摘 要

通常情况下，MCU 绝大数代码都在主程序空间执行，主程序空间通常使用 FLASH 或 ROM 来实现。内部 RAM 存放变量，软件栈等掉电不需要保存信息。对于特定应用，RAM 中执行代码有以下优势。

对于多数 MCU 来说在 RAM 中执行程序的功耗比在 FLASH 中小，对于大部分运行时间执行较小代码量场景可以降低功耗。

可以操作整个主程序空间，在 RAM 中执行程序可以擦除整个 FLASH 空间并写入新的代码。

此外还具备执行效率高，寿命长优点。

本文档介绍和说明在 MDK 开发环境下将代码重定向到 RAM 中执行的方法。本文档使用开发环境位 keil UVision，版本号为 V5.37.0.0。本文档介绍方法适用于芯海科技 MCU。

适用范围

类型	适用产品型号或系列	说明
MCU	所有 ARM 内核的 MCU，包括：CS32F0 系列，CS32F1 系列，CS32L0 系列等	小容量产品的可以参照文档放部分函数在 SRAM 中

版 本

历史版本	修改内容	日期
V1.0	初版生成	2022-12-12

目 录

1. 概述

本章节描述 MDK 环境下链接文件和映射文件相关概念，为后续章节中设置方法做准备。

本文中代码和参考示例以 CS32F103 为例说明。本文中使用硬件和工具信息如表 1 所示。

表 1 使用环境说明

类别	名称	下载路径
MCU	CS32F103	/
开发板	CS32F103_LQFP48 开发板 V1.0	/
下载工具	Jlink	/
SDK	ChipSea.CS32F1XX_DFP.2.0.3	https://www.chipsea.com/product/details/?choice_id=1068#sheji
编译工具	Keil uVision V5.37.0.0	/

1.1. 链接文件

编译器在生成可执行文件时，先将.c 文件编译成.o 文件，然后将多个.o 文件链接成可执行文件。在链接过程中，需要使用链接文件用来指定链接的行为。在 keil 环境下使用分散加载文件（.sct）。

默认情况下 keil 会自动生成分散加载文件。用户需要满足自己特定需求时如部分函数放到 RAM 执行，需要用户自定义分散加载文件（sct）。操作步骤如下：

1．魔术棒（option for target）->linker 选项。

2．去掉 Use Memory Layout from Target Dialog 选项。

3．选择自己的.sct 文件或者点击 Edit 对已有文件进行操作。

关于分散加载文件描述参考 MDK 帮助文件中 armlink reference 中描述。

1.2. 映射文件

MDK 生成的 MAP 文件可以根据用户配置生成不同内容，用户可以查看 map 文件查询所需的信息。map 文件的内容大致分为 5 大类：

1．Section Cross References：模块、段(入口)交叉引用；

2．Removing Unused input sections from the image：移除未使用的模块；

3．Image Symbol Table：映射符号表；

4．Memory Map of the image：内存（映射）分布；

5．Image component sizes：存储组成大小。

map 文件描述参考 MDK 帮助文档中 compiler 用户手册中描述。

1.3. 示例代码

创建一个 test.c 文件，并实现如下三个测试函数，在主函数中调用此函数。

```
void test_func1(void)
{
    printf("test_func1.\r\n");
}
void test_func2(void)
{
    printf("test_func2 .\r\n");
}
void test_func3(void)
{
    printf("test_func3 .\r\n");
}
```

编译后查看其 map 文件找到上述三个函数相关描述。

```
0x080004f8   0x080004f8   0x00000010   Code   RO          56    .text.test_func1    test.o
0x08000508   0x08000508   0x00000010   Code   RO          58    .text.test_func2    test.o
0x08000518   0x08000518   0x00000010   Code   RO          60    .text.test_func3    test.o
```

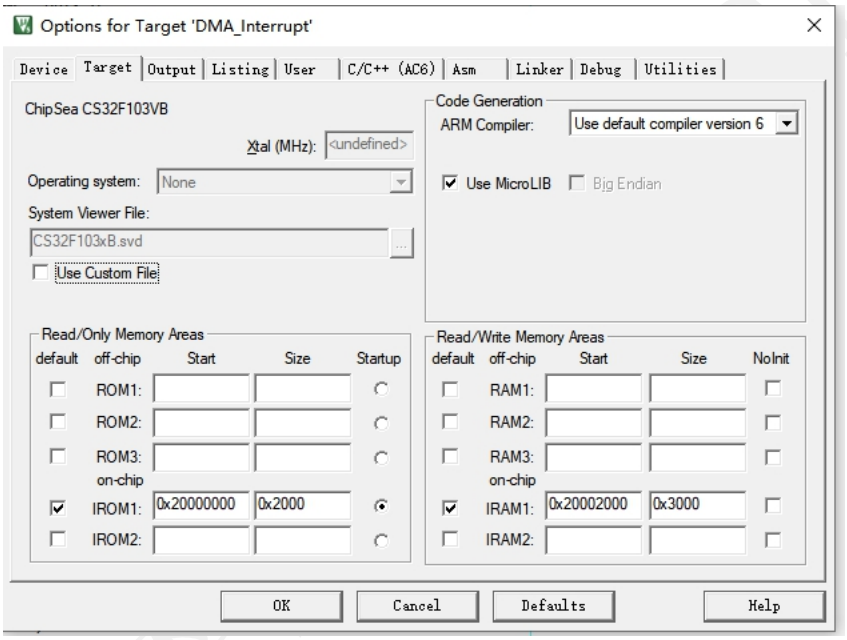
2. RAM 执行程序方法

本章节描述将程序下载到 RAM 中，并执行程序。不同架构的处理器实现 RAM 执行程序存在差异，但基本思路一致，修改链接器设置，将代码链接到 RAM 区间，并更改向量表到 RAM 中。如何执行代码有两种方式：一种是设置 boot 脚或选项字节使处理器从 RAM 执行。另一种是设置 PC 指针执行程序。

2.1. 链接器修改

可以通过 MDK 配置或修改 sct 文件将代码链接到 RAM 中。

修改 MDK 配置方法如下图所示。



修改分散加载文件 sct 如下。

```
LR_IROM1 0x20000000 0x00002000  {    ; load region size_region
  ER_IROM1 0x20000000 0x00002000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
   .ANY (+XO)
  }
  RW_IRAM1 0x20002000 0x00003000  {  ; RW data
   .ANY (+RW +ZI)
  }
}
```

2.2. 向量表地址修改

Cortex-M 要求向量表放到零地址启动，所以设置 SRAM 启动时需要将 SRAM 初始地址防止向量表。在芯海科技提供的 SDK 中通过设置 VECT_TAB_SRAM 宏修改向量表基地址。

编译经过修改后文件检查 map 文件，代码定位到 SRAM 执行了。

```
Memory Map of the image

  Image Entry point : 0x200000ed

  Load Region LR_IROM1 (Base: 0x20000000, Size: 0x000007c8, Max: 0x00002000, ABSOLUTE)

    Execution Region ER_IROM1 (Exec base: 0x20000000, Load base: 0x20000000, Size: 0x000007bc,
Max: 0x00002000, ABSOLUTE)

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

    0x20000000   0x20000000   0x000000ec   Data   RO          611    RESET
startup_cs32f103xb.o
    0x200000ec   0x200000ec   0x00000000   Code   RO          632  * .ARM.Collect$$$$00000000
mc_w.l(entry.o)
    0x200000ec   0x200000ec   0x00000004   Code   RO          641    .ARM.Collect$$$$00000001
mc_w.l(entry2.o)
    0x200000f0   0x200000f0   0x00000004   Code   RO          644    .ARM.Collect$$$$00000004
mc_w.l(entry5.o)
    0x200000f4   0x200000f4   0x00000000   Code   RO          646    .ARM.Collect$$$$00000008
mc_w.l(entry7b.o)
    0x200000f4   0x200000f4   0x00000000   Code   RO          648    .ARM.Collect$$$$0000000A
mc_w.l(entry8b.o)
    0x200000f4   0x200000f4   0x00000008   Code   RO          649    .ARM.Collect$$$$0000000B
mc_w.l(entry9a.o)
    0x200000fc   0x200000fc   0x00000000   Code   RO          651    .ARM.Collect$$$$0000000D
mc_w.l(entry10a.o)
    0x200000fc   0x200000fc   0x00000000   Code   RO          653    .ARM.Collect$$$$0000000F
mc_w.l(entry11a.o)
    0x200000fc   0x200000fc   0x00000004   Code   RO          642    .ARM.Collect$$$$00002712
mc_w.l(entry2.o)
    0x20000100   0x20000100   0x00000024   Code   RO          612    .text
startup_cs32f103xb.o
    0x20000124   0x20000124   0x00000024   Code   RO          656    .text
mc_w.l(init.o)
    0x20000148   0x20000148   0x00000002   Code   RO            8    .text.BusFault_Handler
cs32f10x_it.o
    0x2000014a   0x2000014a   0x00000002   PAD
    0x2000014c   0x2000014c   0x00000002   Code   RO           14    .text.DebugMon_Handler
cs32f10x_it.o
    0x2000014e   0x2000014e   0x00000002   PAD
    0x20000150   0x20000150   0x00000002   Code   RO            4    .text.HardFault_Handler
cs32f10x_it.o
```

```
    0x20000152   0x20000152   0x00000002   PAD
    0x20000154   0x20000154   0x00000002   Code   RO            6    .text.MemManage_Handler
cs32f10x_it.o
    0x20000156   0x20000156   0x00000002   PAD
    0x20000158   0x20000158   0x00000002   Code   RO            2    .text.NMI_Handler
cs32f10x_it.o
    0x2000015a   0x2000015a   0x00000002   PAD
    0x2000015c   0x2000015c   0x00000002   Code   RO           16    .text.PendSV_Handler
cs32f10x_it.o
    0x2000015e   0x2000015e   0x00000002   PAD
    0x20000160   0x20000160   0x00000002   Code   RO           12    .text.SVC_Handler
cs32f10x_it.o
    0x20000162   0x20000162   0x00000002   PAD
    0x20000164   0x20000164   0x00000002   Code   RO           18    .text.SysTick_Handler
cs32f10x_it.o
    0x20000166   0x20000166   0x00000002   PAD
    0x20000168   0x20000168   0x00000110   Code   RO          619    .text.SystemInit
system_cs32f10x.o
    0x20000278   0x20000278   0x00000002   Code   RO           20    .text.USART3_IRQHandler
cs32f10x_it.o
    0x2000027a   0x2000027a   0x00000002   PAD
    0x2000027c   0x2000027c   0x00000002   Code   RO           10    .text.UsageFault_Handler
cs32f10x_it.o
    0x2000027e   0x2000027e   0x00000002   PAD
    0x20000280   0x20000280   0x00000002   Code   RO           45    .text.assert_failed
main.o
    0x20000282   0x20000282   0x00000002   PAD
    0x20000284   0x20000284   0x00000018   Code   RO           31    .text.fputc          log.o
    0x2000029c   0x2000029c   0x00000132   Code   RO          282    .text.gpio_mode_config
cs32f10x_gpio.o
    0x200003ce   0x200003ce   0x00000002   PAD
    0x200003d0   0x200003d0   0x00000068   Code   RO           29    .text.log_init       log.o
    0x20000438   0x20000438   0x0000001e   Code   RO           43    .text.main
main.o
    0x20000456   0x20000456   0x00000002   PAD
    0x20000458   0x20000458   0x0000009e   Code   RO          418    .text.rcu_clk_freq_get
cs32f10x_rcu.o
    0x200004f6   0x200004f6   0x00000002   PAD
    0x200004f8   0x200004f8   0x00000010   Code   RO           56    .text.test_func1
test.o
    0x20000508   0x20000508   0x00000010   Code   RO           58    .text.test_func2
test.o
    0x20000518   0x20000518   0x00000010   Code   RO           60    .text.test_func3
test.o
    0x20000528   0x20000528   0x0000013a   Code   RO          550    .text.usart_init
cs32f10x_usart.o
    0x20000662   0x20000662   0x0000000e   Code   RO          660    i.__scatterload_copy
mc_w.l(handlers.o)
    0x20000670   0x20000670   0x00000002   Code   RO          661    i.__scatterload_null
mc_w.l(handlers.o)
    0x20000672   0x20000672   0x0000000e   Code   RO          662    i.__scatterload_zeroinit
mc_w.l(handlers.o)
    0x20000680   0x20000680   0x00000024   Code   RO          635    i.puts
```

```
mc_w.l(puts.o)
    0x200006a4   0x200006a4   0x00000004   Data   RO          422    .rodata.ADC_pdiv_table
cs32f10x_rcu.o
    0x200006a8   0x200006a8   0x00000010   Data   RO          421    .rodata.APBAHB_pdiv_table
cs32f10x_rcu.o
    0x200006b8   0x200006b8   0x00000005   Data   RO           47    .rodata.str1.1
main.o
    0x200006bd   0x200006bd   0x00000029   Data   RO           62    .rodata.str1.1
test.o
    0x200006e6   0x200006e6   0x00000059   Data   RO          296    .rodata.str1.1
cs32f10x_gpio.o
    0x2000073f   0x2000073f   0x0000005a   Data   RO          576    .rodata.str1.1
cs32f10x_usart.o
    0x20000799   0x20000799   0x00000003   PAD
    0x2000079c   0x2000079c   0x00000020   Data   RO          659    Region$$Table
anon$$obj.o


    Execution Region RW_IRAM1 (Exec base: 0x20002000, Load base: 0x200007c0, Size: 0x00000408,
Max: 0x00003000, ABSOLUTE)

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

    0x20002000   0x200007c0   0x00000004   Data   RW          655    .data
mc_w.l(stdout.o)
    0x20002004   0x200007c4   0x00000004   PAD
        0x20002008         -         0x00000400   Zero   RW             609      STACK
startup_cs32f103xb.o
```

2.3. 下载设置

通过 MDK 将代码下载到 SRAM 并执行。

本文中使用 JLINK 进行下载操作，设置步骤如下。

1．魔术棒（option for target）->Debug 选项。取消 Load Application at Startup 选项，并在 initialization File 中加载 ini 文件。通过设置 ini 文件配置相关参数，可以不通过 boot 脚将启动方式设为 SRAM，debug 用户程序。具体设置如图 1 所示。

ini 内容如下所示。

```
FUNC void Setup (void) {
SP = _RDWORD(0x20000000); // Setup Stack Pointer
PC = _RDWORD(0x20000004); // Setup Program Counter
XPSR = 0x01000000; // Set Thumb bit
_WDWORD(0xE000ED08, 0x20000000); // Setup Vector Table Offset Register
}

LOAD %L INCREMENTAL // Download to RAM
Setup();

g, main
```

2．设置通讯协议，调试速率以及 flash 擦除方式为 Do not Erase 等。注意将 Programming Algorithm 地址设置为 RAM 地址，且 RAM for Algorithm 地址同上述地址不冲突。如图 2 和图 3 所示。

3．设置 utilities 界面选项取消 Update Target before Debugging 选项，如图 4 所示。

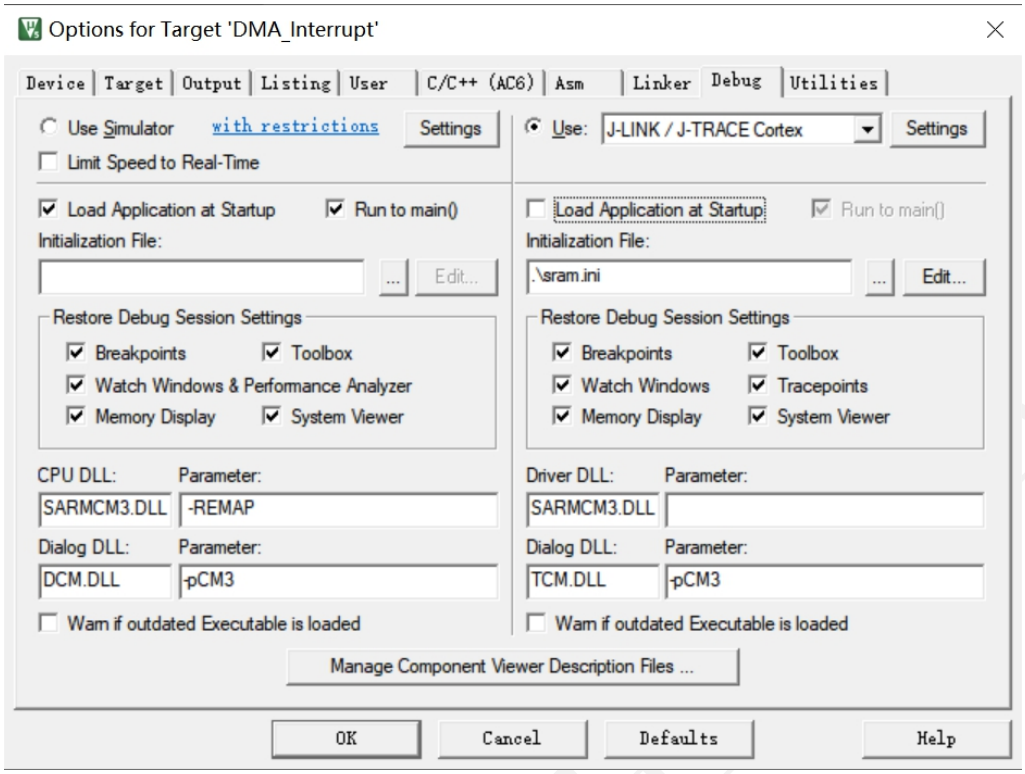


图 1　ini 文件配置相关参数具体设置

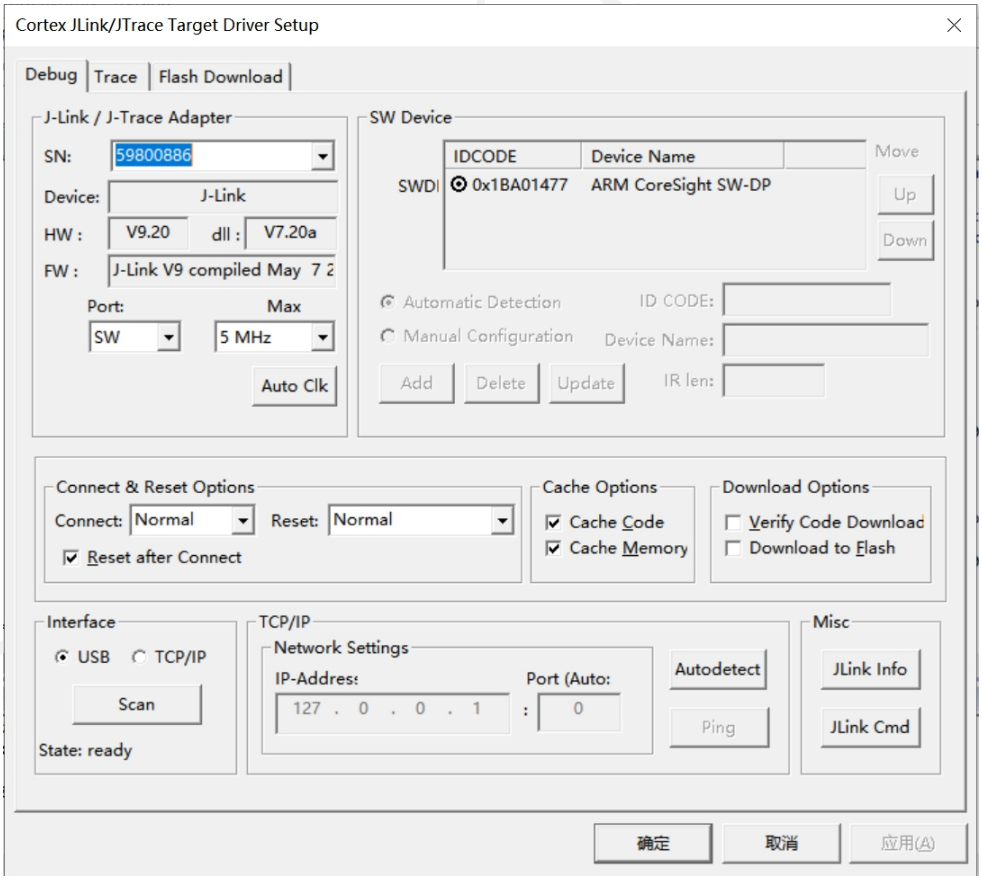


图 2

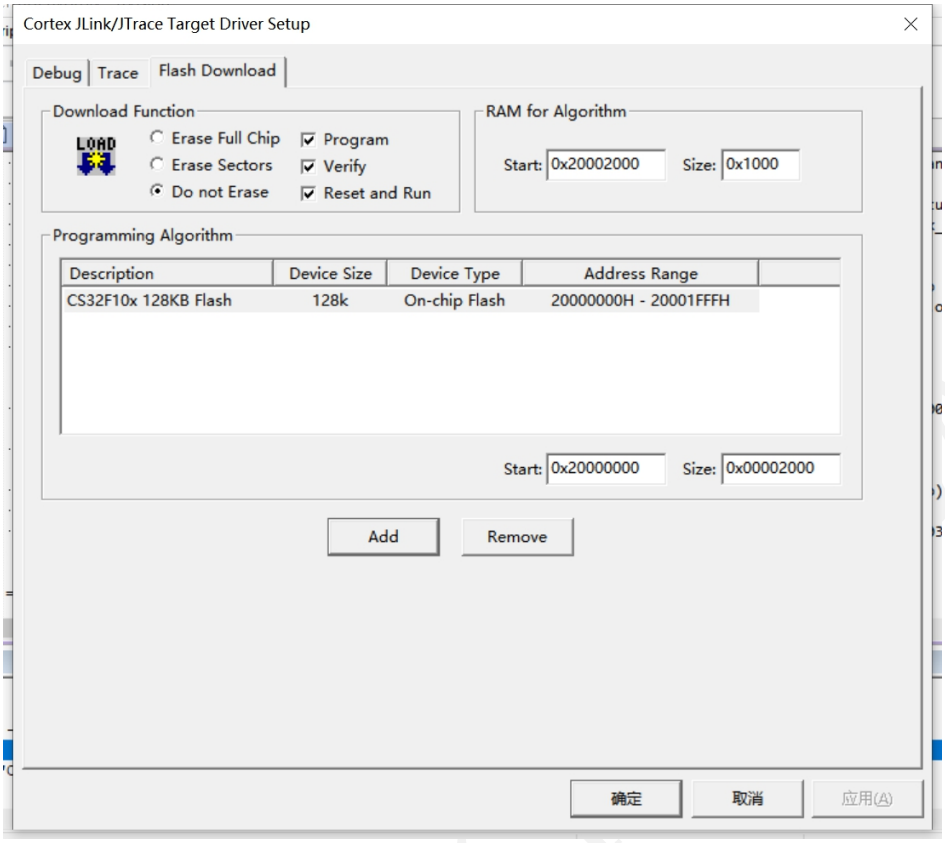


图 3

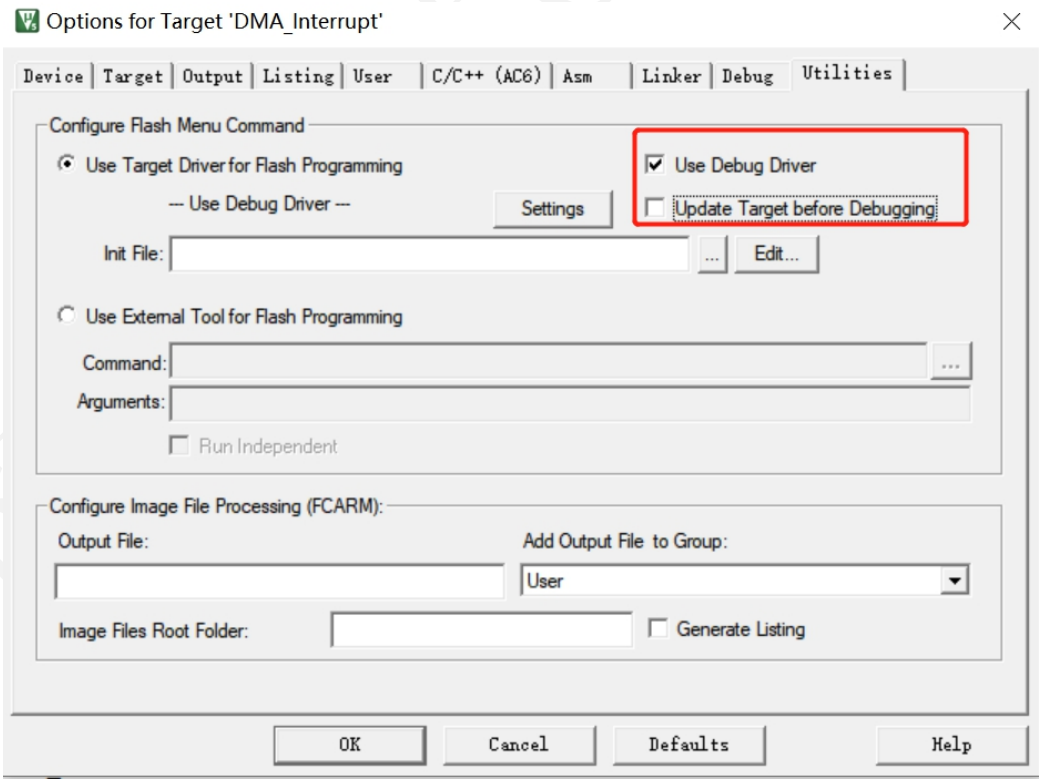


图 4　取消 Update Target before Debugging 选项

3. 关键函数重映射到 RAM 中方法

本章节描述将关键函数放到 RAM 中执行的方法。本章节介绍四种方法实现该功能。

3.1. 自定义 section 指定函数

使用__attribute__((section("UserSectionName"))) 语法来修饰函数，将其放到自定义程序段中。这种方法适用于重定向单个关键函数。下述代码演示将 test_func1 函数放到 SRAM 中。

```
__attribute__((section(".testFunc"))) void test_func1(void)
{
    printf("test_func1.\r\n");
}
```

分散加载文件如下。

```
LR_IROM1 0x08000000 0x00005000  {    ; load region size_region
  ER_IROM1 0x08000000 0x00005000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
   .ANY (+XO)
  }
  RW_IRAM1 0x20000000 0x00005000  {  ; RW data
   .ANY (+RW +ZI)
   *(.testFunc)
  }
}
```

编译后 map 文件中 test 地址如下。

```
test_func1                              0x20000001   Thumb Code    16  test.o(.testFunc)
test_func2                              0x080004f9   Thumb Code    16  test.o(.text.test_func2)
test_func3                              0x08000509   Thumb Code    16  test.o(.text.test_func3)
```

使用#program 语法来修饰函数可以将同文件中多个连续函数放到自定义段中。参考代码如下。其中分散加载文件同上述一致。

```
#pragma clang section text = ".testFunc"  // 适用 ARMCC6
//#pragma arm section code = ".testFunc"  // 适用 ARMCC5
void test_func1(void)
{
    printf("test_func1.\r\n");
}
void test_func2(void)
{
    printf("test_func2 .\r\n");
}
#pragma clang section text = ""   // 适用 ARMCC6
//#pragma arm section code        // 适用 ARMCC5
void test_func3(void)
{
    printf("test_func3 .\r\n");
}
```

编译后结果如下。

```
test_func1                              0x20000001   Thumb Code    16  test.o(.testFunc)
test_func2                              0x20000011   Thumb Code    16  test.o(.testFunc)
test_func3                              0x080004f9   Thumb Code    16  test.o(.text.test_func3)
```

注：此方法不推荐使用，移植性较差。

3.2. 使用分散加载文件链接多文件

通过分散加载文件将具体文件全部重定向到 RAM 中。修改方法如下。

```
LR_IROM1 0x08000000 0x00005000  {    ; load region size_region
  ER_IROM1 0x08000000 0x00005000  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
   .ANY (+XO)
  }
  RW_IRAM1 0x20000000 0x00003000  {  ; RW data
    test.o (+RO +XO)
   .ANY (+RW +ZI)
  }
}
```

编译结果如下

```
test_func1                              0x20000001   Thumb Code    16  test.o(.text.test_func1)
test_func2                              0x20000011   Thumb Code    16  test.o(.text.test_func2)
test_func3                              0x20000021   Thumb Code    16  test.o(.text.test_func3)
```

3.3. 其他方法

早期的 keil 支持使用__RAM 修饰函数将函数放至 ram 中功能，但在 ARMCC5/ARMCC6 时已经淘汰该功能，用户一律通过修改自己的链接文件来完成。为了代码兼容性在 MDK 下可以自定义一个宏将代码放到 SRAM。

```
    #define __ramfunc __attribute__((section(".ramfunc")))
```

免责声明和版权公告

本文档中的信息，包括供参考的 URL 地址，如有变更，恕不另行通知。

本文档可能引用了第三方的信息，所有引用的信息均为“按现状”提供，芯海科技不对信息的准确性、真实性做任何保证。

芯海科技不对本文档的内容做任何保证，包括内容的适销性、是否适用于特定用途，也不提供任何其他芯海科技提案、规格书或样品在他处提到的任何保证。

芯海科技不对本文档是否侵犯第三方权利做任何保证，也不对使用本文档内信息导致的任何侵犯知识产权的行为负责。本文档在此未以禁止反言或其他方式授予任何知识产权许可，不管是明示许可还是暗示许可。

Wi-Fi 联盟成员标志归 Wi-Fi 联盟所有。蓝牙标志是 Bluetooth SIG 的注册商标。

文档中提到的所有商标名称、商标和注册商标均属其各自所有者的财产，特此声明。

芯海科技
CHIPSEA

股票代码:688595