71x_init.s & 71x_vect.s 代码分析

—— 07 年 3 月 1 日

这段时间，为了参加 ST 的大赛，学到很多关于 ARM 的东西，我以前对 ARM 是一窍不通，现在十窍也通了有七八窍了吧。由于我们组做的是 uClinux 的方案，所以先就要弄 bootloader 了，ST 官方有一个开发板有写好了的 bootloader 和 uClinux 的 Patch，ST 网站上有下载，都是基于 ST 官方的函数库来的。所以就先找来学习学习。

这里面最难下手的就是 Init 下的两个汇编文件了，这两个文件就是 ST 发布的 STR710 的函数库里面的两个文件。这篇文章算是我自己学习这两个文件做的笔记，整理一下，发上来，希望对后来的朋友会有帮助。

先说一下，网上有一位叫杜云海的前辈写了三篇 ARM 学习报告，分别是“ARM 映象文件及执行机理”、“BootLoader 源代码级分析第一部分－GNU 之映象机理”、“ARM 学习报告 003——Bios 源码分析”，真的很感谢这位前辈的总结，对我这样的 ARM 新手帮助很大。还没看过的朋友可以在网上搜索一下，看过之后，也许就不用看我这个分析了，因为你自己也可以了。

我主要从程序的执行流程来分析。STR710 启动时会从地址的 0x0 处去执行，重启时也会从 0x00 处去执行，这是硬件强制的，而一般对于 ARM 来说,地址 0x00 处存放的是中断向量表，STR710 也一样，这两个文件在编译后连接时 71x_vect.s 是在前面的，一定要在前面(在 ADS 里面可以修改 armlink 连接文件的顺序)，而且我把 armlink 的 RO_Base 设置在 0x00 处，RW_Base 不设置，也就是紧跟在 RO_Base 后面。在这里程序的加载域和运行时域一样(不明白的可以参考上面提到的杜前辈的 ARM 学习报告)。所以 71x_vect.s 连接后运行时域在 0x00 处，而我们会把最终生成的文件拷贝到 STR710 的内部 FLASH 的 0x00，也就是系统地址空间的 0x4000 0000 处，并把内部 FLASH 映射到系统地址 0x0000 0000 处(通过配置 STR710 的启动模式)，这些系统启动时，就会先看到 71x_vect.s 中的代码，下面我们就来看下这个文件中有什么。

```
12          PRESERVE8

13          AREA Vect, CODE, READONLY
```

12 行是一个伪指令，指示当前文件保持堆栈为 8 字节对齐，13 行说明下面的内容属于 Vect 代码段。这里没什么好说的，接下来几行定义了几个常量，从外部文件导入了很多符号，也导出了几个其他文件需要的符号。从 vector_begin 开始，好戏开始了：

```
vector_begin

89          LDR     PC, Reset_Addr

90          LDR     PC, Undefined_Addr

91          LDR     PC, SWI_Addr

92          LDR     PC, Prefetch_Addr

93          LDR     PC, Abort_Addr

94          NOP                             ; Reserved vector

95          LDR     PC, IRQ_Addr

96          LDR     PC, FIQ_Addr
```

这就是 ARM 的中断向量表了，从 ARM 的数据手册可以知道:

地址(字节地址)	异常	进入模式
0x000 0000	复位	管理
0x000 0004	无定义指令	无定义
0x000 0008	软件中断	管理
0x000 000c	异常(预取)	异常
0x000 0010	异常(数据)	异常
0x000 0014	保留	--
0x000 0018	IRQ	IRQ
0x000 001c	FIQ	FIQ

因为每一条指令刚好 32 位，占用一个字的空间，所以从 89—96 行每行就对于这上表中的地址，而这个地址中存储的内容也就是上面的 LDR PC,...的指令。这就是中断向量表的概念，那么系统启动时会怎么样呢？系统启动时，强制从 0x00 处取指令，这时 CPU 取到的指令也就是 LDR PC, Reset_Addr，下一步 CPU 就会去执行 Reset Addr 处的指令了，并转入了管理模式。如果是复位的话也一样，复位也是一种异常，从这里看 Reset Addr 后的指令也就有点像我们写的中断服务程序了，系统启动或复位时产生中断，然后执行 Reset Addr 处的指令。

从后面紧跟的异常处理地址表可以看到：

```
102 Reset_Addr      DCD     Reset_Handler
```

而 Reset_Handler 是 71x_init.s 中导出的符号，也就是说这里就进入 71x.init.s 中执行代码了。说到这里，我刚开始接触这次比赛时，对这个问题搞不清楚，总以为代码从 71x_init.s 执行，以为它的代码在地址的 0x00 处，现在才知道，其实 71x_init.s 最后连接在什么位置并不是很关键，只和你采用什么中断调用方法有关(后面会讲到，STR710 有 3 中中断调用方法）。

后面是其他异常处理地址表和外设中断地址表，然后定义了两个宏 SaveContext 和 RestoreContext 分别用来在进入各种中断处理时保存上下文，退出时恢复上下文。下面主要分析一下 IRQHander 的内容：

```
221 IRQHandler

222         SUB     lr,lr,#4        ; Update the link register
223         SaveContext r0,r11      ; Save the workspace plus the current
224                                 ; return address lr_ irq and spsr_irq.
225         LDR     lr, =ReturnAddress ; Read the return address.
226         LDR     r0, =EIC_base_addr
227         LDR     r1, =IVR_off_addr
228         ADD     pc,r0,r1        ; Branch to the IRQ handler.
229 ReturnAddress
230                                 ; Clear pending bit in EIC (using the proper IPRx)
231         LDR     r0, =EIC_base_addr
232         LDR     r2, [r0, #CICR_off_addr] ; Get the IRQ channel number.
233         MOV     r3,#1
234         MOV     r3,r3,LSL r2
235         STR     r3,[r0, #IPR_off_addr]   ; Clear the corresponding IPR bit.
236         RestoreContext r0,r11   ; Restore the context and return to the...
237                                 ; ...program execution.
```

首先，在保存上下文前为什么要把 lr 寄存器的内容减 4 呢？这涉及到 ARM 的指令预取的问题，在 ARM7

中，PC 寄存器的值是当前正执行的指令的地址+8 处，这是因为 ARM7 是三级流水线结构，在执行一条指令时，同时译码另一条指令，同时读取第三条指令，还是举个例子来说把，如下：

0x3000 指令 A

0x3004 指令 B

0x3008 指令 C

三条指令在内存中如上分部，当 ARM 执行 0x3000 处的指令 A 时，就在译码 0x3004 处的指令 B，同时读取 0x3008 处的指令 C，所以此时 PC 的值是 0x3008。

假如指令 A 是跳转指令（调用子程序），或是软件中断指令 SWI，那么调用返回后应该继续执行指令 B，此时在执行指令 A 时，系统自动把 PC 即 0x3008 保存到 LR 连接寄存器中，然后对 LR 进行减 4 操作（也是自动的），这样 LR 中的值就是 0x3004,当返回时，把 LR 的值恢复到 PC 中，刚好就是执行指令 B。

假如指令 A 处发生的了 IRQ 或 FIQ 中断，也就是我们现在的情况，那么中断返回应该继续执行指令 A，因为 A 被中断打断了，那么在系统自动把 PC 保存到 LR 中，并自动把 LR-4 后，我们还要再在程序中做一次 LR-4 的操作，也就是 LR-8，才能使 LR 保存指令 A 的地址 0x3000，使得返回时继续执行指令 A，这就是这里为什么要 SUB lr,lr,#4 了。

还有一种情况，就是 Data Abort(数据异常),导致这样的异常的原因应该是上一条指令，也就是 0x2ffc 的指令，所以应该把 LR-8(加上自动减的 4,总共减了 12)。

这里搞清楚了，后面也就简单了，就是先保存上下文，然后跳转到 EIC_IVR 执行，而这时 EIC_IVR 的内容是跳转到当前最高优先级挂起中断例程的指令(这在 71x_init.s 中设置)，然后中断返回后清除相关寄存器的相关位，恢复上下文，程序继续执行...

在后面还有两个宏，IRQ_to_SYS 和 SYS_to_IRQ，这两个宏在后面的每个 Handler 中都有调用，这是用来做什么呢？很明显，IRQ_to_SYS 用来从 IRQ 模式切换到 SYS 模式，另一个刚好相反，如下：

```
321 TOTIMIIRQHandler

322         IRQ_to_SYS
323         BL      TOTIMI_IRQHandler
324         SYS_to_IRQ
```

可以看到，具体的中断例程是在 SYS 模式下运行的，而不是在 IRQ 模式下运行的，为什么呢？原来这和嵌套的中断有关，当系统相应一个 IRQ 中断时，系统进入异常相应状态，就会自动关闭 IRQ 中断使能，不能相应其他中断，如果此时强制打开中断使能，那么如果现在相应另外一个中断，那么新的中断的中断现场就会和原来的中断现场产生冲突，使得程序不一定会正确的执行，也就不能嵌套中断了，上面这就是一种解决办法，把中断例程放到 SYS 模式下运行，在进入 SYS 模式后，开中断使能，就可以接受新的中断了，这样就不会产生冲突，运行完后再返回 IRQ 模式，最终恢复正常运行。

71x_vect.s基本分析完了，下面就分析 71x_init.s 了，主要分析这几个方面的问题，一是堆栈的设置，二是地址重映射，三是 EIC 初始化。

一，设置堆栈：

```
 99         MSR     CPSR_c, #Mode_ABT:OR:F_Bit:OR:I_Bit

100         LDR     SP, =ABT_Stack
```

如上所示，堆栈的设置要通过写 CPSR_c 来切换到不同的模式，再给相应的 SP 寄存器赋值，ARM7 有 6 种工作模式，每种都要设置对应的堆栈指针，在这个程序里，把堆栈安排在了 STR710 内部 SRAM 的高位地址向下增长，这里要说明的是把堆栈安排在片内 SRAM，对提高系统性能有好处。

二，地址重映射：

地址重映射是嵌入式的一个重要概念，特别是在 Bootloader 中，最上面提到的 ARM 学习报告里有详细讲，不多说了，看代码：

```
    ;first we check if RAM is already mapped.Skip if so.

119                 LDR             r1, =CPM_Base_addr
120                 LDRH    r2, [r1, #BOOTCONF_off_addr];Read BOOTCONF Register
121                 AND             r2, r2, #0x03
122                 CMP             r2, #RAM_mask
123                 BEQ             end_remap
124
125     ;copying whole image into RAM
126         LDR     r0, =0                                  ;r0 = start address from which to copy
127         LDR     r3, =|Image$$ZI$$Base|      ;r3 = number of bytes to copy
128         LDR     r1, =RAM_Base               ;r1 = start address where to copy
129 copy_ram
130         LDR     r2, [r0], #4                ;Read a word from the source
131         STR     r2, [r1], #4                ;copy the word to destination
132         SUBS    r3, r3, #4                  ;Decrement number of words to copy
133         BNE     copy_ram
134
135
136         MOV     r0, #RAM_mask
137         ;now do mapping
138         LDR     r1, =CPM_Base_addr
139         LDRH    r2, [r1, #BOOTCONF_off_addr];Read BOOTCONF Register
140         BIC     r2, r2, #0x03               ;Reset the two LSB bits of BOOTCONF Register
141         ORR     r2, r2, r0                  ;change the two LSB bits of BOOTCONF Register
142         STRH    r2, [r1, #BOOTCONF_off_addr];Write BOOTCONF Register
143 end_remap
144 ;now zero init
145                 LDR     r1,     =|Image$$ZI$$Base|
146                 LDR             r3,   =|Image$$ZI$$Limit|
147                 MOV     r2, #0
148 zero_init
149                 STR             r2, [r1], #4
150                 CMP             r1, r3
151                 BCC             zero_init
152
```

这是很重要的一段代码，需要先说明的是，程序的 119 - 143 行是在内部 Flash 中执行的，从 145 行开始，以后所有的代码都是在内部 SRAM 中执行了，这就是地址重映射的效果。119 - 122 行是取 BOOT 寄存器的[1:0]位，判断现在的映射是怎样的，如果内部 SRAM(0x2000 0000)已经映射到系统的 0x0000 0000 处，那么就跳过数据复制和地址重映射的过程，如果内部 SRAM 还没有映射到 0x0000 0000 处，那么就执行 copy_ram。试想，当系统从断电第一次启动时，系统根据三个 BOOT 配置引脚的状态来设置 BOOT 寄存器，我们把 Bootloader

下载到了内部 FLASH 中，所以配置系统从内部 FLASH 启动，这时上面的代码最先在内部 FLASH 中运行，copy_ram 部分的代码就不会跳过，而如果系统从正常运行复位，此时内部 SRAM 已经映射到地址的 0x0000 0000 处了，而且相关代码已经处于内部 SRAM 中了，系统直接从内部 SRAM 运行 Reset_Handler，也就不需要运行 copy_ram 部分的代码了。

从第 126 行开始，准备把 Bootloader 整个复制到 SRAM 中去，126 行载入 0 到 r0，127 行载入要拷贝的数据的终点，这里还是提一下这几个符号的意思，下面几个符号都是由 ARMLink 程序产生的，我们可以在程序中调用。

|Image$$RO$$Base| 代码连接后生成可执行文件的运行时域 RO 输出段起始地址

|Image$$RO$$Limit| 代码连接后生成可执行文件的运行时域 RO 输出段截止地址

|Image$$RW$$Base| 代码连接后生成可执行文件的运行时域 RW 输出段起始地址

|Image$$RW$$Limit| 代码连接后生成可执行文件的运行时域 RW 输出段截止地址

|Image$$ZI$$Base| 代码连接后生成可执行文件的运行时域 ZI 输出段起始地址

|Image$$ZI$$Limit| 代码连接后生成可执行文件的运行时域 ZI 输出段截止地址

在这里，我的前提是在编译连接时，设置 RO_Base 为 0x00,Rw_Base 没设置(即 |Image$$RO$$Limit|)而且，编译生成的文件最终会下载到内部 FLASH 的 0x00 处，那么当代码还在 FLASH 中执行时，其实它的加载域和运行时域是一样的，最终拷贝到内部 SRAM 的 0x00(地址 0x2000 0000 处)，再把内部 SRAM 映射到 0x00 处后，也是一样的。

所以此处的|Image$$ZI$$Base| 即 |Image$$RW$$Limit| 也等于这个代码在加载域时的 RW 段截止地址，这里要拷贝的数据是所有的 RO 和 RW 段，ZI 段不用拷贝，因为 ZI 段是所有要初始化为 0 的变量。128 行把拷贝的目标地址载入 r1(此时内部的 SRAM 的起始地址还只有 0x2000 0000，因为地址还没有重映射),129 - 133 行循环把数据从内部 FLASH 0x00 复制到内部 SRAM 0x2000 0000,每次复制一个字的数据,直到|Image$$RW$$Limit|.

136 - 143 行就是设置 BOOT[1:0]寄存器位，把 SRAM 映射到系统地址空间的 0x00 处，这些寄存器的操作 STR710 的手册中有很详细的资料。从 144 行还是，CPU 取到的指令已经是在内部 SRAM 中的指令了，这样就很巧妙的完成了程序从 FLASH 到 SRAM 的转换。144 - 151 的内容就是把 ZI 段的所以内容初始化为 0.

其实这里遇到的是一种比较简单的情况，因为在很多情况下 RW 段的加载域和运行时域并不一定一样，RW 段的加载域一定是紧跟着 RO 段，但运行时域可以设置，这里不设置 RW 段的运行时域(Rw_Base)，也就是让它的加载域和运行时域一样，其实在这种情况下，只要把程序拷贝的位置和 RO 段的运行时域(Ro_Base)对上，在这里就是：“RO 段的运行时域是从 0x00 处开始，开始映象拷贝在内部 FLASH 的 0x00，并把内部 FLASH 映射在系统地址空间的 0x00”，那么如果不进行地址重映射，代码也可以正常运行，只不过没有在 SRAM 中性能好。而如果你设置了 Rw_Base，那么在运行时就必须把 RW 段的数据从 RO 段的尾部复制到 Rw_Base 处，程序才会在需要用到 RW 段的数据时正常运行，通常会把 RW 段从 ROM 复制到 RAM，再通过地址重映射达到目的。

不知道我说清楚没有，现在真的感觉一个问题懂了，要说清楚也不是一个简单的事，其实我说的很罗嗦，越怕说不明白，就越罗嗦了。

三、EIC 初始化：

刚开始接触 STR710 时，对它的中断处理不是很明白，看这个代码时仔细研究了下，才大致搞明白，以 TOTIMI 中断为例，当程序执行过程中，产生 TOTIMI 中断时，系统会强制跳转到 0x0000 0018 处执行，而

0x0000 0018处的指令就是 LDR PC, IRQ_Addr，然后，IRQ_Addr处执行的就是IRQ_Handler，定义在71x_vect.s中，从其代码可以看出，IRQ_Handler先保存上下文，然后就跳转到EIC_IVR处去执行了，带中断处理代码执行完返回后，IRQ_Handler再清除相关IPR位，恢复上下文。

EIC_IVR的地址是0xFFFF F800 + 18 即0xFFFF F818,它的内容是什么呢？这里涉及到STR710采用那种中断调用方式，STR710可以采用三种中断调用方式，我们先不管它，先看这个程序里是怎么做的。

```
164 EIC_INIT

165         LDR     r3, =EIC_Base_addr
166         LDR     r4, =0x00000000
167         STR     r4, [r3, #ICR_off_addr]   ; Disable FIQ and IRQ
168         STR     r4, [r3, #IER_off_addr]   ; Disable all channels interrupts
169         LDR     r4, =0xFFFFFFFF
170         STR     r4, [r3, #IPR_off_addr]   ; Clear all IRQ pending bits
171         LDR     r4, =0x0C
172         STR     r4, [r3, #FIR_off_addr]   ; Disable FIQ channels and clear FIQ pending bits
173         LDR     r4, =0x00000000
174         STR     r4, [r3, #CIPR_off_addr]  ; Reset the current priority register
```

这几行主要是做一些准备工作。

```
175         LDR     r4, =0xE59F0000

176         STR     r4, [r3, #IVR_off_addr]   ; Write the LDR pc,pc,#offset instruction code in
IVR[31:16]
```

这两行就是关键，可以看到，在EIC初始化的过程中，EIC_IVR的高16位的内容是0xE59F,这是什么意思呢？其实这就是 LDR pc,pc,#offset 这条指令，而#offset就在EIC_IVR的低16位了，我们直到，EIC_IVR的低16位的内容就是当前具有最高优先级的有效中断所对应的EIC_SIR寄存器中低16位的内容，这是由系统控制的。STR710一共有32个EIC_SIR寄存器，每个IRQ中断都对应一个，我们在EIC初始化中的一个重要工作就是设置这些寄存器，它们保存着从EIC_IVR+8到当前中断处理程序处的偏移量，为什么是+8呢？聪明的马上就知道了，还是ARM指令预取的问题，当执行EIC_IVR处的指令时，PC的值实际上是EIC_IVR+8。

从71x_vect.s中知道，32个IRQ中断处理程序设置在异常处理地址表后面，

```
177         LDR     r2, =32                   ; 32 Channel to initialize

178         LDR     r0, =T0TIMI_Addr          ; Read the address of the IRQs address table
179         LDR     r1, =0x00000FFF
180         AND     r0,r0,r1
```

这里R0最终就是 T0TIMI_Addr 相对 0x0000 0000 的偏移量。

```
81          LDR     r5, =SIR0_off_addr        ; Read SIR0 address

182         SUB     r4,r0,#8                  ; subtract 8 for prefetch
183         LDR     r1, =0xF7E8               ; add the offset to the 0x00000000 address(IVR
address + 7E8 = 0x00000000)
184                                           ; 0xF7E8 used to complete the LDR pc,pc,#offset
```

```
opcode
185          ADD     r1,r4,r1                ; compute the jump offset
186 EIC_INI  MOV     r4, r1, LSL #16         ; Left shift the result
187          STR     r4, [r3, r5]            ; Store the result in SIRx register
```

这里 81 行 R5 是 TOTIMI 对应 SIR 寄存器相对 EIC 基地址的偏移，182 行把 R0 减去了 8,应为指令预取的问题。

183 行很有趣，为什么说 0xF7E8 就是 EIC_IVR 的地址相对 0x0000 0000 的偏移量呢？EIC_IVR 的地址是 0xFFFF F818,0xFFFF F818 + 0xF7E8 = 0x1 0000 0000注意，这个地址已经超过 32 位寻址了，需要第 33 根地址线才能找到这个地址，现在 STR710 只有 32 根地址线，最高位的“1”也就被忽略了，系统认为 0x1 0000 0000 就是 0x0000 0000,0xF7E8也就是 EIC_IVR 相对 0x0000 0000的偏移量了。怎么样，很巧妙把。

最终，185 - 187 行把最终的偏移量放到了对应的 SIR 寄存器中了，并在后面的代码中循环完成了其他 SIR 寄存器的设置。

如果还不明白，那只能说我的表述能力还是不行，我画了张图，在后面，清晰的描述了这个问题，看看这张图把。

其实说到这，后面就没什么好说的了，后面设置了几个 SP 的值就跳转到 C 代码去执行了，C 代码大家都容易看了。

最后，关于 STR710 的另外两种中断调用的方式，大家可以参考 ST 官方的示例代码 **APPLICATION NOTE AN1776 “INTERRUPT HANDLING FOR STR7 MICROCONTROLLERS”.**

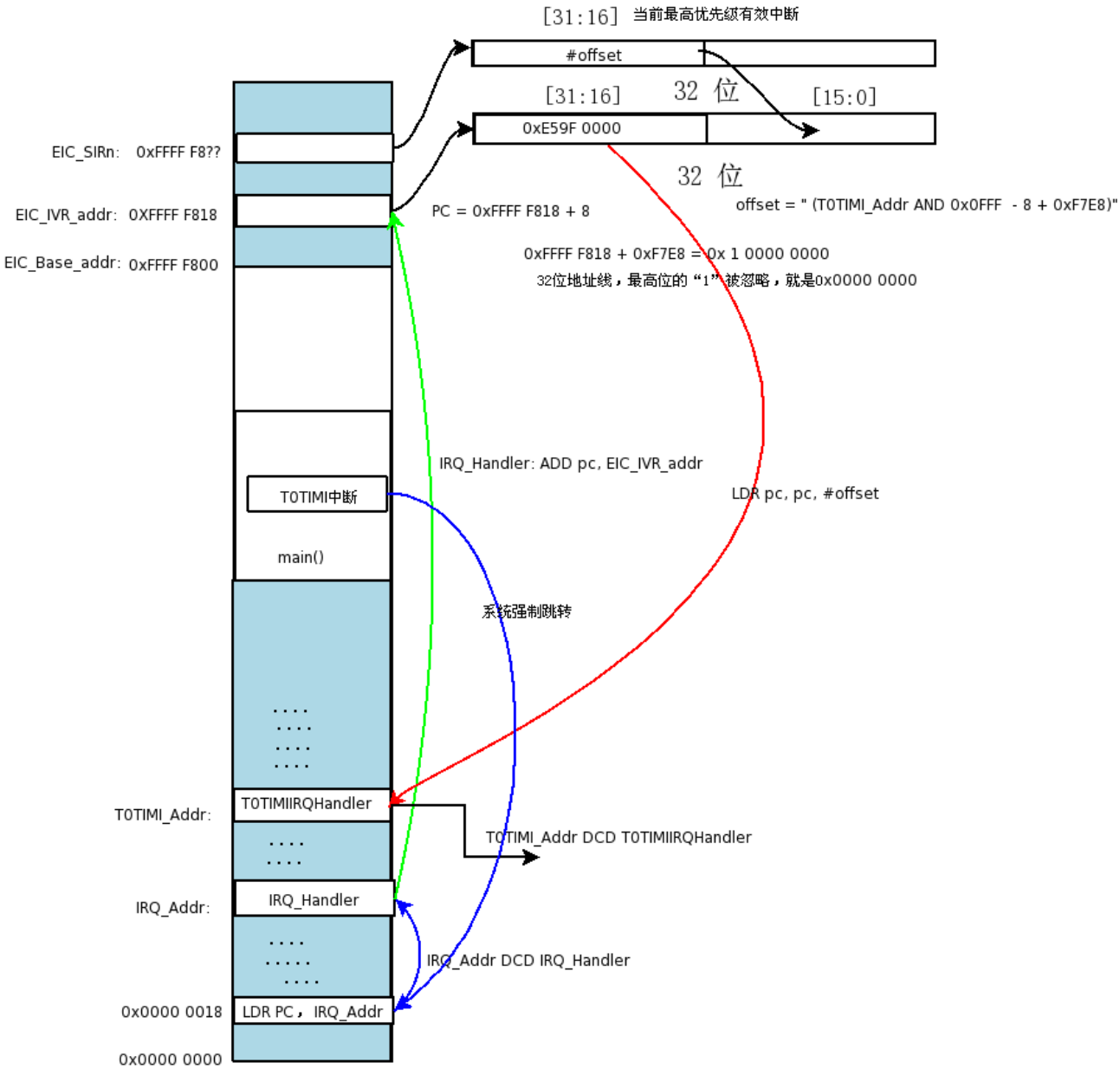
这段时间，我在看如何快速的将 SDT 下的汇编代码移植到 GNU 环境下的汇编代码的问题，到时候再做个类似的总结，就是一个学习的过程。

大家如果有什么问题可以和我联系，也欢迎没事随便聊聊^v^

Email: royllik768@gmail.com

MSN: Co_Coiiz@hotmail.com

如图（1）：


[31:16] 当前最高优先级有效中断
#offset
[31:16]
32 位
[15:0]
0xE59F 0000
32 位
EIC_SIRn: 0xFFFF F8??
EIC_IVR_addr: 0XFFFF F818
EIC_Base_addr: 0xFFFF F800
PC = 0xFFFF F818 + 8
offset = " (T0TIMI_Addr AND 0x0FFF - 8 + 0xF7E8)"
0xFFFF F818 + 0xF7E8 = 0x 1 0000 0000
32位地址线，最高位的“1”被忽略，就是0x0000 0000
IRQ_Handler: ADD pc, EIC_IVR_addr
LDR pc, pc, #offset
T0TIMI中断
main()
系统强制跳转
T0TIMI_Addr:
T0TIMIIRQHandler
T0TIMI_Addr DCD T0TIMIIRQHandler
IRQ_Addr:
IRQ_Handler
IRQ_Addr DCD IRQ_Handler
0x0000 0018
LDR PC， IRQ_Addr
0x0000 0000

图（1）