第14章

# 存储保护单元MPU

译者提示：MPU是CM3的选配件，许多CM3单片机中都没有加入。本章在翻译时，对原文改编比其它章节要强烈，且有一部分内容译自“古文观止”。如果时间不富裕，读者可以选择跳过。

* MPU概览
* MPU的寄存器组
* 启用MPU
* MPU的典型设置

## 14.0 译者添加的引子

MPU进入单片机还是很新鲜的事，为了让读者预先对它更有一点认识，译者加入了引文：

### 引子1：野指针与C语言

回顾一下，什么是指针？指针在内存中实际上是一个无符号整数（unsigned int），但是它的值被赋予特殊的解释：表示变量或函数的地址。所以才被形象地称为“指针”，就好像指向谁家似的。在使用指针前，都必须先让它指向有意义的，并且允许由程序使用的实体——数据和代码。而所谓“野指针”，就是指某个指针变量的值因故超出合法的范围，使其“枪口”乱指。程序逻辑错误、数组越界、堆栈溢出、指针未经初始化、对缓存与缓冲的处理不当、多任务环境中的紊乱危象，甚至是恶意地破坏等，都可以制造出野指针。如果使用野指针去读取或修改内存，则被读取或修改的位置是不可预料的。前者导致读回来的都是垃圾数据，后者则更是“血口喷人”——会破坏未知用途的数据。这常常导致系统发生莫名其妙的功能紊乱，严重时会使系统毫无征兆，没有理由地失控、死机。

野指针就像“肉里的刺，酱里的蛆”一般：一个野指针就足以崩溃整个系统，而且极其隐蔽，很难通过症状来找出是哪里存在野指针，甚至都不能判定症状是否因野指针造成（程序大了其它bug也很多，并且也能导致相同的症状）。野指针的发作概率越小，就越隐蔽，后患也越无穷。对于通常的单片机系统，是没有任何办法来防止野指针的破坏的，完全靠程序员的素质和自律。但智者千虑，必有一失。尤其是当程序规模变得很大时，复杂度会呈指数上升，千头万绪纠缠不清，就算是谨慎如诸葛亮，聪明如比尔·盖茨的天才，也不敢保证没有漏网之鱼。

嵌入式系统开发的首选语言是C语言。C语言的指针功能非常灵活、生猛、桀骜不驯，它是电，它是光，它是C语言中最闪亮的“Super Star”。C语言允许我们几乎随心所欲地把玩各种地址，离汇编语言中“放任自流”的程度也差不远了。可是，要是像汇编那样“明坏”倒也好，偏偏 C语言中的指针还因为语言特性附加了许多十分微妙的“潜规则”，令人防不胜防；更加暧昧的是指针与数组的关系，一维数组与多维数组的关系，多维数组与“星星”点灯的关系，指针与”[]”的二重唱，指针在宏中使用时极易弄巧成拙的暗箱操作……用 C语言的指针功能就像在玩一场勾魂的“野蛮游戏”，不知不觉其实你已“上线”，指针飞舞的世界战火连天，如果不想每天因爆发了却查不出来的bug而以泪洗面，提高警觉快张大双眼是必要的，但年深日久0还是难免有看不清楚而迟早粉身碎骨的时候。在系统程序的开发中，指针更是满天飞遍地爬，程序员在这无间世界里没有想过要逃脱，为什么要逃脱？完全是“你主宰，我崇拜，没有更好的办法，只能爱你，u r my super star”。

这里再说句题外话。如果读者不幸与被C语言的指针给沾上了，就要特别重视内功的修炼。除了要把《C程序设计》以及《C和指针》夜夜放在床头外，更要千方百计地弄到《C陷阱与缺陷》，以及《C专家编程》这两本书，它们两个堪比《葵花宝典》和《九阳真经》。

### 引子2：使命-关键系统

这种系统往往都用于性命攸关的场合，且必须连续无故障地工作，比如，火车调度系统、生命维持系统、大型发动机驱动器、核子反应堆控制、网络/电信的数据交换中枢等。如果失能，将导致惨重的经济与损失，甚至会使无数人死于非命。因此，决不允许这类系统出现上述情况。然而，这些系统的复杂度往往都非常高，几乎不可能由开发人员保证这种可靠性。

因此，需要在硬件水平上加入一个“公安机关”。通过它设置各种类型的“禁地”，并且施加多种规章条例。一旦发现违章，则强制改变执行流和处理器的工作状态，以便可以由软件做进一步的处理。这样，就可以为不同的程序限定一个内存使用范围，从而使野指针或恶意破坏无法影响不允许访问的区域。此即存储器保护单元（MPU）。

有时，对存储器的管理更进一步，做到可以对地址执行变换的程度，此时程序使用的地址未必是真实的存储器地址。它在MPU的基础上，还消灭了内存碎片和浪费，并且能进一步地让应用程序拥有舒适而敞亮的地址空间，从而使程序规模可以扩大甚至数百倍。此即为“存储器管理单元”（MMU）。带MMU的系统，往往也带cache，动态RAM等。这种系统对RAM容量的计量是以MB为单位的。可见，MMU是一个对处理器定位的“分水岭”。对MMU的介绍已经超出了本书的范围。

（本章篇幅虽然较长，但很多内容都是在寄存器的介绍，以及示例代码的反刍上，读者请放松阅读）

## 14.1 MPU概览

在Cortex-M3处理器中可以选配一个存储器保护单元（MPU），它可以实施对存储器（主要是内存和外设寄存器）的保护，从而使软件更加健壮和可靠。如果打算启用MPU，则在使用前，必须根据需要对其编程。如果没有启用MPU，则等同于系统中没有配MPU。MPU有如下的能力可以提高系统的可靠性：

* + 阻止用户应用程序破坏操作系统使用的数据
  + 阻止一个任务访问其它任务的数据区，从而把任务隔开。
  + 可以把关键数据区设置为只读，从根本上消除了被破坏的可能。
  + 检测意外的存储访问，如，堆栈溢出，数组越界。
  + 此外，还可以通过MPU设置存储器regions的其它访问属性，比如，是否缓区，是否缓冲等。

MPU在执行其功能时，是以所谓的“region”为单位的。一个region其实就是一段连续的地址，只是它们的位置和范围都要满足一些限制（对齐方式，最小容量等）。CM3的MPU共支持8个regions。怎么，嫌少？是少了点，不过，还允许把每个region进一步划分成更小的“子region”。此外，还允许启用一个“背景region”（即没有MPU时的全部地址空间），不过它是只能由特权级享用。在启用MPU后，就不得再访问定义之外的地址区间，也不得访问未经授权的region。否则，将以“访问违例”处理，触发MemManage fault。

MPU定义的regions可以相互交迭。如果某块内存落在多个region中，则访问属性和权限将由编号最大的region来决定。比如，若1号region与4号region交迭，则交迭的部分受4号region控制。

## 14.2 MPU的寄存器组

操作MPU是通过访问它的若干寄存器来实现的，如下表所示。

（译者注：此表摘自Cortex-M3 TRM）

|  |  |  |  |
| --- | --- | --- | --- |
| 名字 | 访问 | 地址 | 初值 |
| MPU类型寄存器 MPUTR | RO | 0xe000,ed90 | A |
| MPU控制寄存器 MPUCR | RW | 0xe000,ed94 | 0x0000,0000 |
| MPU region号寄存器MPURNR | RW | 0xe000,ed98 | - |
| MPU region基址寄存器MPURBAR | RW | 0xe000,ed9c | - |
| MPU region属性及容量寄存器(s) MPURASR | RW | 0xed00,eda0 | - |
| MPU region基址寄存器的别名1 | D9C的别名 | 0xed00,eda4 | - |
| MPU region属性及容量寄存器的别名1 | DA0的别名 | 0xed00,eda8 | - |
| MPU region基址寄存器的别名2 | D9C的别名 | 0xed00,edac | - |
| MPU region属性及容量寄存器的别名2 | DA0的别名 | 0xed00,edb0 | - |
| MPU region基址寄存器的别名3 | D9C的别名 | 0xed00,edb4 | - |
| MPU region属性及容量寄存器的别名3 | DA0的别名 | 0xed00,edb8 | - |

让我们来详细地介绍上述寄存器，第一个就是MPU类型寄存器（MPUTR），如表14.1所示

表14.1 MPU类型寄存器MPUTR （地址：0xE０00\_ED90）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 位段 | 名称 | 类型 | 复位值 | 描述 |
| 23:16 | IREGION | R | 0 | MPU支持的指令region数量。因为ARMv7-M只使用单个统一的MPU，此位段永远为零 |
| 15:8 | DREGION | R | 0 | MPU支持的数量。若系统中配了MPU则为8,否则为零 |
| 0 | SEPARATE | R | 0 | 固定为零 |

从表中我们可以看出，通过读取DREGION的值，能够判断芯片中是否配了MPU。

接下来我们看一看MPU控制寄存器MPUCR如表14.2所示

表14.2 MPU控制寄存器MPUCR （地址：0xE０00\_ED94）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 位段 | 名称 | 类型 | 复位值 | 描述 |
| 2 | PRIVDEFENA | RW | 0 | 是否为特权级打开缺省存储器映射（即背景region）。  1=特权级下打开背景region  0=不打开背景region。任何访问违例以及对region外地址区的访问都将引起fault |
| 1 | HFNMIENA | RW | 0 | 1=在NMI和硬fault服务例程中不强制除能MPU  0=在NMI和硬fault服务例程中强制除能MPU |
| 0 | ENABLE | RW | 0 | 使能MPU |

通过把PRIVDEFENA置位，可以在没有建立任何region就使能MPU的情况下，依然允许特权级程序访问所有地址，而只有用户级程序被拒之门外。然而，如果设置了其它的region并且使能了MPU，则背景region与这些region重合的部分，就要受各region的限制。为了方便理解，让我们作一个对比，看看PRIVDEFENA在置位与清零时，系统对访问的限制有何不同，如图14.1所示。

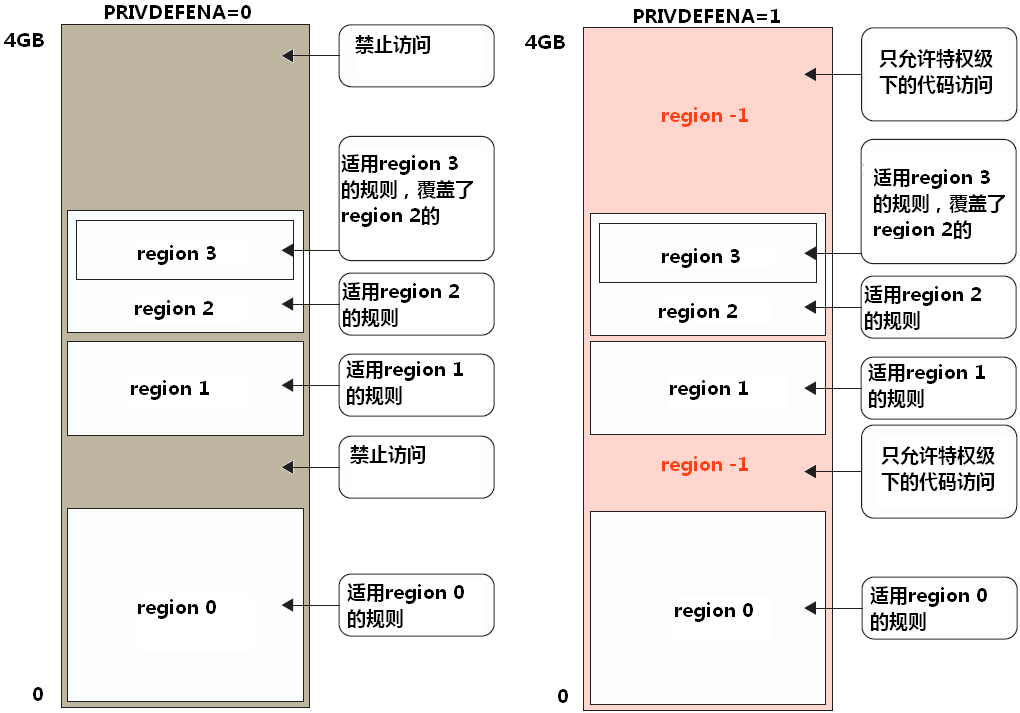


图14.1 PRIVDEFENA的影响

要注意，只要没有极另类的考虑，就要到万事就绪后，最后一步才置位ENABLE位。否则，就有可能因region没有配置好而意外地产生MemManage fault。很多条件下，为安全起见，最好在执行配置MPU的子程前先除能MPU，待执行后再重新使能MPU。

注意：这里有个例外：不管MPU如何限制，响应异常时的取向量操作，以及对系统分区(system partition)的访问总是不受影响的。

译注：这里所说的“系统分区”，作者并没有解释过。估计有可能是包含在前面提到的“SCS”区

When the MPU is enabled, only the system partition and vector table loads are always

accessible.

配置任何一个region之前，都需要在MPU内选中这个region，这可以通过把region号写入MPU region号寄存器(MPURNR)来完成，其定义如表14.3所示

表14.3 MPU region号寄存器MPURNR （地址：0xE０00\_ED98）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 位段 | 名称 | 类型 | 复位值 | 描述 |
| 7:0 | REGION | RW | - | 选择下一个要配置的region。因为只支持8个region，所以事实上只有[2:0]有意义 |

选好了region后，就可以在另外两个寄存器中配置该region的所有属性了。

为了能快速地配置多个regions，还有另一种快捷方式。在MPU region基地址寄存器(MPURBAR)中有两个位段：VALID和REGION，它们配合使用可以绕过MPURNR。MPURBAR的定义如表14.4所示

表14.4 MPU region基址寄存器MPURBAR （地址：0xE０00\_ED9C）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 位段 | 名称 | 类型 | 复位值 | 描述 |
| 31:N | ADDR | RW | - | Region基址字段。N取决于region容量，以使**基址在数值上能被容量整除**。在MPU region属性及容量寄存器中有个SZENABLE位段，它决定ADDR中有多少个位被采用。 |
| 4 | VALID | RW | - | 决定是否理会写入REGION字段的值  1=MPU region号寄存器被REGION覆盖  0=MPU region号寄存器的值保持不变 |
| 3:0 | REGION | RW | - | MPU region覆写位段 |

从表中我们可以看出，基址必须对齐到region容量的边界。举例来说，如果你定义的region容量是64KB，那么它的基址就必须能被64KB整除。这里，像0x0001,0000；0x0002,0000这样的，就是合法的基址（低16位为0）。

如果读取REGION位段，返回的总是当前的region号，并且VALID总返回0。通过设置VALID=1和REGION=n，也可以改变一个region的编号。相比于先设置MPU region号寄存器再设置本寄存器的正统做法而言，这是一个快捷方式。

注意：必须以字的方式来访问本寄存器，否则结果不可预知。

配置好了基地址，我们还需要详细定义region的其它方方面面。这需要设置MPU属性及容量寄存器。这个寄存器是全体可读可写的，且复位值未知，如下表14.5所示：

表14.5 MPU region属性及容量寄存器MPURASR（地址：0xE000\_EDA0）

|  |  |  |  |
| --- | --- | --- | --- |
| 位段 | 长度 | 名称 | 功能 |
| 31:29 | 3 | - | 保留 |
| 28 | 1 | XN | 1=此区禁止取指  2=此区允许取指 |
| 27 | 1 | - | 保留 |
| 26:24 | 3 | AP | 访问许可，如下表所示   |  |  |  |  | | --- | --- | --- | --- | | 值 | 特权级下的许可 | 用户级下的许可 | 典型用法 | | 0b000 | 禁地 | 禁地 | 该区没有存储器，是空地址 | | 0b001 | RW | 禁地 | OS以及系统软件使用的数据区 | | 0b010 | RW | RO | 禁止在用户级下更改的高危地带 | | 0b011 | RW | RW | 共享内存，或彻底开放的设备 | | 0b100 | n/a | n/a | n/a | | 0b101 | RO | 禁地 | OS使用的常量数据 | | 0b110 | RO | RO | 常量数据或只读存储器的地址区 | | 0b111 | RO | RO | 常量数据或只读存储器的地址区 | |
| 23:22 | 2 | － | 保留 |
| 21:19 | 3 | TEX | 类型扩展 |
| 18 | 1 | S | Sharable（可否共享）  1=共享可  0=共享不可 |
| 17 | 1 | C | Cachable（可否缓存）  1=缓存可  0=缓存不可 |
| 16 | 1 | B | Buffable（可否缓冲）  1=缓冲可  0=缓冲不可 |
| 15:8 | 8 | SRD | 子region除能位段。每设置SRD的一个位，就会除能与之对应的一个子region。容量大于128字节的region都被划分成8个容量相同的子region。容量小于等于128字节的region不能再分。更多信息，请参见对子Region的论述。 |
| 7:6 | 2 | - | 保留 |
| 5:1 | 5 | REGIONSIZE | Region容量，单位是字节。容量为1<<(REGIONSIZE+1)，但是最小容量为**32**字节 |
| 0 | 1 | SZENABLE | 1=使能此region 0=除能此region |

表中提到了“子region”的概念（[15:8]）。原来，8个region的定义过于粗枝大叶，因而允许再精雕细琢，把每个region的内部进一步划分成更小的块，这就是子region。但是子region的使用有限制：每个region必须8等分，每份是一个子region，而且所有子region的属性都与“父region”的是相同的。每个子region可以独立地使能或除能（相当于可以部分地使能一个region）：SRD中的8个位，每个位控制一个子region是否被除能。如SRD.3=**0**，则3号子region被除能。如果某个子region被除能，且其对应的地址范围又没有落在其它region中，则对该子region覆盖范围的访问将引发fault。最后，能被“大卸八块”的region，最小也要有256字节。如果是对128字节或者是更小的region划分子region，则后果是不可预料的。

再看它的AP位段，为了详细说明把它做成了一个表中表。AP位段用于限定各种访问权限，这也是加以分区保护的最重要组成部分。

位段[28]的名字是XN（eXecute Never），它决定在本region中是否允许取指。如果不允许取指（清零），则任何指令预取都将触发MemManage fault。这有什么用？通常，可以把新得到的还不受信任的代码先存储到此区，待经过身份鉴定后，再允许它执行。

表中楷体的TEX, S, B和C（整体位于[21:16]），对应着存储系统中比较高级的概念。CM3中没有缓存(cache)，但是CM3是以v7-M的架构设计的，而v7-M支持外部缓存（差不多是L2缓存的地位）以及更先进的存储器系统。按v7-M的规格说明，可以通过对这些位段的编程，来支持多样的内存管理模型。从v6开始，ARM架构支持两级缓存（与x86的缓存系统是异曲同工的），分别是：内部缓存和外部缓存，它们可以有不同的缓存方针(policy)，这些位组合的详细功能如下表所示：

表14.6 TEX,C,B对存储器类型的决定

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| TEX | C | B | 描述 | 存储器类型 | 可否共享 |
| 000 | 0 | 0 | 严格按顺序 | 严格按顺序 | 总是可以 |
| 000 | 0 | 1 | 共享的设备 | 设备 | 总是可以 |
| 000 | 1 | 0 | 片外或片内的“写通”型内存，没有写allocate | 普通 | S位决定 |
| 000 | 1 | 1 | 片外或片内的“写回”型内存，没有写allocate | 普通 | S位决定 |
| 001 | 0 | 0 | 片外或片内的“缓存不可”型内存 | 普通 | S位决定 |
| 001 | 0 | 1 | n/a | n/a | n/a |
| 001 | 1 | 0 | 实现者您说了算 | 您说了算 | 您说了算 |
| 001 | 1 | 1 | 片外或片内的“写回”型，带读和写的allocate | 普通 | S位决定 |
| 010 | 1 | x | 共享不可的设备 | 设备 | 总是不可 |
| 010 | 0 | 1 | n/a | n/a | n/a |
| 010 | 1 | x | n/a | n/a | n/a |
| 1BB | **A** | **A** | 带缓存的内存。BB=适用于片外内存, AA=适用于片内内存 | 普通 | S位决定 |

表中最后一项越发离奇，它是TEX的MSB=1时的情况。此时，如果该region是片内存储器，则由C和B决定其缓存属性（AA）；如果是片外存储器，则由TEX的[1:0]决定其缓存属性（BB）。不管是AA还是BB，每个数值的含义都是相同的，如下表所示：

表14.7 缓存方针编码

|  |  |
| --- | --- |
| 存储器属性编码 (AA and BB) | 高速缓存策略 |
| 00 | 缓存不可 |
| 01 | 写回，读写均有allocate |
| 10 | 写通，写没有allocate |
| 11 | 写回，写没有allocate |

欲知缓存行为和缓存方针的更多详情，请参阅《ARM Architecture Application Level Reference Manual(Ref2)》。

再看本章开头的寄存器表，最后的8个其实是4对，且后3对都第1对的别名，这可真是“狡兔三窟”啊。再仔细看，你会发现它们的地址是连续的。这下是不是有看出一些端倪了？请看下段译自Cortex-M3 TRM的解释：

**9.2.3 使用别名(alias)寄存器访问MPU**

通过寄存器别名机制，你可以使用STM指令加速对regions的初始化——一次可以最多初始化4个。一共有3组别名寄存器。别名以完全相同的方式来访问（真实的）寄存器，它们的存在是为了让你能以“顺序写”(STM指令)来一次更新1-4个region。当无需在某些“临界”区域中以“除能region/更改region属性/使能retgion”的小心方式，来一个个地进行配置时，这个机制就显得特别有用。

下面举一个一次更改4个region的代码例子：

; R1 = 一个指针，指向某RTOS进程控制块中的4个region对子（共8个字）

MOV R0, #NVIC\_BASE

ADD R0, #MPU\_REG\_CTRL

LDM R1, [R2-R9] ; 加载4个region的信息

STM R0, [R2-R9] ; 一句话完成4个region的配置

这么一来，只要事先做好一个配置表格，就可以一气呵成了。

**注意**

你不能使用这些别名来读取regions的内容，因为必须要先写region号。

在C/C++下通常使用memcpy()函数来完成上段汇编的功能。但是，你必须验证CRT库，在实现memcpy()时必须是按字拷贝的——也就是两个long\* 指针之间的拷贝，而不得是char\*, short\*什么的。

本章后面还有一个“一题多解”的例子，最后的解法就是使用这里讲到的思路

## 14.3 启用MPU

MPU寄存器看起来比较复杂，那是自然了，毕竟已经上升到存储器管理的高度。但如果我们胸有成竹——已经想好了对存储器如何划分，这就只是一些繁琐和考验细心的体力活。典型情况下，在启用MPU的系统中，都会有下列的regions。

* + 特权级的程序代码（如OS内核和异常服务例程）
  + 用户级的程序代码
  + 特权级程序的数据存储器，位于代码区中（data\_stack）
  + 用户级程序的数据存储器，位于代码区中（data\_stack）
  + 通用的数据存储器，位于其它存储器区域中（如，SRAM）
  + 系统设备区，只允许特权级访问，如NVIC和MPU的寄存器所有的地址区间
  + 常规外设区，如UART，ADC等。

对于CM3来说，绝大多数region中，都有TEX=0，C=1，B=1。系统设备（如NVIC）必须“严格顺序”（strongly ordered）访问；另一方面，外设regions则可以共享（TEX=0, C=0, B=1）。如果想要在某个region中，确保所有的总线fault都是精确的，就必须把该region严格顺序化（TEX=0, C=0, B=0）。这样一来写缓冲被除能，但也因此产生性能损失的代价。

图14.2给出了MPU初始化序列的流程模式图。在使能MPU前，或者把向量表重定位到了RAM，一定不要忘记为MemManage fault建立向量，并且在NVIC的系统handler控制及状态寄存器SHCSR中使能MemManage fault。只有这样做了，才能在产生MPU违例时，让MemManage fault服务例程得以执行。

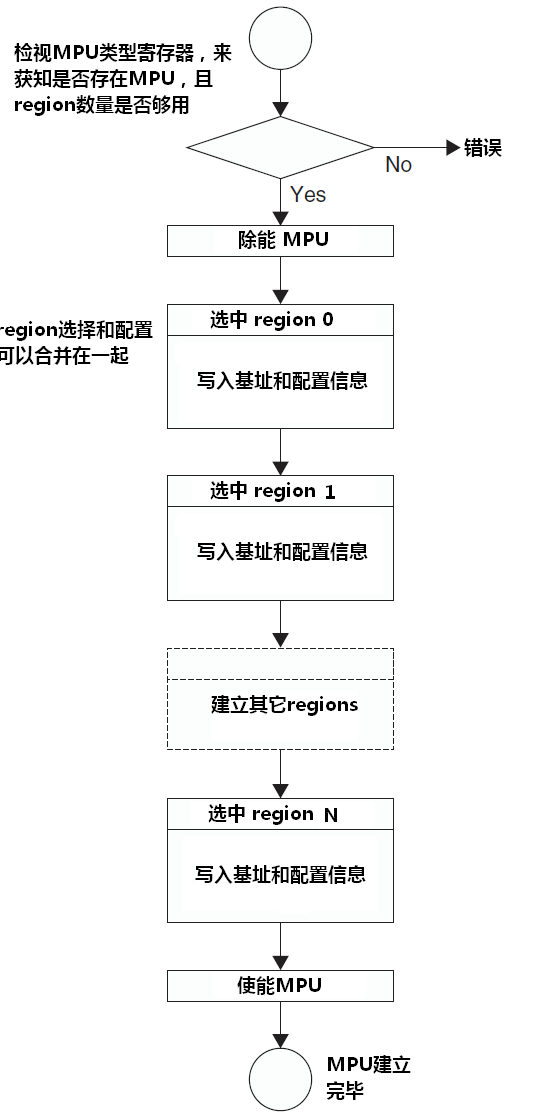


图14.2 MPU初始化序列

下面举一个简单的例子，它只有4个region，则配置代码如下所演示：

LDR R0, =0xE000ED98 ; Region号寄存器

MOV R1, #0 ; 选择region 0

STR R1, [R0]

LDR R1, =0x00000000 ; 基址 = 0x00000000

STR R1, [R0, #4] ; MPU Region 基址寄存器

LDR R1, =0x0307002F ; R/W, TEX=0,S=1,C=1,B=1, 16MB, Enable=1

STR R1, [R0, #8] ; MPU Region 属性及容量寄存器

MOV R1, #1 ; 选择region 1

STR R1, [R0]

LDR R1, =0x08000000 ; 基址 = 0x08000000

STR R1, [R0, #4] ; MPU Region 基址寄存器

LDR R1, =0x0307002B ; R/W, TEX=0,S=1,C=1,B=1, 4MB, Enable=1

STR R1, [R0, #8] ; MPU Region 属性及容量寄存器

MOV R1, #2 ; 选择 region 2

STR R1, [R0]

LDR R1, =0x40000000 ; 基址 = 0x40000000

STR R1, [R0, #4] ; MPU Region 基址寄存器

LDR R1, =0x03050039 ; R/W, TEX=0,S=1,C=0,B=1, 512MB, Enable=1

STR R1, [R0, #8] ; MPU Region 属性及容量寄存器

MOV R1, #3 ; 选择 region 3

STR R1, [R0]

LDR R1, =0xE0000000 ; 基址 = 0xE0000000

STR R1, [R0, #4] ; MPU Region 基址寄存器

LDR R1, =0x03040027 ; R/W, TEX=0,S=1,C=0,B=0, 1MB, Enable=1

STR R1, [R0, #8] ; MPU Region 属性及容量寄存器

MOV R1, #1 ; 准备使能MPU

STR R1, [R0,#-4] ; 使能MPU(0xE000ED98-4=0xE000ED94)

这段代码执行后，生成如下的4个regions:

|  |  |  |  |
| --- | --- | --- | --- |
| 特权级代码 | 0x0000\_0000-0x00FF\_FFFF(16MB) | 全访问 | 缓存可 |
| 特权极数据 | 0x0800\_0000-0x0803\_FFFF(4MB) | 全访问 | 缓存可 |
| 外设 | 0x4000\_0000-0x5FFF\_FFFF(512MB) | 全访问 | 共享设备 |
| 系统控制 | 0xE000\_0000-0xE00F\_FFFF(1MB) | 特权级访问 | 严格顺序，XN |

通过使用基址寄存器的VALID和REGION位段，可以把region选择和基址设置的两个动作合并成一个，从而缩短代码，如下所示：

LDR R0, =0xE000ED9C ; MPU region基址寄存器

LDR R1, =0x00000010 ; 基址=0x00000000, **region=0, valid=1**

STR R1, [R0, #0] ; 设置region 0的基址

LDR R1, =0x0307002F ; R/W, TEX=0,S=1,C=1,B=1, 16MB, Enable=1

STR R1, [R0, #4] ; MPU Region 属性及容量寄存器

LDR R1, =0x08000011 ; 基址=0x08000000, **region=1, valid=1**

STR R1, [R0, #0] ; MPU Region 基址寄存器

LDR R1, =0x0307002B ; R/W, TEX=0,S=1,C=1,B=1, 4MB, Enable=1

STR R1, [R0, #4] ; MPU Region 属性及容量寄存器

LDR R1, =0x40000012 ; 基址=0x40000000, **region=2, valid=1**

STR R1, [R0, #0] ; MPU Region基址寄存器

LDR R1, =0x03050039 ; R/W, TEX=0,S=1,C=0,B=1, 512MB, Enable=1

STR R1, [R0, #4] ; MPU Region属性及容量寄存器

LDR R1, =0xE0000013 ; 基址=0xE0000000, **region=3, valid=1**

STR R1, [R0, #0] ; MPU Region 基址寄存器

LDR R1, =0x03040027 ; R/W, TEX=0,S=1,C=0,B=0, 1MB, Enable=1

STR R1, [R0, #4] ; MPU Region 属性及容量寄存器

MOV R1, #1 ; 使能MPU

STR R1, [R0,#-8] ; MPU控制寄存器(0xE000ED9C-8=0xE000ED94)

看，代码变短了吧！不过，还有比这更厉害的，让代码更短更快。这要通过使用MPU别名寄存器的地址来完成。在MPU属性及容量寄存器(MPUASR)的后面，有3组MPU基址寄存器(MPUBAR)和MPU属性及容量寄存器的别名，连同真实的MPUBAR与MPUASR，它们共有4组，分布在一个连续的8字空间中，于是就可以使用LDM/STM指令来“串烧”，如下所示：

LDR R0, =0xE000ED9C ; MPU reigon基址寄存器

LDR R1, =MPUconfigTab ; 预定义的MPU初始化数值表

**LDMIA R1!, {R2-R9} ; 一气从表中读完8个字**

**STMIA R0!, {R2-R9} ; 一气初始化4个region**

B MPUconfigEnd

**ALIGN 4** ; 此汇编指示字可以确保下述的字定义一定是对齐到字

MPUconfigTab ; 边界的，因为在使用LDM/STM时，地址必须按字对齐

DCD 0x00000010 ; 基址=0x00000000, region=0,valid=1

DCD 0x0307002F ; R/W, TEX=0,S=1,C=1,B=1, 16MB, Enable=1

DCD 0x08000011 ; 基址=0x08000000, region=0,valid=1

DCD 0x0307002B ; R/W, TEX=0,S=1,C=1,B=1, 4MB, Enable=1

DCD 0x40000012 ; 基址=0x40000000, region=0,valid=1

DCD 0x03050039 ; R/W, TEX=0,S=1,C=0,B=1, 512MB, Enable=1

DCD 0xE0000013 ; 基址=0xE0000000, region=0,valid=1

DCD 0x03040027 ; R/W, TEX=0,S=1,C=0,B=0, 1MB, Enable=1

MPUconfigEnd

LDR R0, =0xE000ED94 ; MPU 控制寄存器

MOV R1, #1 ; 使能MPU

STR R1, [R0]

若用此法，显然必须保证：region配置早已安排好了，否则就只能用上面的更通用的办法。为了使软件更有模块化，可以把建立region的工作包装到一个子程序中，不妨名为MpuRegionSetup。它接受相关参数（编号，基址，容量/属性），并执行建立region的工作。主程序通过呼叫它若干次来逐一设置好每个region。

上面的小凉菜吃了三次，想必读者已经腻了吧。下面就上主菜，使用模块化的思路，代码如下所示。这段代码的后面部分还精彩地演示了新好指令BFI和UBFX的使用：

MpuSetup ; 入口函数，它内部呼叫若干子程序来完成MPU设置

PUSH {R0-R6,LR}

LDR R0, =0xE000ED94 ; MPU 控制寄存器

MOV R1, #0

STR R1, [R0] ; 配置前先除能MPU

; --- Region #0 ---

LDR R0, =0x00000000 ; Region 0: 基址 = 0x00000000

MOV R1, #0x0 ; Region 0: Region号 = 0

MOV R2, #0x17 ; Region 0: 容量 = 0x17 (16MB)

MOV R3, #0x3 ; Region 0: AP = 0x3 ( 全访问)

MOV R4, #0x7 ; Region 0: MemAttrib = 0x7

MOV R5, #0x0 ; Region 0: 子region除能=0

MOV R6, #0x1 ; Region 0: {XN, Enable} = 0,1

BL MpuRegionSetup

; --- Region #1 ---

LDR R0, =0x08000000 ; Region 1: 基址 = 0x08000000

MOV R1, #0x1 ; Region 1: Region号 = 1

MOV R2, #0x15 ; Region 1: 容量 = 0x15 (4MB)

MOV R3, #0x3 ; Region 1: AP = 0x3 (全访问)

MOV R4, #0x7 ; Region 1: MemAttrib = 0x7

MOV R5, #0x0 ; Region 1: 子region除能= 0

MOV R6, #0x1 ; Region 1: {XN, Enable} = 0,1

BL MpuRegionSetup

... ; 以相同的方法建立region #2和region #3

; --- Region #4-#7 除能 ---

MOV R0, #4

BL MpuRegionDisable

MOV R0, #5

BL MpuRegionDisable

MOV R0, #6

BL MpuRegionDisable

MOV R0, #7

BL MpuRegionDisable

LDR R0, =0xE000ED94 ; MPU 控制寄存器

MOV R1, #1

STR R1, [R0] ; 使能MPU

POP {R0-R6,PC} ; 返回

MpuRegionSetup

; MPU region 设置及启用子程

; 入口条件:

; R0 = 基址

; R1 = Region号

; R2 = 容量

; R3 = AP (访问许可)

; R4 = MemAttrib ({TEX[2:0], S, C, B})

; R5 = 子region除能

; R6 = {XN,Enable}

PUSH {R0-R1, LR}

BIC R0, R0, #0x1F ; 清零基址中铁定不会用到的位段

BFI R0, R1, #0, #4 ; 把region号插入到R0[3:0]

ORR R0, R0, #0x10 ; 置位VALID位

LDR R1, =0xE000ED9C ; 加载MPU Region基址寄存器的地址

STR R0, [R1] ; 填写之

AND R0, R6, #0x01 ; 读取使能位

UBFX R1, R6, #1, #1 ; 读取XN位

BFI R0, R1, #28, #1 ; 把 XN 插入到 R0[28]

BFI R0, R2, #1 , #5 ; 把region容量(R2[4:0])插入到R0[5:1]中

BFI R0, R3, #24, #3 ; 把AP(R3[2:0])插入到R0[26:24]中

BFI R0, R4, #16, #6 ; 把memattrib(R4[5:0])插入到R0[21:16]中

BFI R0, R5, #8, #8 ; 把子SRD(R5[7:0])插入到R0[15:8]中

LDR R1, =0xE000EDA0 ; 加载MPU Region属性及容量寄存器的地址

STR R0, [R1] ; 填写之

POP {R0-R1, PC} ; 返回

MpuRegionDisable

; 该子程序用于除能一个region

; 入口条件： R0 = 待除能的region号

PUSH {R1, LR}

AND R0, R0, #0xF ; region号只取低4位

ORR R0, R0, #0x10 ; 设置VALID位

LDR R1, =0xE000ED9C ; 加载MPU Region 基址寄存器的地址

STR R0, [R1] ; 填写之

MOV R0, #0

LDR R1, =0xE000EDA0 ; 加载MPU Region 属性及容量寄存器的地址

STR R0, [R1] ; 把它归零，这也蕴涵了除能的命令

POP {R1, PC} ; 返回

在本例中，我们还添加了一个用于除能和“复位”无用region的子程序。当你不知道某个region是否被用过时，使用它来使其“归零”是最安全不过的了。

注意代码中位段操作的几行，想想看，如果用普通的移位和数据传送指令，将会繁琐成什么样子！

## 14.4 MPU的典型设置

在典型的情况下，当需要阻止用户程序访问特权级的数据和代码时，可以启用MPU。在设计MPU regions时，需要考虑到下列的regions：

1. 代码region
   1. 特权极代码，包括初始的向量表
   2. 用户级代码
2. SRAM region
   1. 特权级数据，包括主堆栈
   2. 用户级数据，包括进程堆栈
   3. 特权级位带别名区
   4. 用户级位带别名区
3. 外设
   1. 特权级外设
   2. 用户级外设
   3. 特权级外设的位带别名区
   4. 用户级外设的位带别名区
4. 系统控制空间（NVIC以及调试组件）
   1. 仅允许特权级访问

看，上面列出了11个region，已经超出了MPU支持的最多8个，这可如何是好？不怕，还记得有个“背景region”吗？（忘了的话快去看图14.1）。我们可以把所有的特权级regions都归入背景region中（PRIVDEFENA=1）。这样一来，就只需要明确定义用户级的regions——才5个。剩下3个后备的“槽”，可以用于在外部RAM中（如果有的话）设置额外的regions，也可以用于保护只读数据，还能用于“没收”一部分的RAM等，总之这是大虾们绽放智慧光芒的地方。

### 14.4.1 使用子region除能的示例

在上面的分析中，外设是对用户开放的。但是如果误用某个外设可能导致严重后果的话，我们就需要禁止用户级程序随意访问它。这样一来，就会从外设存储器空间中割下几块肉，使一个完整的空间变成若干个更小的。对付这种情况，有如下的办法：

* + 定义多个用户级外设regions
  + 在用户级外设region中重叠地定义一个特权级的region
  + 在用户级外设region中启用“子region除能机制”。

前两个办法很容易耗尽宝贵的8个“region槽”。芯片在设计时应为每个外设都开出相同容量的外设空间（用不完的就空着），这样才能让开发者容易使用第3种方法，即除能子region。通过除能子region，就很容易地从用户级region中擦掉一部分，让它回到背景region中了。一个例子如图14.3所示

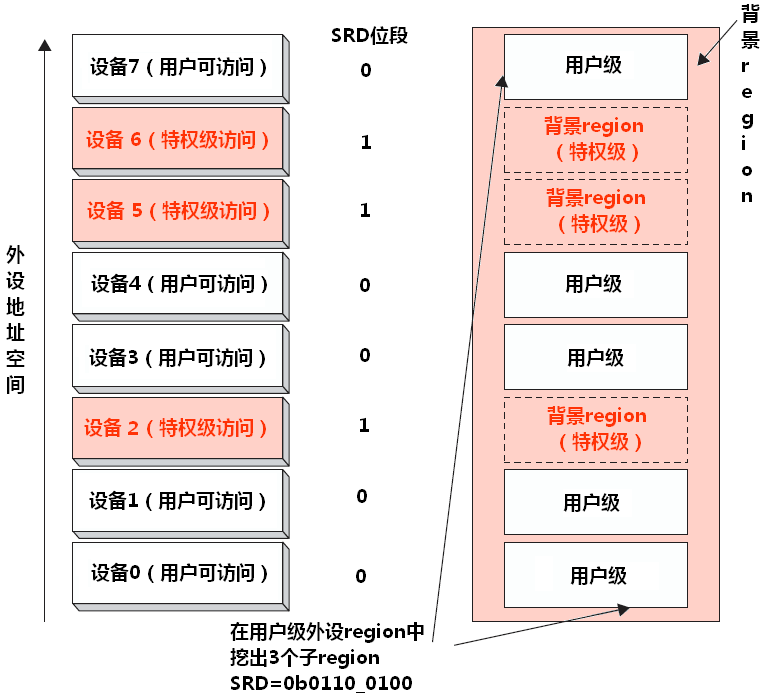


图14.3 “前景”的用户级region被SRD挖出子regions

该技巧也可以用在普通的存储器regions中，但这会使程序更加复杂，所以最好不要玩飘——通常是外设才需要此功能。在使用时，只要把上例中的子region除能参数改为非0即可，如

MOV R5, #0x64 ; Region 1: 子region 2, 5, 6被除能

最后，再根据上一个例子的框架，举一个可能在实际的单片机会出现的例子：

MpuSetup ; 入口函数，它内部呼叫若干子程序来完成MPU设置

PUSH {R0-R6,LR}

LDR R0, =0xE000ED94 ; MPU 控制寄存器

MOV R1, #0

STR R1, [R0] ; 配置前先除能MPU

; --- Region #0 --- 用户级程序

LDR R0, =0x00004000 ; Region 0: 基址 = 0x00004000

MOV R1, #0x0 ; Region 0: Region号 = 0

MOV R2, #0x0D ; Region 0: 容量 = 0x0D (16KB)

MOV R3, #0x3 ; Region 0: AP = 0x3 ( 全访问)

MOV R4, #0x2 ; Region 0: TEX=0,S=0,C=1,B=0

MOV R5, #0x0 ; Region 0: 子region除能=0

MOV R6, #0x1 ; Region 0: {XN, Enable} = 0,1

BL MpuRegionSetup

; --- Region #1 --- 用户级数据

LDR R0, =0x20000000 ; Region 1: 基址 = 0x20000000

MOV R1, #0x1 ; Region 1: Region号 = 1

MOV R2, #0x0B ; Region 1: 容量 = 0x0B (4KB)

MOV R3, #0x3 ; Region 1: AP = 0x3 (全访问)

MOV R4, #0xB ; Region 1: TEX=1,S=0,C=1,B=0

MOV R5, #0x0 ; Region 1: 子region除能= 0

MOV R6, #0x1 ; Region 1: {XN, Enable} =0,1

BL MpuRegionSetup

; --- Region #2 --- 用户级位带别名区

LDR R0, =0x22000000 ; Region 2: 基址 = 0x22000000

MOV R1, #0x2 ; Region 2: Region号 = 2

MOV R2, #0x10 ; Region 2: 容量 = 0x010 (128KB)

MOV R3, #0x3 ; Region 2: AP = 0x3 (全访问)

MOV R4, #0xB ; Region 2: TEX=1,S=0,C=1,B=0

MOV R5, #0x0 ; Region 2: 子region除能= 0

MOV R6, #0x1 ; Region 2: {XN, Enable} =0,1

BL MpuRegionSetup

; --- Region #3 --- 用户级外设

LDR R0, =0x40000000 ; Region 3: 基址 = 0x40000000

MOV R1, #0x3 ; Region 3: Region号 = 3

MOV R2, #0x13 ; Region 3: 容量 = 0x013 (1MB)

MOV R3, #0x3 ; Region 3: AP = 0x3 (全访问)

MOV R4, #0x1 ; Region 3: TEX=1,S=0,C=1,B=0

**MOV** R5, #0x64 ; Region 3: 子region 2,5,6除能

MOV R6, **#0x3**  ; Region 3: {**XN**, Enable} =**1**,1

BL MpuRegionSetup

; --- Region #4 --- 用户级外设的位带别名区

LDR R0, =0x42000000 ; Region 4: 基址 = 0x42000000

MOV R1, #0x4 ; Region 4: Region号 = 4

MOV R2, #0x18 ; Region 4: 容量 = 0x018 (32MB)

MOV R3, #0x3 ; Region 4: AP = 0x3 (全访问)

MOV R4, #0x1 ; Region 4: TEX=1,S=0,C=1,B=0

**MOV** R5, #0x64 ; Region 4: 子region 2,5,6除能

MOV R6, **#0x3**  ; Region 4: {**XN**, Enable} =**1**,1

BL MpuRegionSetup

; --- Region #5 --- 外部RAM

LDR R0, =0x60000000 ; Region 5: 基址 = 0x60000000

MOV R1, #0x5 ; Region 5: Region号 = 5

MOV R2, #0x17 ; Region 5: 容量 = 0x010 (16MB)

MOV R3, #0x3 ; Region 5: AP = 0x3 (全访问)

MOV R4, #0xB ; Region 5: TEX=0,S=0,C=1,B=1

MOV R5, #0x0 ; Region 5: 子region除能= 0

MOV R6, #0x1 ; Region 5: {XN, Enable} =0,1

BL MpuRegionSetup

; --- Region #6 --- 未使用，把它归零

MOV R0, #6

BL MpuRegionDisable

; --- Region #7 --- 未使用，把它归零

MOV R0, #7

BL MpuRegionDisable

（原文中，上例加灰的指令把SRD设置为0x9B，即~0x64，看起来似乎是SRD的某个位为零时才除能对应的子region，与图14.3中给出的二进制数值相反。在译者查阅其它资料后仍然不能确定时，就求助于ARM了。感谢ARM的姜宁先生为译者肯定了正确的答案！——译者注）。

上例的代码执行后，建立的regions如下表所示（假设单片机有32KB flash, 8KB RAM）：

表14.8 上例代码执行后建立的各regions

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **地址范围** | **容量** | **类型** | **存储器属性**  **C,B,A,S,XN** | **MPU region** | **说明** |
| 0000\_0000至  0000\_3FFF | 16KB | RO | C,-,A,-,- | 背景 | 特权级程序 |
| 0000\_4000至  0000\_7FFF | 16KB | RO | C,-,A,-,- | Region #0 | 用户级程序 |
| 2000\_0000至  2000\_0FFF | 4KB | RW | C,B,A,-,- | Region #1 | 用户级数据 |
| 2000\_1000至  2000\_1FFF | 4KB | 特权极RW | C,B,A,-,- | 背景 | 特权级数据 |
| 2200\_0000至  2001\_FFFF | 128KB | RW | C,B,A,-,- | Region #2 | 用户级数据的位带别名区 |
| 2202\_0000至  2203\_FFFF | 128KB | 特权极RW | C,B,A,-,- | 背景 | 特权级数据的位带别名区 |
| 4000\_0000至  400F\_FFFF | 1MB | RW | -,B,-,-,XN | Region #3 | 用户级外设 |
| 4004\_0000至  4005\_FFFF | 128KB | 特权级RW | -,B,-,-,XN | 背景  Region #3中被除能的子region 2 | 在用户级外设地址范围中的特权级外设 |
| 400A\_0000至  400B\_FFFF | 128KB | 特权级RW | -,B,-,-,XN | 背景  Region #3中被除能的子region 5 | 在用户级外设地址范围中的特权级外设 |
| 400C\_0000至  400D\_FFFF | 128KB | 特权级RW | -,B,-,-,XN | 背景  Region #3中被除能的子region 6 | 在用户级外设地址范围中的特权级外设 |
| 4200\_0000至  43FF\_FFFF | 32MB | RW | -,B,-,-,XN | Region #4 | 用户级外设的位带别名区 |
| 4280\_0000至  42BF\_FFFF | 4MB | 特权级RW | -,B,-,-,XN | 背景  Region #4中被除能的子region 2 | 在用户级外设位带别名区地址范围中的特权级外设 |
| 4340\_0000至  437F\_FFFF | 4MB | 特权级RW | -,B,-,-,XN | 背景  Region #4中被除能的子region 5 | 在用户级外设位带别名区地址范围中的特权级外设 |
| 4380\_0000至  43BF\_FFFF | 4MB | 特权级RW | -,B,-,-,XN | 背景  Region #4中被除能的子region 6 | 在用户级外设位带别名区地址范围中的特权级外设 |
| 6000\_0000至  60FF\_FFFF | 16MB | RW | C,B,A,-,- | Region #5 | 外部RAM |
| E000\_0000至  E00F\_FFFF | 1MB | 特权级 | -,-,-,-,XN | 背景 | NVIC,调试组件，以及  私有外设总线 |