Event System？

Interrupt System？

软硬兼施的事件驱动系统

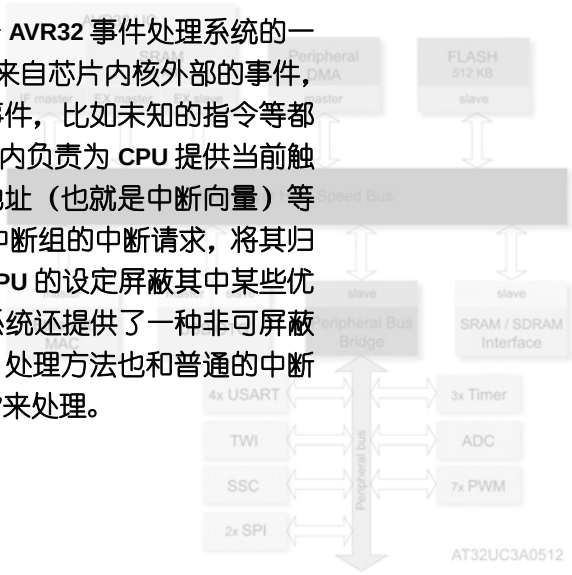
基于消息地图的中断管理

>> Interrupt Controller

第四章

本章引言

AVR32 UC3 中断控制器属于 **AVR32** 事件处理系统的一个子集。根据官方的定义，所有来自芯片内核外部的事件，都被称为中断。而来自内核的事件，比如未知的指令等都属于异常。**AVR32** 中断控制器对内负责为 **CPU** 提供当前触发中断的等级、中断处理程序地址（也就是中断向量）等信息；对外负责收集来自 **64** 个中断组的中断请求，将其归类为 4 个中断优先级，并按照 **CPU** 的设定屏蔽其中某些优先级的中断请求。**AVR32** 中断系统还提供了一种非可屏蔽中断，这类中断的优先级最高，处理方法也和普通的中断不同，通常情况下可以视作异常来处理。



AVR32.UC3.Interrupt.Summary 综述

8.1 Interrupt Controller 中断控制器简介

AVR32 UC3 的 **INTC**（**Interrupt Controller**）中断控制器是一个高度灵活性的模块，用户可以自行配制中断请求的优先级，支持中断嵌套。并且 **INTC** 在硬件上对各级中断的处理延时进行了优化，做到了中断快进快出。

A、支持 **4** 组中断优先级。

B、支持 **64** 组，每组 **32** 个中断号，最多 **2048** 个中断请求。

C、可独立屏蔽任何中断

D、支持独立的不可屏蔽中断（**NMI**）

E、支持中断嵌套

F、硬件中断现场保护，快速中断处理

AVR32.UC3.Interrupt.Feature 产品特性

8.2.1 功能强大的中断控制器

中断控制器（**INTC**）是 **AVR32 UC3** 处理外设事件的核心部件，它对所有的来自外设的中断进行仲裁，并通过硬件自动完成中断处理的关键步骤。

INTC 将所有硬件已分配好的中断号，分为 **64** 组，每组分配 **32** 个中断号，所以 **INTC** 最多可支持 **64X32=2048** 个中断。并且所有的中断号可再行设定 **4** 个优先级，这样既兼顾了软件操作中断优先级的灵活性也大大降低了人为设置所有中断优先级的复杂性。

另外，**AVR32 UC3** 还支持不可屏蔽中断以及中断嵌套，为各种应用提供了最大的灵活性。

8.2.2 快速中断处理机制

中断控制器的 **4** 级优先级以及不可屏蔽中断都有独立的寄存器页，并且在硬件自动处理所有的现场保护操作，因此可以快速处理中断。

AVR32 UC3 的中断进入过程共分为 **3** 个部分。

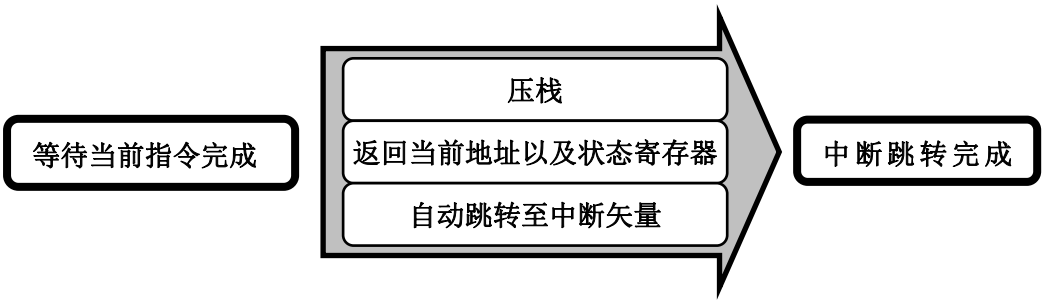


图 **8.2.1** 中断进入过程

在第一个部分中，如果当前的指令是 **ldm, stm, pushm, popm, divs** 和 **divu** 这些执行超过 **5** 个周期的指令时，中断控制器会放弃继续执行这个指令，并跳入下一步，因为在等待完成指令的部分，最多需要 **5** 个时钟周期，而第二部分是固定的 **10** 个周期，最后需要 **1** 个周期来往成中断跳转。因为，对于普通的中断 **AVR32 UC3** 最多需要 **16** 个时钟周期就可完成，而最短 **12** 个就可以完成（当前被打断的指令是单周期指令）。

另外，因为 **NMI** 中断是独立的中断寄存器页，并且不用处理 **R8-R12** 寄存器的压栈操作。所以只需要 **4** 个时钟周期即可完成。因此 **AVR32 UC3** 的 **NMI** 中断最少只需要 **6** 个周期即可进入中断。

AVR32.UC3.Interrupt.Interface 接口与功能概述

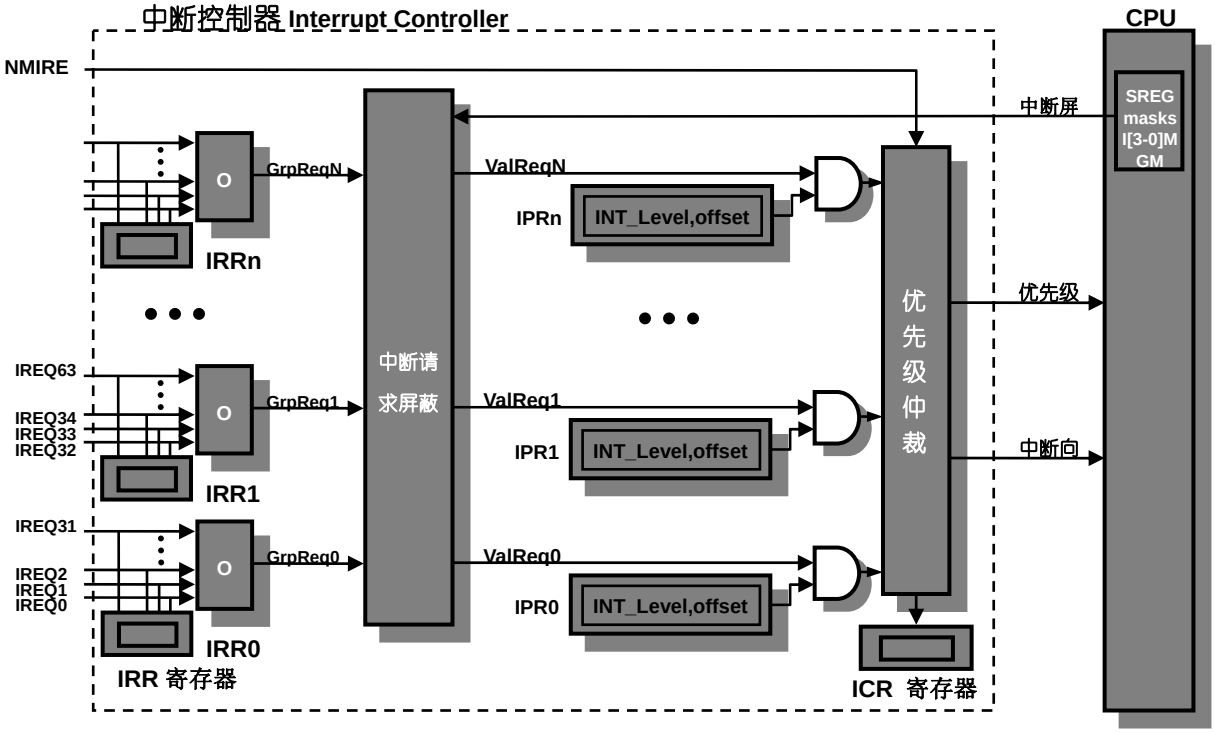


图 8.3.1 中断控制器内部模块逻辑结构示意图

8.3 精兵简政的中断管理策略

AVR32 UC3 中断控制器的中断管理策略非常简单：**64** 个中断组，每个中断组可以从 **4** 个优先级中任意指定一个中断优先级；每个中断组都可以指定一个专门的中断处理程序；每个中断组中最大可以包含 **32** 个中断源。以组（**Group**）而不是单个中断源为单位进行管理，是 **AVR32 UC3** 中断控制器的核心思想。

8.3.1 组内中断管理

理论上每一个中断组都可以最大容纳 **32** 个中断源，这是由 **32** 系统中寄存器的宽度所决定的。实际情况下，**AVR32 UC3** 系列的每一个型号都是以归类存放的思想对中断进行分组的，例如所有的 **GPIO** 中断都被放在同一个组中。这就造成了每个组内实际不同的中断源数量，很多组内往往只有一个中断源，例如 **AVR32 UC3A0** 系列的 **USARTn**，**4** 个 **USART** 都独立的占用了一个中断组。根据器件的不同，详细地中断分组情况请查阅对应的数据手册。对应的内容在数据手册 **Peripherals** 章节的 **Interrupt Request Signal Map** 中。表 **8.3.1** 中列举的是 **UC3A0** 和 **UC3A1** 系列微处理器的中断请求分组情况。

表 **8.3.1 AVR32 UC3A0/A1** 系列中断请求分组

组	组内请求	中断源隶属模块	中断信号名称
0	**0**	**AVR32 UC CPU with optional MPU and optional OCD**	**SYSBLOCK COMPARE**
1	**0**	**External Interrupt Controller**	**EIC0**

	1	External Interrupt Controller	EIC1
	2	External Interrupt Controller	EIC2
	3	External Interrupt Controller	EIC3
	4	External Interrupt Controller	EIC4
	5	External Interrupt Controller	EIC5
	6	External Interrupt Controller	EIC6
	7	External Interrupt Controller	EIC7
	8	Real Time Counter	RTC
	9	Power Manager	PM
	10	Frequency Meter	FREQM
2	0	General Purpose Input/Output	GPIO0
	1	General Purpose Input/Output	GPIO1
	2	General Purpose Input/Output	GPIO2
	3	General Purpose Input/Output	GPIO3
	4	General Purpose Input/Output	GPIO4
	5	General Purpose Input/Output	GPIO5
	6	General Purpose Input/Output	GPIO6
	7	General Purpose Input/Output	GPIO7
	8	General Purpose Input/Output	GPIO8
	9	General Purpose Input/Output	GPIO9
	10	General Purpose Input/Output	GPIO10
	11	General Purpose Input/Output	GPIO11
	12	General Purpose Input/Output	GPIO12
	13	General Purpose Input/Output	GPIO13
3	0	Peripheral DMA Controller	PDCA0
	1	Peripheral DMA Controller	PDCA1
	2	Peripheral DMA Controller	PDCA2
	3	Peripheral DMA Controller	PDCA3
	4	Peripheral DMA Controller	PDCA4
	5	Peripheral DMA Controller	PDCA5
	6	Peripheral DMA Controller	PDCA6
	7	Peripheral DMA Controller	PDCA7
	8	Peripheral DMA Controller	PDCA8
	9	Peripheral DMA Controller	PDCA9
	10	Peripheral DMA Controller	PDCA10
	11	Peripheral DMA Controller	PDCA11
	12	Peripheral DMA Controller	PDCA12
	13	Peripheral DMA Controller	PDCA13
	14	Peripheral DMA Controller	PDCA14
4	0	Flash Controller	FLASHC
5	0	Univerasl Synchronous/Asynchronous Receiver/Transmitter	USART0
6	0	Univerasl Synchronous/Asynchronous Receiver/Transmitter	USART1
7	0	Univerasl Synchronous/Asynchronous Receiver/Transmitter	USART2
8	0	Univerasl Synchronous/Asynchronous Receiver/Transmitter	USART3
9	0	Serial Peripheral Interface	SPI0
10	0	Serial Peripheral Interface	SPI1
11	0	Two-wire Interface	TWI
12	0	Pulse Width Modulation Controller	PWM
13	0	Synchronous Serial Controller	SSC
14	0	Timer/Counter	TC0
	1	Timer/Counter	TC1

	2	Timer/Counter	TC2
15	0	Analog to Digital Converter	ADC
16	0	Ethernet MAC	MACB
17	0	USB 2.0 OTG Interface	USBB
18	0	SDRAM Controller	SDRAMC
19	0	Audio Bistream DAC	DAC

当某一个组内有中断请求发生时，中断控制器通过一个 **32** 位的寄存器 **IRR** 来告知用户此时刻组内有哪些中断源发出了中断请求。**IRR** 是 **Interrupt Request Regiser** 的缩写，只是一个中断控制器用来通知用户当前组内中断源状况的寄存器，因而是只读属性，任何对其进行的写入操作都是无效的。**IRR** 寄存器的每一个二进制位对应一个组内中断源的状态，对应位为“**1**”表示该中断源发出了中断请求，并且发出该中断请求的中断源还没有被清除。

在一个中断组内，当同时有多个中断请求发生时，用户需要自己通过软件来处理优先级的问题，虽然可以按照用户自己的喜好来安排组内的优先级，但是推荐的优先顺序是：编号较大的优先处理——这与后面多个不同优先级中断同时发生决定优先级的方式是相同的。如果你直接使用 **Software Framework** 来处理中断系统，那么中断消息处理程序会自动选择编号最大的中断请求来进行处理。

8.3.2 中断优先级管理

AVR32 UC3 系列的中断系统只有 **4** 个优先级，分别是 **INT0**、**INT1**、**INT2** 和 **INT3**。数字越大优先级越高。在系统状态寄存器 **SR** 中，除了标准的全局中断屏蔽标志 **GM** 以外，还有 **4** 个独立的中断优先级屏蔽标志 **InM**（分别是 **I0M**、**I1M**、**I2M** 和 **I3M**），用于独立的屏蔽来自 **4** 个优先级的中断。可以说，在 **AVR32 UC3** 系列芯片中，中断管理的灵魂就在于这 **4** 个独立的中断屏蔽寄存器。

为了详细说明 **InM** 在中断优先级处理中的核心作用，我们按照普通的中断处理观点，做一个假设：

a、 当同时存在多个中断请求时，系统只响应优先级最高的中断请求。

b、 系统在响应某一个中断请求时会自动把系统状态寄存器 **SR** 中的全局中断屏蔽标志 **GM** 置位，防止中断嵌套的发生。当系统退出中断处理程序时，会自动把 **GM** 标志清除。用户也可以在中断处理程序中人为的清除 **GM** 标志，实现对中断嵌套的支持——也就是低优先级的中断可以打断高优先级的中断。

如果熟悉 **AVR 8** 位单片机，你会发现上面描述的中断处理策略正是应用在 **AVR 8** 位平台上的中断优先级处理策略。对于 **AVR32 UC3** 平台来说，中断处理只是整个系统事件系统的一个子集，在事件系统中，存在一个事件队列，针对这个队列实际上有一个规则：当系统因为 **1** 条机器指令而同时触发多个事件时，将按照事件的优先级顺序依次处理各个事件——也就是说，当一条机器指令触发了多个事件时，这些事件是按照优先级顺序依次进入事件队列的。当系统因为不同的机器指令触发多个事件时，这些事件将按照发生的先后被处理。此时事件之间的优先级是不被考虑的——简单来说，就是对于有多个不同指令产生的事件，将按照发生的先后顺序进入事件处理队列，并不会发生高优先级抢先进入队列的情况。

中断处理是事件处理的一个子集，那么一条机器指令会同时触发多个中断么？如果是，那么这些中断将按照优先级顺序被处理，否则先发生的中断将先被处理，后来的中断即便是高优先级也没有机会去打断低优先级中断处理程序的执行（全局中断标志 **GM** 在处理低优先级中断时被自动置位了）。这里听起来似乎有一些不合理，因为如果简单的通过在低优先级

中断处理程序中打开 **GM** 标志位，那么比该低优先级中断更低优先级的中断也能做越权的事情。这种中断处理策略的特点是简单，但是无法应对复杂的中断处理策略。这也是为什么在 **8** 位 **AVR** 系统中，要么不允许中断嵌套，要么需要仔细面对可能的堆栈溢出的危险。

AVR32 UC3 中断系统实际上采用了一种与前面方法类似但是发展了的做法：在 **SR** 寄存器中增加了对应于中断优先级的屏蔽标志 **InM**。当处理对应优先级的中断时，中断控制器不会去屏蔽全局中断响应标志 **GM**，而是自动将对应等级及其以下等级的中断优先级标志屏蔽。例如，如果系统响应了 **INT2**，会自动将 **I2M**、**I1M** 和 **I0M** 标志统一进行屏蔽，防止发生低优先级的“嵌套”，同时由于 **I3M** 标志没有被屏蔽，**INT3** 中断可以打断当前的中断处理过程。由于用户可以通过软件的方法在中断处理程序中人为的设置或者清除 **InM** 标志，因而可以获得更为灵活“嵌套方案”。

在这种基于优先级独立屏蔽的策略下，全局中断响应标志 **GM** 担当着一票否决的重任，在类似 **μc/os II** 的嵌入式操作系统中可以用于建立中断临界区：

```
# define OS_ENTER_CRITICAL()                    \
        ({__asm__ __volatile__ ("ssrf\t%0" :   : "i" (AVR32_SR_GM_OFFSET));})
# defien OS_EXIT_CRITICAL()                     \
        ({__asm__ __volatile__ ("csrf\t%0" :   : "i" (AVR32_SR_GM_OFFSET));})
```

AVR32 UC3 的中断管理是以组为单位的，每一个中断组都可以独立的指定中断优先级，并为改组指定一个 **14** 位的“组中断（相对）向量（**Autovector**）”。可以认为，**AVR32 UC3** 拥有最大 **64** 个硬件组中断，当组内有任何中断请求时，只要该请求被响应，系统都会执行由 **14** 位“组中断（相对）向量”所指定的“中断（组）处理程序”。中断的优先级是以组为单位分配的，所以同组内的中断拥有相同的优先级，以相同的“中断（组）处理程序”来负责处理。中断组的优先级和中断（相对）向量是通过寄存器 **IPR** 来设置的。其中 **IPR** 是 **Interrupt Priority Register** 的缩写。

由于中断优先级和中断组是多对多的映射关系，所以每个中断优先级都有一个寄存器 **ICR** 来告知用户在该优先级下，当前是哪个中断组触发了中断。**ICR** 中保存的是一个唯一的组编号。当同一优先级下有多个组同时发出中断请求时，编号最小的组将获得响应。**ICR** 是一个只读寄存器。

8.3.3 中断响应

当一个中断源发出中断请求时，中断控制器会首先将设置中断源所在组的 **IRR** 寄存器；在同一个组内，所有的中断请求是以逻辑“或”的关系触发组中断请求的——即，只要中断组内任意一个中断源发出请求都会触发组中断请求，如图 **8.3.1** 所示。

当一个组发出中断请求时，系统会根据该组的 **IPR** 寄存器获取优先级信息，并根据这一信息检查对应优先级的中断屏蔽标志 **InM**——如果 **InM** 被置位，则本次组中断请求将被直接忽略。对某一个优先级来说，可能有多个组同时发出中断请求，此时编号最小的组将得到响应——该组的组编号将被复制到对应优先级的 **ICR** 寄存器中，用以指示是哪个组触发了该优先级的中断。

当一个优先级发出中断请求时，系统会根据当前同时发出请求的优先级高低决定最终送

向 **CPU** 的中断请求。数字越大优先级越高。除了基本的 **4** 个中断优先级以外，这一层处理中还有一个优先级最高的非可屏蔽中断 **NMI**（**None Maskable Interrrupt**）。该中断是不可屏蔽的，不仅因为它没有对应的中断屏蔽标志，更重要的是从图 **8.3.1** 中我们容易发现，**CPU** 对全局中断的屏蔽请求（**GM** 标志）是针对“中断请求屏蔽”单元进行的，而 **NMI** 中断请求是直接送达“优先级仲裁”单元的。简单来说，全局中断屏蔽标志对 **NMI** 来说也是不起作用的。

当一个中断请求被送往 **CPU** 时，只包含两个信息：触发中断的优先级，以及对应的（组）中断 IPR 寄存器中所保存的（相对）中断向量。对于传入的优先级信息，**CPU** 会首先检查事件系统中在同一时刻是否存在比当前中断优先级更高的事件发生，比如 **NMI** 和异常。如果存在，本次中断请求将被延迟处理；否则，应中断优先级及其以下优先级的屏蔽标志 **InM** 将被自动设置。

AVR32 UC3 系统中一个中断处理程序其完整的地址应该是 **32** 位的，而 **IPR** 中记录的中断向量只有 **14** 位。这是一个相对“地址空间中名叫 **EVBA** 的程序段起始地址”的相对偏移量——将 **IPR** 中的 **14** 位 **Autovector** 与 **EVBA** 的地址相加才能获得实际的中断处理程序地址，而这个相加的过程是由 CPU 自动完成的。由于 **14** 地址能表达的偏移范围仅仅是 **0~16KB**，这就限制了中断处理程序必须被放置在 **EVBA** 开始的 **16K** 地址空间中。**NMI** 中断的中断向量是一个相对于 **EVBA** 起始地址的固定偏移量，其操作方式与异常相同。

对于传入的中断请求，**CPU** 会按照事件处理的方法自动将 **PC**、**SR** 寄存器的内容压入栈中。对于普通中断，事件系统还会额外的将通用寄存器 **R8~R12** 以及寄存器 **LR** 的内容压入栈中。对于非可屏蔽中断 **NMI** 来说，并不会存在这些额外的压栈操作。对于普通中断，事件系统会将 **EVBA** 的地址与（相对）中断向量 **Autovector** 的值相加计算出实际的中断处理程序地址，并将 **PC** 指针指向对应位置；对于非可屏蔽中断 **NMI**，事件系统会根据 **EVBA** 的地址增加一个固定的偏移量以后设置 **PC** 指针。

8.3.3 用户中断处理

无论是普通中断还是非可屏蔽中断，都需要用户在处理程序的最后显式的调用汇编指令“**rete**”以实现将保存在堆栈中的内容弹出，恢复事件处理前的系统运行状态。针对这种情况 **Software Framework** 为用户提供了 **2** 种不同的应对策略，具体内容将在后面的章节中详细给出。

对于触发中断的中断源，用户也需要在中断处理程序中通过软件人为的进行清除：比如通过适当的写入操作清除外设中断标志寄存器 IFR 中的中断请求标志等等。为了保险起见，我们应该在实现中断标志清除的写入操作后面增加一个对同一寄存器的读取操作。这一读取操作能够防止同一中断源的误触发。

这是因为，对中断标志清除寄存器的写入操作，从发出指令到实际生效往往存在一定的延时，而在这个延时的过程中，中断处理程序如果已经退出，系统会自动将对应优先级的中断屏蔽标志删除，那么先前还没有来得及清除的中断标志将导致一次意想不到的中断误触发。插入的读取操作能够确保前面的写入操作生效，因而保证中断请求被及时的清除。

AVR32.UC3.Interrupt.API　　硬件驱动

8.4.1 数据结构

AVR32 UC3 系列微处理器支持 **4** 个中断优先级、**64** 个硬件（组）中断向量。当一个中断发生时，系统会直接处理组一级别的中断处理程序。在这一处理程序中，用户应该根据改组的中断请求寄存器 **IRR** 查询究竟是哪个中断源触发了组中断，并根据这一信息调用相关的中断处理函数或者直接进行某些处理。简单说来，当一个中断被响应时，中断控制器只能为我们完成精确到组的中断源定位，对于组内具体的中断源查询需要用户自己编写软件来完成。

在本书中，我们将中断系统中由硬件完成的到组一级别的中断响应称为“初级中断寻址”；将中断组内需要用户编写软件进行定位和处理的中断响应成为“二次中断寻址”；对于经过二次寻址定位到的具体中断源，还需要处理程序查询模块内的中断标志以定位模块内实际发生的中断源的处理过程称为“三次中断寻址”。例如，当一个请求被响应时，系统会通过初次中断寻址定位到具体的组，用户通过编写软件实现二次中断寻址，以确定组内具体是哪个中断源触发了中断；假设系统发现这是一个串口中断，那么在调用对应的中断处理程序时，还需要通过串口模块内的中断寄存器来确定触发中断响应的究竟是串口发送完成中断还是串口接收完成中断……，这一过程就被称为三次中断寻址。

为了简化整个中断处理流程，**Software Framwork** 将初次中断寻址和二次中断寻址的部分封装了起来，通过消息地图的方法进行处理。当一个中断被响应时，对应中断组 **IRR** 寄存器中的内容将被作为输入消息，从对应组的消息地图中查询到对应中断的处理程序，并自动调用该处理程序。每个中断源的中断处理程序，以及实现三次中断寻址的程序则交由用户自己处理了。

8.4.1.1 寄存器接口

当我们通过编译预处理指令 **#include<avr32/io.h>** 为系统增加了对所选芯片的基本输入输出支持时，实际上将根据工程中所设定的芯片型号选择对应的器件头文件 **uc3xxxxx.h**，例如 **uc3a0512.h**。在这类头文件中，系统定义了一系列与 **INTC** 模块相关的常量，其中包括 **INTC** 在地址空间中的位置：

```
#include "avr32/intc.h"
#define AVR32_INTC_NUM                    1
#define AVR32_INTC_ADDRESS                0xFFFF0800
#define AVR32_INTC                (*((volatile avr32_intc_t*)AVR32_INTC_ADDRESS))
#define AVR32_INTC_CLK_PBA                64
#define AVR32_INTC_NUM_INT_GRPS           20
#define AVR32_INTC_NUM_IRQS_PER_GRP0      1
#define AVR32_INTC_NUM_IRQS_PER_GRP1      11
……
```

在这些宏中，**AVR32_INTC_ADDRESS** 定义了 **INTC** 模块在设备总线 **PBA** 上的地址；宏 **AVR32_INTC** 则将该地址绑定为 **avr32_intc_t** 类型的结构体。**avr32_intc_t** 是在头文件 **avr32/intc.h** 中定义的，用以描述 **AVR32 UC3** 中断控制器的寄存器接口，具体的寄存器接口如表 **8.4.1** 所示。

地址偏移	寄存器	名称	读写权限	复位值
0x0000	Interrupt Priority Register 0	IPR0	读/写	0x0000_0000
0x0004	Interrupt Priority Register 1	IPR1	读/写	0x0000_0000

…	…	…	…	…
0x00FC	Interrupt Priority Register 63	IPR63	读/写	0x0000_0000
0x0100	Interrupt Request Register 0	IRR0	只读	N/A
0x0104	Interrupt Request Register 1	IRR1	只读	N/A
…	…	…	…	…
0x01FC	Interrupt Request Register 63	IRR63	只读	N/A
0x0200	Interrupt Cause Register 3	ICR3	只读	N/A
0x0204	Interrupt Cause Register 2	ICR2	只读	N/A
0x0208	Interrupt Cause Register 1	ICR1	只读	N/A
0x020C	Interrupt Cause Register 0	ICR0	只读	N/A

表 8.4.1 AVR32 UC3A 用户接口寄存器一览表

在 **avr32/intc.h** 中相关的定义如下：

```
    /* 访问寄存器IPRn所使用的自定义位段类型 */
typedef struct avr32_intc_ipr_t {
    unsigned int intlev          : 2;
    unsigned int                 :16;
    unsigned int offset          :14;
} avr32_intc_ipr_t;

/* 访问寄存器ICR所使用的自定义位段类型 */
typedef struct avr32_intc_icr_t {
    unsigned int                 :26;
    unsigned int causegrp        : 6;
} avr32_intc_icr_t;

/* 中断控制器寄存器用户接口 */
typedef struct avr32_intc_t {
  union {
        unsigned long        ipr        [64];//0x0000
        avr32_intc_ipr_t     IPR        [64];
  };
    const unsigned long      irr        [64];//0x0100
  union {
    const unsigned long      icr        [4];//0x0200
    const avr32_intc_icr_t   ICR        [4];
  };
} avr32_intc_t;
```

其中 **avr32_intc_icr_t** 和 **avr32_ictc_ipr_t** 分别是系统用来访问 **ICRn** 和 **IPRn** 寄存器所使用位段，在不使用 **Software Framework** 的情况下，用户可以通过这些系统自定义结构实现对中断控制器的访问，例如：

```
    //设置中断组0的优先级为INT3
    AVR32_INTC.IPR[0].lev = 3;
```

一般情况下，使用 **Software Framework** 实现中断管理，我们并不需要关心这些寄存器是如何被访问的。

8.4.1.2 中断事件处理系统

AVR32 Software Framework 为用户封装了一个中断事件处理系统。用户只需要对所要处理的中断源和对应的中断处理程序进行一次注册，并在注册时制定该中断源所在中断组的优先级就可以完成对中断的管理。对于中断控制器的具体设置和内部实现则不需要用户关心。那么这种中断管理系统是如何实现的呢？既然 **Software Framework** 已经为我们将这些具体的实现过程隐藏了起来，为什么我们还要去了解这一过程呢？答案很简单，因为在应对某些极端需求时现有的系统存在优化的可能。

首先明确三个要点：

1、 中断控制器支持 **64** 个中断组；每个中断组都有一个独立可设（相对）中断向量。
2、 系统对中断的相应在硬件上只支持“初次中断寻址”，对组内具体中断源的处理需要用户在（组）中断处理程序中编写对应的代码。
3、 每个组的（相对）中断向量只有 **14** 位宽度，因此（组）中断处理程序只能被安排在 **EVBA** 开始 **16K** 字节的范围内。

实际用户使用的中断管理系统具有以下特点：

1、 用户只知道所有的中断源都可以为之制定一个中断处理函数，当相应的中断源触发中断时，中断处理函数会被系统“自动”调用。
2、 每个中断组内所有中断源都使用同一个中断优先级。对某个组来说，最后一次注册中断时所制定的优先级将立即生效并取代先前的优先级设定。
3、 用户自己指定的中断处理函数并没有任何限定其生成代码存放位置的特殊设置，用户甚至并不需要知道 **EVBA** 和 **16KB** 代码区域范围限制的存在。

对比中断控制器的特性以及在 **Software Framework** 中实际使用的中断管理策略，可以发现用户使用的中断管理接口更简单易用。那么中间究竟存在如何的转换过程以满足中断控制器的特殊要求呢？

首先，在 **Software Framework** 在 **DRIVERS\INTC** 目录中除了常见的 **intc.c** 和 **intc.h** 以外，还有一个 **exception.S**（对 **IAR** 的使用者来说，这个文件是 **exception.S82**）。这是汇编源程序文件。**Software Framework** 在汇编源文件里定义了一个 **EVBA** 区段，并在肯定满足 **16K** 字节的空间范围内定义了 **4** 个特殊的代码段，分别是 **int3**、**int2**、**int1** 和 **int0**，如下所示：

```
//定义EVBA区段
.global _evba
.type _evba, @function
_evba:
        ……

  .balign 4
//定义四个代码段分别对应四个优先级
```

```
_int0:
  // R8-R12, LR, PC and SR are automatically pushed onto the system stack by the
  // CPU upon interrupt entry.
  mov       r12, 0   // Pass the int_lev parameter to the _get_interrupt_handler function.
  call      _get_interrupt_handler
  cp.w      r12, 0   // Get the pointer to the interrupt handler returned by the function.
  movne     pc, r12 // If this was not a spurious interrupt (R12 != NULL), jump to the handler.
  rete                // If this was a spurious interrupt (R12 == NULL), return from event handler.

_int1:
  // R8-R12, LR, PC and SR are automatically pushed onto the system stack by the
  // CPU upon interrupt entry.
  mov       r12, 1   // Pass the int_lev parameter to the _get_interrupt_handler function.
  call      _get_interrupt_handler
  cp.w      r12, 0   // Get the pointer to the interrupt handler returned by the function.
  movne     pc, r12 // If this was not a spurious interrupt (R12 != NULL), jump to the handler.
  rete                // If this was a spurious interrupt (R12 == NULL), return from event handler.

_int2:
  // R8-R12, LR, PC and SR are automatically pushed onto the system stack by the
  // CPU upon interrupt entry.
  mov       r12, 2   // Pass the int_lev parameter to the _get_interrupt_handler function.
  call      _get_interrupt_handler
  cp.w      r12, 0   // Get the pointer to the interrupt handler returned by the function.
  movne     pc, r12 // If this was not a spurious interrupt (R12 != NULL), jump to the handler.
  rete                // If this was a spurious interrupt (R12 == NULL), return from event handler.

_int3:
  // R8-R12, LR, PC and SR are automatically pushed onto the system stack by the
  // CPU upon interrupt entry.
  mov       r12, 3   // Pass the int_lev parameter to the _get_interrupt_handler function.
  call      _get_interrupt_handler
  cp.w      r12, 0   // Get the pointer to the interrupt handler returned by the function.
  movne     pc, r12 // If this was not a spurious interrupt (R12 != NULL), jump to the handler.
     rete                // If this was a spurious interrupt (R12 == NULL), return from event handler.
```

先不管四个区段中汇编代码的意义。由于 **INTC** 所支持的 **64** 个中断组都可以独立的指定一个相对中断向量，如果为每一个这样的中断向量都指定一个 **EVBA** 区段内的代码段，不仅工作量巨大，每次注册新的中断处理函数我们都需要使用汇编代码手工修改 **exception.S**。这对使用 **Software Framework** 从事 **C** 语言快速开发的用户来说是不可容忍的，事实上如果对 **AVR32 UC3** 系列系统构架不是非常了解的话，一般程序员也很难完成这项工作。因此，**Software Framework** 使用了一个折中的方法：只建立四个对应四个优先级的代码段，每个中断组将自己的（相对）中断向量都指向对应优先级的代码段，而所有这些过程都是可以通过 **C** 语言代码来完成的，因此我们可以在 **intc.c** 的中断注册函数中看到这样的代码片断：

```
/* 用户使用的中断注册函数 */
void INTC_register_interrupt(__int_handler handler, unsigned int irq, unsigned int int_lev)
{
    /* 根据中断向量编号irq计算出该中断所属的中断组编号 */
    unsigned int int_grp = irq / AVR32_INTC_MAX_NUM_IRQS_PER_GRP;

    ……

    /* 设置中断源对应组的IPR寄存器，并将中断优先级和exception.S中对应的代码段关联起来 */
    AVR32_INTC.ipr[int_grp] = ipr_val
        [
            /* 根据优先级计算出ipr_val数组中对应的下标 */
            int_lev & (AVR32_INTC_IPR_INTLEV_MASK >> AVR32_INTC_IPR_INTLEV_OFFSET)
        ];
}
```

在这段代码片断中我们注意到，对于制定中断源所在的中断组，函数都通过一个数组 **ipr_val[ ]**配合中断优先级信息对 **IPR** 寄存器进行了设置。数组 **ipr_val[ ]**实际上是 **exception.S** 中定义的数据表格，查阅其代码，前文对 IPR 的操作的意图就一目了然了：

```
  .balign 4
  .global ipr_val
  .type ipr_val, @object
/*
 ipr_val中实际保存的是_intn代码区段相对_evba的地址偏移量，也就是IPR中可以使用的相对中断向量
*/
ipr_val:
                                                    /*请注意这个位置*/
  .word (AVR32_INTC_INT0 << AVR32_INTC_IPR_INTLEV_OFFSET) | (_int0 - _evba),\
        (AVR32_INTC_INT1 << AVR32_INTC_IPR_INTLEV_OFFSET) | (_int1 - _evba),\
        (AVR32_INTC_INT2 << AVR32_INTC_IPR_INTLEV_OFFSET) | (_int2 - _evba),\
        (AVR32_INTC_INT3 << AVR32_INTC_IPR_INTLEV_OFFSET) | (_int3 - _evba)
```

完成了上面的处理，无论 **64** 个中断组中哪个中断源被触发，最终都会落到以四个中断优先级命名的代码区段中。在这些中断处理代码中，我们可以通过对应优先级的 **ICR** 寄存器知晓具体是哪个中断组触发了中断，再根据这个线索查找对应中断组的 **IRR** 寄存器，获取当前组内中断请求的信息。

Software Framework 为每个中断组都建立了一个由函数指针组成的指针数组，数组的下标表示组内中断请求的编号。用户在注册中断时，将传递一个中断处理函数的指针，这个指针被以中断源在组内的编号为下标，直接赋值给指针数组。与中断注册的过程正好相反，中断响应时，（组）中断处理程序根据 **IRR** 寄存器中的信息从指针数组里查找出对应的中断处理程序指针，完成具体中断处理函数的调用。关于中断注册、初始化和调用的详细信息大家可以参考 **intc.c** 和 **exception.S** 中的内容，这里就不再赘述。

观察表 8.3.1 容易发现，很多中断组内只有一个中断源，例如，USART0~3 分别独立占有了中断组 5~8，此时，对组中断的响应就是对具体中断源的相应。在这种情况下，如果仍然默认使用 Software Framework 的操作方式，将因为中断处理函数指针数组的处理过程而浪费宝贵的系统周期。在极端追求速度的情况下，我们可以通过修改 exception.S，并通过简单 C 语言实现这类中断处理函数的特权化（注意，这里并没有破坏 Software Framework，这些特权化的中断处理过程并不影响其它正常的中断处理程序）。

以 AVR32UC3A0512 下的 USART0 为例，假设对应的中断处理函数名为 usart0_isr()。首先在 excption.S 中紧接着 int3 增加一个代码段，并加入跳转到 usart0_isr()的汇编代码：

```
_usart0:
    lda.w    pc, usart0_isr
```

在 ipr_val 的定义后面增加以下代码：

```
  .global usart0_ipr_val
  .type usart0_ipr_val, @object
usart0_ipr_val:
  .word (AVR32_INTC_INT0 << AVR32_INTC_IPR_INTLEV_OFFSET) | (_usart0 - _evba),\
        (AVR32_INTC_INT1 << AVR32_INTC_IPR_INTLEV_OFFSET) | (_usart0 - _evba),\
        (AVR32_INTC_INT2 << AVR32_INTC_IPR_INTLEV_OFFSET) | (_usart0 - _evba),\
        (AVR32_INTC_INT3 << AVR32_INTC_IPR_INTLEV_OFFSET) | (_usart0 - _evba)
```

在中断初始化时，通过下面的函数来实现：

```
void usart0_intc_init(unsigned int int_lev)
{
    AVR32_INTC.ipr[AVR32_USART0_IRQ >> AVR32_INTC_MAX_NUM_IRQS_PER_GRP] =
    usart0_ipr_val
        [
            int_lev & (AVR32_INTC_IPR_INTLEV_MASK >> AVR32_INTC_IPR_INTLEV_OFFSET)
        ];
}
```

完成了以上步骤，当 usart0 中断被触发时，会自动通过_usart0:段直接跳转至 usart0_isr()函数执行。其响应速度接近 NMI 中断。对非可屏蔽中断 NMI 中断处理程序的注册方法类似。由于 NMI 在事件系统中是固定寻址的（相对 EVBA 的一个固定的常量偏移），在 exception.S 中也有默认的定义，因此我们只需要修改此处的汇编代码，实现一个函数跳转即可：

```
/* 跳转到nmi_isr()中断处理函数中，nmi_isr()是一个C语言编写的中断处理函数 */
        .org   0x010
        // NMI.
_handle_NMI:
        lda.w    pc, nmi_isr
```

8.4.2 硬件驱动 API 函数

```
/* 该函数用于完成对整个中断管理系统的初始化 */
extern void INTC_init_interrupts(void);
```

```
/* 该函数用于注册中断源 */
/* 输入参数：中断处理函数指针，中断源编号（在uc3xxxxx.h中定义，比如uc3a0512.h），优先级 */
extern void INTC_register_interrupt
        (
            __int_handler handler,
            unsigned int irq,
            unsigned int int_lev
        );
```

根据中断系统的设定，用户中断处理程序需要在函数的最后加入汇编指令“**rete**”。**Software Framework** 提供了两种方式：

A、使用在线汇编。例如：

```
__asm__ __volatile__ ("rete");
```

B、在中断处理函数之前加入修饰符号**__attribute__((__interrupt__))**（**IAR** 使用**__interrupt** 来修饰）。例如：

```
#if __GNUC__
__attribute__((__interrupt__))
#elif __ICCAVR32__
__interrupt
#endif
void usart0_isr_handler(void)
{
  …
}
```

AVR32.UC3.Interrupt.Reference 相关参考信息

8.5 如何获取进一步的参考信息

- 关于中断源的分组信息，可以参考数据手册**Peripherals**章节的**Interrupt Request Signal Map**。
- 关于中断控制器寄存器接口的具体定义，可以参考**include/avr32**目录下的**intc.h**以及**uc3xxxxx.h**中的相关定义。注意，这里的**intc.h**和**DRIVERS\INTC**目录下的**intc.h**内容是不一样的。
- 关于**Software Framework**中断注册、初始化和调用的详细信息可以参考**DRIVERS\INTC**目录下的**C**语言源程序**intc.c**和汇编语言源程序**exception.S**（**IAR**用户请参考**excption.S82**）的相关内容。用户接口定义请参考**DRIVERS\INTC**目录下的**intc.h**文件。