AN0081
Application Note

AT32F435/437 Security Library Application Note

Introduction

This application note mainly introduces the security library (sLib) application principle of AT32F435/437 MCUs, operation methods and example projects.

Applicable products:

Part number	AT32F435 series
	AT32F437 series

Contents

1 Overview 7

2 Principles 8

2.1 sLib application principles 8

2.2 How to enable sLib protection 10

2.3 How to disable sLib protection 10

2.4 Set and run sLib 11

2.4.1 Don’t set interrupt vector table as SLIB_INSTRUCTION area 12

2.4.2 Relevance between sLib code and user code 12

3 Example code in sLib 15

3.1 Requirements 15

3.1.1 Hardware requirements 15

3.1.2 Software requirements 15

3.2 Example projects 15

3.3 sLib protected code: FIR low-pass filter 16

3.4 Project_L0: example for solution providers 17

3.4.1 Generate execute-only code 17

3.4.2 Set sLib addresses 20

3.4.3 How to enable sLib function 26

3.4.4 Project_L0 flow chart 27

3.4.5 How to generate header file and symbol definition file 29

3.5 Project_L1: example for end users 31

3.5.1 Create a user project 31

3.5.2 Add symbol definition files into project 32

3.5.3 Call sLib functions 34

3.5.4 Project_L1 flow chart 34

3.5.5 sLib protection in debug mode 35

4 Integrate and download codes of solution provider and user 38

4.1 Write code separated on solution provider and end user 38

4.2 Combine solution provider code with end user code 41

5 Revision history 44

List of tables

Table 1. AT32F435/437 series Flash memory size 9
Table 2. Document revision history 44

List of figures

Figure 1. Flash memory map with security library 9
Figure 2. Example of literal pool (1) 11
Figure 3. Example of literal pool (2) 12
Figure 4. Example of sLib function calls a function in the user code area 13
Figure 5. Example of user-defined function 14
Figure 6. Flow chart example 15
Figure 7. Application diagram 16
Figure 8. FIR low-pass filter 16
Figure 9. Keil enters Option window 17
Figure 10. Choose Execute-only Code in Keil 18
Figure 11. Enter Option interface in IAR 18
Figure 12. IAR C/C++ window 19
Figure 13. Enter Properties in AT32 IDE 19
Figure 14. AT32 IDE Miscellaneous settings 20
Figure 15. Flash memory map and RAM segment in the example 20
Figure 16. Linker settings in Keil 21
Figure 17. Keil scatter modification 22
Figure 18. SLIB address definition in icf file 23
Figure 19. Address assignment in icf file 23
Figure 20. Modify RAM and ROM ranges in icf file 23
Figure 21. Modify RAM and ROM ranges in ld file 24
Figure 22. Specify areas in ld file 24
Figure 23. Add the modified ld file 25
Figure 24. Add keywords to avoid compiling error 25
Figure 25. ICP Programmer operation 26
Figure 26. Set parameters in Download Form 27
Figure 27. Project_L0 flow chart 28
Figure 28. Keil Misc controls option 29
Figure 29. Reduced fir_filter_symbol.txt 29
Figure 30. Set IAR Build Actions option 30
Figure 31. Edit steering_file.txt 30
Figure 32. Add post-build in AT32 IDE 31
Figure 33. Modified scatter file 32

Figure 34. Modified icf file 32
Figure 35. Add files generated in project_l0 32
Figure 36. Add symbol definition file in Keil 33
Figure 37. Change symbol definition file to Object file 33
Figure 38. Add symbol definition file in IAR 33
Figure 39. Add ld file path in AT32 IDE environment 34
Figure 40. Project_L1 flow chart 35
Figure 41. “Show Disassembly at Address” window 36
Figure 42. “Show Code at Address” setting 36
Figure 43. View code 36
Figure 44. View code in Memory window 36
Figure 45. View SLIB_READ_ONLY start sector in Memory 37
Figure 46. SLIB write protection test 37
Figure 47. Write protection error interrupt 37
Figure 48. Generate bin file of SLIB code 38
Figure 49. Online programming to MCU via ICP 39
Figure 50. Offline programming to MCU via AT-Link 40
Figure 51. End user programs code to MCU 41
Figure 52. Create offline project 42
Figure 53. Add project file 43

1 Overview

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

In response to this demand, AT32F435/437 series is equipped with a security library, known as sLib, with the aim of preventing important IP codes from being altered or read by end user program, so as to safeguard the rights of solution providers.

Here this document will detail the application logics behind AT32F435/437 series' security library and its software usage

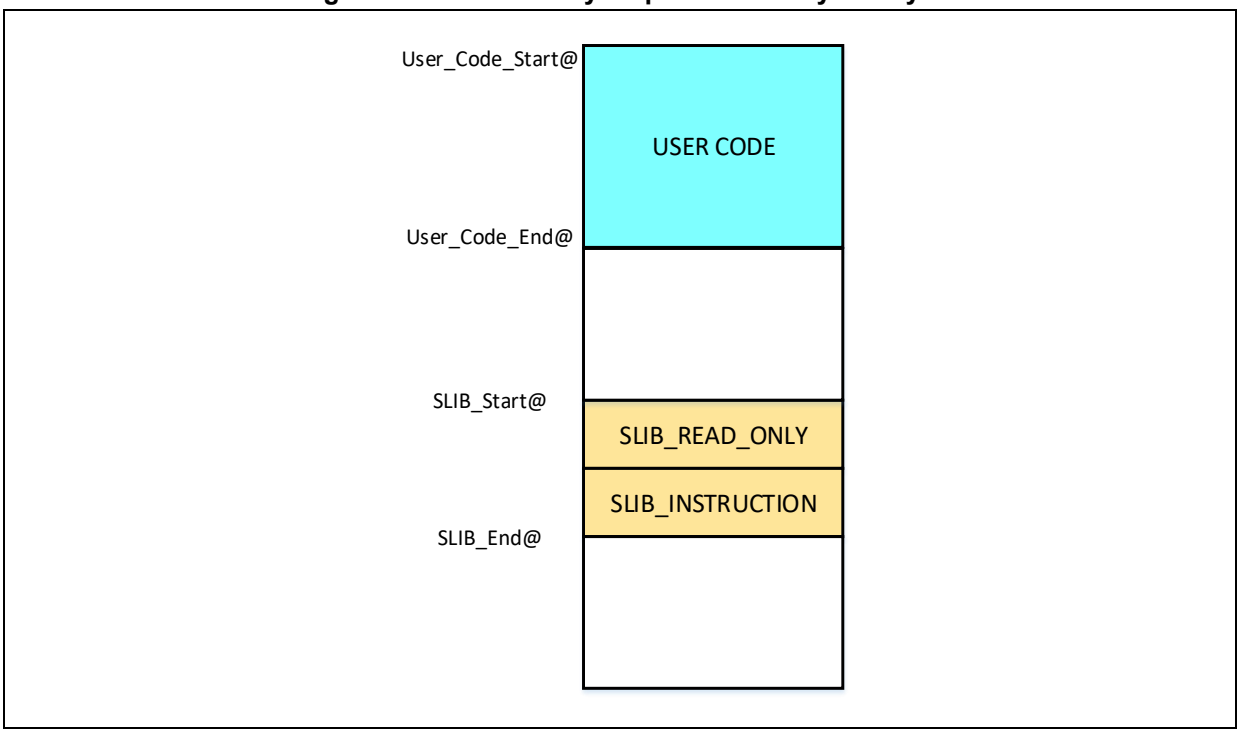
2 Principles

2.1 sLib application principles

- Any part of Flash memory can be designated as a security library (sLib) with password. This sLib is used for storing critical algorithms by solution providers while the remaining memory area can be used for secondary development by end users.
- 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 but cannot be programmed.
- Program codes in the SLIB_INSTRUCTION area can only be fetched (only executable) by MCU through I-Code. They cannot be read out by reading access (including ISP/ICP/debug mode or boot from internal RAM) via D-Code, for accessing SLIB_INSTRUCTION by reading operation will return all 0xFF.
- Program codes and data in sLib 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.
- Program codes and data in sLib are not erased when end users perform a mass erase on the main Flash memory.
- After the sLib feature is enabled, users can also unlock this protection by writing a correct password in the SLIB_PWD_CLR register. Once sLib is unlocked, MCU will erase the whole main memory, including sLib. This kind of design is to protect program codes against leakage even if the password set by solution providers is leaked.

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

Figure 1. Flash memory map with security library



The 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 AT32F435/437 series MCUs. In this example, the sLib can be set to the last position of the minimum zero-wait area for the specified device.

Table 1. AT32F435/437 series Flash memory size

Part number	Internal Flash size (byte)	Sector size (byte)	Configurable range
AT32F435xC AT32F437xC	256K	2K	Sector 0 ~ 127 (0x08000000 ~ 0x0803FFFF)
AT32F435xG AT32F437xG	1024K	2K	Sector 0 ~ 511 (0x08000000 ~ 0x080FFFFF)
AT32F435xM AT32F437xM	4032K	4K	Sector 0 ~ 1007 (0x08000000 ~ 0x083EFFFF)

2.2 How to enable sLib protection

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

Follow the procedures below to enable Flash memory sLib:

- Check the OBF bit of the FLASH_STS and FLASH_STS2 registers to confirm that there is no other ongoing programming operation;
- Write 0xA35F6D24 to the SLIB_UNLOCK register to unlock security library;
- Check if unlock operation is successful by checking the SLIB_ULKF bit in the SLIB_MISC_STS register;
- Set the sectors to be protected, including the start and end addresses of sLib, through the SLIB_SET_RANGE0 register;
- Set the sectors to be protected, including the start and end addresses of SLIB_INSTRUCTION and the sLib enable bit, through the SLIB_SET_RANGE1 register;
- Wait until the OBF bit becomes “0”;
- Set a sLib password through the SLIB_SET_PWD register;
- Wait until the OBF bit becomes “0”;
- Program codes to be stored into sLib;
- Perform system reset, and reload sLib setting word;
- Read the SLIB_STS0/STS1/STS2 register to verify the security library settings.

Special attention to be paid to the following:

- It is allowed to set sLib in the main Flash memory only; see Table 1 for the configurable range.
- The security library code must be programmed by sectors, with its start address aligned with the address of main Flash memory.
- Interrupt vector table as a data type is typically placed on the first sector (sector 0) of Flash memory. As a result, sector 0 should not be set as SLIB_INSTRUCTION.

For details on sLib setting register, please refer to *AT32F435/437 Series Reference Manual*.

For the program code on enabling sLib, please refer to “slib_enable()” in the main.c of project_l0 example case. Besides, it is also possible to set sLib through ICP or ISP programming tool, which will be described in the subsequent sections.

2.3 How to disable sLib protection

After sLib feature is enabled, it is possible for users to unlock it by writing the previously set password in the SLIB_PWD_CLR register. Once sLib is disabled, the device will perform mass erase on the main Flash memory, including erasing contents in the sLib area.

Follow the procedures below to disable sLib:

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

2.4 Set and run sLib

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

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

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

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

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

Figure 2. Example of literal pool (1)

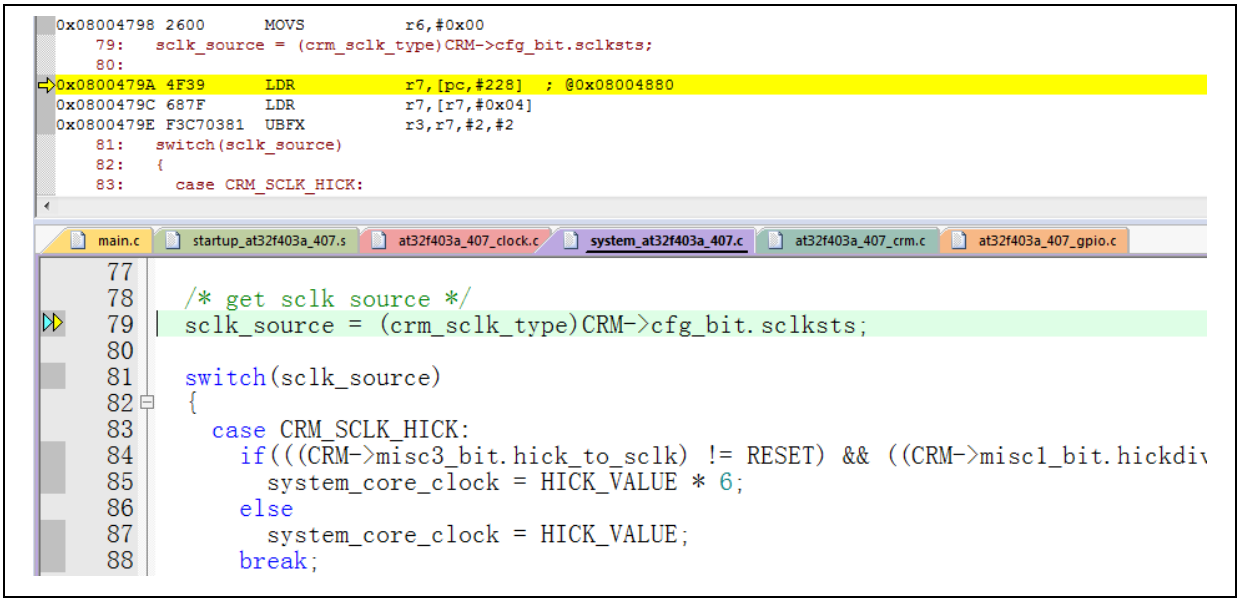
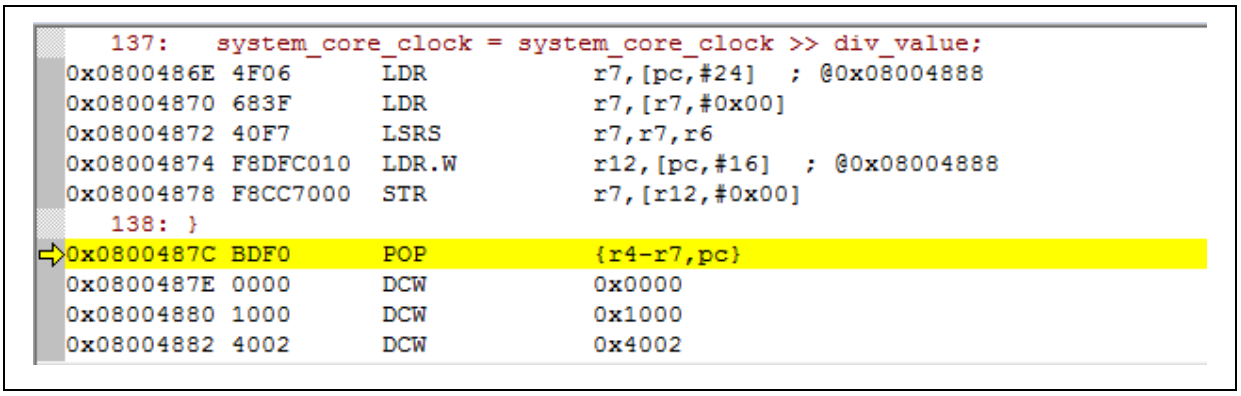


Figure 3. Example of literal pool (2)



2.4.1 Don’t set interrupt vector table as SLIB_INSTRUCTION area

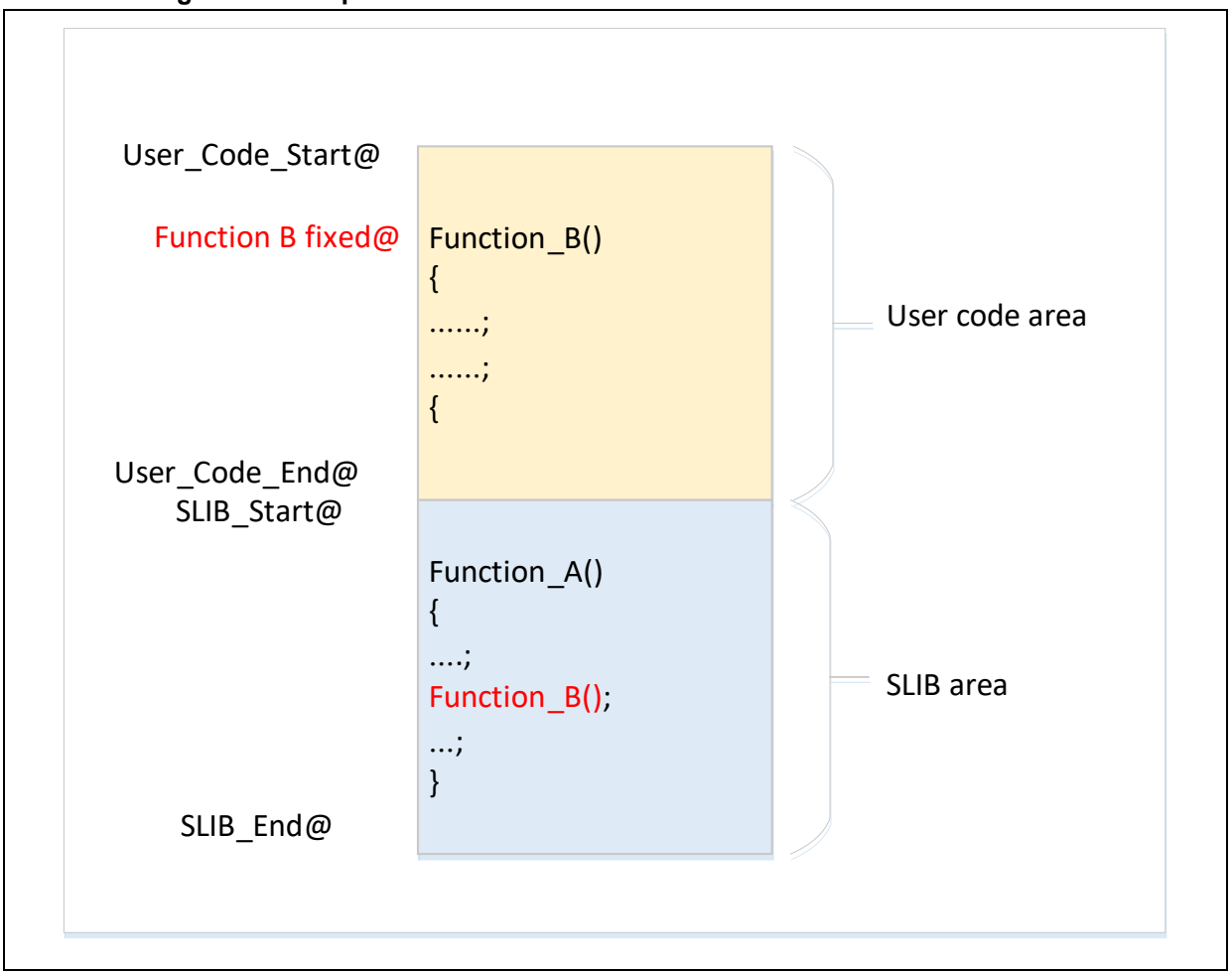
Interrupt vector table contains entry addresses of all interrupt handlers which are readable by MCU through D-Code. In most cases, the table is located at sector 0 with start address 0x08000000 in Flash memory. Therefore, the following rule should be respected when setting sLib instruction area.

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

2.4.2 Relevance between sLib code and user code

IP-code protected by sLib is able to call functions from a function library in the user code area. In this scenario, IP-Code will also carry the addresses of such functions, allowing PC (program counter) to jump to them while executing IP-Code. Once sLib is enabled, such functions’ addresses are unchangeable. This means that these addresses in the user code area must be fixed or remain unchanged; otherwise, PC will jump to a wrong address and fail to work. Based on this, before setting sLib, it is necessary to place all functions relating to IP-Code in sLib to avoid such problem. Figure 4 gives an example on how a protected Function_A() calls Function_B() in user code area.

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



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

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

Figure 5. Example of user-defined function

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

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

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

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

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

}


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

  return (s);
}
```

3 Example code in sLib

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

The AT32F435 and AT32F437 series MCUs have the same sLib feature. In this application note, the AT32F437 series is used for demonstration.

3.1 Requirements

3.1.1 Hardware requirements

- AT-START-F437 evaluation board with embedded AT32F437ZMT7 microcontroller
- AT-Link debugger which is used to debug programs

3.1.2 Software requirements

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

3.2 Example projects

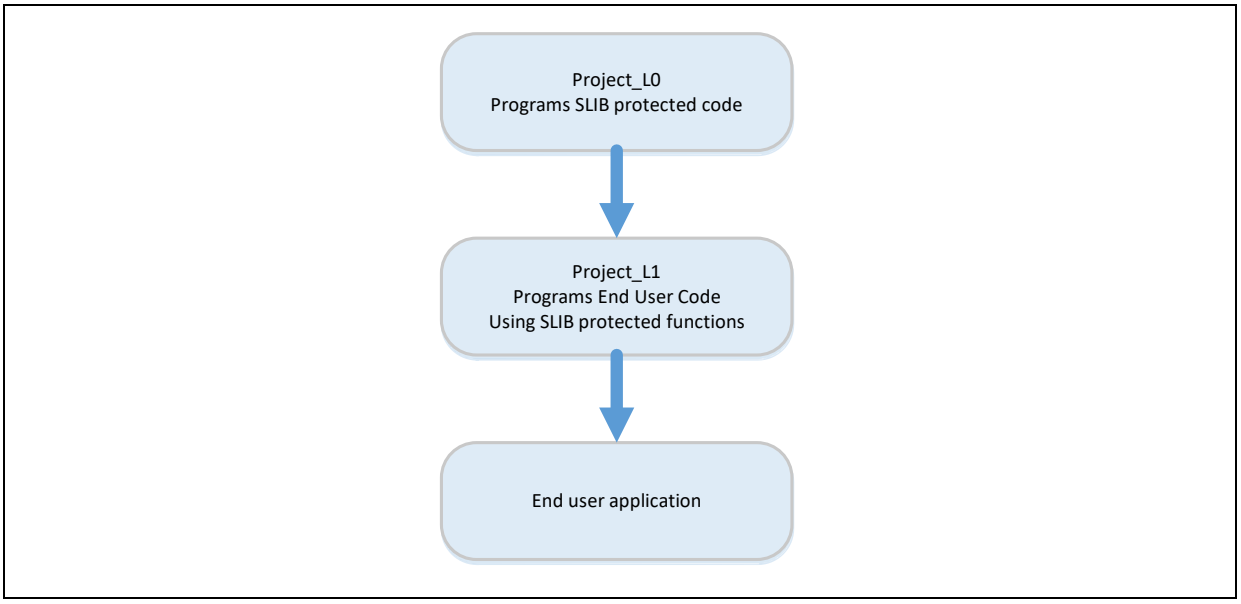
This application note offers two example projects demonstrating how software provider develops IP-Code to meet end user application requirements.

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

Algorithms developed in Project_L0 will be downloaded and programmed into AT32F437 device in advance with sLib function being enabled. Meanwhile, the following information are also available to end user programs.

- Main Flash memory map, indicating the area owned by sLib and the area that can be developed by users.
- Header files containing algorithm function definitions, for user programs to call.
- Symbol definition file, containing the addresses of IP-Code functions, for end users to call. See Figure 6 below for reference.

Figure 6. Flow chart example



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

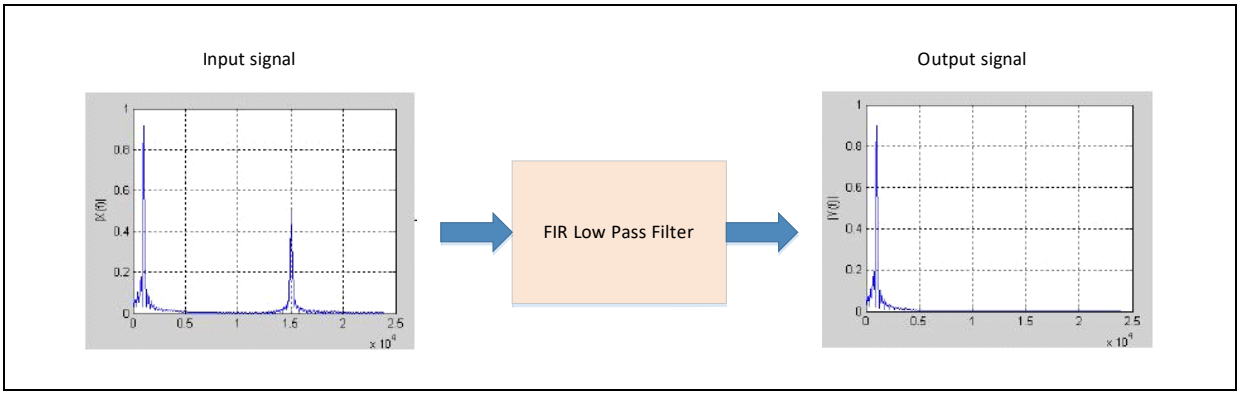
Figure 7. Application diagram

3.3 sLib protected code: FIR low-pass filter

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

In the example, the input signal of low-pass filter is from two sine wave signals with 1KHz and 15KHz respectively. The cut-off frequency is 6KHz for this low-pass filter. After going through lowpass filter, 15KHz signal is filtered, leaving only 1KHz sine wave output. Figure 8 shows a diagram of FIR low-pass filter function.

Figure 8. FIR low-pass filter



The following CMSIS DSP functions and files will be used:

- arm_fir_init_f32()

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

- arm_fir_f32()

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

- FIR_lowpass_filter()

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

- fir_coefficient.c

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

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

3.4 Project_L0: example for solution providers

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

- Compile algorithm-related functions as execute-only ones.
- Place algorithm code in the specified sector (referred to as sector A) in the main Flash memory.
- Place coefficients of filter functions in the specified sector (referred to as sector B) in the main Flash memory.
- Execute “FIR_lowpass_filter()” in the main program to verify.
- After successful verification, set sector A as SLIB_INSTRUCTION area and sector B as SLIB_READ_ONLY area. This step can be done by calling “slib_enable()” in the main program of this example, or by using Artery ICP Programmer tool (recommended).
- Generate header files and symbol definition files used for calling low-pass filter functions by end user programs.

3.4.1 Generate execute-only code

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

For more information on literal pools and branch table, please refer to Section 2.4.

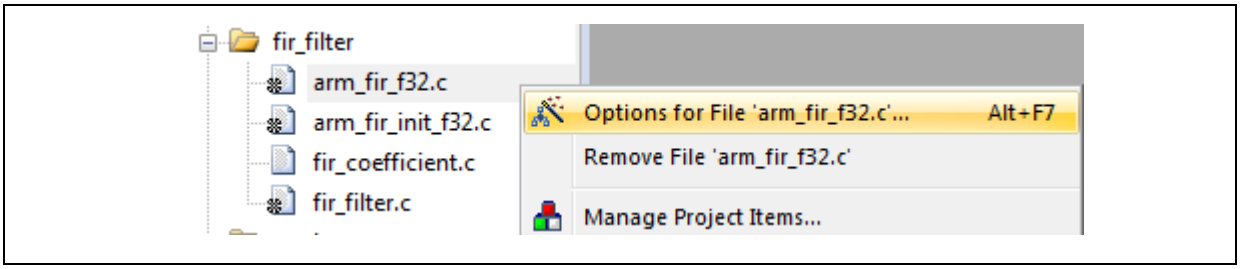
Set as follows:

Keil® µvision: use Execute-only Code option

Proceed as follows:

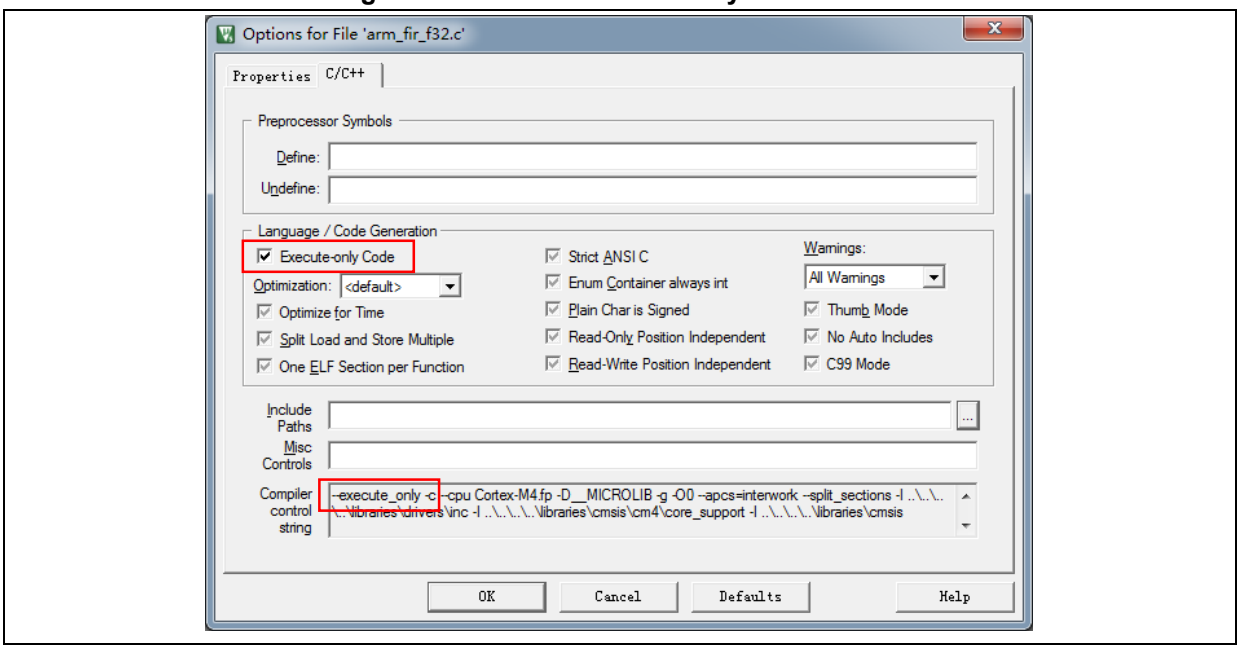
- Choose a C file group or an individual C file. In the example, the would-be protected C files are included in the “fir_filter” group.
- Right click and choose corresponding file, for example, Option for File ‘arm_fir_f32.c’, as shown in Figure 9.

Figure 9. Keil enters Option window



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

Figure 10. Choose Execute-only Code in Keil



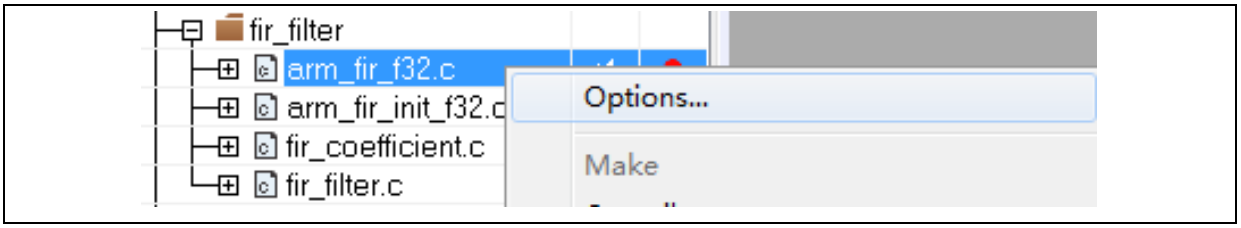
- The *arm_fir_f32.c*, *arm_fir_init_f32.c* and *fir_filter.c* files are in the SLIB_INSTRUCTION area, and these files need to be set as generating execute-only code.

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

Proceed as follows:

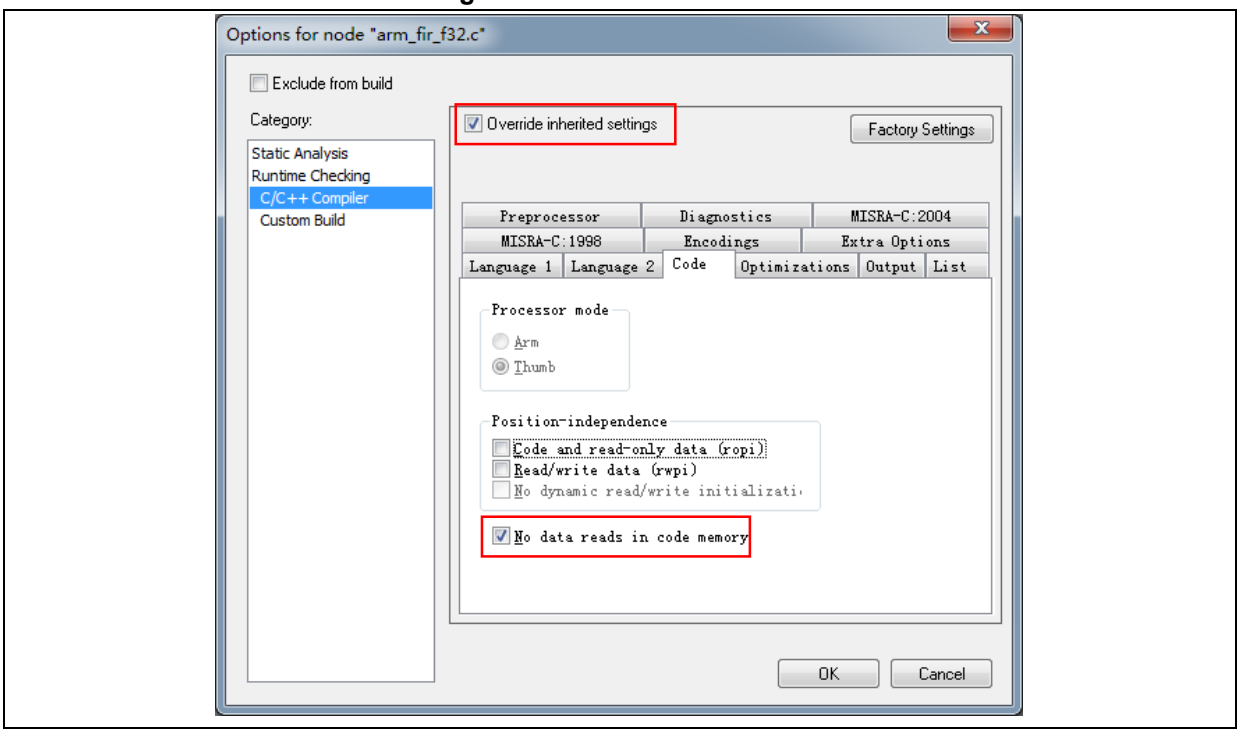
- Select the corresponding file in the “fir_filter” group; right click and select “Option”.

Figure 11. Enter Option interface in IAR



- Go to "C/C++" interface and tick “Override inherited settings” and “No data read in code memory”.

Figure 12. IAR C/C++ window



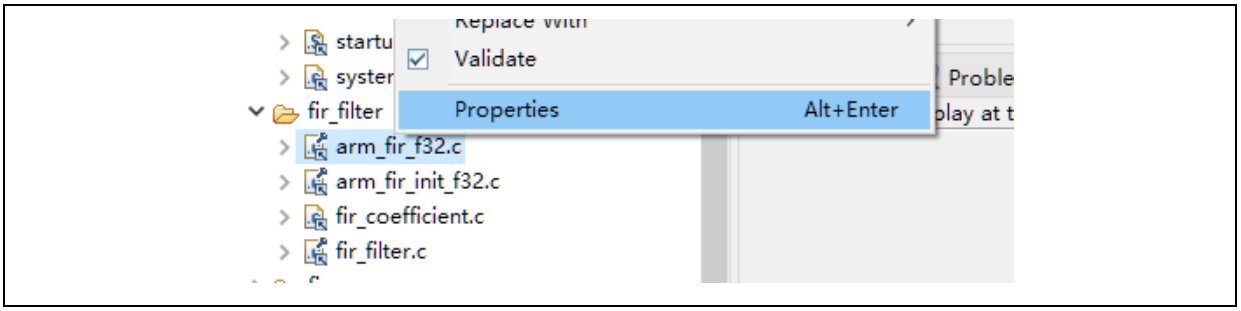
- The *arm_fir_f32.c*, *arm_fir_init_f32.c* and *fir_filter.c* files are in the SLIB_INSTRUCTION area, and these files need to be set as generating execute-only code.

AT32 IDE: Add keywords to "Other compiler flags"

Proceed as follows:

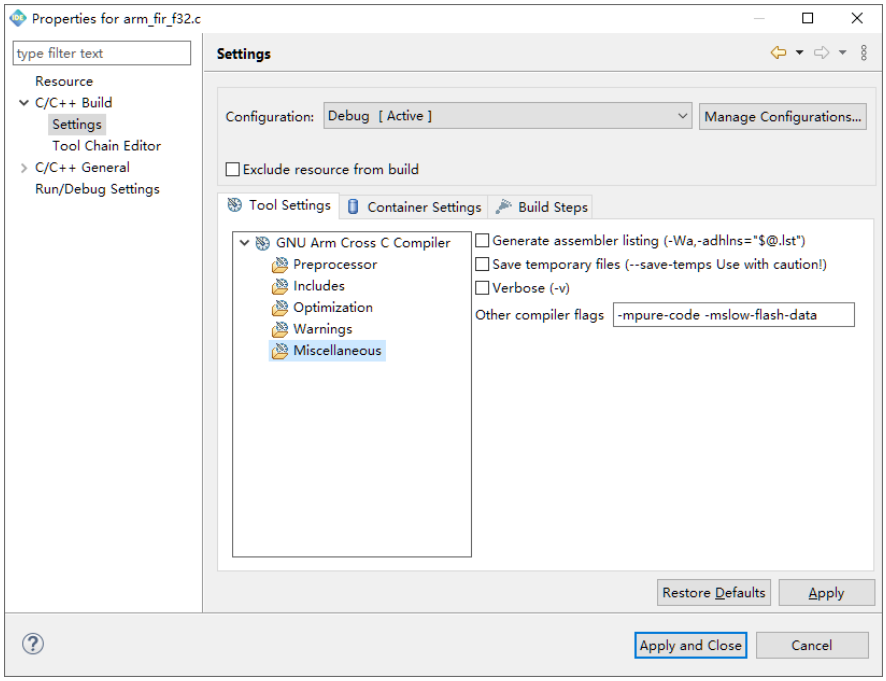
- Select the corresponding file in the "fir_filter" group; right click and select "Properties".

Figure 13. Enter Properties in AT32 IDE



- Go to C/C++ Build->Settings->GNU ARM Cross C Complier->Miscellaneous, enter keywords "-mpure-code" and "-mslow-flash-data" into "Other compiler flags", and then click "Apply and Close".

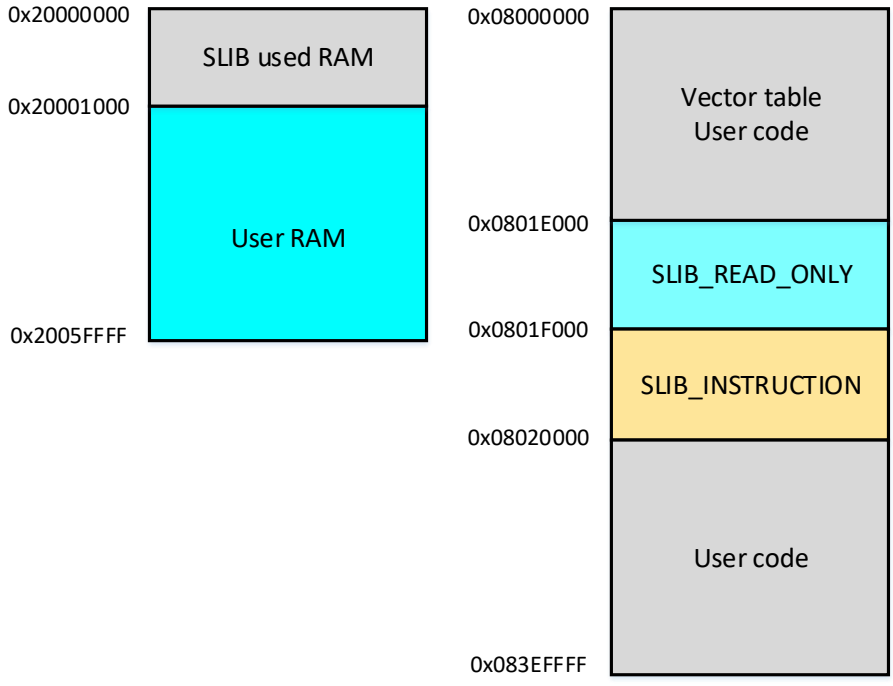
Figure 14. AT32 IDE Miscellaneous settings



3.4.2 Set sLib addresses

As mentioned in the previous sections, the sector 0 of Flash memory is used to store interrupt vector tables. Figure 15 below shows Flash memory map and RAM range distribution. The RAM segment is mainly aimed at preventing the use of the same RAM by sLib-protected code and user code.

Figure 15. Flash memory map and RAM segment in the example

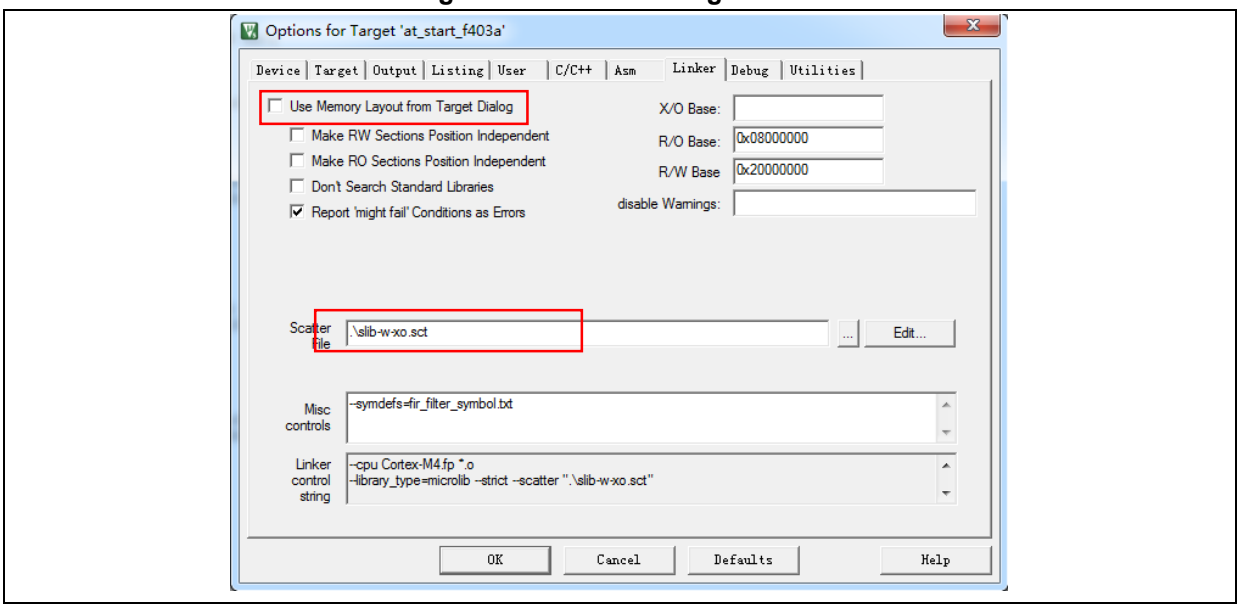


Keil® µvision's scatter file

Proceed as follows:

- Go to Project → Options for Target→Linker, cancel "Use memory layout from Target Dialog" option, and then click "Edit" to open "slib-w-xo.sct" for modification, as shown in Figure 16.

Figure 16. Linker settings in Keil



- Open "scatter file", place an object file of the code which needs to be put in the SLIB_INSTRUCTION area in to a dedicated load area named "LR_SLIB_INSTRUCTION", and change its mark to "execute-only (+XO)". Place the area occupied by SLIB_READ_ONLY to a dedicated load area named "LR_SLIB_READ_ONLY" to avoid the compiler compiling other non-IP-Code functions to the SLIB area. The RW_IRAM1 is assigned 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 17. Keil scatter modification

```
LR_IROM1 0x08000000 0x01E000  {    ; load region size_region
  ER_IROM1 0x08000000 0x01E000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
  }

  RW_IRAM1 0x20000000 0x00001000  {  ; RAM used for slib code
   fir_filter.o (+RW +ZI)
  }

  RW_IRAM2 0x20001000 0x0005F000  {  ; user RW data
   .ANY (+RW +ZI)
  }

}

LR_SLIB_READ_ONLY 0x0801E000 0x00001000  {    ; sLib read-only area
  ER_SLIB_READ_ONLY 0x0801E000 0x00001000  {
   fir_coefficient.o (+RO)
  }
}

LR_SLIB_INSTRUCTION 0x0801F000 0x00001000  {    ; slib instruction area
  ER_SLIB_INSTRUCTION 0x0801F000 0x00001000  {  ; load address = execution address
 arm_fir_init_f32.o (+XO)
 arm_fir_f32.o (+XO)
 fir_filter.o (+XO)
  }
}

LR_IROM2 0x08020000 0x003D0000  {    ; user code area
  ER_IROM2 0x08020000 0x003D0000  {  ; load address = execution address
   .ANY (+RO)
  }
}
```

- With regard to the RAM used by IP-Code and the constant arrangement address used by the data security library FIR low-pass filter functions, in addition to above-mentioned method, it is also possible to use Keil's "__attribute__((at(address)))" descriptor to place variables or constants at a fixed address.

IAR's ICF file

Proceed as follows:

- Open the "icf" file under "\project_l0\IAR_V8.2\", and add three new load areas as shown in Figure 18. The SLIB_RAM is reserved for the algorithm functions.

Figure 18. SLIB address definition in icf file

```
/* SLIB read-only area */
define symbol __ICFEDIT_region_SLIB_READ_ONLY_start__ = 0x0801E000;
define symbol __ICFEDIT_region_SLIB_READ_ONLY_end__   = 0x0801EFFF;

/* SLIB instruction area */
define symbol __ICFEDIT_region_SLIB_INST_start__ = 0x0801F000;
define symbol __ICFEDIT_region_SLIB_INST_end__   = 0x0801FFFF;

define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__   = 0x2005FFFF;

/* SLIB RAM region */
define symbol __ICFEDIT_region_SLIB_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_SLIB_RAM_end__   = 0x20000FFF;
```

- 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 also reserved.

Figure 19. Address assignment in icf file

```
/* Reserved 0x0801E000 ~ 0x0801FFFF 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 0x2005F000 ~ 0x2005FFFF 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__];
```

- Modify the RAM and ROM used by IP-Code in “icf” file, as shown in Figure 20.

Figure 20. Modify RAM and ROM ranges in icf file

```
place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };

place in ROM_region   { readonly };

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

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


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

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

- With regard to the RAM used by IP-Code and the constant arrangement address used by the data security library FIR low-pass filter functions, in addition to above-mentioned method, it is also possible to use IAR’s @ descriptor to place the variables or constants at a fixed address.

AT32 IDE: ld file

Proceed as follows:

- Modify “ld” file and specify the area required for sLib, as shown in Figure 21.

Figure 21. Modify RAM and ROM ranges in ld file

```
MEMORY
{
FLASH_1 (rx)       : ORIGIN = 0x08000000, LENGTH = 120K
SLIB_READ_ONLY (x) : ORIGIN = 0x0801E000, LENGTH = 4K
SLIB_INST (r)      : ORIGIN = 0x0801F000, LENGTH = 4K
FLASH_2 (rx)       : ORIGIN = 0x08020000, LENGTH = 3904K
SLIB_RAM (xrw)     : ORIGIN = 0x20000000, LENGTH = 4K  /* used for SLIB code */
RAM (xrw)          : ORIGIN = 0x20001000, LENGTH = 380K
}
```

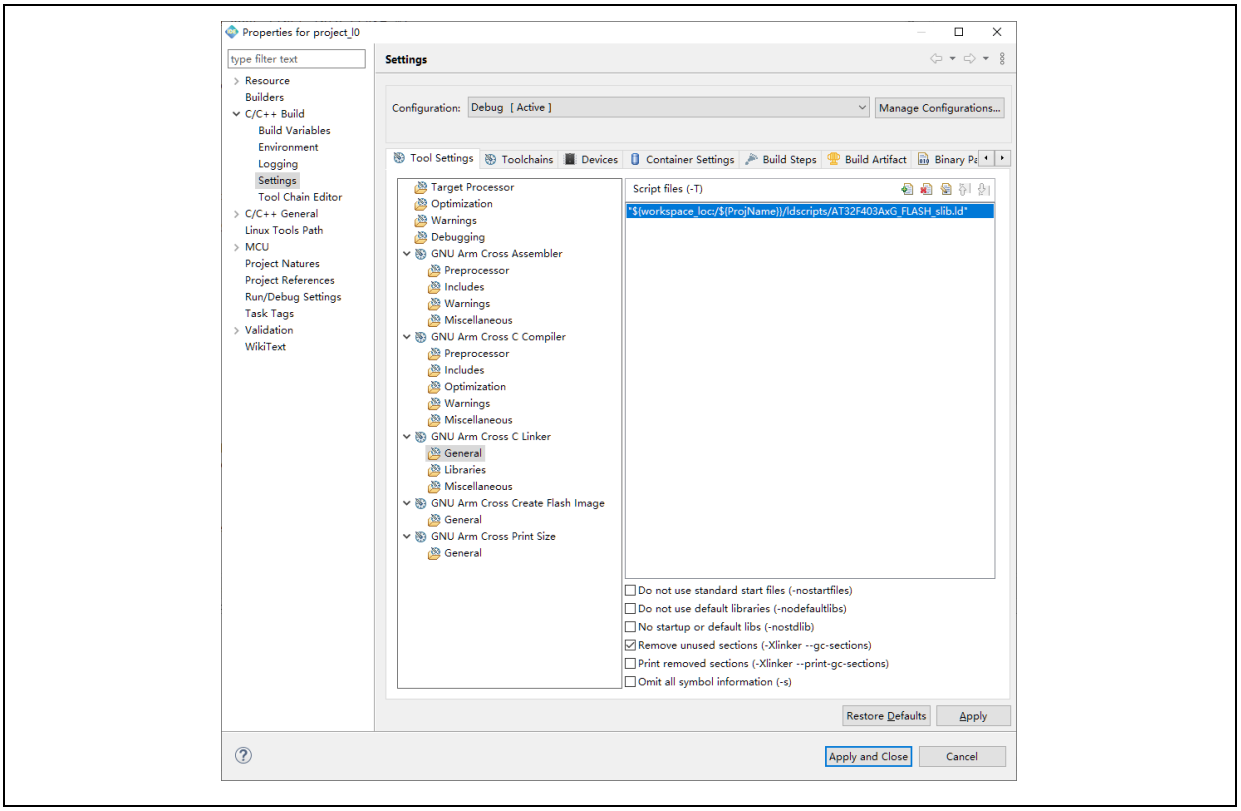
- Place the algorithm code to “.slib_inst section” and low-pass filter coefficients to “.slib_read_only section”, and specify global variables used by algorithm to “.slib_ram section”, as shown in Figure 22.

Figure 22. Specify areas in ld file

```
.slib_inst :
{
  . = ALIGN(4);
  *fir_filter.o (.text .text*);
  *arm_fir_f32.o (.text .text*);
  *arm_fir_init_f32.o (.text .text*);
  . = ALIGN(4);
} > SLIB_INST

.slib_read_only :        /* SLIB_READ_ONLY area */
{
  . = ALIGN(4);
  *fir_coefficient.o (.rodata .rodata*);
  . = ALIGN(4);
} > SLIB_READ_ONLY

.slib_ram :          /* Used for SLIB */
{
  . = ALIGN(4);
  *fir_filter.o (.data .data*);
  *fir_filter.o (.bss .bss*);
  . = ALIGN(4);
} > SLIB_RAM
```

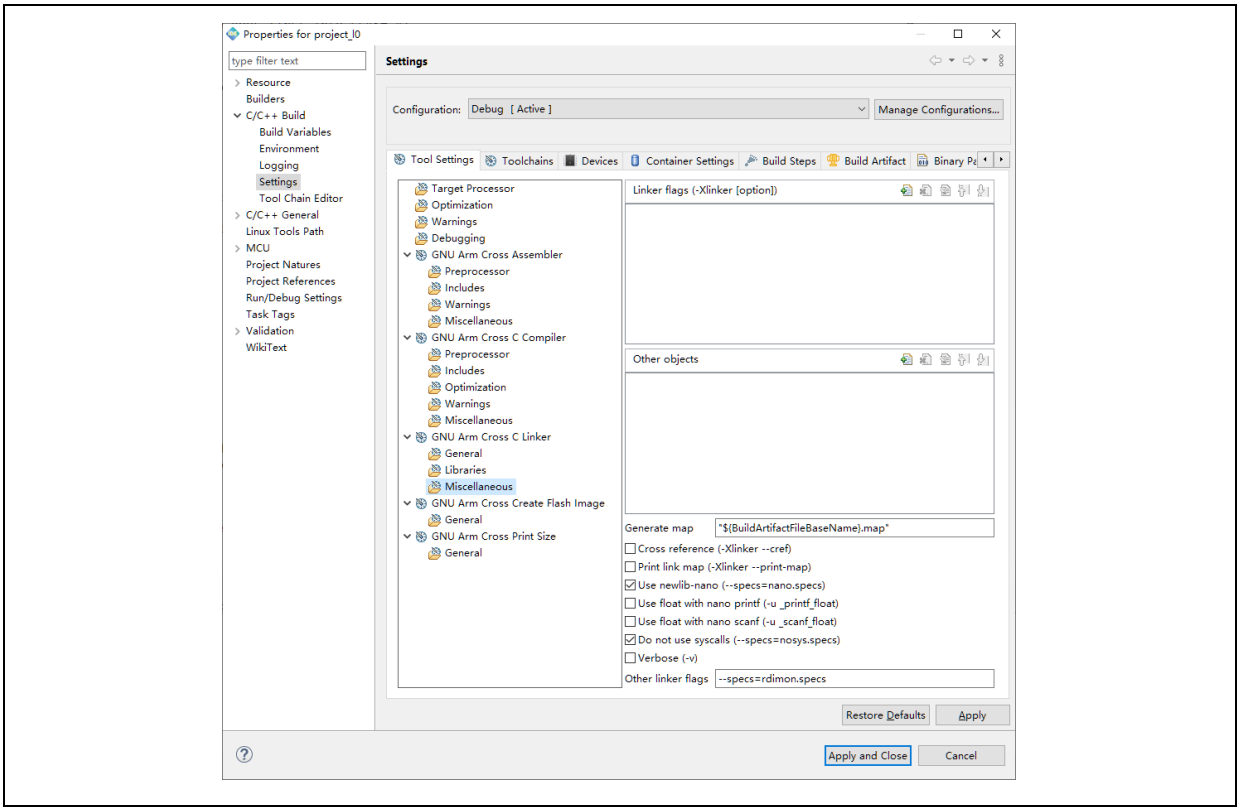
- Go to Project->Properties->C/C++ Build->Setting->GNU ARM Cross C Linker->General, and add the modified “ld” file to “Script files”.

Figure 23. Add the modified ld file



- This example uses GCC “libm.a” library. Go to Properties->GNU ARM Cross C Linker->Miscellaneous, and enter “Other linker flags” to “--specs=rdimon.specs” to avoid linker error message, as shown in Figure 24.

Figure 24. Add keywords to avoid compiling error



3.4.3 How to enable sLib function

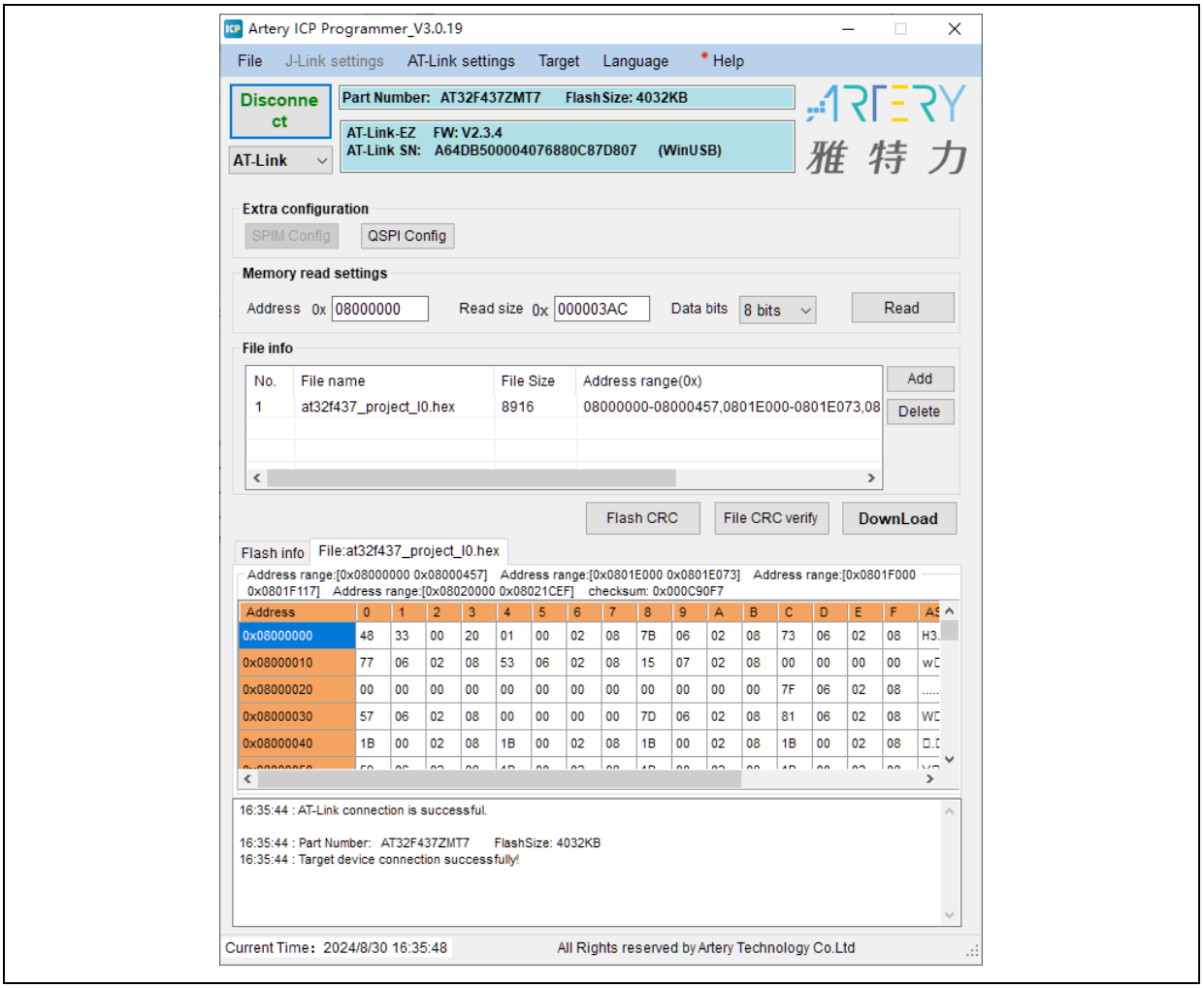
There are two ways to enable sLib function as follows:

(1) Use Artery ICP Programmer (recommended)

If use ICP Programmer, follow the steps below:

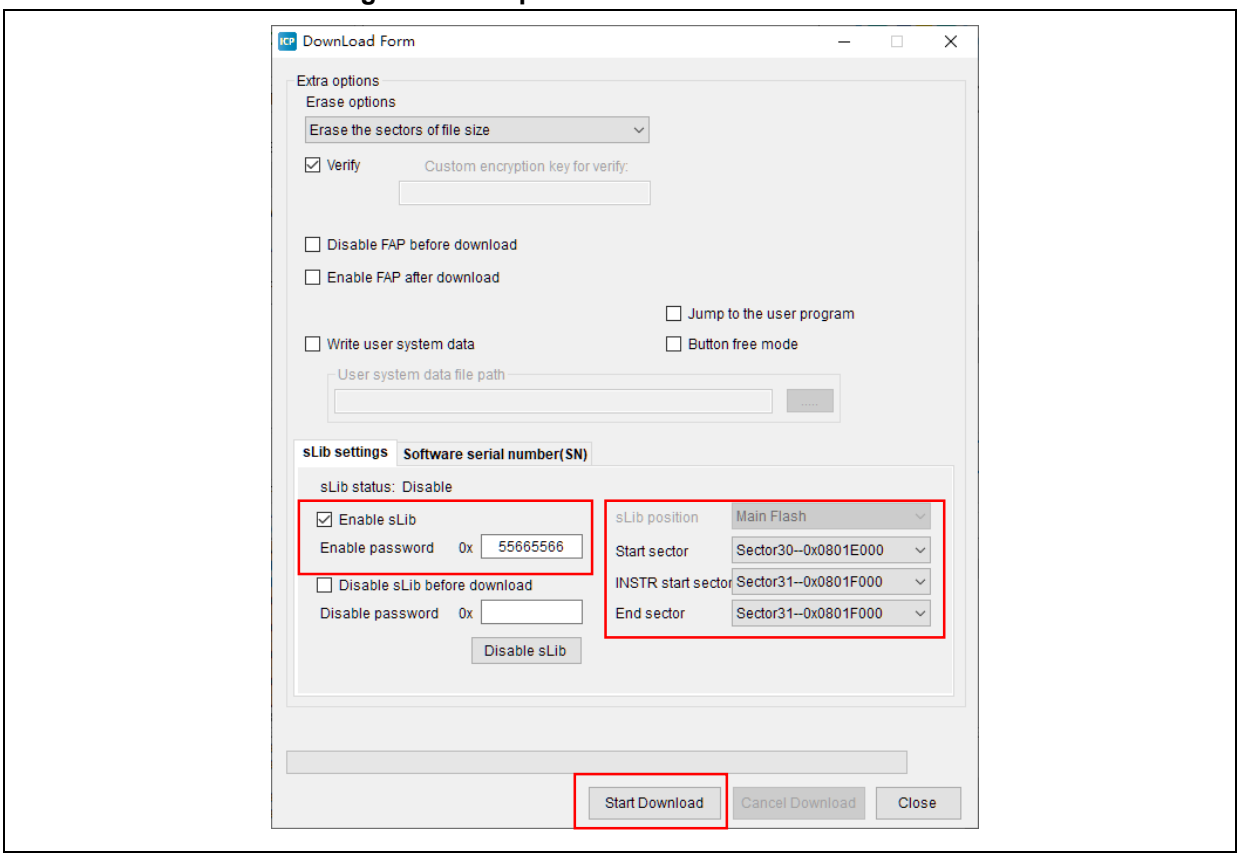
- Connect AT-Link to AT-START-F437 evaluation board and supply power to it.
- Open ICP Programmer, select AT-Link connection, add Project_L0 example and generate HEX or BIN files, as shown below:

Figure 25. ICP Programmer operation



- Click “Download”, a “Download Form” will pop out displaying sLib-related settings parameters. Set the corresponding sectors for sLib, set the enable password (user-defined) and tick “Enable sLib” and then click “Start Download” to complete programming and enable sLib successfully, as shown in Figure 26.

Figure 26. Set parameters in Download Form



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

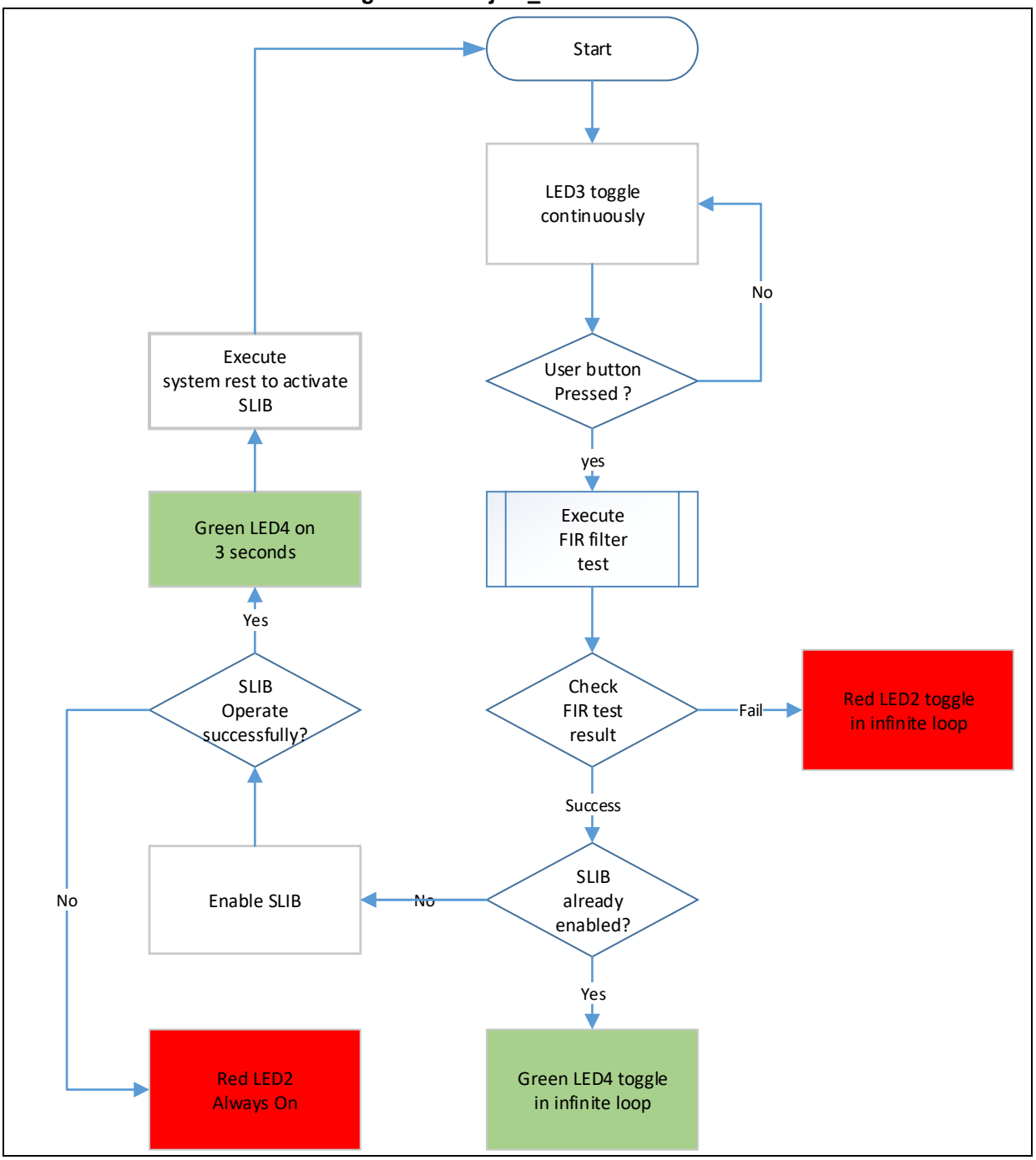
(2) Use main.c in slib_enable()

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

3.4.4 Project_L0 flow chart

In this example, FIR low-pass filter calculates the input signal “testInput_f32_1kHz_15kHz ” – a mixed signal of 1KHz and 15KHz sine waves, and outputs a 1KHz sine-wave data and stores it at testOutput. Then this output data will be compared with MATLAB-calculated data stored at refOutput. If error is lower than expected (SNR is greater than pre-defined threshold), a green LED on the evaluation board will start blinking; otherwise, a red LED will start blinking. Figure 27 shows a flow chart of Project_L0.

Figure 27. Project_L0 flow chart



To run this example code, follow the procedures below:

(1) Use Keil® µvision to open Project_L0 under "\utilities\AT32F435_437_slib_demo\project_l0\mdk_v5\" and start compiling.

(2) Prior to download, first check sLib or read/write protection (FAP/EPP) is enabled for the chip on AT-START-F437 evaluation board. If enabled, use ICP tool to unlock this protection before starting download.

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

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

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

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

3.4.5 How to generate header file and symbol definition file

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

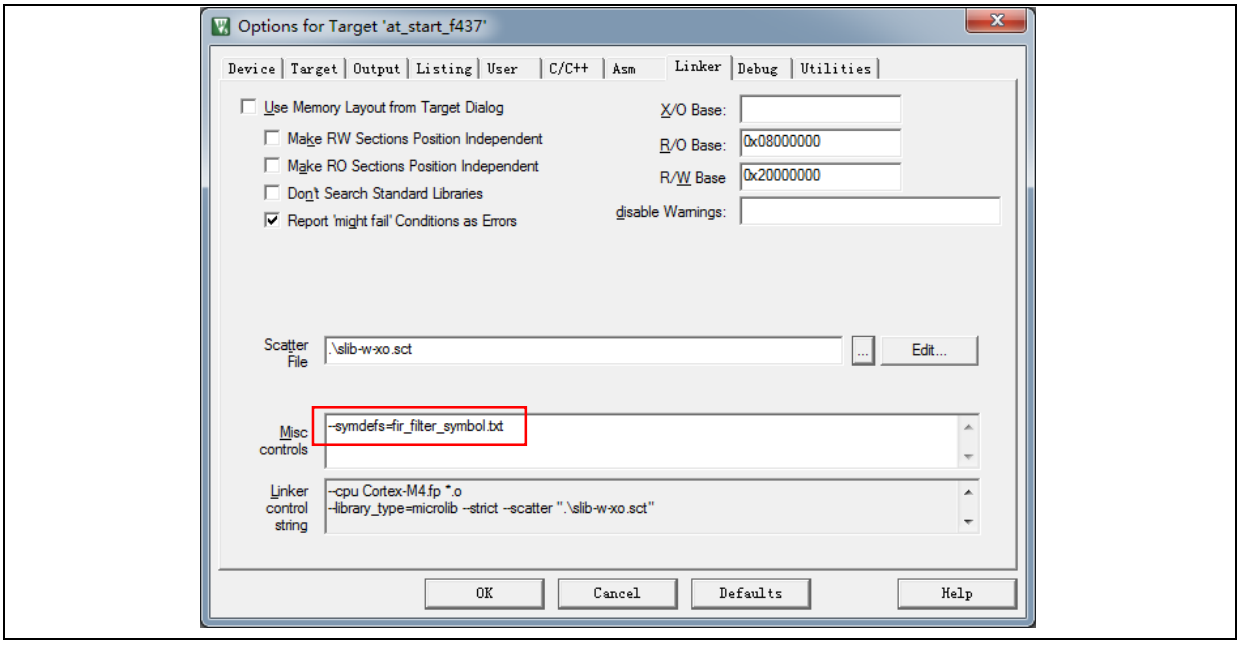
How to generate symbol definition file depends on toolchains used.

Use Keil® µvision to generate a symbol definition file

Proceed as follows:

- Go to Options for Target → Linker window.
- In “Misc controls” column, add the command “--symdefs=fir_filter_symbol.txt”, as shown in Figure 28.

Figure 28. Keil Misc controls option



- After compiling the whole project, a symbol definition file named “fir_filter_symbol.txt” is created under “project_l0\mdk_v5\Objects”.
- Such symbol definition file contains all symbol definitions related to the project, and thus some of them should be removed so as to reserve low-pass filter function definitions which will be used by end users. The reduced fir_filter_symbol.txt is shown below.

Figure 29. Reduced fir_filter_symbol.txt

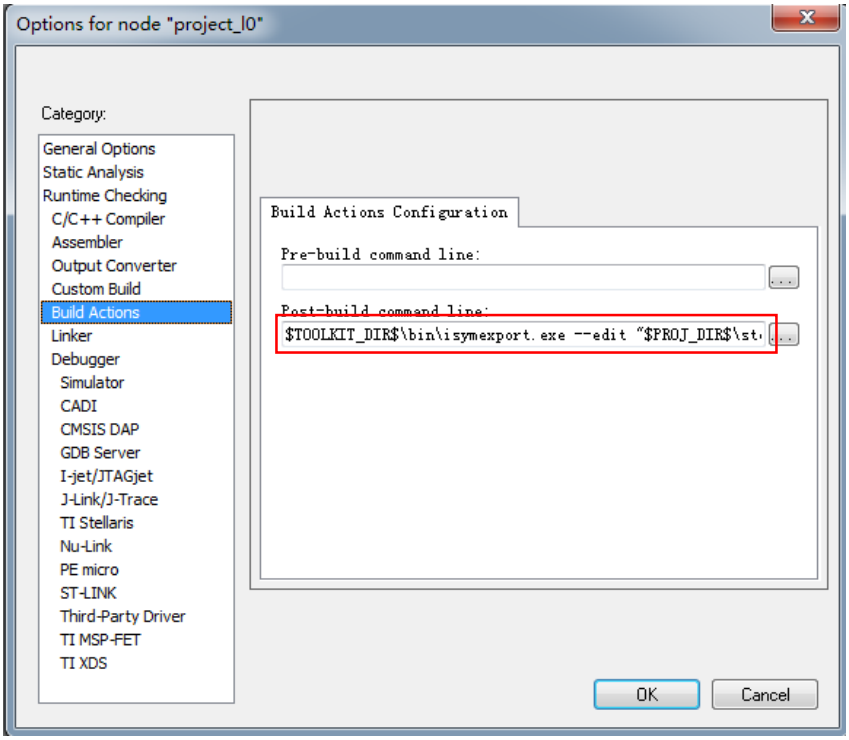
```
#<SYMDEFS># ARM Linker, 5060960: Last Updated: Fri Aug 30 10:32:19 2024

0x0801f001 T FIR_lowpass_filter
```

Use IAR to generate symbol definition files

Proceed as follows:

- Go to Project→Option→Build Actions.

Figure 30. Set IAR Build Actions option



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

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

- The “fir_filter_symbol.o” refers to a symbol definition file. The “steering_file.txt” stored under “project_l0\iar_v8.2” is used to select which symbol of functions need to be created. Then edit them according to the contents in the sLib, as shown in Figure 31 in which "show" is a command used to select a function.

Figure 31. Edit steering_file.txt

```
show FIR_lowpass_filter
```

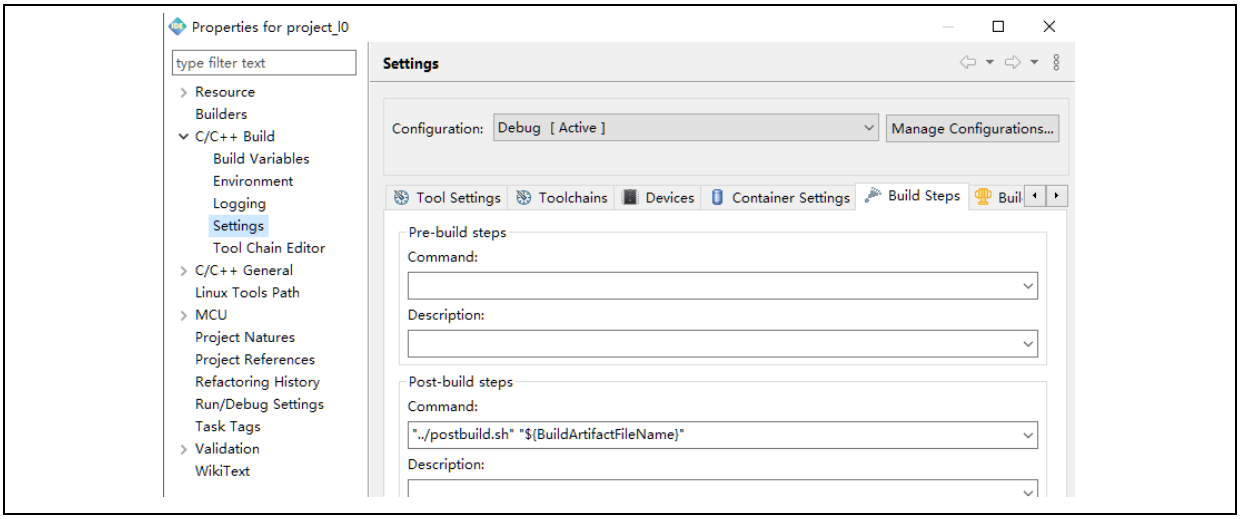
Use AT32 IDE to generate a symbol definition file

Proceed as follows:

- Create a “keep_sym.txt” to select the symbols to be generated, and edit according to the content called by sLib.
- Create a “postbuild.sh” (its content is shown in the project) to generate “.ld” file containing function name and address.
- Select project_l0, and right click to select “Properties”.

- Go to C/C++ Build->Settings->Build Steps->Post-build steps->Command, and enter "../postbuild.sh" "${BuildArtifactFileName}", and then click “Apply”.
- After successful compiling, a “keep_sym_app.ld” file is generated under DEBUG folder.

Figure 32. Add post-build in AT32 IDE



3.5 Project_L1: example for end users

Project_L1 example needs to use FIR low-pass filter functions that are debugged in Project_L0 and programmed into AT32F437 Flash memory with sLib enabled.

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

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

Cautions:

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

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

3.5.1 Create a user project

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

Figure 15shows Flash memory map used in this example. It is necessary for end users to separate such sLib area from other areas, so as to prevent codes from being placed into sLib.

Proceed as follows:

Keil® µvision’s scatter file

Based on the “end_user_code.sct” under “project_l1\mdk_v5”, users can divide main Flash

memory into two segments. The area in between is sLib area. Besides, the SLib_SRAM area is reserved for RAM, as shown in Figure 33.

Figure 33. Modified scatter file

```
LR_IROM1 0x08000000 0x0001E000  {    ; load region size_region
  ER_IROM1 0x08000000 0x0001E000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
  }
  RW_IRAM1 0x20001000 0x0005F000  {  ; RW data
   .ANY (+RW +ZI)
  }

 ; 0x20000000 ~ 0x20000FFF   RAM reserved for SLIB code

}

; 0x0801E000 ~ 0x0801FFFF is SLIB area

LR_IROM2 0x08020000 0x003D0000  {    ; load region size_region
  ER_IROM2 0x08020000 0x003D0000  {  ; load address = execution address
   .ANY (+RO)
  }
}
```

IAR's ICF file

Users can refer to the following content in the "enduser.icf" file under "project_l1\iar_V8.2\".

Figure 34. 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__];
```

AT32 IDE's ld file

Users can refer to the "FLASH_enduser.ld" file under "project_l1\at32_ide\", and add the content shown below.

Figure 35. Add files generated in project_l0

```
INCLUDE keep_sym_app.ld
```

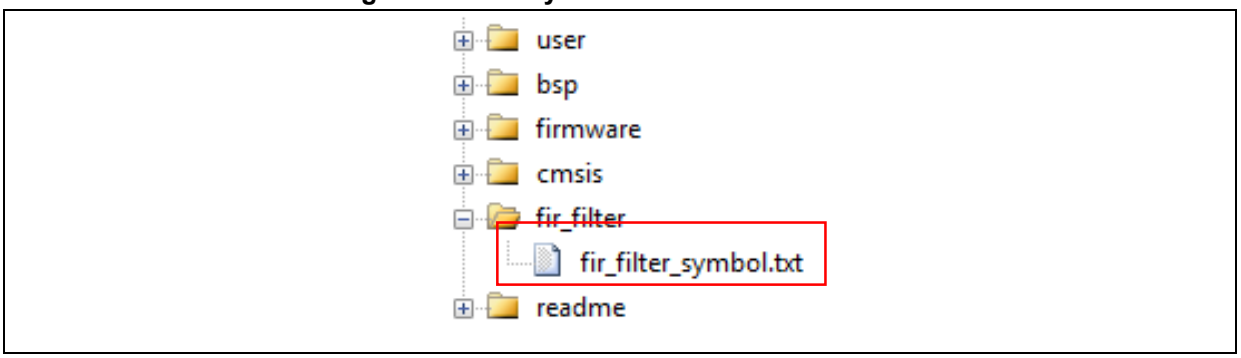
3.5.2 Add symbol definition files into project

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

Add a symbol definition file in Keil® µvision environment

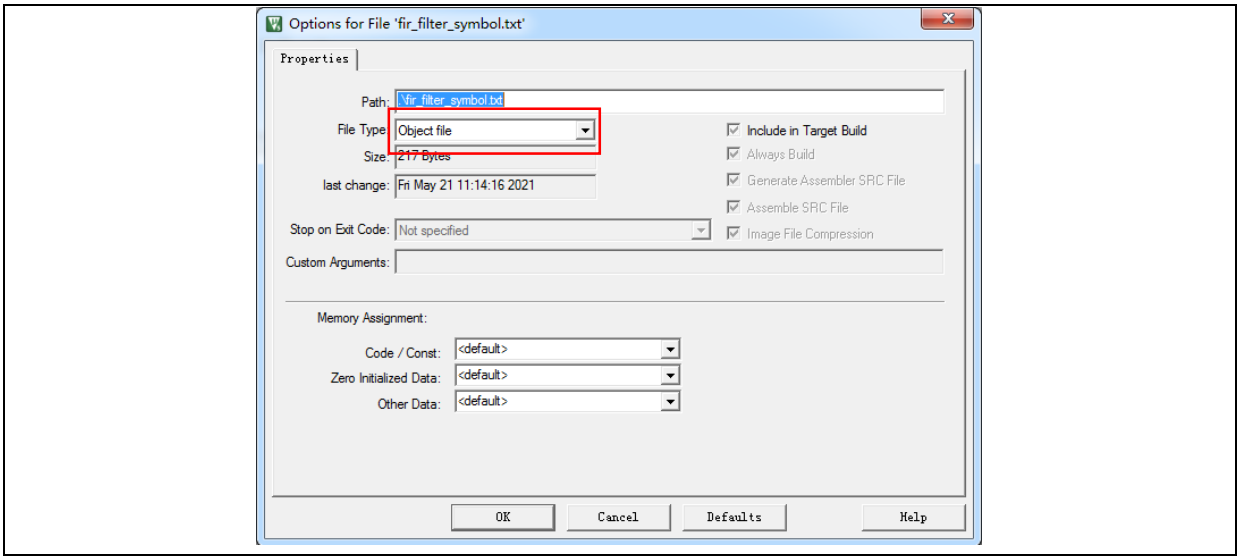
Add "fir_filter_symbol.txt" file into project, as shown in Figure 36.

Figure 36. Add symbol definition file in Keil



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

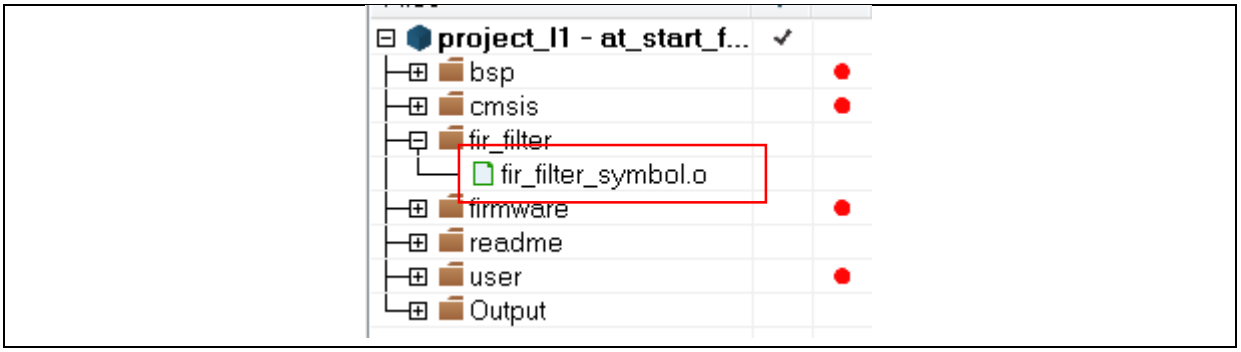
Figure 37. Change symbol definition file to Object file



Add a symbol definition file in IAR environment

Add “fir_filter_symbol.o” file into “fir_filter” group, as shown in Figure 38.

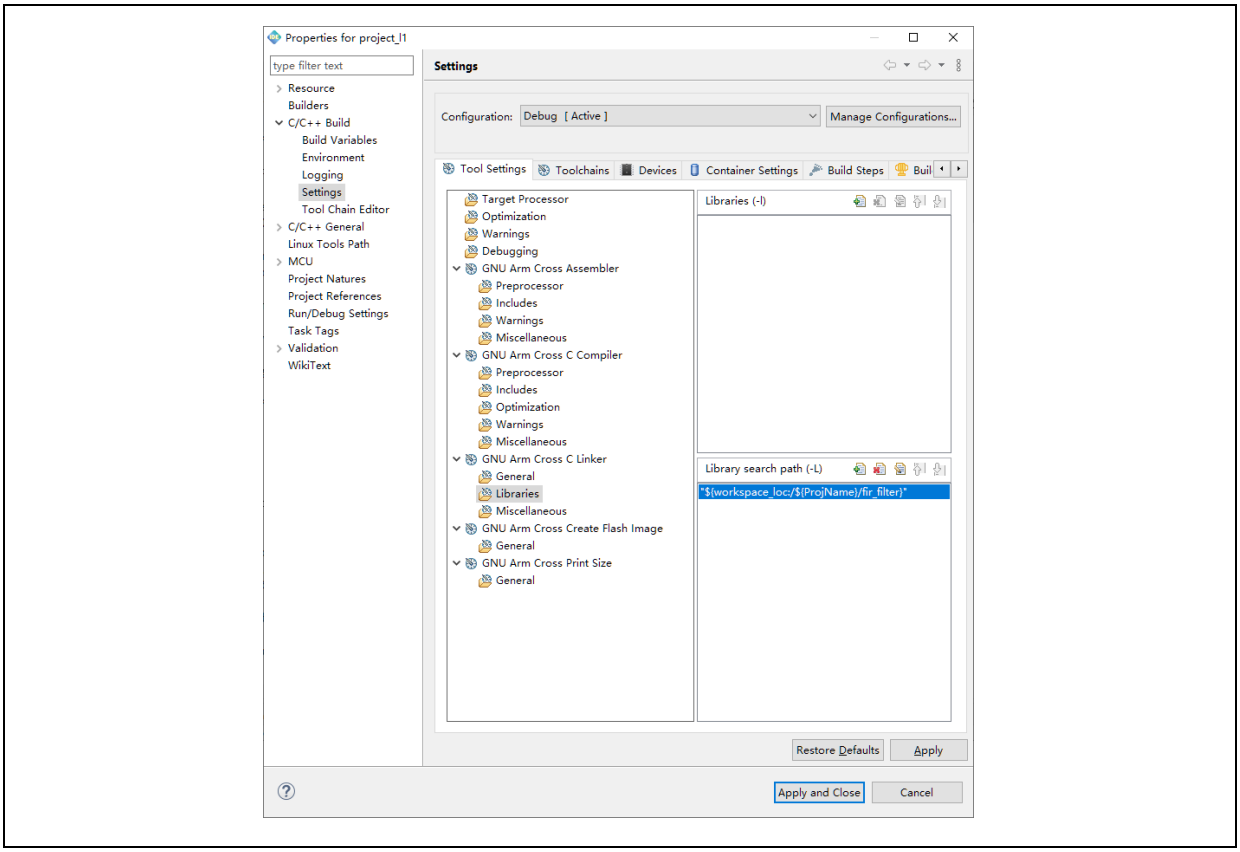
Figure 38. Add symbol definition file in IAR



Add a symbol definition file in AT32 IDE environment

Add the “keep_sym_app.ld” file stored in “fir_filter” into the project. Go to Project->Properties->C/C++ Build->Setting->GNU ARM Cross C Linker->Library to add the path of “keep_sym_app.ld” file.

Figure 39. Add ld file path in AT32 IDE environment



3.5.3 Call sLib functions

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

Proceed as follows:

FIR_lowpass_filter(inputF32, outputF32, TEST_LENGTH_SAMPLES);

With:

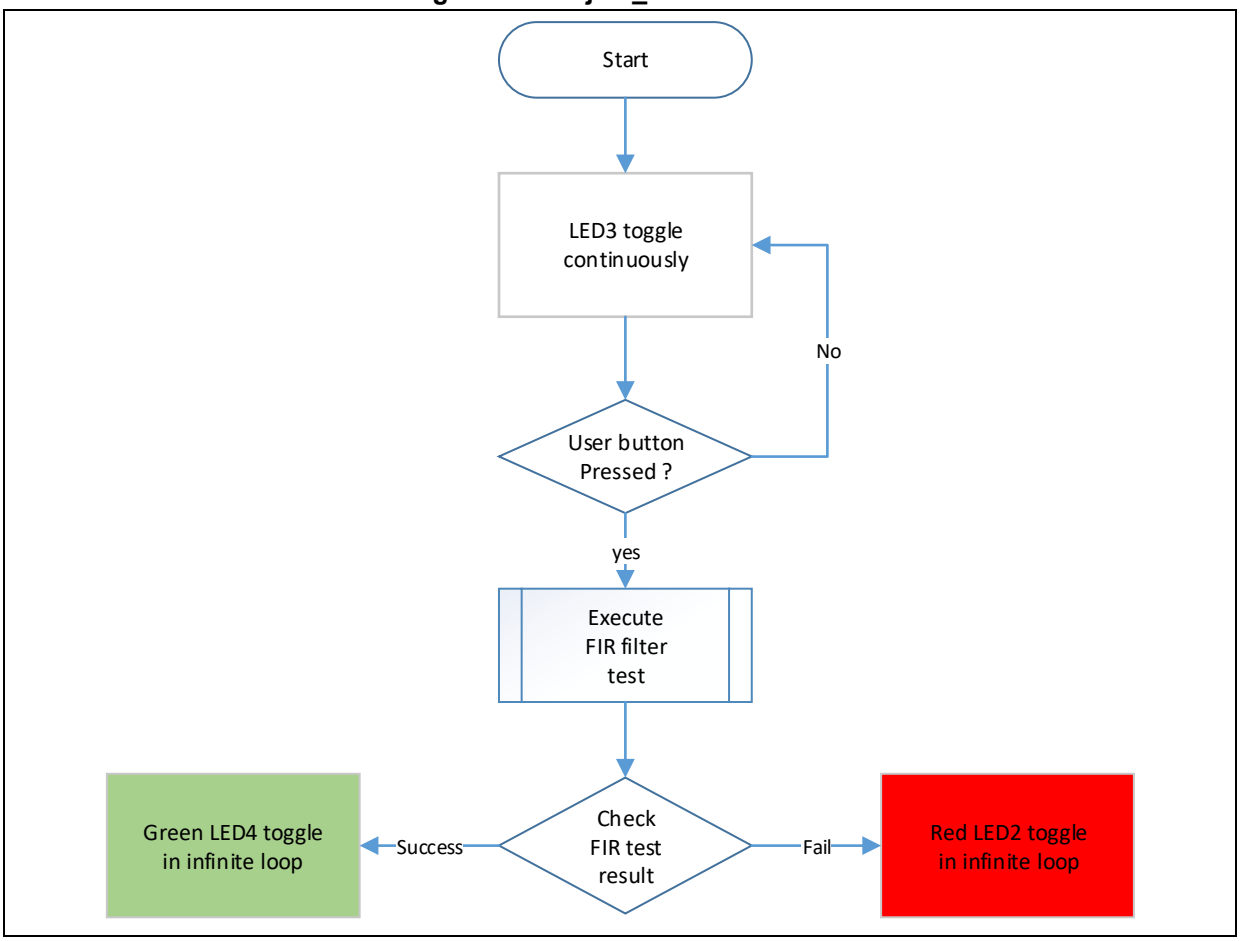
- *inputF32*: pointer to data table storing input signals
- *outputF32*: pointer to data table storing output signals
- *TEST_LENGTH_SAMPLES*: the size of signal samples to be processed

3.5.4 Project_L1 flow chart

Project_L1 flow chart is shown in Figure 40.

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

Figure 40. Project_L1 flow chart

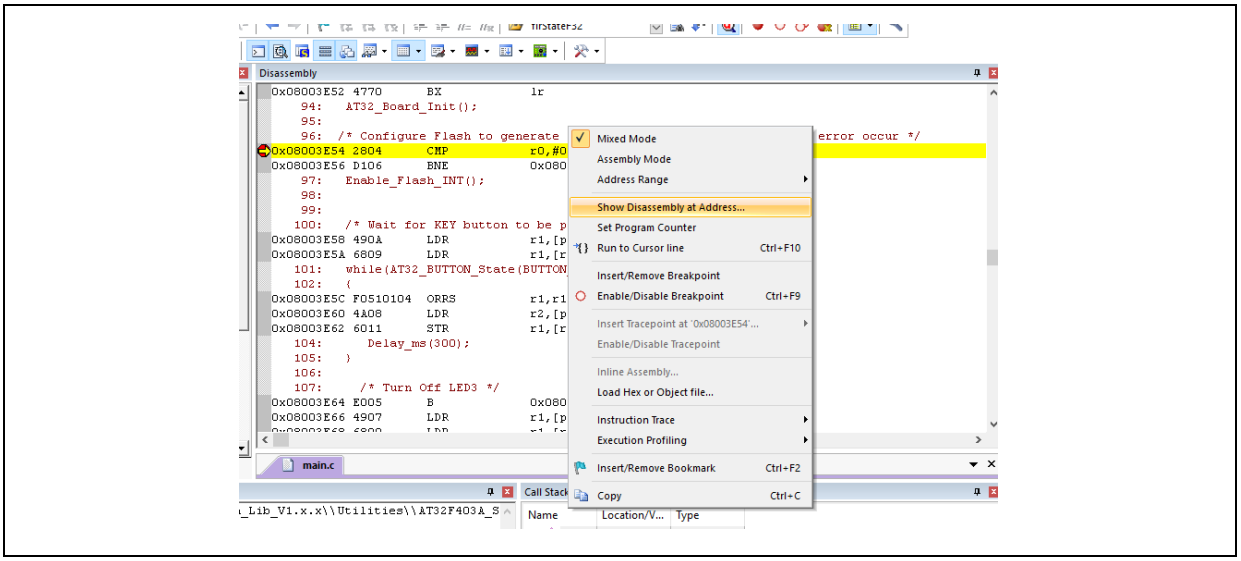


3.5.5 sLib protection in debug mode

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

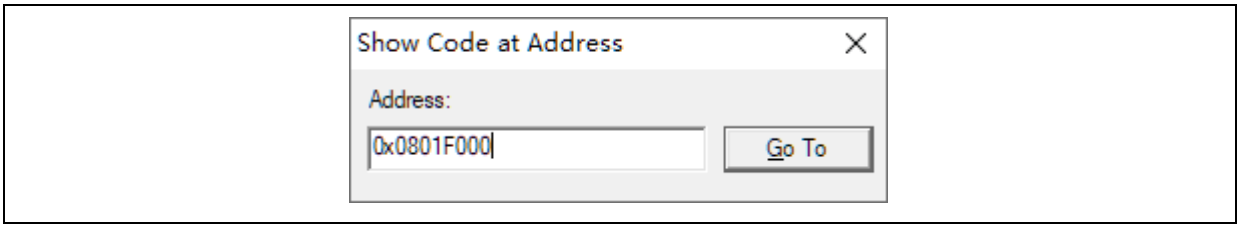
- Open Project_L1 and recompile;
- Click “Start/Stop Debug Session” to enter debug mode;
- In “Disassembly” window, right click and choose “Show Disassembly at Address”, as shown in Figure 41.

Figure 41. "Show Disassembly at Address" window



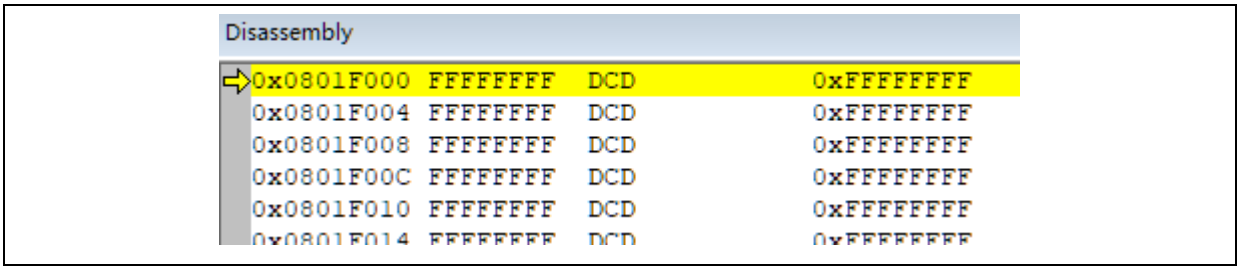
- Enter the start address of SLIB_INSTRUCTION.

Figure 42. "Show Code at Address" setting



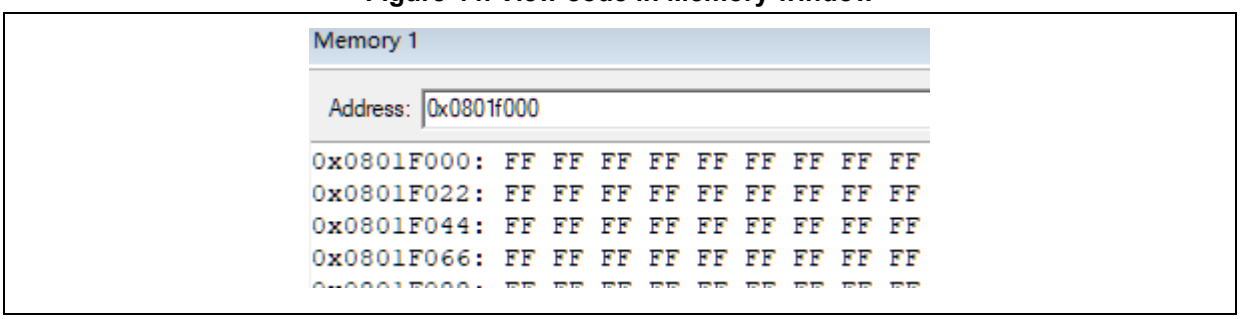
- From the SLIB_INSTRUCTION address, all codes are 0xFFFFFFFF.

Figure 43. View code



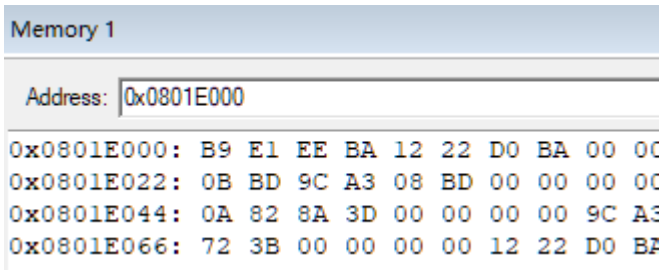
- Similarly, in "Memory" window, enter the SLIB_INSTRUCTION address and return all 0xFF.

Figure 44. View code in Memory window



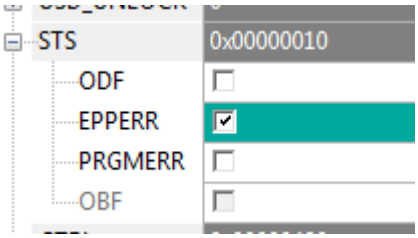
- In “Memory” window, enter the start address of SLIB_READ_ONLY. Because this area is readable by D-Code, we can see their original data.

Figure 45. View SLIB_READ_ONLY start sector in Memory



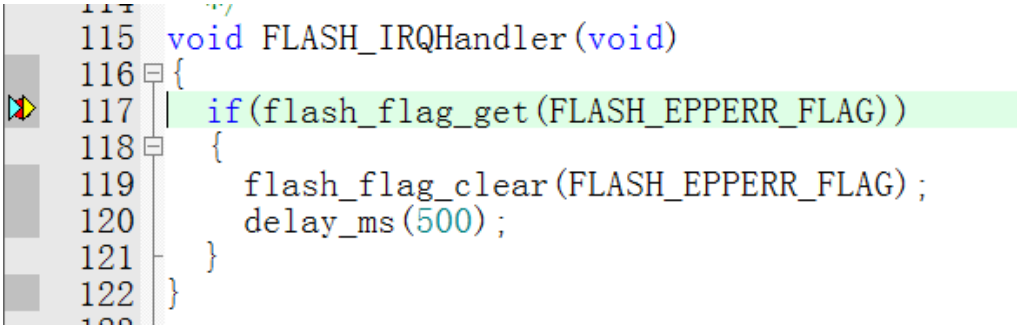
- In “Memory” window, when you click data in SLIB area twice to try to modify them, the EPPERR bit in the FLASH_STS register will be set to 1 as a warning, indicating that they are write-protected.

Figure 46. SLIB write protection test



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

Figure 47. Write protection error interrupt



4 Integrate and download codes of solution provider and user

After the completion of code design on both solution providers and end users, these code should be downloaded into the same MCU device. In this scenario, data security issue should be taken into account. In the subsequent sections, two download procedures based on Project_L0 and Project_L1 are recommended as a reference.

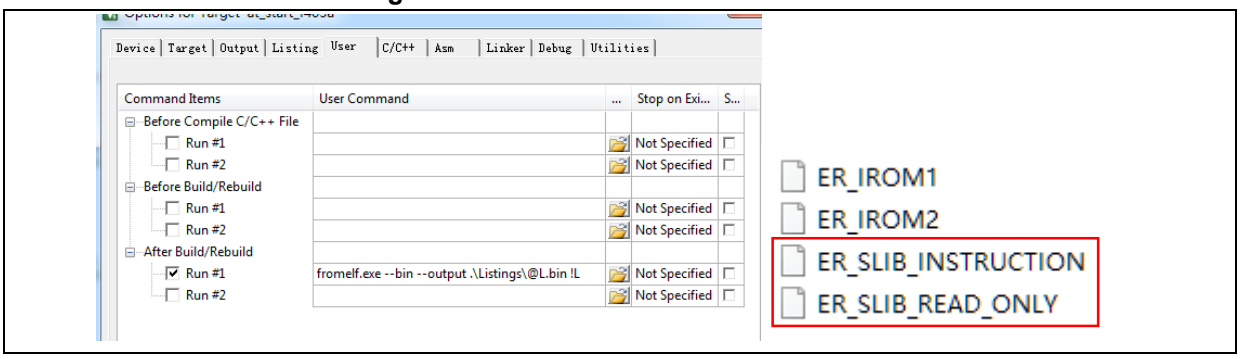
The procedures involve AT-Link offline download mode, with its details being described in ICP user guide and AT-Link user manual.

4.1 Write code separated on solution provider and end user

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

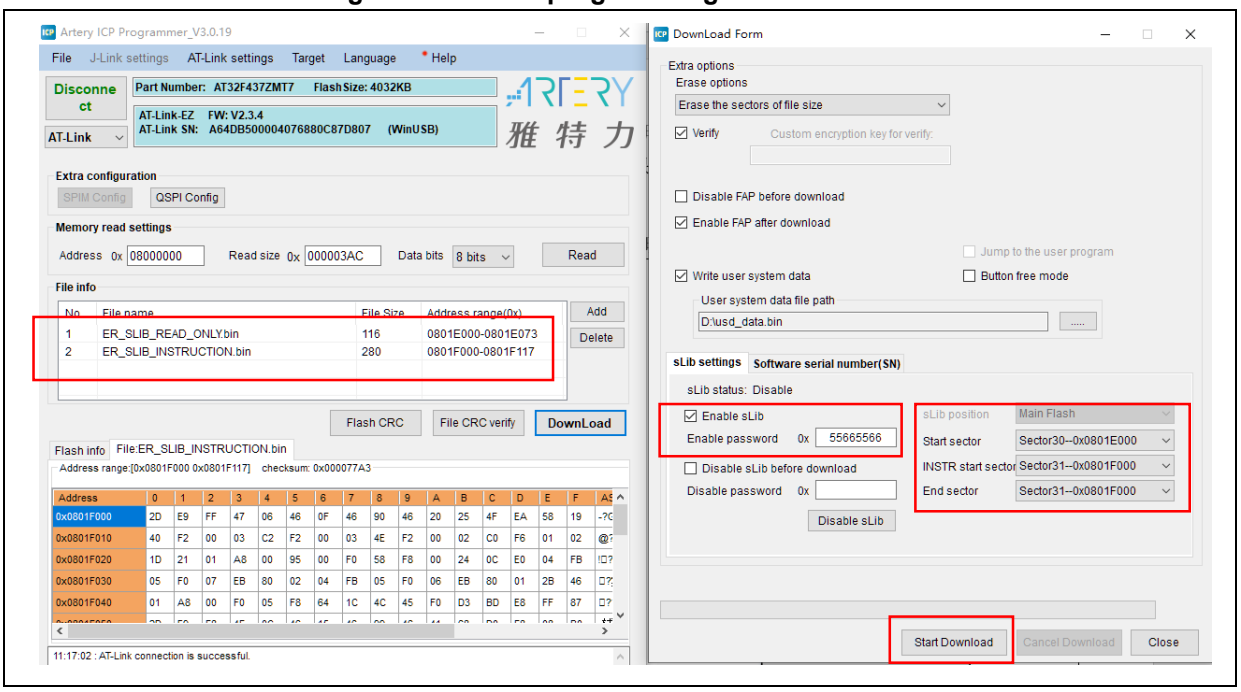
(1) The solution provider uses IDE software or ICP tool to save the sLib code in the compiled project as BIN or HEX file. For example, in the Keil project, add “fromelf.exe --bin --output .\Listings\@L.bin !L” in the “user” option to generate a “.bin” file of the corresponding firmware, and add a suffix “.bin” to the sLib area file. In this example, they are “ER_SLIB_INSTRUCTION.bin” and “ER_SLIB_READ_ONLY.bin”, corresponding to the SLIB-INSTRUCTION file and SLIB-DATA file. Users can also use the latest ICP tool to open the project HEX file and then click File->Save as BIN file, as shown below.

Figure 48. Generate bin file of SLIB code



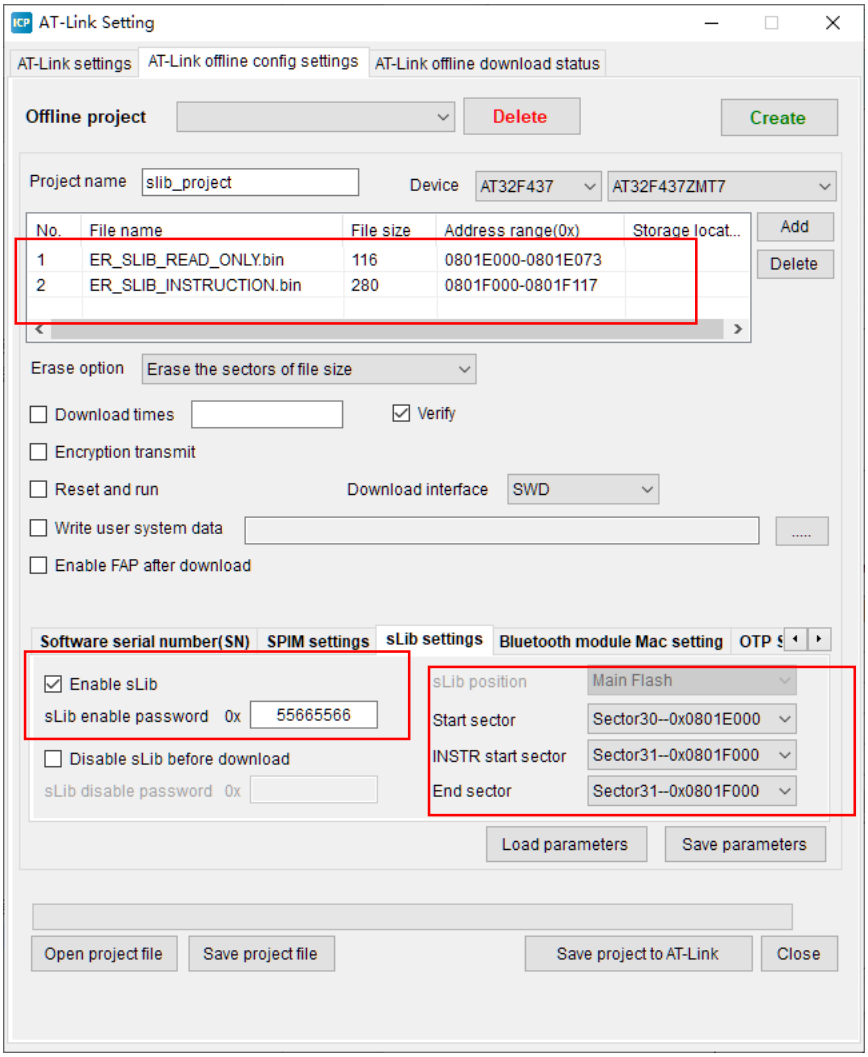
(2) Use ICP tool to program “ER_SLIB_INSTRUCTION.bin” and “ER_SLIB_DATA.bin” to MCU online, as shown below.

Figure 49. Online programming to MCU via ICP



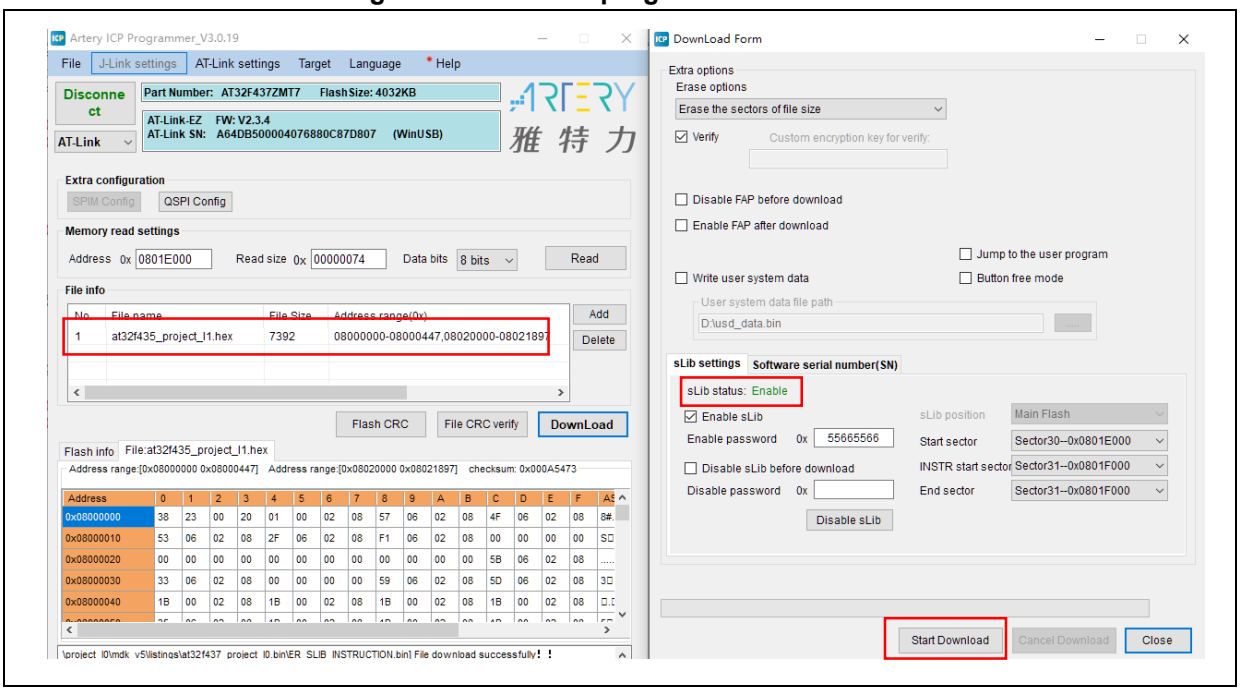
(3) Users can also use ICP tool to configure an offline project and save it to AT-Link, and then complete offline programming to MCU through AT-Link, as shown below.

Figure 50. Offline programming to MCU via AT-Link



(4) After completing step 2 or 3, end users can get a MCU device with programmed sLib code (sLib status: enabled), and program the application code to MCU through online or offline programming, as shown in Figure 51.

Figure 51. End user programs code to MCU



4.2 Combine solution provider code with end user code

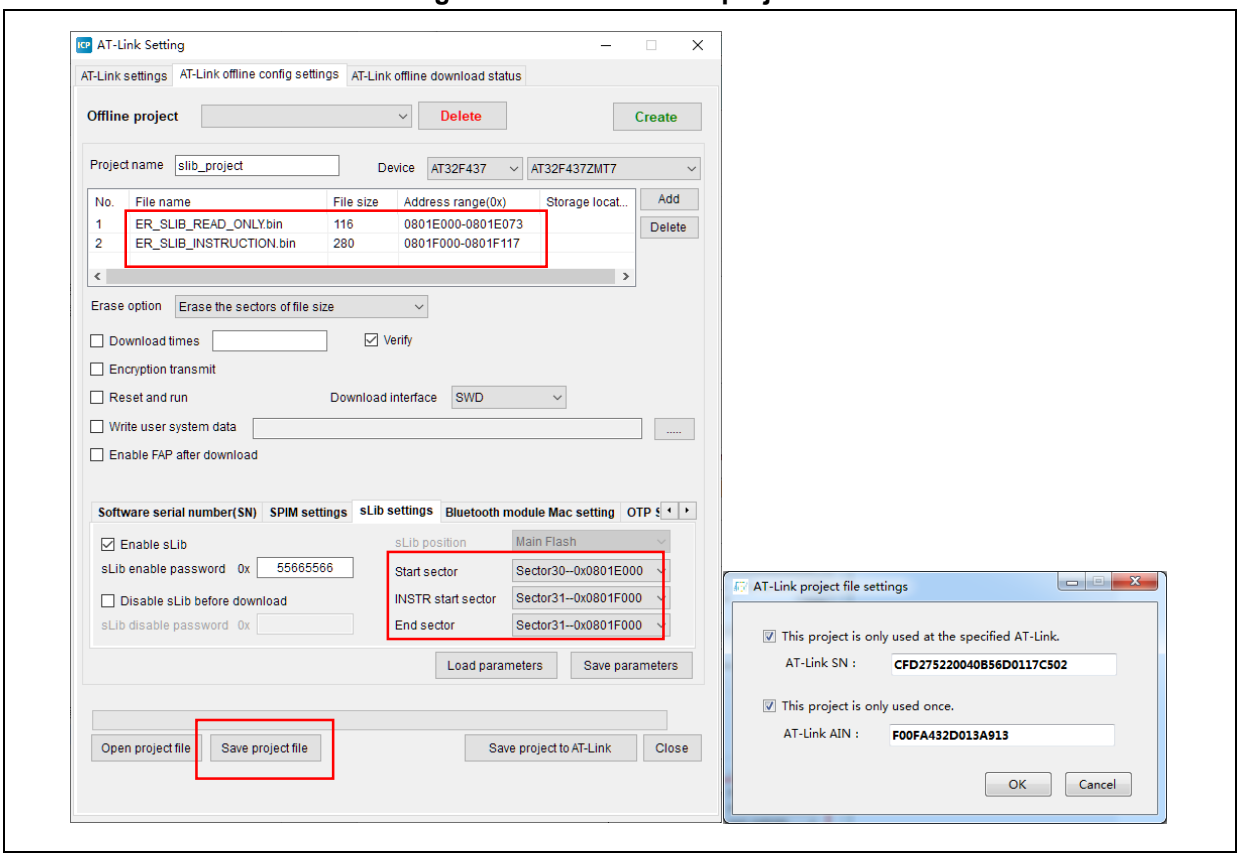
SLIB code from solution provider and end user code are integrated into an offline project, which is then downloaded into MCU via AT-Link offline mode.

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

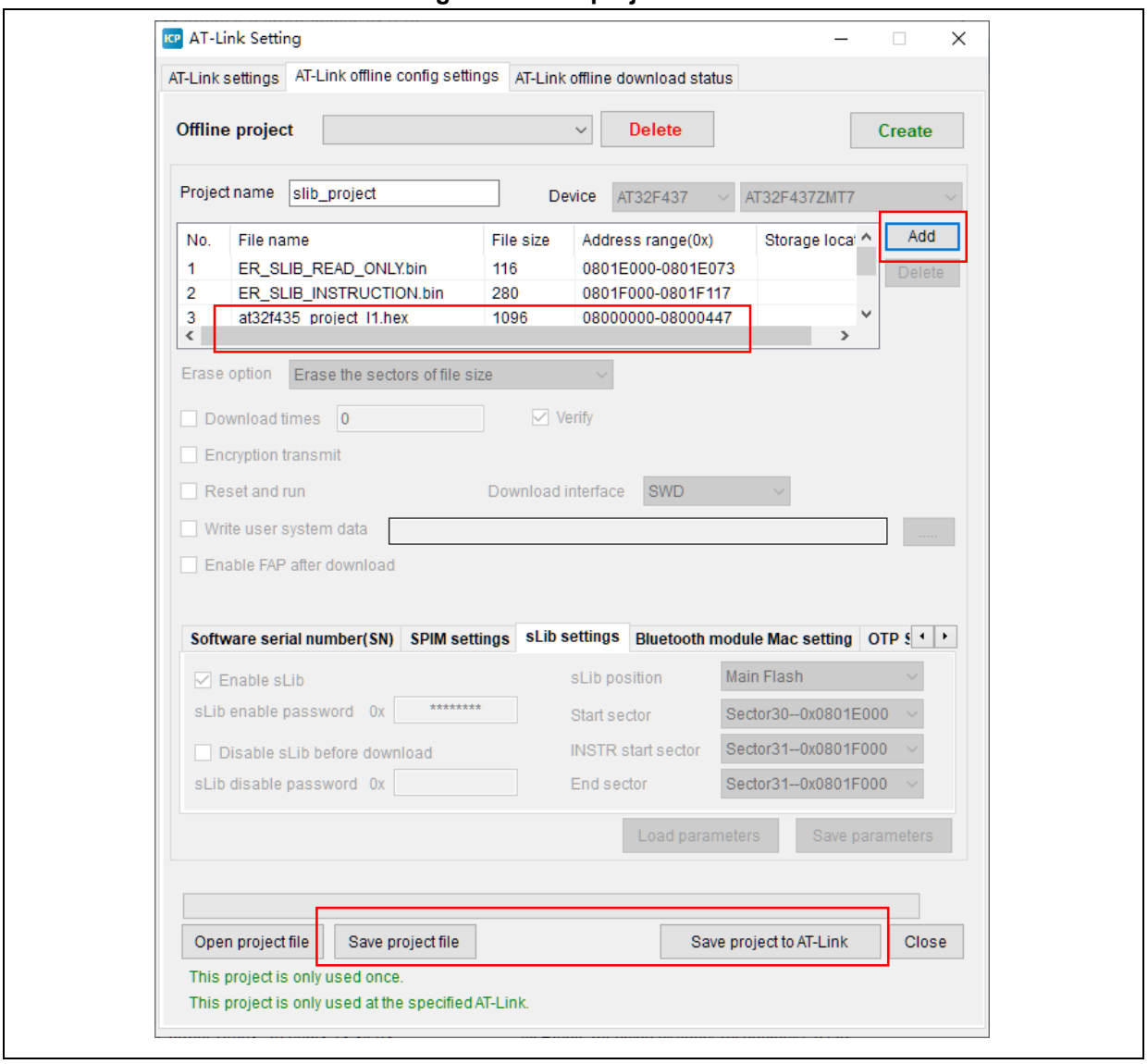
Figure 52. Create offline project



(3) For end users, they can use ICP to open such offline project, and click “Add” to add user application code into such project, and save it to PC or directly to AT-Link, and then perform offline download to finish the whole operation. Figure 53 shows how to add a project file.

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

Figure 53. Add project file



5 Revision history

Table 2. Document revision history

Date	Version	Revision note
2021.9.8	2.0.0	Initial release.
2024.8.30	2.0.1	1. Modified the configurable range of sLib in Flash and SRAM. 2. Added AT32 IDE support and relevant description.

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 granted 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 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 on any patent, copyright or other intellectual property right.

Purchasers hereby agree 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 aircraft application; (C) any aerospace application or environment; (D) any weapon application, and/or (E) or other uses where the failure of the device or product could result in personal injury, death, property damage. Purchasers' unauthorized use of them in the aforementioned applications, even if with a written notice, is solely at purchasers' risk, and Purchasers are solely responsible for meeting all legal and regulatory requirements in such use.

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