PowerPC 基于 VxWorks 的中断初始化分析

1、本文简介

本文主要介绍 P2020 芯片中 vxWorks 中断初始化过程(部分讲解是以 linux 为例)。P2020 属于 PPC85XX 系列，内核为 e500v2，它是 PowerPC 体系结构中主要应用于通信领域的片子，PowerPC 体系结构规范发布于 1993 年，它是一个 64 位规范（也包含 32 位子集）。但几乎所有常规的 PowerPC 的芯片都是 32 位的，除了部分新型高端芯片（eg:IBM RS/6000 等）。

本文主要章节如下：

2 参考书籍介绍
3 OEA 相关寄存器(中断相关)
4 PowerPC 架构异常处理机制(P2020 为代表）
5 vxWorks 中断初始化过程

2.参考书籍

(1) IBM 官方文档，关于 PowerPC Architecture Boook I II III ，其中 Book III 主要讲解与操作系统相关的寄存器。

下载链接:

https://www.ibm.com/developerworks/systems/library/es-archguide-v2.html.

Although the first PowerPC architecture specification was crafted specifically for desktop systems, it was written as three books, to distinguish the application- and system-level programming models:

- Book I, the user instruction set architecture (UISA), defined the application-level programming model for all PowerPC devices. The PowerPC UISA represents the unchanging mitochondrial DNA, defining the application-level instruction set and programming model common across all architectural variants and preserved by the Power ISA definition.
- Book II, the virtual environment architecture (VEA), defined resources that support the time-base aspects of the memory model, and features for multiprocessor implementations; it is preserved by the Power Architecture definition.
- Book III, the operation environment architecture (OEA), defined operating system–level facilities such as memory translation and interrupts for desktop implementation.

The modular structure made it possible for other architecture-compliant devices, such as the first embedded PowerPC processors, to leverage the PowerPC UISA without being con
desktop-oriented operating system features.

(2) <<E500CORERM.pdf>> 该手册描述的是 e500 核的工作过程以及 Reg 详细描述,包括电源管理,中断处理,Cache,MMU.

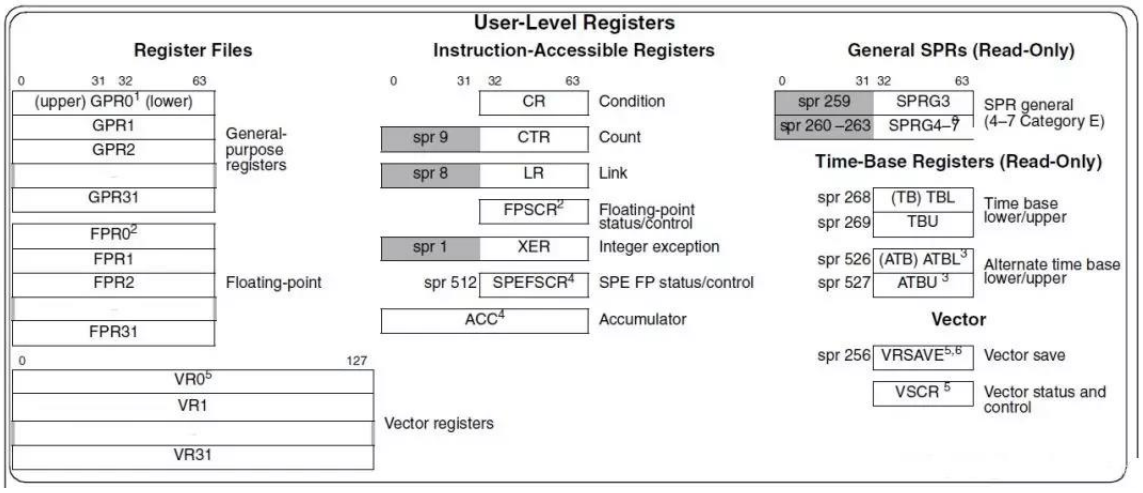
(3) <<P2020RM.pdf>> 该手册主要描述的是芯片与外设相关的控制器，但很少介绍到 CPU 通用寄存器，特殊寄存器等.

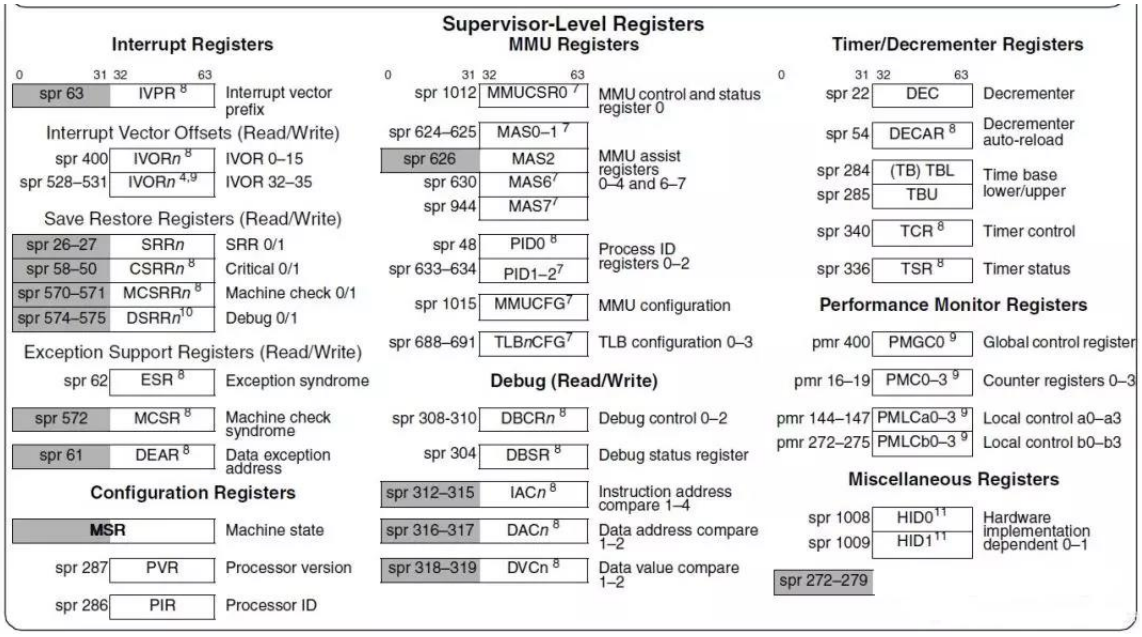
(4) <<821INSTSET.pdf>> PowerPC 汇编指令集介绍

(5) PowerPC 汇编指令简单介绍

（2.3.4）都可以在 NXP 官网中下载

3 OEA 相关寄存器(中断相关)
PowerPC 处理器有 32 个（32 位或 64 位）GPR（通用寄存器）以及诸如 PC（程序计数器，也称为 IAR / 指令地址寄存器 或 NIP / 下一指令指针）、LR（链接寄存器）、CR（条件寄存器）等各种其它寄存器。有些 PowerPC CPU 还有 32 个 64 位的 FPR(浮点寄存器). 下图展示了 Power ISA2.04 寄存器模式，且值得注意的是 PowerPC 有两种执行模式，分别为用户模式(UserMode) 和特权模式(SupervisorMode)，MSR[PR]=0 表示特权模式。Linux 内核运行在特权模式，而普通程序运行在用户模式。VxWorks 全部运行在特权模式。





3.1 OEA 寄存器集包括四类寄存器(具体介绍只涉及中断相关的，其它请参考下来的链接)

本小章节参考：32 位 PowerPC 构架通用寄存器分析及总结

(1) 配置寄存器 (Configuration Registers)

MSR 寄存器：（与中断息息相关）

定义处理器的状态，它可以被 mtmsr, sc, rfi 指令修改；可以被 mfmsr 读取；

详细说明:基于603E内核的PowerPC处理器,中断向量为一个固定的地址，可以通过设置MSR[IP]位来决定这个固定地址是 0x0,还是 0xfff0_0000;而 E500 内核在进入中断和异常处理程序时，不能关闭 MMU，因此不能使用物理地址作为中断向量，而应使用 IVPR 和 IVOR 寄存器保存相应的中断向量(在下面中断机制章节详细介绍),下面为 e500 核的 MSR:

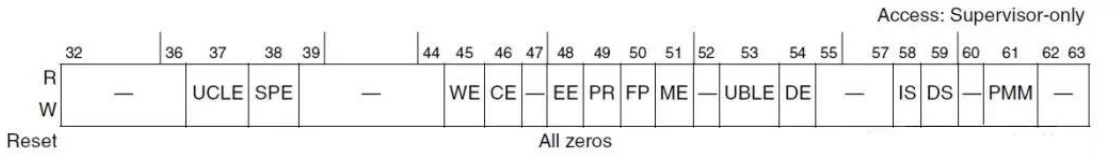


Figure 2-2. Machine State Register (MSR)

中断的开关由寄存器 MSR 来控制的

CE:if set 1,Critical input and watchdog timer interrupts are enabled.

EE:if set 1,External input, decrementer, fixed-interval timer and performance monitor interrupts are enabled

ME:if set 1, Machine check interrupts are enabled.

DE:if set 1,Debug interrputs are enabled if DBCR0[IDM] = 1.

MSR[EE]这个 bit 很重要，中断的使能要靠它。
注意：当发生中断后，MSR[EE]会自动置 0 屏蔽所有其他的中。
如果在 MSR[EE]置 1 前，又来了一个中断
1.此中断是边沿触发的，那么此中断丢失。此种中断就是电平变化，一次就没了
2.此中断时水平触发的，此中断不会丢失。此种中断的电平会一直 active，直到硬件处理它

PVR 寄存器：

定义寄存器模型的版本和处理器的版

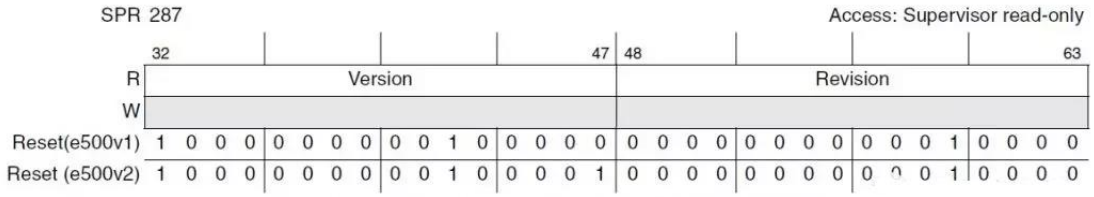


Figure 2-3. Processor Version Register (PVR)

（2）内存管理寄存器（Memory Management Registers）

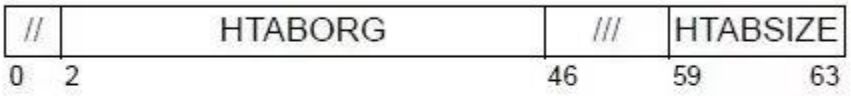
BAT 寄存器：

OEA 定义了四组 BAT 指令寄存器（IBAT0U-IBAT3U 和 IBAT0L-IBAT3L），也定义了四组 BAT 数据寄存器（DBAT0U-DBAT3U 和 DBAT0U-DBAT3U）

SDR1 寄存器：

该寄存器定义了用于虚拟地址转换为物理地址所需要的页表基地址

The SDR1 register is shown in Figure 24.



Bits	Name	Description
2:45	HTABORG	Real address of Page Table
59:63	HTABSIZE	Encoded size of Page

SR 寄存器：

OEA 定义了 16 个 32 位的 SR 寄存器（SR0-SR15）

（3）中断处理寄存器（Interrupt Handing Register）

详细介绍请看《E500CORE.pdf》2.7 Interrupt Registers

This section describes the following register bits and their fields:

- Section 2.7.1.1, “Save/Restore Register 0/1 (SRR0 and SRR1)”
- Section 2.7.1.2, “Critical Save/Restore Register 0/1 (CSRR0 and CSRR1)”
- Section 2.7.1.3, “Data Exception Address Register (DEAR)”
- Section 2.7.1.4, “Interrupt Vector Prefix Register (IVPR)”
- Section 2.7.1.5, “Interrupt Vector Offset Registers (IVORs)”
- Section 2.7.1.6, “Exception Syndrome Register (ESR)”

（4）多功能寄存器（Miscellaneous Registers）

TB（time base）寄存器：CPU 时间片基准

DEC 寄存器：这是一个 32 位的递减计数器

EAR（External Access Register）寄存器：用于访问外部设备

DABR（Data address breakpoint register）：用来控制数据地址断点功能

PIR（Processor identification register）：在多处理器的芯片上用来标识一个核，例如在 MPC8641d 芯片上有两个 E600 的 core，就用 PIR 来定位其中的 core。

3.3 通用寄存器与专用寄存器的用途

本小章节参考：PowerPC 汇编指令以及通用与专用寄存器介绍

r0 在函数开始（function prologs）时使用。

r1 堆栈指针，相当于ia32架构中的esp寄存器，idapro把这个寄存器反汇编标识为sp。

r2 内容表（toc）指针，idapro把这个寄存器反汇编标识为rtoc。系统调用时，它包含系统调用号（这个好像跟系统有关吧）。

r3 作为第一个参数和返回值。

r4-r10 函数或系统调用开始的参数。

r11 用在指针的调用和当作一些语言的环境指针。

r12 它用在异常处理和glink（动态连接器）代码。

r13 保留作为系统线程ID。

r14-r31 作为本地变量，非易失性。

专用寄存器的用途：

lr 链接寄存器，它用来存放函数调用结束处的返回地址。

ctr 计数寄存器，它用来当作循环计数器，会随特定转移操作而递减。

xer 定点异常寄存器，存放整数运算操作的进位以及溢出信息。

msr 机器状态寄存器，用来配置微处理器的设定。

cr 条件寄存器，它分成8个4位字段，cr0-cr7，它反映了某个算法操作的结果并且提供条件分支的机制。

寄存器r1、r14-r31是非易失性的，这意味着它们的值在函数调用过程保持不变。寄存器r2也算非易失性，但是只有在调用函数在调用后必须恢复它的值时才被处理。

寄存器r0、r3-r12和特殊寄存器lr、ctr、xer、fpscr是易失性的，它们的值在函数调用过程中会发生变化。此外寄存器r0、r2、r11和r12可能会被交叉模块调用改变，所以函数在调用的时候不能采用它们的值。

条件代码寄存器字段cr0、cr1、cr5、cr6和cr7是易失性的。cr2、cr3和cr4是非易失性的，函数如果要改变它们必须保存并恢复这些字段。

在AIX上，svca指令（sc是PowerPC的助记符）用来表示系统调用，r2寄存器指定系统调用号，r3-r10寄存器
行系统调用指令之前有两个额外的先决条件：LR寄存器必须保存返回系统调用地址的值并且在系统调用前执行cr0、cr6、cr6、cr6指令

在 target\h\arch\ppc\toolPpc.h 中有对以上寄存器的重定义

```
#define p0  r3  /* argument register, volatile */
#define p1  r4  /* argument register, volatile */
#define p2  r5  /* argument register, volatile */
#define p3  r6  /* argument register, volatile */
#define p4  r7  /* argument register, volatile */
#define p5  r8  /* argument register, volatile */
#define p6  r9  /* argument register, volatile */
#define p7  r10 /* argument register, volatile */
#define glr0    r0  /* prologs(PO,EABI), epilogs, glink routines(EABI) /
             * language specific purpose(SVR4), volatile */
#define glr1    r11 /* prologs, epilogs, as Pascal environment pointer(EABI)
             * language specific purpose (SVR4)
             * calls by pointer, as Pascal environment(PO),
             * volatile */
#define glr2    r12 /* prologs, epilogs, glink routines, calls by
             * pointer(EABI), language specific purpose (SVR4),
             * glue code, exception handling (PO), volatile */
#define retval0 r3  /* return register 0, volatile */
#define retval1 r4  /* return register 1, volatile */
```

4 PowerPC 架构异常处理机制(P2020 为代表）

4.1 PowerPC 中断系统(P2020)

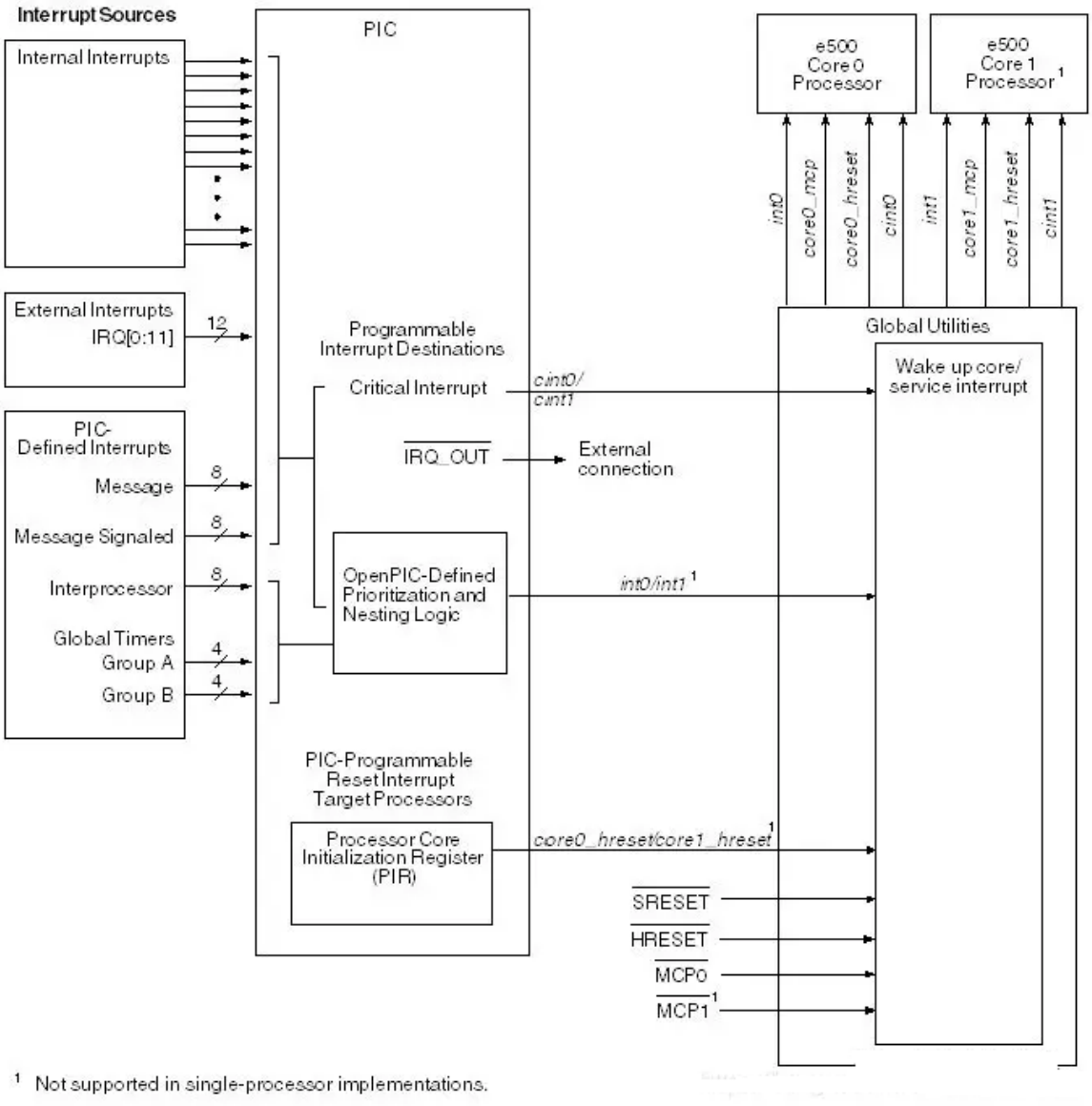
从 CPU 的角度来讲，中断源可以分为自己内核产生的异常和 PIC 提供的中断。

异常：是 e500 核产生的，它是同步产生(可以预知的),如非法指令，或访问存储器时出现 TLB Miss(Data | Instruction)等.

<<E500CORERM.pdf>>5.11.1 描述了 PowerPC 架构可能出现的全部异常(这里描述包括中断)

中断：是 e500 核外部引脚产生的，由 PIC 送到内核处理，它是异步产生的(无法预知的）.大致分为一般中断(int)（包括内外部等），关键中断(cint)和机器检查中断(machine check). 下图为中断源架构图，其中 Message 数量应该是写错了(理论为 4).

从图中可以看出 PowerPC 处理器中断系统由内核中断以及异常处理系统和 PIC 中断控制器构成.



4.2 中断向量

在 E500 内核中，使用 IVPR 和 IVORx 寄存器共同确定中断或者异常程序的入口地址。其中，IVPR 寄存器提供中断程序入口地址的第 0~15 位，IVORx 提供中断程序入口地址的第 16~27 位，而中断程序的入口地址的第 28~31 位为 0。IVORx 与异常的对应关系如下图所示：

2.7.1.4 Interrupt Vector Prefix Register (IVPR)

The e500 implements IVPR as it is defined by the Book E architecture. It is used with IVORs to determine the vector address. IVPR[32–47] provides the high-order 16 bits of the address of the exception processing routines. The 16-bit vector offsets are concatenated to the right of IVPR[32–47] to form the address of the exception processing routine.

2.7.1.5 Interrupt Vector Offset Registers (IVORs)

The e500 implements the IVORs as defined by the Book E architecture, but use only IVOR*n*[48–59], as shown in Figure 2-9, to hold the quad-word index from the base address provided by the IVPR for each interrupt type.

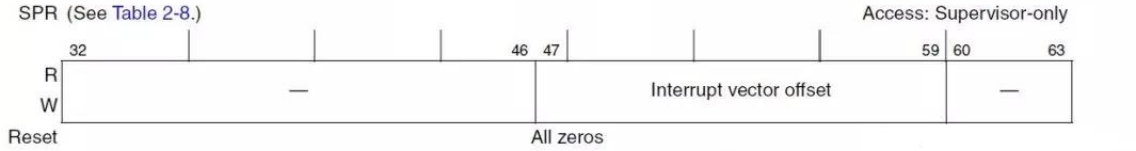


Figure 2-9. Interrupt Vector Offset Registers (IVORs)

Table 2-8. IVOR Assignments

IVOR Number	SPR	Interrupt Type
IVOR0	400	Critical input
IVOR1	401	Machine check
IVOR2	402	Data storage
IVOR3	403	Instruction storage
IVOR4	404	External input
IVOR5	405	Alignment
IVOR6	406	Program
IVOR7	407	Floating-point unavailable (Not supported on the e500)
IVOR8	408	System call
IVOR9	409	Auxiliary processor unavailable (Not supported on the e500)
IVOR10	410	Decrementer
IVOR11	411	Fixed-interval timer interrupt
IVOR12	412	Watchdog timer interrupt
IVOR13	413	Data TLB error
IVOR14	414	Instruction TLB error
IVOR15	415	Debug

每类中断都有自己的中断向量，通过它能计算出中断 handler 的指令地址。
指令地址的计算方法：
IVPR[32-47] || IVORn[48-59] || 0b0000
这样就得到一个 32 个 bit 地址

有 2 个中断向量对程序员来说比较亲切：
一个是 IVOR4 用来处理外部中断和内部 SOC 中断
一个是 IVOR8 用来处理系统调用

随便找一个 Start.S 进行简单的分析

```
185 /* Setup interrupt vectors 设置 IVPR 寄存器*/
186 lis r1,TEXT_BASE@h     /* load value to r1 */
187 mtspr IVPR, r1   /* write r1 to IVPR */
197 li r1,0x0500
198 mtspr IVOR4,r1 /* 4: External interrupt */
  .......
205 li r1,0x0900
206 mtspr IVOR8,r1 /* 8: System call */
207 /* 9: Auxiliary processor unavailable(unsupported) */
208 li r1,0x0a00
209 mtspr IVOR10,r1 /* 10: Decrementer */
......
```

看 board/pq37pc/pq37pc_8560/config.mk 里有
TEXT_BASE = 0xFF800000
所以 IVPR = 0xFFF80000
IVOR10 = 0x00000a00
计算 IVPR[32-47] || IVPOR10[48-59] || 0b0000 得 0xFFF80a00

4.3 中断源寄存器

中断向量只能确定类型，里面保存了 interrupt handler 的地址。
在 interrupt handler 里，需要根据中断源寄存器来进一步确定到底是哪里发生了中断

E500 内部中断源有 64 个(部分 reserve)，来自 TSEC，LBC，DMA，CPM 等

寄存器组 PIC_IIVPRn 与 P2020 内部 SOC 中断一一对应（n 取值 0~63）

5_0200	Internal interrupt n vector/priority register (PIC_IIVPR0)	32	R/W	8080_0000h	9.3.45/440
5_0210	Internal interrupt n destination register (PIC_IIDR0)	32	R/W	0000_0001h	9.3.46/441
5_0220	Internal interrupt n vector/priority register (PIC_IIVPR1)	32	R/W	8080_0000h	9.3.45/440
5_0230	Internal interrupt n destination register (PIC_IIDR1)	32	R/W	0000_0001h	9.3.46/441
5_0240	Internal interrupt n vector/priority register (PIC_IIVPR2)	32	R/W	8080_0000h	9.3.45/440
5_0250	Internal interrupt n destination register (PIC_IIDR2)	32	R/W	0000_0001h	9.3.46/441
5_0260	Internal interrupt n vector/priority register (PIC_IIVPR3)	32	R/W	8080_0000h	9.3.45/440
5_0270	Internal interrupt n destination register (PIC_IIDR3)	32	R/W	0000_0001h	9.3.46/441
5_0280	Internal interrupt n vector/priority register (PIC_IIVPR4)	32	R/W	8080_0000h	9.3.45/440
5_0290	Internal interrupt n destination register (PIC_IIDR4)	32	R/W	0000_0001h	9.3.46/441
5_02A0	Internal interrupt n vector/priority register (PIC_IIVPR5)	32	R/W	8080_0000h	9.3.45/440
5_02B0	Internal interrupt n destination register (PIC_IIDR5)	32	R/W		
5_02C0	Internal interrupt n vector/priority register (PIC_IIVPR6)	32	R/W	8080_0000h	9.3.45/440

Table 9-2. Internal interrupt assignments

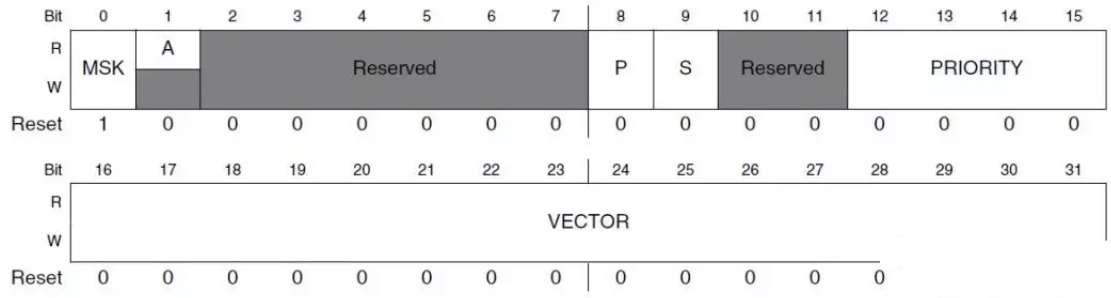
Internal Interrupt Number	Interrupt Source
0	L2 cache
1	ECM
2	DDR DRAM controller
3	eLBC controller
4	DMA 1 channel 1
5	DMA 1 channel 2
6	DMA 1 channel 3
7	DMA 1 channel 4
8	PCI Express Port 3
9	PCI Express Port 2
10	PCI Express Port 1
11	Reserved
12	USB
13	eTSEC 1 transmit
14	eTSEC 1 receive
15	eTSEC 3 transmit
16	eTSEC 3 receive

E500 外部中断源有 12 个，来自外部引脚 IRQ[0:11]
寄存器组 EIVPRn 与 MPC8560 外部中断一一对应（n 取值 0~11）

5_0000	External interrupt n (IRQn) vector/priority register (PIC_EIVPR0)	32	R/W	8000_0000h	9.3.43/437
5_0010	External interrupt n (IRQn) destination register (PIC_EIDR0)	32	R/W	0000_0001h	9.3.44/439
5_0020	External interrupt n (IRQn) vector/priority register (PIC_EIVPR1)	32	R/W	8000_0000h	9.3.43/437
5_0030	External interrupt n (IRQn) destination register (PIC_EIDR1)	32	R/W	0000_0001h	9.3.44/439
5_0040	External interrupt n (IRQn) vector/priority register (PIC_EIVPR2)	32	R/W	8000_0000h	9.3.43/437
5_0050	External interrupt n (IRQn) destination register (PIC_EIDR2)	32	R/W	0000_0001h	9.3.44/439
5_0060	External interrupt n (IRQn) vector/priority register (PIC_EIVPR3)	32	R/W	8000_0000h	9.3.43/437
5_0070	External interrupt n (IRQn) destination register (PIC_EIDR3)	32	R/W	0000_0001h	9.3.44/439
5_0080	External interrupt n (IRQn) vector/priority register (PIC_EIVPR4)	32	R/W		

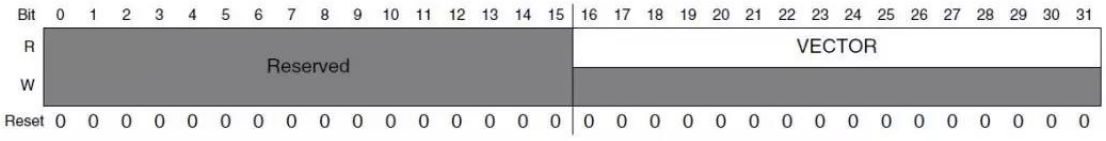
下面具体介绍外部源寄存器

Address: 4_0000h base + 1_0000h offset + (32d × i), where i=0d to 11d



MSK: 若置 1，此中断源的中断被无视.
A: 若置 1,此中断源有中断发生
P: 若置 1, active-high; 若置 0, active-low.
S: 若置 1,此中断是水平触发。若置 0,此中断是边界触发
PRIORITY: 优先级 0-15。15 是最高优先级，0 时相当于无视此中断。
VECTOR: 硬件中断号。中断发生后处于 pengding 时，此字段被写在寄存器 IACK 中

Address: 4_0000h base + A0h offset = 4_00A0h



PIC_IACK field descriptions

Field	Description
0–15 -	This field is reserved. Reserved
16–31 VECTOR	Interrupt vector. Vector of the highest pending interrupt

总结：处理内，外中断的 handle 都是同一个 functions(流程)，EntInt-->Handle-->ExitInt,在处理函数中读取 IACK,就知道是哪个中断源触发了中断,然后做对应的处理

知道中断向量 和 中断源，系统处理中断的大致流程就能理出来了，以外部中断 eg：

知道了 中断向量 和 中断源，系统处理中断的大致流程就能理出来了，以外部中断举例：

1. 硬件上，外部中断管脚8上传来一个中断

CPU 的 PIC 将寄存器 EIVPR8 的VECTOR 字段拷贝到寄存器 IACK 中

这个VECTOR字段就是所谓的“硬中断号”，它是可编程的。不同OS，硬中断号分配方法不一样。

2. 然后PIC 通知 CPU 的 core 跳转到下列地址，执行interrupt handler

IVPR[32-47] || IVOR4[48-59] || 0b0000

IVOR4 就是中断对应的中断向量寄存器了，它在uboot中是这样初始化的

SET_IVOR(4, ExternalInput)

注意，外部中断、内部中断的 interrupt handler 都是 ExternalInput

3.软件上 ExternalInput 做下列事情：

读寄存器IACK，获取VECTOR字段，获得“硬中断号”

接着保存中断现场，然后再接着处理，因OS而异。

Linux中是根据“硬中断号”获取“逻辑中断号”，然后调用逻辑中断号上所挂的中断处理例程

EXCEPTION(0x0500, ExternalInput, do_IRQ, EXC_XFER_LITE)

前面说的都E500中断架构，下面讲Linux在此架构下的中断机制的实现

先简单列下中断处理流程：

第一步：

中断来了根据中断向量表（在head_fs_booke.S中初始化）来执行handler

如IVOR4，就执行ExternalInput() —> do_IRQ()

第二步：

do_IRQ()通过读IACK获取“硬件中断号”

第三步：

硬件中断号m —> 逻辑中断号n

从而找到irq_desc[n]

第四步：

遍历执行irq_desc[n]上action链表（这些action是通过request_irq注册的）

上面基本上就是一个硬中断的处理过程，由上可知的，它需要三个初始化：

1. 中断向量的初始化
2. 中断源的初始化 （即硬件中断号的分配）
3. ISR中断处理例程的注册（也就是上面的action）

4.4 外部中断处理流程

1.首先 E500 内核清除在指令完成队列 CQ 中所有的执行，然后把正在执行的指令序列下一条指令地址保存到中断寄存器 SRR0(Save/RestoreRegiser 0)

2.把当前 MSR 的内容保存到 SRR1

3.把 MSR 某些比特置为 0，如 MSR[SPE,WE,EE,PR,FP,FE0,FE1,IS,DS]are 0 by all interrupts.

（E500 内核将 MSR 寄存器的 CE、ME、DE 位保留，其他位全部清零。因此 E500 内核在进行外部中断处理程序时，仍然可以被 Critical 中断，Machine check 中断和调试中断程序重入，但是不能被外部中断立即重入。在 Linux PowerPC 中，外部中断处理程序会选择合适时机使能 MSR 寄存器的 EE 位，以支持外部中断的重入）

4.在新的 MSR 状态下， E500 内核将根据 IVPR，IVOR4 寄存器确定中断向量，从中断向量偏移处开始指令读取和执行

5. 在中断处理程序执行完毕后，使用 rfi 指令进行中断返回。rfi 指令将从 SRR1 寄存器中恢复 MSR 寄存器的值，并从 SRR0 寄存器中获得程序返回地址。rfi 指令在进行程序正文切换之前还会进行指令和数据的同步，还给被中断的程序一个“干净”的空间，之后 E500 内核进行中断返回。

5.vxWorks 中断初始化过程(跳过 bootrom 过程)

usrInit(prjConfig.c)--> {sysStart(startType)-->intVecBaseSet ((FUNCPTR *) VEC_BASE_ADRS)}

-->excVecInit()

（1）sysStart (first C code executed from usrInit) 主要作用是清除 BSS,以及设置中断向量的基地址

```
Code in arget\config\comps\src\usrStartup.c

中断向量的基地址一般设置为0x0000_0000 | 0xfff0_00000
macro in BSP/config.h
#define VEC_BASE_ADRS           LOCAL_MEM_LOCAL_ADRS
#define LOCAL_MEM_LOCAL_ADRS    0x00000000

void sysStart(int startType){

#if (CPU_FAMILY == PPC) || (CPU_FAMILY == MIPS)

    /*
     * For PPC and MIPS, the call to vxSdaInit() must be the first operation
     * in sysStart(). This is because vxSdaInit() sets the SDA registers
     * (r2 and r13 on PPC, gp on MIPS) to the SDA base values. No C code
     * must be placed before this call.
     */

    _WRS_ASM ("");   /* code barrier to prevent compiler moving vxSdaInit() */
    vxSdaInit ();    /* this MUST be the first operation in usrInit() for PPC */
    _WRS_ASM ("");   /* code barrier to prevent compiler moving vxSdaInit() */

#endif  /* (CPU_FAMILY == PPC) || (CPU_FAMILY == MIPS) */
...

#ifdef  CLEAR_BSS
        bzero (edata, end - edata);
#endif  /* CLEAR_BSS */

    sysStartType = startType;
    intVecBaseSet ((FUNCPTR *) VEC_BASE_ADRS);
...

#ifdef _WRS_CONFIG_USE_MEMDESC
    sysMemDescInit ();
#endif
}
```

(2) intVecBaseSet

```
Code in target\src\arch\ppc\intArchLib.c

说明：此阶段函数并未真正设置中断向量基地址到e500内核中，只是对_ppcExcIntVecBase赋值的
     过程，实际设置的过程在excVecInit里面。

int       (* _func_intLevelSetRtn) (int) = NULL;
int       (* _func_intEnableRtn) (int) = NULL;
int       (* _func_intDisableRtn) (int) = NULL;

上面的函数指针初始化在vxbEpicIntCtlr.c里面

int intEnable
    (
    int intLevel                /* interrupt level to enable */
    )
    {
    /* execute VxBus Legacy interrupt enable routine first if supported */

    if ((_func_vxbIntEnable != NULL) &&
        (_func_vxbIntEnable (intLevel) != ERROR))
        return (OK);

    if (_func_intEnableRtn != NULL)
        return (_func_intEnableRtn (intLevel));

    return (ERROR);
    }

void intVecBaseSet(FUNCPTR * baseAddr){

  /* 主要_func_intVecBaseSetRtn实际为空，在vxWorks目前阶段函数指针未赋值
  ** 可在vxWorks shell 中 _func_intVecBaseSetRtn == 1 测试返回值
  */
    if (_func_intVecBaseSetRtn != NULL)
        _func_intVecBaseSetRtn (baseAddr);
```

```

    _ppcExcIntVecBase = baseAddr;

    _ppcAllocationQuantumSize = _CPU_ALLOC_ALIGN_SIZE;
    _ppcStackAlignSize = _CPU_STACK_ALIGN_SIZE;

    ...
}

FUNCPTR * intVecBaseGet (void)
{
    if (_func_intVecBaseGetRtn == NULL)
        return (_ppcExcIntVecBase);

    return (_func_intVecBaseGetRtn ());
}
```

（3）excVecInit

```
Code in target\src\arch\ppc\excArchLib.c

STATUS excVecInit (void)
    {
    FAST int ix;

#if !defined(_WRS_CONFIG_WRHV_GUEST) || defined(_VB_PISA_EHV)
    if (excExtendedVectors == TRUE)
        {
        entOffset     = EXT_ENT_OFF;
        isrOffset     = EXT_ISR_OFF;
        exitOffset    = EXT_EXIT_OFF;
#ifdef _EXC_OFF_CRTL
    ...
#endif  /* _EXC_OFF_CRTL */
        }
    else
        {
     ...
        }

    excVecBaseSet(intVecBaseGet());

    for (ix = 0; excBlTbl[ix].excHandler != (void (*)()) NULL; ix++)
        {
        excVecConnectCommon (excBlTbl[ix].vecOff,
                             excBlTbl[ix].vType,
                             excBlTbl[ix].excHandler,
                             excBlTbl[ix].vecOffReloc);
        }
```

```
#ifndef _WRS_CONFIG_WRHV_GUEST
#ifdef  IVOR0
    excIvorInit();
#endif  /* IVOR0 */
#endif  /* _WRS_CONFIG_WRHV_GUEST */

    ...

    /*
     * Now that the vectors are set up, and provided we can safely do
     * so prior to MMU setup, set the recoverability indicator if so
     * equipped.  (If excVecBase and excVecBaseAltAdrs differ,
     * no exceptions can be handled until the MMU has been set up.)
     * We don't enable Machine Check exceptions here because excHandler
     * is not ready, i.e. taskIdCurrent is a meaningless value now.
     * We postpone it to usrRoot and use taskMsrDefault to enable it.
     */
    if (excVecBaseAltAdrs == excVecBase)
        vxMsrSet (vxMsrGet()
#ifdef  _PPC_MSR_RI
                            | _PPC_MSR_RI
#endif  /* _PPC_MSR_RI */
        );

#ifndef _WRS_CONFIG_WRHV_GUEST
    /* Used for the generic layered exception handler */
    hdlrBase = excVecBase + _EXC_OFF_END;
    /* save the Data and/or Instruction MMU selected */
    hdlrCodeBase = excVecBaseAltAdrs + _EXC_OFF_END;

#if (CPU==PPC85XX)
# if !defined(PPC_e200) && !defined(PPC_e500mc) || defined(PPC_e6500)
    installE500ParityErrorRecovery();
# endif /* !PPC_e200 && !PPC_e500mc || PPC_e6500 */
#endif /* (CPU==PPC85XX) */
#endif  /* _WRS_CONFIG_WRHV_GUEST */

```

```
#else   /* _WRS_CONFIG_WRHV_GUEST && !_VB_PISA_EHV */
    {
    char * addr;
    int excSize = (int)&_func_exc_handler_end - (int)&_func_exc_handler;
    int intSize = (int)&_func_int_handler_end - (int)&_func_int_handler;

    bzero ((void *)0, _EXC_OFF_END);

    for (addr = 0; addr < (char *)_EXC_OFF_END; addr+=0x100)
        {
        bcopy ((void *)&_func_exc_handler, (void *)addr, excSize);
        CACHE_TEXT_UPDATE ((void *)addr, excSize);
        }
    /* Set interrupt handlers */
    bcopy ((void *)&_func_int_handler, (void *)_EXC_OFF_INTR, intSize);
    CACHE_TEXT_UPDATE ((void *)_EXC_OFF_INTR, intSize);

#ifdef _EXC_OFF_DIRECT_INTR
    bcopy ((void *)&_func_int_handler, (void *)_EXC_OFF_DIRECT_INTR, intSize);
    CACHE_TEXT_UPDATE ((void *)_EXC_OFF_DIRECT_INTR, intSize);
#endif /* _EXC_OFF_DIRECT_INTR */

#ifdef _EXC_OFF_DECR
    bcopy ((void *)&_func_int_handler, (void *)_EXC_OFF_DECR, intSize);
    CACHE_TEXT_UPDATE ((void *)_EXC_OFF_DECR, intSize);
#endif /* _EXC_OFF_DECR */
    }
#endif  /* _WRS_CONFIG_WRHV_GUEST && !_VB_PISA_EHV */

    return (OK);
    }
```

下面按 excVecInit 里面函数的初始化顺序讲解：

```
macro in target\h\arch\ppc\private\exArchLibP.h

#   define    EXT_ENT_OFF     3      /* offset for ext intEnt/excEnt */
#   define    EXT_ISR_OFF     8      /* offset for ext ISR or exc handler */
#   define    EXT_EXIT_OFF     15    /* offset for ext intExit/excExit */

描述的是在每类中断向量里面中断入口函数，处理函数，退出函数相对于这类向量地址的偏移量，目前我还没理解值为啥是这个数。

(1)excVecInit里面有:
entOffset     = EXT_ENT_OFF;
isrOffset     = EXT_ISR_OFF;
exitOffset    = EXT_EXIT_OFF;

(2)excVecInit 调用 excVecBaseSet(intVecBaseGet()); 此时才真正设置了中断向量基地址

   void excVecBaseSet(FUNCPTR * baseAddr){
        ...
        excVecBase = (vectorBase)((uint32_t)baseAddr & 0x0ffff0000);
        vxIvprSet ((int) excVecBase);
        ...
    }

  /* code in vxALib.s */
  FUNC_BEGIN(vxIvprGet)
        mfspr   p0, IVPR
        blr
  FUNC_END(vxIvprGet)


   FUNC_BEGIN(vxIvprSet)
        mtspr   IVPR, p0
        blr
   FUNC_END(vxIvprSet)

```

(3)excVecConnectCommon 该程序是安装所有中断默认处理的函数

函数涉及vxWorks中一个重要的数据结构异常向量表excBlTbl[]

```
typedef struct excTbl
{
    vecTblOffset  vecOff;              /* vector offset */
    EXC_TYPE      vType;               /* exception type */
    void          (*excHandler) ();    /* exception handler routine */
    vecTblOffset  vecOffReloc;         /* relocated vector offset */
} EXC_TBL;
```

vecTblOffset 为UINT32，在此代表的意义为相对于中断向量表基地址的偏移量(简单来说就是vecOff用来设置IVORs寄存器的,当然基地值为0的

异常向量表的表项数量和内容随CPU不同而不同，对于P2020处理器而言，其异常向量表中的表项如下：

```
{
    {_EXC_OFF_CRTL,        V_CRIT_INT,  excIntHandle, 0}, /* critical int */
#ifdef _PPC_MSR_MCE
    {_EXC_OFF_MACH,        V_MCHK_EXC,  excExcHandle, 0}, /* machine chk  */
#elif  _PPC_MSR_CE
    {_EXC_OFF_MACH,        V_CRIT_EXC,  excExcHandle, 0}, /* machine chk  */
#endif /* _PPC_MSR_MCE */
    {_EXC_OFF_DATA,        V_NORM_EXC,  excExcHandle, 0}, /* data storage */
    {_EXC_OFF_INST,        V_NORM_EXC,  excExcHandle, 0}, /* instr access */
    {_EXC_OFF_INTR,        V_NORM_INT,  excIntHandle, 0}, /* ext int */
#ifdef _EXC_OFF_DIRECT_INTR
    {_EXC_OFF_DIRECT_INTR, V_NORM_INT,  excIntHandle, 0}, /* ext int */
#endif /* _EXC_OFF_DIRECT_INTR */
    {_EXC_OFF_ALIGN,       V_NORM_EXC,  excExcHandle, 0}, /* alignment */
    {_EXC_OFF_PROG,        V_NORM_EXC,  excExcHandle, 0}, /* program */
    {_EXC_OFF_FPU,         V_NORM_EXC,  excExcHandle, 0}, /* fp unavail */
    {_EXC_OFF_SYSCALL,     V_NORM_EXC,  excExcHandle, 0}, /* syscall */
    {_EXC_OFF_APU,         V_NORM_EXC,  excExcHandle, 0}, /* auxp unavail*/
    {_EXC_OFF_DECR,        V_NORM_INT,  excDecrHandle, 0}, /* decrementer */
    {_EXC_OFF_FIT,         V_NORM_INT,  excIntHandle, 0}, /* fixed timer */

    {_EXC_OFF_WD,          V_CRIT_INT,  excIntHandle, 0}, /* watchdog */

```

```
    {_EXC_OFF_DATA_MISS,   V_NORM_EXC,  excExcHandle, 0}, /* data TLB miss */
    {_EXC_OFF_INST_MISS,   V_NORM_EXC,  excExcHandle, 0}, /* inst TLB miss */
    {_EXC_OFF_DBG,         V_CRIT_EXC,  excExcHandle, 0}, /* debug events */
#ifdef _WRS_ALTIVEC_SUPPORT
    {_EXC_ALTIVEC_UNAVAILABLE, V_NORM_EXC,excExcHandle, 0}, /* altivec unav */
    {_EXC_ALTIVEC_ASSIST,  V_NORM_EXC, excExcHandle, 0},  /* altivec asst */
#endif /* _WRS_ALTIVEC_SUPPORT */
#ifdef _WRS_SPE_SUPPORT
    {_EXC_OFF_SPE,         V_NORM_EXC,  excExcHandle, 0}, /* SPE */
    {_EXC_OFF_VEC_DATA,    V_NORM_EXC,  excExcHandle, 0}, /* vector data */
    {_EXC_OFF_VEC_RND,     V_NORM_EXC,  excExcHandle, 0}, /* vector round */
#endif /* _WRS_SPE_SUPPORT */
    {_EXC_OFF_PERF_MON,    V_NORM_INT,  excIntHandle, 0}, /* perf monitor */
}

表格中涉及的宏target\h\arch\ppc\excPpcLib.h
macro in
#define _EXC_OFF_CRTL           0x0100  /* Critical Input */
#define _EXC_OFF_MACH           0x0200  /* Machine Check */
#define _EXC_OFF_DATA           0x0300  /* Data Storage */
#define _EXC_OFF_INST           0x0400  /* Instruction Storage */
#define _EXC_OFF_INTR           0x0500  /* External Input */
#define _EXC_OFF_ALIGN          0x0600  /* Alignment */
#define _EXC_OFF_PROG           0x0700  /* Program */
#define _EXC_OFF_FPU            0x0800  /* Floating Point Unavailable */
#define _EXC_OFF_SYSCALL        0x0900  /* System Call */
#define _EXC_OFF_APU            0x0a00  /* Auxiliary Processor Unavailable */
#define _EXC_OFF_DECR           0x0b00  /* Decrementer */
#define _EXC_OFF_FIT            0x0c00  /* Fixed Interval Timer */
#define _EXC_OFF_WD             0x0d00  /* Watchdog Timer */
#define _EXC_OFF_DATA_MISS      0x0e00  /* Data TLB Error */
#define _EXC_OFF_INST_MISS      0x0f00  /* Instruction TLB Error */
#define _EXC_OFF_DBG            0x1000  /* Debug exception */

```

```
106 typedef enum excType
107     {
108     V_NORM_EXC = 0,
109     V_NORM_INT /* 重点对象 */
110 #ifdef  _PPC_MSR_CE
111     ,V_CRIT_EXC
112     ,V_CRIT_INT
113 #ifdef  _PPC_MSR_MCE
114     ,V_MCHK_EXC
115 #endif  /* _PPC_MSR_MCE */
116 #endif  /* _PPC_MSR_CE */
117     } EXC_TYPE;
118
119 /* 中断的入口函数以及出口函数 */
120 LOCAL EXC_WRAPPERS excTypeRtnTbl[] =
121     {
122     {excEnt, excExit},                  /* V_NORM_EXC */
123     {intEnt, intExit},                  /* V_NORM_INT */
124 #ifndef _VB_PISA_EHV
125 #ifdef  _PPC_MSR_CE
126     {excCrtEnt, excCrtExit},            /* V_CRIT_EXC */
127     {intCrtEnt, intCrtExit},            /* V_CRIT_INT */
128 #ifdef  _PPC_MSR_MCE
129     {excMchkEnt, excMchkExit}           /* V_MCHK_EXC */
130 #endif  /* _PPC_MSR_MCE */
131 #endif  /* _PPC_MSR_CE */
132 #else  /* _VB_PISA_EHV */
133
134 /* XXX need spurious exception handler */
135 # ifdef  _PPC_MSR_CE
136     {NULL, NULL},                       /* V_CRIT_EXC */
137     {NULL, NULL},                       /* V_CRIT_INT */
138 #  ifdef  _PPC_MSR_MCE
139     {NULL, NULL}                        /* V_MCHK_EXC */
140 #  endif  /* _PPC_MSR_MCE */
141 # endif  /* _PPC_MSR_CE */
142 #endif  /* _VB_PISA_EHV */
143     };
```

```
总的说来：excVecConnectCommon 是对中断向量表做处理，根据中断向量表第一个参数vecoff,计算该类中断的内存地址位置，然后将存根机器代
/* copy the stub to the vector location */
bcopy((char *)stub, (char *)cVec, stubSize);

(4)excIvorInit 设置IVOR寄存器
macro in excPpcLib.h
/* Mappings between vector names and corresponding IVORs, for excALib.s */
#define IVOR0_VAL   _EXC_OFF_CRTL       /* Critical Input */
#if ((defined  PPC_440x5) || (CPU == PPC465))
#define IVOR1_VAL   _EXC_OFF_MCRECOV    /* recoverable Machine Check */
#else   /* PPC_440x5 || PPC465 */
#define IVOR1_VAL   _EXC_OFF_MACH       /* Machine Check */
#endif  /* PPC_440x5 || PPC465 */
#define IVOR2_VAL   _EXC_OFF_DATA       /* Data Storage */
#define IVOR3_VAL   _EXC_OFF_INST       /* Instruction Storage */
#define IVOR4_VAL   _EXC_OFF_INTR       /* External Input */
#define IVOR5_VAL   _EXC_OFF_ALIGN      /* Alignment */
#define IVOR6_VAL   _EXC_OFF_PROG       /* Program */
#define IVOR7_VAL   _EXC_OFF_FPU        /* Floating Point Unavailable */
#define IVOR8_VAL   _EXC_OFF_SYSCALL    /* System Call */
#define IVOR9_VAL   _EXC_OFF_APU        /* Auxiliary Processor Unavailable */
#define IVOR10_VAL  _EXC_OFF_DECR       /* Decrementer */
#define IVOR11_VAL  _EXC_OFF_FIT        /* Fixed Interval Timer */
#define IVOR12_VAL  _EXC_OFF_WD         /* Watchdog Timer */
#define IVOR13_VAL  _EXC_OFF_DATA_MISS  /* Data TLB Error */
#define IVOR14_VAL  _EXC_OFF_INST_MISS  /* Instruction TLB Error */
#define IVOR15_VAL  _EXC_OFF_DBG        /* Debug exception */

```

```
code in .\target\src\arch\ppc\excALib.s
FUNC_EXPORT(excIvorInit)
FUNC_BEGIN(excIvorInit)
        li      p0, IVOR0_VAL
        mtspr   IVOR0, p0
        li      p0, IVOR1_VAL
        mtspr   IVOR1, p0
        li      p0, IVOR2_VAL
        mtspr   IVOR2, p0
        li      p0, IVOR3_VAL
        mtspr   IVOR3, p0
        li      p0, IVOR4_VAL
...
FUNC_END(excIvorInit)

```


存根代码表：

```
LOCAL INSTR excConnectCode[]=
    {
    /*  data          word    byte     opcode  operands              */
    0x7c7343a6,     /*  0     0x00     mtspr   SPRG3, p0             */
#if     defined(_EXC_OFF_CRTL)
# if    defined(T4_ERRATUM_CPU6198) && defined(_WRS_CONFIG_SMP)
    0x7c6000a6,     /*  1     0x04     mfmsr   p0                    */
    0x546303da,     /*  2     0x08     rlwinm  p0,p0,0,15,13  clear MSR[CE] */
    0x7c7b8ba6,     /*  3     0x0c     mtspr   MCSRR1,p0             */
    0x3c600000,     /*  4     0x10     lis     p0,HI(mtmsrwa)        */
    0x60630000,     /*  5     0x14     ori     p0,p0,LO(mtmsrwa)     */
    0x7c7a8ba6,     /*  6     0x18     mtspr   MCSRR0,p0             */
    0x4c00004c,     /*  7     0x1c     rfmci                         */
# else
    0x7c6000a6,     /*  1     0x04     mfmsr   p0                    */
    0x546303da,     /*  2     0x08     rlwinm  p0,p0,0,15,13  clear MSR[CE] */
    0x7c600124,     /*  3     0x0c     mtmsr   p0                    */
    0x60000000,     /*  4     0x10     nop                           */
# endif /* T4_ERRATUM_CPU6198 && _WRS_CONFIG_SMP */
#elif   defined(_WRS_PPC_64BIT)
    0x7c6000a6,     /*  1     0x04     mfmsr   p0                    */
    0x786300c0,     /*  2     0x08     clrldi  p0,p0,3 clear MSR[SF] */
    0x7c600164,     /*  3     0x0c     mtmsrd  p0                    */
    0x4c00012c,     /*  4     0x10     isync                         */
#endif  /* _EXC_OFF_CRTL, _WRS_PPC_64BIT */
    /* If either of the above, add 4 words/0x10 bytes to following offsets */
    0x7c6802a6,     /*  1     0x04     mflr    p0                    */
    0x48000001,     /*  2(6)  0x08/18  bl      xxxEnt                */
    0x38610000,     /*  3     0x0c     addi    r3, sp, 0             */
    0x9421fff0,     /*  4     0x10     stwu    sp, -FRAMEBASESZ(sp)  */
    0x48000001,     /*  5(9)  0x14/24  bl      xxxHandler            */
    0x38210010,     /*  6     0x18     addi    sp, sp, FRAMEBASESZ   */
    0x48000001      /*  7(11) 0x1c/2c  bl      xxxExit               */
    };
```

系统起来后在 vxWorks shell 中输入 d 0x500（外部中断的内存地址），可以看到数据和 excConnectCode 存根一致

```
->
-> d 0x500
NOTE: memory values are displayed in hexadecimal.
0x00000500:  7c73 43a6 7c60 00a6 5463 03da 7c60 0124  *|sC.|`..Tc..|`.$*
0x00000510:  6000 0000 7c68 02a6 4819 8729 3861 0000  *`...|h..H..)8a..*
0x00000520:  9421 fff0 481d 3a9d 3821 0010 4819 89d5  *.!..H.:.8!..H...*
0x00000530:  bccd 6ee2 7bbc e78f e5dd 9fcb bf67 f6ff  *..n.{........g..*
0x00000540:  def8 7bef f32f 7df9 ea26 f3f1 67cc 1a72  *..{../}..&..g..r*
0x00000550:  32e2 edee 3ae1 2d69 b5ad 3bf9 0571 c55b  *2...:.-i..;..q.[*
0x00000560:  fa5f c94e a340 fef8 c313 cc5f 37f6 fea8  *._.N.@....._7...*
0x00000570:  86d7 8437 271a efce d3f5 cdf2 57dd 04e7  *...7'.......W...*
0x00000580:  255c e0da 5e79 9ff3 df9f 678a 5b2d 1355  *%\..^y....g.[-.U*
0x00000590:  fe9d 01fe 865a f755 ff9a c8b0 dfe5 cb11  *.....Z.U........*
0x000005a0:  2fdf e67e 783f dbfb 6dca 77ef ecf9 a7f7  */..~x?..m.w.....*
0x000005b0:  ace1 bfd9 67fa 3d93 bce6 70f1 b707 ffdd  *....g.=...p.....*
0x000005c0:  db0d 6fbb d1f6 75f0 ac9b e6f9 19a4 3ff4  *..o...u.......?.*
0x000005d0:  c1b7 c893 de38 8fd7 2c9a 9e7f 3bab a9db  *.....8..,...;...*
0x000005e0:  5557 3cd3 19ba 3bb7 daf2 e8f2 8227 dac6  *UW<...;......'..*
0x000005f0:  5d7d aafd e9d2 dda3 6a3b afea aa3b 617f  *]}.....
value = 0 = 0x0
->
```

中断向量内存图分析：

```
P2020 BSP 中使用的中断向量内存分布图:
-------------
|    0x0    |   vector base address
-------------
|   0x100   |   Critical Input
-------------
|   0x200   |   Machine Check
-------------
|   0x300   |   Data Storage
-------------
|   0x400   |   Instruction Storage
-------------
|   0x500   |   External Input[0x500开始处存放excConnectCode存根]
|  cVec[3]  |   存放c函数地址高16位 cVec = 0x500 [3] = EXT_ENT_OFF
|  cVec[4]  |   存放entInt函数地址低16位
|  cVec[8]  |   存放&excHandlers[n]地址，其中excHandlers存放默认的ISR,该函数会读取IACK
|  cVec[9]  |   [8] = EXT_ISR_OFF,存放ISR函数的低16位
|  cVec[15] |   存放exitInt函数地址高16位 [15] = EXT_EXIT_OFF
|  cVec[16] |   存放exitInt函数地址低16位
|    ...    |
-------------
|   0x600   |   Alignment
-------------
|   0x700   |   Program
-------------
|    ...    |   /* execPpcLib.h 有定义 */
-------------

总的说来中断处理流程为：中断产生后，e500内核根据MSR，IVPR,IVORs等寄存器来确认异常类型，
以及该类中断向量地址，然后进行存根机器代码-->entInt(汇编)保存现场-->ISR(确认中断源，
处理的functions)-->exitInt(汇编)恢复现场，至于PC<-->SRR0 ,MSR<-->SRR1是在存
还是在entInt与exitInt中操作，有待细化...
```