AN0004
Application Note

AT32 Performance Optimization

Introduction

This application note describes how to improve AT32 performance by software.

Note: The corresponding code in this application note is developed on the basis of V2.x.x BSP provided by Artery. For other versions of BSP, please pay attention to the differences in usage.

Applicable products:

Part number	AT32Fxx

Contents

List of Tables

List of Figures

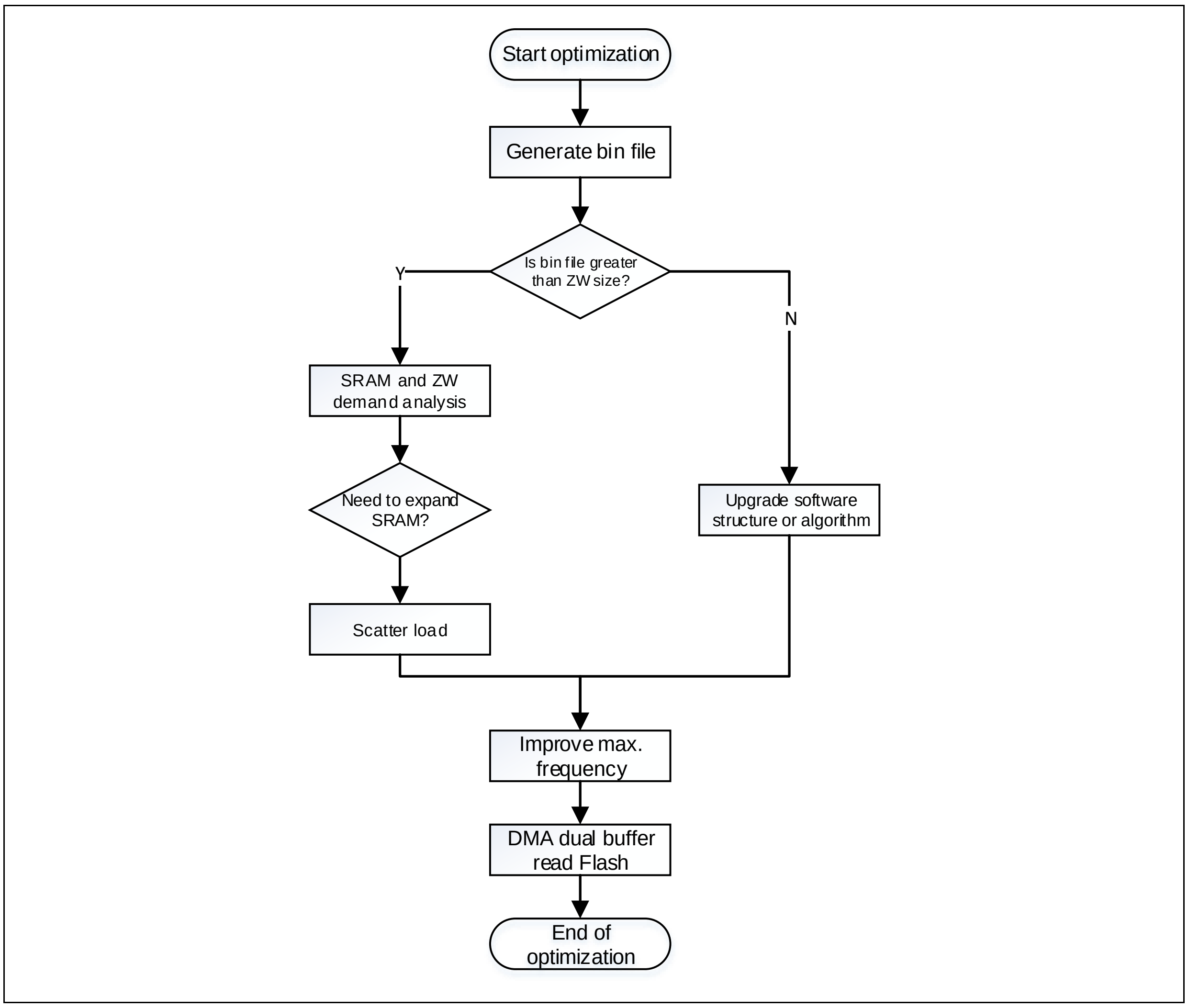
1 Overview

Performance optimization is the result of multiple tuning. Before this, it is necessary for the user to have a deep understanding of the hardware and software structures and parameters of the system. For hardware, these include Flash size, SRAM size, ZW and NZW size, maximum frequency and other parameters. For software, the user should have a basic understanding of the whole procedure, execution time of code and algorithm, frequency of accessing to critical data and other information. Then optimize the system step by step according to the actual projects to improve performance. Use the following procedures to improve performance:

1) First, generate a bin file to judge whether the further optimization is needed
2) Check whether the SRAM needs to be expanded or not
3) Check the map table to have an overall understanding of the file structure
4) Adjust the code link structure to implement scatter load
5) Increase the system operating frequency
6) Use DMA dual buffer to read more Flash data

Figure 1 shows a rough flowchart for performance optimization

Figure 1. Optimization flowchart


Start optimization
Generate bin file
Is bin file greater than ZW size?
Y
N
SRAM and ZW demand analysis
Need to expand SRAM?
Scatter load
Upgrade software structure or algorithm
Improve max. frequency
DMA dual buffer read Flash
End of optimization

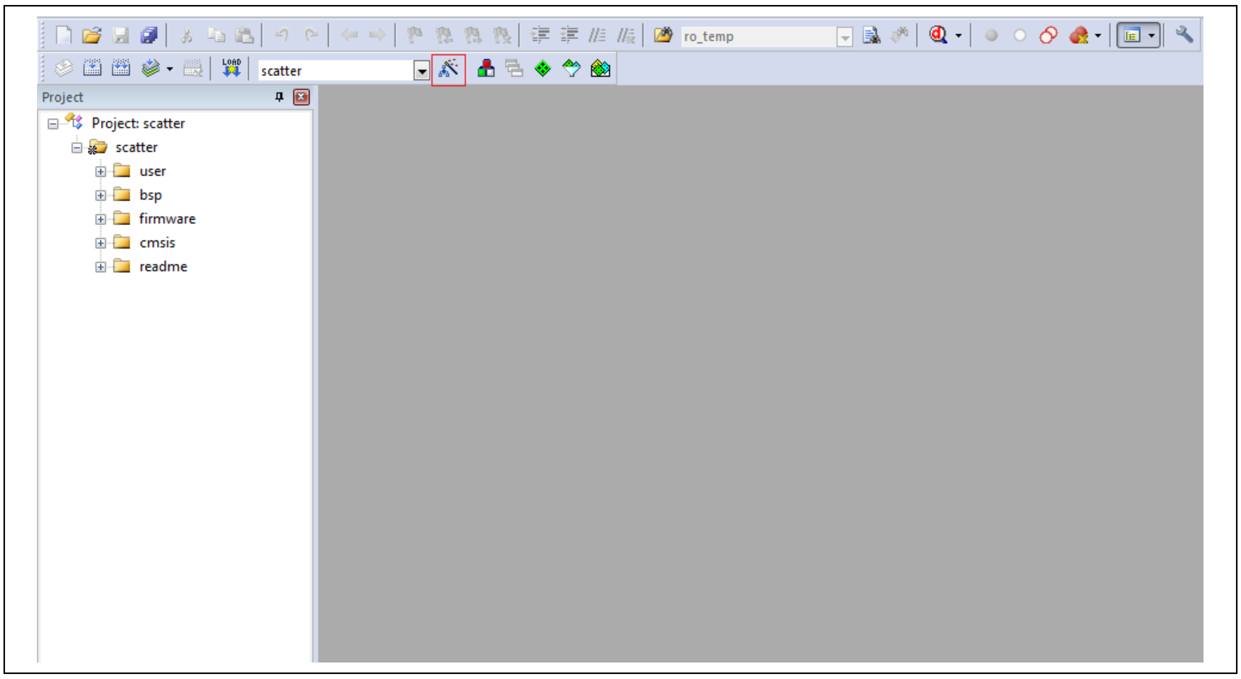
2 Bin file judgment

2.1 Generate bin file

In Keil environment, build and compile to generate a binary file for chip programming.

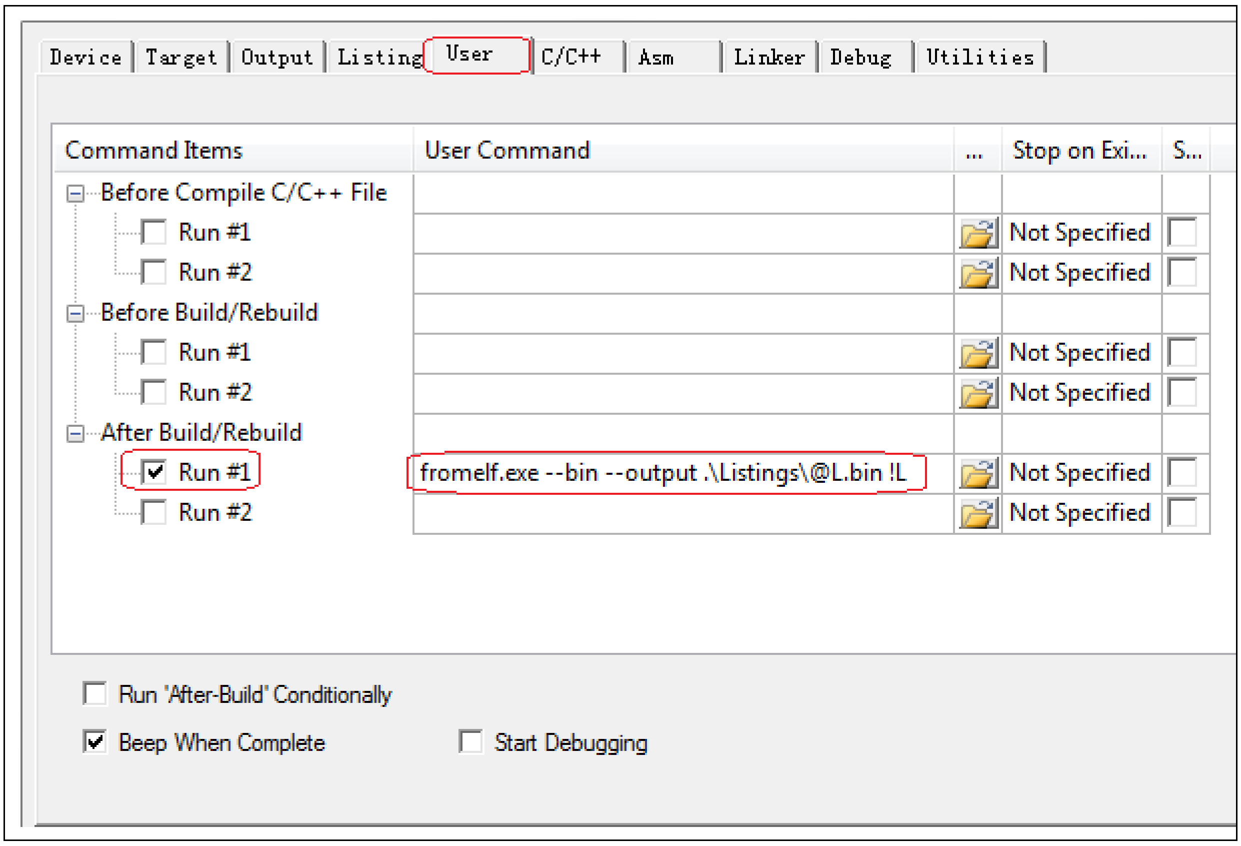
Use Keil to open a project, and then click on *Options for Target* icon, as shown below.

Figure 2. OptionForTarget icon



Then, the following window (Figure 3) will appear. Click on *User* and tick *Run #1* under *After Build/Rebuild*, and type in the command line: *fromelf.exe --bin --output .\Listings\@L.bin !.*

Figure 3. Compile command



Command Items	User Command	...	Stop on Exi...	S...
Before Compile C/C++ File				
☐ Run #1			Not Specified	☐
☐ Run #2			Not Specified	☐
Before Build/Rebuild				
☐ Run #1			Not Specified	☐
☐ Run #2			Not Specified	☐
After Build/Rebuild				
✓ Run #1	fromelf.exe --bin --output .\Listings\@L.bin !L		Not Specified	☐
☐ Run #2			Not Specified	☐

☐ Run 'After-Build' Conditionally

✓ Beep When Complete ☐ Start Debugging

2.2 View bin file size

After compiling, find out the binary bin file generated and check its size. Considering that AT32F4xx MCUs (depending on their respective datasheet) come with internal zero-wait storage space, so it is unnecessary to optimize the structure of the execution file when the size of the bin file is within the zero-wait area. In this case, we focus on the optimization of software architecture and algorithm, or increasing maximum frequency to achieve better performance.

The subsequent sections are written based on the condition when the bin file size exceeds that of zero-wait area. These include the optimization of the execution file structure, and SRAM demand analysis.

3 SRAM expansion

AT32F4xx series MCUs contain internal SRAM and zero-wait storage space by default (refer to their respective datasheet). However, the user can also reconfigure the size of SRAM and zero-wait area according to their special needs, as shown below:

Figure 4. AT32F403A SRAM and zero-wait configuration

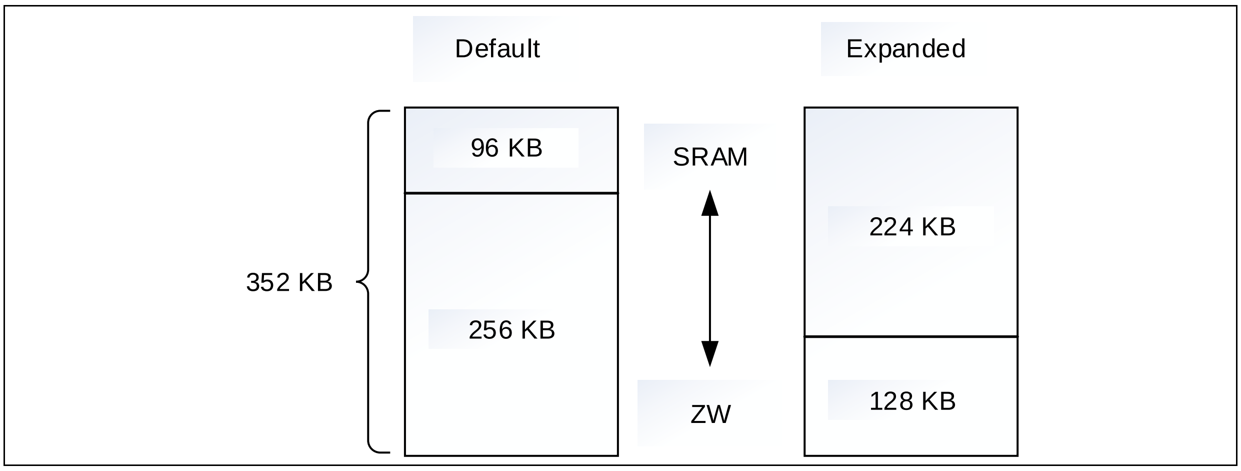


Figure 5. AT32F413 SRAM and zero-wait configuration

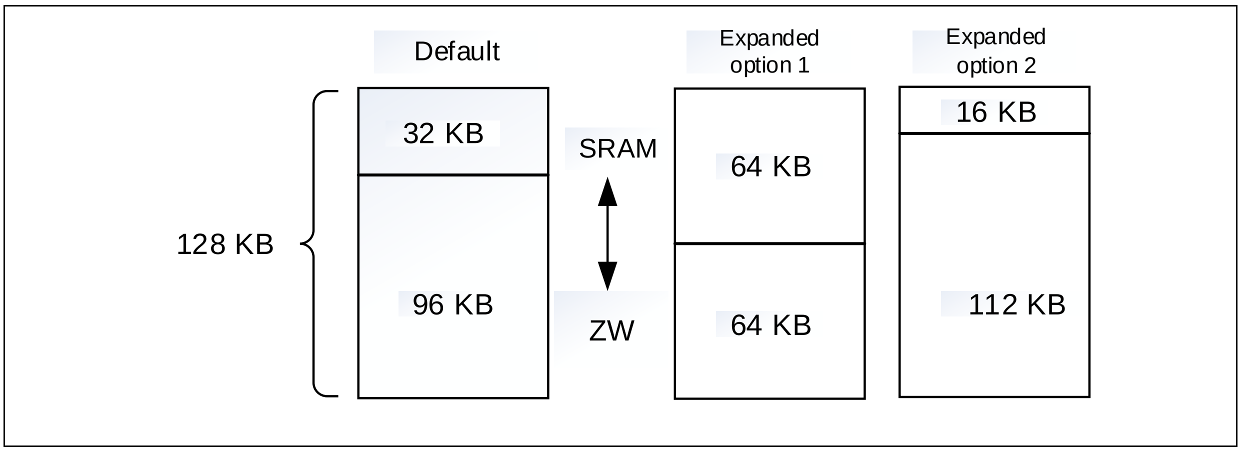
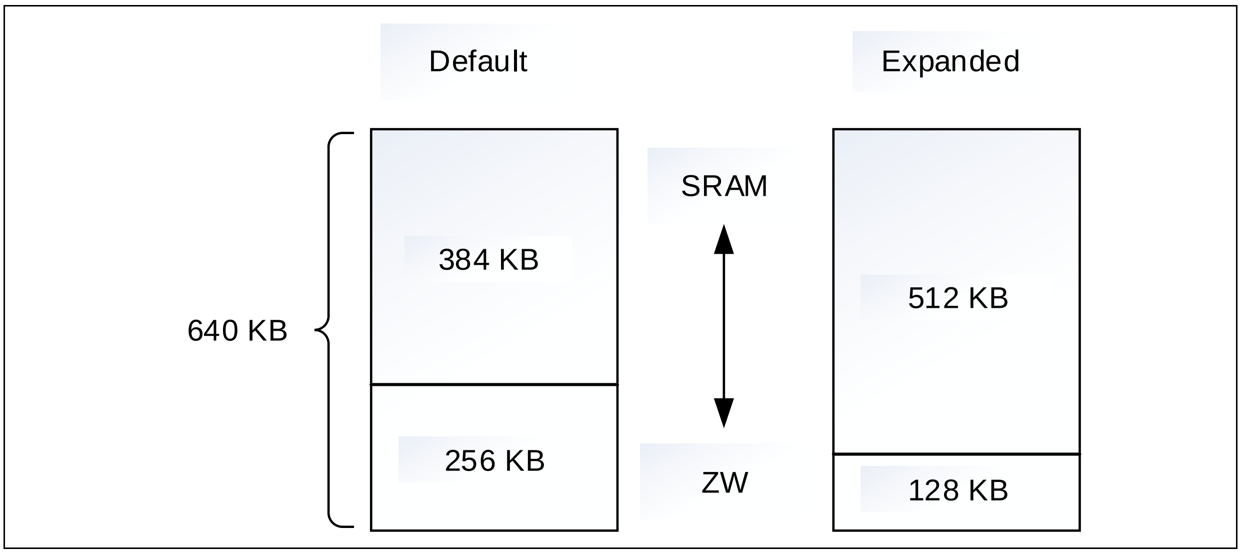


Figure 6. AT32F435 SRAM and zero-wait configuration



3.1 SRAM demand analysis and trade-offs

Zero-wait refers to a preloaded storage area located between Flash and SRAM in order to achieve quick start and high-speed running. After power-on, the hardware will automatically copy the zero-wait-sized data from Flash's start address to the zero-wait area to ensure that the system can quickly access to the pre-loaded data during running.

As a storage area during system run, SRAM is used to store instructions, data, stack and others.

From this point of view, the larger the size of these two areas, the better they are. To make appropriate allocation and trade-offs, it is recommended to expand SRAM configuration in the following conditions:

1) When a bin file is much larger than the size of zero-wait area;
2) When the frequently-used data is greater than or close to the size of zero-wait area;
3) There are a large number of process scheduling and temporary variables in OS during system run;
4) There are many nested functions and interrupts and temporary variables during system run.

3.2 How to expand SRAM configuration

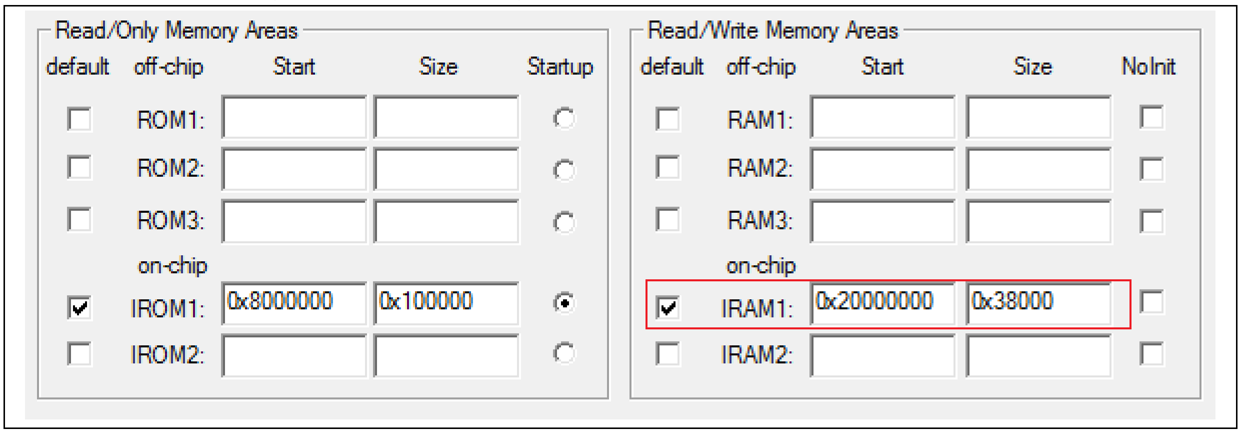
For SRAM expansion, please refer to *AN0026_Extending_SRAM_in_User's_Program*.

3.3 Cautions on SRAM expansion

When using Keil as a development tool, the default SRAM size is compiled and linked as long as Pack file is selected. However, the hardware SRAM resources are adjustable while only one setting can be defaulted in Pack, resulting in some differences between actual and default configurations. In this example, we use AT32F403A series MCU, while other series are also similar in this regard:

In project, click on *Options for Target* and select *Target*, and the following window will appear:

Figure 7. SRAM configuration window in Keil environment



Read/Only Memory Areas

default	off-chip	Start	Size	Startup
☐	ROM1:			○
☐	ROM2:			○
☐	ROM3:			○
	on-chip			
✓	IROM1:	0x8000000	0x100000	●
☐	IROM2:			○

Read/Write Memory Areas

default	off-chip	Start	Size	NoInit
☐	RAM1:			☐
☐	RAM2:			☐
☐	RAM3:			☐
	on-chip			
✓	IRAM1:	0x20000000	0x38000	☐
☐	IRAM2:			☐

The above red outline shows the default SRAM start address and size designated by AT32 Pack file, indicating that the software code uses the default SRAM size 0x38000 (224KB) during compile and link. According to previous sections, we know that the SRAM size of AT32F403A series MCU is 0x18000 (96 KB) by default, which can be expanded to 224 KB. Thus, there are two situations to be considered when using At32 Pack file.

a) When SRAM expansion mode (224 KB) is used, the running address linked after software compiling is in consistent with its physical address, so no problem would happen.

b) When SRAM default mode (96 KB) is used, no problem would occur either as long as the RAM requirements in software code are less than 96 KB or when SRAM requirements are between 96 KB and 224 KB. However, because the running address linked does not match the actual physical address, an error would take place when this bin file is downloaded to MCU to run, which is what we should pay attention to during use. To avoid this happen, either change the above SRAM size marked in red to 0x18000 or expand SRAM to 224 KB to ensure that the physical SRAM size is the same as the one set in Keil.

4 Scatter loading

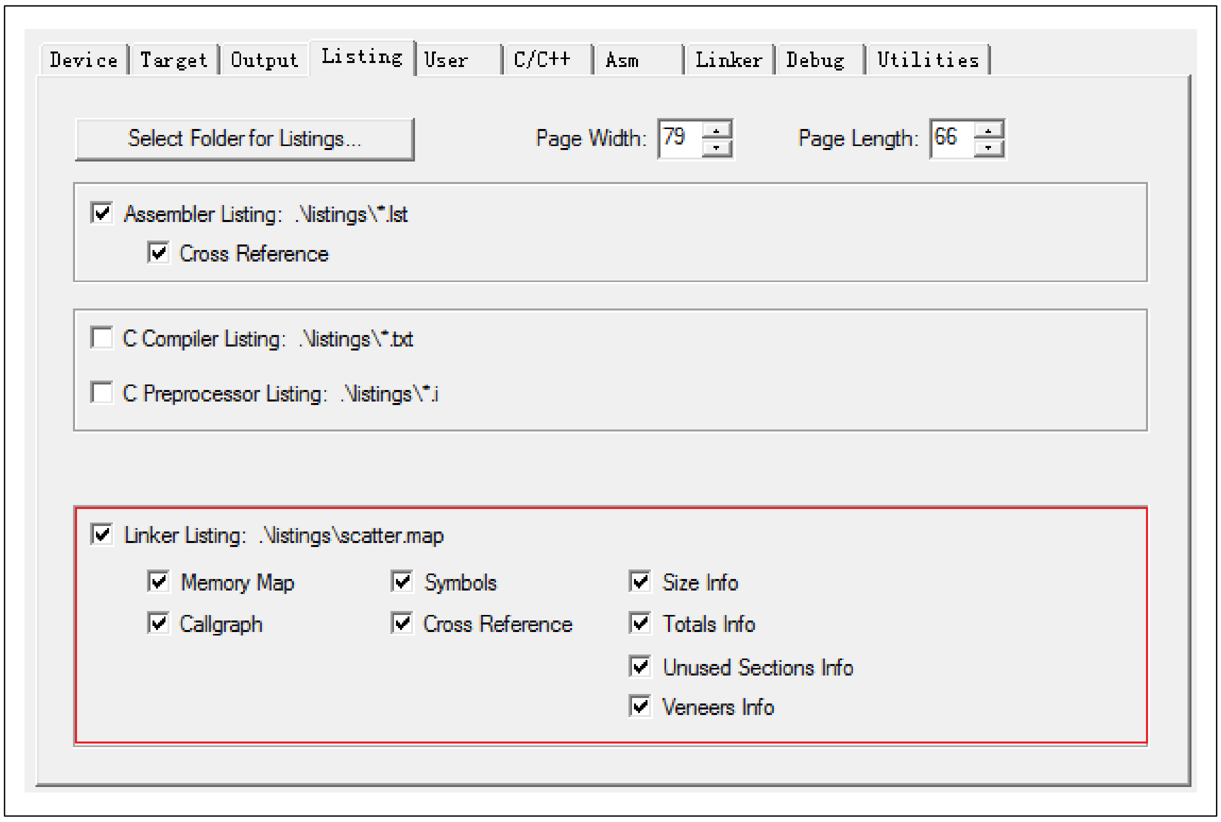
Code scatter loading can be used to improve execution frequency and performance by adjusting the link structure of execution files. This requires the user to gain a good command of the address distribution and size of storage space.

4.1 View map relations

The purpose of learning map relation is to have a better understanding of the composition and distribution structure of executable files so as to be prepared well for subsequent running and scatter loading.

In Keil, after compiling source code, the linker would generate a map file while linking various target files. This file mainly contains cross-link information, so that the user can understand cross reference among various symbols and get complete information such as Code, RO Data, RW Data and ZI Data. This includes “Cross Reference”, “Unused Sections Info”, “Symbols”, “Memory Map’, “Totals Info” and so on. Keil project configuration generates map information by default, with the detailed configuration mode as follows:

Figure 8. Keil compile information output option configuration



Page Width: 79
Page Length: 66

[✓] Assembler Listing: .\listings*.lst
[✓] Cross Reference

[ ] C Compiler Listing: .\listings*.txt
[ ] C Preprocessor Listing: .\listings*.i

[✓] Linker Listing: .\listings\scatter.map
[✓] Memory Map
[✓] Symbols
[✓] Size Info
[✓] Callgraph
[✓] Cross Reference
[✓] Totals Info
[✓] Unused Sections Info
[✓] Veneers Info

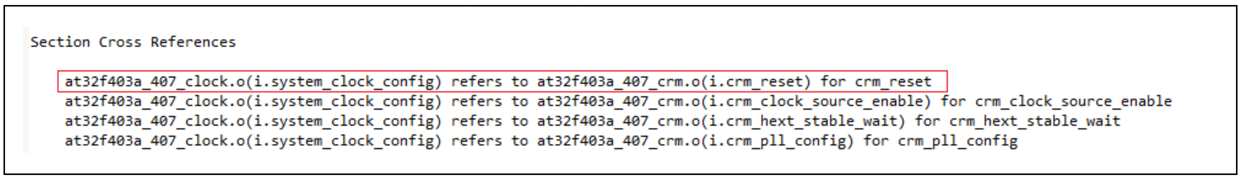
Select *Options for Target* and click on *Listing*, the user can choose which kind of information needs to be output in a map file according to their needs.

After compiling and generating a map file, the user can open and view map information with Notepad.

1) Cross reference

This section gives a full list of symbol reference between *.o files. Because *.o files are compiled and generated by .s or c/c++ source file, it is independent between files and sections with a file, which are linked by a linker through cross reference between them, with the link details recorded here, as shown in Figure 9.

Figure 9. Symbol reference



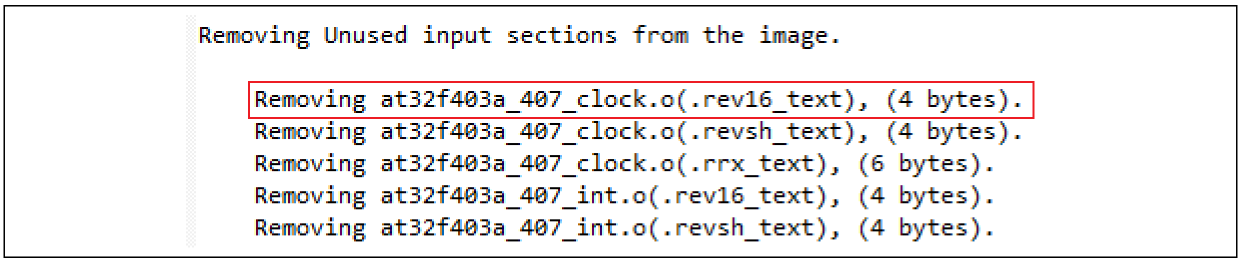
```
Section Cross References

    at32f403a_407_clock.o(i.system_clock_config) refers to at32f403a_407_crm.o(i.crm_reset) for crm_reset
    at32f403a_407_clock.o(i.system_clock_config) refers to at32f403a_407_crm.o(i.crm_clock_source_enable) for crm_clock_source_enable
    at32f403a_407_clock.o(i.system_clock_config) refers to at32f403a_407_crm.o(i.crm_hext_stable_wait) for crm_hext_stable_wait
    at32f403a_407_clock.o(i.system_clock_config) refers to at32f403a_407_crm.o(i.crm_pll_config) for crm_pll_config
```

The sentence in red means that *system_clock_config* function in *at32f403a_407_clock.c* file calls the *crm_reset* function in *at32f403a_407_crm.c* file.

2) Unused sections info

This part lists the unused sections in a project found during link. These unused sections will be removed, which means that the unused sections are not added into *.axf files, instead of being deleted from *.o files. This operation can prevent these unused data from occupying program space, as shown in Figure 10.

Figure 10. Remove unused sections



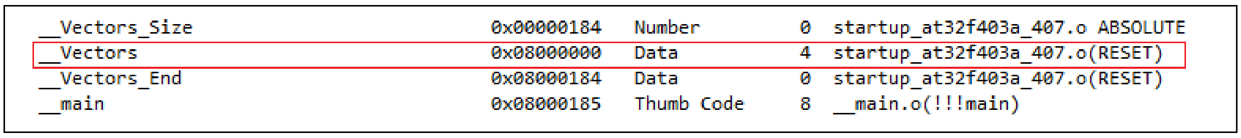
```
Removing Unused input sections from the image.

    Removing at32f403a_407_clock.o(.rev16_text), (4 bytes).
    Removing at32f403a_407_clock.o(.revsh_text), (4 bytes).
    Removing at32f403a_407_clock.o(.rrx_text), (6 bytes).
    Removing at32f403a_407_int.o(.rev16_text), (4 bytes).
    Removing at32f403a_407_int.o(.revsh_text), (4 bytes).
```

These 4 bytes of unused sections in the at32f403a_407.o target file will not be linked to the final *.axf file.

3) Symbols

It is shown in Figure 11.

Figure 11. Symbols



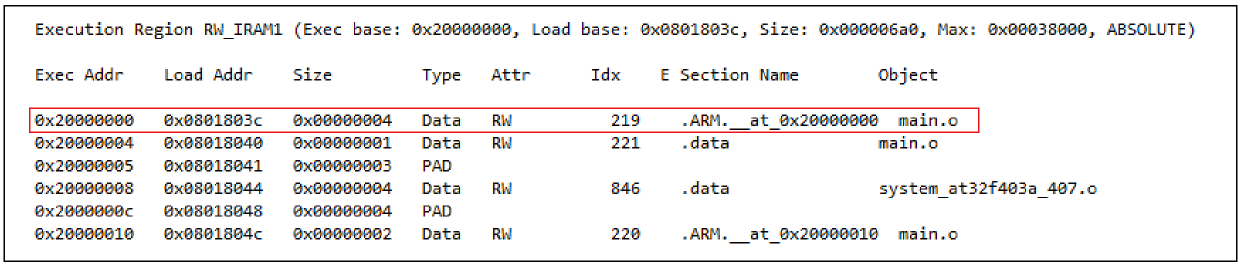
```
__Vectors_Size                  0x00000184   Number         0  startup_at32f403a_407.o ABSOLUTE
__Vectors                       0x08000000   Data           4  startup_at32f403a_407.o(RESET)
__Vectors_End                   0x08000184   Data           0  startup_at32f403a_407.o(RESET)
__main                          0x08000185   Thumb Code     8  __main.o(!!!main)
```

__Vectors is the symbol name, and refers to the vector table address here; 0x08000000 means the corresponding address of the *__Vectors* symbol.

4) Memory map

This table gives a full list of the storage address (loading address) of each section, and execution address and size for each section, as shown in Figure 12.

Figure 12. Memory distribution



```
Execution Region RW_IRAM1 (Exec base: 0x20000000, Load base: 0x0801803c, Size: 0x000006a0, Max: 0x00038000, ABSOLUTE)

Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name        Object

0x20000000   0x0801803c   0x00000004   Data   RW         219    .ARM.__at_0x20000000  main.o
0x20000004   0x08018040   0x00000001   Data   RW         221    .data               main.o
0x20000005   0x08018041   0x00000003   PAD
0x20000008   0x08018044   0x00000004   Data   RW         846    .data               system_at32f403a_407.o
0x2000000c   0x08018048   0x00000004   PAD
0x20000010   0x0801804c   0x00000002   Data   RW         220    .ARM.__at_0x20000010  main.o
```

The part in red means that the RW-data specified as the .ARM.__at_0x20000000 section in the *main.o* target file is stored in address 0x0801803C of ROM. After the system runs, load its initialization data to address 0x20000000 of RAM and then execute, with the length of 0x4 bytes. Combined with the example demo code, this means that after compiling all the variables of rw_temp with the specified address in main.c, the initialization value 10 is stored in address 0x0801803C in Flash, and after system running, it is loaded to address 0x20000000 of RAM.

From this map file, we know that it provides detailed information about the storage address of each section and the load address of RW-data in RAM. But these load addresses can be configured by modifying the scatter file *.sct through the use of (__attribute__) in a code. More sophisticated control can be implemented according to the frequency of access to code or data so as to achieve the optimization of code load structures.

4.2 How to implement scatter loading

To implement scatter loading, there are two options in Keil. The first one is to use attribute in a code, mainly used in the scatter loading for functions, arrays and variables. The other is to modify .sct files, mainly used in the scatter loading of the whole target files. It is recommended to use attribute to achieve scatter loading because it is flexible, easy-to-read and convenient. Refer to the following content on how to scatter load the functions, arrays, variables and files. For demo, please refer to project\at_start_f403a\scatter.

Method 1: Load a function to a specified location

For example, the *button_isr* function in the *main.c* file is loaded to 0x08018000, and the user can specify the *button_isr* function at the function definition in .c file.

```
void button_isr(void) __attribute__ ((section(".ARM.__at_0x08018000")));
```

Method 2: Load array to a specified location

```
uint8_t rw_data[2] __attribute__ ((section(".ARM.__at_0x20000010"))) = {0x1, 0x2};
```

Method 3: Load a variable to a specified location

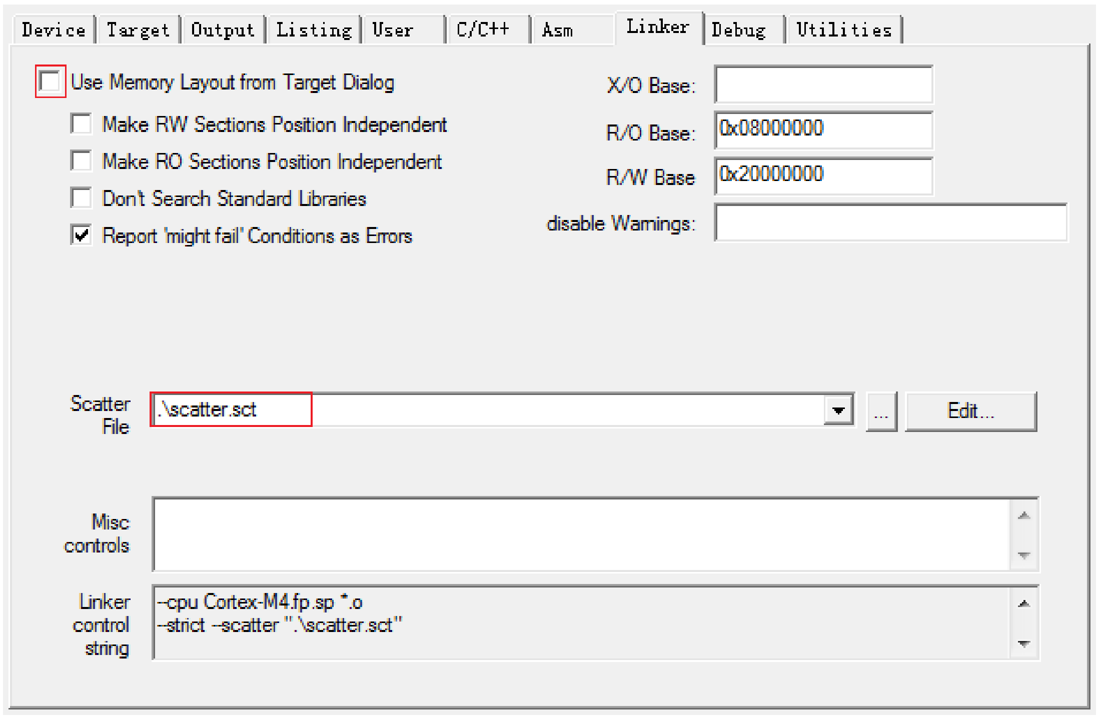
For example, c code can be directly modified as follows:

```
const uint32_t ro_temp __attribute__ ((section(".ARM.__at_0x08010000"))) = 10;      //RO
uint32_t rw_temp __attribute__ ((section(".ARM.__at_0x20000000"))) = 10;            //RW
```

Method 4: Load a target file to a specified location

When performing scatter load in Keil by modifying .sct files, the project configuration should be adjusted as follows. For more information on how to modify .sct file, please refer to .sct grammar rule.

Figure 13. sct file edit in Keil



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

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

Scatter File: .\scatter.sct

Misc controls:

Linker control string:
--cpu Cortex-M4.fp.sp *.o
--strict --scatter ".\scatter.sct"

Where, *Use Memory Layout from Target Dialog* is ticked by default, so it should be unticked. Go to *Scatter File* and click on *Edit* to start editing .sct file.

For example, scatter load the target file of at32f403a_407_board.c.

```
; *************************************************************
; *** Scatter-Loading Description File generated by uVision ***
; *************************************************************

LR_IROM1 0x08000000 0x00020000  {    ; load region size_region
  ER_IROM1 0x08000000 0x00020000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
   .ANY (+XO)
  }
  RW_IRAM1 0x20000000 0x00038000  {  ; RW data
   .ANY (+RW +ZI)
  }
}

LR_IROM2 0x08020000 0x00020000  {    ; load region size_region
  ER_IROM2 0x08020000 0x00020000  {  ; load address = execution address
   at32f403a_407_board.o (+RO)
  }
}
```

The LR_IROM2 area indicates that the target file compiled and generated by at32f403a_407_board.c is loaded to address 0x08020000, and the LR_IROM2 area is 0x00020000. In other words, after linking the target file compiled and generated by at32f403a_407_board.c to 128 KB Flash, the area size is 128 KB.

5 Increase maximum frequency

Increasing maximum frequency is mainly aimed at improving clock frequency of MCU core and the corresponding buses. The clock period for each instruction executed by core is fixed, so the improvement of max frequency can reduce the executing time of each instruction. That's why we can improve system performance by increasing max frequency.

5.1 Configuration method

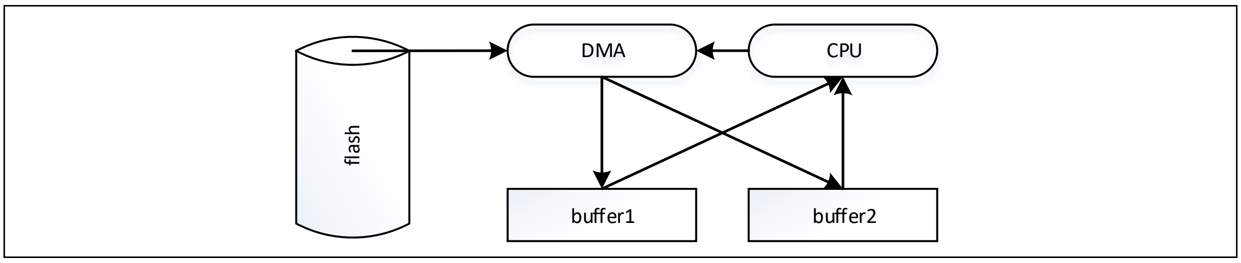
Please refer to AN0082_AT32F403A_407_CRM_Start_Guide for more information on configuration.

6 DMA read Flash

Based on previous sections, AT32F4xx series MCU has a certain size of zero-wait area. When MCU is powered, it would automatically load a zero-wait-sized data from Flash start address into the zero-wait storage area to ensure rapid read operation. But the data outside Flash zero-wait area can only be read through Flash access process, which is slower in speed compared to zero-wait area or SRAM, and the larger the data, the slower it is. To cope with this issue, we provide the following proposals to achieve rapid access to data outside Flash zero-wait area.

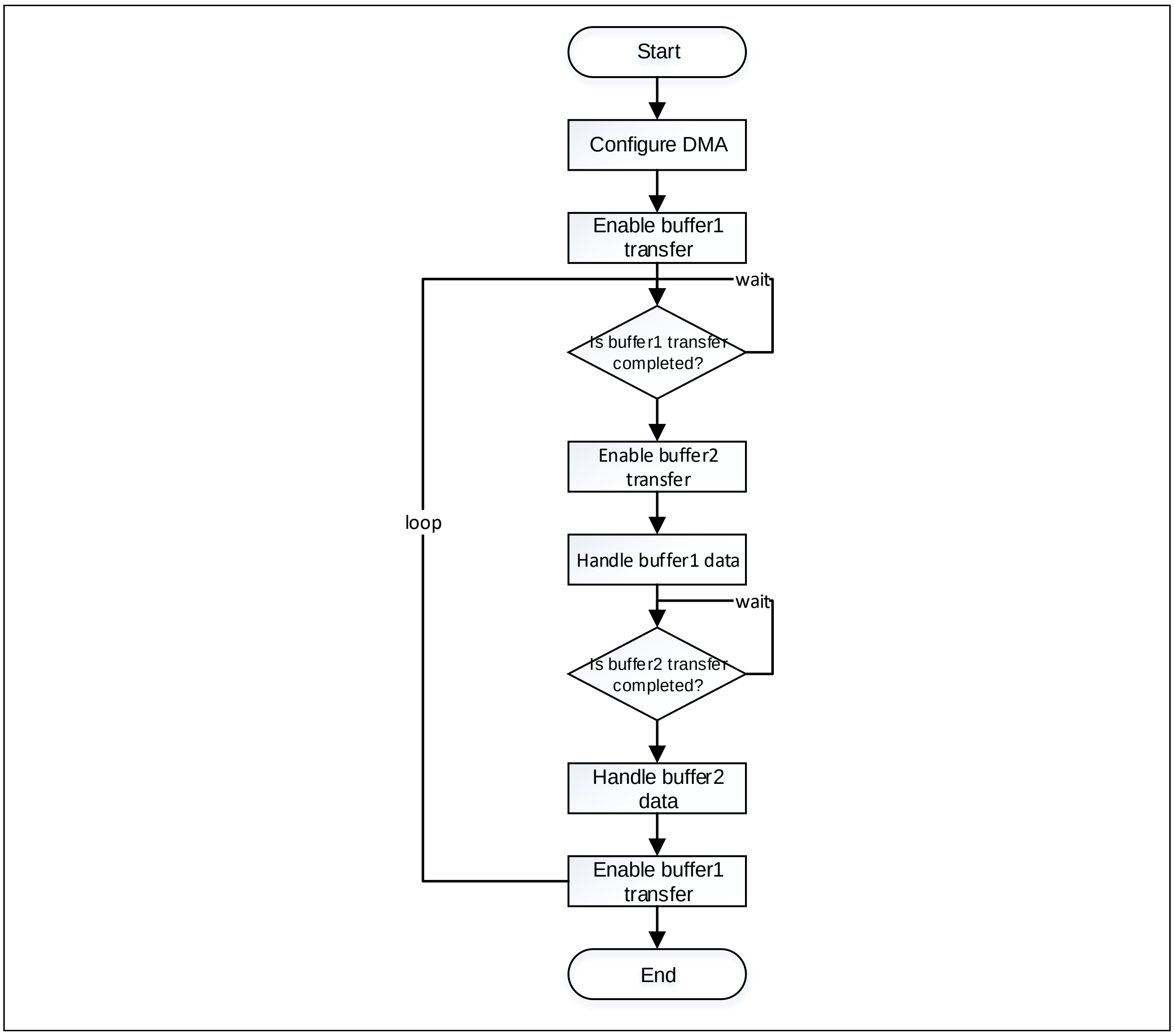
The model is shown below:

Figure 14. DMA dual buffer model



To minimize the impact on CPU execution and on-going data processing, read Flash operation is performed by DMA and alternate use of dual buffer. The general work flow is as follows:

Figure 15. Handling process of DMA dual buffer



The use of this model is subject to specific needs. For example, the Flash data block address, buffer size configuration and DMA configuration demo, please refer to project\at_start_f403a\double_buffer.

7 Revision history

Table 1. Document revision history

Date	Version	Revision note
2021.12.10	2.0.0	Updated contents and codes to V2.