

- 一、前言
- 二、通常 IAR 中 M0/M3/M4 Reloacte 重定向代码到不同的 RAM 空间的方法和直接运用在 IMXRT 上存在的问题
 - 问题 1: RT 的 RAM 类型比较多, 不能直接使用 `_ramfunc` 分配代码到不同 RAM 区域
 - 问题 2: `#pragma location= address+ICF` 文件中定义存放 section 方法, 会对定义中断 ISR 函数以及中断 ISR 中调用的 `#pragma location` 函数无效。
- 三、IAR 中 Reloacte 代码到不同的 RAM 空间的具体方法
 - 方案 1: 程序代码和中断向量表默认运行在 Flash, 手动拷贝 `#pragma location` 定义代码段到 RAM
 - 方案 2: 程序代码和中断向量表默认运行在 RAM 中, 使用 `except` 排除函数或者.o 文件运行在 Flash 中
- 四、总结:

一、前言

IMXRT 是 NXP 推出的一系列性价比很高的跨界处理器, 包括 RT101x/RT102x/RT105x/RT106x 以及 RT117X 系列, 可以让用户以 MCU 的开发方式和价格, 实现接近 MPU 的处理性能。作为一款最大主频 1GHz 的 MCU, IMXRT 与普通 MCU 在开发方式上基本一致, 稍有区别处在于电源设计, Flash 存放以及内部 RAM 的配置, 其中前两个点 NXP 官网有很多文档介绍, 本文档以 IMXRT1020 芯片为例就第三个点展开讨论。IMXRT1020 芯片的 RAM 空间分为四种类型: ITCM, DTCM, OCRAM 以及外部 SDRAM, 其中前面三个属于芯片内部 RAM, 合计 256K, 支持用户静态分配, SDRAM 属于外置 RAM, 最大支持到 1.5G 扩展空间。ITCM 和 DTCM 是直接挂在芯片内核总线, 速度可以达到与内核同频的 528M, OCRAM 挂在 Sys AXI 64 位总线, 速度只能到达 132M, 外部 SDRAM 速度则可达到 266M, 而同时 RT1020 内又有各 16K 的指令 cache 和数据 cache, 用于提高代码在外部 Nor Flash 中 XIP 执行的效率。从速度的角度看, 将所有的用户代码分配在 ITCM/DTCM, 能够发挥到最大性能, 从存储空间大小的角度看, 代码存放在 SDRAM 或者外部 Flash 最简单, 而从 USB/DMA 使用的角度来看, RAM 空间分配在 OCRAM 空间最为方便。所以这些不同 RAM 类型速度/大小的差异和 cache 的存在, 就决定了要想让 RT1020 性能发挥到最大, 就需要用户根据客户实际应用手动修改 RT1020 内部 RAM 空间中 ITCM/DTCM/OCRAM 的大小分配, 并定位关键代码和数据到指定 RAM 空间中运行。

对于 RT 而言, 在当前很多文档都会提到建议对执行速度有要求的代码或者数据分别存放在 ITCM 和 DTCM, 如一些核心的处理算法, 而对一些对速度要求

不高的非关键代码，数据以及一些常量参数存储在 OCRM，外部 SDRAM(如果有外扩 SDRAM 的话)或者内部 Flash 中(RT 支持外部 Nor Flash XIP)。但具体如何实现这个功能，却没有好的例子可供参考。因为这里面牵涉到 IAR 启动过程，ICF 文件的语法，RT 内部 RAM 空间分布以及启动流程，所以比较复杂，尤其是和 IAR 有关的前面两项，非常玄妙。本文结合一个客户的具体应用，描述两种 RT 实际使用过程中如何灵活的 Reloacte 代码到不同的 RAM 空间，以最大程度的发挥 RT 的计算性能。

二、通常 IAR 中 M0/M3/M4 Reloacte 重定向代码到不同的 RAM 空间的方法和直接运用在 IMXRT 上存在的问题

通常 M0/M3/M4 Reloacte 代码到不同的 RAM 空间的方法有两种：一种是使用 `_ramfunc` 关键词，一种是使用 `#pragma location` 关键词，前者的缺点是需要针对每个需要定位到 RAM 的函数分别手动添加该关键词(无法同时将调用的子函数也定位到目标 RAM)，步骤繁琐，不太适合有很多子函数层级调用的函数，并且无法指定代码到绝对地址段(ITCM,DTCM,OCRAM,SDRAM 都一样)，使用起来不太方便。而后者则可以借助 `#pragma default_variable_attributes` 和 `#pragma default_function_attributes` 的方法实现对单个函数，连续多个函数，甚至整个文件.o 的灵活定向分配，能部分解决将调用子函数也一并定位到 RAM 区域，但两种方式实际在 RT 上使用客户依然遇到各种各样的问题。

问题 1: RT 的 RAM 类型比较多，不能直接使用 `_ramfunc` 分配代码到不同 RAM 区域

对于 M0/M3/M4 而言，大部分芯片的存储区域只包含 Flash 和一种类型的 RAM(可能会分有几块，但速度没有差别)。默认代码存储在 Flash 中，如果需要 Reloacte 代码到 RAM 空间，只需要加一个关键词 `_ramfunc`，代码便会编译到 RAM 空间。其实现原理是：`_ramfunc` 定义的函数在编译时会存放到 `.textrw_init` 段(flash 地址)，对应的 RAM 空间存放在 `.textrw` 段(RAM 地址)，IAR 代码启动后由 `_iar_program_start` 函数自动从 `.textrw_init` 段拷贝到 `.textrw` 段，从而实现代码运行在 RAM 区间中。但这种方式并不能完全适用于 RT，因为 M0/M3/M4 只有一种 RAM 类型，而 RT 包括三种 RAM 类型 ITCM/DTCM/OCRAM，甚至还可能是 SDRAM，如果使用 `_ramfunc` 只能指定代码到其中一种 RAM 类型*(具体编译器会编译到哪个 RAM 类型取决于 ICF 链接配置文件的默认设置，SDK 大都默认在 ITCM)。而实际应用中受限于各种 RT

内部各种 RAM 的大小和运行性能要求，则需要灵活存放在 ITCM/DTCM/OCRAM 和 SDRAM，那就要求用户事先理解到这个机制并手动修改链接配置文件，而很多用户对 ICF 的语法都不是很熟悉。

```
PUBWEAK Reset_Handler
SECTION .text:CODE:REORDER:NOROOT(2)
Reset_Handler
CPSID I ; Mask interrupts
LDR R0, =0xE00ED08
LDR R1, =_vector_table
STR R1, [R0]
LDR R2, [R1]
MSR MSP, R2
LDR R0, =SystemInit
BLX R0
CPSIE I ; Unmask interrupts
LDR R0, =_iar_program_start
BX R0
```

大宇笔记不重名

问题 2: #pragma location= address+ICF 文件中定义存放 section 方法，会对定义中断 ISR 函数以及中断 ISR 中调用的 #pragma location 函数无效。

按照 IAR 如下链接的方法，笔者做了在 M4 芯片和 RT 分别作了尝试，发现存在同样的问题，会在使用 #pragma location="xxx section" 定义中断 ISR 函数以及中断 ISR 中调用的 #pragma location= 定义的函数，使用时均无效，对变量的操作不影响。<https://www.iar.com/support/tech-notes/linker/how-do-i-place-a-group-of-functions-or-variables-in-a-specific-section/>

以下两张截图是在中断 ISR 中调用的 #pragma location= 定义的函数的例子，第一张是 ICF 文件的修改，第二张是使用 #pragma location= 声明了一个 delay_ms 函数到 DTCM 的 RAM 区间，并在中断 ISR 函数 SysTick_Handler 中调用该函数。从右下角图可以看到 delay_ms 分配的地址 0x20000001 确实位于 DTCM 区域，但是下载时会出现 verify 错误，而且代码无法正常运行，究其原因是因为尽管设定了 CODE_IN_ITEM 段去 initialize by copy，但实际上并未起作用。而如果不在中断 ISR 函数 SysTick_Handler 中调用 delay_ms，而在 main 函数中调用，delay_ms 程序运行正常。

image-20210911222646069 image-20210911222722010

考虑到客户的实际应用，笔者在此问题上花了很多时间去分析原因，分别在 M4 芯片和 RT 分别作了尝试，发现存在同样的问题。总结下来根本原因在于 IAR startup 启动过程的链接保护机制，见如下截图。IAR 提供了一种 Linker

机制，可以保护 startup 代码被 ICF 文件的‘initialize by copy’所影响，任何 startup 代码所涉及到的函数，包括_low_level_init 函数本身， startup 代码所调用的函数以及与 startup 代码放到同一个.c 文件的其他函数代码，都不会被‘initialize by copy’自动搬移到用户定义的区域，只有在 copy 初始化完成后被调用的代码才会被自动 copy 到 RAM 空间。结合到上面笔者遇到的问题，因为 SysTick_Handler 是在 startup 代码中定义的，这也就意味着 SysTick_Handler 函数本身和其调用的 delay_ms 都不会被自动 copy 到想要 Relocated 的 RAM 区间，需要用户自己去实现 startup 代码所调用的函数 copy 到 RAM 这一点，否则调用时就会出现上面提到的校验错误，导致程序运行出错。

同样的表述和提醒，在 IAR 的开发手册 EWARM_DevelopmentGuide.ENU.pdf 也有描述。

回想一个问题，为何使用问题 1 中使用_ramfunc 定义的 delay_ms 函数同样也被 SysTick_Handler 函数调用过，运行正常的，而问题 2 的处理方法是无效的呢。原因在于，.textrw_init 和.textrw 对于 IAR 来说是 IAR 默认既定的 section 段，_ramfunc 定义的函数会被自动放在.textrw_init(拷贝源地址) 和.textrw(拷贝目标地址) section 区域，然后在 init by copy 时拷贝到 RAM 区域。而#pragmama location= 用户自定义 section 这种方式定义的函数在 flash 中的存储是随机的，IAR 只知道其要拷贝到的 RAM 目的地址，并不知道要从 Flash 的那个地址去拷贝，而且也不会去拷贝。

```
initialize by copy { readwrite, section .textrw, section CODE IN ITCM }; 大宇笔记不重名  
do not initialize { section .noinit };
```

更近一步，既然 IAR 不会去拷贝用户自己定义的 section 段，那是不是对于中断中调用用户自定义 section 段的函数就无解了呢？当然不是。有两种解决问题的思路：

第一种： initialize by copy 既然受限于 Linker 保护，不能手动拷贝，那就手动去干。参见 IAR 的开发手册 EWARM_DevelopmentGuide.ENU 中的如下截图， initialize manually 表示手动去拷贝， MYSECTION 表示要拷贝到的 RAM 区间，然后重点来了， MYSECTION_init 是什么呢？实际上其表示代码在 Flash 的存放区域，也就是在运行时拷贝到 RAM 区间的源地址。DoInit 函数很好理解，是从 Flash 拷贝到 RAM 的过程。其中，_section_begin，

`__section_size` 是 IAR 中预定义的获取 section 起始地址和大小的操作运算符。

Simple copying example with an implicit block

Assume that you have some initialized variables in MYSECTION. If you add this directive to your linker configuration file:

```
initialize manually { section MYSECTION };
```

you can use this source code example to initialize the section:

```
#pragma section = "MYSECTION" RAM中的存储位置  
#pragma section = "MYSECTION_init" Flash中的存储位置
```

```
void DoInit()  
{  
    char * from = __section_begin("MYSECTION_init"); 拷贝源地址, 目的地址以及拷贝函数  
    char * to = __section_begin("MYSECTION");  
    memcpy(to, from, __section_size("MYSECTION"));  
}
```

大宁笔记不重名

第二种：参见以下链接中 IAR TN27158，如下截图是笔者在 RT1020 上做的修改，其思路是在 startup 启动代码中建立第二个中断向量表，第一个向量表仅保留几个 system 中断相关的中断 ISR，负责告知 SP 和 PC，并运行到 `__low_level_init` 函数(最关键的是里面不会包含实际中断服务函数的名称，从而绕过对 ISR 中断服务函数的声明)。而第二个中断向量表则直接定位到 RAM，会包含所有的内核和外设 ISR，这样第二个中断向量表中所调用的任何函数就不会受到前面提到的 Linker 保护的限制。代码会在 initial by copy 阶段自动从 Flash 中拷贝 relocate 到 RAM 的代码。

<https://www.iar.com/support/tech-notes/general/copy-interrupt-vector-arm7-core-fromflash-to-ram-at-startup-v.5.30/> 以下截图中，绿色框体部分是新添加的代码，一方面为存放在 section.invec Flash 区间的函数填满内核相关的 ISR 函数，此处填充的函数名称为 DefaultISR (注意此处的名字很讲究，不要和 `__vector_table_RAM` 中定义的 ISR 的名称相冲突)，另一方面重新申请了一段存放在 section.intvec_RAM 区间的新的中断向量表 `__vector_table_RAM`，可以看到 `__vector_table` 和 `__vector_table_RAM` 区间都包含了 SP 和 PC 指针。除此之外，还需要在 ICF 文件中定义中断向量表 `__vector_table_RAM` 要存放的区域，见如下，这里只是提一下，后续会有更详细介绍。

```

initialize by copy { readwrite, readonly, section .textrw } //YNN
except{
    section .intvec, /* Don't copy interrupt table */
    section .init_array, /* Don't copy C++ init table */
    readonly section .boot_hdr.conf,
    .....
};
do not initialize { section .noinit };

```

place at address mem: m_interrupts_start { readonly section .intvec };
 place at start of ITCM_region {section .intvec_RAM }; //YNN

```

SECTION .intvec:CODE:NOROOT(2)
EXTERN __iar_program_start
EXTERN SystemInit
PUBLIC __vector_table
PUBLIC __vector_table_0x1c
PUBLIC __Vectors
PUBLIC __Vectors_End
PUBLIC __Vectors_Size

DATA
__vector_table
DCD sfe(CSTACK)
DCD Reset_Handler

DCD DefaultISR ; NMI Handler
DCD DefaultISR ; Hard Fault Handler
DCD DefaultISR ; MPU Fault Handler
DCD DefaultISR ; Bus Fault Handler
DCD DefaultISR ; Usage Fault Handler
DCD 0 ; Reserved
DCD 0 ; Reserved
DCD 0 ; Reserved
DCD 0 ; Reserved
DCD DefaultISR ; SVCcall Handler
DCD DefaultISR ; Debug Monitor Handler
DCD 0 ; Reserved
DCD DefaultISR ; PendSV Handler
DCD DefaultISR ; SysTick Handler

SECTION .intvec_RAM:CODE:ROOT(2)
PUBLIC __vector_table_RAM

DATA
__vector_table_RAM
DCD sfe(CSTACK)
DCD Reset_Handler

DCD NMI_Handler ;NMI Handler
DCD HardFault_Handler ;Hard Fault Handler
DCD MemManage_Handler ;MPU Fault Handler
DCD BusFault_Handler ;Bus Fault Handler
DCD UsageFault_Handler ;Usage Fault Handler

vector table 0x1c

```

找出了问题的原因，具体如何操作就相对容易多了，限于篇幅，会在后面文章中继续介绍，敬请期待。

三、IAR 中 Reloacte 代码到不同的 RAM 空间的具体方法

基于上面提出的两种解决办法，尽管都可以去绕过 startup 启动代码的 linker 保护机制，但使用场景不太一样，各有优劣。对于第一种方式：程序代码和中断向量表默认运行在 Flash，只有使用 `#pragma location="xxx section"` 定义的函数会运行在 RAM 中。对于第二种方式：程序代码和中断向量表默认运行在 RAM 中，只有使用 `except` 排除在外的函数或者.o 文件会运行在 Flash 中。两者相比，前者优点在于节省 RAM，效率略差，后者由于代码大都是在 RAM 中，其优点在于效率很高，缺点就是 RAM 占用大。以上找到问题点，有了思路，那还是回到文章开始部分，具体怎么去干，去修改？依然是花开两朵，各表一枝。两者都以 RT1020 SDK 2.5.0 的 `iled_blinky` 代码的 `flexspi_nor_debug` 配置为例。看到这里读者可能会觉得太复杂，其实我想说，的确十分复杂，复杂在去找到这个问题点，找到症结之后，修改就简单了。

方案 1：程序代码和中断向量表默认运行在 Flash，手动拷贝

`#pragma location` 定义代码段到 RAM

原因上面已经讲清楚了，下面直接记录操作的步骤。

步骤 1： 修改链接配置文件 `MIMXRT1021xxxxx_flexspi_nor.icf`，定义手动拷贝区域的源地址和目的地址，

```

initialize by copy { readwrite, section DATA_IN_DTCM}; //自动初始化变量
initialize manually { , section .textw }; //强制手动初始化_ramfunc 定义的函数
initialize manually { section CODE_IN_ITCM }; //强制手动初始化定义到 ITCM 的函数
initialize manually { section CODE_IN_SDRAM }; //强制手动初始化定义到 SDRAM 的函数
do not initialize { section .noinit };

```

```

define block CodeRelocate { section .textw_init}; //定义_ramfunc 定义的函数在 Flash 中的存放地址
define block CodeRelocateRam { section .textw }; //定义_ramfunc 定义的函数要拷贝到的 RAM 地址
define block CodeRelocate1 { section CODE_IN_ITCM_init }; //定义 section CODE_IN_ITCM 在 Flash 的地址
define block CodeRelocateRam1 { section CODE_IN_ITCM }; //定义 section CODE_IN_ITCM 在 RAM 的地址
define block CodeRelocate2 { section CODE_IN_SDRAM_init }; //定义 section CODE_IN_SDRAM 在 Flash 地址
define block CodeRelocateRam2 { section CODE_IN_SDRAM }; //定义 section CODE_IN_SDRAM 在 RAM 地址

```

```

place at address mem: m_interrupts_start { readonly section .intvec };
place at address mem:m_boot_hdr_conf_start { section .boot_hdr.conf };
place at address mem:m_boot_hdr_ivt_start { section .boot_hdr.ivt };
place at address mem:m_boot_hdr_boot_data_start { readonly section .boot_hdr.boot_data };
place at address mem:m_boot_hdr_dcd_data_start { readonly section .boot_hdr.dcd_data };

```

```

keep{ section .boot_hdr.conf, section .boot_hdr.ivt, section .boot_hdr.boot_data, section .boot_hdr.dcd_data };

```

大宁笔记不重名

```

place in TEXT_region { readonly };
place in DTCM_region { block RW };
place in DTCM_region { block ZI };
place in DTCM_region { last block HEAP };
place in NCACHE_region { block NCACHE_VAR };
place in CSTACK_region { block CSTACK };

```

```

place in DTCM_region { section DATA_IN_DTCM };
place in ITCM_region { block CodeRelocateRam }; //指定各个 section 实际存放地址
place in TEXT_region { block CodeRelocate };
place in ITCM_region { block CodeRelocateRam1 };
place in TEXT_region { block CodeRelocate1 };
place in SDRAM_region { block CodeRelocateRam2 };
place in TEXT_region { block CodeRelocate2 };

```

大宁笔记不重名

步骤 2: 找到 system_MIMXRT1021.c, 在文件起始部位添加变量声明, 为下一步拷贝过程做准备。

```

#pragma section = ".data"
#pragma section = ".data_init"
#pragma section = ".bss"
#pragma section = "CodeRelocate"
#pragma section = "CodeRelocateRam"
#pragma section = "CodeRelocate1"
#pragma section = "CodeRelocateRam1"
#pragma section = "CodeRelocate2"
#pragma section = "CodeRelocateRam2"
extern uint32_t _VECTOR_TABLE[];
extern uint32_t _VECTOR_RAM[];
extern uint32_t _RAM_VECTOR_TABLE_SIZE;

```

大宁笔记不重名

步骤 3: 在 system_MIMXRT1021.c 的 SystemInit 函数结尾处添加手动拷贝的代码, 以下拷贝过程和步骤 1 中 ICF 文件配置是一一对应的, 其实这个步骤和_iar_program_start 拷贝数据的原理一样。

```

#if 0
    if (__VECTOR_RAM != __VECTOR_TABLE) /* Copy the vector table to RAM */

        {
            for (n = 0; n < 0x3ff; n++){
                __VECTOR_RAM[n] = __VECTOR_TABLE[n];
                if(n>0x300)
                {
                    m=2;
                }
            }
        }
        /* Point the VTOR to the new copy of the vector table */
        SCB->VTOR = (uint32_t)__VECTOR_RAM; //重定向中断向量表到 RAM 中
#endif

#if 1
    uint8_t* code_relocate_ram = __section_begin("CodeRelocateRam"); //拷贝 section textrw 段到 RAM
    uint8_t* code_relocate = __section_begin("CodeRelocate"); //参见上一步骤 ICF 的配置
    uint8_t* code_relocate_end = __section_end("CodeRelocate");

    n = code_relocate_end - code_relocate; /* Copy functions from ROM to RAM */
    while (n--)
        *code_relocate_ram++ = *code_relocate++;
#endif

#if 1
    uint8_t* code_relocate_ram1 = __section_begin("CodeRelocateRam1"); //拷贝 section CODE_IN_ITCM 段到
    RAM 地址
    uint8_t* code_relocate1 = __section_begin("CodeRelocate1"); //参见上一步骤 ICF 的配置
    uint8_t* code_relocate_end1 = __section_end("CodeRelocate1");

    n = code_relocate_end1 - code_relocate1; /* Copy functions from ROM to RAM */
    while (n--)
        *code_relocate_ram1++ = *code_relocate1++;
#endif

#if 1
    uint8_t* code_relocate_ram1 = __section_begin("CodeRelocateRam1"); //拷贝 section CODE_IN_ITCM 段到
    RAM 地址
    uint8_t* code_relocate1 = __section_begin("CodeRelocate1"); //参见上一步骤 ICF 的配置
    uint8_t* code_relocate_end1 = __section_end("CodeRelocate1");

    n = code_relocate_end1 - code_relocate1; /* Copy functions from ROM to RAM */
    while (n--)
        *code_relocate_ram1++ = *code_relocate1++;
#endif

#if 1
    uint8_t* code_relocate_ram2 = __section_begin("CodeRelocateRam2"); //拷贝 section CODE_IN_SDRAM
    段到 RAM 地址
    uint8_t* code_relocate2 = __section_begin("CodeRelocate2");
    uint8_t* code_relocate_end2 = __section_end("CodeRelocate2");

    n = code_relocate_end2 - code_relocate2; /* Copy functions from ROM to RAM */
    while (n--)
        *code_relocate_ram2++ = *code_relocate2++;
#endif
#endif
SystemInitHook();

```

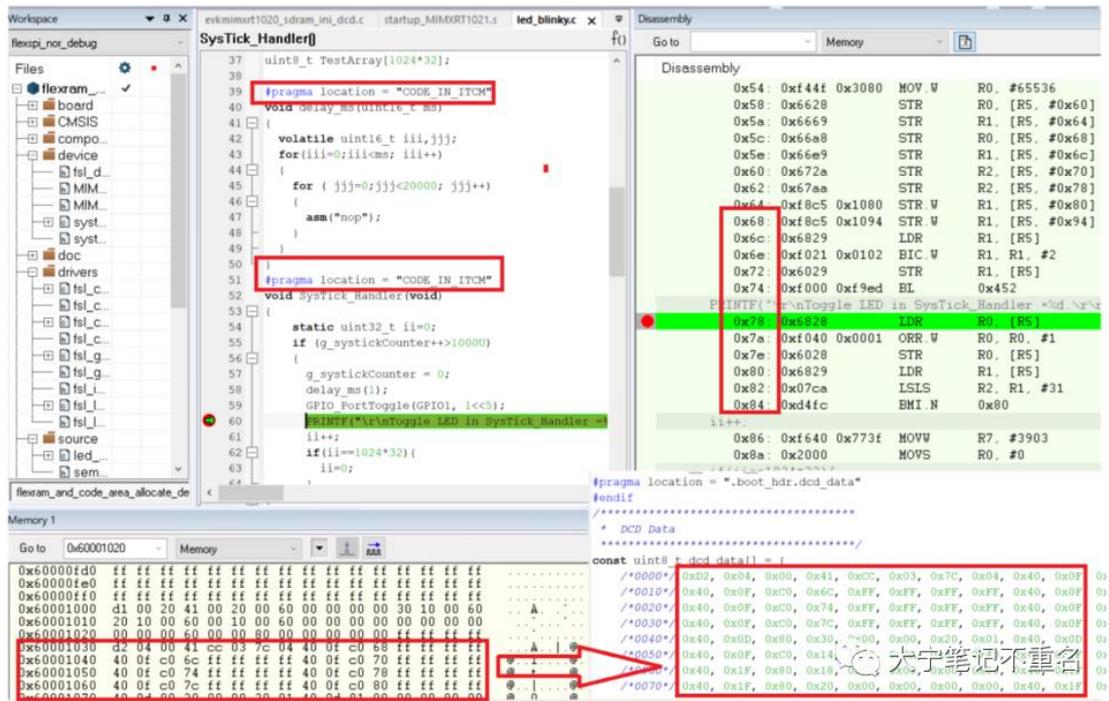
步骤 4: 为打算 relocate 到指定 RAM 区域的函数添加 #pragma location = 操作。以下是一个实例。

本实例中，都是使用 `#pragma location` 对单个函数进行 `relocate`，其实 `#pragma location` 支持对多个连续函数进行 `relocate`，可以参考以下链接，此处不再赘述。<https://www.iar.com/support/tech-notes/linker/how-do-i-place-a-group-of-functions-or-variables-in-a-specific-section/> **步骤 5**：查看生成的 `map` 文件和代码运行结果。

从 `map` 文件可以看出，大部分代码都是运行在 `0x6000xxxx` 的 Flash 中，在步骤 4 中通过 `#pragma location = "CODE_IN_ITCM"` 定义的 `SysTick_Handler` 和 `delay_ms` 都位于 ITCM 中，而通过 `#pragma location = "DATA_IN_DTCM"` 定义的数组 `TestArray` 位于 DTCM 区间，同时也能看到 `main` 函数是存放在 Flash 区间的，完全符合预期。

Entry	Address	Size	Type	Object
Serial_UartInit	0x6000'2ea9	0x68	Code Gb	serial_port_uart.o [1]
Serial_UartWrite	0x6000'2f11	0x52	Code Gb	serial_port_uart.o [1]
StrFormatPrintf	0x6000'2c8d	0x21c	Code Gb	fsl_str.o [1]
SysTick_Config	0x6000'3f89	0x34	Code Lc	led_blinky.o [1]
SysTick_Handler	0x53	0x46	Code Gb	led_blinky.o [1]
SysTick_Handler::ii	0x2000'0074	0x4	Data Lc	led_blinky.o [1]
SystemCoreClock	0x2000'0000	0x4	Data Gb	system_MIMXRT1021.o [1]
SystemCoreClockUpdate	0x6000'38fb	0xe4	Code Gb	system_MIMXRT1021.o [1]
SystemInit	0x6000'3811	0xea	Code Gb	system_MIMXRT1021.o [1]
TestArray	0x2000'007c	0x8000	Data Gb	led_blinky.o [1]
.....				
delay_ms	0x11	0x42	Code Gb	led_blinky.o [1]
.....				
main	0x6000'3fc3	0x5a	Code Gb	led_blinky.o [1]
n	0x2000'0078	0x4	Data Gb	system_MIMXRT1021.o [1]
qspi_flash_config	0x6000'0000	0x200	Data Gb	evkmimxrt1020_flexspi_nor_config.o [1]

编译代码，运行到中断服务函数 `SysTick_Handler`，从汇编可以看到代码运行在 ITCM 区域，左下角 `0x60001030` 是 DCD 数据的存放信息，对照代码中 DCD 的数组，可以看到完全一致。重新上电代码运行正常。



方案 2: 程序代码和中断向量表默认运行在 RAM 中, 使用 **except** 排除函数或者.o 文件运行在 Flash 中

步骤 1: 找到 `startup_MIMXRT1021.s`, 添加如下红色部分代码, 建立第二个中断向量表;

```

SECTION intvec:CODE:NOROOT(2)
EXTERN __iar_program_start
EXTERN SystemInit
.....

DATA
__vector_table
DCD sfe(CSTACK)
DCD Reset_Handler

/*-----YNN-----*/
DCD DefaultISR ; NMI Handler
DCD DefaultISR ; Hard Fault Handler
DCD DefaultISR ; MPU Fault Handler
DCD DefaultISR ; Bus Fault Handler
DCD DefaultISR ; Usage Fault Handler
DCD 0 ; Reserved
DCD 0 ; Reserved
DCD 0 ; Reserved
DCD 0 ; Reserved
DCD DefaultISR ; SVCall Handler
DCD DefaultISR ; Debug Monitor Handler
DCD 0 ; Reserved
DCD DefaultISR ; PendSV Handler
DCD DefaultISR ; SysTick Handler

SECTION intvec_RAM:CODE:ROOT(2)
PUBLIC __vector_table_RAM
DATA
__vector_table_RAM
DCD sfe(CSTACK)
DCD Reset_Handler

/*-----YNN-----*/
DCD NMI_Handler ;NMI Handler
DCD HardFault_Handler ;Hard Fault Handler

```

 大宁笔记不重名

步骤 2: 修改链接配置文件 MIMXRT1021xxxxx_flexspi_nor.icf, 将步骤 1 中声明的第二个中断向量表放在 ITCM 区域, 并使所有代码都放在 RAM, 然后再排除掉不希望运行在 RAM 中的文件。

```

define block CSTACK with alignment = 8, size = __size_cstack__ { };
define block HEAP with alignment = 8, size = __size_heap__ { };
define block RW { readwrite };
define block ZI { zi };
define block NCACHE_VAR { section NonCacheable, section NonCacheable.init };

initialize by copy { readwrite, readonly, section .textw } //YNN 添加 readonly, 使所有代码都放在 RAM
except{ //排除不希望从RAM运行的代码
和数据
    section .intvec, /* Don't copy interrupt table */
    section .init_array, /* Don't copy C++ init table */
    readonly section .boot_hdr.conf, //防止 XIP 启动配置也被放在 RAM 区域
    readonly section .boot_hdr.ivt,
    readonly section .boot_hdr.boot_data,
    readonly section .boot_hdr.dcd_data,
    readonly object fsl_gpio.o, //避免把 fsl_gpio.c 的所有函数放在 RAM 中运行
    //readonly object fsl_lpuart.o, //将 fsl_lpuart.c 的所有函数放在 RAM 中运行
};
do not initialize { section .noinit };

place at address mem: m_interrupts_start { readonly section .intvec }; //第一个中断向量表放在 Flash 区域
place at start of ITCM_region { section .intvec_RAM }; //YNN 将第二个中断向量表放在 ITCM 区域

place at address mem:m_boot_hdr_conf_start {section .boot_hdr.conf};
place at address mem:m_boot_hdr_ivt_start { section .boot_hdr.ivt };
place at address mem:m_boot_hdr_boot_data_start { readonly section .boot_hdr.boot_data };
place at address mem:m_boot_hdr_dcd_data_start { readonly section .boot_hdr.dcd_data };

keep{ section .boot_hdr.conf, section .boot_hdr.ivt, section .boot_hdr.boot_data, section .boot_hdr.dcd_data };

place in TEXT_region { readonly };
place in ITCM_region { block RW };
place in DTCM_region { block ZI };
place in DTCM_region { last block HEAP };
place in DTCM_region { block NCACHE_VAR };
place in CSTACK_region { block CSTACK };

```

大宁笔记不重名

注 1: 具体 ICF 的语法, 读者可以从 IAR 相关文档查看。说实话, 真是博大精深, 笔者曾 N 多次掉入坑中, 就一个 `readonly object fsl_gpio.o`, `readonly code object fsl_gpio.o`, `object fsl_gpio.o` 实际产生的区别, 就花了好长时间去验证和理解, 请参见笔者另外一个文档, IAR 连接配置文件 ICF 中 `readonly code object fsl_lpuart.o`和 `readonly object fsl_lpuart.o` 区别。

注 2: 以下关于 RT1020 XIP 部分的配置一定要配置 `except`, 否则会出现代码重新上电后无法启动的情况。

```

Except{
    readonly section .boot_hdr.conf, //防止 XIP 启动配置也被放在 RAM 区域
    readonly section .boot_hdr.ivt,
    readonly section .boot_hdr.boot_data,
    readonly section .boot_hdr.dcd_data,
}

```

大宁笔记不重名

步骤 3: 在 `main` 函数第一句修改 `SCB->VTOR` 的值, 以匹配新的中断向量表的地址

```

int mm;
int main(void)
{
    SCB->VTOR = 0x00001000; //YNN
    /* Define the init structure for the output LED pin*/
    gpio_pin_config_t led_config = {kGPIO_DigitalOutput, 0, kGPIO_NoIntmode};

    /* Board pin init */
    BOARD_InitPins();
    BOARD_BootClockRUN();
    BOARD_InitDebugConsole();
}

```

大宁笔记不重名

具体赋予 SCB->VTOR 的值取决于步骤 2 中的定义，对于笔者的代码而言，新的中断向量表是放在 ITCM 的起始地址的，也就是 0x00001000。

```

define symbol m_data1_start      = 0x00001000; //YNN ITCM 64K
define symbol m_data1_end        = 0x0000FFFF;
define region ITCM_region = mem:[from m_data1_start to m_data1_end];
place at start of ITCM_region {section .intvec_RAM};

```

大宁笔记不重名

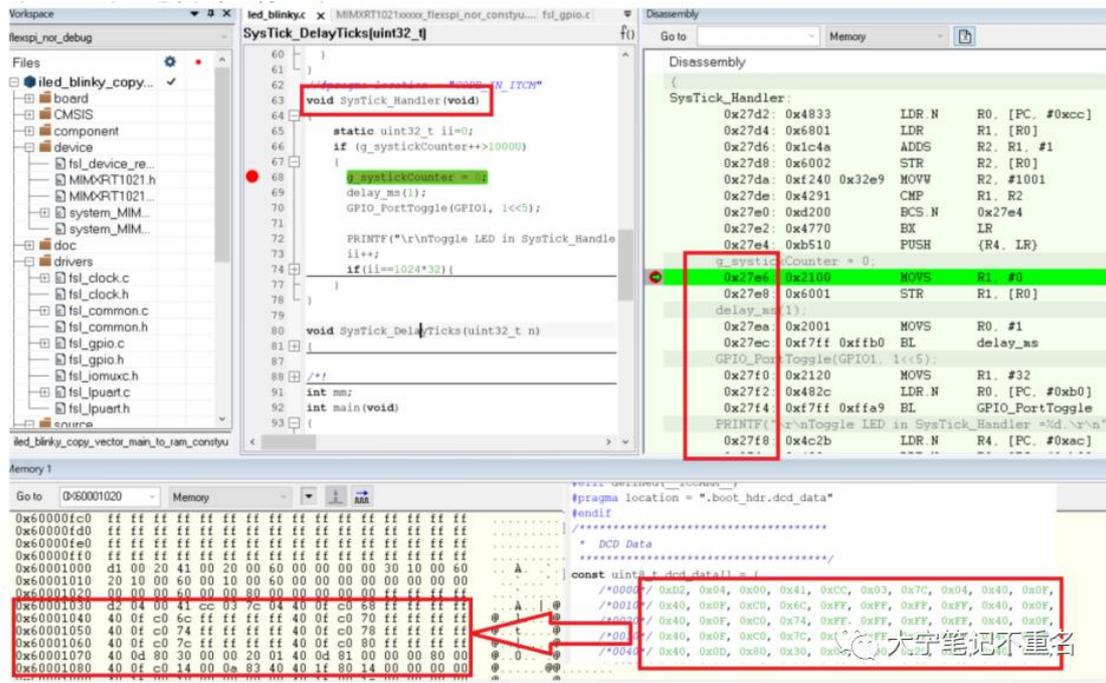
步骤 4：查看生成的 map 文件和代码运行结果。

从下图可以看到，用户函数，main 代码和 _vector_table_RAM 都定位在 RAM 中了，只有 except 的 fsl_gpio.c 文件包含的函数存放在 RAM 中，符合预期。

Entry	Address	Size	Type	Object
DbgConsole_Printf	0x22bb	0x2c	Code	Gb fsl_debug_console.o [1]
DbgConsole_SendData	0x2199	0x44	Code	Gb fsl_debug_console.o [1]
GPIO_GetInstance	0x6000'2905	0x2c	Code	Lc fsl_gpio.o [1]
GPIO_PinInit	0x6000'2931	0x52	Code	Gb fsl_gpio.o [1]
GPIO_PinSetInterruptConfig	0x6000'29dd	0x80	Code	Gb fsl_gpio.o [1]
GPIO_PinWrite	0x6000'2983	0x3a	Code	Gb fsl_gpio.o [1]
GPIO_PortToggle	0x274b	0x6	Code	Lc led_blinky.o [1]
GPIO_SetPinInterruptConfig	0x6000'2903	0x2	Code	Lc fsl_gpio.o [1]
HAL_UartGetStatus	0x28c1	0xe	Code	Lc lpuart_adapter.o [1]
HAL_UartInit	0x28cf	0xd6	Code	Gb lpuart_adapter.o [1]
.....				
__low_level_init	0x6000'2c3b	0x4	Code	Gb low_level_init.o [2]
__vector_table	0x6000'2000		Data	Gb startup_MIMXRT1021.o [1]
__vector_table_0x1c	0x101c		Data	Gb startup_MIMXRT1021.o [1]
__vector_table_RAM	0x1000		Data	Gb startup_MIMXRT1021.o [1]
__call_main	0x6000'2c29		Code	Gb cmain.o [4]
.....				
main	0x2819	0x6c	Code	Gb led_blinky.o [1]
mm	0x2000'8058	0x4	Data	Gb led_blinky.o [1]
qspi_flash_config	0x6000'0000	0x200	Data	Gb evkmimxrt1020_flexspi_nor_config.o [1]

大宁笔记不重名

编译代码，运行到中断服务函数 SysTick_Handler，从汇编可以看到代码运行在 ITCM 区域，左下角 0x60001030 是 DCD 数据的存放信息，对照代码中 DCD 的数组，可以看到完全一致。重新上电代码运行正常。



四、总结：

从以上步骤描述，两种方法都能实现 `relocate` 代码到指定 RAM 区域，但从实现步骤上来看，第二种方法操作更简单一些，不需要对每个函数分别进行处理，尤其是在很多层级代码调用的场合，缺点是对 RAM 占用会变大。而第一种方法，对认识代码搬运的实质更清晰一些，最大程度的节省 RAM 空间，考虑到 RT1020 有 256K RAM 空间，应该足以应付大部分应用，所以建议用户使用第二种办法。