AN0071
Application Note

AT32F421 Security Library Application Note

Introduction

This application note introduces the security library (sLib) application principle of AT32F421 MCUs, operation methods and example projects.

Applicable products:

Part number	AT32F421

Contents

List of tables

List of figures

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 AT32F421 series MCUs are designed with a security library (sLib) to protect important IP-Codes against being changed or read by the end user's program.

This application note details the sLib application principle and operation methods of AT32F421 MCUs.

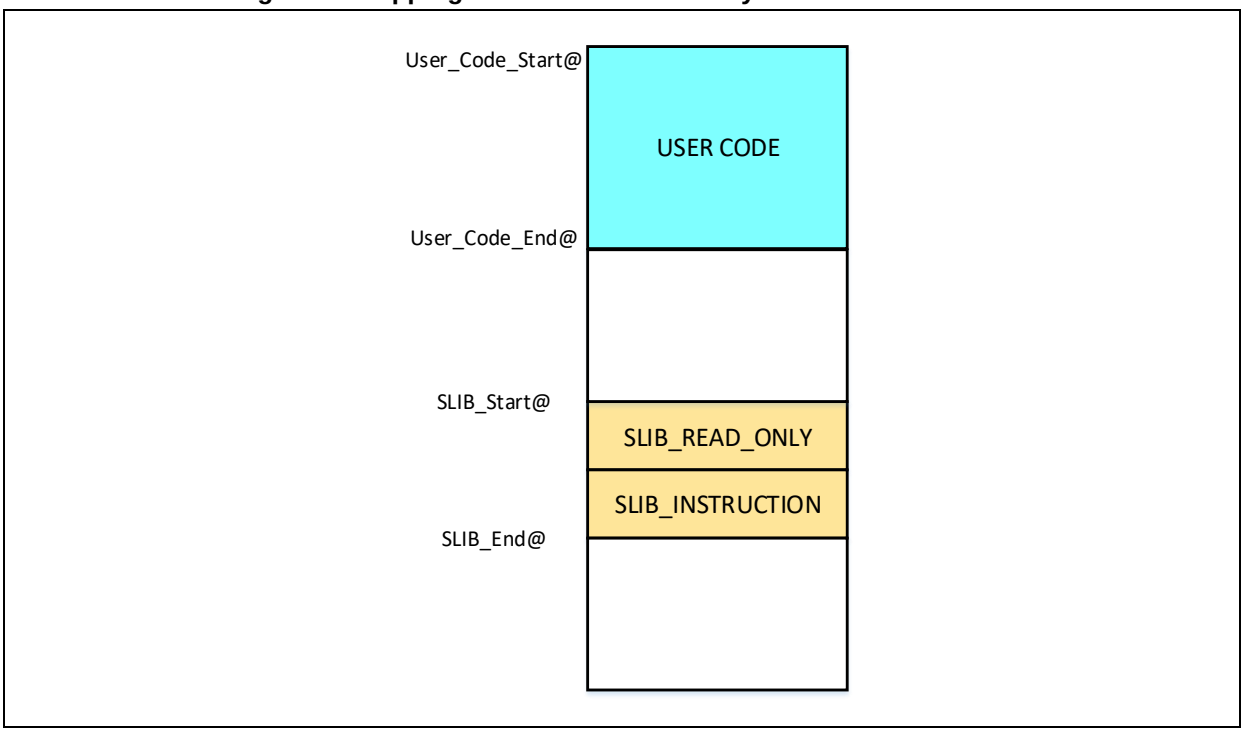
2 Application principles

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 contains data security library (SLIB_READ_ONLY) and instruction security library (SLIB_INSTRUCTION); users can set part of or the whole security library as SLIB_READ_ONLY or SLIB_INSTRUCTION.
- Data in the SLIB_READ_ONLY area can only be read through I-Code and D-Code and cannot be programmed.
- Program code 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 bus (including ISP/ICP/debug mode and programs that boot from internal RAM). When reading the SLIB_INSTRUCTION area, values are all read 0xFF or 0x00.
- 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 programmed 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 AT32F421 series MCUs. In addition, once the boot memory is programmed as memory extension area, the entire 4 KB area can be used as security library.

Table 1. Flash size of AT32F421 MCUs

Model	Internal Flash size (Byte)	Sector size (Byte)	Configurable range
AT32F421x4	16K	1K	Sector 0 ~ 15(1) (0x08000000 ~ 0x08003FFF)
AT32F421x6	32K	1K	Sector 0 ~ 31(1) (0x08000000 ~ 0x08007FFF)
AT32F421x8	64K	1K	Sector 0 ~ 63(1) (0x08000000 ~ 0x0800FFFF)

(1) Sector 0 cannot be configured as the instruction security library.

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 by writing 0xA35F6D24 to the SLIB_UNLOCK register. Then check the SLIB_ULKF bit in the SLIB_MISC_STS register to verify if it is unlocked successfully. If successful, write the programmed value 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 if it is unlocked successfully;
- Set the sectors to be protected in the SLIB_SET_RANGE register, including the SILB start/end addresses and the start address of instruction security library;
- 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 a system reset, and then reload the security library setting words;
- Read the SLIB_STS0/STS1 register to verify the security library settings.

Note:

- It is allowed to set security library in the main Flash memory and its extension area; refer to Table 1 for the configuration range;
- The security library code must be programmed by sectors, with its start address aligned with the address of main Flash memory or its extension area;
- The interrupt vector table is in data type and usually placed in the first sector (sector0, which should not be configured as security library instruction area) of the main Flash memory.

For details about security library setting register, refer to *AT32F421 Series Reference Manual*.

The security library can be enabled by the *slib_enable()* function in *main.c* file of project_l0. In addition, users can use Artery ICP or ISP tools for configuration.

2.3 How to disable sLib protection

The security library protection can be disabled by writing the previously programmed 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 programmed password to the SLIB_PWD_CLR register;
- Perform a system reset, and then reload security library setting words;
- Read the SLIB_STS0 register to verify 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, 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 the CRM_CFG register in a PC (program counter) indirect addressing manner, and the address of the 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 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.

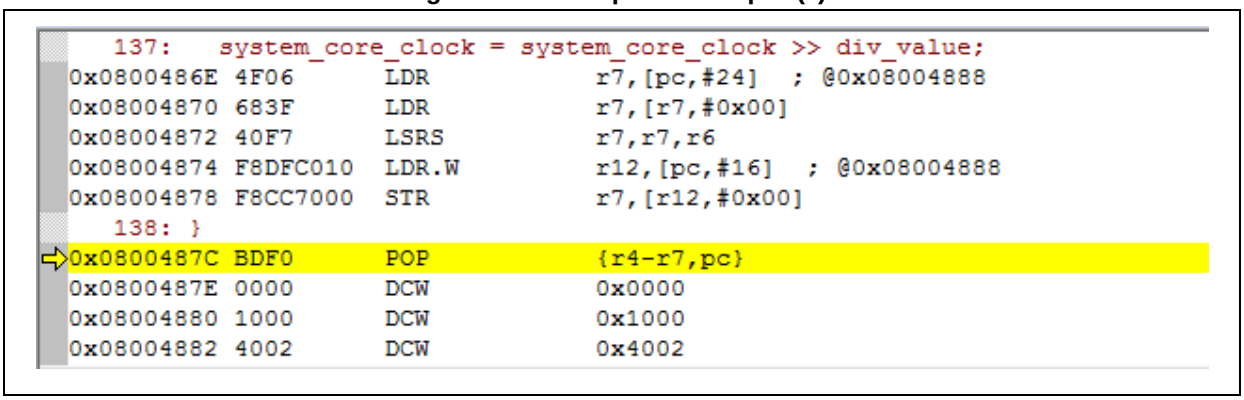
Figure 2. Literal pool example (1)

```
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:
```

main.c | startup_at32f403a_407.s | at32f403a_407_clock.c | system_at32f403a_407.c | at32f403a_407_crm.c | at32f403a_407_gpio.c

```
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)



2.4.1 Setting interrupt vector table as sLib instruction area not allowed

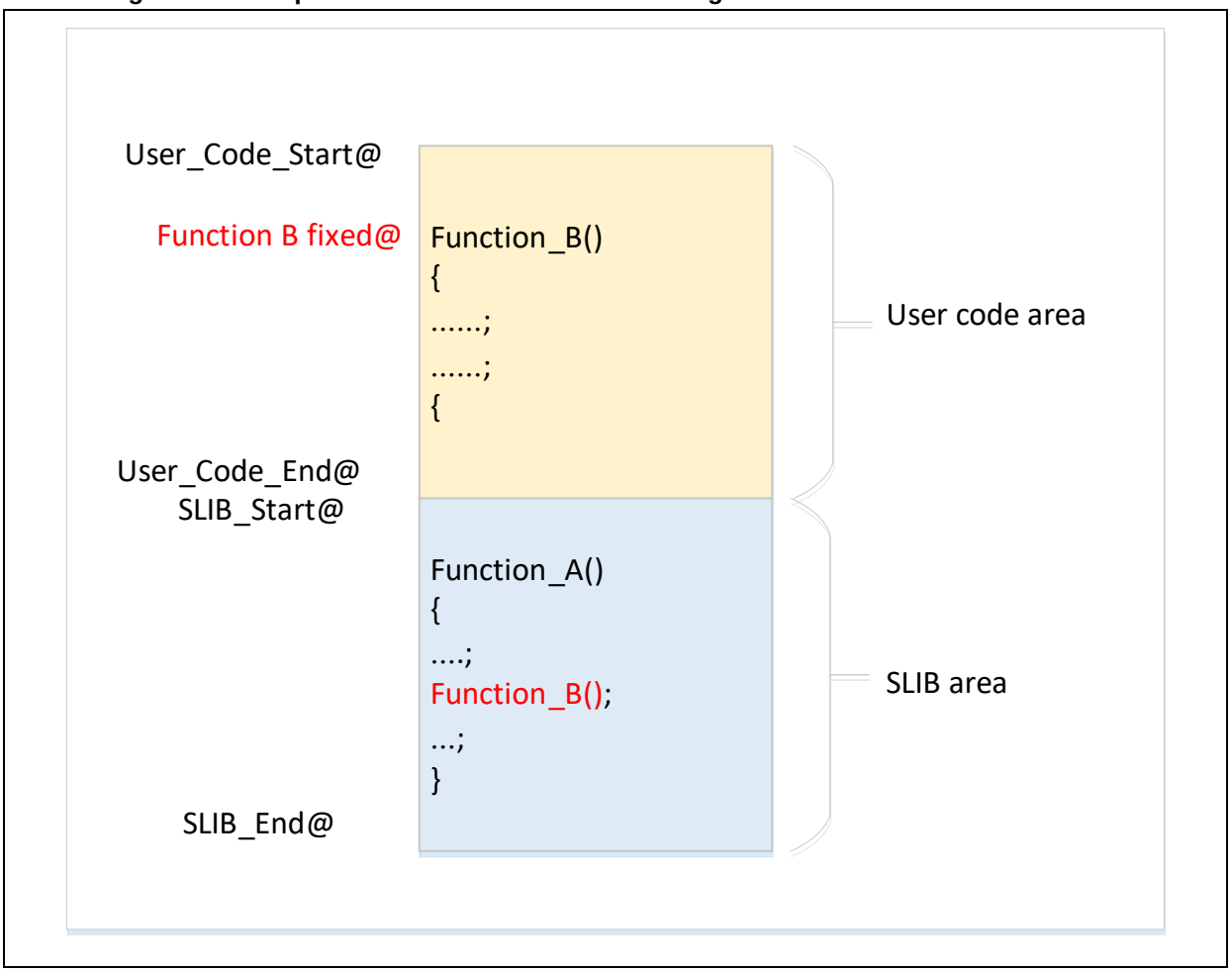
The interrupt vector table contains entry point address of each interrupt handler, which is read by MCU via D-Code bus. Generally, the interrupt vector table is located in the first sector (sector0, starting address: 0x08000000). Therefore, the following rules must be followed when setting the instruction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

- Do not configure the first sector of the main Flash memory as sLib instruction area.

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 cannot 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).
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 AT32F421 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 in the SLIB_READ_ONLY area. 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 EBAC4E50  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 protection is enabled, all contents in the sLib protected area 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 the SLIB_READ_ONLY area, so that programs in 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 Hardwar requirements

- AT-START-F421 demo board with AT32F421C8T7 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 or ISP programming tools for enabling and disabling sLib

3.2 Overview

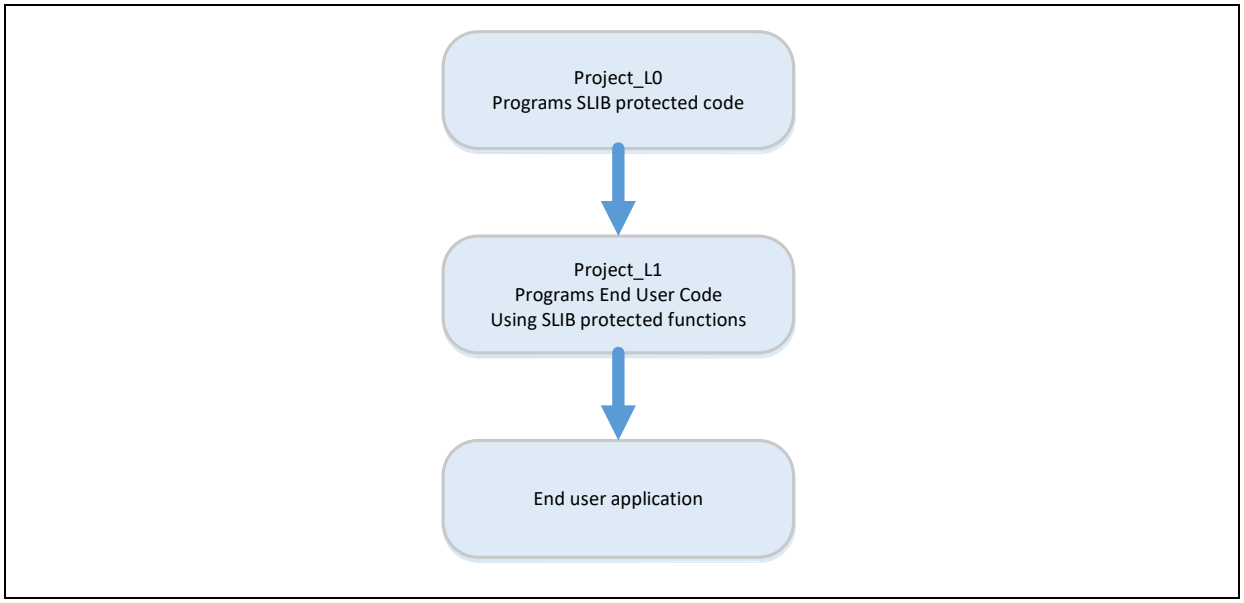
This application note provides two sample projects to demonstrate that software developers develop IP-Code for end-user applications.

- 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 AT32F421 chip and configured as sLib protected. In addition, the following settings are available for the end-user applications.

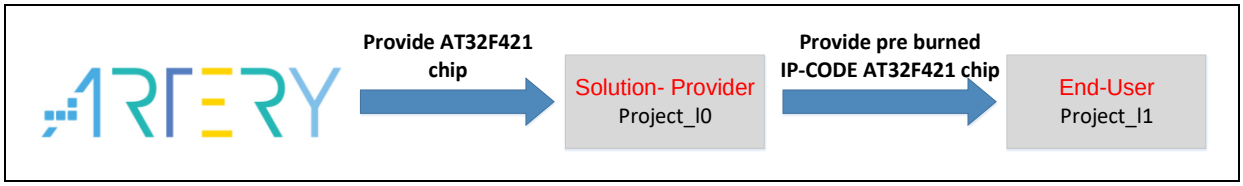
- Main Flash memory mapping, showing the area occupied by sLib and the area where users can develop programs;
- Header file that contains algorithm function definitions, and end users can call relevant functions;
- Symbol definition file, which contains the actual address of each IP-Code function, so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

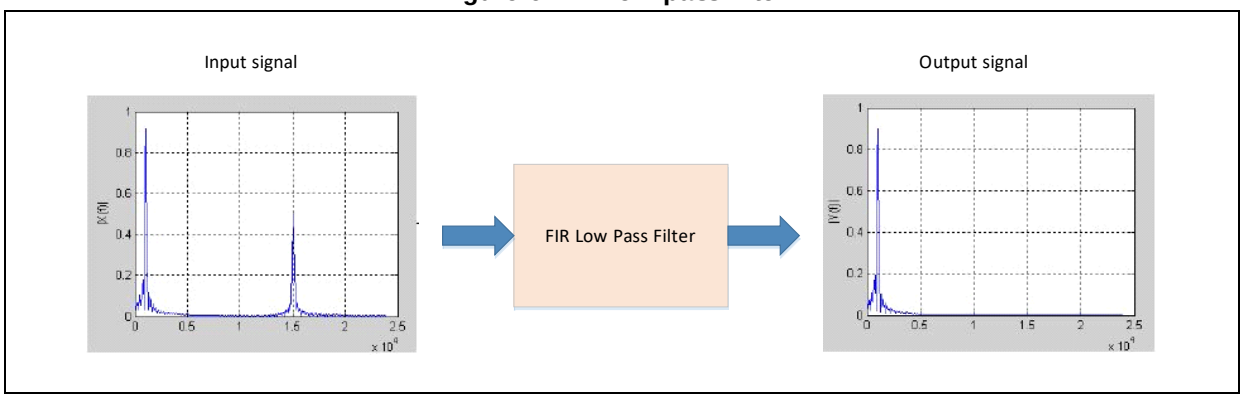


3.3 SLIB protected code: FIR low-pass filter

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. It is called by the end user and 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 read-only area in the example.

Since the AT32F421 series does not have an embedded hardware FPU, the floating-point arithmetic library functions in this example are used for signal processing and floating-point operations.

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 sector12 to sector19 (address: 0x08003000 ~ 0x08004FFF);
- Place the filter function coefficients, floating-point arithmetic library and C library used by the program to the main Flash memory sector4 to sector11 (address: 0x08001000 ~ 0x08002FFF);
- Execute *FIR_lowpass_filter()* in the main program to verify its correctness;
- If correct, configure sector12 to sector19 as instruction security library and sector4 to sector11 as read-only area, which can be completed by calling *slib_enable()* function in the main program or using Artery ICP Programmer (recommended);
- 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 pool and branch table.

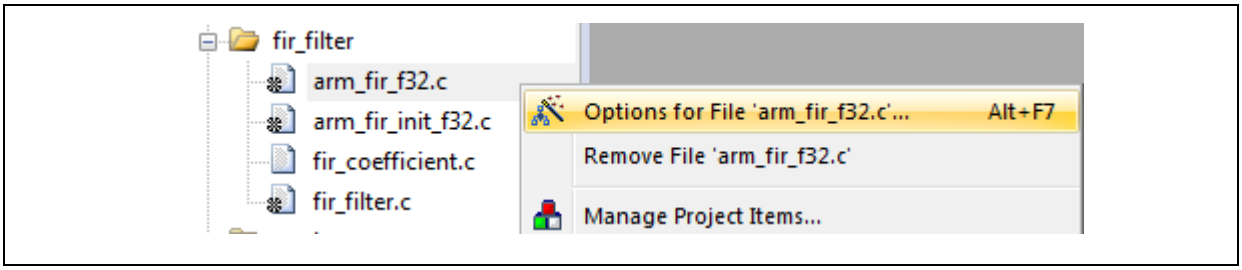
For example, Keil® µvision has Execute-only Code option, which can be set as follows:

Keil® µvision: Set Execute-only Code option

Operate as follows:

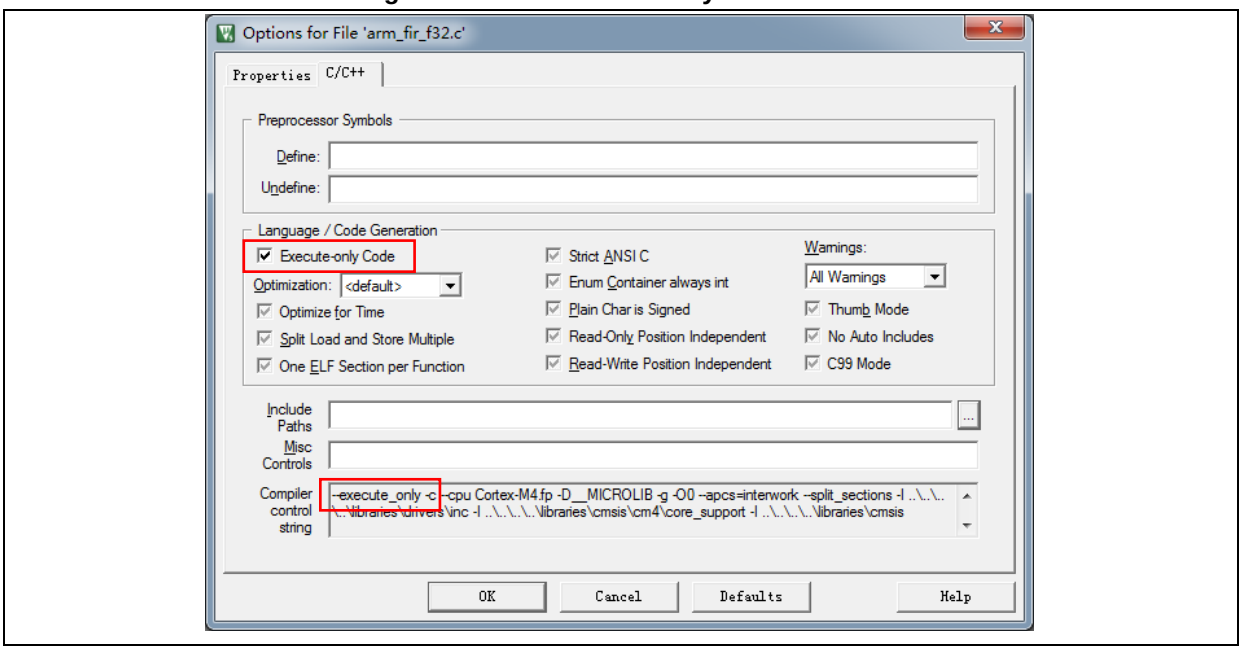
- Select C file group or individual C file (in this example, the C files to be protected are placed in "fir_filter");
- Right click and select the corresponding files (for example, the *Option for File 'arm_fir_f32.c'*), as shown in Figure 10;

Figure 10. Enter Option interface in Keil



- Tick "Execute-only Code" in the C/C++ 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



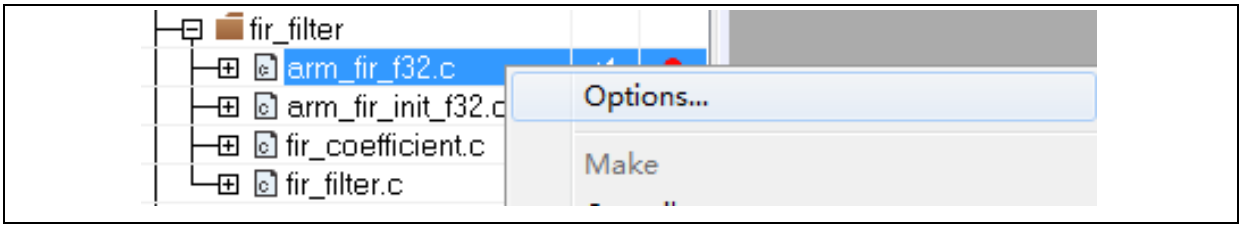
- The a*rm_fir_f32.c*, *arm_fir_init_f32.c* and *fir_filter.c* are in the SLIB_INSTRUCTION area, and these files need to be set as generating execute-only code.

IAR: Set No data read in code memory option

Operate as follows:

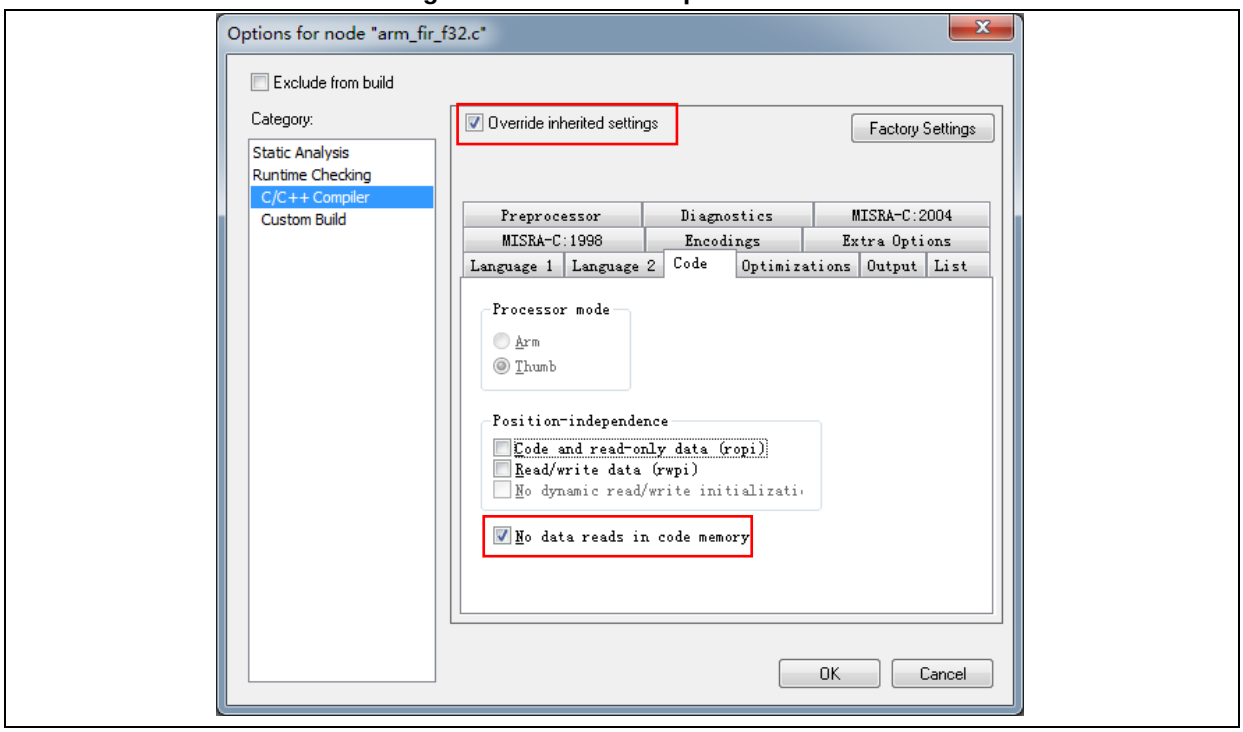
- Select the corresponding file in the *fir_filter* group; right click and select Option;

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

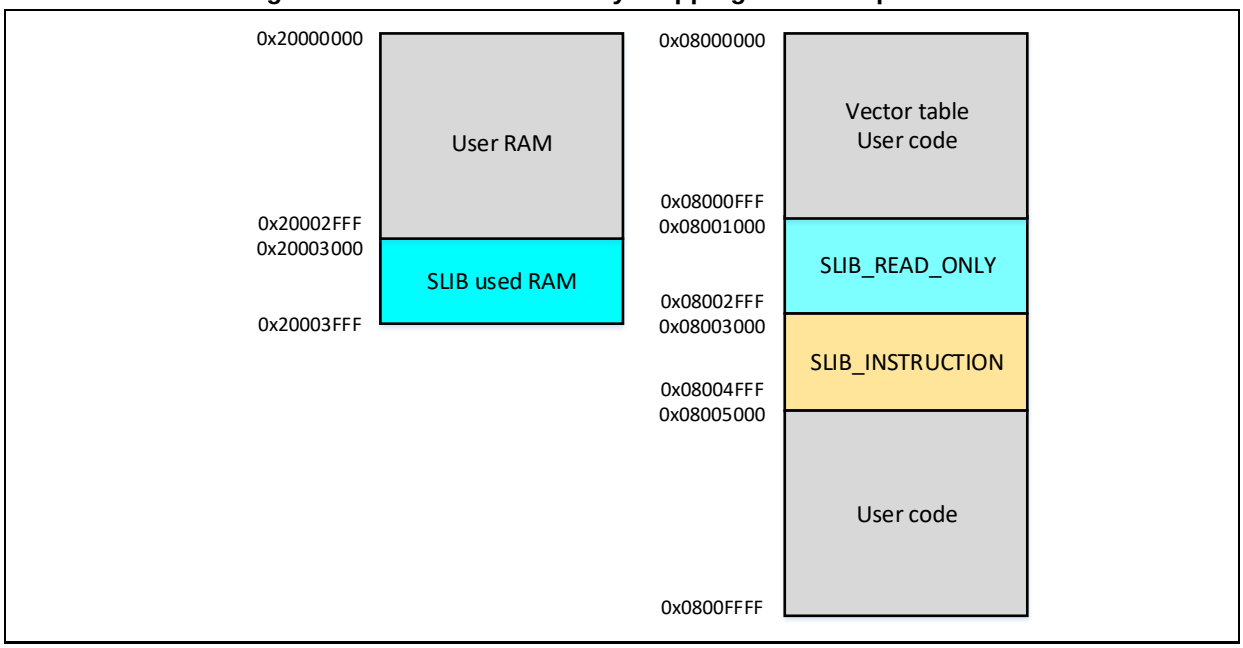


- The *arm_fir_f32.c*, *arm_fir_init_f32.c* and *fir_filter.c* are in the SLIB_INSTRUCTION area, and these files need to be configured as generating execute-only code.

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 4 in this example, with sectors 12-19 being set as instruction security library and sector 4-11 being set as read-only area. 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

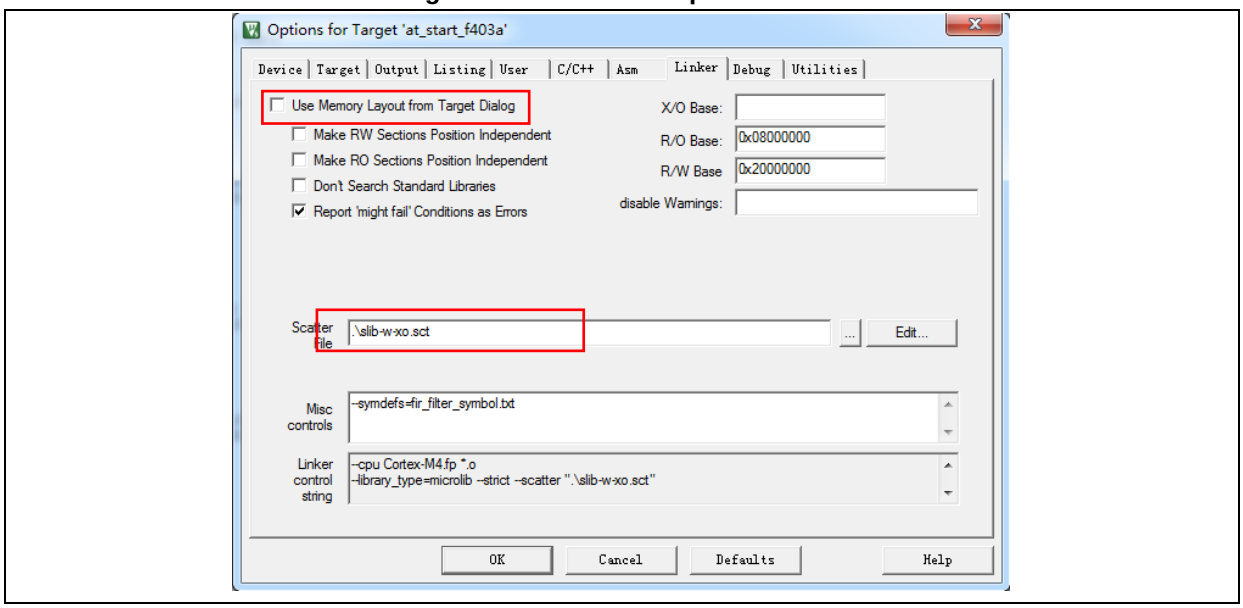


Keil® µvision: scatter file

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 12 and occupies eight sectors) and modify the label to “execute-only (+XO)”. Place the area occupied by SLIB_READ_ONLY to a dedicated loading area named “LR_SLIB_READ_ONLY” to avoid the compiler compiling other non-IP-Code functions to the SLIB area. The RW_IRAM2 assigns the region from 0x20003000 to 0x20003FFF to the sLib algorithm functions to avoid the same RAM region being used by end-user project, causing fault or error in program execution process.

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 0x00003000  {  ; user RW data
   .ANY (+RW +ZI)
  }

  RW_IRAM2 0x20003000 0x00001000  {  ; RAM used for slib code
   fir_filter.o (+RW +ZI)
  }
}

; user can also use this section to place slib read-only
LR_SLIB_READ_ONLY 0x08001000 0x00002000  {    ; sLib read-only
  ER_SLIB_READ_ONLY 0x08001000 0x00002000  {
   fir_coefficient.o (+RO)
   *armlib* (+RO)
  }
}

LR_SLIB_INST 0x08003000 0x00002000  {    ; slib inst
  ER_SLIB_INST 0x08003000 0x00002000  {  ; load address = execution address
    arm_fir_init_f32.o (+XO)
    arm_fir_f32.o (+XO)
    fir_filter.o (+XO)
  }
}

LR_IROM2 0x08005000 0x0000B000  {    ; user code area
  ER_IROM2 0x08005000 0x0000B000  {  ; load address = execution address
   .ANY (+RO)
  }
}
```

Note: Use *armlib* to compile the floating-point arithmetic library functions and C standard library to LR_SLIB_READ_ONLY 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 0x20003000, 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]  @ 0x20003000 ;
#elif defined ( __CC_ARM )
static float32_t firStateF32[BLOCK_SIZE + NUM_TAPS - 1] __attribute__((at(0x20003000)));
#endif
```

- The start address of read-only area is sector 4 (0x08001000). To compile the constants used by FIR low-pass filter to this address, users can modify the scatter file as aforementioned, or use 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] @ 0x08001000 ={
#elif defined ( __CC_ARM )
const float32_t firCoeffs32[NUM_TAPS] __attribute__((at(0x08001000))) = {
#endif
```

IAR: ICF file

Operate as follows:

- Open the *icf* file in "\project_l0\IAR_V8.2\", and add three new loading areas as shown in Figure 19. The SLIB_RAM region reserves the corresponding RAM (0x20003000 to 0x20003FFF) for the algorithm functions.

Figure 19. 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__   = 0x08002FFF;

/* SLIB instruction area */
define symbol __ICFEDIT_region_SLIB_INST_start__ = 0x08003000;
define symbol __ICFEDIT_region_SLIB_INST_end__   = 0x08004FFF;

define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__   = 0x20002FFF;

/* SLIB RAM region */
define symbol __ICFEDIT_region_SLIB_RAM_start__ = 0x20003000;
define symbol __ICFEDIT_region_SLIB_RAM_end__   = 0x20003FFF;
```

- 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

```
/* 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 0x20003000 ~ 0x20003FFF 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 the RAM used by IP-Code, users can use the IAR @ descriptor to load variables to a fixed address (0x20003000) or modify the *icf* file, as shown in Figure 21.

Figure 21. Modify SLIB used 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 read-only area is sector 4 (0x08001000). To compile the constants, m7Mx_tl.a (floating-point arithmetic library) and rt7Mx_tl.a (C library) used by FIR low-pass filter to this address, as shown in Figure 22.

Figure 22. Modify SLIB read-only area 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,
                                    ro object m7Mx_tl.a,
                                    ro object rt7Mx_tl.a };
```

3.4.3 Enable sLib protection

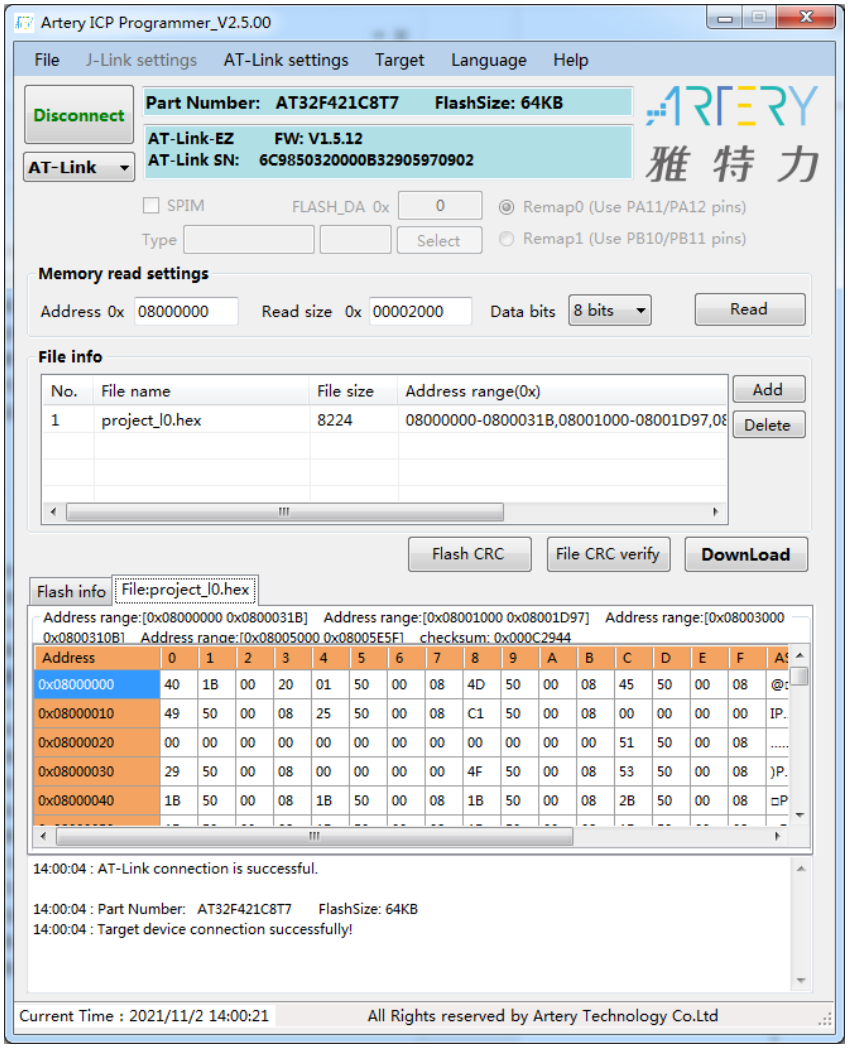
There are two methods to enable sLib protection:

(1) Use Artery ICP Programmer (recommended)

It is recommended to use Artery ICP Programmer as follows:

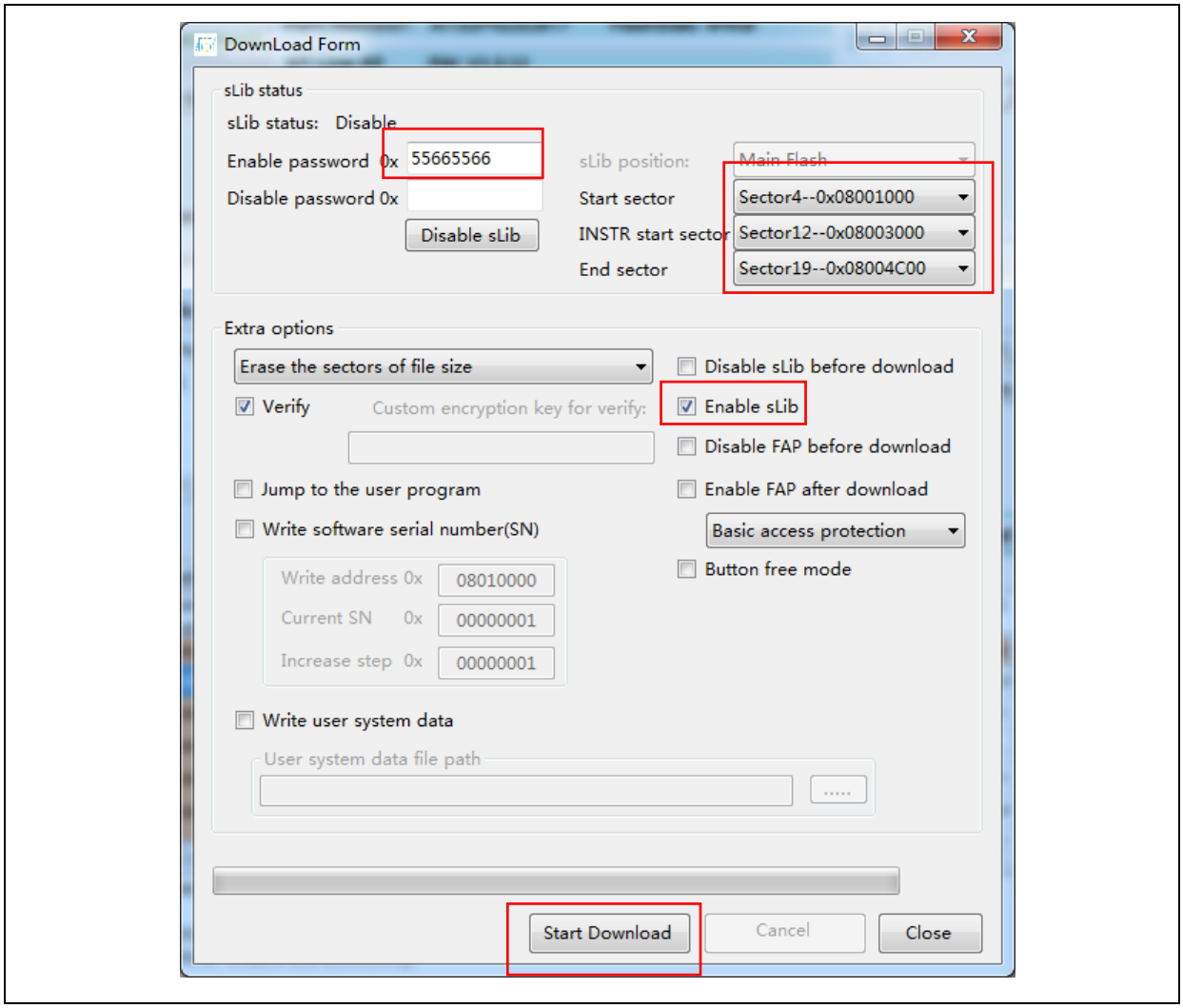
- Connect AT-Link emulator to AT-START-F421 board and then power on;
- Open ICP Programmer, select AT-Link for connection, and add the HEX or BIN file generated by Project_L0, as shown in Figure 23.

Figure 23. Configure ICP Programmer



- Click “Download” and the “Download Form” pops up, which shows sLib status and relevant parameters. Set the start sector, INSTR start sector and end sector; set the enable password as “0x55665566” (user-defined) and tick “Enable sLib”; then click “Start Download” to complete programming and enable sLib successfully, as shown in Figure 24.

Figure 24. Set parameters in Download Form



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

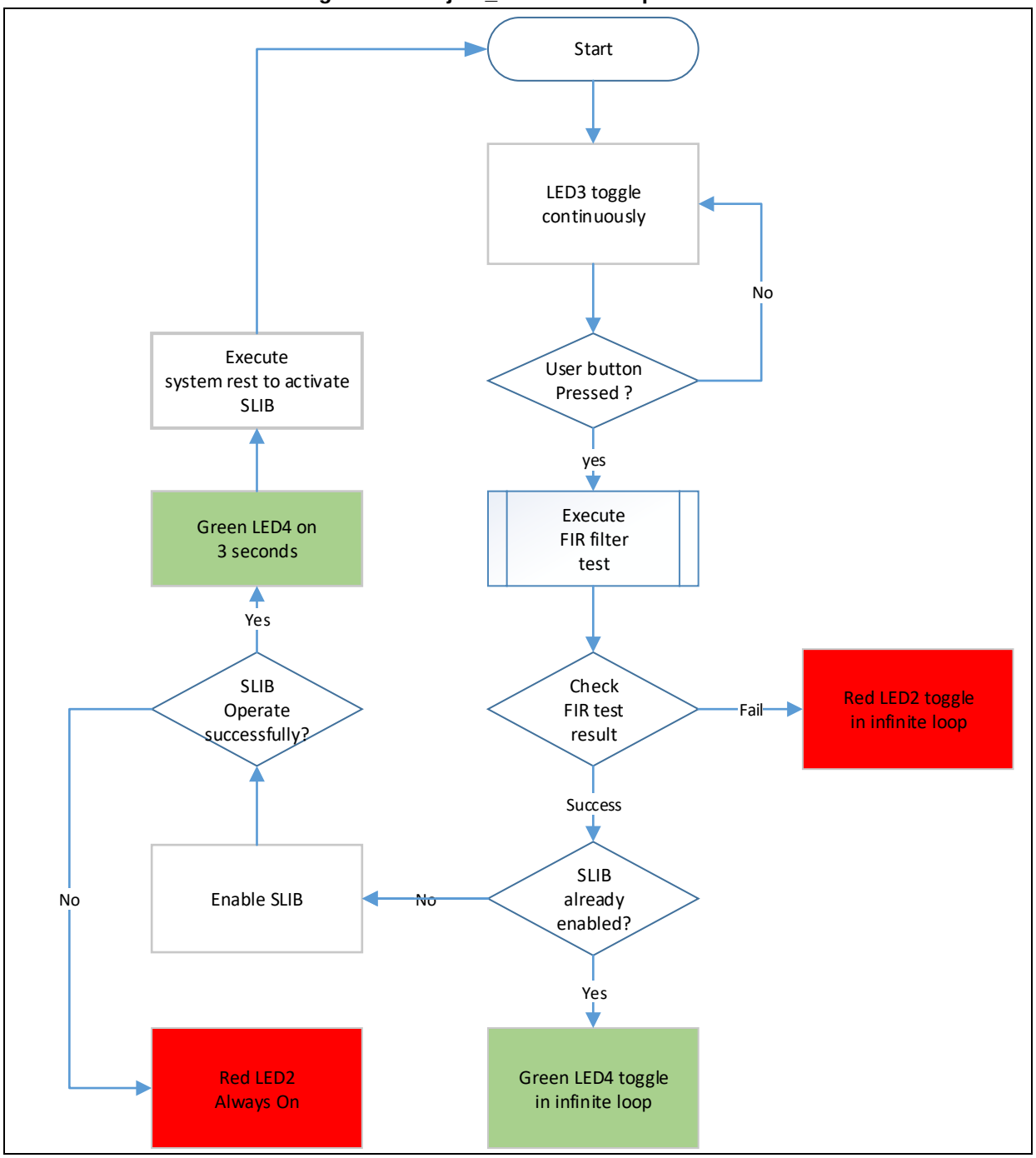
(2) Use *slib_enable()* in main.c

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 25 shows the Project_L0 execution process.

Figure 25. 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\AT32F421_slib_demo\project_l0\mdk_v5\, and then compile;

(2) Before downloading the code, check whether the chip on AT-START-F421 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.

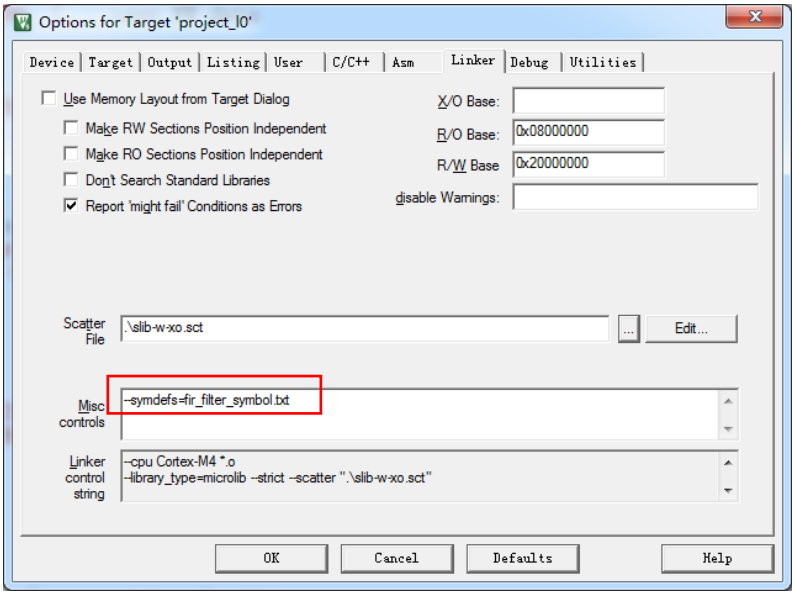
The generation of symbol definition file is related to the specific toolchain being used.

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 26;

Figure 26. Set Misc controls in Keil



- After compiling the project, a symbol definition file named "fir_filter_symbol.txt" is generated under "project_l0\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 27;

Figure 27. Contents of modified fir_filter_symbol.txt

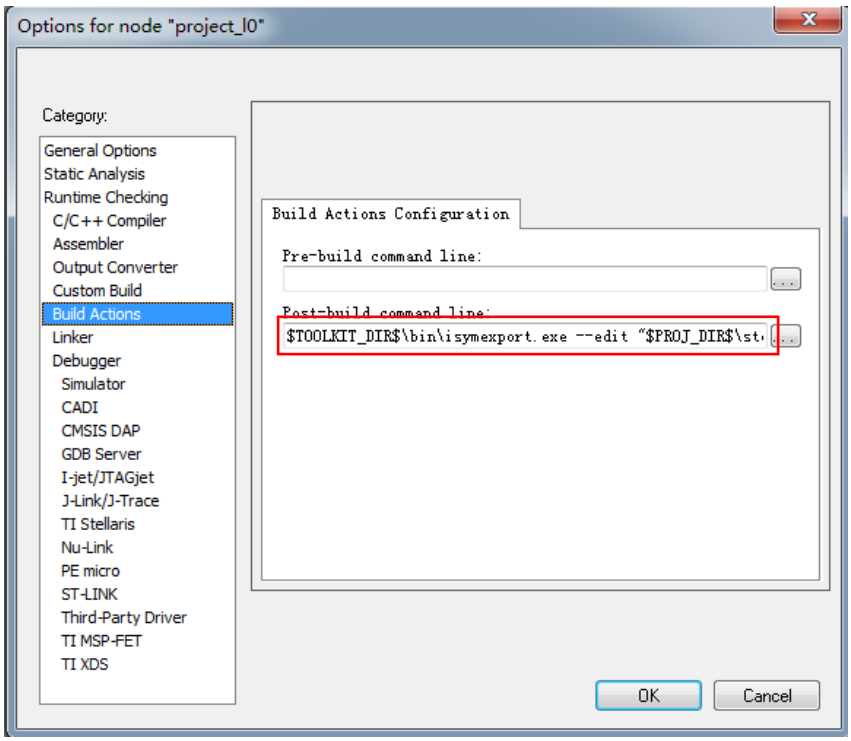
```
0x08003001 T FIR_lowpass_filter
```

Use IAR to generate symbol definition file

Operate as follows:

- Select Project→Option→Build Actions

Figure 28. 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_l0\iar_v8.2”, which is used to select function symbols to be generated. Users can manually edit the contents called by sLib. As shown in Figure 29, the "show" is the command used to select functions.

Figure 29. Edit steering_file.txt contents

```
show FIR_lowpass_filter
```

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 AT32F421 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 those 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. In the main Flash memory, sector 4 to sector 19 are occupied by security library, which should be isolated by using the linker control file to avoid code being compiled to this region.

Keil® µvision: scatter file

Refer to the *end_user_code.sct* 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 region behind 0x20003000 in the RAM should be reserved, as shown in Figure 30.

Figure 30. 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 0x00003000  {  ; RW data
   .ANY (+RW +ZI)
  }

 ; 0x20003000 ~ 0x20003FFF   RAM reserved for SLIB code

}

; 0x08001000 ~ 0x08004FFF is SLIB area

LR_IROM2 0x08005000 0x0000B000  {    ; load region size_region
  ER_IROM2 0x08005000 0x0000B000  {  ; load address = execution address
   .ANY (+RO)
  }
}
```

IAR: ICF file

Refer to the *enduser.icf* under "project_l1\iar_V8.2\", as shown in Figure 31.

Figure 31. 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__];
```

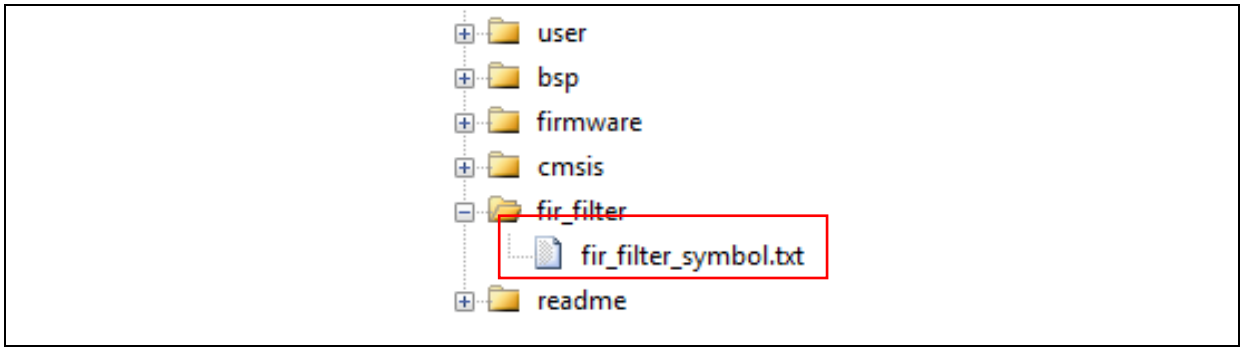
3.5.2 Add symbol definition file to project

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® µvision

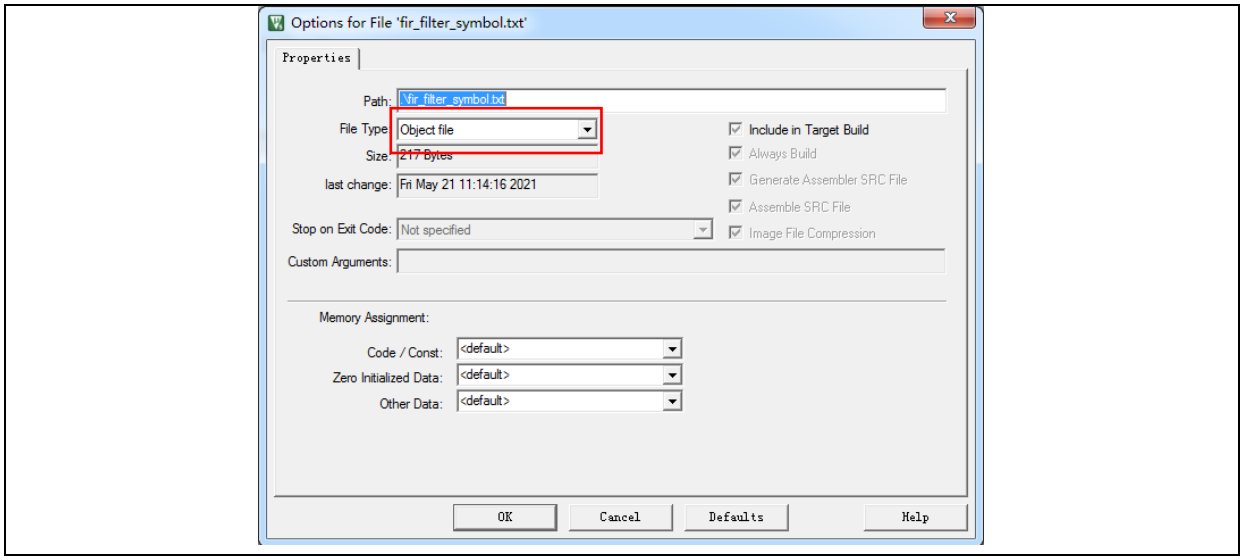
Add *fir_filter_symbol.txt* to the project, as shown in Figure 32.

Figure 32. 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 33.

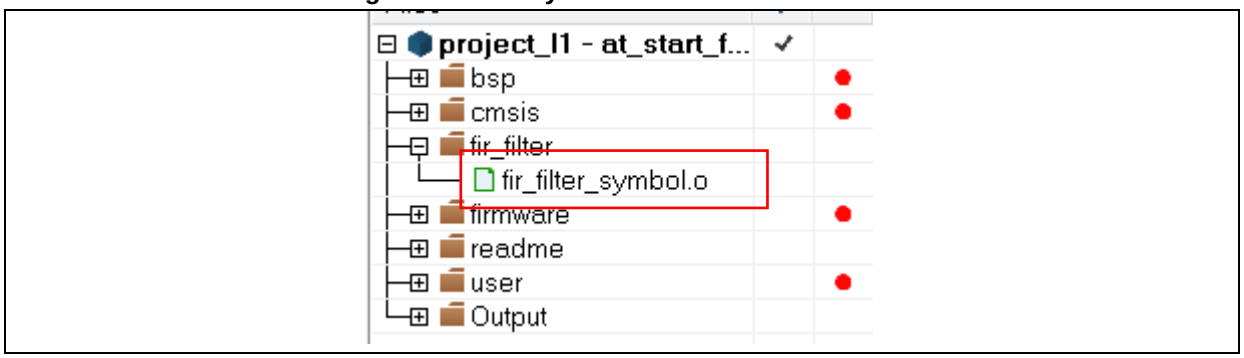
Figure 33. Modify symbol definition file type to “Object file”



Add symbol definition file in IAR

Add the fir_filter_symbol.o (Object) to fir_filter, as shown in Figure 34.

Figure 34. Add symbol definition file in IAR



3.5.3 Call functions in SLIB-protected area

After 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:

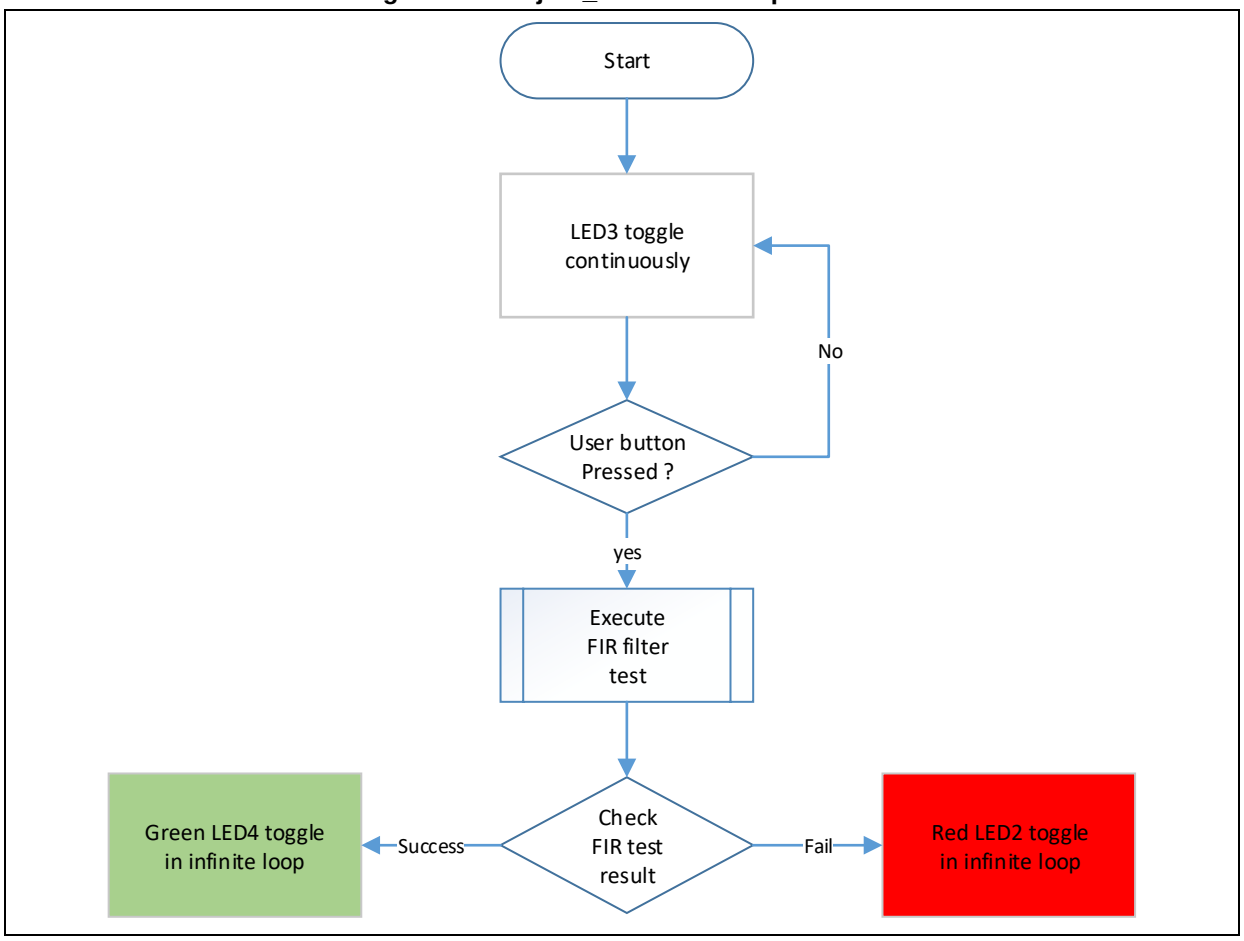
- *inputF3:* pointer to input signal data table;
- *outputF32*: pointer to output signal data table;
- *TEST_LENGTH_SAMPLES*: the number of signal samples to be processed.

3.5.4 Project_L1 execution process

Figure 35 shows the execution process of Project_L1:

- Start execution and LED3 keeps 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 35. Project_L1 execution process

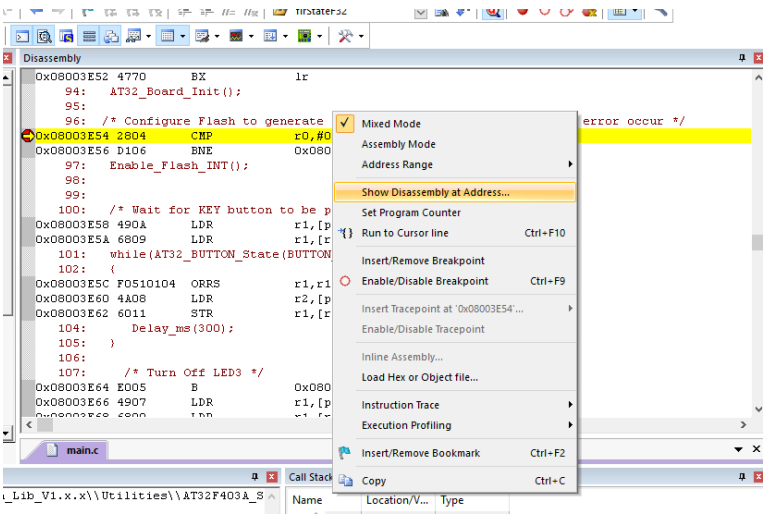


3.5.5 SLIB protection in debug mode

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.

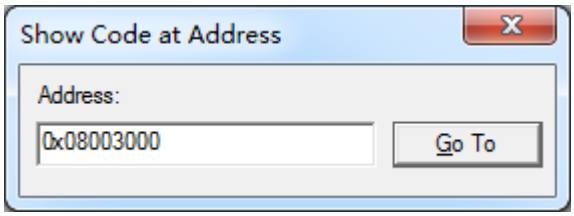
- Open Project_L1 and compile;
- Click “Start/Stop Debug Session” to enter debug mode;
- Right click in the “Disassembly” interface and select ”Show Disassembly at Address”, as shown in Figure 36.

Figure 36. Enter Show Disassembly at Address



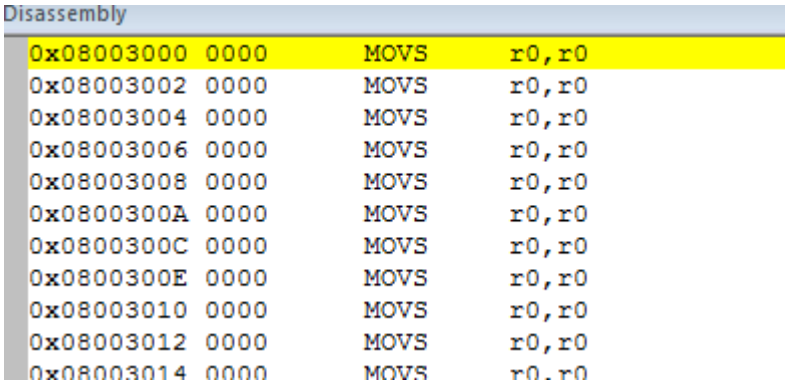
- Enter the address “0x08003000” of SLIB_INSTRUCTION start sector (sector 12);

Figure 37. Set Show Code at Address



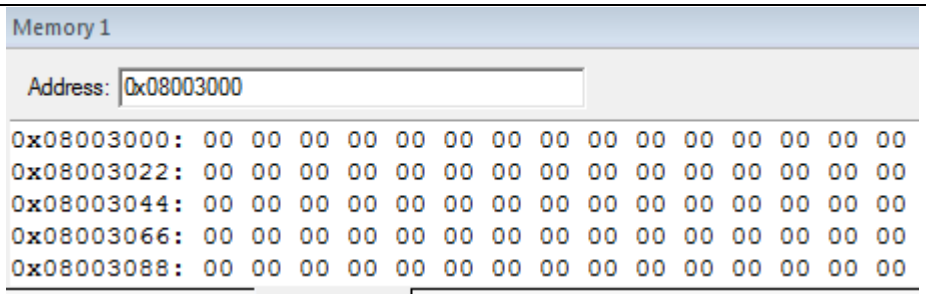
- As shown in Figure 38, codes from 0x08003000 are all 0x00000000;

Figure 38. View codes



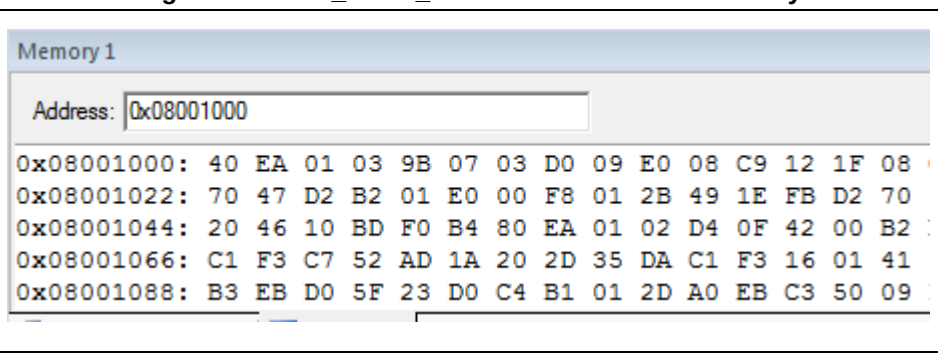
- Similarly, enter address “0x08001000” in “Memory” window, and codes are all 0x00, as shown in Figure 39.

Figure 39. View codes in Memory



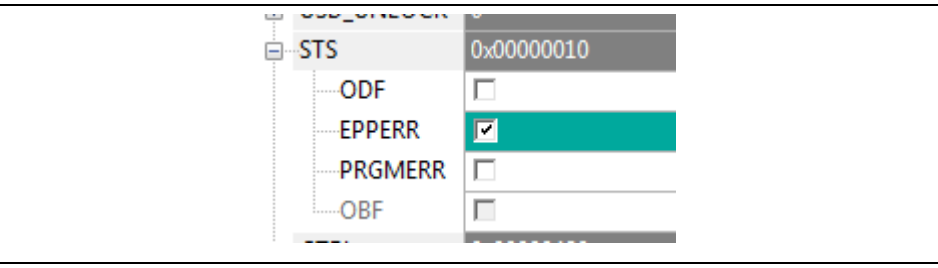
- In the "Memory" window, enter the address 0x08001000 of SLIB_READ_ONLY start sector (sector 1); this region is allowed to be read through D-Code bus, so that original values can be found, as shown in Figure 40.

Figure 40. SLIB_READ_ONLY start sector in Memory



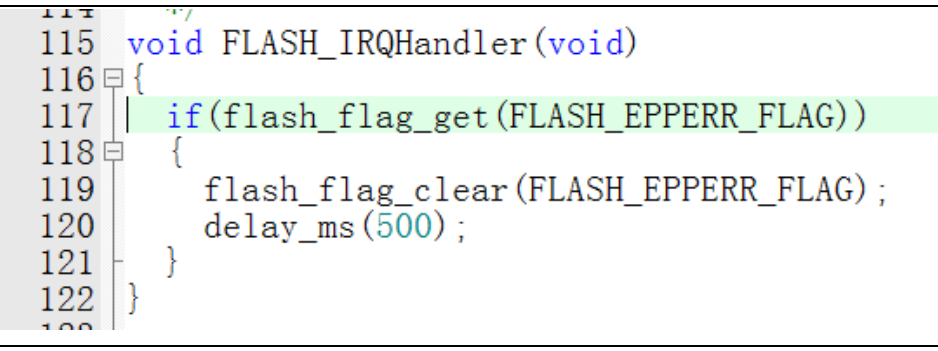
- Click to modify the value of 0x08003000 in the code, and a warning message will be issued by setting EPPERR=1 in the FLASH_STS register, indicating the protection is enabled.

Figure 41. SLIB write test



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

Figure 42. Write protection error interrupt



4 Integrate codes and download

After codes of the solution provider and end user are configured, 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.

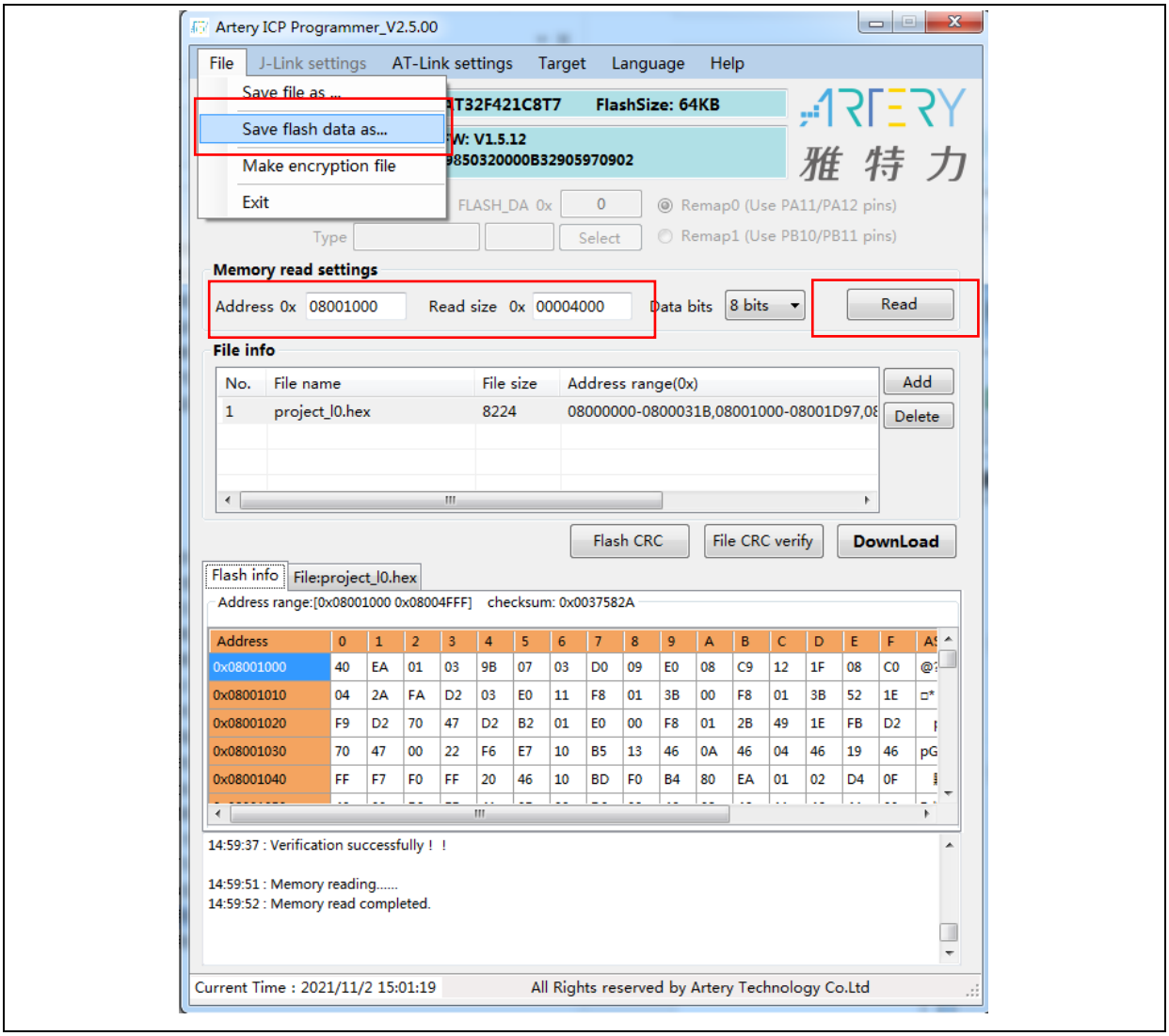
This operation involves offline downloading mode of AT-Link. For details, refer to operation manuals of ICP and AT-Link.

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

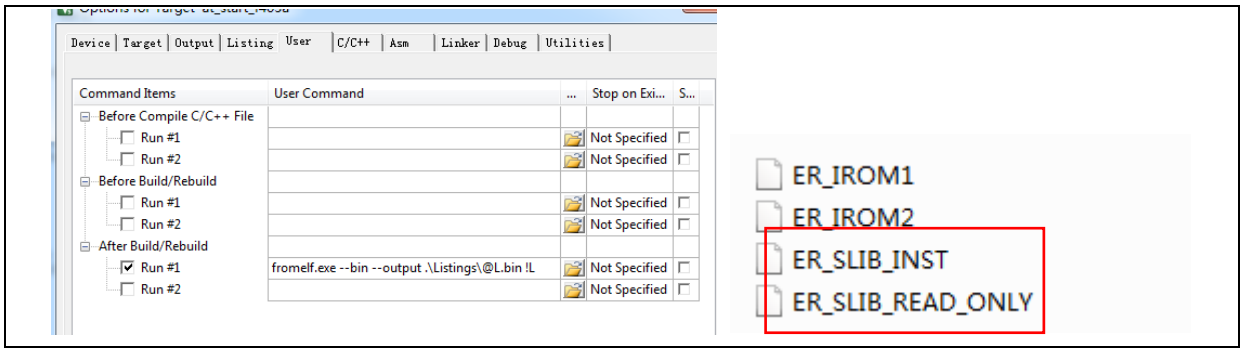
(1) **Method A:** The solution provider uses ICP tool to save the SLIB code in the compiled project as BIN or HEX file: download the complete project to MCU (do not configure SLIB and FAP), read the corresponding SLIB codes (0x08001000~0x08004FFF) by using the memory access function, and then click “File-Save Flash data as” to save codes as BIN or HEX file. In this example, it is named “slib.bin”, as shown in Figure 43.

Figure 43. Save SLIB codes



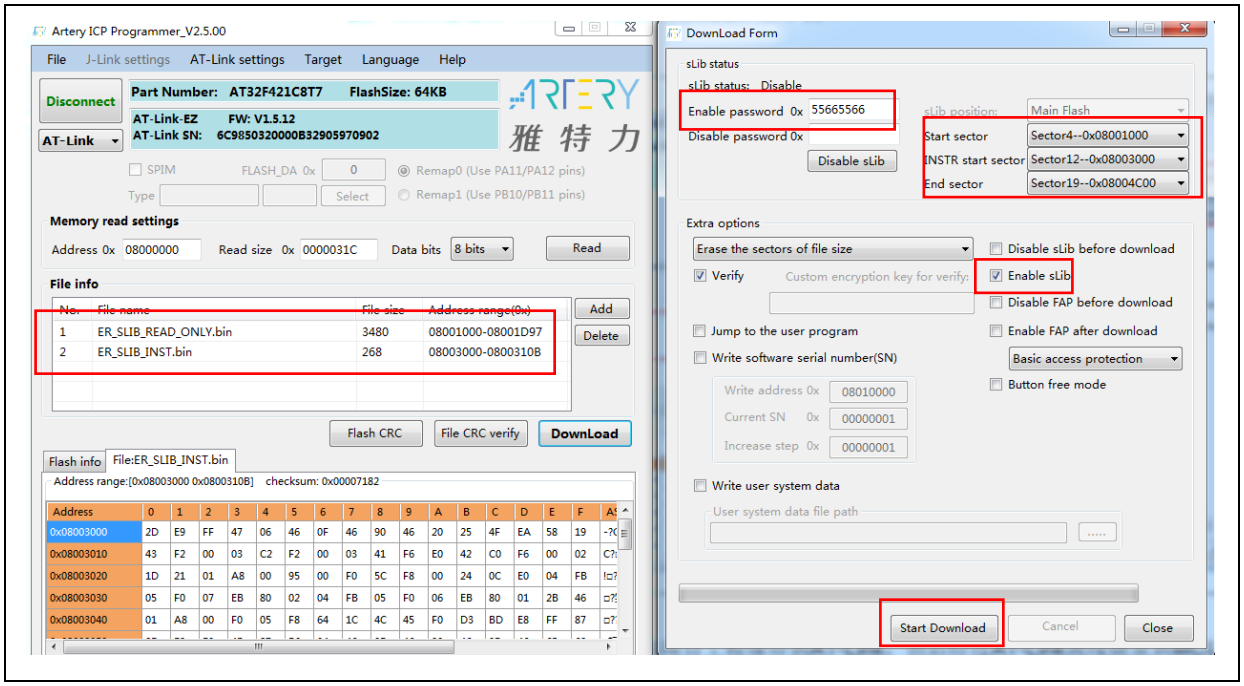
Method B: The solution provider uses the compiled project to generate a .bin file directly, and take the corresponding section in the SLIB area. For example, in the KEIL project, add "fromelf.exe --bin --output .\Listings\@L.bin !L" in the "user" option to generate a .bin file of the corresponding firmware, and add a suffix ".bin" to the SLIB area file. In this example, they are "ER_SLIB_INST.bin" and "ER_SLIB_READ_ONLY.bin", corresponding to the SLIB-INST file (0x08003000) and SLIB-READ-ONLY file (0x08001000), as shown in Figure 44.

Figure 44. Generate .bin file of SLIB code



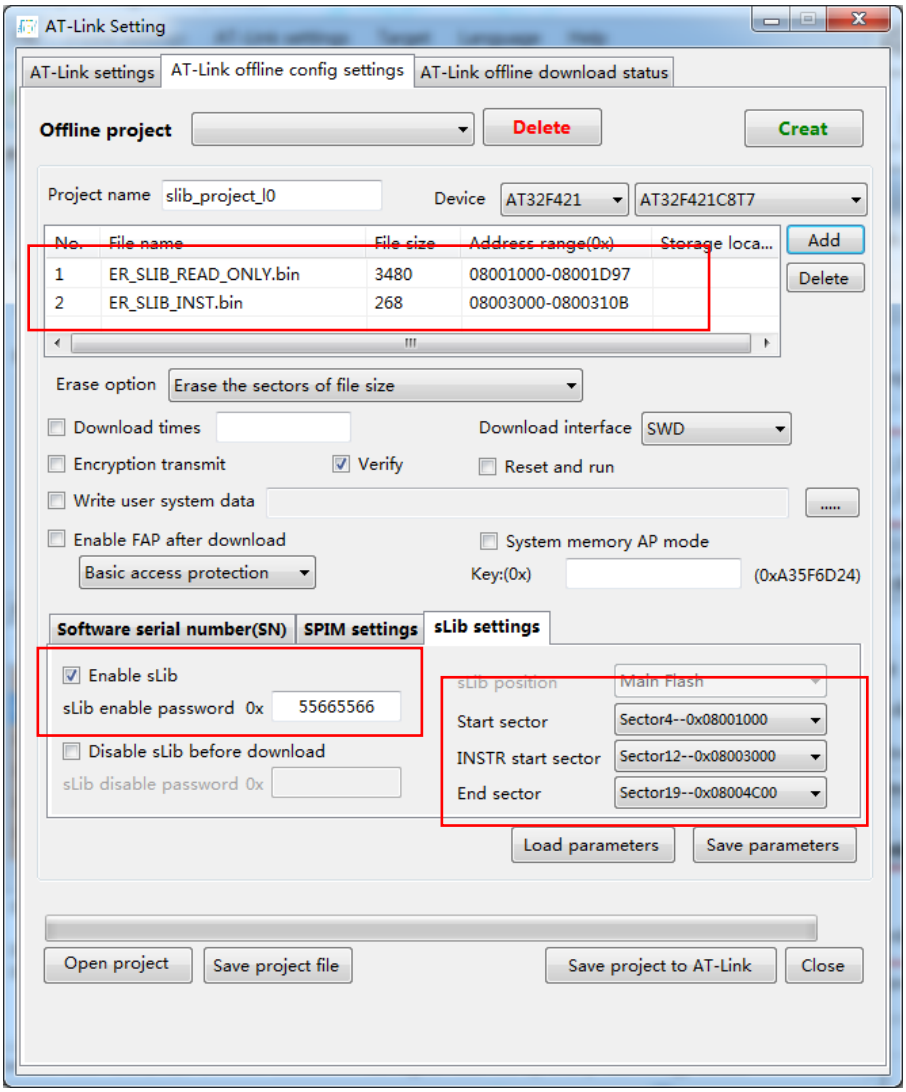
(2) Use ICP Programmer to program the .bin file to MCU, as shown in Figure 45.

Figure 45. Online programming to MCU in ICP



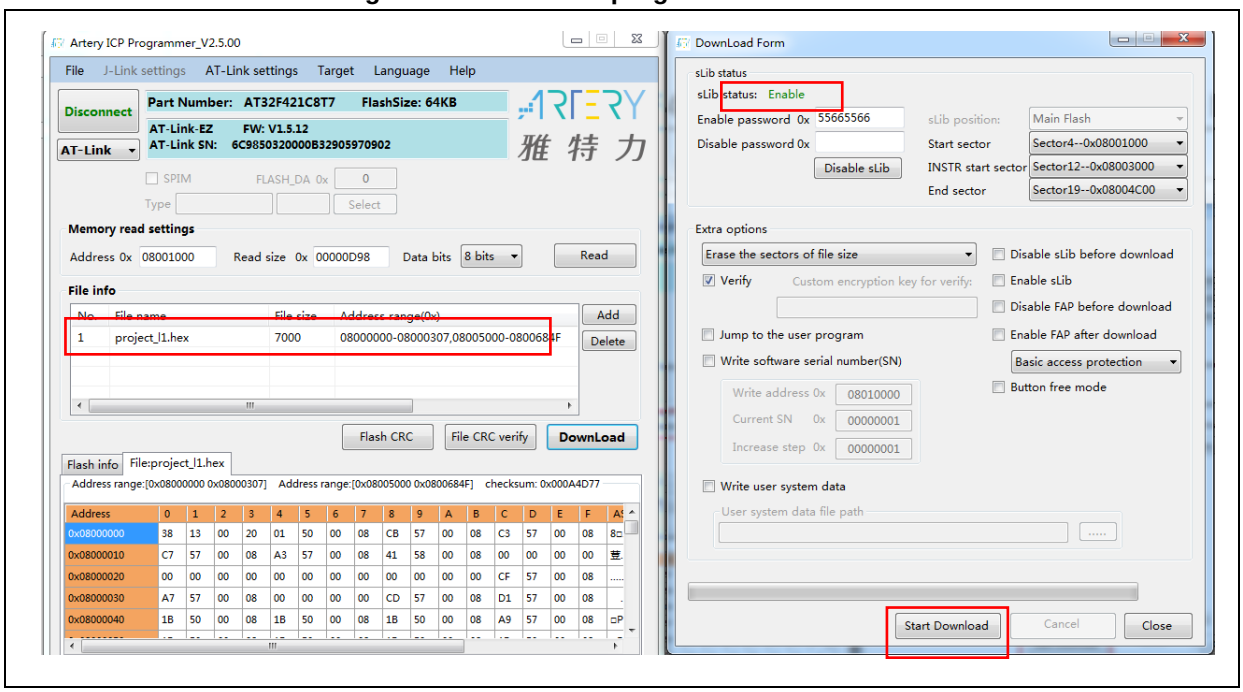
(3) End users also can use ICP Programmer to set an offline project and save it to AT-Link, and then complete offline programming to MCU through AT-Link, as shown in Figure 46.

Figure 46. 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 47.

Figure 47. End users program codes to MCU



4.2 Integrate and program codes

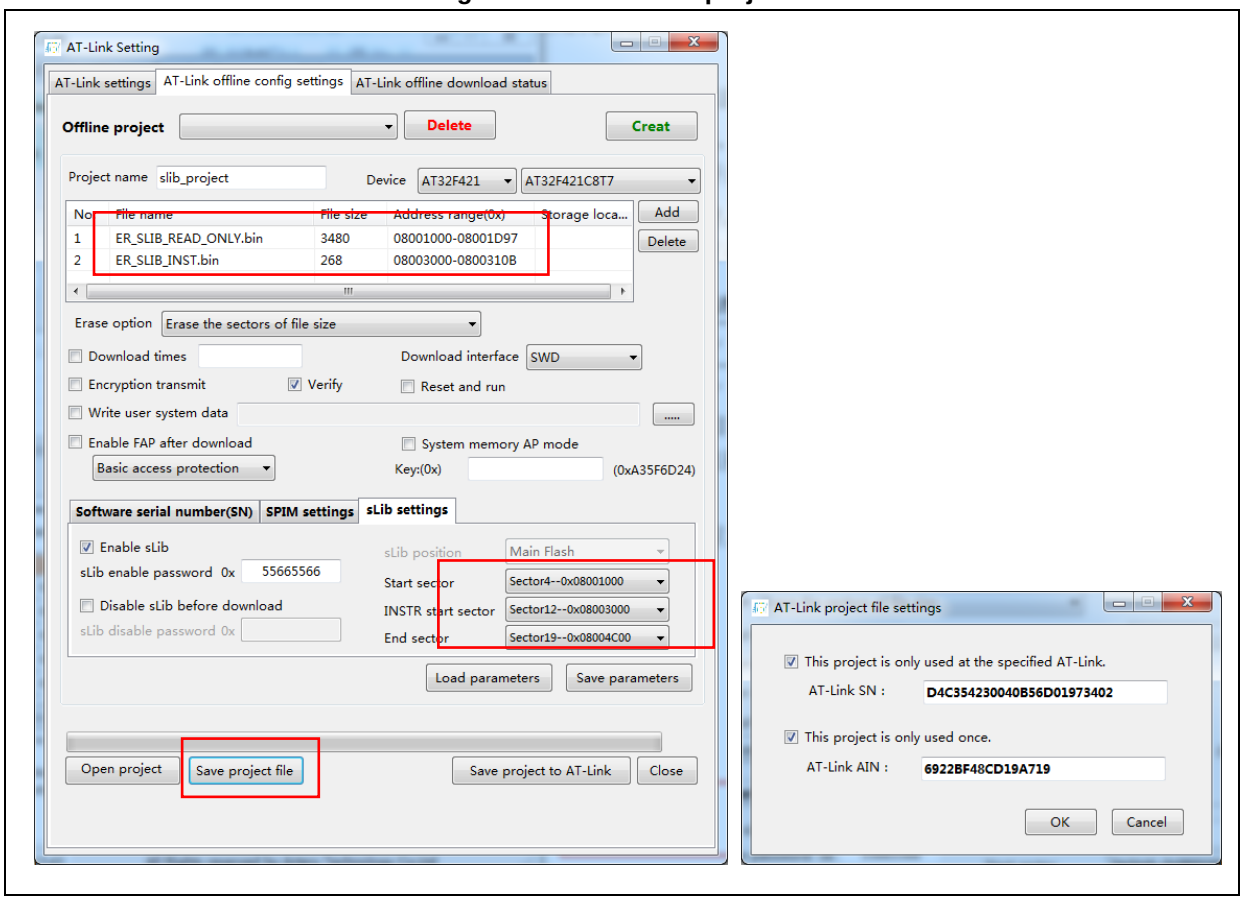
Integrate the SLIB code of solution provider and the end user application code to an offline project, and then download the integrated code to MCU through AT-Link offline programming. The process is as follows:

(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 as an encrypted slib.benc file 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.

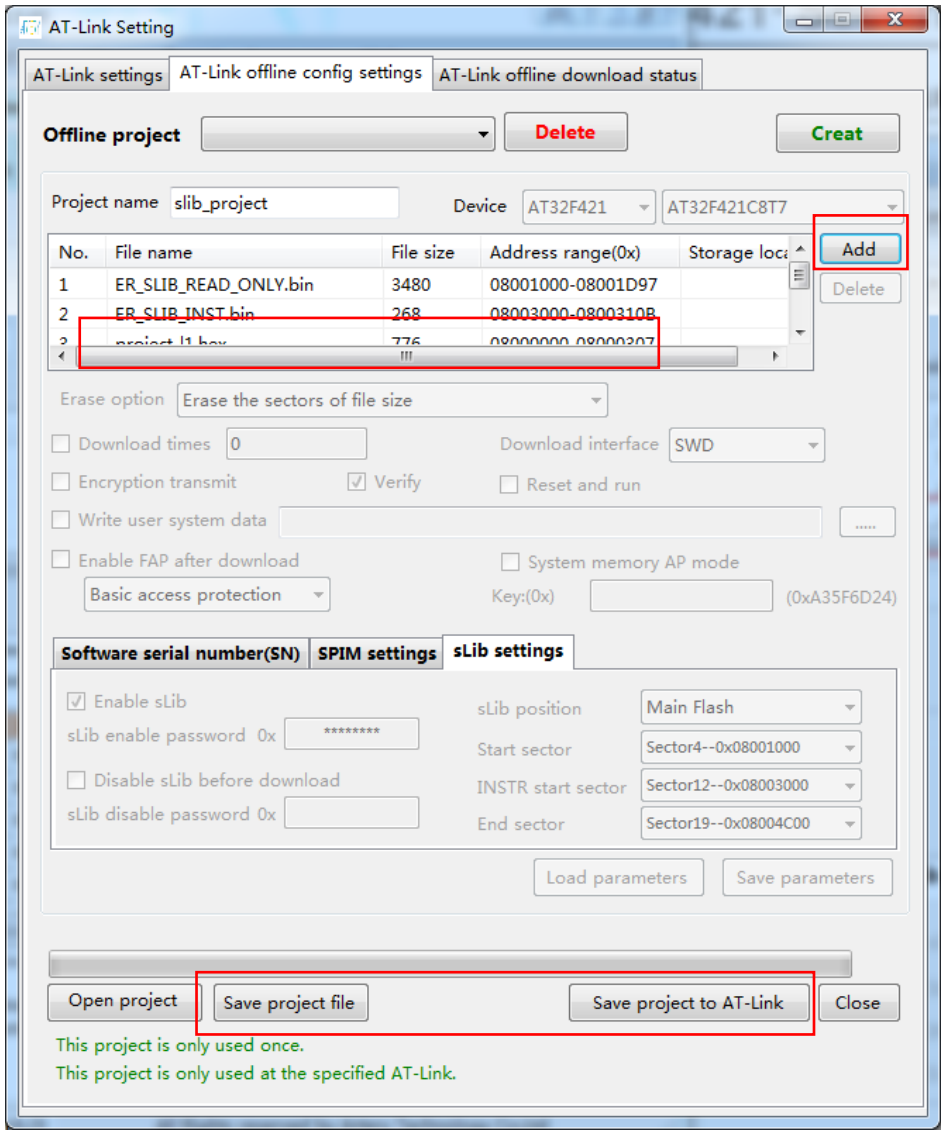
Figure 48. Set offline project



(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 49 shows how to add the project file.

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

Figure 49. Add project file



5 Revision history

Table 2. Document revision history

Date	Version	Revision note
2021.11.2	2.0.0	Initial release.
2023.03.21	2.0.1	Modified screenshots.

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.