如何开发与存储位置无关的 STM32 应用

关键字：PIC， PIE，位置

1. 前言

最近有客户询问，能否使用 STM32CubeIDE 在编译时通过设置某个编译选项，让 STM32 应用与存储位置无关。这样的优势是能使同一个固件被烧在 STM32 Flash 里的不同位置， 而在系统 Bootloader 里只需要跳到相应的位置就可以正常执行固件代码。客户希望 STM32 代码从 Flash 里执行，不复制到 RAM 里；客户希望是一个完整的映像，而不仅仅是其中某个函数做到了位置无关。

2. 分析

在嵌入式场景下，不一定有操作系统。即使有操作系统，一般也是 RTOS。一般 RTOS 没有一个通用的程序加载器。因此，存储位置无关的需求，在这时可以说无关紧要。但是，如果客户需要进行在线固件更新，例如 IoT 应用的固件升级，那么位置无关就存在价值了。位置无关之后，对于不同的软件版本，不需要频繁的为烧写位置的反复改变而修改编译链接脚本。也不需要在代码里显式的在两个 Bank 之间进行切换。

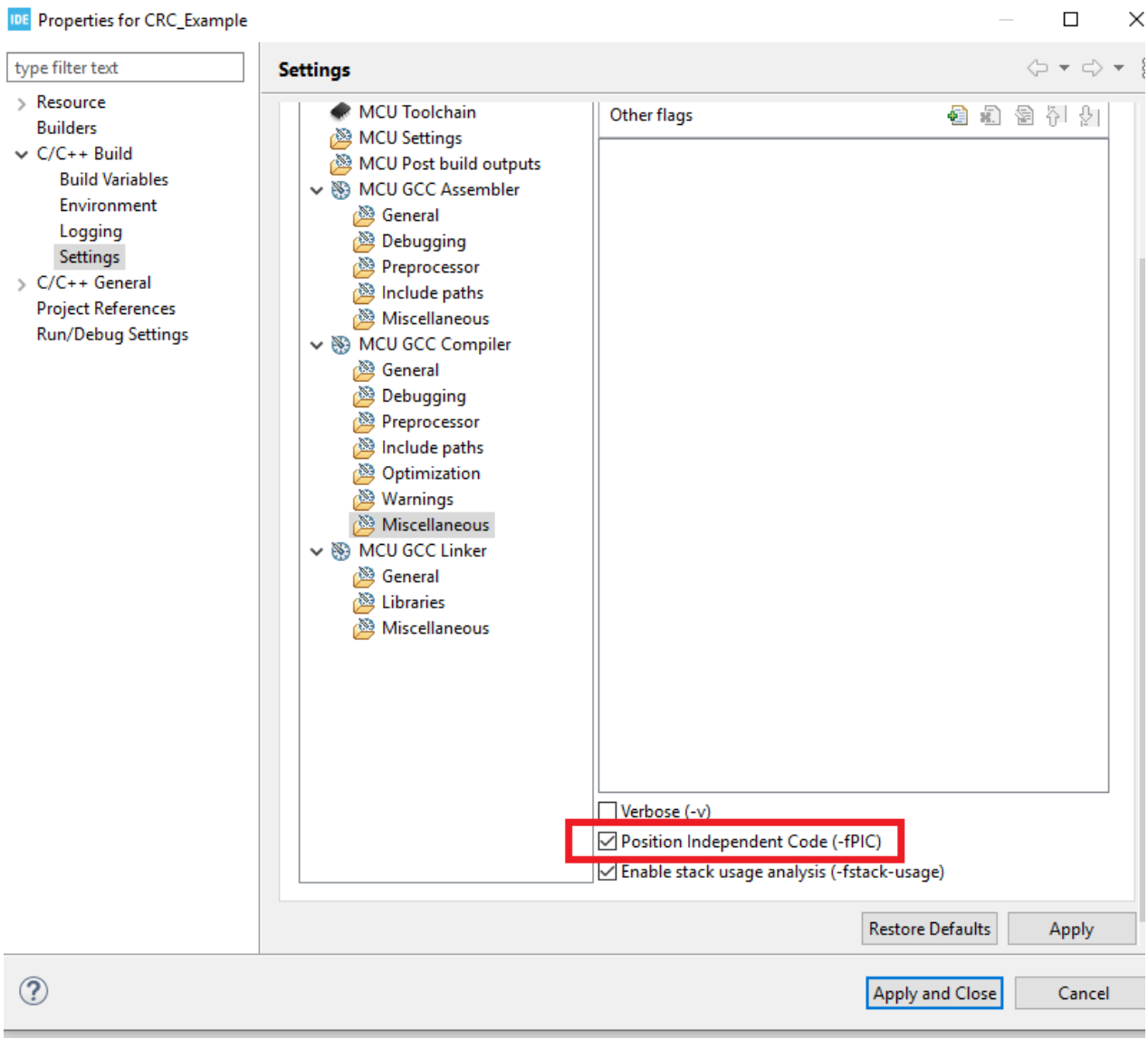
最简单的情况是所有的代码都复制到内存执行。因为 Flash 的功能只是进行存储，自然对 Flash 的位置没有任何要求。但大部分 MCU 用户面临的真实案例都是 Flash 比较大，例如 ，1M 字节 ；RAM 比较小，例如，128K 字节。在这种情况下，代码在 Flash 原地执行就是一个必须的选择。

Flash 位置改变，会影响从 Bootloader 跳转之后的固件执行时的 PC 指针，也就是 PC 指针值会发生相应的变化。位置无关的原理，是让应用程序经过编译后所生成的映像，其中的代码和数据，都是基于相对代码的位置进行引用。那么，当应用被搬到不同位置时，他们的相对位置不变，从而执行不受影响。

代码和数据基于绝对地址还是基于相对地址，是由编译器所决定。以客户要求的 STM32CubeIDE 编译工具为例，我们可以看到在[Project]->[Properties]->[C/C++ Build]->[Settings]->[Tool Settings]->[MCU GCC Compiler]->[Miscellaneous]已经有一项[Position Independent Code (-fPIC)]。

是否只要选一下-fPIC 选项就大功告成了呢？答案是没有那么简单。

图 1 选择编译选项-fPIC



事实上，对于完整应用程序工程，用户应该经过这些步骤将其变成位置无关：

- 选择正确的编译器选项
- 去掉或者替换掉那些包含绝对位置的库文件
- 修改代码中的 Flash 绝对地址（这里以 STM32H7 的 CRC_Example 例程为例，其他情况下有可能要修改更多）
  - 在 startup_xxx.s 汇编代码里的 sidata
  - 在 system_xxx.c 里的 SCB->VTOR 以及中断向量表内容
  - GOT

对于完整工程，要正确的跳转到应用程序进行执行，还需要由 Bootloader 向应用程序提供或者由应用程序在链接时自身解析计算，得到以下信息：

- Flash 偏移量
- 中断向量表的开始以及结束地址
- GOT 的开始以及结束地址

我们接下来就举例说明这些步骤。

3. 步骤

3.1. 选择正确的编译器选项

如果我们不使用任何编译选项，编出来的代码会怎么样？我们可以通过.list 文件进行查看。.list 文件在 STM32 例程中默认生成，如果没有请勾选如下选项， 在 [Project]->[Properties]->[C/C++ Build]->[Settings]->[Tool Settings]->[MCU Post Build outputs]->[Generate list file]，可参考下图。

图 2 生成.list 文件

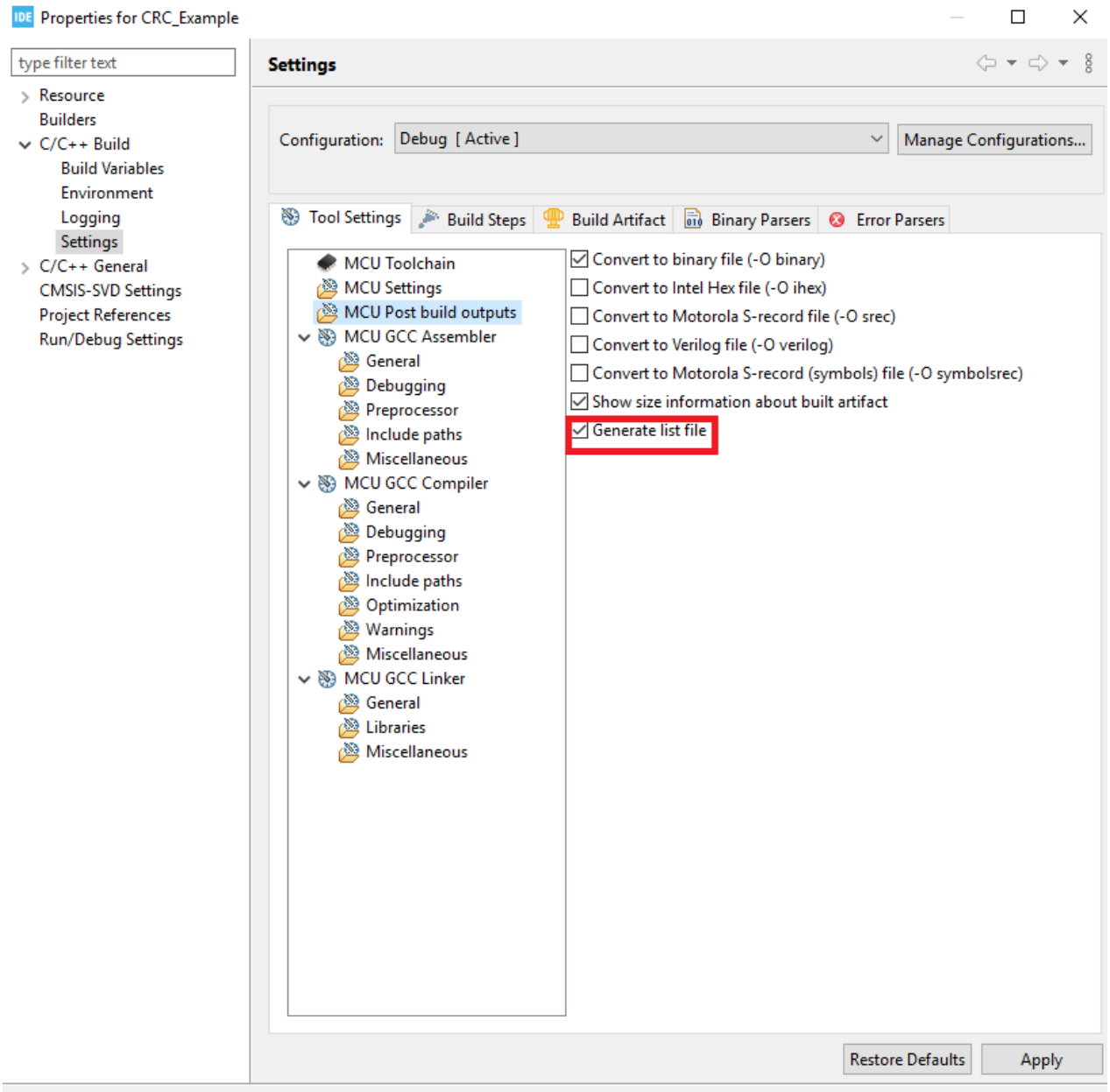
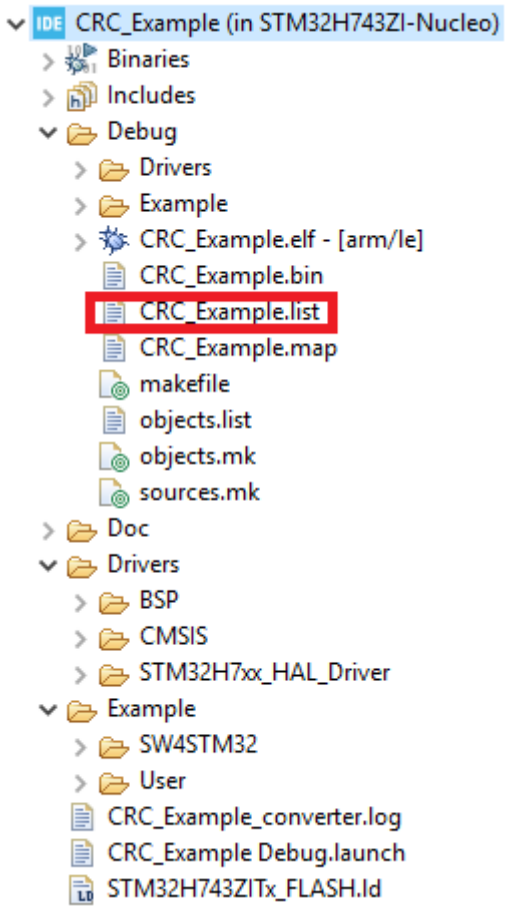


图 3　找到.list 文件



对于我们熟悉的函数 HAL_IncTick，

```
__weak void HAL_IncTick(void)
{
  uwTick += (uint32_t)uwTickFreq;
}
```

我们可以从.list 中找到如下反汇编代码。

```
__weak void HAL_IncTick(void)
{
 80006fc:   b480        push    {r7}
 80006fe:   af00        add r7, sp, #0
  uwTick += (uint32_t)uwTickFreq;
 8000700:   4b06        ldr r3, [pc, #24]   ; (800071c <HAL_IncTick+0x20>)
 8000702:   781b        ldrb    r3, [r3, #0]
 8000704:   461a        mov r2, r3
 8000706:   4b06        ldr r3, [pc, #24]   ; (8000720 <HAL_IncTick+0x24>)
 8000708:   681b        ldr r3, [r3, #0]
 800070a:   4413        add r3, r2
 800070c:   4a04        ldr r2, [pc, #16]   ; (8000720 <HAL_IncTick+0x24>)
 800070e:   6013        str r3, [r2, #0]
}
 8000710:   bf00        nop
 8000712:   46bd        mov sp, r7
 8000714:   f85d 7b04   ldr.w   r7, [sp], #4
 8000718:   4770        bx  lr
 800071a:   bf00        nop
 800071c:   20000028    .word   0x20000028
 8000720:   20000034    .word   0x20000034
```

我们看到代码中直接使用了变量的绝对地址，例如 0x2000 0028。我们不要被 literal pool 文字池的使用所迷惑，那个基于 PC 的操作只是为了取变量的绝对地址，例如，0x2000 0028，并没有将绝对地址变成相对地址。

当然大家说这里是 RAM 地址，没有关系。我们选择这个函数来说明，是因为位置无关的编译器选项是不区分 RAM 还是 Flash 里的变量，而这个函数最简单容易理解。如果我们查看另外一个复杂一点的函数，例如，HAL_RCC_ClockConfig，我们可以看到以下对 Flash 里变量的直接使用。这就不妙了，因为一旦改变了 Flash 下载的位置，在绝对地址处就取不出变量的真实内容了。

```
8001cd6:   4618        mov r0, r3
8001cd8:   3718        adds    r7, #24
8001cda:   46bd        mov sp, r7
8001cdc:   bd80        pop {r7, pc}
8001cde:   bf00        nop
8001ce0:   58024400    .word   0x58024400
8001ce4:   080023c4    .word   0x080023c4
8001ce8:   20000020    .word   0x20000020
8001cec:   2000001c    .word   0x2000001c
8001cf0:   20000024    .word   0x20000024
```

我们没有办法一个一个查找修改所有的变量。当然这里的变量是指全局变量。如果要修改，我们希望编译器能把他们集中在一起。对于此，编译器提供了多个编译选项。例如，PIC 是位置无关代码， PIE 是位置无关执行。PIC 和 PIE 这两者类似，但是存在一个显著的差异是 PIE 会对部分全局变量优化。我们可以观察到用两种不同编译选项的效果。

如果使用-fPIC

```
__weak void HAL_IncTick(void)
{
 80004a8:   4a05        ldr r2, [pc, #20]   ; (80004c0 <HAL_IncTick+0x18>)
  uwTick += (uint32_t)uwTickFreq;
 80004aa:   4b06        ldr r3, [pc, #24]   ; (80004c4 <HAL_IncTick+0x1c>)
{
 80004ac:   447a        add r2, pc
  uwTick += (uint32_t)uwTickFreq;
 80004ae:   58d1        ldr r1, [r2, r3]
 80004b0:   4b05        ldr r3, [pc, #20]   ; (80004c8 <HAL_IncTick+0x20>)
 80004b2:   6808        ldr r0, [r1, #0]
 80004b4:   58d3        ldr r3, [r2, r3]
 80004b6:   781b        ldrb    r3, [r3, #0]
 80004b8:   4403        add r3, r0
 80004ba:   600b        str r3, [r1, #0]
}
 80004bc:   4770        bx  lr
 80004be:   bf00        nop
 80004c0:   17fffb64    .word   0x17fffb64
 80004c4:   00000008    .word   0x00000008
 80004c8:   00000004    .word   0x00000004
```

其中 80004C0 地址处包含的是 GOT 自身的偏移量，存在 r2 里，要在两次取全局变量 uwTickFreq 和 uwTick 时引用。GCC 编译器引入 GOT 全局偏移量表来解决全局变量的绝对地址的问题。在之前对绝对地址的直接使用，现在被转化成先取得 GOT 入口相对于 PC 的偏移，再获得实际变量相对于 GOT 入口的偏移，从而得到实际变量的地址。计算公式如下：

实际变量的绝对地址=PC + GOT 相对于 PC 的偏移 + 变量地址相对于 GOT 的偏移

GOT 只有一个，如果代码放在不同的位置，代码自身就可以根据 Bootloader 传递过来的信息，或者自行计算来对 GOT 进行更新。这样变量的地址就和新的 Flash 偏移相匹配。

如果使用-fPIE

```
__weak void HAL_IncTick(void)
{
 80004a0:   4b05        ldr r3, [pc, #20]   ; (80004b8 <HAL_IncTick+0x18>)
  uwTick += (uint32_t)uwTickFreq;
 80004a2:   4a06        ldr r2, [pc, #24]   ; (80004bc <HAL_IncTick+0x1c>)
{
 80004a4:   447b        add r3, pc
  uwTick += (uint32_t)uwTickFreq;
 80004a6:   589a        ldr r2, [r3, r2]
 80004a8:   4b05        ldr r3, [pc, #20]   ; (80004c0 <HAL_IncTick+0x20>)
 80004aa:   6811        ldr r1, [r2, #0]
 80004ac:   447b        add r3, pc
 80004ae:   781b        ldrb    r3, [r3, #0]
 80004b0:   440b        add r3, r1
 80004b2:   6013        str r3, [r2, #0]
}
 80004b4:   4770        bx  lr
 80004b6:   bf00        nop
 80004b8:   17fffb6c    .word   0x17fffb6c
 80004bc:   00000004    .word   0x00000004
 80004c0:   17fffb58    .word   0x17fffb58
```

这里可以看到 80004c0 对应的 uwTick（可以从 str 指令结合 C 语言源代码快速知道它对应于 uwTick）不再使用 GOT 偏移，而是相对于 PC 的偏移（与前文相比，多了一条指令 “add r3,pc”）。换句话说，PIE 对局部的全局变量做了优化。这个优化显然不是我们所需要的。因为如此以来，RAM 变量的地址就会随着 PC 的不同而不同。而我们则希望所有对 RAM 的用法不发生变化。

为了能够修改 GOT 内容，我们选择将 GOT 最终存放在 RAM 中，导致代码中对 GOT 的寻址也是使用了相对于 PC 的偏移。而因为 RAM 有限，或者因为没有虚拟内存的缘故，我们不希望 RAM 的用法有所不同，否则，可能代价很大。这时，一旦 Flash 代码位置发生变化引起 PC 指针变化，GOT 就无法找到。因此，即使我们不使用 PIE，PIC 也没有办法单独使用。为了确保没有任何存放在 RAM 里的变量的位置是相对于 PC 的偏移。我们应该使用如下所有编译选项，single-pic-base 让系统只使用一个 PIC 基址，就是下文反汇编中看到 r9；no-pic-data-is-text-relative 则让编译器不要让任何变量相对于 PC 寻址。

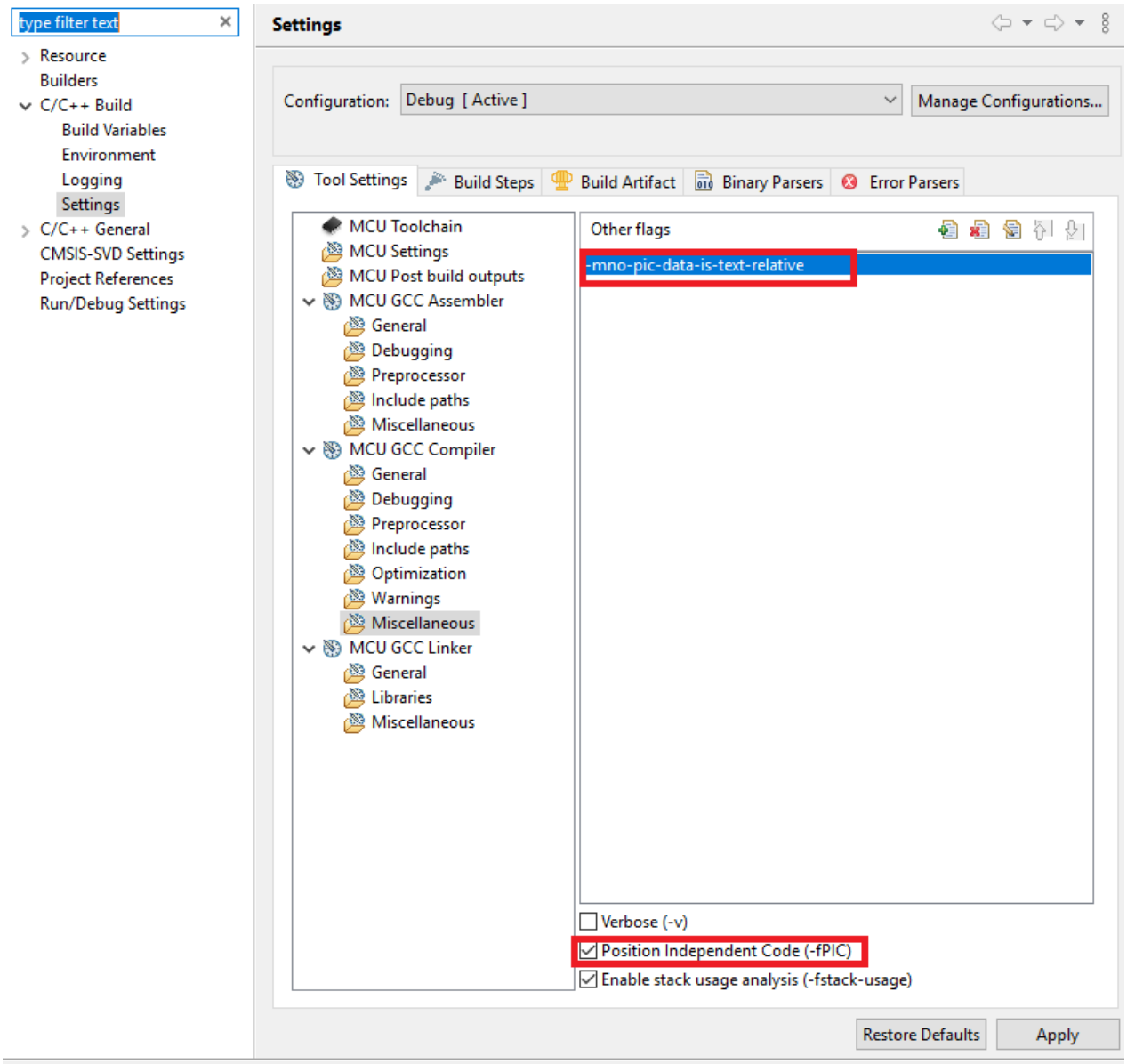
- -fPIC
- -msingle-pic-base
- -mno-pic-data-is-text-relative

在使用 no-pic-data-is-text-relative 时，经测试，有 single-pic-base 的效果；反之，则不然。所以这里可以去掉 single-pic-base，改成

- **-fPIC**
- **-mno-pic-data-is-text-relative**

如下图所示：

图 4 选择正确的编译选项



这样实际变量的绝对地址，就变成

实际变量的绝对地址=PIC 基址 + GOT 相对于 PIC 基址的偏移 + 变量地址相对于 GOT 的偏移

使用以上编译选项，这样我们看到 HAL_IncTick 就如下所示：

```
__weak void HAL_IncTick(void)
{
 8000720:   b480        push    {r7}
 8000722:   af00        add r7, sp, #0
  uwTick += (uint32_t)uwTickFreq;
 8000724:   4b09        ldr r3, [pc, #36]   ; (800074c <HAL_IncTick+0x2c>)
 8000726:   f859 3003   ldr.w   r3, [r9, r3]
 800072a:   781b        ldrb    r3, [r3, #0]
 800072c:   461a        mov r2, r3
 800072e:   4b08        ldr r3, [pc, #32]   ; (8000750 <HAL_IncTick+0x30>)
 8000730:   f859 3003   ldr.w   r3, [r9, r3]
 8000734:   681b        ldr r3, [r3, #0]
 8000736:   4413        add r3, r2
 8000738:   4a05        ldr r2, [pc, #20]   ; (8000750 <HAL_IncTick+0x30>)
 800073a:   f859 2002   ldr.w   r2, [r9, r2]
 800073e:   6013        str r3, [r2, #0]
}
 8000740:   bf00        nop
 8000742:   46bd        mov sp, r7
 8000744:   f85d 7b04   ldr.w   r7, [sp], #4
 8000748:   4770        bx  lr
 800074a:   bf00        nop
 800074c:   00000010    .word   0x00000010
 8000750:   00000014    .word   0x00000014
```

这样所有在 RAM 里的全局变量都是相对于 GOT 的偏移。注意，这个时候你编译出来的代码现在没有办法进行测试，尽管你只是改了编译选项。这是因为 PIC 的基址需要你通过寄存器 r9 显式指定。在本例中，我们在链接脚本里如下定义 GOT 的位置：

```
/* Initialized data sections goes into RAM, load LMA copy after code */
.data :
{
  . = ALIGN(4);
   sdata = .;        /* create a global symbol at data start */
  GOT_START = .;
  *(.got*)
  GOT_END = .;
  *(.data)           /* .data sections */
  *(.data*)          /* .data* sections */

  . = ALIGN(4);
  _edata = .;        /* define a global symbol at data end */
} >DTCMRAM AT> FLASH
```

因此，我们可以很容易的从.map 文件中获得 GOT_START 的 RAM 地址，0x2000 0000，它就是 PIC 的基址。如果想测试编译器选项是否如我们所期望，我们可以在 Reset_Handler 开始部分加上如下语句（参考后文内存布局的代码）：

```
ldr r9, =GOT_START
```

经过测试，我们可以确信，编译器选项的改动对我们最终执行结果没有影响。

值得注意的是，STM32 用户的代码，例如 RTOS 的移植， 也可能使用寄存器 r9。在这种情况，用户应当解决冲突。一般情况寄存器 r9 对应用程序并不是必要的。

3.2. 去掉或者替换掉那些包含绝对位置的库文件

我们要将位置无关的库去掉或者替换掉。在 STM32 参考代码里，我们需要 startup_xxx.s 里 C 库调用去掉。示例如下：

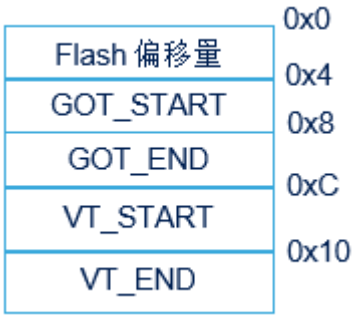
```
/* Call static constructors */
/*bl __libc_init_array*/
```

3.3. 修改 Flash 绝对地址

3.3.1. 内存布局

如果要对代码中的 Flash 绝对地址进行修改，我们需要知道存放 Flash 绝对地址的 RAM 起始和结束地址，以及需要增加或减少的 Flash 偏移量。存放 Flash 绝对地址的 RAM 起始和结束地址，在编译时可以让应用代码本身借助自身链接脚本在链接时导出的变量得到，然后由应用程序在运行时存放在 RAM 中的固定位置；也可以在编译后从.map 文件或使用工具解析 elf 文件获得，然后作为应用程序一部分的元信息，例如，给应用程序加个头部存放元信息，由 Bootloader 下载并解析，将其放入到 RAM 固定位置。

我们规划在一段 RAM 里按如下顺序存放如下元信息，它可以是应用程序本身在最初阶段自我存放在这里，也可以简单的由 Bootloader 解析元信息后，跳转到应用程序之前就存放在这里。



我们在前文已经在链接脚本中定义了 GOT_START 和 GOT_END，我们还需要在链接脚本中定义 VT_START 和 VT_END。如下图所示：

```
.isr_vector :
{
  . = ALIGN(4);
  VT_START = .;
  KEEP(*(.isr_vector))
  VT_END = .;
  . = ALIGN(4);
} >FLASH
```

如果我们希望 Bootloader 仅仅是做简单的跳转，我们可以将规划这段内存的工作，交给应用程序的初始化部分（在“`ldr sp, =_estack`”之前）。假定 0x0 处对应为 0x2400 0000，参考代码如下：

```
Reset_Handler:
  /*Store offset into r8 and 0x24000000*/
  mov r8, PC
  sub r8, 4
  ldr r1, =Reset_Handler
  sub r8, r1
  add r8, #1
  ldr r1, =0x24000000
  str r8, [r1, #0]

  /*Store GOT_START to 0x24000000 +0x4*/
  add r1, #4
  ldr r2, =GOT_START
  str r2, [r1, #0]

  /*Store GOT_END to 0x24000000 + 0x8*/
  add r1, r1, #4
  ldr r2, =GOT_END
  str r2, [r1, #0]

  /*Store VT_START to 0x24000000 +0xC*/
  add r1, #4
  ldr r2, =VT_START
  str r2, [r1, #0]

  /*Store VT_END to 0x24000000 + 0x10*/
  add r1, #4
  ldr r2, =VT_END
  str r2, [r1, #0]

  /*Setup gloabl PIC database*/
  ldr r9, =GOT_START
```

如果我们在 C 代码里进行要使用元信息，则 C 的宏定义如下：

```
#define FLSAH_OFFSET_CONTAINER 0x24000000
#define GOT_START_CONTAINER 0x24000004
#define GOT_END_CONTAINER  0x24000008
#define VT_START_CONTAINER 0x2400000C
#define VT_END_CONTAINER 0x24000010
#define VT_RAM_START 0x24001000
```

对于汇编代码，我们已经将偏移量存放在 r8 里，在需要时可直接使用。

3.3.2. 汇编代码

3.3.2.1. _sidata

在默认的 STM32 工程中，还有一些对变量绝对地址的使用。在 startup_xxx.s 有许多地方使用绝对地址，它们不能被编译器收集到 GOT 中。其中，默认在链接脚本里的_sidata，标志 flash 里 RAM 数据区的 Flash 位置，需要修改。

```
                0x000000000800259c                _sidata = LOADADDR (.data)

.data           0x0000000020000000       0x30 load address 0x000000000800259c
                0x0000000020000000                . = ALIGN (0x4)
                0x0000000020000000                _sdata = .
                0x0000000020000000                GOT_START = .
 *(.got*)
                0x0000000020000000                GOT_END = .
 *(.data)
 .data          0x0000000020000000       0x1c Drivers/BSP/stm32h7xx_nucleo/stm32h7xx_nucleo.o
                0x0000000020000000                COM_USART
 .data          0x000000002000001c        0x8 Drivers/CMSIS/system_stm32h7xx.o
                0x000000002000001c                SystemCoreClock
                0x0000000020000020                SystemD2Clock
 .data          0x0000000020000024        0x5 Drivers/STM32H7xx_HAL_Driver/stm32h7xx_hal.o
                0x0000000020000024                uwTickPrio
                0x0000000020000028                uwTickFreq
 *fill*         0x0000000020000029        0x3
 .data          0x000000002000002c        0x4 Example/User/main.o
                0x000000002000002c                uwExpectedCRCValue
 *(.data*)
                0x0000000020000030                . = ALIGN (0x4)
                0x0000000020000030                _edata = .
```

注意，变量绝对地址本身不是个问题，而对它解应用，取它的内容才会发生错误。而这里的 _sidata 是要被初始化代码使用，目的是将 Flash 的内容搬移到 RAM 里。我们显然要对_sidata 进行修改，否则无法取得正确的内容到 RAM 里。

根据前文的内存布局，我们可以把 Flash 的偏移量从内存中放置在寄存器 r8 里，例如：

```
ldr r1, = 0x24000000
str r8, [r1, #0]
```

则我们只需要一行简单的代码 “add r3,r8” 就可以修正_sidata 的地址。

```
CopyDataInit:
  ldr  r3, = sidata
  add r3, r8 /*add flash offset if needed*/
  ldr  r3, [r3, r1]
  str  r3, [r0, r1]
  adds  r1, r1, #4
```

3.3.3. C 代码

3.3.3.1. 公共函数

如果一段内存的数据都是硬编码，我们只需要一个公共函数就可以对其循环进行修正。我们需要知道什么样的地址之外不是 Flash 地址，那么就对这样的值不做修改。例如，我们定义 0x1fff ffff 之外的就不是 Falsh 地址，相应的宏定义如下：

```
#define FLASH_ADDR_MAX 0x1fffffff
```

我们将该函数命名为 UpdateOffset，参考代码如下：

```
void UpdateOffset(unsigned int start, unsigned int end) {
    unsigned int *i;
    for (i = (unsigned int*) start; i < (unsigned int*) end; i++) {
        if (*i < FLASH_ADDR_MAX && *i != 0) {
            *i = *i + *(unsigned int*) (FLSAH_OFFSET_CONTAINER);
        }
    }
}
```

3.3.3.2. SCB->VTOR

在 C 语言中如果使用赋值语句进行硬编码，编译器也无法进行收集。例如在 system_stm32xxxx.c 中的 SystemInit 有如下语句：

```
  /* Configure the Vector Table location add offset address ------------------*/
#ifdef VECT_TAB_SRAM
  SCB->VTOR = D1_AXISRAM_BASE  | VECT_TAB_OFFSET;       /* Vector Table Relocation in Internal SRAM */
#else
  SCB->VTOR = FLASH_BANK1_BASE | VECT_TAB_OFFSET;       /* Vector Table Relocation in Internal FLASH */
#endif
```

对于 VTOR 的值，随着映像放在不同的位置，肯定是不一样的。于是，我们先将将这里的赋值语句去掉，参考代码如下：

```
#ifdef VECT_TAB_SRAM
/*  SCB->VTOR = D1_AXISRAM_BASE  | VECT_TAB_OFFSET; */      /* Vector Table Relocation in Internal SRAM */
#else
/*  SCB->VTOR = FLASH_BANK1_BASE | VECT_TAB_OFFSET; */      /* Vector Table Relocation in Internal FLASH */
#endif
```

中断向量表相关的内容需要修改，包括两部分：

- 中断向量表的内存位置
- 中断向量表的内容

我们应该将中断向量表复制到 RAM 里，通过 UpdateOffset 函数修正其中包含的所有 Flash 绝对地址的值，同时通过对 SCB->VTOR 赋值来将中断向量表的位置指向我们修改过内容的 RAM 地址。注意，**VTOR 所指向的地址 VT_RAM_START 要按照 ARM 要求，根据中断总大小向上进行 2 的幂次对齐**，例如，37 个字大小要使用 64 个字对齐。另外，中断向量表的内容，也包含有 RAM 地址，对此，我们并不需要修改。当然，UpdateOffset 函数已经考虑到这一点，所以我们可以直接使用它。更新中断向量表以及 VTOR 的参考代码如下：

```
void UpdateVTOR(void) {
    unsigned int vt_start = *((unsigned int*) VT_START_CONTAINER)
            + *(unsigned int*) (FLSAH_OFFSET_CONTAINER);
    unsigned int vt_end = *((unsigned int*) VT_END_CONTAINER)
            + *(unsigned int*) (FLSAH_OFFSET_CONTAINER);
    unsigned int *i = (unsigned int*) vt_start;
    unsigned int *pRAM_VTOR = (unsigned int*) VT_RAM_START;
    while (i < (unsigned int*) vt_end) {
        *pRAM_VTOR++ = *i++;
    }
    UpdateOffset(VT_RAM_START, VT_RAM_START + vt_end - vt_start);
    SCB->VTOR = VT_RAM_START;
}
```

我们要在任何需要中断的代码之前将中断向量表更新完毕。这里我们在 startup_xxx.s 的 “bl SystemInit”前加入对 UpdatVTOR 的调用。

```
  bl UpdateVTOR
/* Call the clock system intitialization function.*/
  bl  SystemInit
```

3.3.3.3. GOT

编译器已经将 C 语言中所有全局变量的地址都收集到 GOT 中，因此我们很容易对其 Flash 地址的内容进行修正，参考代码如下：

```
void UpdateGOT(void) {
    unsigned int got_start = *((unsigned int*) GOT_START_CONTAINER);
    unsigned int got_end = *((unsigned int*) GOT_END_CONTAINER);
    UpdateOffset(got_start, got_end);
}
```

我们要在所有需要全局变量的应用之前更新 GOT，所有我们要将其放在 SystemInit 之前，如下所示：

```
  bl UpdateGOT
  bl UpdateVTOR
/* Call the clock system intitialization function.*/
  bl  SystemInit
```

4. 总结

除非你仅仅是运行一小块代码，否则开发位置无关的 STM32 完整工程，不仅仅要设置正确的编译器选项，还要保证它所链接的预编译的库不含有绝对地址引用，要保证所有源代码里没有对绝对地址的硬编码，包括修改 data 区的 Flash 起始地址，中断向量表的内容与位置，以及 GOT 的内容。

参考文献

文件编号	文件标题	版本号	发布日期
1	UM2609 STM32CubeIDE user guide	3	18-Feb-2021

文档中所用到的工具及版本

STM32CubeIDE 1.6.0

版本历史

日期	版本	变更
2021 年 12 月 14 日	1.0	首版发布