Antes do seu programa ser executado o microcontrolador ARM precisa ser inicializado

ARM Vector Remapping

- Inicialização do clock da CPU.
- Configuração do barramento de dados.
- Inicialização da CPU.
- (...)

Após o boot a função main() é chamada.

Programa blink na KL25Z

do\workspace.kds\EX03_BLINK\Project_Settings\Startup_Code

Nome	Data de modificaç...	Tipo	Tamanho
startup.c	02/12/2017 09:20	Arquivo C	5 KB

```c
__attribute__((naked)) void __thumb_startup(void)
{
int addr = (int)__SP_INIT;

    /* setup the stack before we attempt anything el
       skip stack setup if __SP_INIT is 0
       assume sp is already setup. */
    __asm (
    "mov  r0,%0\n\t"
    "cmp  r0,#0\n\t"
    "beq  skip_sp\n\t"
    "mov  sp,r0\n\t"
    "sub  sp,#4\n\t"
    "mov  r0,#0\n\t"
    "mvn  r0,r0\n\t"
    "str  r0,[sp,#0]\n\t"
    "add  sp,#4\n\t"
    "skip_sp:\n\t"
    ::"r"(addr));

    /* Setup registers */
    __init_registers();

    /* setup hardware */
    __init_hardware();
```

```c
    /* SUPPORT_ROM_TO_RAM */
    __copy_rom_sections_to_ram();

    /* initializations before main, user specific */
    __init_user();

#ifndef __ATOLLIC__
    _start();
#else
    /* zero-fill the .bss section */
    zero_fill_bss();
    /* Run static constructors */
    __libc_init_array();
    main();
#endif

    /*  should never get here */
    while (1);

}
```

Depois do boot vem a chamada da funcão Main()

O acesso a memória RAM é muito mais rápido do que a ROM.

Por esse motive, o programa em ROM é copiado para RAM durante o boot/startup

```c
/* SUPPORT_ROM_TO_RAM */
__copy_rom_sections_to_ram();
```

Chamada durante o boot

```c
/*
 *  Routine that copies all sections the user marked as ROM into
 *  their target RAM addresses ...
 *
 *  __S_romp is defined in the linker command file
 *  It is a table of RomInfo
 *  structures.  The final entry in the table has all-zero
 *  fields.
 */
void __copy_rom_sections_to_ram(void)
{

  int       index;

  if (__S_romp == 0L) return;

  /*
   *  Go through the entire table, copying sections from ROM to RAM.
   */
  for (index = 0;
       __S_romp[index].Source != 0 ||
       __S_romp[index].Target != 0 ||
       __S_romp[index].Size != 0;
       ++index)
  {
    __copy_rom_section( __S_romp[index].Target,
                        __S_romp[index].Source,
                        __S_romp[index].Size );

  }
}
```

```c
/*
 *  Routine to copy a single section from ROM to RAM ...
 */
void __copy_rom_section(unsigned long dst, unsigned long src, unsigned long size)
{
  unsigned long len = size;

  const unsigned int size_int = sizeof(int);
  const unsigned int mask_int = sizeof(int)-1;

  const unsigned int size_short = sizeof(short);
  const unsigned int mask_short = sizeof(short)-1;

  const unsigned int size_char = sizeof(char);

  if( dst == src || size == 0)
  {
    return;
  }


  while( len > 0)
  {

    if( !(src & mask_int) && !(dst & mask_int) && len >= size_int)
    {
      *((int *)dst)  = *((int*)src);
      dst += size_int;
      src += size_int;
      len -= size_int;
    }
    else if( !(src & mask_short) && !(dst & mask_short) && len >= size_short)
    {
      *((short *)dst)  = *((short*)src);
      dst += size_short;
      src += size_short;
      len -= size_short;
    }
    else
    {
      *((char *)dst)  = *((char*)src);
      dst += size_char;
      src += size_char;
      len -= size_char;
    }
  }
}
```

_S_romp é definido no linker.ld

```
_romp_at = ___ROM_AT + SIZEOF(.data);
.romp : AT(_romp_at)
{
  __S_romp = _romp_at;
  LONG(___ROM_AT);
  LONG(_sdata);
  LONG(___data_size);
  LONG(0);
  LONG(0);
  LONG(0);
} > m_data
```

__SP_INIT. O endereço inicial do *Stack Pointer* também é feito no linker.ld

```
/* Highest address of the user mode stack */
_estack = 0x20003000;    /* end of m_data */
__SP_INIT = _estack;
__stack = _estack;

/* Generate a link error if heap and stack don't fit into RAM */
__heap_size = 0x00;                    /* required amount of heap  */
__stack_size = 0x0400;                 /* required amount of stack */

MEMORY {
  m_interrupts (RX) : ORIGIN = 0x00000000, LENGTH = 0x000000C0
  m_text       (RX) : ORIGIN = 0x00000410, LENGTH = 0x0001FBF0
  m_data       (RW) : ORIGIN = 0x1FFFF000, LENGTH = 0x00004000
  m_cfmprotrom  (RX) : ORIGIN = 0x00000400, LENGTH = 0x00000010
}
```

```
__attribute__((naked)) void __thumb_startup(void)
{
int addr = (int)__SP_INIT;

    /* setup the stack before we attempt anything el
       skip stack setup if __SP_INIT is 0
       assume sp is already setup. */
    __asm (
    "mov  r0,%0\n\t"
    "cmp  r0,#0\n\t"
    "beq  skip_sp\n\t"
    "mov  sp,r0\n\t"
    "sub  sp,#4\n\t"
    "mov  r0,#0\n\t"
    "mvn  r0,r0\n\t"
    "str  r0,[sp,#0]\n\t"
    "add  sp,#4\n\t"
    "skip_sp:\n\t"
    ::"r"(addr));
```

e depois é configurado no boot

Para alocar uma determinada função em ROM para a RAM podemos utilizar a diretiva __attribute__((section (".text.fastcode")))

```
__attribute__ ((section (".text.fastcode")))
void toggle_red(void) {
    GPIOB_PTOR = (1 << 18);
}
```

```
.text.fastcode
                0x00000524        0x14 main.o
                0x00000524                toggle_red
```

```
/* zero-fill the .bss section */
zero_fill_bss();
```

No Process-Expert o setor .bss (Block Started by Symbol) é inicializado com zero.

Isso não é uma regra!

```
static void zero_fill_bss(void)
{
  extern char __START_BSS[];
  extern char __END_BSS[];

  unsigned long len = __END_BSS - __START_BSS;
  unsigned long dst = (unsigned long) __START_BSS;

  const int size_int = sizeof(int);
  const int mask_int = sizeof(int)-1;

  const int size_short = sizeof(short);
  const int mask_short = sizeof(short)-1;

  const int size_char = sizeof(char);

  if( len == 0)
  {
    return;
  }


  while( len > 0)
  {
    if( !(dst & mask_int) && len >= size_int)
    {
      *((int *)dst)  = 0;
      dst += size_int;
      len -= size_int;
    }
    else if( !(dst & mask_short) && len >= size_short)
    {
      *((short *)dst)  = 0;
      dst += size_short;
      len -= size_short;
    }
    else
    {
      *((char *)dst)  = 0;
      dst += size_char;
      len -= size_char;
    }
  }
}
```

Set de Instruções do ARM

ARM e THUMB(-2)

32-bits **16-bits**

1. Normalmente utilizado em MCUs que executam o program pela ROM/Flash.
2. Perdem menos tempo acessando a ROM, fato que implica em um consumo menor de energia elétrica!
3. Desempenho menor do que Set ARM.

Set de Instruções do ARM

ARM e THUMB(-2)

32-bits **16-bits**

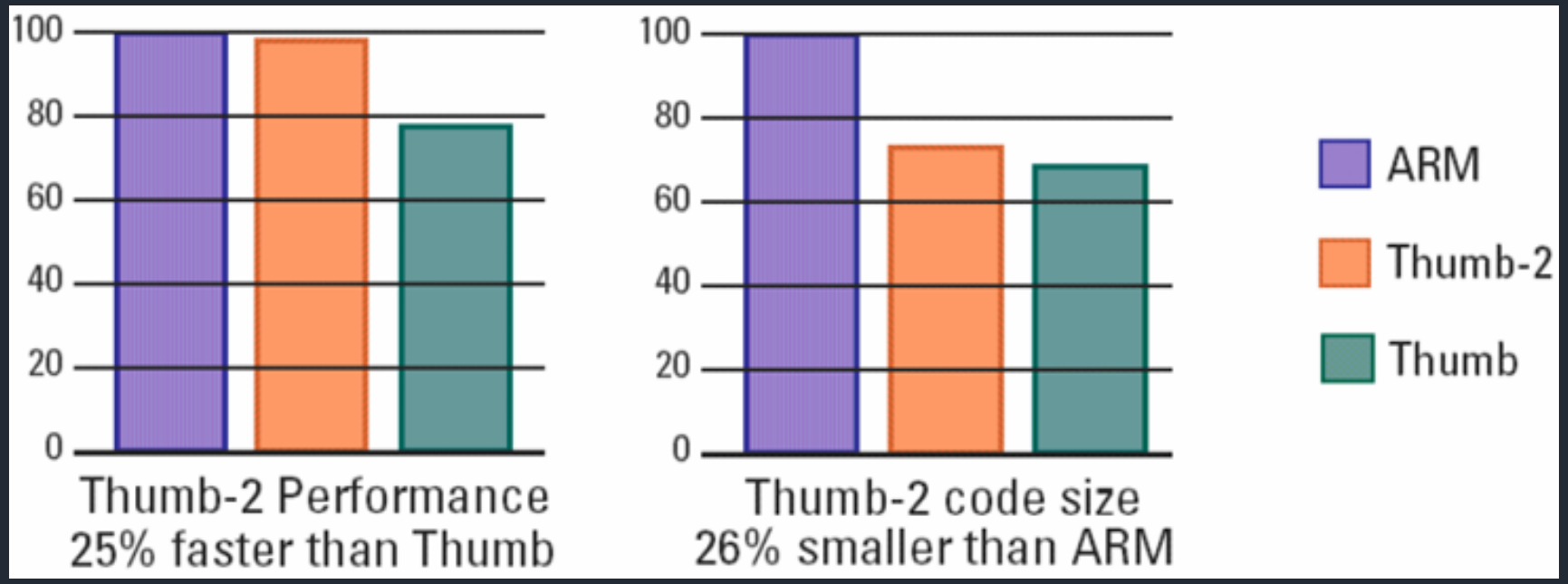
```
$(BINDIR)\low_level_init.o: $(BLDDIR)\low_level_init.c $(APP_DEP)
    $(CC) -marm $(CCFLAGS) $(CCINC) $<

$(BINDIR)\blinky.o: $(BLDDIR)\blinky.c $(APP_DEP)
    $(CC) -mthumb $(CCFLAGS) $(CCINC) $<
```

Set de Instruções do ARM

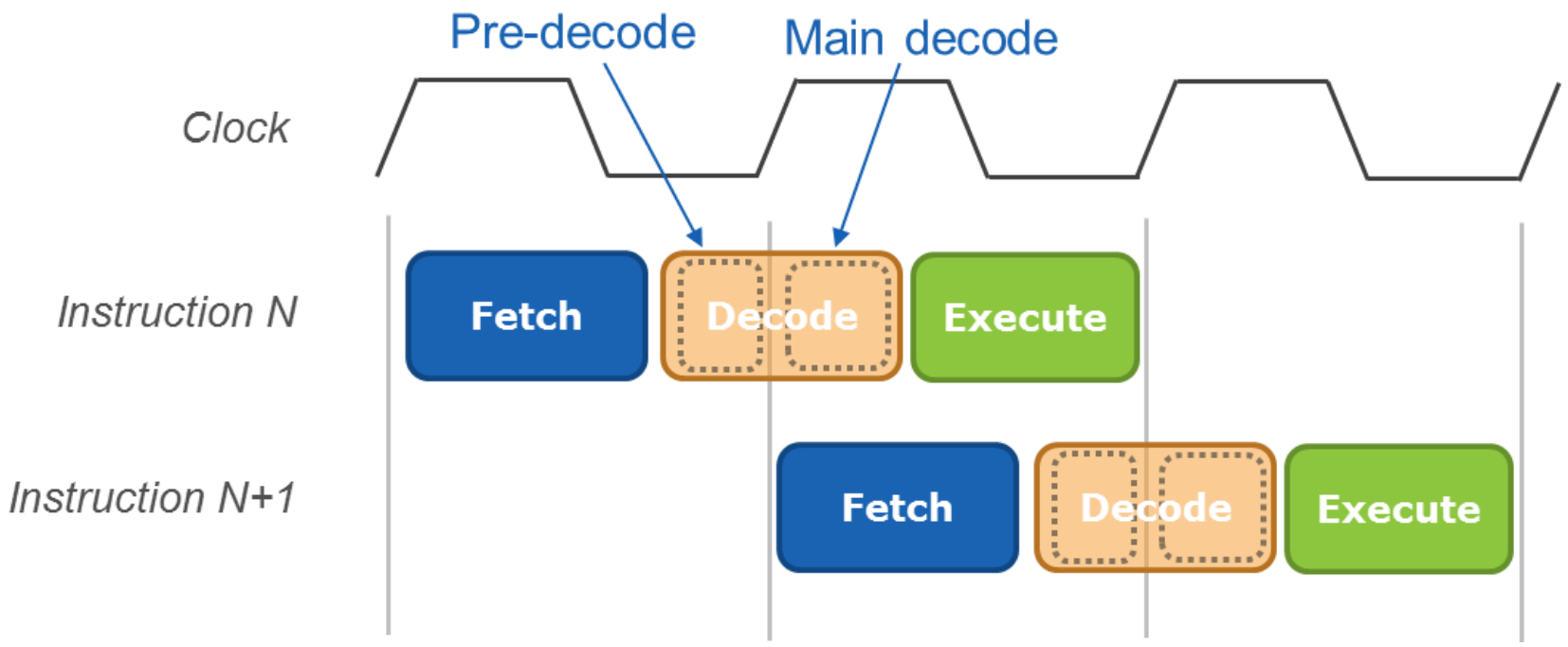
ARM e THUMB(-2)

32-bits

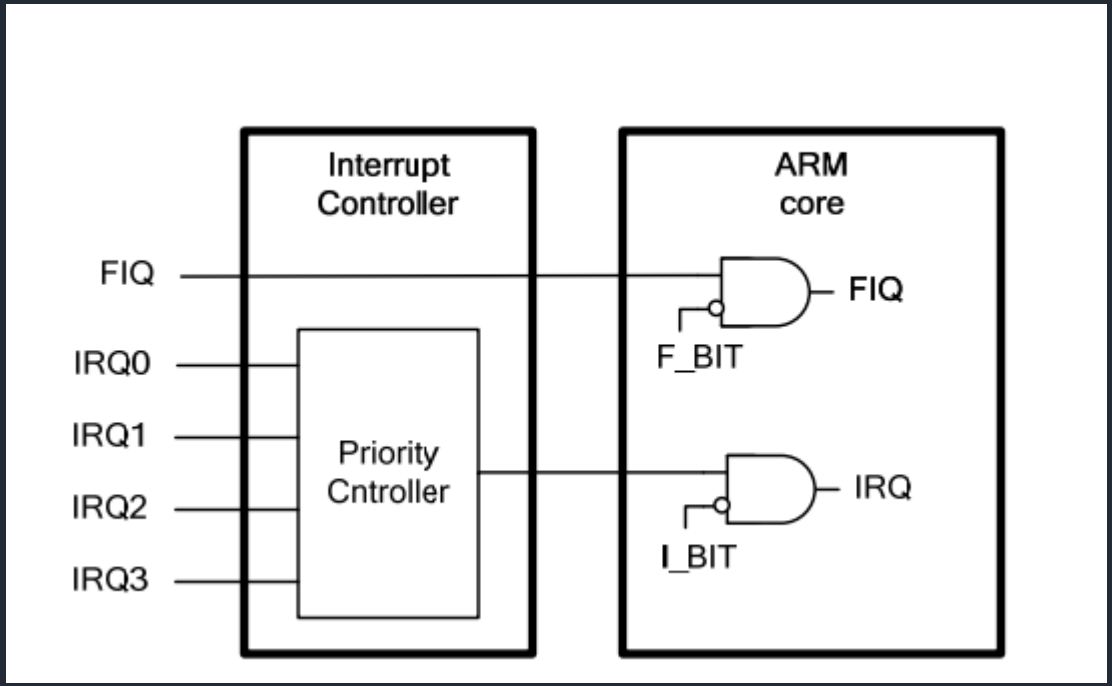
16-bits



O ARM Cortex M0+ utiliza apenas 2 estágios (*pipelines*) na execução das instruções e opera em THUMB-2.



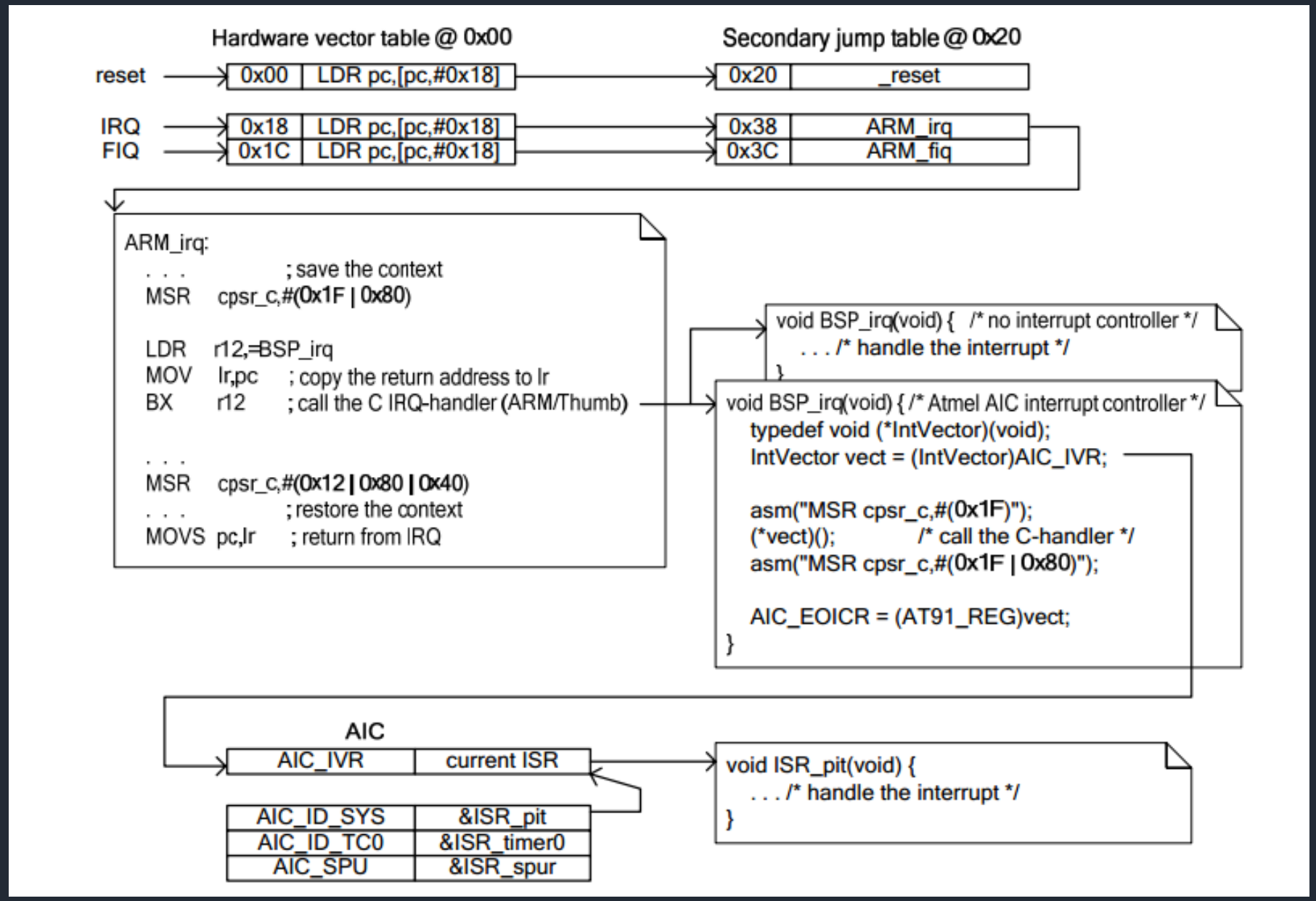
FIQ é diferente de IRQ
FIQ não possui controlador de prioridades como as IRQs.



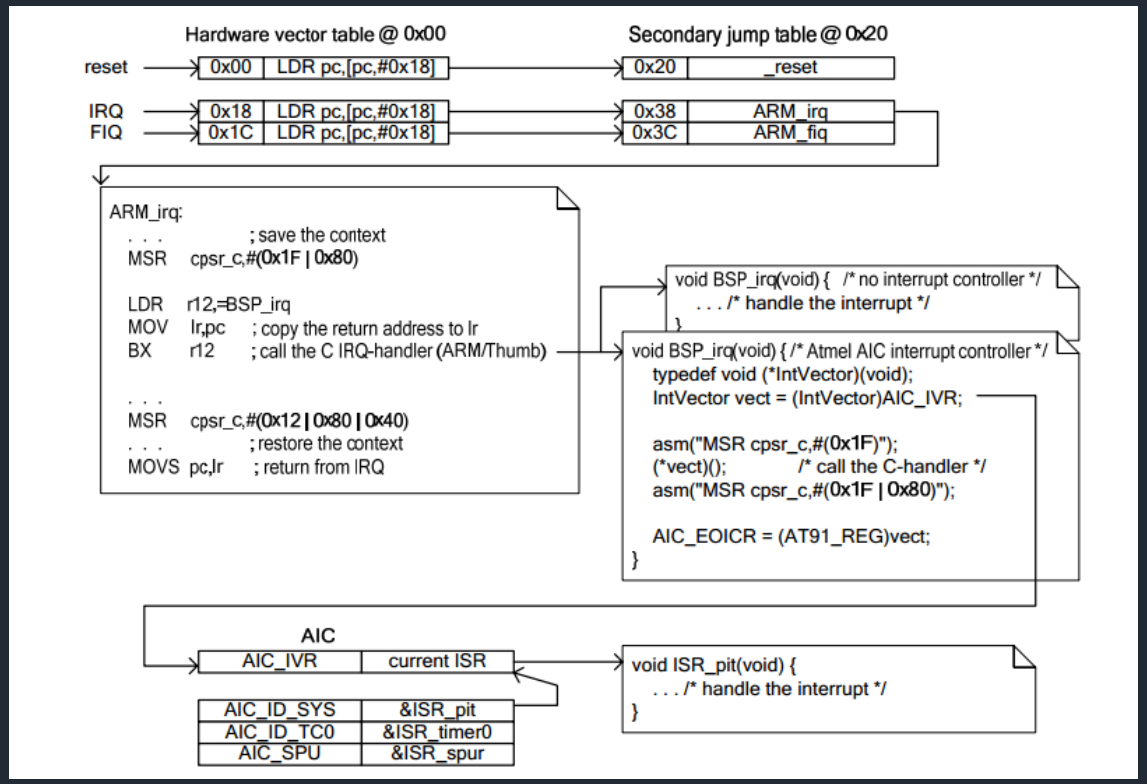
No ARM, o context nas operações de interrupções devem ser salvas pelo programador!

Isso torna o setup de interrupção do ARM determinístico.

Interrupção no ARM



A função BSP_irq() é utilizada para obter o endereço da ISR (Interrupt Service Routine) do controlador de interrupção e invocar a função ISR.

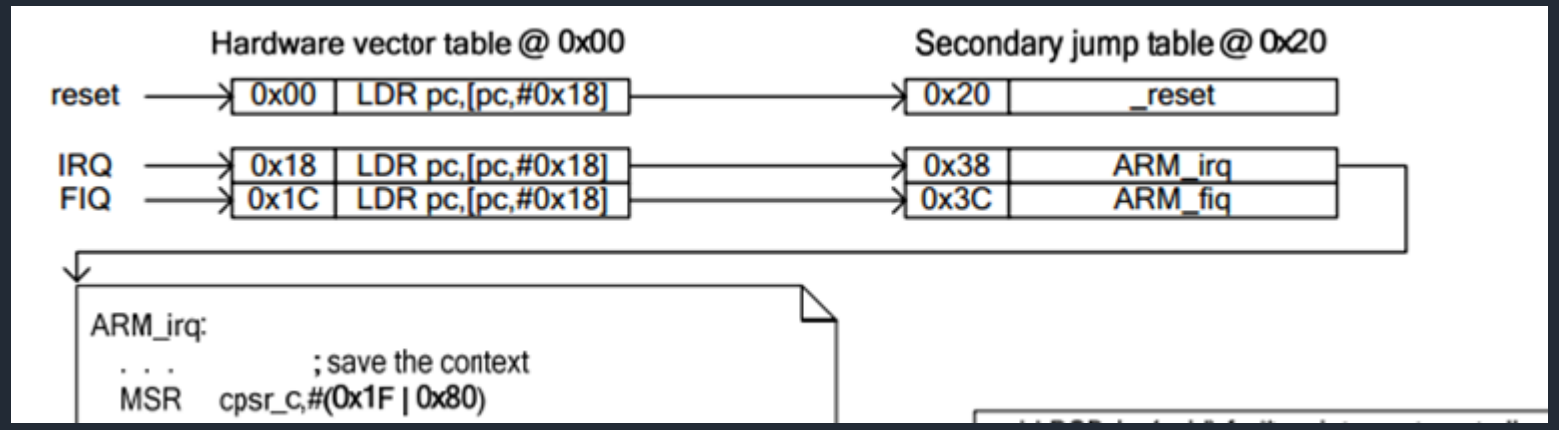


Quando é vetorado uma interrupção (#0x18 – tabela de vetores) o valor correspondente da tabela secundaria é carregada no PC (Contador de Programa) .

```
0x00:    LDR pc,[pc,#0x18]        /* Reset                  */
0x04:    LDR pc,[pc,#0x18]        /* Undefined Instruction  */
0x08:    LDR pc,[pc,#0x18]        /* Software Interrupt     */
0x0C:    LDR pc,[pc,#0x18]        /* Prefetch Abort         */
0x10:    LDR pc,[pc,#0x18]        /* Data Abort             */
0x14:    LDR pc,[pc,#0x18]        /* Reserved               */
0x18:    LDR pc,[pc,#0x18]        /* IRQ vector             */
0x1C:    LDR pc,[pc,#0x18]        /* FIQ vector             */
```

Exemplo: Reset

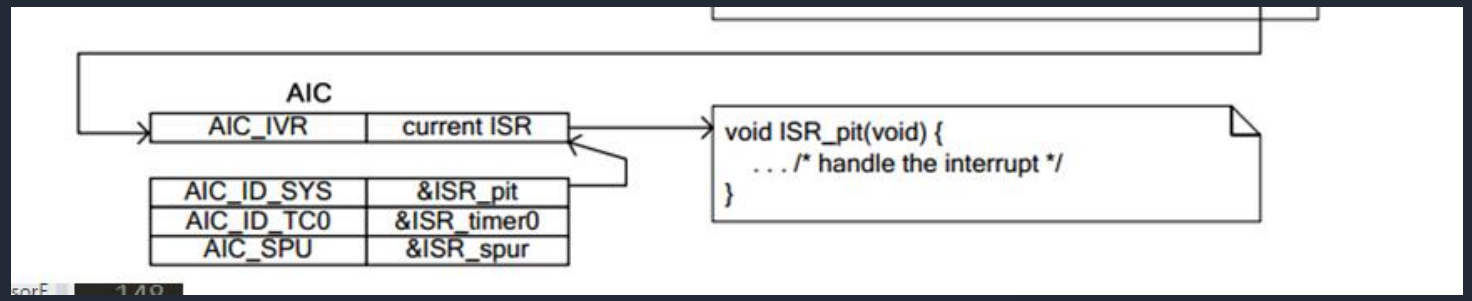
#0x18 + 0x00 (+8 pipeline) = 0x20 (endereço de memória seguinte carregado no PC)



1. **Por meio da tabela secundária é possível endereçar toda a memória (32-bits).**
2. **É possível remapear a tabela secundária a qualquer momento .**

```
ORG 0x18
LDR pc,[pc,#-0xF20]
```

0x20 – 0xF20 = 0xFFFFF100 (onde 0x20 é o valor do PC quando é executado a instrução de end. 0x18 é executado




```c
__attribute__ ((section(".isr_vector")))
void (* const g_pfnVectors[])(void) =
{
    (void *)&pulStack[STACK_SIZE],          // The initial stack pointer
    ResetHandler,                           // The reset handler
    NMIIntHandler,                          // The NMI handler
    HardFaultIntHandler,                    // The hard fault handler
    0, 0, 0, 0, 0, 0, 0,                    // Reserved
    SVCIntHandler,                          // SVCall handler
    0, 0,                                   // Reserved
    PendSVIntHandler,                       // The PendSV handler
    SysTickIntHandler,                      // The SysTick handler

    DMA0IntHandler,                         // DMA channel 0 transfer comp
                                            // and error handler
    DMA1IntHandler,                         // DMA channel 1 transfer comp
                                            // and error handler
    DMA2IntHandler,                         // DMA channel 2 transfer comp
                                            // and error handler
    DMA3IntHandler,                         // DMA channel 3 transfer comp
                                            // and error handler
    0,                                      // Reserved
    FTFAIntHandler,                         // Command complete and read collision
    LVDIntHandler,                          // Low-voltage detect, low-voltage warning
    LLWUIntHandler,                         // Low Leakage Wakeup
```

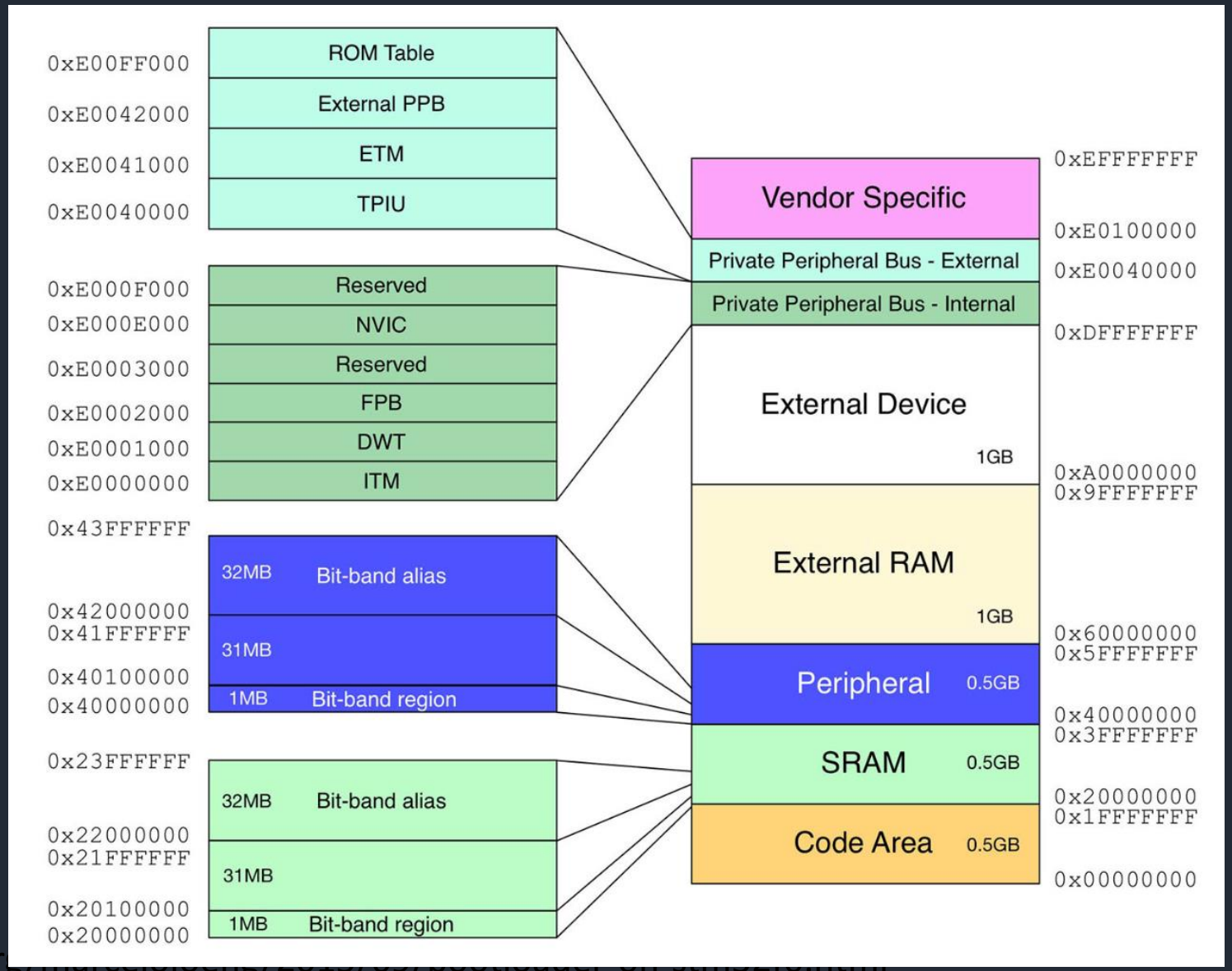
```c
typedef enum IRQn {
  /* Core interrupts */
  NonMaskableInt_IRQn          = -14,
  HardFault_IRQn               = -13,
  SVCall_IRQn                  = -5,
  PendSV_IRQn                  = -2,
  SysTick_IRQn                 = -1,

  /* Device specific interrupts */
  DMA0_IRQn                    = 0,
  DMA1_IRQn                    = 1,
  DMA2_IRQn                    = 2,
  DMA3_IRQn                    = 3,
  Reserved20_IRQn              = 4,
  FTFA_IRQn                    = 5,
  LVD_LVW_IRQn                 = 6,
  LLWU_IRQn                    = 7,
  I2C0_IRQn                    = 8,
  I2C1_IRQn                    = 9
```

O controlador de interrupção somente permite a preempção para outra IRQ se for de mais mais alta prioridade do que a IRQ corrente.

Sendo assim, uma ISR_timer0() não pode interromper ela mesma.

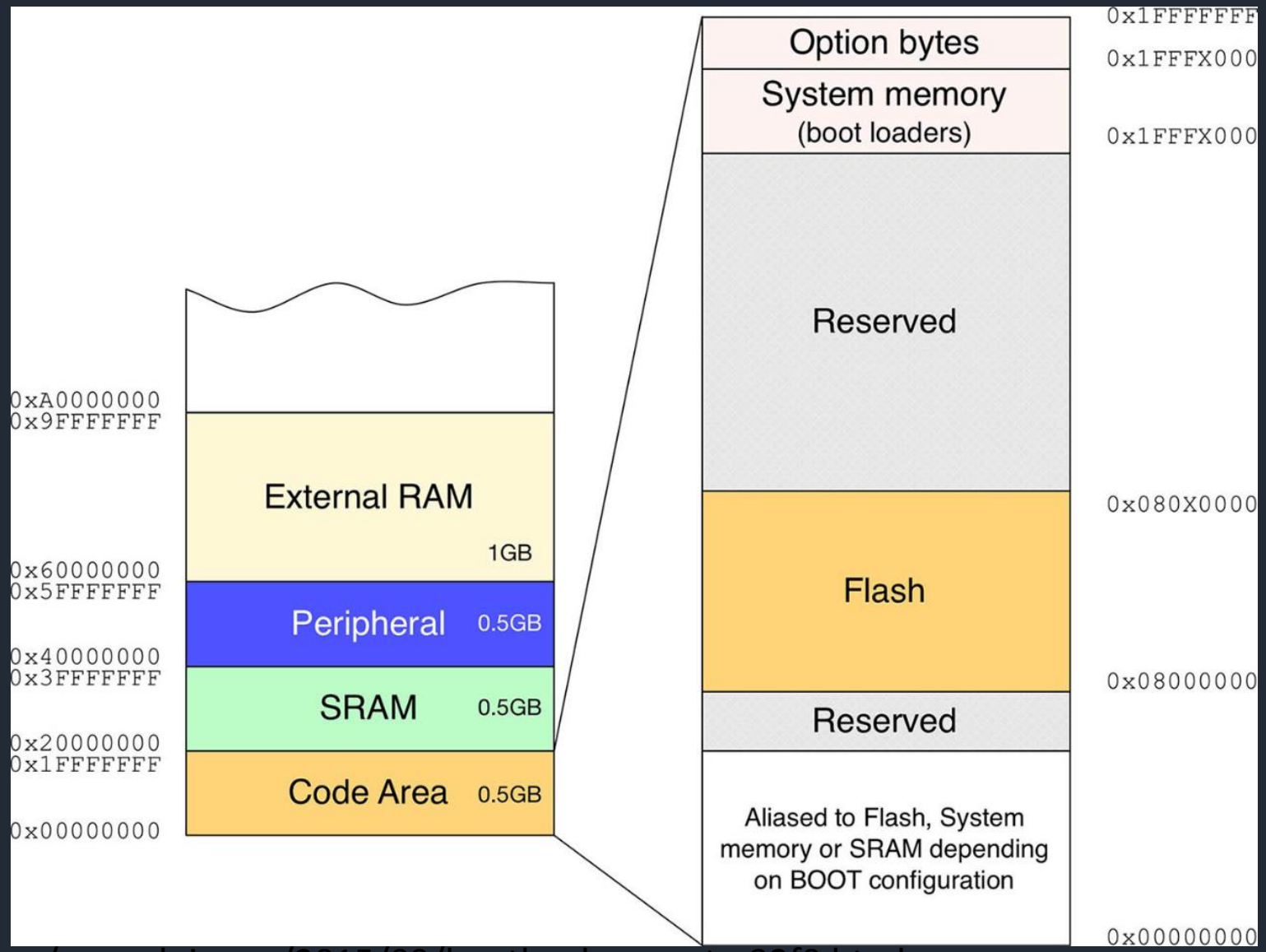
```
/*.....................................................................*/
void ISR_timer0(void) {                                        /* Timer0 ISR */
    uint32_t volatile dummy = AT91C_BASE_TC0->TC_SR;    /* clear int soruce */
    eventFlagSet(TIMER0_FLAG);                     /* set the TIMER0 event flag */

    (void)dummy;        /* suppress warning "dummy" was set but never used */
}
```

Memória



http://marcelojo.org/marcelojoeng/2013/09/bootloader-em-stm32f0.html

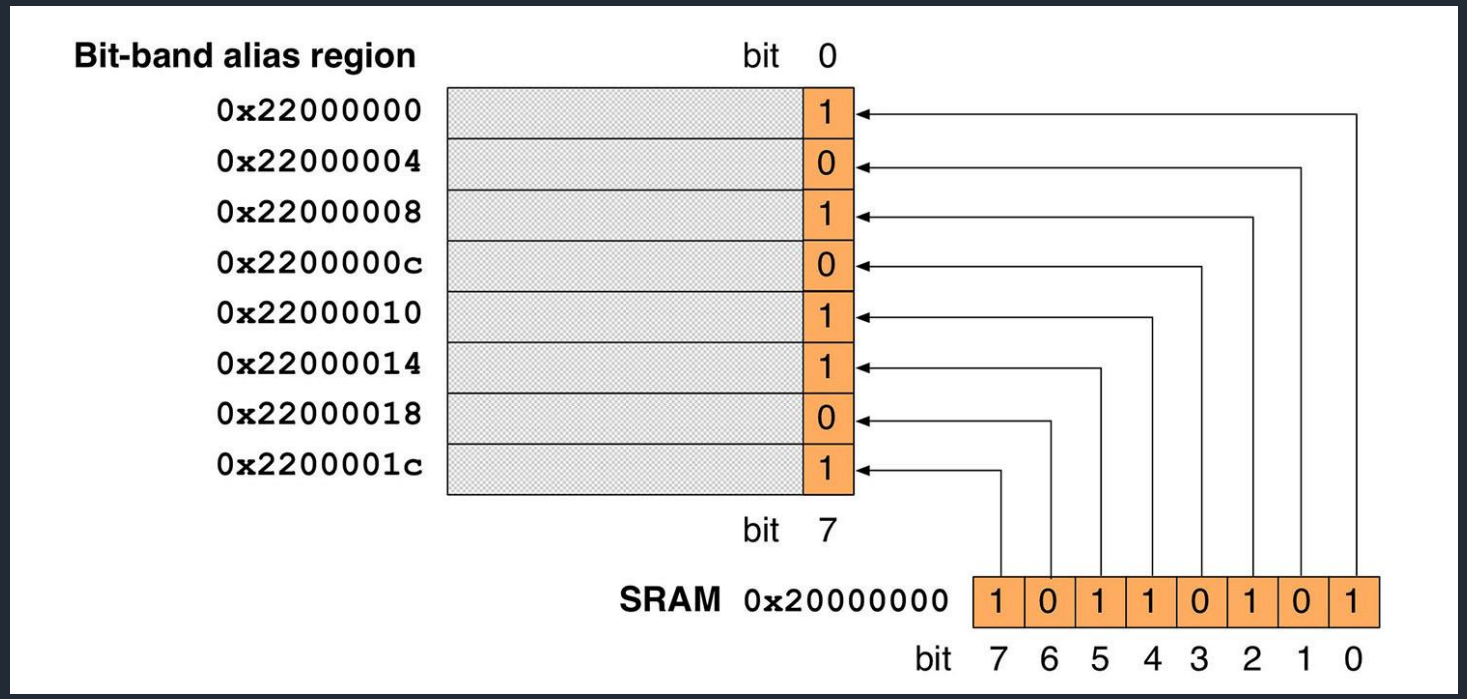
Memória



http://marcelojo.org/marcelojoeng/2015/09/bootloader-on-stm32f0.html

Memória

Bit-band é a capacidade de mapear cada bit de uma determinada área de memória para uma palavra inteira de maneira atomica.



mapeamento de memória do endereço SRAM 0x2000 0000 na região de faixas de bits (primeiros 8 de 32 bits mostrados)

http://marcelojo.org/marcelojoeng/2015/09/bootloader-on-stm32f0.html

Bootloader (exemplo)

Flash area	Flash memory addresses	Size (byte)	Name	Description
Main Flash memory	0x0800 0000 - 0x0800 07FF	2 Kbytes	Page 0	Sector 0
	0x0800 0800 - 0x0800 0FFF	2 Kbytes	Page 1	
	. . .	. . .	. . .	. . .
	0x0801 F000 - 0x0801 F7FF	2 Kbytes	Page 62	Sector 31[1]
	0x0801 F800 - 0x0801 FFFF	2 Kbytes	Page 63	
	. . .	. . .	. . .	. . .
	0x0803 F000 - 0x0803 F7FF	2 Kbytes	Page 126	-
	0x0803 F800 - 0x0803 FFFF	2 Kbytes	Page 127	-
Information block	0x1FFF C800 - 0x1FFF F7FF	12 Kbytes[2]	-	System memory
	0x1FFF D800 - 0x1FFF F7FF	8 Kbytes[3]	-	System memory
	0x1FFF F800 - 0x1FFF F80F	2 x 8 byte	-	Option byte

Exception number	IRQ number	Vector	Offset
16+n	n	IRQn	0x40+4n
. . .		. . .	. . .
18	2	IRQ2	0x48
17	1	IRQ1	0x44
16	0	IRQ0	0x40
15	-1	SysTick, if implemented	0x3C
14	-2	PendSV	0x38
13		Reserved	
12			
11	-5	SVCall	0x2C
10		Reserved	
9			
8			
7			
6			
5			
4			
3	-13	HardFault	0x10
2	-14	NMI	0x0C
1		Reset	0x08
		Initial SP value	0x04
			0x00

http://marcelojo.org/marcelojoeng/2015/09/bootloader-on-stm32f0.html

Bootloader (exemplo)

```
/* Memory definition */
MEMORY
{
  RAM (xrw)       : ORIGIN = 0x200000C0, LENGTH = 32K - 192  /* 192 for vector tab
  BOOTLOADER (rx) : ORIGIN = 0x08000000, LENGTH = 4K
  FIRMWARE (rx)   : ORIGIN = 0x08001000, LENGTH = 256K - 4K
}
/* Main app start address symbol */
_main_app_start_address = 0x08001000;

/* For main application, change BOOTLOADER by FIRMWARE */
REGION_ALIAS("ROM", BOOTLOADER );

/* Sections */
SECTIONS
{
  /* The startup code into ROM memory */
  .isr_vector :
  {
    . = ALIGN(4);
    KEEP(*(.isr_vector)) /* Startup code */
    . = ALIGN(4);
  } >ROM

  /* The program code and other data into ROM memory */
  .text :
  {
    . = ALIGN(4);
```

http://marc

Bootloader (exemplo)

```c
/**
 * Jump to application
 */
void JumptoApp(void)
{
    // disable global interrupt
    __disable_irq();

    // Disable all peripheral interrupts
    CPU_NVIC_DisableIRQ(SysTick_IRQn);
    CPU_NVIC_DisableIRQ(USART2_IRQn);

    CPU_NVIC_DisableIRQ(WWDG_IRQn);
    CPU_NVIC_DisableIRQ(RTC_IRQn);
    CPU_NVIC_DisableIRQ(FLASH_IRQn);
    CPU_NVIC_DisableIRQ(RCC_IRQn);
    CPU_NVIC_DisableIRQ(EXTI0_1_IRQn);
    CPU_NVIC_DisableIRQ(EXTI2_3_IRQn);
    CPU_NVIC_DisableIRQ(EXTI4_15_IRQn);
    CPU_NVIC_DisableIRQ(DMA1_Channel1_IRQn);
    CPU_NVIC_DisableIRQ(DMA1_Channel2_3_IRQn);
    CPU_NVIC_DisableIRQ(DMA1_Channel4_5_IRQn);
    CPU_NVIC_DisableIRQ(ADC1_IRQn);
    CPU_NVIC_DisableIRQ(TIM1_BRK_UP_TRG_COM_IRQn);
    CPU_NVIC_DisableIRQ(TIM1_CC_IRQn);
    CPU_NVIC_DisableIRQ(TIM3_IRQn);
    CPU_NVIC_DisableIRQ(TIM6_IRQn);
    CPU_NVIC_DisableIRQ(TIM7_IRQn);
    CPU_NVIC_DisableIRQ(TIM14_IRQn);
    CPU_NVIC_DisableIRQ(TIM15_IRQn);
```

```c
    CPU_NVIC_DisableIRQ(I2C2_IRQn);
    CPU_NVIC_DisableIRQ(SPI1_IRQn);
    CPU_NVIC_DisableIRQ(SPI2_IRQn);
    CPU_NVIC_DisableIRQ(USART1_IRQn);
    CPU_NVIC_DisableIRQ(USART2_IRQn);
    CPU_NVIC_DisableIRQ(USART3_6_IRQn);

    // main app start address defined in linker file
    extern uint32_t _main_app_start_address;

    uint32_t MemoryAddr = (uint32_t)&_main_app_start_address;
    uint32_t *pMem = (uint32_t *)MemoryAddr;

    // First address is the stack pointer initial value
    __set_MSP(*pMem); // Set stack pointer

    // Now get main app entry point address
    pMem++;
    void (*pMainApp)(void) = (void (*)(void))(*pMem);

    // Jump to main application (0x0800 0004)
    pMainApp();
}
```

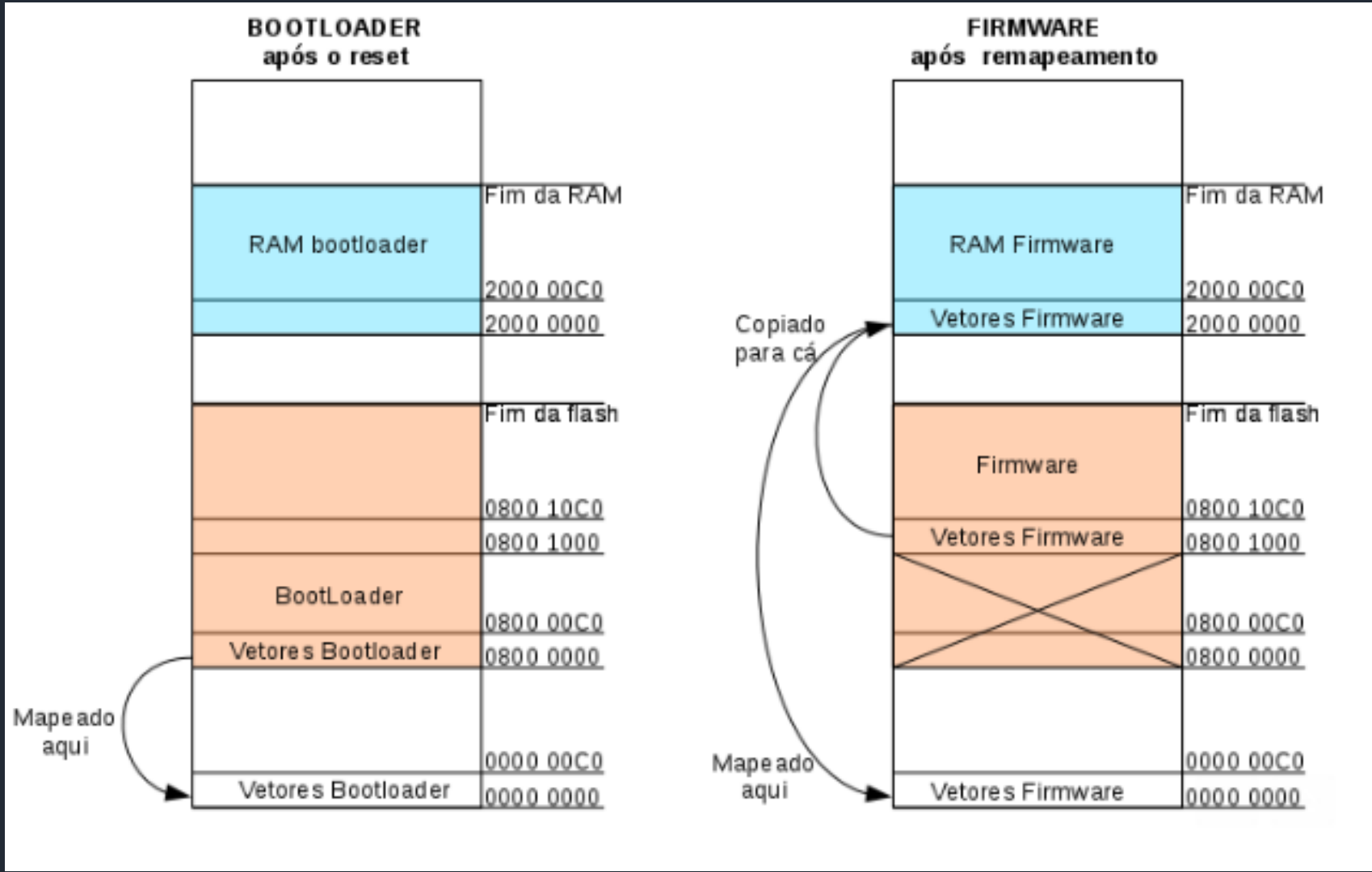
Bootloader (exemplo)

```c
#define FIRMWARE_START_ADDR                    (uint32_t)(&_main_app_start_address)
```

```c
void Remap_Table(void)
{
    // Copy interrupt vector table to the RAM.
    volatile uint32_t *VectorTable = (volatile uint32_t *)0x20000000;
    uint32_t ui32_VectorIndex = 0;

    for(ui32_VectorIndex = 0; ui32_VectorIndex < 48; ui32_VectorIndex++)
    {
        VectorTable[ui32_VectorIndex] = *(__IO uint32_t*)((uint32_t)FIRMWARE_START
    }

    __HAL_RCC_AHB_FORCE_RESET();

    //  Enable SYSCFG peripheral clock
    __HAL_RCC_SYSCFG_CLK_ENABLE();

    __HAL_RCC_AHB_RELEASE_RESET();

    // Remap RAM into 0x0000 0000
    __HAL_SYSCFG_REMAPMEMORY_SRAM();
}
```

http://marcelojo.org/marcelojoeng/2015/09/bootloader-on-stm32f0.html

Bootloader (exemplo)



http://marcelojo.org/marcelojoeng/2015/09/bootloader-on-stm32f0.html

Importantes links sobre o gcc++

https://stackoverflow.com/questions/15265295/understanding-the-libc-init-array
http://static.grumpycoder.net/pixel/uC-sdk-doc/initfini_8c_source.html

Obrigado