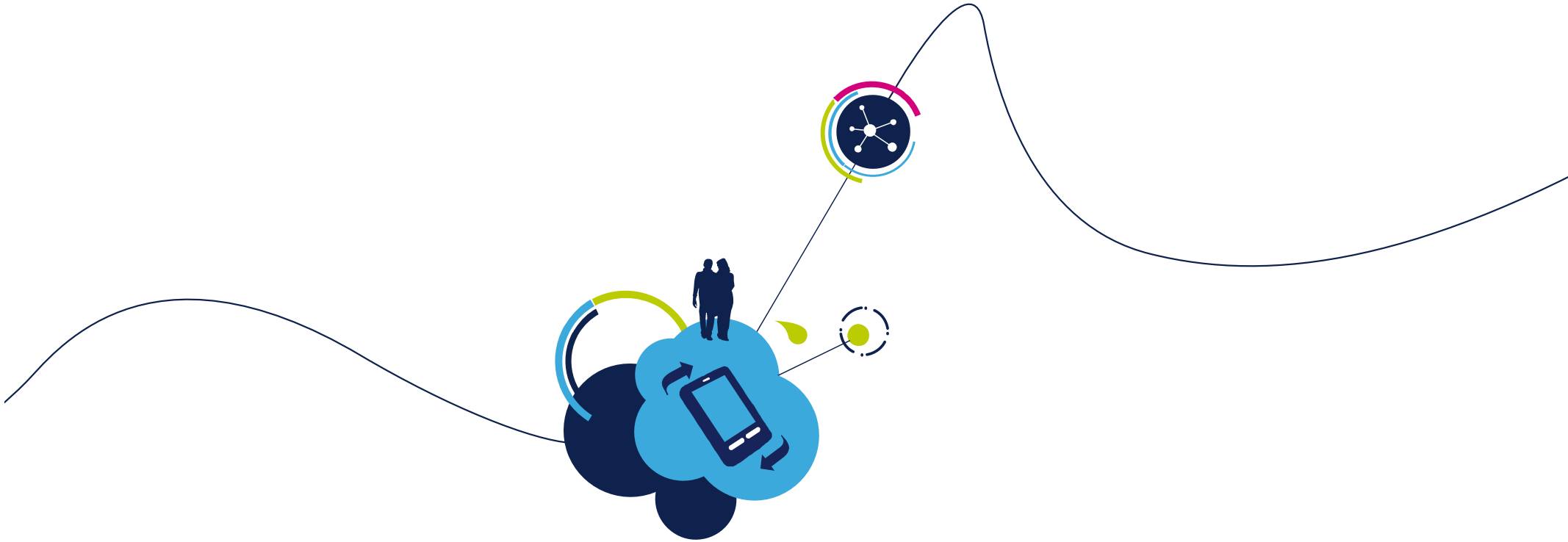


Cortex-M4 vs M3

life.augmented

CM4与CM3的区别

- 丰富的指令集
  - 单指令多数据的指令集（SIMD）
  - 扩展的单周期32位的乘法累加器（MAC）
  - 饱和运算指令
  - 单精度浮点运算指令
- 浮点运算单元（FPU）
- FPU寄存器
- 中断响应和返回

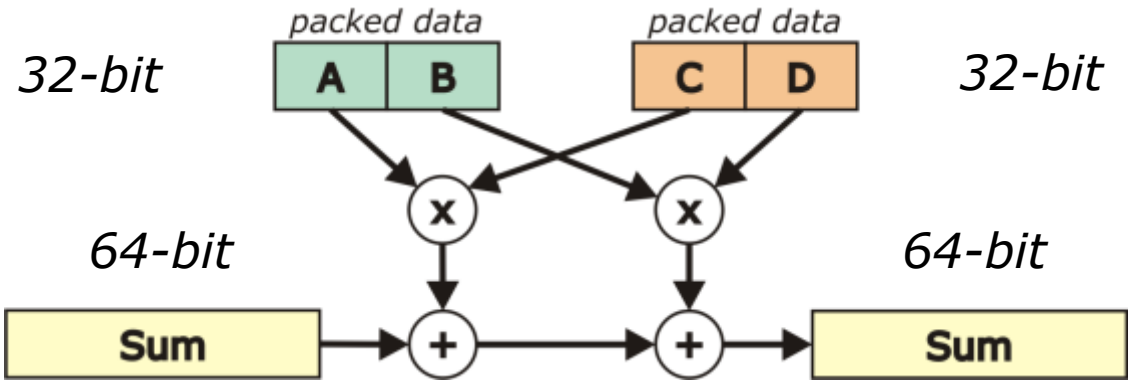
丰富的指令集

	Cortex-M3	Cortex-M4
浮点运算指令	---	Y
MAC	Y(多周期）	Y(单周期）
SIMD	---	Y
饱和指令	只有USAT,SSAT	Y

单周期SIMD指令

- SIMD(Single Instruction Multiple Data)
- 一次同时操作多个数据
- SIMD 在单周期内完成多次运算。见下面的例子：

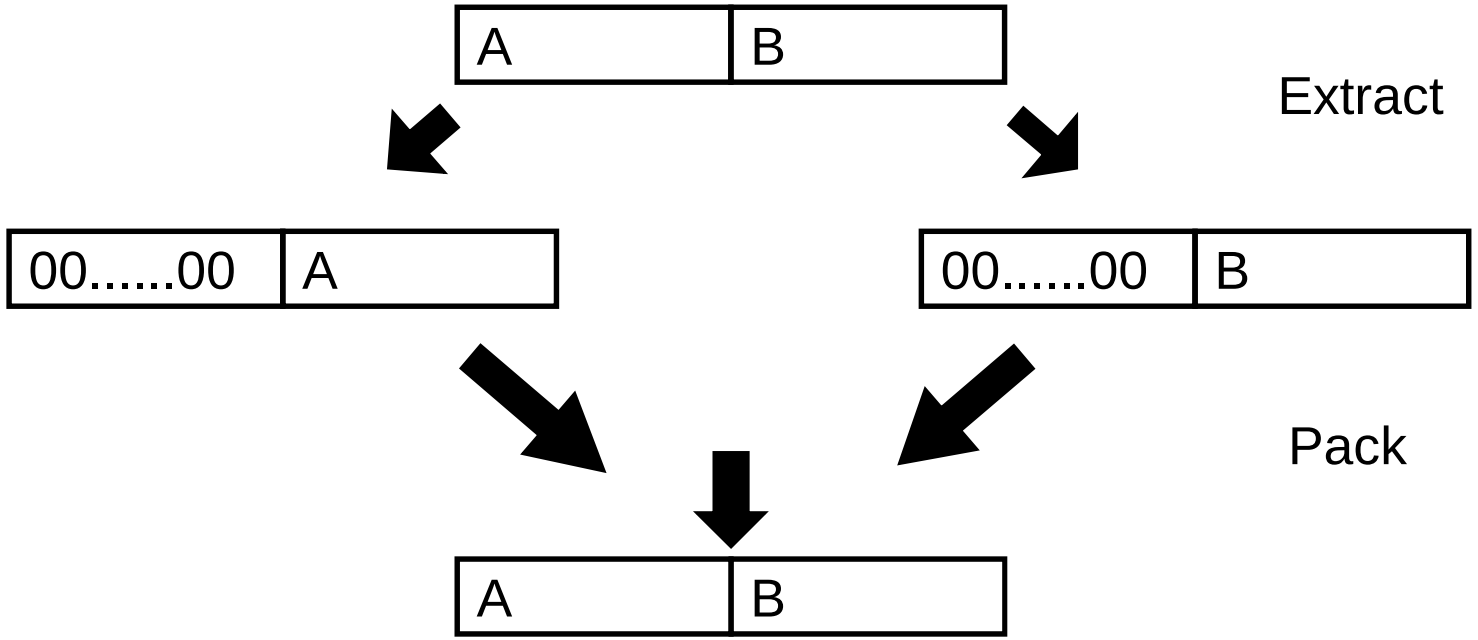
  Sum = Sum + (A x C) + (B x D)



- SIMD对打包数据（packed data)进行操作

打包数据类型(packed data)

- 四字节长度的数据或者两半字长度的数据打包成一个字长度的数据
- 访问打包数据结构更有效率
- SIMD指令可以操作打包数据类型
- 由指令对数据进行打包和释放



单周期MAC

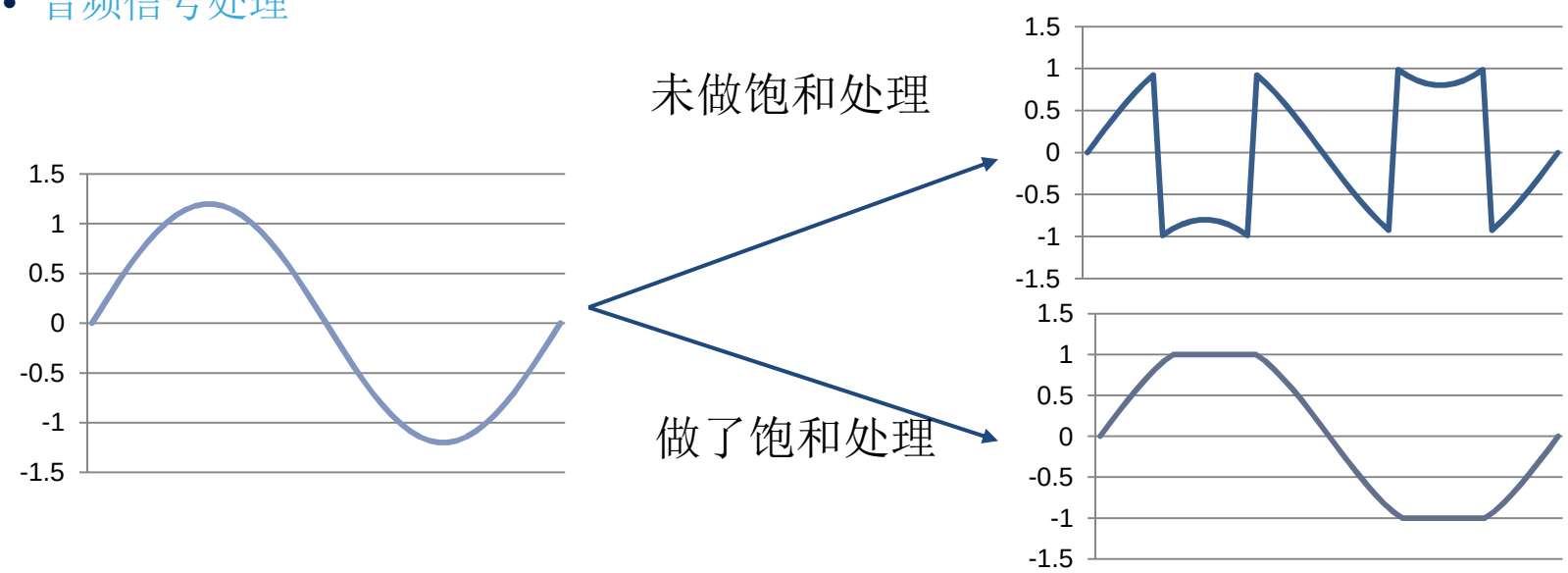
■以下所有指令在Cortex-M4中都是单周期执行

OPERATION	INSTRUCTIONS	CM3	CM4
16 x 16 = 32	SMULBB, SMULBT, SMULTB, SMULTT	n/a	1
16 x 16 + 32 = 32	SMLABB, SMLABT, SMLATB, SMLATT	n/a	1
16 x 16 + 64 = 64	SMLALBB, SMLALBT, SMLALTB, SMLALTT	n/a	1
16 x 32 = 32	SMULWB, SMULWT	n/a	1
(16 x 32) + 32 = 32	SMLAWB, SMLAWT	n/a	1
(16 x 16) ± (16 x 16) = 32	SMUAD, SMUADX, SMUSD, SMUSDX	n/a	1
(16 x 16) ± (16 x 16) + 32 = 32	SMLAD, SMLADX, SMLSD, SMLSDX	n/a	1
(16 x 16) ± (16 x 16) + 64 = 64	SMLALD, SMLALDX, SMLSLD, SMLSLDX	n/a	1
32 x 32 = 32	MUL	1	1
32 ± (32 x 32) = 32	MLA, MLS	2	1
32 x 32 = 64	SMULL, UMULL	5-7	1
(32 x 32) + 64 = 64	SMLAL, UMLAL	5-7	1
(32 x 32) + 32 + 32 = 64	UMAAL	n/a	1
32 ± (32 x 32) = 32 (upper)	SMMLA, SMMLAR, SMMLS, SMMLSR	n/a	1
(32 x 32) = 32 (upper)	SMMUL, SMMULR	n/a	1

饱和运算指令

- 使用饱和运算指令，指定饱和边界来防止变量溢出。不用通过软件来判断变量是否在有效范围内，减轻CPU的负担。

- 举例
  - 音频信号处理



  - 控制应用
    - PID控制中出现积分饱和时，通过饱和运算自动将值限制在允许范围内。

浮点运算指令

- 用于FPU单元的单精度浮点运算指令。这部分指令都是用V开头的汇编指令
- 只有在FPU开启的状态下，才能运行这些指令。
- 如果在FPU没有开启的状态下，执行了浮点运算指令，系统会产生一个硬fault异常

CM4与CM3的区别

- 丰富的指令集
  - 单指令多数据的指令集（SIMD）
  - 扩展的单周期32位的乘法累加器（MAC）
  - 饱和运算指令
  - 单精度浮点运算指令
- 浮点运算单元（FPU）
- FPU寄存器
- 中断响应和返回

浮点单元（FPU）

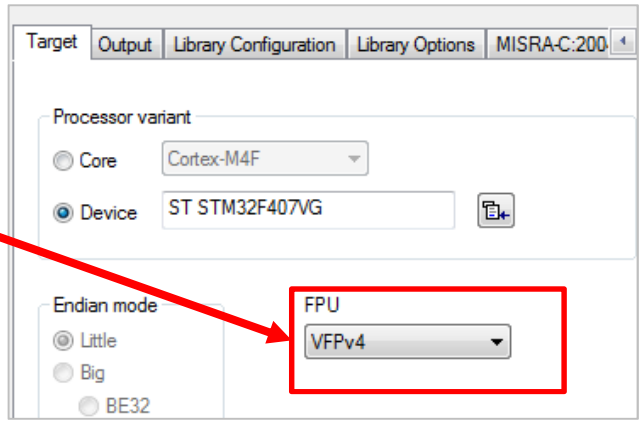
- FPU是什么
  - 独立于CPU的一个浮点运算单元，可以使能或关闭。
  - 支持单精度浮点数的运算：加、减、乘、除、乘加、平方根…
  - 整数、单精度浮点（32位）、半精度浮点（16位）之间的数据格式转换
  - Cortex-M4有一组专门用于FPU单元的单精度浮点运算指令。这部分指令都是用V-开头的汇编指令，仅在FPU功能被使能时使用。
- FPU怎么用
  - 芯片复位后FPU默认是关闭的。需要在CPACR寄存器中设置打开（将CPACR[23:20]设置为0xF）
  - 编译器也设置了相应的FPU功能开启/关闭的选项，在编译时需要告诉编译器是否开启FPU功能
  - 编译器一旦开启FPU功能，在处理单精度浮点运算的语句时就会用带V-开头的汇编指令进行编译
- 如果编译器使能了FPU功能，而芯片未开启FPU单元，程序运行到浮点语句时就会出现异常。相反，如果编译器未使能FPU功能，芯片即使开启了FPU单元，程序仍然会按照未开启FPU的代码执行。

life.augmented

IAR中如何配置FPU(1/2)

- 如果只需要进行简单的浮点运算（加减乘除等）

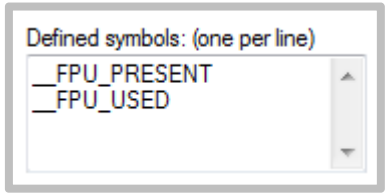
Step1: 在IAR中选择FPU，Option->General Options->Target->FPU: 选择“VFPv4”



Step2: 在system_stm32fxxx.c的SystemInit（）函数加入下面的代码。

```
/* FPU settings ------------------------------------------------------------*/
 #if (__FPU_PRESENT == 1) && (__FPU_USED == 1)
   SCB->CPACR |= ((3UL << 10*2)|(3UL << 11*2));  /* set CP10 and
CP11 Full Access */
 #endif
```

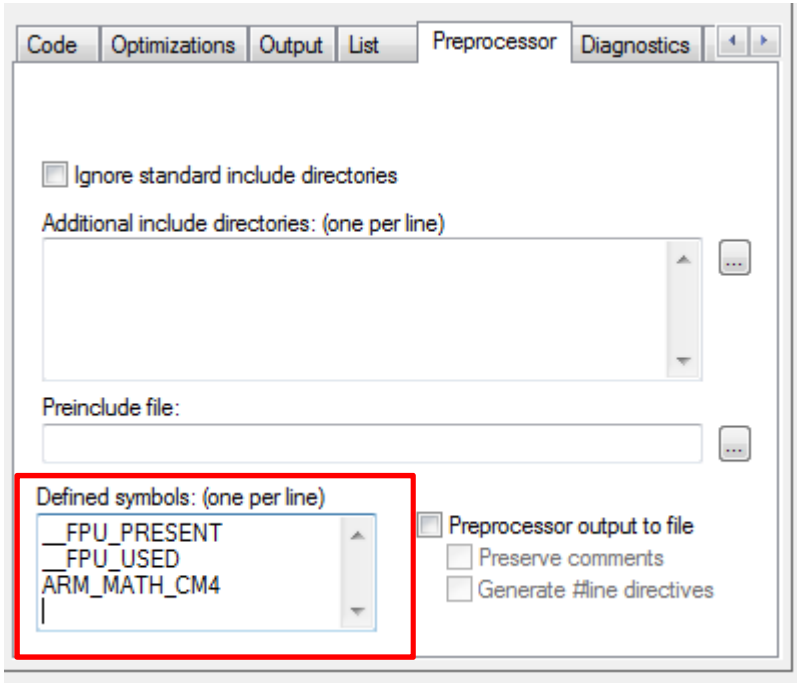
Step3: 在IAR的Option->C/C++ Compiler->Preprocessor选项卡的Defined symbols中加入：__FPU_PRESENT，__FPU_USED



life.augmented

IAR中如何配置FPU(2/2)

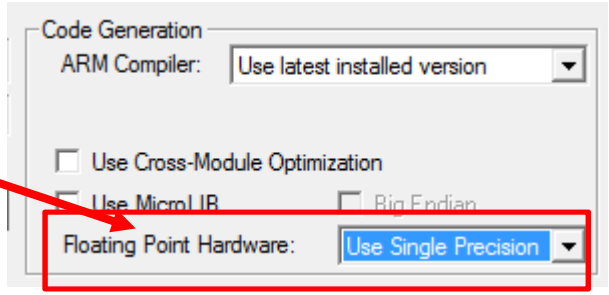
- 如果还需要进行复杂的浮点运算,比如开方，三角函数，则还需要做以下操作：
  - 检查是否已经包含arm_math.h 头文件。
  - Option->C/C++ Compiler->Preprocessor选项卡的Defined symbols中加入语句ARM_MATH_CM4



Keil中如何配置FPU(1/2)

- 如果只需要进行简单的浮点运算（加减乘除等）

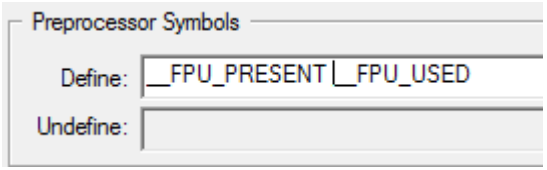
Step1:在Project的Options for target"xxxx"的Target选项卡的Code generation栏的Floating Point Hardware选择“Use single precision”



Step2: 在system_stm32fxxx.c的SystemInit（）函数加入下面的代码。

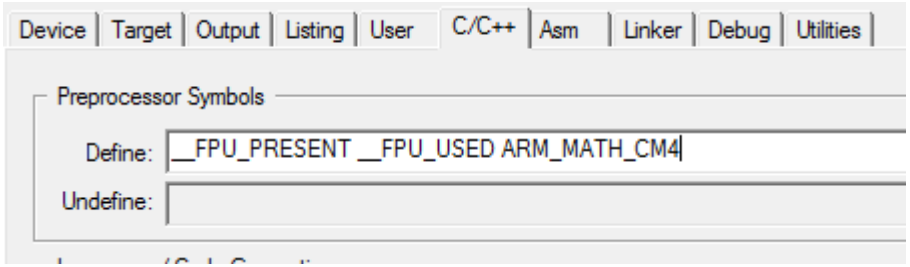
```
/* FPU settings ------------------------------------------------------------*/
 #if (__FPU_PRESENT == 1) && (__FPU_USED == 1)
   SCB->CPACR |= ((3UL << 10*2)|(3UL << 11*2));  /* set CP10 and
CP11 Full Access */
 #endif
```

Step3:在Project的Options for target"xxxx"的C/C++ 选项卡Preprocessor Symbols标签的Define栏中加入：__FPU_PRESENT，__FPU_USED



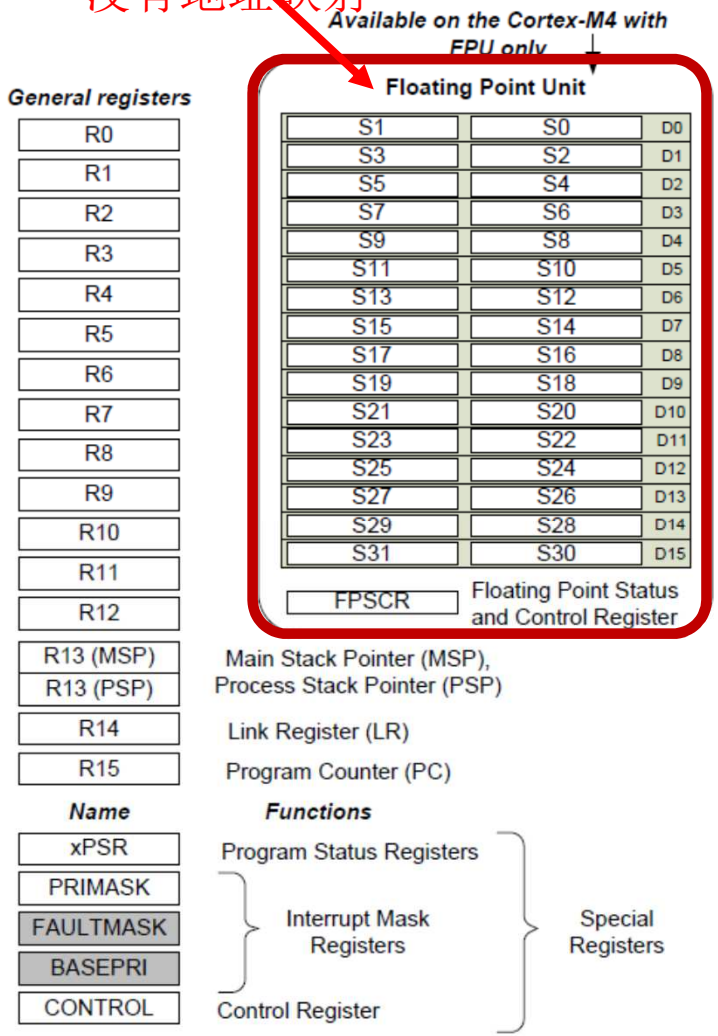
Keil中如何配置FPU(2/2)

- 如果还需要进行复杂的浮点运算,比如开方，三角函数，则还需要做以下操作：
  - 检查是否已经包含arm_math.h 头文件。
  - 在Project的Options for target”xxxx”的C/C++ 选项卡Preprocessor Symbols标签Define栏中加入语句ARM_MATH_CM4



FPU相关的寄存器

通过寄存器名字访问，
没有地址映射



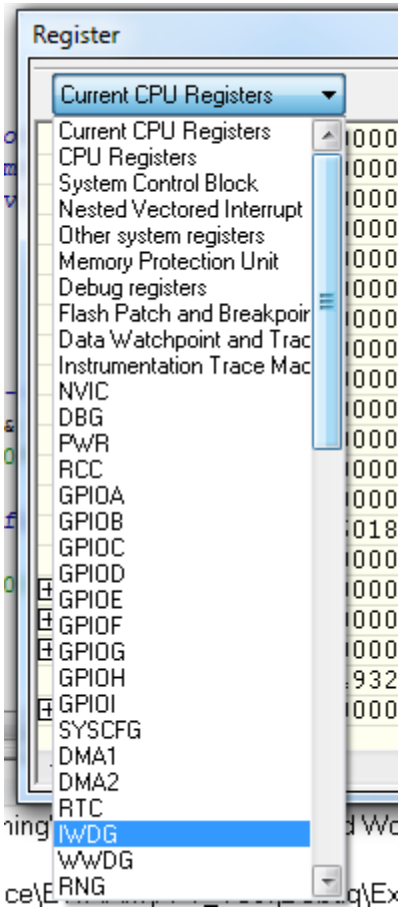
- 新增32个**32位（单精度）**的寄存器S0~S31，也可看做16个**64位（双精度）**的寄存器。
- 新增浮点状态和控制寄存器FPSCR
- 新增CPACR, FPCCR, FPCAR, FPDSCR
- 进行浮点运算时会用到这些寄存器

地址	名称
0xE000_ED88	CPACR
0xE000_EF34	FPCCR
0xE000_EF38	FPCAR
0xE000_EF3C	FPDSCR

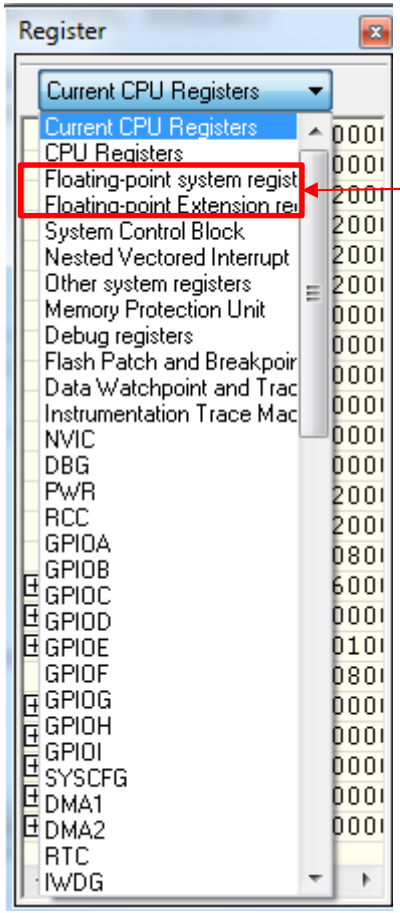
life.augmented

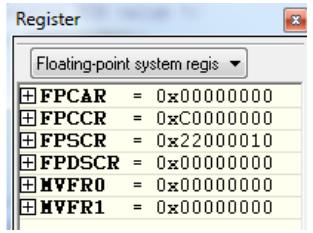
带FPU和不带FPU的寄存器对比

不带FPU

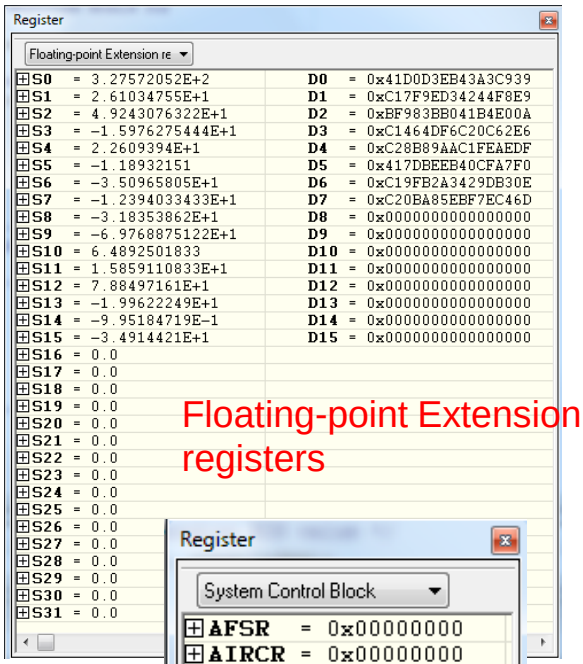


带FPU

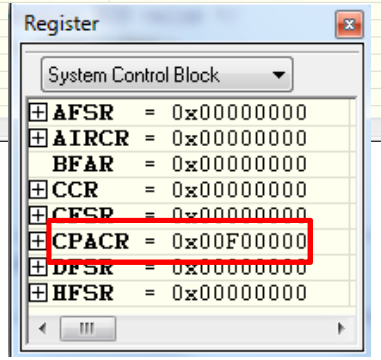




Floating-point System registers



Floating-point Extension registers



life.augmented

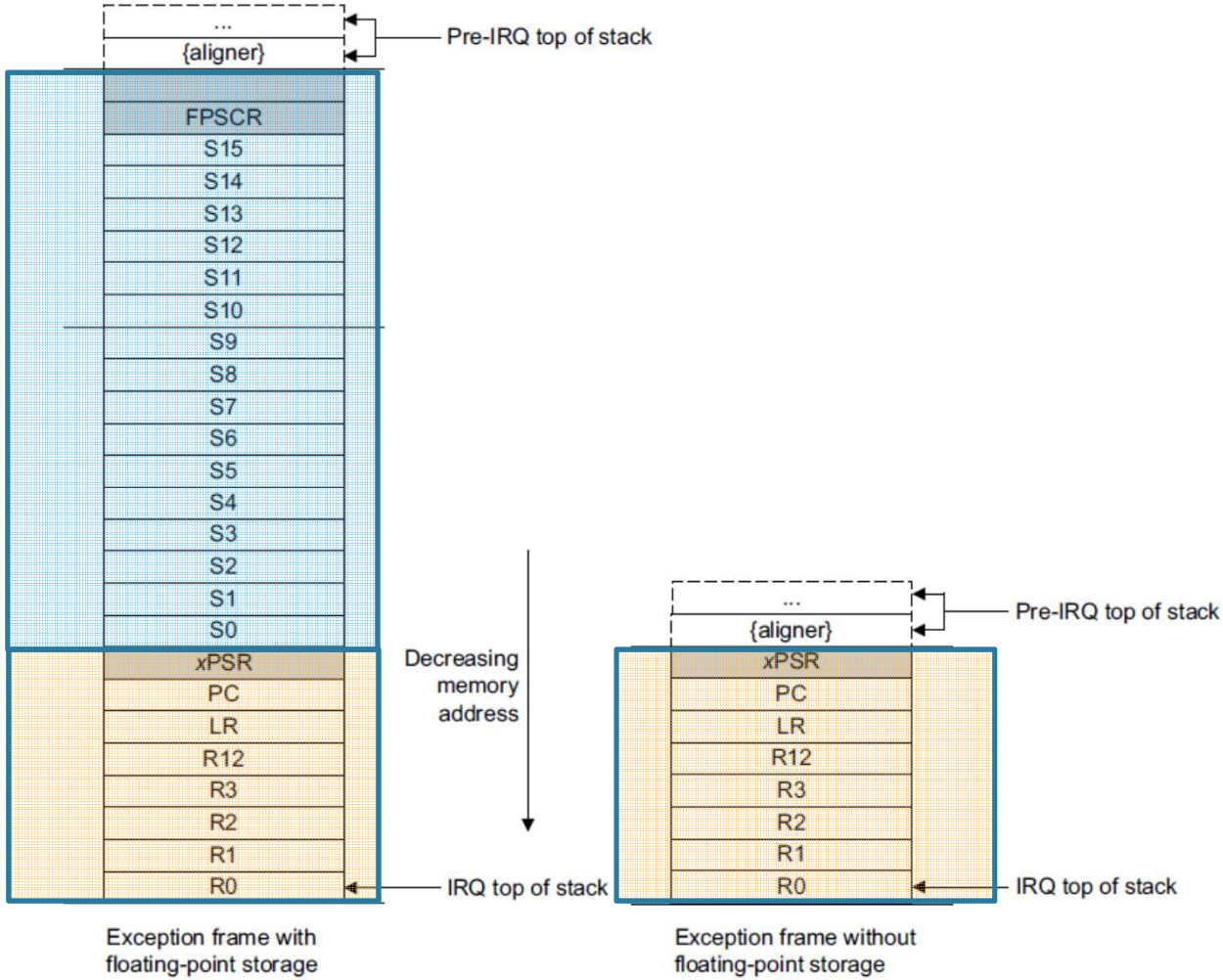
CM4与CM3的区别

- 丰富的指令集
  - 单指令多数据的指令集（SIMD）
  - 扩展的单周期32位的乘法累加器（MAC）
  - 饱和运算指令
  - 单精度浮点运算指令
- 浮点运算单元（FPU）
- FPU寄存器
- 中断响应和返回

中断响应和退出

- Cortex-M4的中断响应和退出机制和Cortex-M3大体相同
- 同样支持咬尾中断和晚到中断机制
- 因为多了对浮点运算的支持，在中断响应和退出时增加了对FPU扩展寄存器的保护
- 入栈浮点寄存器组会带来以下影响：
  - 会扩大stack frame所占的存储区域
  - 增加中断响应延迟
  - 在OS环境下增加上下文切换时间

两种Stack Frame



使用了
FPU

未使用
FPU

Cortex-M4的CONTROL 寄存器

FPCA只在CORTEX-M4中有效

位	名称	描述
[31:3]	----	保留
2	FPCA	0：没有用到FPU 1：用到过FPU Cortex-M4通过这一位的值来决定响应中断时是否要对浮点状态进行保护。
1	SPSEL	0：当前使用MSP指针 1：当前使用PSP指针
0	nPRIV	0：特权级的线程模式 1：非特权级的线程模式

Cortex-M4的EXC_RETURN值

EXC_RETURN位段的定义：

位段	含义
[31:5]	EXC_RETURN的标识，必须全为1
4	0 = 栈帧中预留了FPU寄存器的空间 1 = 栈帧不包括FPU寄存器，与Cortex-M3一样
3	0 = 返回后进入Handler模式 1 = 返回后进入线程模式
2	0 = 从主堆栈中做出栈操作，返回后使用MSP， 1 = 从进程堆栈中做出栈操作，返回后使用PSP
1	保留，必须为0
0	0 = 返回ARM状态 1 = 返回Thumb状态，在CM3中必须为1

EXC_RETURN的值

EXC_RETURN[31:0]	描述
0xFFFF_FFF1	使用MSP出栈，返回Handler模式，并使用MSP。不需要出栈浮点状态寄存器组。
0xFFFF_FFF9	使用MSP出栈，返回线程模式，并使用MSP。不需要出栈浮点状态寄存器组。
0xFFFF_FFFD	使用PSP出栈，返回线程模式，并使用PSP。不需要出栈浮点状态寄存器组。
0xFFFF_FFE1	使用MSP出栈，返回Handler模式，并使用MSP。需要出栈浮点状态寄存器组。
0xFFFF_FFE9	使用MSP出栈，返回线程模式，并使用MSP。需要出栈浮点状态寄存器组。
0xFFFF_FFED	使用PSP出栈，返回线程模式，并使用PSP。需要出栈浮点状态寄存器组。

Lazy Stacking

- Lazy stacking机制：在下面情况下，跳过对浮点扩展寄存器的入栈操作（仅预留空间）以避免中断延迟的增加
  - 中断处理函数不使用FPU
  - 被中断的程序未曾用到FPU
- 如果在执行中断处理函数时用到了FPU，在执行到第一条浮点指令时，内核暂停；硬件入栈浮点寄存器
- Lazy stacking是可以通过软件使能和关闭的。
  - 使能Lazy stacking 必须同时 置位FPCCR寄存器的LSPEN位和ASPEN位
  - 清除LSPEN位可以关闭Lazy stacking

FPU寄存器入栈策略的选择

FPCCR.LSPEN	FPCCR.ASPEN	描述
0	0	取消自动状态保存。中断响应时不入栈FPU寄存器。 应用场景： 1. 应用中没有用到OS或者多任务调度，如果没有任何中断异常用到FPU。 2. 在应用程序代码中只有一个中断用到FPU。如果有多个中断用到FPU，那么中断嵌套必须被禁止。可以通过把所有的中断优先级设置位相同优先级实现。
0	1	关闭Lazy stacking，仅打开自动状态保存。 如果用到FPU，CONTROL.FPCA位自动置1.中断响应时，硬件自动入栈S0~S15和FPSCR寄存器。
1	1	打开Lazy stacking,打开自动状态保存。 如果用到FPU，CONTROL.FPCA位自动置1.如果响应中断时，CONTROL.FPCA为1，处理器在堆栈中预留FPU寄存器的空间，同时将FPCCR.LSPACT位置1.但是PFU寄存器并没有马上入栈，直到在中断处理函数中用到FPU时再入栈。
1	0	非法配置

几个重要的标志位

浮点寄存器入栈/出栈过程中用到的标志位

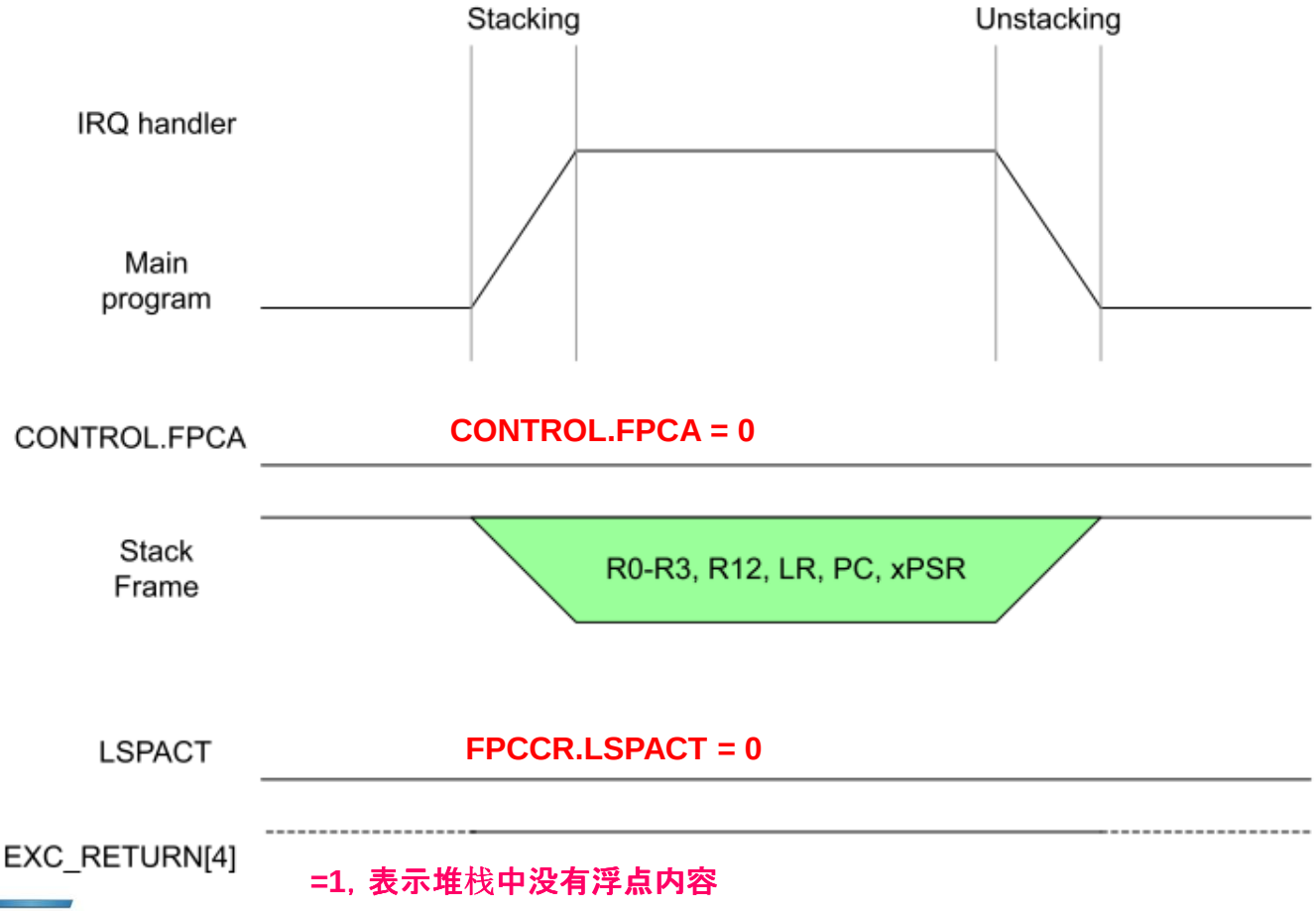
标志位	描述
CONTROL.FPCA	1 = 当前上下文中用到FPU 0 = 当前上下文中没有用到FPU
FPCCR.LSPACT	1 = 进入Lazy状态（栈帧中预留了FPU寄存器的空间，但没有实际入栈） 0 = 退出Lazy状态（实际入栈后被硬件清零或者中断返回时硬件清零）
EXC_RETURN[4]	0 = 栈帧中包括了FPU寄存器空间 1 = 栈帧中不包括FPU寄存器空间

FPCAR寄存器

位	名称	描述
[31:3]	ADDRESS	没有出栈的浮点寄存器空间在栈帧中的位置
[2:0]	----	保留

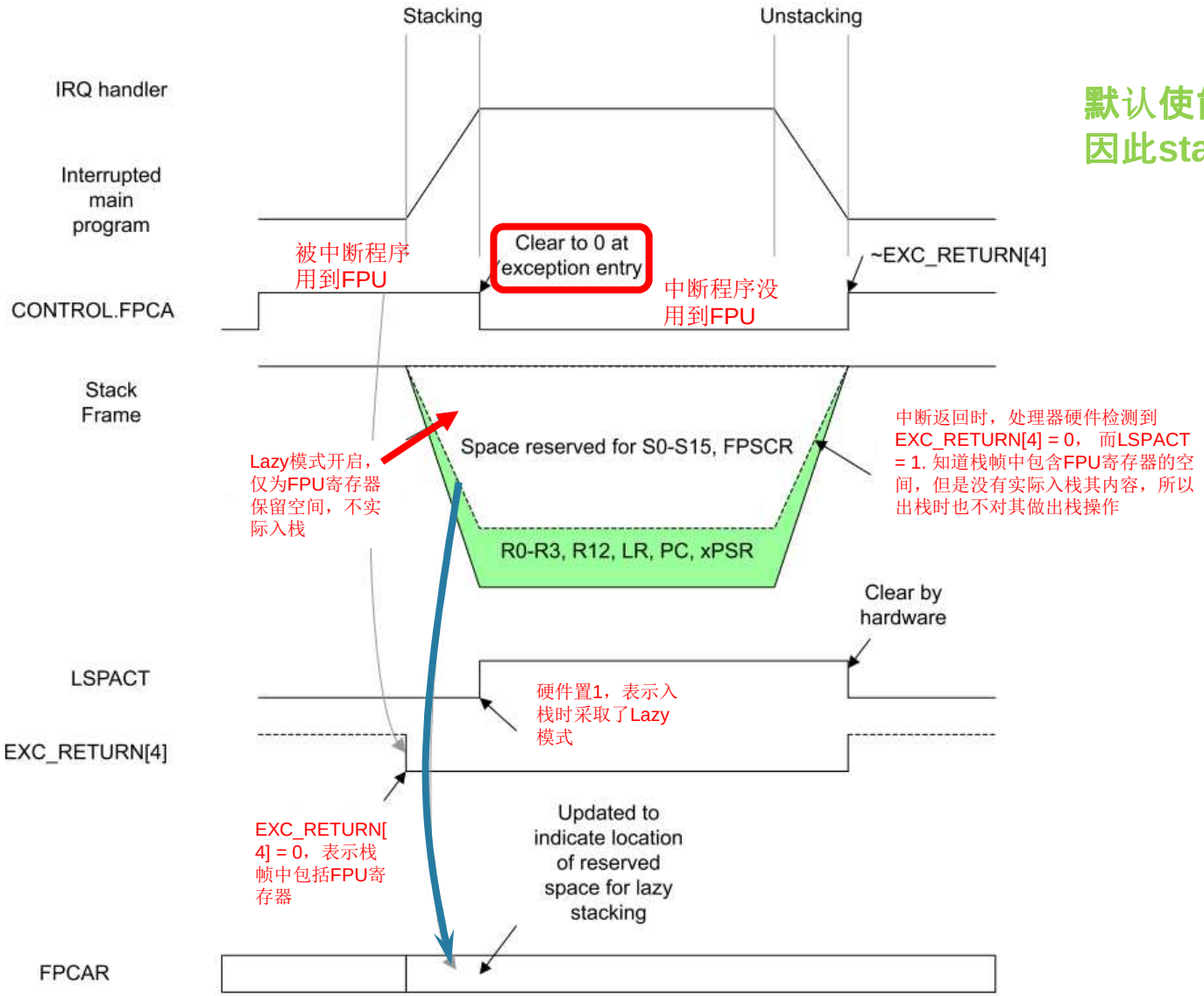
Lazy Stacking Case A
——被中断的程序和ISR都不用到FPU

- 堆栈内容和CM3一样
- 和没有实现FPU的CM4一样



life.augmented

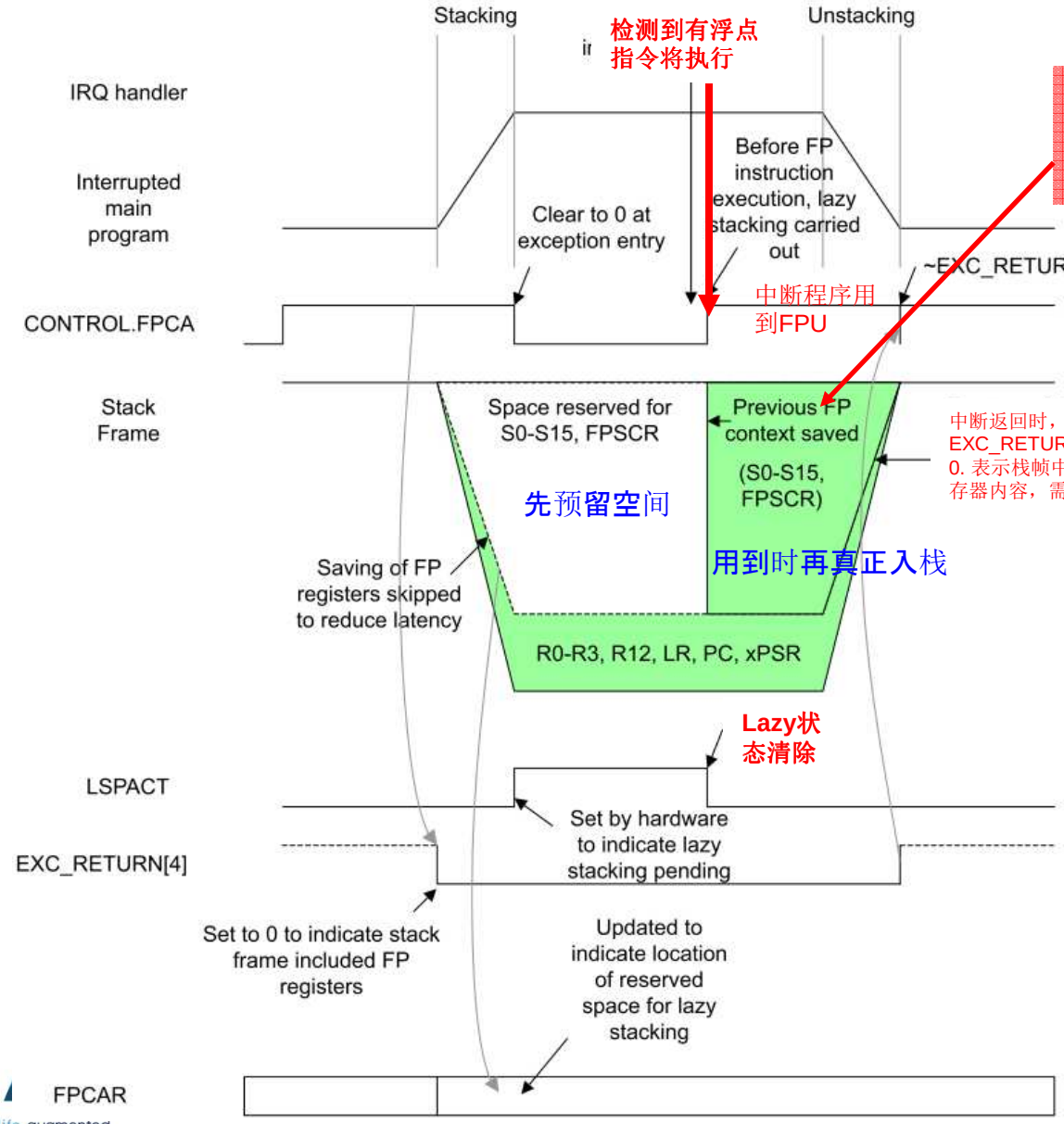
Lazy Stacking Case B
——被中断的程序用到浮点，ISR中没有



默认使能了Lazy Stacking模式，因此stack中有空间无内容

life.augmented

Lazy Stacking Case C
——被中断的程序和ISR都要用到FPU



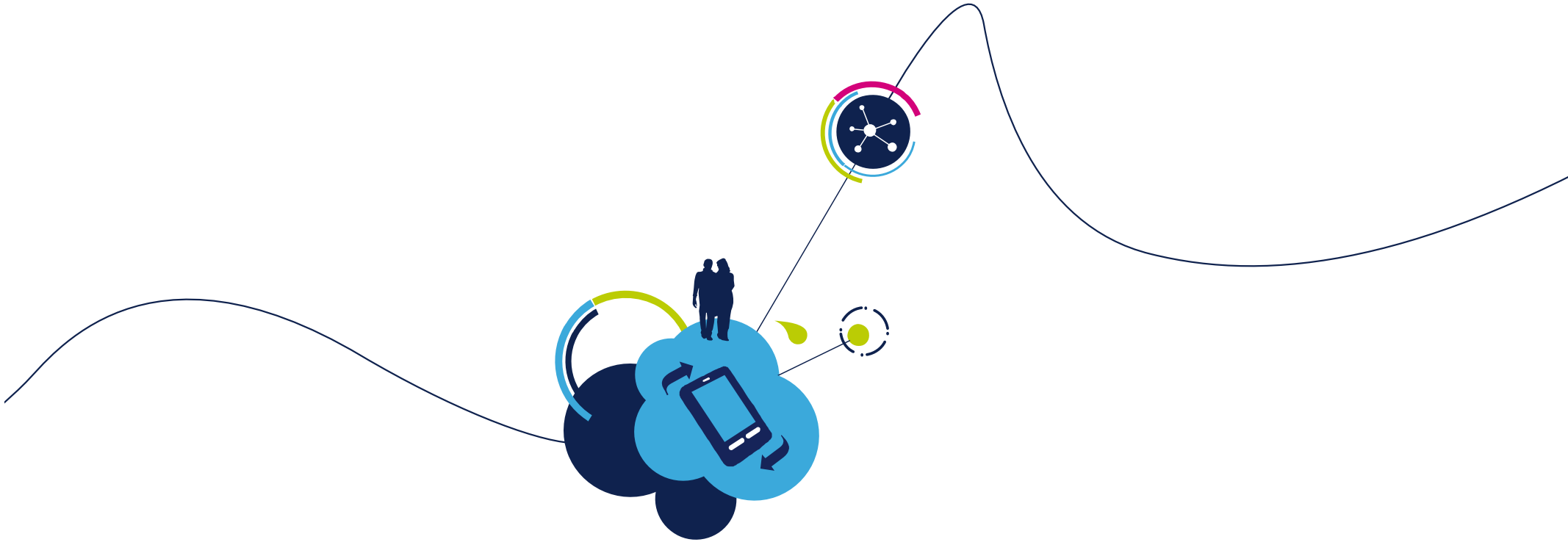
在执行浮点指令前，将未入栈的浮点寄存器入栈

CONTROL.2（FPCA）
中断外用FPU=1，
中断内先未用FPU=0，后用到FPU=1

中断返回时，处理器硬件检测到EXC_RETURN[4] = 0， LSPACT = 0. 表示栈帧中包含已经入栈的FPU寄存器内容，需要做相应出栈操作

LSPACT
刚入栈时采取了lazy模式，因此被硬件置1
执行到ISR的浮点指令时，需要浮点入栈，因此硬件清零

life.augmented



Cortex-M0 vs M3

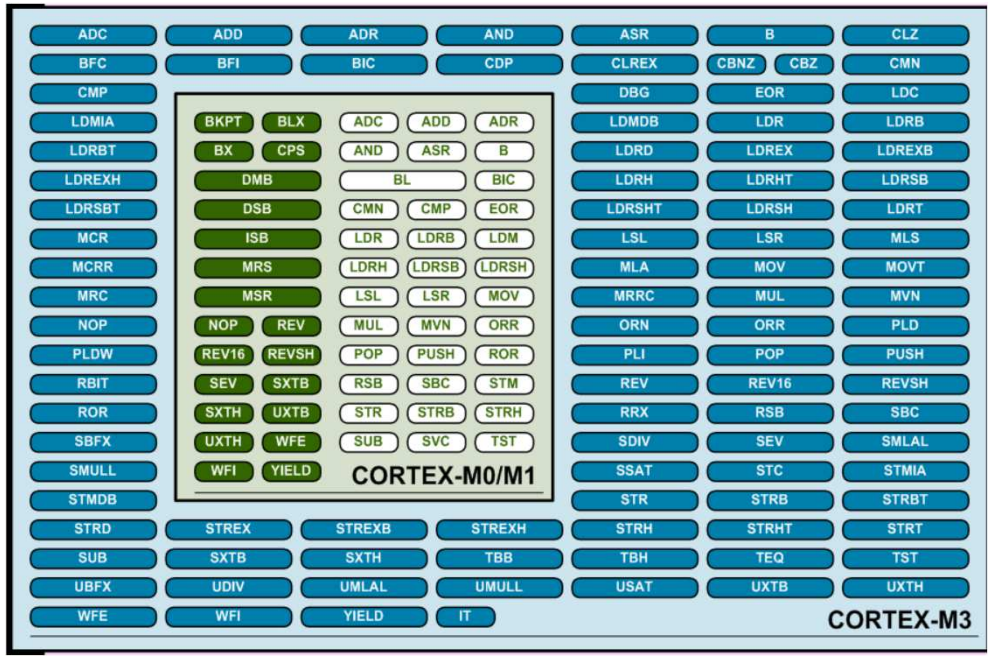
life.augmented

CM0与CM3的区别

- 更小的指令集
- 系统模式
- 中断与异常
- 存储器系统

更小的指令集

- Cortex-M0 只支持56条指令（完成简单的I/O控制任务和数据处理）
- 其中大部分是16-bit指令



	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q											
IPSR																Exception Number
EPSR						ICI/IT	T			ICI/IT						

M0中不可用

- xPSR寄存器中没有ICI/IT位
  （中断可持续指令位/IF-THEN指令状态位）

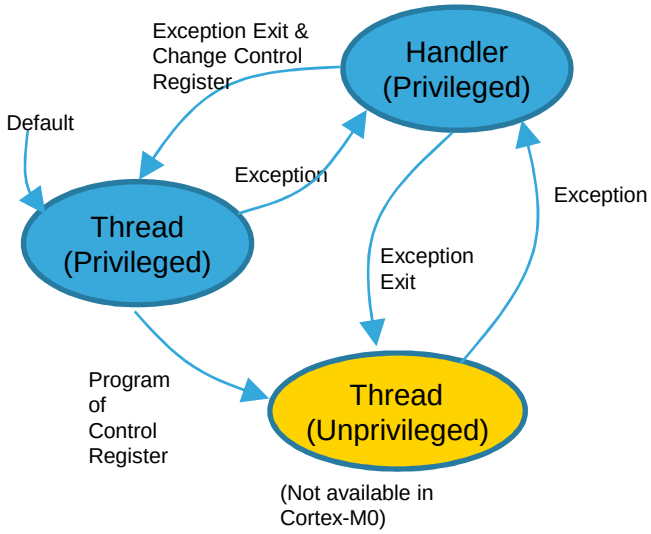
life.augmented

总线和系统模型

- Cortex-M0基于ARMv6-M架构,只有一条系统总线
- 没有定义非特权模式，对所有资源都可以访问。相应的CONTROL寄存器中BIT0也不再定义
- 两种操作模式，两个堆栈指针

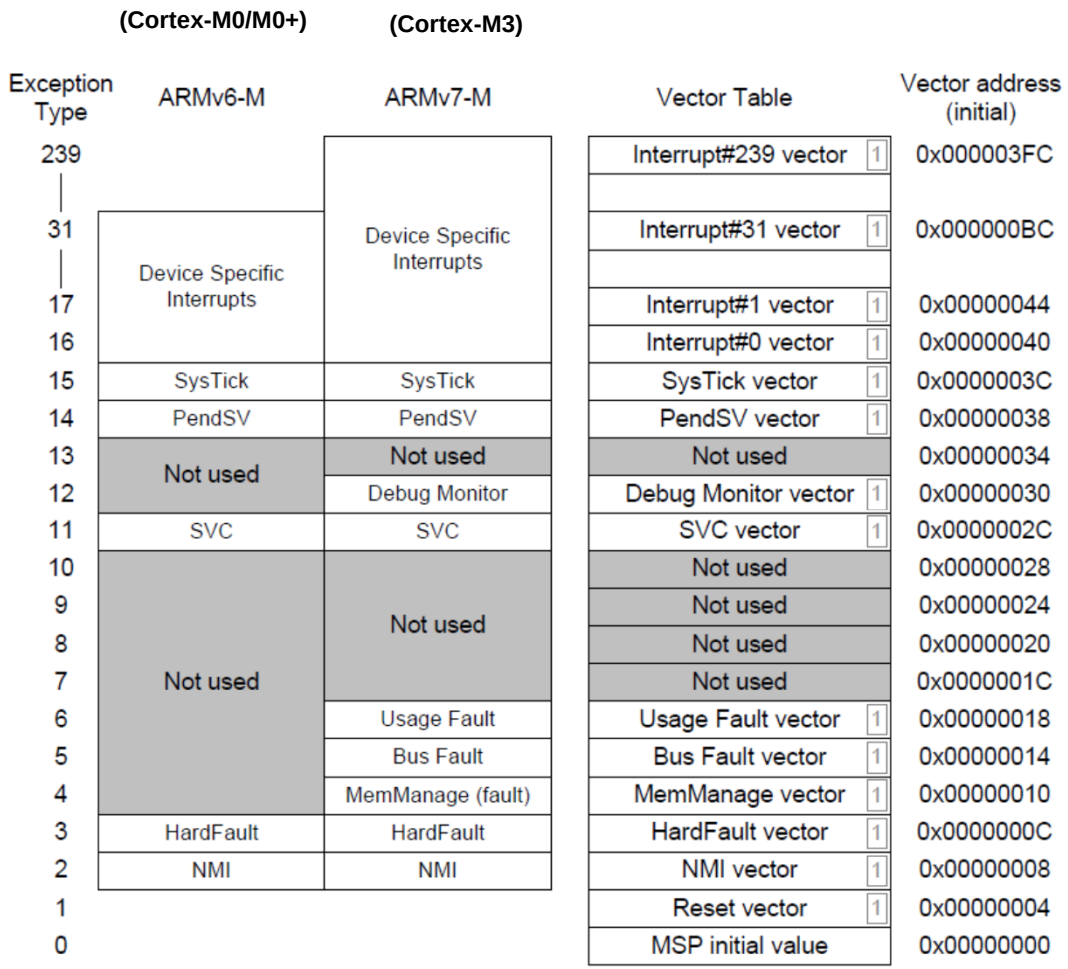
CONTROL寄存器

	31:3	2	1	0
Cortex-M3	reserve		SPSEL	**nPRIV**
Cortex-M0/M0+(ARMv6-M)	reserve		SPSEL	Not available



CM0与CM3的区别——异常和中断

- 仅支持6个系统异常和32个外部中断。
- 不支持存储器管理fault，总线fault，用法fault和调试监视器异常。所有的fault都由硬件fault异常进行处理

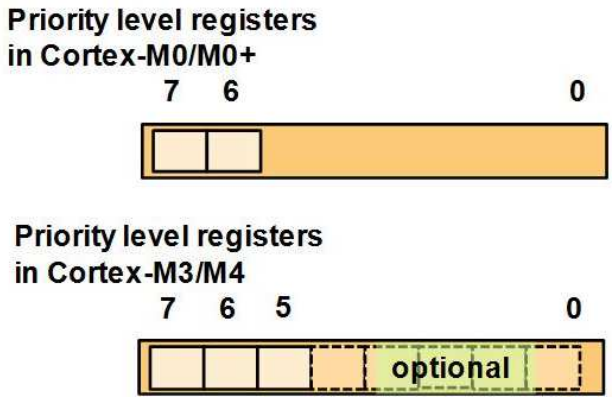


Exception Type	(Cortex-M0/M0+) ARMv6-M	(Cortex-M3) ARMv7-M	Vector Table	Vector address (initial)
239		Device Specific Interrupts	Interrupt#239 vector [1]	0x000003FC
31	Device Specific Interrupts		Interrupt#31 vector [1]	0x000000BC
17			Interrupt#1 vector [1]	0x00000044
16			Interrupt#0 vector [1]	0x00000040
15	SysTick	SysTick	SysTick vector [1]	0x0000003C
14	PendSV	PendSV	PendSV vector [1]	0x00000038
13	Not used	Not used	Not used	0x00000034
12		Debug Monitor	Debug Monitor vector [1]	0x00000030
11	SVC	SVC	SVC vector [1]	0x0000002C
10	Not used	Not used	Not used	0x00000028
9			Not used	0x00000024
8			Not used	0x00000020
7			Not used	0x0000001C
6		Usage Fault	Usage Fault vector [1]	0x00000018
5		Bus Fault	Bus Fault vector [1]	0x00000014
4		MemManage (fault)	MemManage vector [1]	0x00000010
3	HardFault	HardFault	HardFault vector [1]	0x0000000C
2	NMI	NMI	NMI vector [1]	0x00000008
1			Reset vector [1]	0x00000004
0			MSP initial value	0x00000000

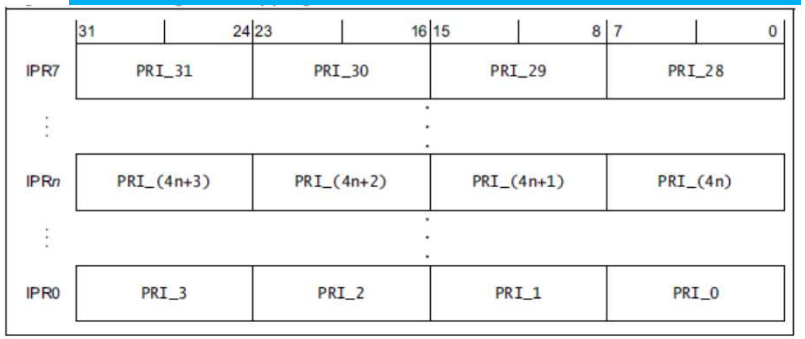
life.augmented

CM0与CM3的区别
——中断优先级

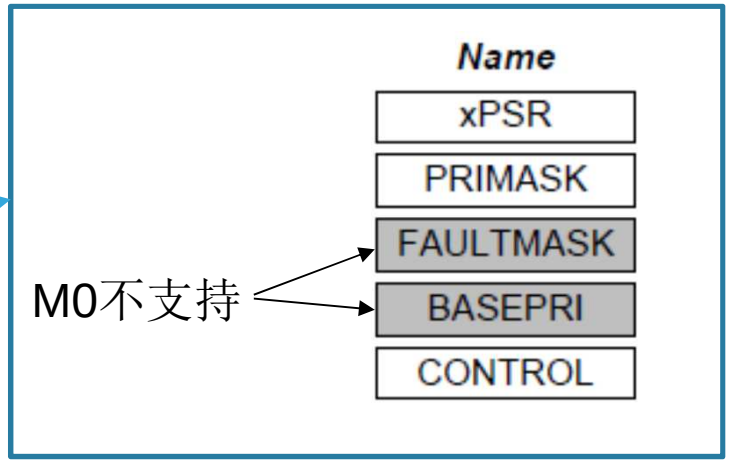
- 支持3个固定高优先级和4个可编程的优先级。
- 优先级配置寄存器为8位，但只能使用最高两位。也不分抢占优先级和亚优先级



STM32F0中断优先级寄存器，开始地址：0xE000_E400
（只能按字访问）

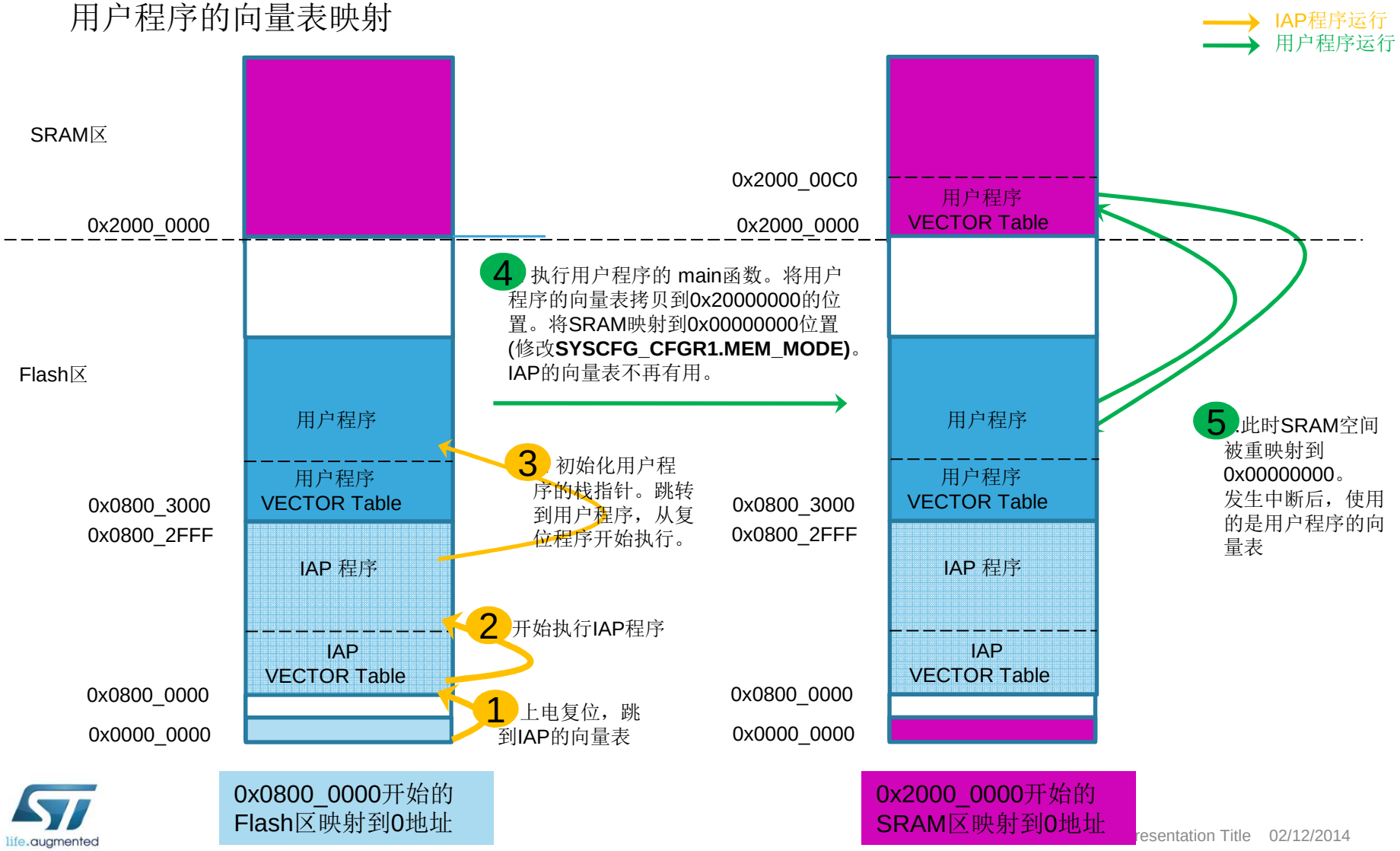


- 没有中断状态寄存器
- 没有软件触发中断寄存器
- 不支持动态修改优先级
- 优先级屏蔽
- NVIC寄存器只支持字访问



基于CM0的STM32芯片的IAP应用

- CM0不支持向量表重定位，使用STM32可以通过存储器重映射功能实现用户程序的向量表映射



STM32的IAP例程

判断是否需要更新程序

更新用户程序

跳转到用户程序；
初始化用户程序的栈指针

IAP的main函数

```
/* Test if Key push-button on STM320518-EVAL Board is pressed */
if (STM_EVAL_PBGetState(BUTTON_KEY) == 0x00)
{
  /* If Key is pressed, execute the IAP driver in order to re-program the Flash */
  IAP_Init();

  /* Display main menu */
  Main_Menu ();
}
else
{/* Keep the user application running */

  /* Test if user code is programmed starting from address "APPLICATION_ADDRESS" */
  if (((*(__IO uint32_t*)APPLICATION_ADDRESS) & 0x2FFE0000 ) == 0x20000000)
  {
    /* Jump to user application */
    JumpAddress = *(__IO uint32_t*) (APPLICATION_ADDRESS + 4);
    Jump_To_Application = (pFunction) JumpAddress;

    /* Initialize user application's Stack Pointer */
    __set_MSP(*(__IO uint32_t*) APPLICATION_ADDRESS);

    /* Jump to application */
    Jump_To_Application();
  }
}
```

将用户程序的向量表
拷贝到0x20000000

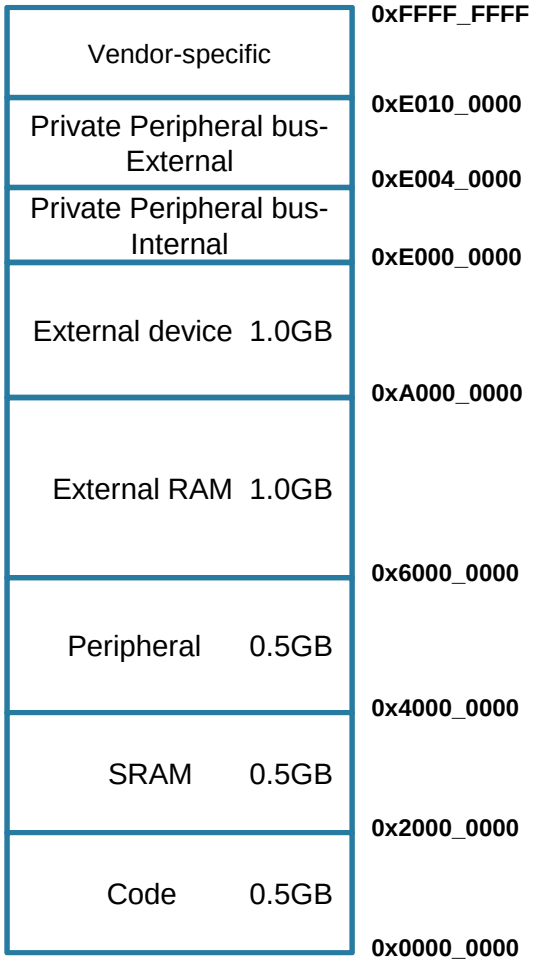
将SRAM映射到
0x00000000

用户程序的main函数

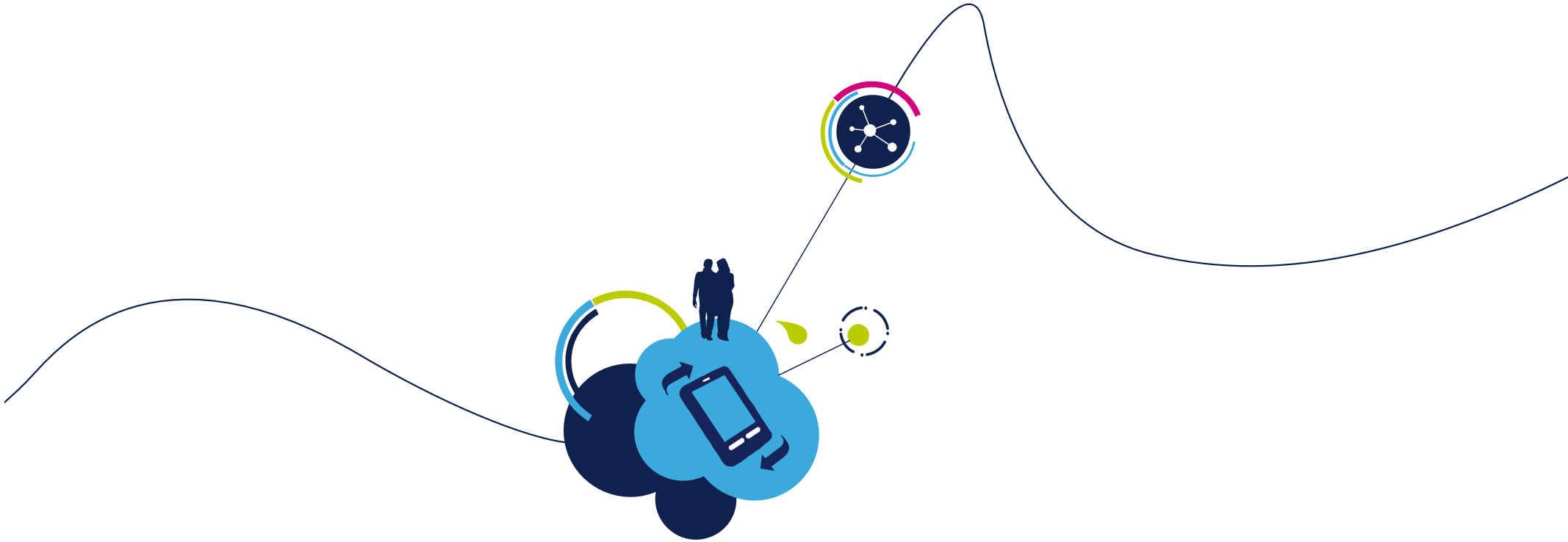
```
/*                                          the internal SRAM at 0x20000000 ***/
  #pragma location = 0x20000000
    __no_init __IO uint32_t VectorTable[48];
                                          apped at the base of the application
    load address 0x08003000) to the base address of the SRAM at 0x20000000. */
  for(i = 0; i < 48; i++)
  {
    VectorTable[i] = *(__IO uint32_t*)(APPLICATION_ADDRESS + (i<<2));
  }

  /* Enable the SYSCFG peripheral clock*/
  RCC_APB2PeriphResetCmd(RCC_APB2Periph_SYSCFG, ENABLE);
  /* Remap SRAM at 0x00000000 */
  SYSCFG_MemoryRemapConfig(SYSCFG_MemoryRemap_SRAM);
```

* 可参考AN4065

CM0与CM3的区别——存储器



- CM0同样支持4G的存储空间，存储器映射定义也相同
- 不支持Bit-banding
- 所有数据传输必须是对齐的
- 不支持MPU



Cortex-M0+ vs M3

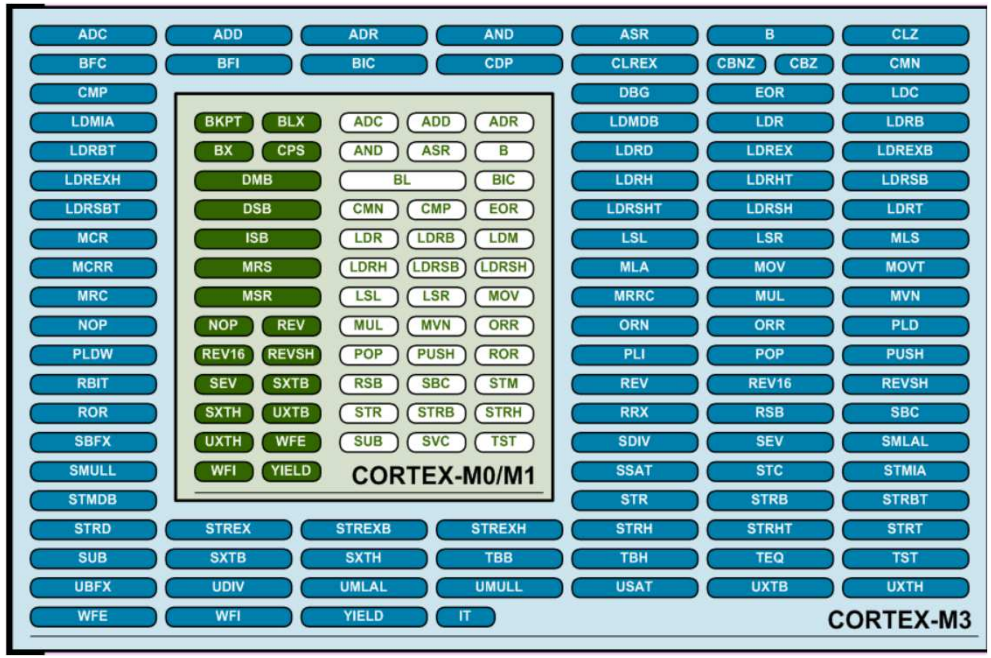
life.augmented

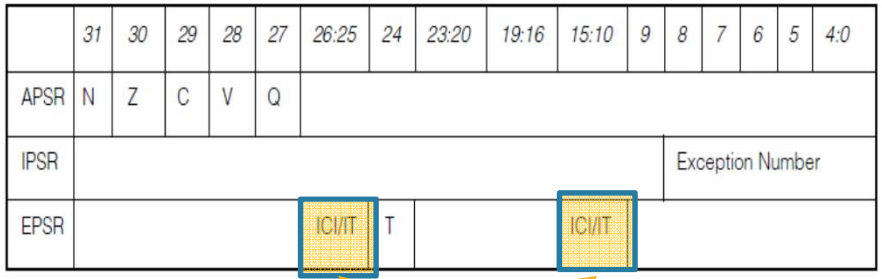
CM0+ 与CM3的对比

- 更小的指令集
- 一条总线/两级流水线/两个特权级别
- 单周期I/O接口
- 存储器系统
- 中断与异常
- 低功耗模式

更小的指令集

- Cortex-M0+和Cortex-M0一样只支持56条指令（完成简单的I/O控制任务和数据处理）
- 其中大部分是16-bit指令





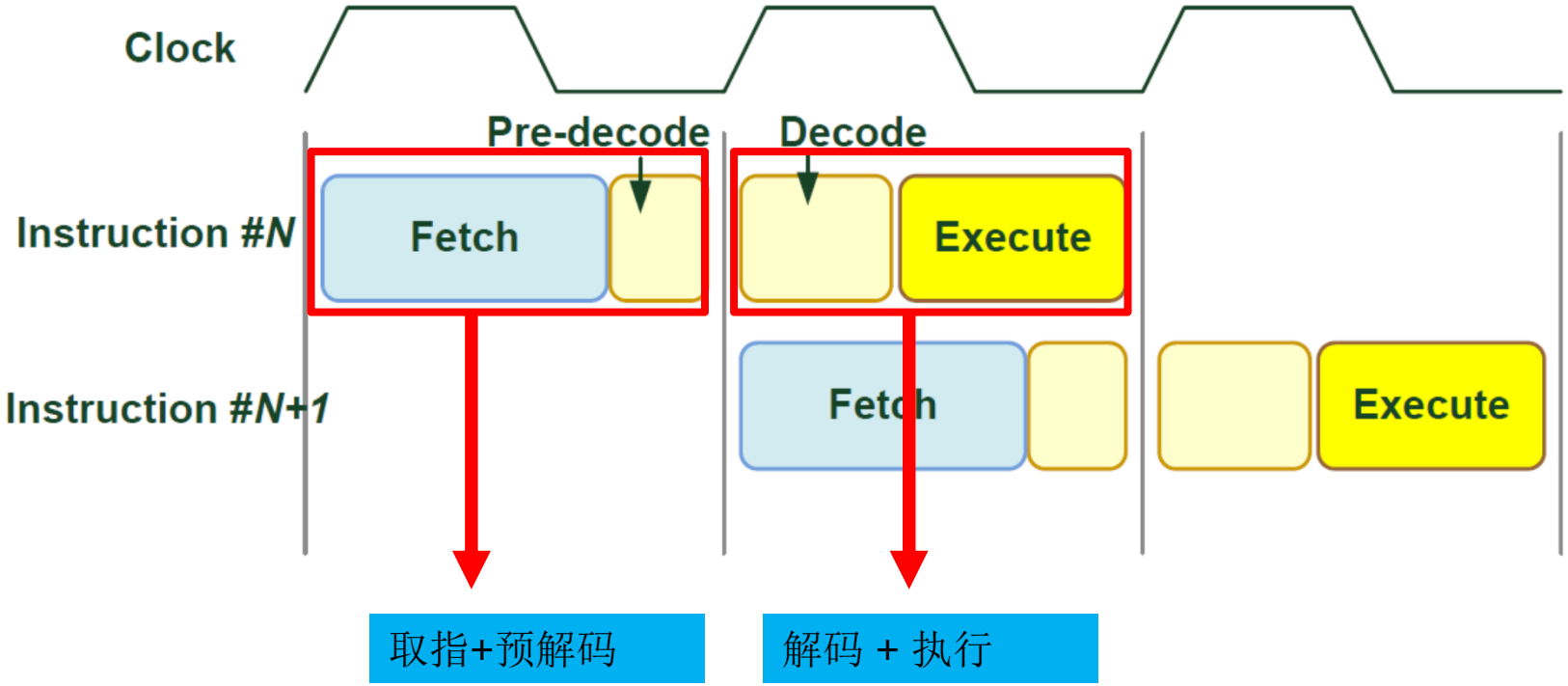
M0+中不可用

- xPSR寄存器中没有ICI/IT位
  （中断可持续指令位/IF-THEN指令状态位）

life.augmented

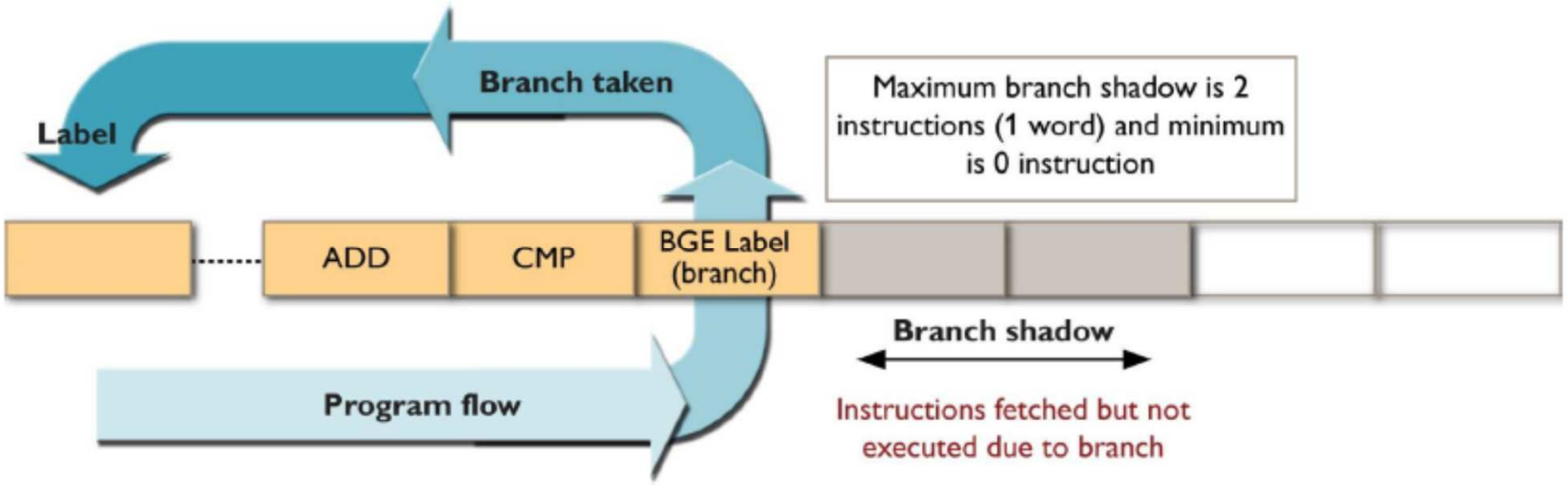
一条总线，两级流水线

- Cortex-M0+与Cortex-M3一样有两级特权级别：特权级和非特权级
- Cortex-M0+基于ARMv6-M架构,只有一条系统总线
- 优化的两级流水线架构
- 进一步降低了能耗，提高了性能



Branch shadow

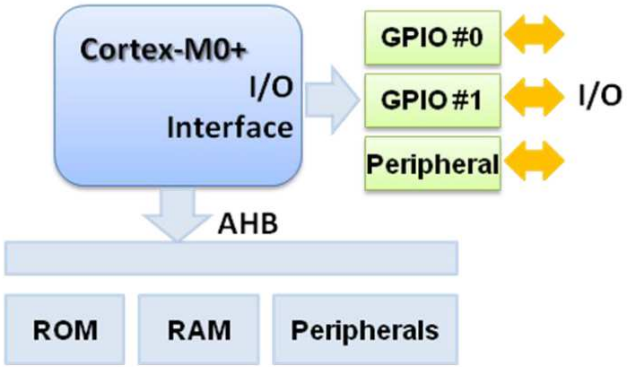
Branch shadow: 执行当前指令时，已经被预取出的指令个数（长度）。

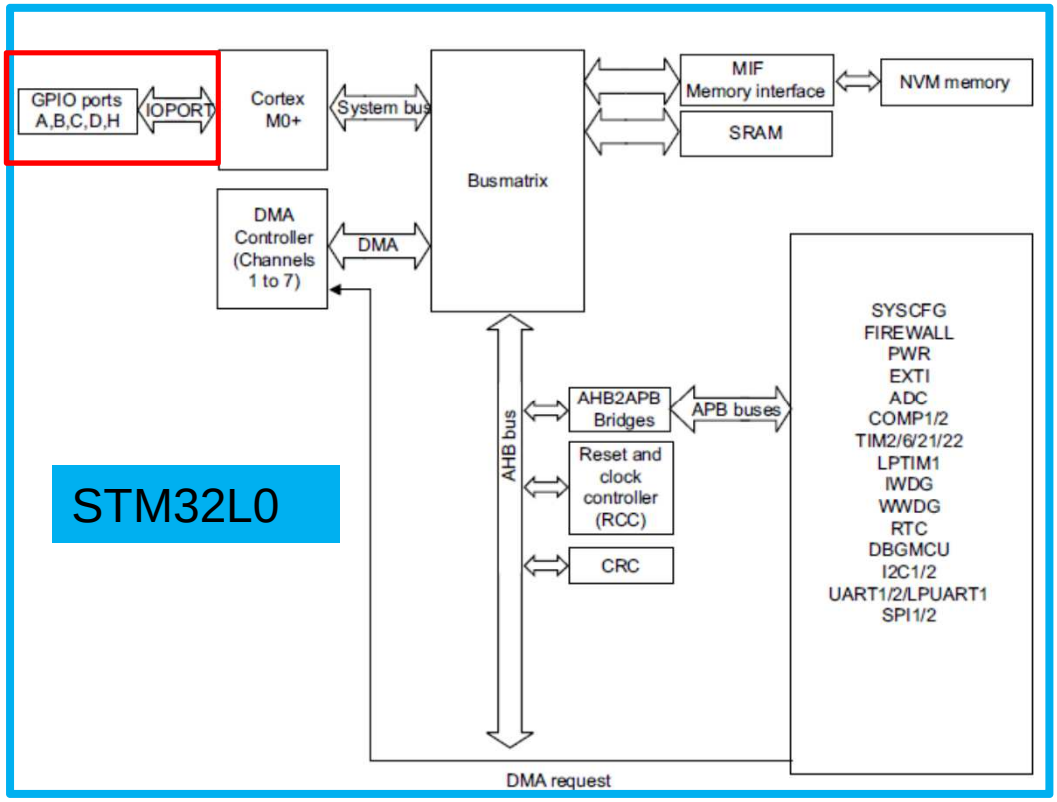


两级流水线，减少Branch shadow带来的能耗损失。

单周期I/O接口

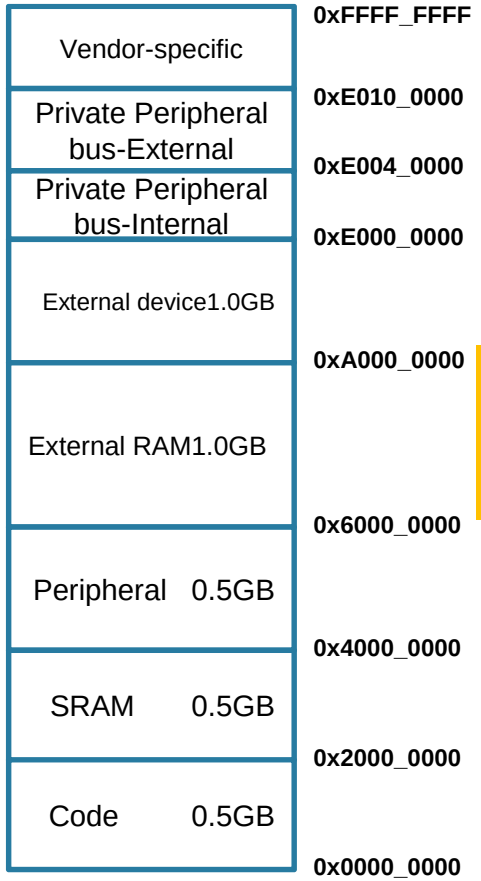
- 单周期I/O接口（可选）
  - 32-bit 接口
  - 支持32/16/8-bit传输
  - 支持存储器映射，可以像普通外设一样进行访问
  - 存储器映射地址由芯片生产商定义





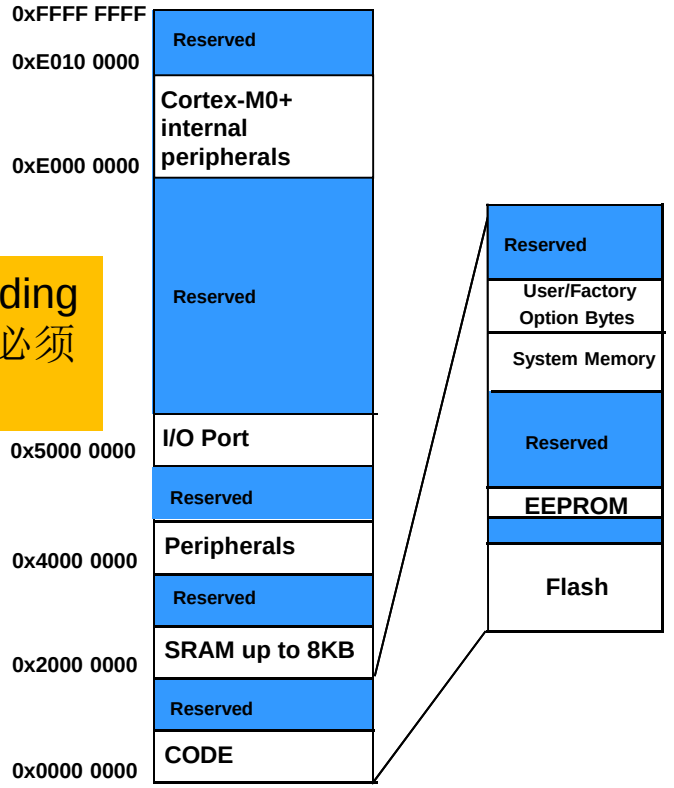
CM0+与CM3的区别
——存储器系统

Cortex-M0+预定义



- 不支持Bit-banding
- 所有数据传输必须是对齐的

STM32L0的定义



CM0+与CM3的区别——异常和中断

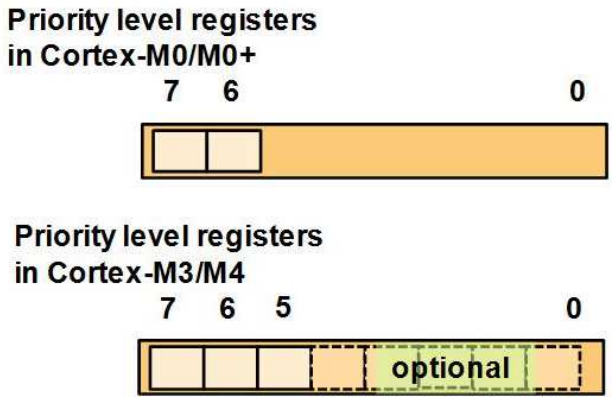
- 仅支持6个系统异常和32个外部中断。
- 不支持存储器管理fault，总线fault，用法fault和调试监视器异常。所有的fault都由硬件fault异常进行处理

	(Cortex-M0/M0+)	(Cortex-M3)		
Exception Type	ARMv6-M	ARMv7-M	Vector Table	Vector address (initial)
239		Device Specific Interrupts	Interrupt#239 vector [1]	0x000003FC
31	Device Specific Interrupts		Interrupt#31 vector [1]	0x000000BC
17			Interrupt#1 vector [1]	0x00000044
16			Interrupt#0 vector [1]	0x00000040
15	SysTick	SysTick	SysTick vector [1]	0x0000003C
14	PendSV	PendSV	PendSV vector [1]	0x00000038
13	Not used	Not used	Not used	0x00000034
12		Debug Monitor	Debug Monitor vector [1]	0x00000030
11	SVC	SVC	SVC vector [1]	0x0000002C
10	Not used	Not used	Not used	0x00000028
9			Not used	0x00000024
8			Not used	0x00000020
7			Not used	0x0000001C
6		Usage Fault	Usage Fault vector [1]	0x00000018
5		Bus Fault	Bus Fault vector [1]	0x00000014
4		MemManage (fault)	MemManage vector [1]	0x00000010
3	HardFault	HardFault	HardFault vector [1]	0x0000000C
2	NMI	NMI	NMI vector [1]	0x00000008
1			Reset vector [1]	0x00000004
0			MSP initial value	0x00000000

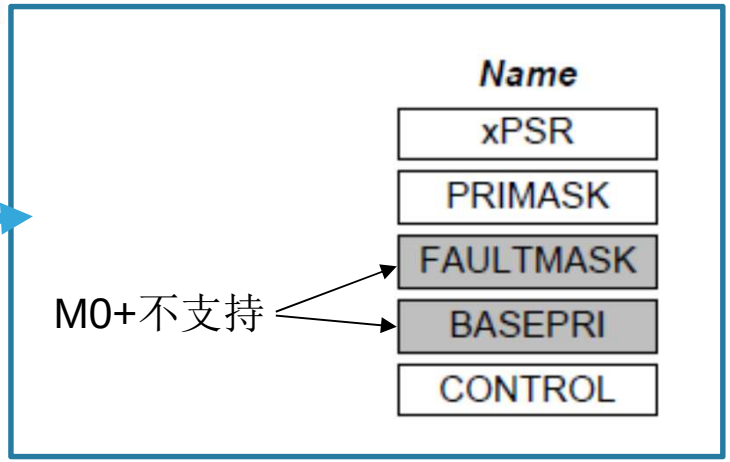
life.augmented

CM0+与CM3的区别——中断优先级

- 支持3个固定高优先级和4个可编程的优先级。
- 优先级配置寄存器为8位，但只能使用最高两位。也不分抢占优先级和亚优先级



- 没有中断状态寄存器
- 没有软件触发中断寄存器
- 不支持动态修改优先级
- 优先级屏蔽
- CM0+与CM3一样支持中断向量表重定位
- NVIC寄存器只支持字访问



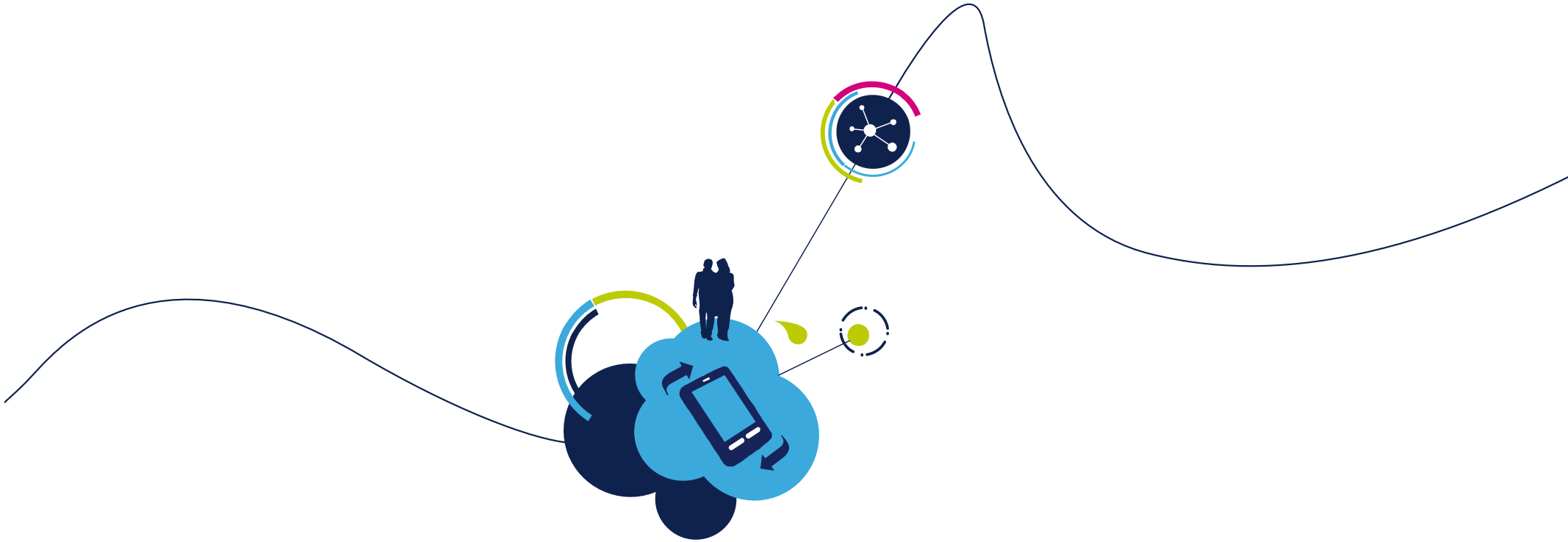
STM32L低功耗模式

STM32L支持5种低功耗模式

Cortex-M		STM32			
		STM32F系列		STM32L系列	
低功耗模式	说明	低功耗模式	说明	低功耗模式	说明
				低功耗运行模式	内核和外设保持运行状态。内部电源变换器工作在低功耗模式下，系统时钟频率降低。
睡眠模式	内核停止工作，外设继续工作	睡眠模式 (Sleep mode)	内核停止工作，外设继续工作	睡眠模式	内核停止工作，外设继续工作。
				低功耗睡眠模式	内核停止工作，外设继续工作。内部电源变换器工作在低功耗模式下，系统时钟频率降低。
Deep sleep mode	停止系统时钟，关闭PLL和Flash	停止模式 (Stop mode)	所有时钟停止	停止模式	所有时钟停止
		待机模式 (Standby mode)	内核区域掉电	待机模式	内核区域掉电

STM32L系列低功耗模式的进入与退出

模式	进入	唤醒	对VCORE域时钟影响	对VDD域时钟影响	内部电源变换器	IO 状态	唤醒延迟
低功耗运行模式	设置**LPSDSR** 位和**LPRUN** 位 + **设置系统时钟**	恢复系统时钟和内部电源变换器的工作模式	无	无	低功耗模式	所有I/O口的状态和运行时保持一致	无
睡眠模式	**WFI**	任意中断	CPU时钟关闭，不影响其他时钟	无	正常模式		无
	WFE	唤醒事件					
低功耗睡眠模式	设置**LPSDSR** 位 + **WFI**	任意中断		无	低功耗模式		电源变换器的改变时间+FLASH的唤醒时间
	设置**LPSDSR** 位 + **WFE**	唤醒事件					
停止模式	设置**PDDS**和**LPSDSR**位 + 设置**SLEEPDEEP**位 + **WFI** 或 **WFE**	任意EXTI中断 (在EXTI寄存器中配置的**内部**或者**外部**中断源)	所有的VCORE域时钟都关闭	HSI，HSE和MSI都关闭	正常模式/低功耗模式		MSI RC的唤醒时间+电源变换器的唤醒时间 + FLASH的唤醒时间 唤醒的典型值为7.9us
待机模式	设置**PDDS**和 + 设置**SLEEPDEEP**位 + **WFI** 或 **WFE**	WKUP引脚的上升沿，RTC报警 (Alarm A 或者 Alarm B)，RTC唤醒事件，RTC时间戳事件，RTC侵入事件，NRST引脚的外部复位，IWDG复位			关闭	所有I/O口都保持在高阻状态	V_{REFINT} 开的情况下唤醒典型值为57.2us V_{REFINT} 关的情况下唤醒典型值为2.4ms



Cortex-M系列总对比

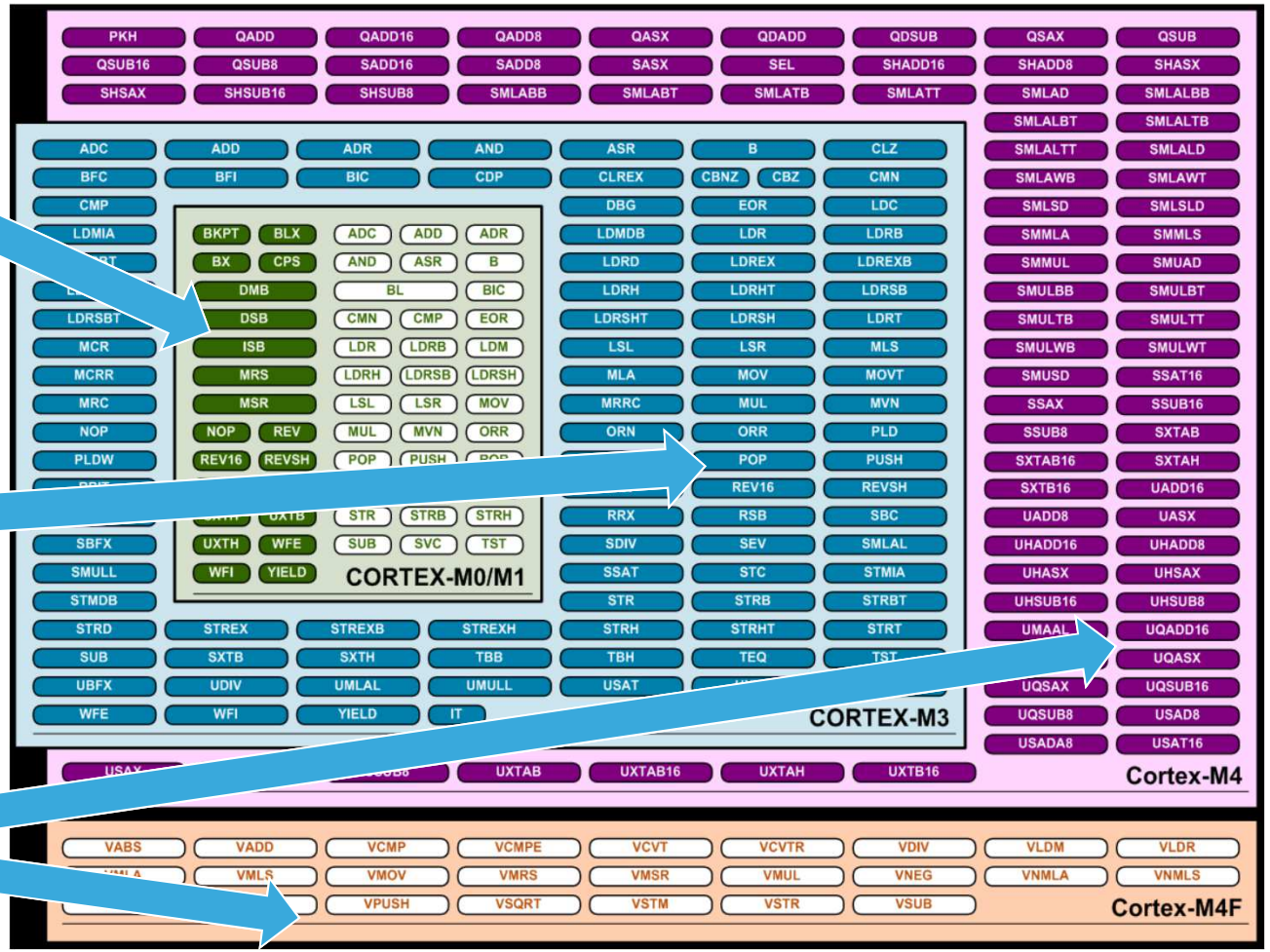
life.augmented

指令集

Cortex M0/M0+
•Cortex-M3指令的子集

Cortex M3
•Thumb2指令集（16位and 32 位指令）

Cortex M4
• DSP指令集
• 浮点运算指令

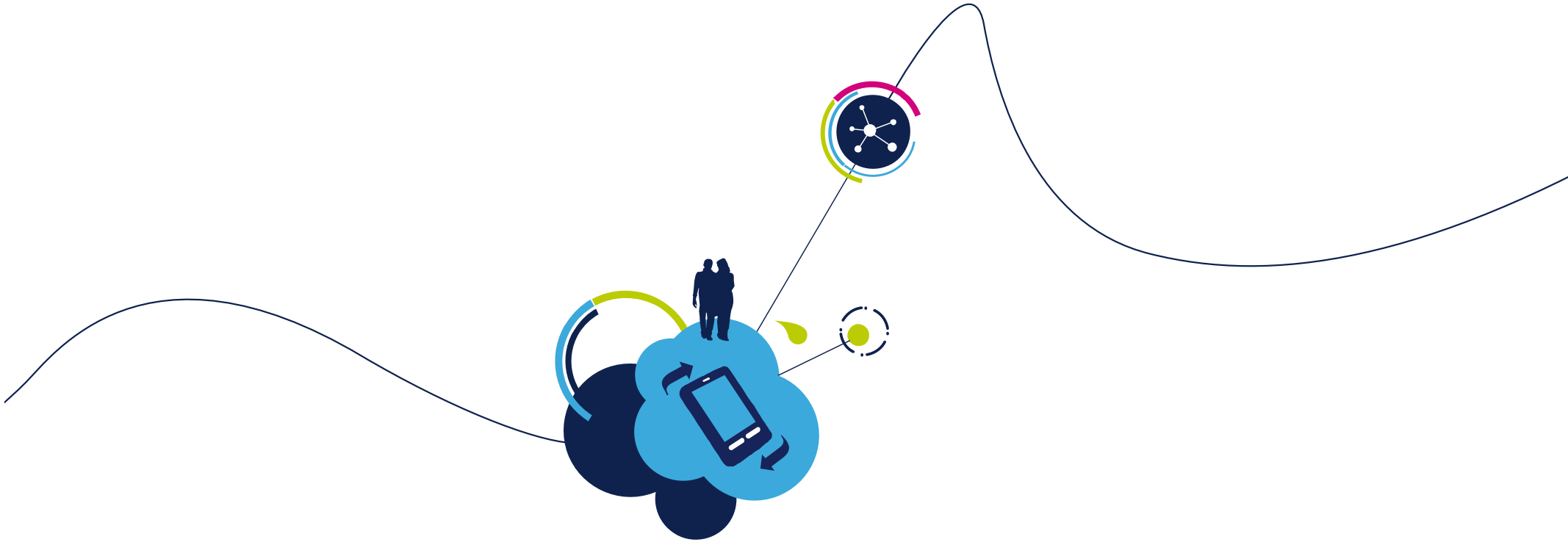


Cortex-M各系列内核性能比较

	CM0+	CM0	CM3	CM4
架构版本	ARMv6-M	ARMv6-M	ARMv7M	ARMv7ME
流水线	2级流水线	3级流水线	3级流水线	3级流水线
特权级别	特权级/非特权级	特权级	特权级/非特权级	特权级/非特权级
Bus Interface	1	1	3	3
指令集	Thumb,Thumb-2 subset	Thumb,Thumb-2 subset	Thumb + Thumb-2	Thumb + Thumb-2
Hardware divide	NO	NO	YES	YES
单周期 SIMD/DSP	NO	NO	NO	YES
MAC	NO	NO	YES(多周期)	YES（单周期）
饱和运算指令	NO	NO	仅有USAT, SSAT	YES
FPU	NO	NO	NO	YES
CoreMark/MHz	2.42	2.33	3.32	3.40
DMIPS/MHz	0.84	0.94	1.25	1.25

Cortex-M各系列内核性能比较

	CM0+	CM0	CM3	CM4
中断/异常	NMI + 1 to 32	NMI+ 1 to	NMI+ 1 to 240	NMI + 1 to 240
中断优先级	4	4	8 to 256	8 to 256
向量表重定位	YES(可选）	NO	YES	YES
软件触发中断寄存器	NO	NO	YES	YSE
中断状态寄存器	NO	NO	YES	YES
中断寄存器访问	32-bit	32-bit	8/16/32-bit	8/16/32-bit
动态优先级调整	No	NO	YES	YES
存储器保护	MPU(可选)	No	MPU(可选)	MPU(可选)
唤醒中断控制器（WIC）	YES	YES	YES	YES
低功耗模式	Integrated WFI and WFE Instructions and Sleep On Exit capability Sleep & Deep Sleep Signals Optional Retention Mode with ARM Power Management Kit	Integrated WFI and WFE Instructions and Sleep On Exit capability Sleep & Deep Sleep Signals Optional Retention Mode with ARM Power Management Kit	Integrated WFI and WFE Instructions and Sleep On Exit capability. Sleep & Deep Sleep Signals. Optional Retention Mode with ARM Power Management Kit	Integrated WFI and WFE Instructions and Sleep On Exit capability. Sleep & Deep Sleep Signals. Optional Retention Mode with ARM Power Management Kit
位操作	NO	NO	Bit-banding	Bit-banding



Cortex-M系列软件移植

life.augmented

02/12/2014

Cortex-M系列的软件兼容性(1/2)

- 向上路径(M0->M3->M4)的移植直接了当
  - Cortex-Mx内核的指令集是Cortex-My内核(x<y)指令集的子集。软件可以直接移植过去，性能则随着MCU时钟频率的提高以及冯诺依曼到哈佛架构的改进而得到提升
  - 推荐对代码进行重新编译
    - 从M0内核升级到M3内核：重新编译可以充分利用更高效的指令，比如硬件除法
    - 从M0/M3内核升级到带FPU的M4内核：某些代码需要使用内联函数重新编写，以利用M4内核高级的DSP/SIMD指令
    - 对于STM32家族内的产品，丰富的外设也相互兼容以保证移植的顺畅

Cortex-M系列的软件兼容性(2/2)

- 向下路径(M3->M0/M0+)的移植需要注意
  - M0内核的NVIC和SCB寄存器只能字访问，M3可支持字、半字、字节访问
  - M3内核的某些NVIC和SCB寄存器在M0中没有
    - Interrupt Active Status Register
    - Software Trigger Interrupt Register
    - Vector Table Offset Register
    - 某些 fault status registers
  - 如果使用了汇编，某些指令不再被M0/M0+支持；对于C代码，某些指令如硬件除法，编译器会自动调用C库函数来处理除法操作
  - M0/M0+不支持非对齐数据传输
  - M0/M0+不再具有M3/M4所支持的位带操作
  - 使用用户线程模式和MPU特性的应用则不能移植到M0内核上
  - 某些M0的MCU支持存储器重映射，使用了M3特有的矢量表重定位特性的应用，可以使用存储器重映射来处理

- 向下路径(M4->M0/M0+)的移植需要注意
  - M4与M3内核基本相似，所以从M3移植的方法也适用M4
  - 某些用到DSP,浮点运算的应用，不适用M0/M0+

Thanks!