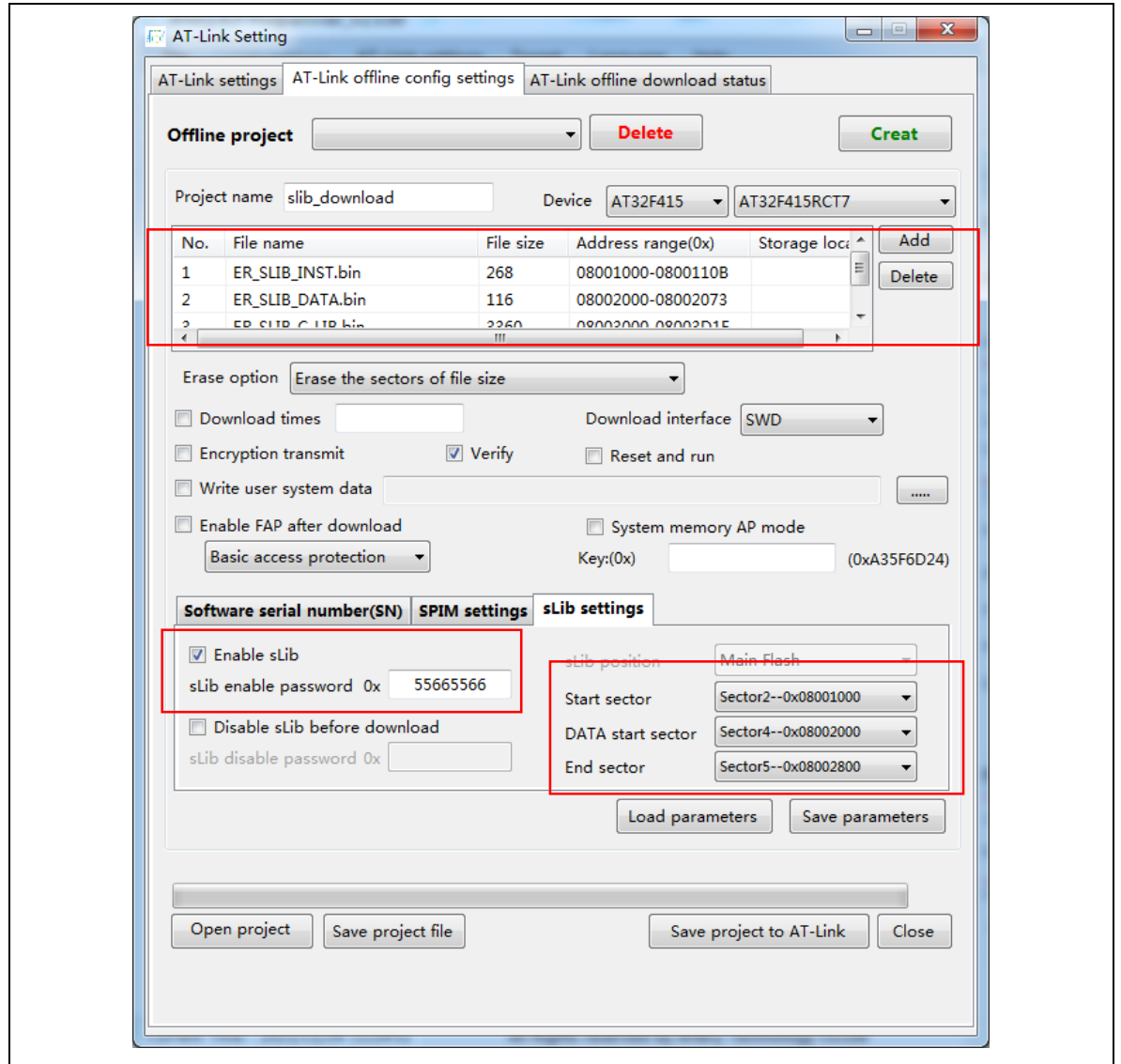
- (2) Use ICP Programmer to program the *ER\_SLIB\_INSTRUCTION.bin*, *ER\_SLIB\_DATA.bin* and *ER\_SLIB\_C\_LIB.bin* to MCU, as shown in Figure 46.

Figure 46. Online MCU programming in ICP



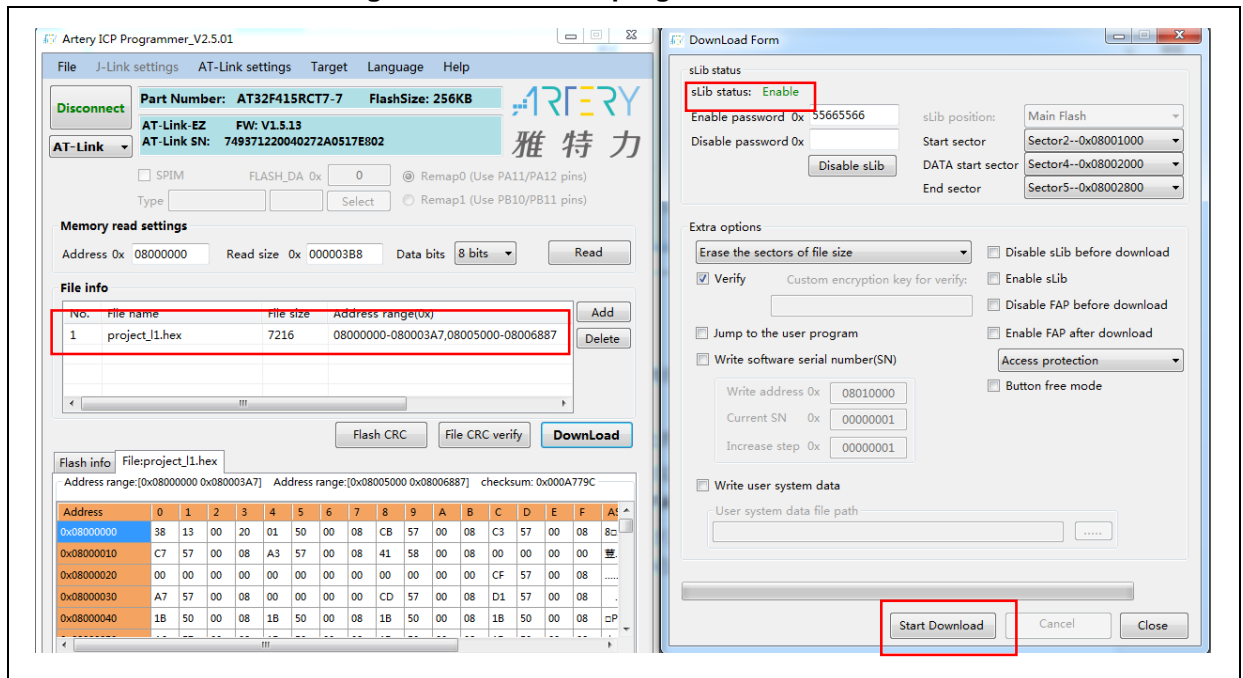
- (3) End users also can use ICP Programmer to set an offline project and save to AT-Link, and then complete offline programming to MCU through AT-Link, as shown in Figure 47.

Figure 47. Offline programming to MCU via AT-Link



- (4) After completing step 2/3, end users can get the MCU with programmed SLIB area (SLIB status: enabled), and program the application code to MCU through online or offline programming, as shown in Figure 48.

**Figure 48. End users program codes to MCU**



## 4.2 Integrate and program codes

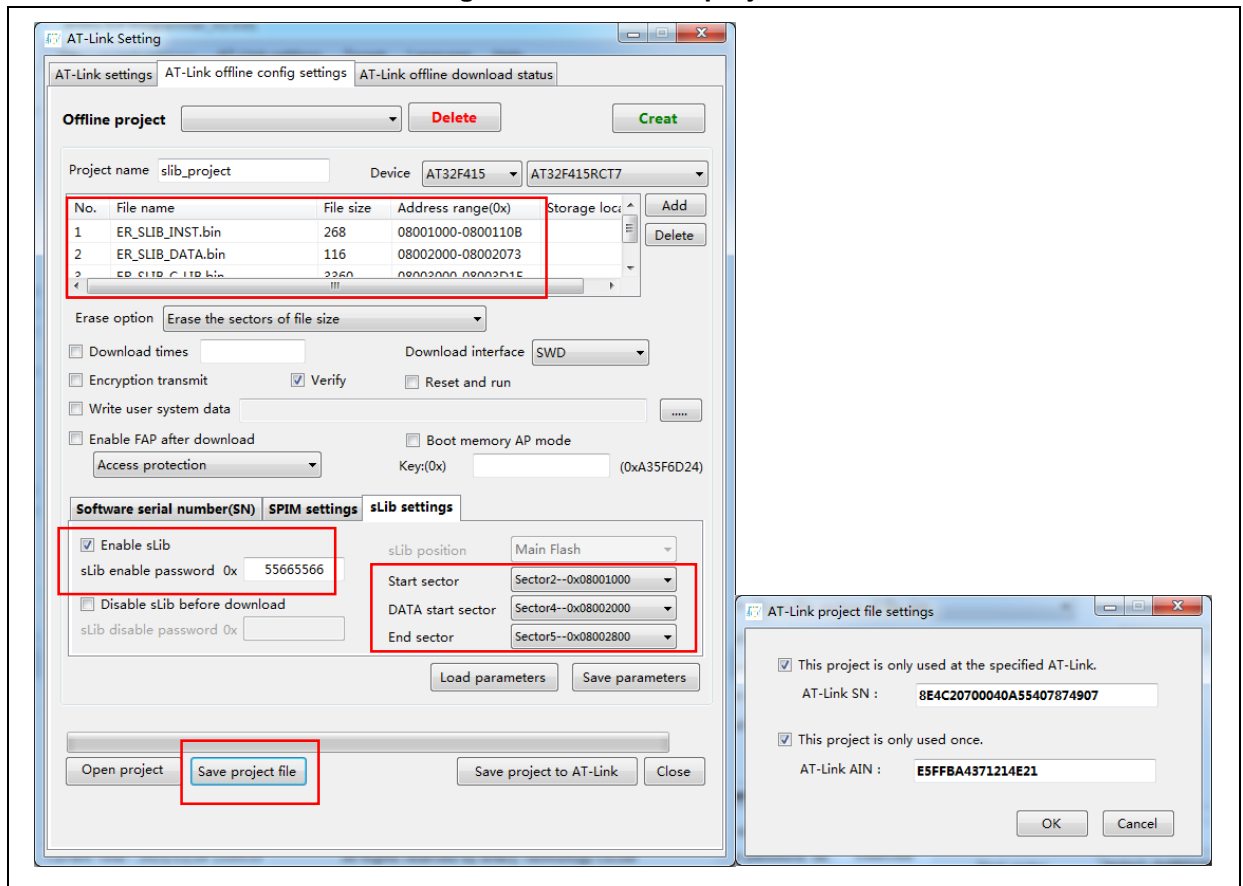
Integrate the SLIB code of solution provider and the end user application code to an offline project, and then download the integrated code to MCU through AT-Link offline programming. The process is as follows:

- (1) The solution provider handles the compiled project as aforementioned to get a slib.bin file;
- (2) The solution provider uses ICP Programmer to generate an offline project and save it to PC. Parameters (such as number of download, project files binding to AT-Link and enable FAP after download) can be configured as needed. Save the offline project as follows.

*Note: The offline project is encrypted. To enhance security, the solution provider also can set the slib.bin file to an encrypted slib.benc and then add it to the offline project. In this case, the offline project can only be used on the AT-Link with the corresponding encryption key.*



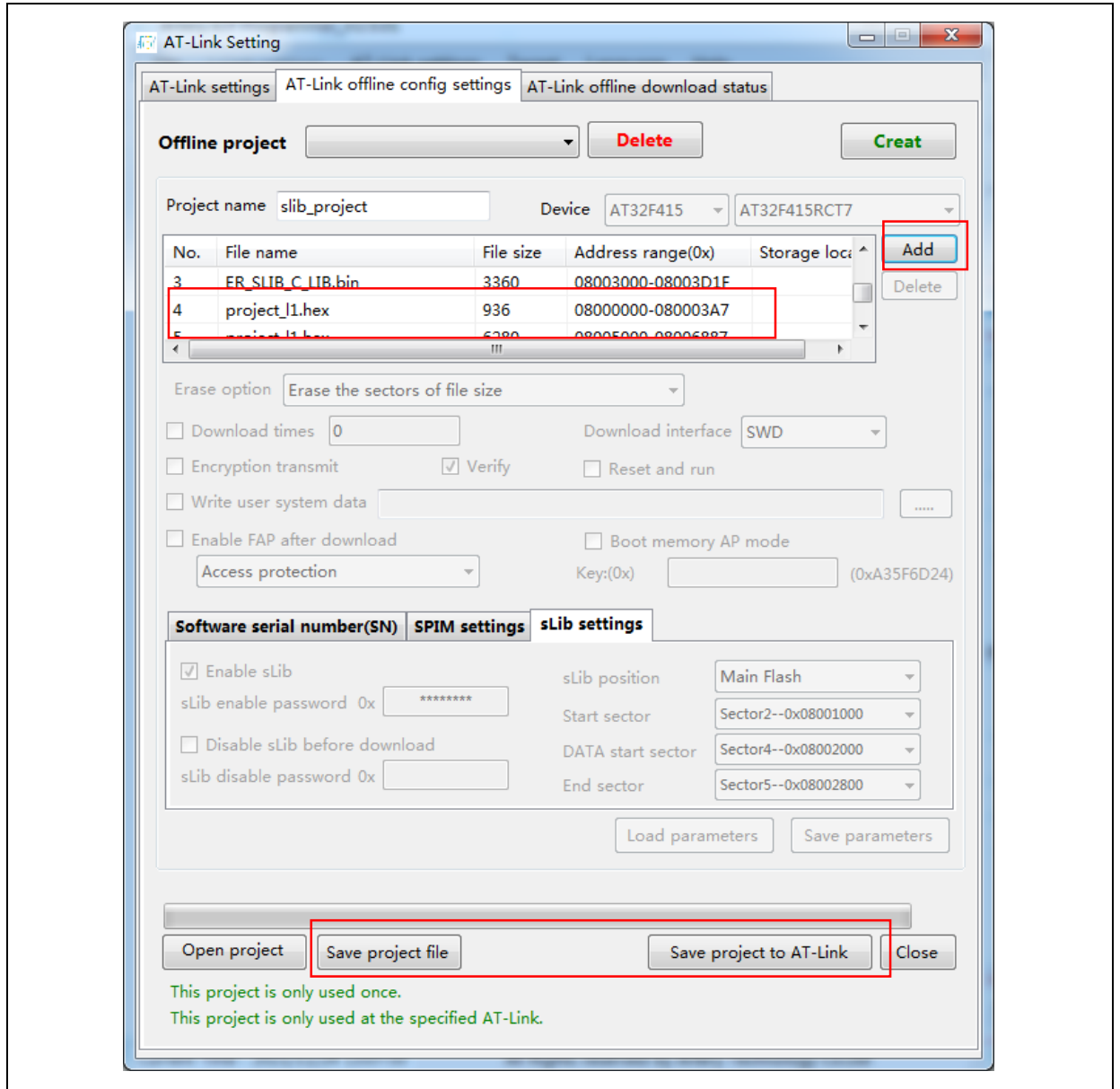
Figure 49. Set offline project



- (3) After obtaining the offline project, the end user should use ICP Programmer to open the project file and add the application codes to the offline project; then save to PC or AT-Link, and perform offline download. Figure 50 shows how to add the project file.

*Note: To protect codes from being leaked or decoded, do not other change settings when adding code file to the offline project, which requires the solution provider to configure the final settings in advance.*

Figure 50. Add project file



## 5 Revision history

Table 2. Document revision history

Date	Version	Revision note
2021.11.16	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, and ARTERY assumes no liability whatsoever relating to the choice, selection or use of the ARTERY products and services described herein.

No license, express or implied, to any intellectual property rights is granted under this document. If any part of this document deals with any third party products or services, it shall not be deemed a license grant by ARTERY for the use of such third party products or services, or any intellectual property contained therein, or considered as a warranty regarding the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

Unless otherwise specified in ARTERY's terms and conditions of sale, ARTERY provides no warranties, express or implied, regarding the use and/or sale of ARTERY products, including but not limited to any implied warranties of merchantability, fitness for a particular purpose (and their equivalents under the laws of any jurisdiction), or infringement of any patent, copyright or other intellectual property right.

Purchasers hereby agrees that ARTERY's products are not designed or authorized for use in: (A) any application with special requirements of safety such as life support and active implantable device, or system with functional safety requirements; (B) any air craft application; (C) any automotive application or environment; (D) any space application or environment, and/or (E) any weapon application. Purchasers' unauthorized use of them in the aforementioned applications, even if with a written notice, is solely at purchasers' risk, and is solely responsible for meeting all legal and regulatory requirement in such use.

Resale of ARTERY products with provisions different from the statements and/or technical features stated in this document shall immediately void any warranty grant by ARTERY for ARTERY products or services described herein and shall not create or expand in any manner whatsoever, any liability of ARTERY.