Application Note

AN2472/D
Rev. 0, 5/2003

MPC500 Enhanced Interrupt Controller (EIC)

Steve Mihalik
TSPG Applications

This application note addresses features of the enhanced interrupt controller (EIC) available on MPC56x/MPC53x devices. It is intended to supplement application note AN2109/D, *MPC555 Interrupts*. When using this document, AN2109/D should be used as a reference for general terms, supplementary information on interrupts and exceptions, and example interrupt routines.

Section 1, “Introduction,” introduces the new features of the EIC. Section 2, “Exception Vector Relocation Background,” describes exception vector relocation, which is typically used with the EIC. Section 3, “EIC Features,” gives details of the new EIC features. Section 4, “Interrupt Initialization Steps,” and Section 5, “Interrupt Handler Steps,” list the steps used to initialize and service the EIC. Section 6, “Program Examples,” provides sample code that can be used to implement the scenarios discussed in Section 5. Section 7, “Conclusion,” summarizes the overhead for several examples and lists other considerations. Appendix A, “IRQ Pin Behavior,” discusses how to clear the interrupt and how IRQ0 differs from the other IRQ pins.

The core of this application note is in its examples of how to initialize and service an interrupt from either the regular interrupt controller or the EIC. These steps are summarized the in the following tables:

- Table 11, “Interrupt Initialization Steps”
- Table 12, “Interrupt Handler Steps”

As shown in the conclusion, the time required to move from an interrupt request to an interrupt service routine (ISR) is typically under one microsecond for context saves that call a C function. For functions that need only a minimal assembly ISR, this response time is reduced to a few hundred nanoseconds. These timings assume a 56-MHz operation in non-serialized mode, executing from internal Flash.

1 Introduction

The MPC56x/MPC53x devices contain an EIC in addition to the backwards-compatible regular interrupt controller found in the MPC555/MPC556 devices. One of the advantages of the EIC is its ability to reduce latency by using one or more of the following new features:

- **An increased number of interrupt levels** that eliminates the need to decode the UIPEND register and reduces the need for level sharing. Each interrupt source in the IMB peripherals has its own level instead of sharing level 7 as in the regular interrupt controller. This effectively adds 32 additional interrupt levels.

- **External interrupt relocation** that provides automatic branching to a location for each interrupt level and pin instead of requiring that software decode either the SIVEC or SIPEND registers. This allows for a cleaner and faster design, especially when there are different amounts of context to be saved and restored.
- **Lower priority request masking** that eliminates the need, when nesting interrupts, to manipulate the SIMASK register at the start and end of interrupt service routines.

As discussed in AN2109/D, the terms “interrupt” and “exception” are not always consistent in documentation. For this application note, the term “interrupt” refers to external interrupt exceptions.

In addition, the term “interrupt handler” includes the following:

- Context save
- Determination of the interrupt source (if necessary)
- Branch to and execution of an interrupt service routine
- Context restore
- Return from interrupt

CAUTION

It is possible that re-enabling interrupts too quickly after servicing them will cause a false recognition of the same interrupt. This situation could occur if interrupts are re-enabled immediately after negating an interrupt. The propagation time for the negation of an interrupt request (for example, clearing the interrupt’s status bit) to reach the CPU core could take 5 or 6 clocks for IMB peripherals. If interrupts were immediately re-enabled, then the same interrupt could be falsely recognized again. Normally, this is not an issue since the negation propagation time is less than a few instructions.

Long interrupt routines, that save/restore processor context and especially GPRs, usually satisfy this requirement. Most likely this problem may occur for very short interrupt routines, written in assembler. The way to check that the interrupt is not re-enabled too fast is to watch IRQOUT pin and a instruction watchpoint, set on an instruction that re-enables the interrupts (as RFI, MTMSR or MTSPR EIE,Rx). The IRQOUT pin should negate before the watchpoint pulse. Of course, the IRQOUT and IWP*n* functions should be chosen on corresponding pins.

2 Exception Vector Relocation Background

The enhanced interrupt controller’s new interrupt features are typically used with exception table relocation. Exception table relocation “relocates” exception vectors to give them offsets of 8 bytes instead of 256, providing additional choices for the base of the table. The MPC56x/MPC53x devices offer more options for this feature than MPC55x devices; one of these options is the ability to locate the exception vector table in internal SRAM. Software examples included in this document will show how to use all of these new features.

Exception vector relocation was created primarily to save Flash space. The common practice of using a branch instruction in the vector instead of the entire exception routine wastes a significant number of bytes. Exception vector relocation reduces the number of wasted bytes by “relocating” exception vectors from offsets spaced 256 bytes (100 hex) apart to offsets 8 bytes (10 hex) apart, as shown in Table 2. With

relocation, unimplemented vectors, which normally use 256 bytes, use only 8 bytes, thereby reducing the number of unused bytes from 252 to 4. (The extra 4 bytes are necessary if using code compression, where a branch instruction may require more than 32 bits.)

The bits listed in Table 1 show the control bits for exception vector relocation. Note that the BBCMCR[OERC] bits in the MPC56x/MPC53x differ from those found in the MPC55x in the following ways:

1. There are two OERC bits, allowing more base offset choices for the exception vector table.
2. OERC bits are now in the reset configuration word, like the other control bits in the table.

Table 1. Exception Vector Relocation Control Bits

Register[Bit Field]	Bit Field Name	Description
MSR[IP]	Instruction Prefix	Controls the main base address, either at 0x0 or 0xFFF0 0000.
BBCMCR[ETRE]	Exception table relocation enable	Enables exception vector address relocation. Addresses are separated by 8 bytes instead of 256 bytes. (Requires MSR[IP] =1.)
BBCMCR[OERC0:1]	Other exception relocation enable	Provide additional offsets to the base address when relocation is used.
IMMR[ISB]	Internal memory space base	Moves exception table base with internal memory space. (Requires MSR[IP] = 1 and BBCMCR[ETRE] = 1.)

Two other MPC56x/MPC53x exception table relocation differences are also shown in Table 2:

1. The system reset/NMI interrupt relocated vector has a different address for the MPC56x/MPC53x if the device is in compression mode (using EN_COMP control bit).
2. The external interrupt relocated vector has different and multiple addresses for the MPC56x/MPC53x if the enable external interrupt relocation control bit is set (described in Section 3.2, “External Interrupt Relocation (EIR)”).

Table 2. Exception Vector Table Alternatives

Exception	MPC 5xx without relocation		MPC55x with relocation		MPC56x/MPC53x with relocation			
	Exception Vector							
MSR[IP]	IP = 0	IP = 1	IP = 1	IP = 1	IP = 1	IP = 1	IP = 1	IP = 1
BBCMCR[ETRE]	—	ETRE = 0	ETRE = 1	ETRE = 1	ETRE = 1	ETRE = 1	ETRE = 1	ETRE = 1
BBCMCR[OERC]	—	—	OERC = 0	OERC = 1	OERC = 00	OERC = 01	OERC = 10	OERC = 11
IMMR[ISB]	—	—	ISB = 000	ISB = 000	ISB = 000	ISB = 000	ISB = 000	ISB = 000
Reserved	0x0000 0000	0xFFF0 0000	0x0000 0000	0x0000 8000	0x0000 0000	0x0001 0000	0x0008 0000	0x003F E000
System Reset, NMI Interrupt	0x0000 0100	0xFFF0 0100	0x0000 0008	0x0000 0008 [1]	—	—	—	—
EN_COMP= 0	—	—	—	—	0x0000 0008	0x0001 0008	0x0008 0008	0x003F E008
EN_COMP= 1	—	—	—	—	0x0000 00B8	0x0001 00B8	0x0008 00B8	0x003F E0B8
Machine Check	0x0000 0200	0xFFF0 0200	0x0000 0010	0x0000 8010	0x0000 0010	0x0001 0010	0x0008 0010	0x003F E010
Reserved	0x0000 0300	0xFFF0 0300	0x0000 0018	0x0000 8018	0x0000 0018	0x0001 0018	0x0008 0018	0x003F E018
Reserved	0x0000 0400	0xFFF0 0400	0x0000 0020	0x0000 8020	0x0000 0020	0x0001 0020	0x0008 0020	0x003F E020
External Interrupt	0x0000 0500	0xFFF0 0500	0x0000 0028	0x0000 8028	—	—	—	—
EIR inactive	—	—	—	—	0x0000 0028	0x0001 0028	0x00008 0028	0x003F E028
EIR active	—	—	—	—	EIR Table	EIR Table	EIR Table	EIR Table
Alignment	0x0000 0600	0xFFF0 0600	0x0000 0030	0x0000 8030	0x0000 0030	0x0001 0030	0x0008 0030	0x003F E030
Program	0x0000 0700	0xFFF0 0700	0x0000 0038	0x0000 8038	0x0000 0038	0x0001 0038	0x0008 0038	0x003F E038
Floating-point Unavailable	0x0000 0800	0xFFF0 0800	0x0000 0040	0x0000 8040	0x0000 0040	0x0001 0040	0x0008 0040	0x003F E040
Decrementer	0x0000 0900	0xFFF0 0900	0x0000 0048	0x0000 8048	0x0000 0048	0x0001 0048	0x0008 0048	0x003F E048
Reserved	0x0000 0A00	0xFFF0 0A00	0x0000 0050	0x0000 8050	0x0000 0050	0x0001 0050	0x0008 0050	0x003F E050
Reserved	0x0000 0B00	0xFFF0 0B00	0x0000 0058	0x0000 8058	0x0000 0058	0x0001 0058	0x0008 0058	0x003F E058
System Call	0x0000 0C00	0xFFF0 0C00	0x0000 0060	0x0000 8060	0x0000 0060	0x0001 0060	0x0008 0060	0x003F E060
Trace	0x0000 0D00	0xFFF0 0D00	0x0000 0068	0x0000 8068	0x0000 0068	0x0001 0068	0x0008 0068	0x003F E068

Table 2. Exception Vector Table Alternatives (continued)

Exception	MPC 5xx without relocation		MPC55x with relocation		MPC56x/MPC53x with relocation			
Floating Point Assist	0x0000 0E00	0xFFF0 0E00	0x0000 0070	0x0000 8070	0x0000 0070	0x0001 0070	0x0008 0070	0x003F E070
served	0x0000 0F00	0xFFF0 0F00	0x0000 0078	0x0000 8078	0x0000 0078	0x0001 0078	0x0008 0078	0x003F E078
Software Emulation	0x0000 1000	0xFFF0 1000	0x0000 0080	0x0000 8080	0x0000 0080	0x0001 0080	0x0008 0080	0x003F E080
Reserved	0x0000 1100	0xFFF0 1100	0x0000 0088	0x0000 8088	0x0000 0088	0x0001 0088	0x0008 0088	0x003F E088
Reserved	0x0000 1200	0xFFF0 1200	0x0000 0090	0x0000 8090	0x0000 0090	0x0001 0090	0x0008 0090	0x003F E090
Instruction Protection Error	0x0000 1300	0xFFF0 1300	0x0000 0098	0x0000 8098	0x0000 0098	0x0001 0098	0x0008 0098	0x003F E098
Data Protection Error	0x0000 1400	0xFFF0 1400	0x0000 00A0	0x0000 80A0	0x0000 00A0	0x0001 00A0	0x0008 00A0	0x003F E0A0
Reserved	0x0000 1500– 0x0000 1BFF	0xFFF0 1500– 0xFFF0 1BFF	0x0000 00A8– 0x0000 00DF	0x0000 80A8– 0x0000 80DF	0x0000 00A8– 0x0000 00DF	0x0001 00A8– 0x0001 00DF	0x0008 00A8– 0x0008 00DF	0x003F E0A8– 0x003F E0DF
Data Breakpoint	0x0000 1C00	0xFFF0 1C00	0x0000 00E0	0x0000 80E0	0x0000 00E0	0x0001 00E0	0x0008 00E0	0x003F E0E0
Instruction Breakpoint	0x0000 1D00	0xFFF0 1D00	0x0000 00E8	0x0000 80E8	0x0000 00E8	0x0001 00E8	0x0008 00E8	0x003F E0E8
Maskable External Breakpoint	0x0000 1E00	0xFFF0 1E00	0x0000 00F0	0x0000 80F0	0x0000 00F0	0x0001 00F0	0x0008 00F0	0x003F E0F0
Non-Maskable External Breakpoint	0x0000 1F00	0xFFF0 1F00	0x0000 00F8	0x0000 80F8	0x0000 00F8	0x0001 00F8	0x0008 00F8	0x003F E0F8

[1] System reset / NMI uses 0x0000 0008 instead of 0x0000 8008 because HRESET clears the OERC bit. Note that the type of reset should be taken into account since HRESET changes BBCMCR but SRESET does not. Also, the reset configuration word (RCW) value for the MPC56x/MPC53x defines the BBCMCR[OERC] field upon HRESET assertion.

3 EIC Features

3.1 Increased Number of Interrupt Levels

Description: Enabling the EIC automatically increases the number of interrupt levels from 8 to 40. With the inclusion of the 8 IRQ input pins, the decoding capability increases to 48 interrupt sources. The regular interrupt controller has 16 interrupt sources, as is shown in Table 10. The new 32 levels are available for IMB peripheral devices; USIU devices continue to use interrupt levels 0 through 7.

Benefit: The EIC's increased number of interrupt levels means that there are additional Interrupt_Code bit field values in the SIVEC register, thus eliminating the need to include software for decoding levels 8–31 in the UIPEND register. The additional levels also have their own MASK and PEND bits in the associated registers.

NOTE

Some peripherals, such as the TPU, have one level that corresponds to multiple interrupt sources (TPU channels) inside the peripheral. Once the level identifies the peripheral, it may be necessary to further identify, with software, the source inside that peripheral.

New Bits/Registers: The EIC is enabled with a new control bit as shown in Table 3. Once the EIC is enabled, the SIMASK and SIPEND registers are no longer used and are replaced by the new registers listed in Table 4. In addition, the SIVEC[Interrupt_Code] bit field values change as shown in table 5.

Table 3. EIC Control Bit

Register[Bit Field]	Bit Field Name	Description
SIUMCR[EICEN]	EIC enable	When set, enables EIC. Reset default: the regular interrupt controller is enabled instead of the EIC.

Table 4. EIC Registers

Register(s)	Name	Description
SIMASK2 SIMASK3	SIU interrupt mask register 2 SIU interrupt mask register 3	Both registers replace the SIMASK register used with the regular interrupt controller. Incorporates additional levels for masking interrupts.
SIPEND2 SIPEND3	SIU interrupt pending register 2 SIU interrupt pending register 3	Both registers replace SIPEND register used with the regular interrupt controller. Incorporates additional levels for identifying pending interrupts.

Table 5. Interrupt Codes and Offsets for Regular Interrupt Controller and EIC

Interrupt Source	Regular Controller			Enhanced Controller			
	Number	SIVEC [Interrupt Code]		Number	SIVEC [Interrupt Code]		Offset if Using EIR Branch Table (Hex)
		Binary	Hex		Binary	Hex	
External IRQ0/ (Input Pin [1])	0 (highest priority)	00000000	0x00	0 (highest priority)	00000000	0x00	0x0000
Level 0	1	00000100	0x04	1	00000100	0x04	0x0008
IMB_IRQ0	—	—	—	2	00001000	0x08	0x0010
IMB_IRQ1	—	—	—	3	00001100	0x0C	0x0018
IMB_IRQ2	—	—	—	4	00010000	0x10	0x0020
IMB_IRQ3	—	—	—	5	00010100	0x14	0x0028
External IRQ1/ (Input Pin)	2	00001000	0x08	6	00011000	0x18	0x0030
Level 1	3	00001100	0x0C	7	00011100	0x1C	0x0038
IMB_IRQ4	—	—	—	8	00100000	0x20	0x0040
IMB_IRQ5	—	—	—	9	00100100	0x24	0x0048
IMB_IRQ6	—	—	—	10	00101000	0x28	0x0050
IMB_IRQ7	—	—	—	11	00101100	0x2C	0x0058
External IRQ2/ (Input Pin)	4	00010000	0x10	12	00110000	0x30	0x0060
Level 2	5	00010100	0x14	13	00110100	0x34	0x0068
IMB_IRQ8	—	—	—	14	00111000	0x38	0x0070
IMB_IRQ9	—	—	—	15	00111100	0x3C	0x0078
IMB_IRQ10	—	—	—	16	01000000	0x40	0x0080
IMB_IRQ11	—	—	—	17	01000100	0x44	0x0088
External IRQ3/ (Input Pin)	6	00011000	0x18	18	01001000	0x48	0x0090
Level 3	7	00011100	0x1C	19	01001100	0x4C	0x0098
IMB_IRQ12	—	—	—	20	01010000	0x50	0x00A0
IMB_IRQ13	—	—	—	21	01010100	0x54	0x00A8
IMB_IRQ14	—	—	—	22	01011000	0x58	0x00B0
IMB_IRQ15	—	—	—	23	01011100	0x5C	0x00B8
External IRQ4/ (Input Pin)	8	00100000	0x20	24	01100000	0x60	0x00C0
Level 4	9	00100100	0x24	25	01100100	0x64	0x00C8
IMB_IRQ16	—	—	—	26	01101000	0x68	0x00D0
IMB_IRQ17	—	—	—	27	01101100	0x6C	0x00D8
IMB_IRQ18	—	—	—	28	01110000	0x70	0x00E0

Table 5. Interrupt Codes and Offsets for Regular Interrupt Controller and EIC (continued)

Interrupt Source	Regular Controller			Enhanced Controller			
	Number	SIVEC [Interrupt Code]		Number	SIVEC [Interrupt Code]		Offset if Using EIR Branch Table (Hex)
		Binary	Hex		Binary	Hex	
IMB_IRQ19	—	—	—	29	01110100	0x74	0x00E8
External IRQ5/ (Input Pin)	10	00101000	0x28	30	01111000	0x78	0x00F0
Level 5	11	00101100	0x2C	31	01111100	0x7C	0x00F8
IMB_IRQ20	—	—	—	32	10000000	0x80	0x0100
IMB_IRQ21	—	—	—	33	10000100	0x84	0x0108
IMB_IRQ22	—	—	—	34	10001000	0x88	0x0110
IMB_IRQ23	—	—	—	35	10001100	0x8C	0x0118
External IRQ6/ (Input Pin)	12	00110000	0x30	36	10010000	0x90	0x0120
Level 6	13	00110100	0x34	37	10010100	0x94	0x0128
IMB_IRQ24	—	—	—	38	10011000	0x98	0x0130
IMB_IRQ25	—	—	—	39	10011100	0x9C	0x0138
IMB_IRQ26	—	—	—	40	10100000	0xA0	0x0140
IMB_IRQ27	—	—	—	41	10100100	0xA4	0x0148
External IRQ7/ (Input Pin)	14	00111000	0x38	42	10101000	0xA8	0x0150
Level 7	15 (lowest priority)	00111100	0x3C	43	10101100	0xAC	0x0158
IMB_IRQ28	—	—	—	44	10110000	0xB0	0x0160
IMB_IRQ29	—	—	—	45	10110100	0xB4	0x0168
IMB_IRQ30	—	—	—	46	10111000	0xB8	0x0170
IMB_IRQ31	—	—	—	47 (lowest priority)	10111100	0xBC	0x0178
Reserved	16-31	—	—	—	—	—	—

1 The IRQ0/input pin is a special case that causes a non-maskable exception. Though it uses the reset vector, it does not cause a reset. If the IRQ0 pin is used, it is recommended that the reset vector be used to service it (instead of attempting to use the interrupt vector). See Appendix A, “IRQ Pin Behavior,” for more information.

3.2 External Interrupt Relocation (EIR)

Description: EIR relocates external interrupt exceptions to 1 of 48 unique addresses instead of a single interrupt exception vector table address. Each of the 40 EIC levels and 8 IRQ input pins are automatically decoded and directed their own vector address by program execution. These vector addresses are based on a unique, user-defined (in the EIBADR) offset for that interrupt. See Table 9 for more information.

Benefits: One benefit of EIR is that it speeds up interrupt processing by eliminating the need to determine which level or IRQ pin caused the interrupt, a step that requires about seven assembly instructions. (See AN2109/D, Section 7.3.3.2, “Example 3: exceptions.s File,” Step 4 for more information.) Eliminating these seven instructions reduces the net instruction overhead.

NOTE

An exception routine that calls C functions will have additional overhead because it must save and restore at least 17 registers on the stack (see AN2109/D, Sections 6.1 and 6.2). Although EIR eliminates seven instructions, it is likely to require one of the following trade-offs:

1. Additional code. Code that saves and restores registers (context) on the stack must be duplicated for each interrupt, thereby increasing overall code size. If ROM size is not limited, this trade-off is not an issue.

2. Reduction in time saved. If common save and restore routines are used by the interrupt exceptions (to reduce ROM size), extra instructions must be inserted to use them. These extra instructions reduce some of the time saved by eliminating the decode instructions.

A potential large benefit of EIR is that it simplifies the ability to configure interrupt overhead for individual interrupts. This is important because interrupt exception routine overhead can vary dramatically depending on the number of registers saved. For example, an interrupt exception routine that calls a C routine and saves all 32 general purpose registers plus a few others can have an overhead of 99 instructions. However, an interrupt exception routine using only assembler can have an overhead of just 37 instructions—a significant reduction. (See AN2109/D, Section 8, Table 2 for more information.) Therefore, it may be desirable to have different context switch scenarios: one for critical interrupt routines written entirely in assembly language and one for normal interrupt routines that call C functions.

If EIR is not used, different context switches can still be implemented, but the design is not as clean as when the EIR feature is used. For example, some initial context, such as the machine state, would likely be saved, then the interrupt source decoded and branched to, then a second context implemented for a various amount of additional registers. (See AN2109/D, Sections 6.1 and 6.2 for more information.) EIR simplifies designs because context is saved at a vector for each interrupt source. The same is true for a restoring context(s) at the end of the software handler routine.

New Bit Fields/Registers: The EIR feature is controlled by a new bit shown in Table 6. The base of the external relocation table listed in Table 7 must be initialized.

Table 6. EIR Control Bit

Register[Bit Field]	Bit Field Name	Description
BBCMCR[EIR]	Enhanced EIR Enable	When set, activates the external interrupt relocation function. Reset default: EIR function is disabled.

Table 7. EIR Register

Register(s)	Name	Description
EIBADR	External Interrupt Relocation Table Base Address Register	Contains the base address for the external interrupt relocation table. Only the most significant 20 bits are used.

3.3 Lower Priority Request Masking (LPRM)

Description: LPRM is designed to simplify the interrupt handler routines that nest interrupts. When re-enabling interrupts (by setting MSR[EE] = 1), it is usually desirable to prevent the recognition of interrupts with a priority lower than or identical to the one being serviced.

LPRM masks the lower and same-priority interrupts during external interrupt exception routines. Once this feature is enabled, only higher priority interrupts will be recognized during an external interrupt exception (after interrupts are re-enabled).

Benefits: LPRM is beneficial in that it simplifies and speeds up the masking of lower and same-priority interrupt requests by eliminating the need to manipulate the SIMASK2 and SIMASK3 registers, resulting in less overall software overhead. Table 8 compares the software steps involved in an external interrupt exception routine that nests interrupts.

Table 8. Comparison of Steps for Nesting Interrupts in External Interrupt Exception Routine

	Without LRPM	With LRPM
During prolog of external interrupt exception service routine	1. Save current SIMASK values on stack 2. Determine current interrupt request 3. Clear same and lower priority bits in SIMASK	—
During epilog of external interrupt exception service routine	Restore SIMASK from stack	Clear the corresponding SISR bit for that interrupt.

New Bit Fields/Registers: LPRM is enabled with a new control bit, as shown in Table 9. The SISR registers, shown in Table 10, contain bits that reflect which priority interrupt is in service. Software must clear the appropriate bit at the end of the external interrupt exception routine by writing a 1 to it.

Table 9. LPRM Control Bit

Register[Bit Field]	Bit Field Name	Description
SIUMCR[LP_MASK]	Low Priority Masking Request Enable	When set, enables LPRM. Reset default: disabled.

Table 10. LPRM Registers

Register(s)	Name	Description
SISR2 SISR3	Interrupt In-Service Register 2 Interrupt In-Service Register 3	Each bit corresponds to an interrupt request bit from SIPEND2 and SIPEND3 which is not masked by SIMASK2 and SIMASK3. Once the hardware sets a bit, same- and lower-priority interrupts are not recognized until software clears the bit.

4 Interrupt Initialization Steps

The steps used to initialize the regular interrupt controller and EIC features are listed below. Table 11 summarizes these steps based on the following examples (which are discussed in Section 5 and coded in Section 6):

1. Using regular interrupt controller only
2. Using the EIC with the additional levels
3. Using the EIC with external interrupt relocation
4. Using the EIC with external interrupt relocation and lower priority request masking

Step 1: Initialize the Exception Table Relocation Control Bits

The exception table relocation feature is enabled by setting MSR[IP] and BBCMCR[ETRE]. The exception table base is selected by setting appropriate BBCMCR[OERC] bits. As seen in Table 2, internal SRAM is one choice for the table base. Using internal SRAM for an exception table has the following benefits:

- It eliminates the need to program the Flash when external SRAM is not available.
- It allows dynamically changing exception processing during application execution.

All exception table relocation control bits are first initialized at the negation of the HRESET signal from the reset configuration word. This tells the processor where the reset vector is located, that is, where to start executing code. It is not necessary to set these bits again in software unless a change is desired.

Step 2: Initialize the Interrupt Controller

If at least one peripheral uses interrupt levels above 7, then the UMCR[IRQMUX] bits must be changed from their reset value to the bit settings shown in the table below. For example, if the peripherals used up to interrupt level 22, then IRQMUX should be set to 2. This allows the interrupt controller (regular or enhanced) to see from levels 0 to 23.

Highest Interrupt Level Used	UMCR[IRQMUX]
7	0 (reset value)
15	1
23	2
31	3

The EIC must be enabled (by setting the SIUMCR[EIC] bit) before any of the EIC features can be used. If the external interrupt relocation feature is used, the EIBADR register must be initialized to set the base for external interrupt exceptions and setting the BBCMCR[EIR] bit. In addition, the SIUMCR[LPMASK_EN] bit must be set if using the lower priority request mask feature.

Step 3: For Each Module

Each module will have similar steps for each interrupt source. These steps are repeated from the regular interrupt controller as listed in application note AN2109, with the following exceptions:

- The EIC has additional levels, so peripheral level mapping will likely be different for optimal design.
- The EIC uses the SIMASK2 and SIMASK3 registers instead of SIMASK for masking interrupts.

The steps below are ordered logically, but may not provide the most efficient code.

Step 3.1: Module-Specific Initialization

Specific initializations (such as setting baud rate for serial channels, setting clocks and dividers for timed I/O, setting queue management for buffered peripherals, and assigning functions to channels) are performed for each device. A complete list is outside the scope of this document.

Step 3.2: Level Assignment

As stated in AN2109, levels must be assigned carefully since they imply priority. Following are some key principles of level assignment:

- Lower levels have higher priority.
- External interrupt pins have fixed priorities instead of level assignments.
- To reduce latency, each interrupt source should be mapped to its own level.
- If the regular interrupt controller is used, the UMCR[IRQMUX] field should eventually be set appropriately if UIMB peripherals use levels greater than 7. (This step should be taken after all interrupt initializations are complete in order to prevent a situation in which a value is set in one initialization routine, then changed to a conflicting value in another initialization routine.)
- If the EIC is used, the additional 32 levels should be used to reduce or eliminate level sharing.
- Level registers for UIMB peripherals use either a single 5-bit field or a combination of 3- and 2-bit fields as discussed in Section 3.6 of AN2109.

Step 3.3: Enable Interrupt Sources

Each interrupt source (except IRQ pins) must be enabled. On-chip modules have individual enable bits which must be set.

Step 3.4: Set Appropriate Mask Bits in SIMASK or SIMASK2 and SIMASK3

Individual interrupt levels are masked by corresponding bits in the SIMASK register for the regular interrupt controller, and SIMASK2 and SIMASK3 for the EIC. Setting a mask bit to 1 enables interrupt levels to pass to the interrupt controller.

Step 4: Set MSR[EE]

After all the interrupt sources have been initialized, the external interrupts must be enabled by setting the MSR[EE] bit. Since the processor is recoverable now” the MSR[RI] bit should also be set. See AN2109 for additional discussion on these two bits. Both bits can be set by writing to the EIE special purpose register, as in the following example:

```
mtspr       EIE, r0         ; Set MSR[EE] and MSR[RI] bits
```

Table 11. Interrupt Initialization Steps

pt Initialization Steps		Example 1 Regular Interrupt Controller w/o Exception Table Relocation	Example 2 EIC w/o Exception Table Relocation	Example 3 EIC • Exception Table Relocation • External Interrupt Relocation	Example 4 EIC • Exception Table Relocation • External Interrupt Relocation • Lower Priority Request Masking
1. Initialize exception table relocation control bits. [1]		BBCMCR[ETRE] [2] = 0		MSR[IP][2]= 1 BBCMCR[ETRE][2] = 1 BBCMCR[OERC][2] = 00, 01, 10 or 11	
2. Initialize interrupt controller.		Set UMCR[IRQMUX]	Set UMCR[IRQMUX] SIUMCR[EICEN] = 1	Set UMCR[IRQMUX] SIUMCR[EICEN] = 1 Load EIBADR BBCMCR[EIR] = 1	Set UMCR[IRQMUX] SIUMCR[EICEN] = 1 Load EIBADR BBCMCR[EIR] = 1 SIUMCR[LPMASK_EN] =1
3. For each module:	• 3.1 Specific initialization	Set up specific module initialization, e.g. SCI: baud rate, etc.			
	• 3.2 Assign level	Assign level for each module.	Use additional new level numbers.		
	• 3.3 Enable interrupt	Set the interrupt enable bit for each module.			
	• 3.4 Set appropriate mask	Set appropriate SIMASK bit to recognize interrupt.	Set appropriate SIMASK2 and SIMASK3 bit instead of SIMASK bit to recognize interrupt.		
4. Set MSR[EE]		Write to SPR EIE.			

1 Initialized in the reset configuration word (except OERC in MPC55x) and typically do not need to be changed by software.
2 Initialized at HRESET from reset configuration word.

5 Interrupt Handler Steps

The steps used to service an interrupt from the regular controller or EIC are discussed below. Since they parallel the steps described for the regular interrupt controller in AN2109, the descriptions below are summarized from AN2109 for steps that are the same.

Table 12 summarizes these steps for the four examples described in Section 4 and Section 6.

Step 1: Save Machine Context

"Machine context" refers to special purpose registers SRR0 and SRR1. Once an external interrupt exception is recognized, hardware loads SRR0 and SRR1 with the state of the machine (from the MSR) and the address of the next instruction that was to be executed. Software should save these two registers on the stack, which requires a scratch register that also must be saved.

Step 2: Indicate Recoverable State (MSR[RI] = 1) and Re-enable Interrupts (If Nesting)

Any new exception will cause hardware to overwrite new values to SRR0 and SRR1. However, since their values are saved on the stack, the machine can recover. Software should set the MSR[RI] bit to indicate this recoverable state. Writing to the SPR EID sets the RI bit while keeping the EE bit clear, so further interrupts are disabled. An example of this instruction is:

```
mtspr    EID, r0          ;Set MSR[RI] to indicate recoverable state
```

However, if nested interrupts are desired and lower priority request masking is enabled, the external interrupt exception can be re-enabled. Interrupts can be re-enabled by setting the MSR[EE] bit. The simplest way to accomplish this is to write to the SPR EIE (instead of SPR EID), which sets *both* the MSR[EE] and MSR[RI] bits. An example of this instruction is:

```
mtspr    EIE, r0          ;Set MSR[RI] to indicate recoverable state
                          ;and set MSR[EE] to re-enable interrupts
```

NOTE

If the LPRM feature is not used for nested interrupts, SIMASK register(s) must be saved and modified before setting the MSR[EE] bit. Modification of the SIMASK register(s) would be to temporarily mask off further interrupts at that level or lower.

Step 3: Save Other Appropriate Context

"Other appropriate context" is any other register on the stack that might be changed during the exception routine. If writing in assembler, "other appropriate context" includes just a few registers. If calling a C function, it includes additional registers. Sometimes, saving "other appropriate context" includes all the GPRs plus a few others. The number of registers that must be saved as "other appropriate context" impacts the total overhead of the exception. For additional information, see AN2109, Section 5, "Determining Which Registers to Save and Where to Save Them," and Section 8, "Conclusion."

Step 4: Determine Interrupt Source

Without using the External Interrupt Relocation Feature. While the SIPEND register can be used to determine the level or IRQ pin that causes an interrupt, a more convenient method is to read the SIVEC[InterruptCode] value listed in Table 10. The value is added to the base of a branch table and the resulting address is loaded into a register for branching. It is important to remember that the interrupt codes differ for the regular interrupt controller and EIC.

If using the regular interrupt controller and levels above seven, the UIPEND register is used. The first "1" from the left is the highest priority pending interrupt. Using the cntzw instruction, the number of zeroes

before the first one is counted. The resulting number is used as an index to another branch table for levels 7-31. See exceptions.s Example 1 for an illustrative implementation.

Using the External Interrupt Relocation Feature. The EIR automatically branches interrupts to a vector unique to that level, so it is not necessary to decode a level or IRQ pin.

Step 5: Execute Interrupt Service Routine (ISR)

The interrupt handler will need to take some additional interrupt-related steps depending on whether the features listed below are used.

Step 5.1: Branch to and Execute ISR

The ISR for a particular level or IRQ pin is reached via the branch table. The interrupt request condition must be negated in the ISR, otherwise the interrupt may be falsely recognized a second time after exiting from the external interrupt exception routine.

Step 5.2: Clear Masking in Lower Priority Interrupts (If Nesting)

When interrupt nesting is enabled, it is assumed that interrupts of the same and lower priority have already been temporarily masked off. This can be accomplished by directly saving and modifying the SIMASK or the SIMASK2 and SIMASK3 registers, or by using the lower priority masking feature in the EIC.

At this time, the original masking must be restored by removing the temporary mask. Using LPRM, this means clearing the bit in the SISR corresponding the current priority.

SISR bits are cleared by setting them to 1; clearing them to 0 has no effect. It is important not to fall into the trap of using a C statement like the following:

```
USIU.SISR2.B.IMBIRQ6 = 1;          Incorrect method for clearing the IMBIRQ6 bit
```

The above statement normally produces code which reads the SISR2 register, inserts a 1 into the appropriate location, then writes it back. All bit locations in that register originally having 1s get those same 1s written back, clearing the register! Instead, use a statement such as the one below which writes a single 1 to the desired bit location.

```
USIU.SISR2.R = 0x00200000;         Correct method to clear IMBIRQ6 bit
```

Step 6: Restore Contexts

Now the registers previously stored on the stack should be restored. It is important that the MSR[RI] status bit is cleared before SRR0 and SRR1 are restored, since an exception during that time may make the machine unrecoverable.

Step 7: Return to Program

Finally, a simple return from interrupt (rfi) instruction is used to return to the next instruction after the interrupt was taken in the prior code and restore the state of the machine.

Table 12. Interrupt Handler Steps

Interrupt Handler Steps		Example 1 Regular Interrupt Controller	Example 2 EIC	Example 3 EIC • Exception Table Relocation • External Interrupt Relocation	Example 4 EIC • Exception Table Relocation • External Interrupt Relocation • Lower Priority Request Masking
1. Save "Machine Context."		Save SRR0 and SRR1.		Automatically branch to unique interrupt routine, then save SRR0 and SRR1.	
2.1 Indicate the state is recoverable.		Set MSR[RI] = 1 by writing to SPR EID.			Set MSR[EE] = 1 and MSR[RI] = 1 by writing to SPR EIE.
2.2 If nesting interrupts, re-enable them.		—			Re-enable interrupts.
3. Save other appropriate context (registers).		Save GPRs, etc. as necessary.		Save GPRs, etc. as necessary. Amount to save may vary by interrupt service routine.	
4. Determine interrupt source.		Decode SIVEC or SIPEND. If level > 7, also decode UIPEND.	Decode SIVEC (new values) or SIPEND2 and SIPEND3.	—	
5. Execute interrupt ISR.	5.1 Branch to ISR. (Negate any interrupt request in handler.)	Use branch table(s).	Use expanded branch table.	No branch table used. Each level and pin has unique starting address.	
	5.2 If nesting, undo masking of lower priority interrupts.	—			Clear appropriate bit in register SISR2 and SISR3.
6. Restore contexts (Indicate state is irrecoverable before restoring SRR0 and SRR1).		Restore registers, clearing MSR[RI] appropriately near end of routine.		Restore registers, clearing MSR[RI] appropriately near end of routine. Amount to restore may vary by interrupt service routine.	
7. Return to program.		Execute rfi instruction.			

6 Program Examples

The following examples illustrate different techniques for handling interrupt exceptions. The registers saved and restored (“context”) in the interrupt exception routines are determined by the assumption that a C function is called. Fewer registers need to be saved/restored if only assembly language is used; conversely, more registers need to be saved/restored if an operating system is used.

The four example interrupt programs use the following interrupt features:

- Example 1: regular interrupt controller
- Example 2: EIC
- Example 3: EIC with external interrupt relocation (EIR)
- Example 4: EIC with external interrupt relocation & lower priority request masking

Two interrupt devices are used for all examples as in the table listed below. Example 4 uses a third interrupt source of an IRQ pin that is driven by a MIOS GPIO pin during a different interrupt service routine. This forces a nested interrupt condition.

Level / IRQ Pin	Source	Condition
Level 6 from IMB	MIOS Counter Sub Module 6	Overflow
Level 22 from IMB	MIOS Counter Sub Module 22	Overflow
IRQ1 and IRQ3 [Example 4 only]	IRQ1 and IRQ3 input pins	MIOS GPIO pin driven low during Level 22 interrupt service routine.

Interrupt initialization in this document is written for illustration, not necessarily to optimize code.

Processor initialization is minimal for these examples. Following is a list of items that often need initialization:

1. SYPCR: disable watchdog timer
2. SIUMCR: disable data show cycles
3. PLPRCR: increase clock frequency and optionally wait for PLL to lock
4. UMCR: set UIMB bus to full speed using HSPEED bit
5. SPR560 (BBCMCR): enable burst buffer, enable branch target buffers
6. SPR158 (ICTRL): Increase processing speed by exiting serialized mode

The following files are used for each example:

main.c	C routines including: • Simple initialization for CPU • Initialization of interrupt sources • initialization of interrupt controller features • Interrupt service routines called from exception.s
exceptions.s	Assembly routine which performs the interrupt handler, including calling appropriate interrupt service routine
makefile	Common for all examples other than changing the EXECUTABLE name.
link file	Common for all examples.

6.1 Sample Makefile Used in Examples

```
# makefile_regular:  makes sample program using regular interrupt controller
# Used with DiabData compiler version 4.4a

OBJS               = main_regular.o exceptions_regular.o

CC      = dcc
AS      = das
LD      = dcc
DUMP    = ddump

HEADER_PATH = d:\mydoc555\m565r101

COPTS   = -tPPC555EH:cross -@E+err.log -g -c -O -I$(HEADER_PATH)
AOPTS   = -tPPC555EH:cross -@E+err.log -g
LOPTS   = -tPPC555EH:cross -@E+err.log -Ws -m2 -lm -l:crt0.o
EXECUTABLE = irq_regular


.SUFFIXES: .c .s

default: $(EXECUTABLE).elf $(EXECUTABLE).s19

.c.o :
          $(CC) $(COPTS) -o $*.o $<

.s.o :
          $(AS) $(AOPTS) $<

$(EXECUTABLE).elf: makefile $(OBJS)
          $(LD) $(LOPTS) $(OBJS) -o $(EXECUTABLE).elf -Wm etas_evb.lin > $(EXECUTABLE).map
          $(DUMP) -tv $(EXECUTABLE).elf >>$(EXECUTABLE).map

# Generate s record file for flashing

$(EXECUTABLE).s19: $(EXECUTABLE).elf
          $(DUMP) -Rv -o $(EXECUTABLE).s19 $(EXECUTABLE).elf
Link file for all examples:

/* etas_evb.lin General link file for MPC5xx */
/* Memory locations 0 - 0x1ffC are reserved for exception table. */
/* Memory locations 0x2000 -0x2080 are reserved for Ext Interrupt Relocation Table */
```

```
MEMORY
{
            internal_flash: org = 0x2080, len = 0x5df90
    internal_ram:             org = 0x3f9800, len = 0x67F0
}

SECTIONS
{
          GROUP : {
              .text (TEXT)   : {
                    *(.text)
                    *(.rodata)
                    *(.init)
                    *(.fini)
                    *(.eini)
                    . = (.+15) & ~15;
              }
              .sdata2 (TEXT) : {}
          } > internal_flash

          GROUP : {
              .data  (DATA) LOAD(ADDR(.sdata2)+SIZEOF(.sdata2)) : {}
              .sdata (DATA) LOAD(ADDR(.sdata2)+SIZEOF(.sdata2)+SIZEOF(.data)) : {}
              .sbss  (BSS) : {}
              .bss   (BSS) : {}
          } > internal_ram
}

__SP_INIT       = ADDR(internal_ram)+SIZEOF(internal_ram);
__SP_END        = ADDR(internal_ram);
__DATA_ROM      = ADDR(.sdata2)+SIZEOF(.sdata2);
__DATA_RAM      = ADDR(.data);
__DATA_END                      = ADDR(.sdata)+SIZEOF(.sdata);
__BSS_START     = ADDR(.sbss);
__BSS_END       = ADDR(.bss)+SIZEOF(.bss);

__HEAP_START                = ADDR(.bss)+SIZEOF(.bss);
__HEAP_END                            = ADDR(internal_ram)+SIZEOF(internal_ram);
```

6.2 Example 1: Regular Interrupt Controller

This example is a baseline that can be used to compare the different feature sets of the EIC. It is similar to Example 3 published in AN2109.

C Source: Initializes the chip, interrupt functions, MIOS counters 6 and 22 to cause interrupts on levels 6 and 22, loops waiting for interrupt. Also includes level_6 and level_22 interrupt handler functions which are branched to from the external interrupt exception routine.

Assembly Source: Includes external interrupt handler code which saves context, calls the appropriate interrupt service routine, and restores context.

6.2.1 Example 1: C Source

```
#include "mpc565.h"
#include "m_common.h"

          UINT32    loopcnt = 0;                  // Dummy loop counter
          int mc6_irq_ctr = 0;                    // MIOS submodule  6 interrupt counter
          int mc22_irq_ctr = 0;                   // MIOS submodule 22 interrupt counter


void init_565 (void)
{
          USIU.SYPCR.R = 0XFFFFFF03;              // Disable watchdog timer
          USIU.PLPRCR.B.MF = 0x00d;               // Run at 56 MHz
          while (USIU.PLPRCR.B.SPLS == 0);        // Wait for PLL to lock
          UIMB.UMCR.B.HSPEED = 0;                 // Run IMB at full clock speed
}

void init_mios (void)
{
                                                  // STEP 1:  MODULE SPECIFIC INIT
                                                    // MIOS: enable FREEZE, enable & set prescaler
          MIOS14.MIOS14MCR.B.FRZ = 1;            // MIOS- Activate MIOB freeze if IMB freeze occurs
          MIOS14.MCPSMSCR.B.FREN = 1;             // MIOS- Stop MIOS1 if IMB3 freeze is active
          MIOS14.MCPSMSCR.B.PREN = 1;             //MIOS- PRescaler ENable:  MCPSM counter enabled
          MIOS14.MCPSMSCR.B.PSL = 0;              //MIOS- Clock prescaler set to divide by 16

                                                  // CTR 6: Count to 64K with clock of 56 MHz/16/4
          MIOS14.MMCSM6CNT.R = 2;                 // CTR 6 - set CouNTer (and latch) to  near zero
          MIOS14.MMCSM6SCR.B.CLS = 3;             // CTR 6 - CLock Selected for prescaler
          MIOS14.MMCSM6SCR.B.CP = 0xfc;           // CTR 6 - Clock Prescaler:  divide by 4
          MIOS14.MMCSM6SCR.B.FREN = 1;            // CTR 6 - FReeze ENabled if MIOB freeze occurs

                                                  // CTR 22: Count to 64K with clock of 56MHz/16/8
```

```
        MIOS14.MMCSM22CNT.R  = 0;              // CTR 22- set CouNTer (and latch) to zero
        MIOS14.MMCSM22SCR.B.CLS = 3;           // CTR 22- CLock Selected for prescaler
        MIOS14.MMCSM22SCR.B.CP  = 0xf8;        // CTR 22- Clock Prescaler:  divide by 8
        MIOS14.MMCSM22SCR.B.FREN = 1;          // CTR 22- FReeze ENabled if MIOB freeze occurs

                                               // STEP 2:  LEVEL ASSIGNMENT
        MIOS14.MIOS14LVL1.B.LVL = 6;           // Set counter 22 interrupt level to 22
        MIOS14.MIOS14LVL1.B.TM  = 2;
        MIOS14.MIOS14LVL0.B.LVL = 6;           // Set counter  6 interrupt level to 6
        MIOS14.MIOS14LVL0.B.TM  = 0;

                                               // STEP 3:  ENABLE INTERRUPT
        MIOS14.MIOS14ER1.B.EN22 = 1;           // Enable counter 22 overflow to cause interrupt
        MIOS14.MIOS14ER0.B.EN6 = 1;            // Enable counter  6 overflow to cause interrupt

                                               // STEP 4:  SET APPROPRIATE SIMASK BITS
        USIU.SIMASK.B.LVM7 = 1;                // Enable level 7 interrupt (and levels 8-31)
        USIU.SIMASK.B.LVM6 = 1;                // Enable level 6 interrupt
}

void main ()
{
        init_565();                            // Perform a simple CPU init

        UIMB.UMCR.B.IRQMUX = 3;                // Enable up to 31 interrupt levels in UIMB

        init_mios();                           // Initialize 2 MIOS counters

        asm (" mtspr EIE, r0");                // Enable all interrupts

        while (1)
        {
                loopcnt++;                     // Up count while waiting for interrupts
        }
}

void level_6_isr (void)
{
//  Summary: service IMB interrupt on level 6 (MIOS CTR 6) -
//    1. Clear flag in interrupt status register (MIOS14SR0:1) by:
//                A. reading the register,
//                B. then write 0 to bit of the active interrupt flag and 1s to other bits
//    2. Increment appropriate software counter for that interrupt

        unsigned int irq_status = 0;           // Dummy variable to read status register
```

```
        irq_status = MIOS14.MIOS14SR0.R;        // Read MIOS interrupt status register
        MIOS14.MIOS14SR0.R = 0xffbf;            // Write 0 to active IRQ flag; 1s to other bits
        mc6_irq_ctr++;                          // Increment global variable for this ISR
}

void level_22_isr (void)
{
//  Summary: service IMB interrupt on level 22 (MIOS CTR 22) -
//    1. Clear flag in interrupt status register (MIOS14SR0:1) by:
//                A. reading the register,
//                B. then write 0 to bit of the active interrupt flag and 1's to other bits
//    2. Increment appropriate software counter for that interrupt

        unsigned int irq_status = 0;            // Dummy variable to read status register

        irq_status = MIOS14.MIOS14SR1.R;        // Read MIOS interrupt status register
        MIOS14.MIOS14SR1.R = 0xffbf;            // Write 0 to active IRQ flag; 1's to other bits
        mc22_irq_ctr++;                         // Increment global variable for this ISR
}
```

6.2.2 Example 1: Assembly Source

```
        .section .abs.00000100
              b _start                ; System reset exception, per crt0 file

        .section .abs.00000500
              b interrupt_exception_handler

        .text
interrupt_exception_handler:

        .equ SIVEC,         0x2fc01c            ;Register address
prolog:
                                    ; STEP 1:  SAVE "MACHINE CONTEXT"
        stwu sp,  -80 (sp); Create stack frame and store back chain
        stw       r3, 36 (sp)       ; Save working register
        mfsrr0    r3; Get SRR0
        stw       r3, 12 (sp)       ; and save SRR0
        mfsrr1    r3                ; Get SRR1
        stw       r3, 16 (sp)       ; and save SRR1


                                    ; STEP 2:  MAKE MSR[RI] RECOVERABLE
        mtspr     EID, r3           ; Set recoverable bit
```

```
                        ; Now debugger breakpoints can be set


                        ; STEP 3: SAVE OTHER APPROPRIATE CONTEXT
mflr    r3      ; Get LR
stw     r3, 8 (sp)          ; and save LR
mfxer   r3                  ; Get XER
stw     r3, 20 (sp)         ; and save XER
mfspr   r3, CTR             ; Get CTR
stw     r3, 24 (sp)         ; and save CTR
mfcr    r3      ; Get CR
stw     r3, 28 (sp)         ; and save CR
stw     r0, 32 (sp)         ; Save R0
stw     r4, 40 (sp)         ; Save R4 to R12
stw     r5, 44 (sp)
stw     r6, 48 (sp)
stw     r7, 52 (sp)
stw     r8, 56 (sp)
stw     r9, 60 (sp)
stw     r10, 64 (sp)
stw     r11, 68 (sp)
stw     r12, 72 (sp)

                        ; STEP 4: DETERMINE INTERRUPT SOURCE
lis     r3, SIVEC@ha        ; Load higher 16 bits of SIVEC address
lbz     r3, SIVEC@l (r3)    ; Load Interrupt Code byte from SIVEC
                        ; Interrupt Code will be jump table index

lis     r4, IRQ_table@h ; Load interrupt jump table base address
ori     r4, r4, IRQ_table@l
add     r4, r3, r4          ; Add index to table base address
mtlr    r4                  ; Load result address to link register


                        ; STEP 5: BRANCH TO INTERRUPT SERVICE ROUTINE
blrl            ; Jump to Execution Routine (subroutine)
                        ; (After returning here, restore context)


                        ; STEP 6: RESTORE CONTEXT
lwz     r0, 32 (sp)         ; Restore gprs except R3
lwz     r4, 40 (sp)
lwz     r5, 44 (sp)
lwz     r6, 48 (sp)
lwz     r7, 52 (sp)
lwz     r8, 56 (sp)
```

```
        lwz     r9, 60 (sp)
        lwz     r10, 64 (sp)
        lwz     r11, 68 (sp)
        lwz     r12, 72 (sp)
        lwz     r3, 20 (sp)                 ; Get XER
        mtxer   r3                          ; and restore XER
        lwz     r3, 24 (sp)                 ; Get CTR
        mtctr   r3                          ; and restore CTR
        lwz     r3, 28 (sp)                 ; Get CR
        mtcrf   0xff, r3                    ; and restore CR
        lwz     r3, 8 (sp)                  ; Get LR
        mtlr    r3                          ; and restore LR
        mtspr   NRI, r3                     ; Clear recoverable bit, MSR[RI]
                                                    ; Note: breakpoints CANNOT be set
                                                    ; from now thru the rfi instruction
        lwz     r3, 12 (sp)                 ; Get SRR0 from stack
        mtsrr0  r3                          ; and restore SRR0
        lwz     r3, 16 (sp)                 ; Get SRR1 from stack
        mtsrr1  r3                          ; and restore SRR1
        lwz     r3, 36 (sp)                 ; Restore R3
        addi    sp, sp, 80                  ; Clean up stack



                                            ; STEP 7:  Return to Program
        rfi                                 ; End of Interrupt

; ================================================================
; Branch table for the different SIVEC Interrupt Code values:

IRQ_table:                                  ; Branch forever if isr is not written

irq_0_trap_error:   b       irq_0_trap error    ; irq_0 should generate NMI [1]
;irq_0_trap_error:  rfi                         ; For production code
irq_1:              b       irq_1
level_1:            b       level_1
irq_2:              b       irq_2
level_2:            b       level_2
irq_3:              b       irq_3
level_3:            b       level_3
irq_4:              b       irq_4
level_4:            b       level_4
irq_5:              b       irq_5
```

[1]The irq_0_trap_error interrupt generates NMI instead of this vector. However, improper interrupt programming might cause this vector. For debugging, the use of a trap handler that saves SRR0 and SRR1 is recommended. Saving these registers allows backtracking to find the offending code. For production code, it is recommended that an rfi instruction is used.

```
level_5:            b         level_5
irq_6:              b         irq_6
                    b         level_6_isr         ; Branch to C isr
irq_7:              b         irq_7

check_level_7:                          ; Determine highest pending level up to level 31

          .equ UIPEND,        0x307fa0            ;Register address

          lis       r3, UIPEND@ha; Read UIPEND to r3
          lwz       r3, UIPEND@l (r3)

          rlwinm    r3, r3, 0, 7, 31; Clear bits read from UIPEND register
                                ; corresponding to levels 0-6.  If one of these
                                ; levels 0-6 was set, it occurred because either:
                                ; 1) SIMASK is blocking a pending IRQ
                                ; -or- 2) the IRQ occurred just after checking SIVEC.
          cntlzw    r3, r3          ; Count number of non-pending interrupt levels
          rotlwi    r3, r3, 0x2     ; Rotate number left 2 (multiply x 4) to prepare for jump table

          lis       r4, level_table@h ; Load interrupt jump table base address
          ori       r4, r4, level_table@l
          add       r4, r3, r4            ; Add index to table base address
          mtctr     r4                    ; Load result address to COUNT reg. (must preserve LR)
          bctr

level_table:                  ; Branch table for levels 0-31.
                                ; Since levels 0-6 were masked just above, there will not
                                ; branches to them here, hence called "illegal".
illegal_0:          b         illegal_0
illegal_1:          b         illegal_1
illegal_2:          b         illegal_2
illegal_3:          b         illegal_3
illegal_4:          b         illegal_4
illegal_5:          b         illegal_5
illegal_6:          b         illegal_6
level_7:            b         level_7
level_8:            b         level_8
level_9:            b         level_9
level_10:           b         level_10
level_11:           b         level_11
level_12:           b         level_12
level_13:           b         level_13
level_14:           b         level_14
level_15:           b         level_15
level_16:           b         level_16
```

```
level_17:           b         level_17
level_18:           b         level_18
level_19:           b         level_19
level_20:           b         level_20
level_21:           b         level_21
                    b         level_22_isr ; Branch to C isr
level_23:           b         level_23
level_24:           b         level_24
level_25:           b         level_25
level_26:           b         level_26
level_27:           b         level_27
level_28:           b         level_28
level_29:           b         level_29
level_30:           b         level_30
level_31:           b         level_31
```

6.3 Example 2: EIC

Example 2 is the same as example 1 except that it uses the EIC.

C Source: Same as in example 1 except for:

- enabling the EIC.

Assembly Source: Same as in example 1 except for:

- the expanded and different branch table to accommodate new Interrupt Code values.

6.3.1 Example 2: C Source

```
#include "mpc565.h"
#include "m_common.h"

          UINT32    loopcnt = 0;               // Dummy loop counter
          int mc6_irq_ctr = 0;                 // MIOS submodule  6 interrupt counter
          int mc22_irq_ctr = 0;                // MIOS submodule 22 interrupt counter


void init_565 (void)
{
          USIU.SYPCR.R = 0XFFFFFF03;           // Disable watchdog timer
          USIU.PLPRCR.B.MF = 0x00d;            // Run at 56 MHz
          while (USIU.PLPRCR.B.SPLS == 0);     // Wait for PLL to lock
          UIMB.UMCR.B.HSPEED = 0;              // Run IMB at full clock speed
}

void init_mios (void)
{
```

```
                                        // STEP 1:  MODULE SPECIFIC INIT
                                        // MIOS: enable FREEZE, enable & set prescaler
        MIOS14.MIOS14MCR.B.FRZ = 1;     // MIOS- Activate MIOB freeze if IMB freeze occurs
        MIOS14.MCPSMSCR.B.FREN = 1;     // MIOS- Stop MIOS1 if IMB3 freeze is active
        MIOS14.MCPSMSCR.B.PREN = 1;     // MIOS- PRescaler ENable:  MCPSM counter enabled
        MIOS14.MCPSMSCR.B.PSL = 0;      // MIOS- Clock prescaler set to divide by 16

                                        // CTR 6: Count to 64K with clock of 56 MHz/16/4//
CTR 6: set timeout = 40 MHz/16/4
        MIOS14.MMCSM6CNT.R = 2;         // CTR 6 - set CouNTer (and latch) to near zero
        MIOS14.MMCSM6SCR.B.CLS = 3;     // CTR 6 - CLock Selected for prescaler
        MIOS14.MMCSM6SCR.B.CP = 0xfc;   // CTR 6 - Clock Prescaler:  divide by 4
        MIOS14.MMCSM6SCR.B.FREN = 1;    // CTR 6 - FReeze ENabled if MIOB freeze occurs

                                        // CTR 22: Count to 64K with clock of 56 MHz/16/8
        MIOS14.MMCSM22CNT.R = 0;        // CTR 6 - set CouNTer (and latch) to zero
        MIOS14.MMCSM22SCR.B.CLS = 3;    // CTR 22- CLock Selected for prescaler
        MIOS14.MMCSM22SCR.B.CP  = 0xf8; // CTR 22- Clock Prescaler:  divide by 8
        MIOS14.MMCSM22SCR.B.FREN = 1;   // CTR 22- FReeze ENabled if MIOB freeze occurs


                                        // STEP 2:  LEVEL ASSIGNMENT
        MIOS14.MIOS14LVL1.B.LVL = 6;    // Set counter 22 interrupt level to 22
        MIOS14.MIOS14LVL1.B.TM = 2;
        MIOS14.MIOS14LVL0.B.LVL = 6;    // Set counter  6 interrupt level to 6
        MIOS14.MIOS14LVL0.B.TM  = 0;

                                        // STEP 3:  ENABLE INTERRUPT
        MIOS14.MIOS14ER1.B.EN22 = 1;    // Enable counter 22 overflow to cause interrupt
        MIOS14.MIOS14ER0.B.EN6 = 1;     // Enable counter  6 overflow to cause interrupt

                                        // STEP 4:  SET APPROPRIATE SIMASK BITS
        USIU.SIMASK3.B.IMBIRQ22 = 1;    // Enable IMB level 22 interrupt
        USIU.SIMASK2.B.IMBIRQ6 = 1;     // Enable IMB level 6 interrupt
}


void main ()
{
        init_565();                     // Perform a simple CPU init

        UIMB.UMCR.B.IRQMUX = 3;         // Enable up to 31 interrupt levels in UIMB
        USIU.SIUMCR.B.EICEN = 1;        // Enable EIC

        init_mios();                    // Initialize 2 MIOS counters

        asm (" mtspr EIE, r0");         // Enable all interrupts
```

```
          while (1)
          {
                    loopcnt++;                  // Up count while waiting for interrupts
          }
}

void imb_irq_6_isr (void)

{
// Summary: service IMB interrupt on level 6 (MIOS CTR 6)
//    1. Clear flag in interrupt status register (MIOS14SR0:1) by:
//                A. reading the register,
//                B. then write 0 to bit of the active interrupt flag and 1's to other bits
//    2. Increment appropriate software counter for that interrupt

          unsigned int irq_status = 0;          // Dummy variable to read status register

          irq_status = MIOS14.MIOS14SR0.R;      // Read MIOS interrupt status register
          MIOS14.MIOS14SR0.R = 0xffbf;          // Write 0 to active IRQ flag; 1's to other bits
          mc6_irq_ctr++;                        // Increment global variable for this ISR}
}

void imb_irq_22_isr (void)
{
//  Summary: service IMB interrupt on level 22 (MIOS CTR 22)
//    1. Clear flag in interrupt status register (MIOS14SR0:1) by:
//                A. reading the register,
//                B. then write 0 to bit of the active interrupt flag and 1's to other bits
//    2. Increment appropriate software counter for that interrupt

          unsigned int irq_status = 0;          // Dummy variable to read status register

          irq_status = MIOS14.MIOS14SR1.R;      // Read MIOS interrupt status register
          MIOS14.MIOS14SR1.R = 0xffbf;          // Write 0 to active IRQ flag; 1's to other bits
          mc22_irq_ctr++;                       // Increment global variable for this ISR
}
```

6.3.2 Example 2: Assembly Source

```
          .section .abs.00000100
                    b _start            ; System reset exception, per crt0 file

          .section .abs.00000500
                    b interrupt_exception_handler
```

```
          .text
interrupt_exception_handler:

          .equ      SIVEC,              0x2fc01c  ;Register addresses

                                                  ; STEP 1:  SAVE "MACHINE CONTEXT"
          stwu sp, -80 (sp); Create stack frame and store back chain
          stw       r3, 36 (sp)                   ; Save working register
          mfsrr0    r3                            ; Get SRR0
          stw       r3, 12 (sp)                   ; and save SRR0
          mfsrr1    r3                            ; Get SRR1
          stw       r3, 16 (sp)                   ; and save SRR1


                                                  ; STEP 2:  MAKE MSR[RI] RECOVERABLE
          mtspr     EID, r3                       ; Set recoverable bit
                                                  ; Now debugger breakpoints can be set



                                                  ; STEP 3: SAVE OTHER APPROPRIATE CONTEXT
          mflr      r3                            ; Get LR
          stw       r3, 8 (sp)                    ; and save LR
          mfxer     r3                            ; Get XER
          stw       r3, 20 (sp)                   ; and save XER
          mfspr     r3, CTR                       ; Get CTR
          stw       r3, 24 (sp)                   ; and save CTR
          mfcr      r3                            ; Get CR
          stw       r3, 28 (sp)                   ; and save CR
          stw       r0, 32 (sp)                   ; Save R0
          stw       r4, 40 (sp)                   ; Save R4 to R12
          stw       r5, 44 (sp)
          stw       r6, 48 (sp)
          stw       r7, 52 (sp)
          stw       r8, 56 (sp)
          stw       r9, 60 (sp)
          stw       r10, 64 (sp)
          stw       r11, 68 (sp)
          stw       r12, 72 (sp)

                                                  ; STEP 4: DETERMINE INTERRUPT SOURCE
          lis       r3, SIVEC@ha                  ; Load higher 16 bits of SIVEC address
          lbz       r3, SIVEC@l (r3)              ; Load Interrupt Code byte from SIVEC
                                                  ; Interrupt Code will be jump table index

          lis       r4, IRQ_table@h               ; Load interrupt jump table base address
          ori       r4, r4, IRQ_table@l
```

```
        add     r4, r3, r4              ; Add index to table base address
        mtlr    r4                      ; Load result address to link register

                                        ; STEP 5: BRANCH TO INTERRUPT SERVICE ROUTINE
        blrl                            ; Jump to Execution Routine (subroutine)
                                        ; (After returning here, restore context)


                                        ; STEP 6: RESTORE CONTEXT
        lwz     r0, 32 (sp)             ; Restore gprs except R3
        lwz     r4, 40 (sp)
        lwz     r5, 44 (sp)
        lwz     r6, 48 (sp)
        lwz     r7, 52 (sp)
        lwz     r8, 56 (sp)
        lwz     r9, 60 (sp)
        lwz     r10, 64 (sp)
        lwz     r11, 68 (sp)
        lwz     r12, 72 (sp)
        lwz     r3, 20 (sp)             ; Get XER
        mtxer   r3                      ; and restore XER
        lwz     r3, 24 (sp)             ; Get CTR
        mtctr   r3                      ; and restore CTR
        lwz     r3, 28 (sp)             ; Get CR
        mtcrf   0xff, r3                ; and restore CR
        lwz     r3, 8 (sp)              ; Get LR
        mtlr    r3                      ; and restore LR
        mtspr   NRI, r3                 ; Clear recoverable bit, MSR[RI]
                                        ; Note: breakpoints CANNOT be set
                                        ; from now thru the rfi instruction
        lwz     r3, 12 (sp)             ; Get SRR0 from stack
        mtsrr0  r3                      ; and restore SRR0
        lwz     r3, 16 (sp)             ; Get SRR1 from stack
        mtsrr1  r3                      ; and restore SRR1
        lwz     r3, 36 (sp)             ; Restore R3
        addi    sp, sp, 80              ; Clean up stack


                                        ; STEP 7:  Return to Program
        rfi                             ; End of Interrupt

; ================================================================
; EIC Branch table for SIVEC[InterruptCode]:

IRQ_table:                              ; Branch forever if routine is not written
```

```
IRQ_0_TRAP_ERROR:   b         IRQ_0_TRAP_ERROR     ; IRQ_0 should generate NMI [1]
;IRQ_0_TRAP_ERROR:  rfi                            ; For production code
LEVEL_0:            b         LEVEL_0
IMB_IRQ_0:          b         IMB_IRQ_0
IMB_IRQ_1:          b         IMB_IRQ_1
IMB_IRQ_2:          b         IMB_IRQ_2
IMB_IRQ_3:          b         IMB_IRQ_3
EXT_IRQ_1:          b         EXT_IRQ_1
LEVEL_1:            b         LEVEL_1
IMB_IRQ_4:          b         IMB_IRQ_4
IMB_IRQ_5:          b         IMB_IRQ_5
                    b         imb_irq_6_isr        ; Branch to C isr
IMB_IRQ_7:          b         IMB_IRQ_7
EXT_IRQ_2:          b         EXT_IRQ_2
LEVEL_2:            b         LEVEL_2
IMB_IRQ_8:          b         IMB_IRQ_8
IMB_IRQ_9:          b         IMB_IRQ_9
IMB_IRQ_10:         b         IMB_IRQ_10
IMB_IRQ_11:         b         IMB_IRQ_11
EXT_IRQ_3:          b         EXT_IRQ_3
LEVEL_3:            b         LEVEL_3
IMB_IRQ_12:         b         IMB_IRQ_12
IMB_IRQ_13:         b         IMB_IRQ_13
IMB_IRQ_14:         b         IMB_IRQ_14
IMB_IRQ_15:         b         IMB_IRQ_15
EXT_IRQ_4:          b         EXT_IRQ_4
LEVEL_4:            b         LEVEL_4
IMB_IRQ_16:         b         IMB_IRQ_16
IMB_IRQ_17:         b         IMB_IRQ_17
IMB_IRQ_18:         b         IMB_IRQ_18
IMB_IRQ_19:         b         IMB_IRQ_19
EXT_IRQ_5:          b         EXT_IRQ_5
LEVEL_5:            b         LEVEL_5
IMB_IRQ_20:         b         IMB_IRQ_20
IMB_IRQ_21:         b         IMB_IRQ_21
                    b         imb_irq_22_isr       ; Branch to C isr
IMB_IRQ_23:         b         IMB_IRQ_23
EXT_IRQ_6:          b         EXT_IRQ_6
LEVEL_6:            b         LEVEL_6
IMB_IRQ_24:         b         IMB_IRQ_24
IMB_IRQ_25:         b         IMB_IRQ_25
IMB_IRQ_26:         b         IMB_IRQ_26
```

[1]The irq_0_trap_error interrupt generates NMI instead of this vector. However, improper interrupt programming might cause this vector. For debugging, the use of a trap handler that saves SRR0 and SRR1 is recommended. Saving these registers allows backtracking to find the offending code. For production code, it is recommended that an rfi instruction is used.

```
IMB_IRQ_27:         b         IMB_IRQ_27
EXT_IRQ_7:          b         EXT_IRQ_7
LEVEL_7:            b         LEVEL_7
IMB_IRQ_28:         b         IMB_IRQ_28
IMB_IRQ_29:         b         IMB_IRQ_29
IMB_IRQ_30:         b         IMB_IRQ_30
IMB_IRQ_31:         b         IMB_IRQ_31
```

6.4 Example 3: EIC with External Interrupt Relocation

This example adds the feature that enables interrupts to automatically “relocate” their exception vector to level-number- or IRQ-pin-based individual addresses instead of sharing one address, the external interrupt exception address.

One of the interrupt service routines is done entirely in assembler, rather than calling a C ISR, to demonstrate the reduced context save-and-restore in its handler.

C Source: Same as example 2 except for:

- enabling external interrupt relocation
- setting a base table address
- enabling exception table relocation.
- removing C ISR for IMB Level 22 interrupt

Assembly Source: Modified from example 2 to:

- accommodate unique starting addresses for each interrupt level and pin
- include complete ISR for IMB Level 22 interrupt

Context saving and restoring must now be accomplished by either duplicating code for each interrupt or, as done in this example, having interrupt handlers which call C ISRs call common context save and restore routines.

When the ISR is implanted in only assembler, as in IMB level 22, the context save and restore is minimal and likely best done in line. The assembler handler implemented in this example does not save SRR0 and SRR1 nor does it set MSR[RI]. Hence, breakpoints cannot be set anywhere inside the interrupt handler.

For illustrative purposes, an alternate handler for IMB level 22 is shown below with the following differences:

- Saves SRR0 and SRR1 and sets MSR[RI] to allow breakpoints in the handler (but takes extra time)
- Saves CR on stack, allowing compare instructions in handler/ISR
- Branches to ISR instead of in line coding
- ISR uses “andi.” instruction which modifies CR (which is why CR is put on stack here).

```
imb_irq_22_handler:                 ; This interrupt routine saves context only required for a
                                    ;simple assembly routine.
                                ;Stackframe:
                                    ;     Offset from SP:             Register Saved:
                                    ;         0x14                        CR
                                    ;         0x10                        r4
                                    ;         0xc                         r3
```

```
                                    ;         0x8                         SRR1
                                    ;         0x4                         SRR0
                                    ;         0x0                         backchain (old SP)

                                    ; STEP 1:  SAVE "MACHINE CONTEXT"
          stwu sp, -18 (sp)         ; Create stack frame and store back chain
          stw       r3, 0xc (sp)    ; Save working register
          mfsrr0    r3              ; Get SRR0
          stw       r3, 4 (sp)      ; and save SRR0
          mfsrr1    r3              ; Get SRR1
          stw       r3, 8 (sp)      ; and save SRR1

                                    ; STEP 2:  MAKE MSR[RI] RECOVERABLE
          mtspr     EID, r3         ; Set recoverable bit
                                    ; Now debugger breakpoints can be set

                                    ; STEP 3: SAVE OTHER APPROPRIATE CONTEXT
          stw       r4, 0x10 (sp)
          mfcr      r3
          stw       r3, 0x14 (sp)

                                    ; STEP 4: DETERMINE SOURCE OF INTERRUPT
                                    ; STEP 5: BRANCH TO INTERRUPT SERVICE ROUTINE
          b         imb_irq_22_isr


imb_irq_22_isr:                     ; ISR in assembler put here.
                                    ; Branch instructions were used instead of the quicker
                                    ; inline implementation for trace measurement purposes.
          lis       r3, MIOS14SR1@ha    ; Read Mios interrupt status register to r4
          lhz       r4, MIOS14SR1@l(r3)
          andi.     r4, r4, 0xffbf      ; Write "0" to submodule 22 flag to clear it
          sth       r4, MIOS14SR1@l(r3)

          lis       r4, mc22_irq_ctr@ha ; Increment ISR counter for modulus counter of submodule 22
          lwz       r3, mc22_irq_ctr@l (r4)
          addi      r3, r3, 1
          stw       r3, mc22_irq_ctr@l (r4)
          b         restore_imb_irq_22; Branch to a restore routine which compliments the context save
```

6.4.1 Example 3: C Source

```
#include "mpc565.h"
#include "m_common.h"

          UINT32 loopcnt = 0;                   // Dummy loop counter
```

```
          int mc6_irq_ctr = 0;                    // MIOS submodule  6 interrupt counter
          int mc22_irq_ctr = 0;                   // MIOS submodule 22 interrupt counter


void init_565 (void)
{
          USIU.SYPCR.R = 0XFFFFFF03;              // Disable watchdog timer
          USIU.PLPRCR.B.MF = 0x00d;               // Run at 56 MHz
          while (USIU.PLPRCR.B.SPLS == 0);        // Wait for PLL to lock
    UIMB.UMCR.B.HSPEED = 0;                       // Run IMB at full clock speed
}

asm void init_ext_int_rel (void)
{
! "r0"
                                                  ; Tell compiler any scratch registers used
; ASSUMPTIONS-- Reset configuration word (or debugger) set the following bits:
;        bit     19   ETRE = 1     Exception Table Relocation Enable is set
;        bits 24:25 OERC = 00    Exception table base = 0 for MPC56x and MPC53x
;        bits 28:30 ISB  = 000   Internal memory Space Base = 0x0

      lis      r0, EIR_table@h            ; Load EIBADR (SPR 529) with table base address
      ori      r0, r0, EIR_table@l
      mtspr    529, r0

      mfspr    r0, 560              ; Set BBCMCR[EIR]=1, spr560 bit 20, to enable int. relocation
      ori      r0, r0, 0x3801             ; also set ETRE = 1, BE = 1 and DCAE = 1
      mtspr    560, r0
}

void init_mios (void)
{
                                          // STEP 1:  MODULE SPECIFIC INIT
                                          // MIOS: enable FREEZE, enable & set prescaler
          MIOS14.MIOS14MCR.B.FRZ = 1;     // MIOS- Activate MIOB freeze if IMB freeze occurs
          MIOS14.MCPSMSCR.B.FREN = 1;     // MIOS- Stop MIOS1 if IMB3 freeze is active
          MIOS14.MCPSMSCR.B.PREN = 1;     // MIOS- PRescaler ENable:  MCPSM counter enabled
          MIOS14.MCPSMSCR.B.PSL = 0;      // MIOS- Clock prescaler set to divide by 16

                                          // CTR 6: Count to 64K with clock of 56 MHz/16/4
          MIOS14.MMCSM6CNT.R = 2;         // CTR 6 - set CouNTer (and latch) to near zero
          MIOS14.MMCSM6SCR.B.CLS = 3;     // CTR 6 - CLock Selected for prescaler
          MIOS14.MMCSM6SCR.B.CP = 0xfc;   // CTR 6 - Clock Prescaler:  divide by 4
          MIOS14.MMCSM6SCR.B.FREN = 1;    // CTR 6 - FReeze ENabled if MIOB freeze occurs

                                         // CTR 22: Count to 64K with clock of 56 MHz/16/8
```

```
        MIOS14.MMCSM22CNT.R  = 0;          // CTR 22- set CouNTer (and latch) to zero
        MIOS14.MMCSM22SCR.B.CLS = 3;       // CTR 22- CLock Selected for prescaler
        MIOS14.MMCSM22SCR.B.CP  = 0xf8;    // CTR 22- Clock Prescaler:  divide by 8
        MIOS14.MMCSM22SCR.B.FREN = 1;      // CTR 22- FReeze ENabled if MIOB freeze occurs

                                           // STEP 2:  LEVEL ASSIGNMENT
        MIOS14.MIOS14LVL1.B.LVL = 6;       // Set counter 22 interrupt level to 22
        MIOS14.MIOS14LVL1.B.TM = 2;
        MIOS14.MIOS14LVL0.B.LVL = 6;       // Set counter  6 interrupt level to 6
        MIOS14.MIOS14LVL0.B.TM  = 0;

                                           // STEP 3:  ENABLE INTERRUPT
        MIOS14.MIOS14ER1.B.EN22 = 1;       // Enable counter 22 overflow to cause interrupt
        MIOS14.MIOS14ER0.B.EN6 = 1;        // Enable counter  6 overflow to cause interrupt

                                           // STEP 4:  SET APPROPRIATE SIMASK BITS
        USIU.SIMASK3.B.IMBIRQ22 = 1;       // Enable IMB level 22 interrupt
        USIU.SIMASK2.B.IMBIRQ6 = 1;        // Enable IMB level 6 interrupt
}

void main ()
{
        init_565();                 // Perform a simple CPU init

        UIMB.UMCR.B.IRQMUX = 3;     // Enable up to 31 interrupt levels in UIMB
        USIU.SIUMCR.B.EICEN = 1;    // Enable EIC
        init_ext_int_rel();         // Init. Ext. Interrupt & Exception Table Relocation

        init_mios();                // Initialize 2 MIOS counters

        asm (" mtspr EIE, r0");     // Enable all interrupts

        while (1)
        {
                loopcnt++;          // Up count while waiting for interrupts
        }
}

void imb_irq_6_isr (void)
{
//  Summary: service IMB interrupt on level 6 (MIOS CTR 6) -
//    1. Clear flag in interrupt status register (MIOS14SR0:1) by:
//                A. reading the register,
//                B. then write 0 to bit of the active interrupt flag and 1's to other bits
//    2. Increment appropriate software counter for that interrupt
```

```
        unsigned int irq_status = 0;  // Dummy variable to read status register

        irq_status = MIOS14.MIOS14SR0.R;        // Read MIOS interrupt status register
        MIOS14.MIOS14SR0.R = 0xffbf;            // Write 0 to active IRQ flag; 1's to other bits
        mc6_irq_ctr++;                          // Increment global variable for this ISR
}
```

6.4.2 Example 3: Assembly Source

```
        .globl              EIR_table
        .equ      MIOS14SR1, 0x306c40

        .section .abs.00000008
                  b _start            ; System reset exception, per crt0 file for ETRE=1

        .text

imb_irq_6_handler:                    ; This interrupt handler saves context
                                      ; necessary to call a c function

                                      ; STEP 1:  SAVE "MACHINE CONTEXT"
        stwu sp, -80 (sp); Create stack frame and store back chain
        stw       r3, 36 (sp)         ; Save working register
        mfsrr0    r3                  ; Get SRR0
        stw       r3, 12 (sp)         ; and save SRR0
        mfsrr1    r3                  ; Get SRR1
        stw       r3, 16 (sp)         ; and save SRR1


                                      ; STEP 2:  MAKE MSR[RI] RECOVERABLE
        mtspr               EID, r3   ; Set recoverable bit
                                      ; Now debugger breakpoints can be set


                                      ; STEP 3: SAVE OTHER APPROPRIATE CONTEXT
        mflr      r3                  ; Get LR
        stw       r3, 8 (sp)          ; and save LR
        bl        finish_saving_context

                                      ; STEP 4: DETERMINE SOURCE OF INTERRUPT
                                      ; STEP 5: BRANCH TO INTERRUPT SERVICE ROUTINE
        bl                  imb_irq_6_isr

                                      ; Return from C routine here
                                      ; STEP 6: RESTORE CONTEXT
        b                   restore_context
```

```
;***********************************************
imb_irq_22_handler:          ; This interrupt routine saves context
                                    ; only required for a minimal assembly routine.
                                    ; CAUTION:  do not set breakpoints in this handler
                                    ;           because SRR0:1 are not saved nor MSR[RI] set.
                                    ; Stack frame is not used.  Stack organization is:
                                    ;         Offset from SP              Register Saved
                                    ;                  -0x8                       r4
                                    ;                  -0x4                       r3

                                    ; STEP 1:  SAVE "MACHINE CONTEXT"
                                    ; STEP 2:  MAKE MSR[RI] RECOVERABLE

                                    ; STEP 3: SAVE OTHER APPROPRIATE CONTEXT
          stw       r3, -0x4 (sp)   ; Save working registers r3 and r4
          stw       r4, -0x8 (sp)

                                    ; STEP 4: DETERMINE SOURCE OF INTERRUPT
                                    ; STEP 5: BRANCH TO INTERRUPT SERVICE ROUTINE
imb_irq_22_isr:
          lis       r3, MIOS14SR1@ha    ; Read Mios interrrupt status register to r4
          lhz       r4, MIOS14SR1@l(r3)
          rlwinm    r4, r4, 0, 26, 24   ; Clear bit 25 of r4, submodule 22's flag
          sth       r4, MIOS14SR1@l(r3)

          lis       r4, mc22_irq_ctr@ha; Increment ISR counter for modulus counter of submodule 22
          lwz       r3, mc22_irq_ctr@l (r4)
          addi      r3, r3, 1
          stw       r3, mc22_irq_ctr@l (r4)

                                    ; STEP 6: RESTORE CONTEXT
restore_imb_irq_22:
          lwz       r4, -0x8 (sp)           ; Restore r4
          lwz       r3, -0x4 (sp)           ; Restore r3

                                    ; STEP 7:  Return to Program
          rfi                       ; End of Interrupt

;***********************************************

finish_saving_context:
                                    ; Save other registers to allow calling a
                                    ; c function

          mfxer     r3              ; Get XER
```

```
        stw     r3, 20 (sp)         ; and save XER
        mfspr   r3, CTR             ; Get CTR
        stw     r3, 24 (sp)         ; and save CTR
        mfcr    r3                  ; Get CR
        stw     r3, 28 (sp)         ; and save CR
        stw     r0, 32 (sp)         ; Save R0
        stw     r4, 40 (sp)         ; Save R4 to R12
        stw     r5, 44 (sp)
        stw     r6, 48 (sp)
        stw     r7, 52 (sp)
        stw     r8, 56 (sp)
        stw     r9, 60 (sp)
        stw     r10, 64 (sp)
        stw     r11, 68 (sp)
        stw     r12, 72 (sp)
        blr                         ; Return to interrupt routine


restore_context:
        lwz     r0, 32 (sp)         ; Restore gprs except R3
        lwz      r4, 40 (sp)
        lwz      r5, 44 (sp)
        lwz     r6, 48 (sp)
        lwz     r7, 52 (sp)
        lwz     r8, 56 (sp)
        lwz     r9, 60 (sp)
        lwz     r10, 64 (sp)
        lwz     r11, 68 (sp)
        lwz     r12, 72 (sp)
        lwz     r3, 20 (sp)         ; Get XER
        mtxer   r3                  ; and restore XER
        lwz     r3, 24 (sp)         ; Get CTR
        mtctr   r3                  ; and restore CTR
        lwz     r3, 28 (sp)         ; Get CR
        mtcrf   0xff, r3            ; and restore CR
        lwz     r3, 8 (sp)           Get LR
        mtlr    r3                  ; and restore LR
        mtspr   NRI, r3                     ; Clear recoverable bit, MSR[RI]
                                            ; Note: breakpoints CANNOT be set
                                            ; from now thru the rfi instruction
        lwz     r3, 12 (sp)                 ; Get SRR0 from stack
        mtsrr0  r3                          ; and restore SRR0
        lwz     r3, 16 (sp)                 ; Get SRR1 from stack
        mtsrr1  r3                          ; and restore SRR1
        lwz     r3, 36 (sp)                 ; Restore R3
        addi    sp, sp, 80                  ; Clean up stack
```

```
                                        ; STEP 7:  Return to Program
        rfi                             ; End of Interrupt



; ===================================================================
; External Interrupt Relocation branch table
; NOTE:  the .space directive may not be suitable for code compression
; Table reserves 96 words starting on a boundary a multiple of 0x2000

        .section .abs.00002000

EIR_table:                  ; Branch forever if routine is not written

EXT_IRQ_0_TRAP_ERROR:  ba       EXT_IRQ_0_TRAP_ERROR     ; EXT_IRQ_0 should generate NMI [1]
;EXT_IRQ_0_TRAP_ERROR: rfi                                ; For production code
                       .space 4
LEVEL_0:               ba       LEVEL_0
                       .space 4
IMB_IRQ_0:             ba       IMB_IRQ_0
                       .space 4
IMB_IRQ_1:             ba       IMB_IRQ_1
                       .space 4
IMB_IRQ_2:             ba       IMB_IRQ_2
                       .space 4
IMB_IRQ_3:             ba       IMB_IRQ_3
                       .space 4
EXT_IRQ_1:             ba       EXT_IRQ_1
                       .space 4
LEVEL_1:               ba       LEVEL_1
                       .space 4
IMB_IRQ_4:             ba       IMB_IRQ_4
                       .space 4
IMB_IRQ_5:             ba       IMB_IRQ_5
                       .space 4
IMB_IRQ_6:             ba       imb_irq_6_handler        ; Branch to handler
                       .space 4
IMB_IRQ_7:             ba       IMB_IRQ_7
                       .space 4
```

[1]The EXT_IRQ_0 interrupt generates NMI instead of this vector. However, improper interrupt programming might cause this vector. For debugging, the use of a trap handler that saves SRR0 and SRR1 is recommended. Saving these registers allows backtracking to find the offending code. For production code, it is recommended that an rfi instruction is used.

```
EXT_IRQ_2:          ba      EXT_IRQ_2
                    .space 4
LEVEL_2:            ba      LEVEL_2
                    .space 4
IMB_IRQ_8:          ba      IMB_IRQ_8
                    .space 4
IMB_IRQ_9:          ba      IMB_IRQ_9
                    .space 4
IMB_IRQ_10:         ba      IMB_IRQ_10
                    .space 4
IMB_IRQ_11:         ba      IMB_IRQ_11
                    .space 4
EXT_IRQ_3:          ba      EXT_IRQ_3
                    .space 4
LEVEL_3:            ba      LEVEL_3
                    .space 4
IMB_IRQ_12:         ba      IMB_IRQ_12
                    .space 4
IMB_IRQ_13:         ba      IMB_IRQ_13
                    .space 4
IMB_IRQ_14:         ba      IMB_IRQ_14
                    .space 4
IMB_IRQ_15:         ba      IMB_IRQ_15
                    .space 4
EXT_IRQ_4:          ba      EXT_IRQ_4
                    .space 4
LEVEL_4:            ba      LEVEL_4
                    .space 4
IMB_IRQ_16:         ba      IMB_IRQ_16
                    .space 4
IMB_IRQ_17:         ba      IMB_IRQ_17
                    .space 4
IMB_IRQ_18:         ba      IMB_IRQ_18
                    .space 4
IMB_IRQ_19:         ba      IMB_IRQ_19
                    .space 4
EXT_IRQ_5:          ba      EXT_IRQ_5
                    .space 4
LEVEL_5:            ba      LEVEL_5
                    .space 4
IMB_IRQ_20:         ba      IMB_IRQ_20
                    .space 4
IMB_IRQ_21:         ba      IMB_IRQ_21
                    .space 4
IMB_IRQ_22:         ba      imb_irq_22_handler; Branch to handler
                    .space 4
```

```
IMB_IRQ_23:          ba       IMB_IRQ_23
                     .space 4
EXT_IRQ_6:           ba       EXT_IRQ_6
                     .space 4
LEVEL_6:             ba       LEVEL_6
                     .space 4
IMB_IRQ_24:          ba       IMB_IRQ_24
                     .space 4
IMB_IRQ_25:          ba       IMB_IRQ_25
                     .space 4
IMB_IRQ_26:          ba       IMB_IRQ_26
                     .space 4
IMB_IRQ_27:          ba       IMB_IRQ_27
                     .space 4
EXT_IRQ_7:           ba       EXT_IRQ_7
                     .space 4
LEVEL_7:             ba       LEVEL_7
                     .space 4
IMB_IRQ_28:          ba       IMB_IRQ_28
                     .space 4
IMB_IRQ_29:          ba       IMB_IRQ_29
                     .space 4
IMB_IRQ_30:          ba       IMB_IRQ_30
                     .space 4
IMB_IRQ_31:          ba       IMB_IRQ_31
```

6.5 Example 4: EIC with External Interrupt Relocation and Lower Priority Request Masking

To the features in example 3, this example adds nested interrupt capability using the lower priority request masking (LPRM) feature in the EIC.

C Source: Same as example 3 except for:

- enabling LPRM
- initializing two more interrupts: IRQ1 and IRQ3 pins and a MIOS GPIO pin to drive them
- clearing, via ISRs, the appropriate SISR bit for a particular interrupt
- added ISRs for IRQ1 and IRQ3 pins

NOTE

Appendix A contains additional information on IRQ pin behavior.

Assembly Source: Same as example 3 except interrupts are now re-enabled after saving SRR0 and SRR1.

CAUTION

Nested interrupts always require more care from the system designer. For example, after an interrupt request is negated in the ISR, enough time should be allowed for that negation to reach the CPU core before interrupts of the same priority are enabled again. For IMB peripherals, this propagation time could take up to 5 or 6 clocks. In this example, the C instructions provide adequate time before same and lower priority interrupts are enabled in the SISRs.

6.5.1 Example 4: C Source

```
#include "mpc565.h"
#include "m_common.h"

          UINT32 loopcnt = 0;                // Dummy loop counter
          int mc6_irq_ctr = 0;               // MIOS submodule  6 interrupt counter
          int mc22_irq_ctr = 0;              // MIOS submodule 22 interrupt counter
          int irq_1_ctr = 0;                 // IRQ1 input pin interrupt counter
          int irq_3_ctr = 0;                 // IRQ3 input pin interrupt counter

void init_565 (void)
{
          USIU.SYPCR.R = 0XFFFFFF03;         // Disable watchdog timer
          USIU.PLPRCR.B.MF = 0x00d;          // Run at 56 MHz
          while (USIU.PLPRCR.B.SPLS == 0);   // Wait for PLL to lock
          UIMB.UMCR.B.HSPEED = 0;            // Run IMB at full clock speed
          USIU.PDMCR.B.SLRC = 1;        // Normal, not slow, slew rate for MIOS, TOUCAN C pins
}

asm void init_ext_int_rel (void)
{
! "r0"
                                       ; Tell compiler any scratch registers used
; ASSUMPTIONS-- Reset configuration word (or debugger) set the following bits:
;        bit     19 ETRE = 1     Exception Table Relocation Enable is set
;        bits 24:25 OERC = 00    Exception table base = 0 for MPC56x and MPC53x
;        bits 28:30 ISB = 000   Internal memory Space Base = 0x0

          lis       r0, EIR_table@h     ; Load EIBADR (spr 529) with table base address
          ori       r0, r0, EIR_table@l
          mtspr     529, r0

          mfspr     r0, 560   ; Set BBCMCR[EIR]=1, spr560 bit 20, to enable int. relocation
          ori       r0, r0, 0x0800
          mtspr     560, r0
```

```
}

void init_mios (void)
{
                                        // STEP 1:  MODULE SPECIFIC INIT
                                        // MIOS: enable FREEZE, enable & set prescaler
          MIOS14.MIOS14MCR.B.FRZ = 1;   // MIOS- Activate MIOB freeze if IMB freeze occurs
          MIOS14.MCPSMSCR.B.FREN = 1;   // MIOS- Stop MIOS1 if IMB3 freeze is active
          MIOS14.MCPSMSCR.B.PREN = 1;   // MIOS- PRescaler ENable:  MCPSM counter enabled
          MIOS14.MCPSMSCR.B.PSL = 0;    // MIOS- Clock prescaler set to divide by 16
          MIOS14.MIOS14MCR.B.STOP = 1;  // MIOS - stop all clocks until counters initialized

                                        // CTR 22: Count to 64K with clock of 40 MHz/16/4
          MIOS14.MMCSM22CNT.R  = 0;             // CTR 22- set CouNTer (and latch) to zero
          MIOS14.MMCSM22SCR.B.CLS = 3;          // CTR 22- CLock Selected for prescaler
          MIOS14.MMCSM22SCR.B.CP  = 0xfc;       // CTR 22- Clock Prescaler:  divide by 4
          MIOS14.MMCSM22SCR.B.FREN = 1;         // CTR 22- FReeze ENabled if MIOB freeze occurs

                                        // CTR 6: Count to 64K with clock of 40 MHz/16/8
          MIOS14.MMCSM6CNT.R = 0;               // CTR 6 - set CouNTer (and latch) to zero
          MIOS14.MMCSM6SCR.B.CLS = 3;           // CTR 6 - CLock Selected for prescaler
          MIOS14.MMCSM6SCR.B.CP = 0xf8;         // CTR 6 - Clock Prescaler:  divide by 8
          MIOS14.MMCSM6SCR.B.FREN = 1;          // CTR 6 - FReeze ENabled if MIOB freeze occurs

          MIOS14.MIOS14MCR.B.STOP = 0;  // Restart MIOS clock now that ctrs are initialized

                                        // STEP 2:  LEVEL ASSIGNMENT
          MIOS14.MIOS14LVL1.B.LVL = 6;          // Set counter 22 interrupt level to 22
          MIOS14.MIOS14LVL1.B.TM = 2;
          MIOS14.MIOS14LVL0.B.LVL = 6;          // Set counter  6 interrupt level to 6
          MIOS14.MIOS14LVL0.B.TM  = 0;

                                        // STEP 3:  ENABLE INTERRUPT
          MIOS14.MIOS14ER1.B.EN22 = 1;          // Enable counter 22 overflow to cause interrupt
          MIOS14.MIOS14ER0.B.EN6 = 1;           // Enable counter  6 overflow to cause interrupt

                                        // STEP 4:  SET APPROPRIATE SIMASK BITS
          USIU.SIMASK3.B.IMBIRQ22 = 1;          // Enable IMB level 22 interrupt
          USIU.SIMASK2.B.IMBIRQ6 = 1;           // Enable IMB level 6 interrupt
}

void init_irq_1 (void)
// Assumes initMIOS already executed. MIOS GPIO pin gets initialized here.
{
                                        // STEP 1:  MODULE SPECIFIC INIT
          USIU.SIUMCR.B.MLRC = 0;               // Configure pins for IRQ functions
```

```
        USIU.SIEL.B.ED1 = 1;          // Set IRQ1 input pin to level (0) or edge (1) trigger
        MIOS14.MPIOSM32DR.B.D6 = 1;             // Initialize MIOS GPIO data 6 = 1 (connected to
IRQ1, IRQ3 pins)
        MIOS14.MPIOSM32DDR.B.DDR6 = 1;          // Set MIOS GPIO data direction = output

                                                // STEP 2:  LEVEL ASSIGNMENT
                                      // No assignment -- IRQ pins have fixed priorities

                                                // STEP 3:  ENABLE INTERRUPT
                                                // No enable - done by MIOS and hardware jumper

                                                // STEP 4:  SET APPROPRIATE SIMASK BITS
        USIU.SIMASK2.B.IRQ1 = 1;                // Enable IRQ1 interrupt
}

void init_irq_3 (void)
// Assumes initMIOS already executed.
{
                                                // STEP 1:  MODULE SPECIFIC INIT
        USIU.SIEL.B.ED3 = 1;          // Set IRQ3 input pin to level (0) or edge (1) trigger
//      MIOS14.MPIOSM32DR.B.D5 = 1;   // Initialize MIOS GPIO data 5 = 1 (connected to IRQ3 pin)
//      MIOS14.MPIOSM32DDR.B.DDR5 = 1;// Set MIOS GPIO data direction = output
                                      // STEP 2:  LEVEL ASSIGNMENT
                                      // No assignment -- IRQ pins have fixed priorities

                                      // STEP 3:  ENABLE INTERRUPT
                                      // No enable - done by MIOS and hardware jumper

                                      // STEP 4:  SET APPROPRIATE SIMASK BITS
        USIU.SIMASK2.B.IRQ3 = 1;      // Enable IRQ1 interrupt
}

void main ()
{
        init_565();                   // Perform a simple CPU init

        UIMB.UMCR.B.IRQMUX = 3;       // Enable up to 31 interrupt levels in UIMB
        USIU.SIUMCR.B.EICEN = 1;      // Enable EIC
        init_ext_int_rel();           // Init. Ext. Interrupt & Exception Table Relocation
        USIU.SIUMCR.B.LPMASK_EN = 1;  // Enable Low Priority Mask for nested interrupts

        init_mios();                  // Initialize 2 MIOS counters(slow ctr 6 & fast ctr 22)
        init_irq_1();                 // Enable input pin IRQ1 edge sensitive & MIOS GPIO pin
        init_irq_3();                 // Enable input pin IRQ3 edge sensitive

        asm (" mtspr EIE, r0");       // Enable all interrupts
```

```
          while (1)
          {
                    loopcnt++;          // Up count while waiting for interrupts
          }
}


void ext_irq_1_isr (void)
{
// Summary: service external interrupt pin 1 -
//    1. Clear interrupt flag for IRQ1 in SIPEND register by writing a "1" to SIPEND
//    2. Increment appropriate software counter for that interrupt

          USIU.SIPEND2.R = 0x02000000;            // Since IRQ1 is edge sensitive, must clear it
                                                  // by writing 1 to IRQ1 bit, 0's to other bits
          irq_1_ctr++;                            // Increment global variable for this ISR

          USIU.SISR2.R = 0x02000000;              // Clear appropriate SISR bit for this ISR only
}

void ext_irq_3_isr (void)
{
// Summary: service external interrupt pin 3 -
//    1. Clear interrupt flag for IRQ1 in SIPEND register by writing a "1" to SIPEND
//    2. Increment appropriate software counter for that interrupt

          USIU.SIPEND2.R = 0x00002000;            // Since IRQ1 is edge sensitive, must clear it
                                                  // by writing 1 to IRQ1 bit, 0's to other bits
          irq_3_ctr++;                            // Increment global variable for this ISR

          USIU.SISR2.R = 0x00002000;              // Clear appropriate SISR bit for this ISR only
}

void imb_irq_6_isr (void)
{
//  Summary: service IMB interrupt on level 6 (MIOS CTR 6) -
//    1. Clear flag in interrupt status register (MIOS14SR0:1) by:
//                A. reading the register,
//                B. then write 0 to bit of the active interrupt flag and 1's to other bits
//    2. Increment appropriate software counter for that interrupt
//    3. Toggle MIOS GPIO data 6 pin to drive active external IRQ pins 1 and 3
//    4. Clear appropriate interrupt mask bit in SISR register

          unsigned int irq_status = 0;            // Dummy variable to read status register

          irq_status = MIOS14.MIOS14SR0.R;        // Read MIOS interrupt status register
```

```
          MIOS14.MIOS14SR0.R = 0xffbf;            // Write 0 to active IRQ flag; 1's to other bits
          mc6_irq_ctr++;                          // Increment global variable for this ISR

IRQ: MIOS14.MPIOSM32DR.B.D6 = 0;                  // Output a 0 on MIOS GPIO data 6, forcing IRQ1
                                                  //      and IRQ3 active at the same time
          MIOS14.MPIOSM32DR.B.D6 = 1;             // Output a 1 on MIOS GPIO data 6

          USIU.SISR2.R = 0x00200000;              // Clear appropriate SISR bit for this ISR only
}

void imb_irq_22_isr (void)
{
//  Summary: service IMB interrupt on level 6 (MIOS CTR 6) -
//    1. Clear flag in interrupt status register (MIOS14SR0:1) by:
//                A. reading the register,
//                B. then write 0 to bit of the active interrupt flag and 1's to other bits
//    2. Increment appropriate software counter for that interrupt
//    3. Clear appropriate interrupt mask bit in SISR register

          unsigned int irq_status = 0;            // Dummy variable to read status register

          irq_status = MIOS14.MIOS14SR1.R;// Read MIOS interrupt status register
          MIOS14.MIOS14SR1.R = 0xffbf;            // Write 0 to active IRQ flag; 1's to other bits
          mc22_irq_ctr++;                         // Increment global variable for this ISR

          USIU.SISR3.R = 0x20000000;              // Clear appropriate SISR bit for this ISR only
}
```

6.5.2 Example 4: Assembly Source

```
          .globl              EIR_table

          .section .abs.00000008
                    ba _start           ; System reset exception, per crt0 file for ETRE=1

          .text

ext_irq_1_handler:                      ; This interrupt handler saves context
                                        ; necessary to call a c function

                                        ; STEP 1:  SAVE "MACHINE CONTEXT"
          stwu      sp, -80 (sp)        ; Create stack frame and store back chain
          stw       r3, 36 (sp)         ; Save working register
          mfsrr0    r3                  ; Get SRR0
          stw       r3, 12 (sp)         ; and save SRR0
          mfsrr1    r3                  ; Get SRR1
          stw       r3, 16 (sp)         ; and save SRR1
```

```
                                    ; STEP 2:  MAKE MSR[RI] RECOVERABLE and SET EE
          mtspr EIE, r3             ; Set MSR[EE] bit to re-enable interrupts
                                     and MSR[RI] bit to indicate recoverable status

                                    ; STEP 3: SAVE OTHER APPROPRIATE CONTEXT
          mflr      r3              ; Get LR
          stw       r3, 8 (sp)      ; and save LR
          bl        finish_saving_context

                                    ; STEP 4: DETERMINE SOURCE OF INTERRUPT
                                    ; STEP 5: BRANCH TO INTERRUPT SERVICE ROUTINE
          bl        ext_irq_1_isr

                                    ; Return from C routine here
                                    ; STEP 6: RESTORE CONTEXT
          b         restore_context


ext_irq_3_handler:                  ; This interrupt handler saves context
                                    ; necessary to call a c function

                                    ; STEP 1:  SAVE "MACHINE CONTEXT"
          stwu      sp, -80 (sp)    ; Create stack frame and store back chain
          stw       r3, 36 (sp)     ; Save working register
          mfsrr0    r3              ; Get SRR0
          stw       r3, 12 (sp)     ; and save SRR0
          mfsrr1    r3              ; Get SRR1
          stw       r3, 16 (sp)     ; and save SRR1


                                    ; STEP 2:  MAKE MSR[RI] RECOVERABLE and SET EE
          mtspr EIE, r3             ; Set MSR[EE] bit to re-enable interrupts
                                    ; and MSR[RI] bit to indicate recoverable status

                                    ; STEP 3: SAVE OTHER APPROPRIATE CONTEXT
          mflr      r3              ; Get LR
          stw       r3, 8 (sp)      ; and save LR
          bl        finish_saving_context

                                    ; STEP 4: DETERMINE SOURCE OF INTERRUPT
                                    ; STEP 5: BRANCH TO INTERRUPT SERVICE ROUTINE
          bl        ext_irq_3_isr

                                    ; Return from C routine here
```

```
                                    ; STEP 6: RESTORE CONTEXT
          b         restore_context


imb_irq_6_handler:                  ; This interrupt handler saves context
                                    ; necessary to call a c function

                                    ; STEP 1:  SAVE "MACHINE CONTEXT"
          stwu sp, -80 (sp)         ; Create stack frame and store back chain
          stw       r3, 36 (sp)     ; Save working register
          mfsrr0    r3              ; Get SRR0
          stw       r3, 12 (sp)     ; and save SRR0
          mfsrr1    r3              ; Get SRR1
          stw       r3, 16 (sp)     ; and save SRR1


                                    ; STEP 2:  MAKE MSR[RI] RECOVERABLE and SET EE
          mtspr EIE, r3             ; Set MSR[EE] bit to re-enable interrupts
                                    ; and MSR[RI] bit to indicate recoverable status

                                    ; STEP 3: SAVE OTHER APPROPRIATE CONTEXT
          mflr      r3              ; Get LR
          stw       r3, 8 (sp)      ; and save LR
          bl        finish_saving_context

                                    ; STEP 4: DETERMINE SOURCE OF INTERRUPT
                                    ; STEP 5: BRANCH TO INTERRUPT SERVICE ROUTINE
          bl        imb_irq_6_isr

                                    ; Return from C routine here
                                    ; STEP 6: RESTORE CONTEXT
          b         restore_context


imb_irq_22_handler:                 ; This interrupt handler saves context
                                    ; necessary to call a c function

                                    ; STEP 1:  SAVE "MACHINE CONTEXT"
          stwu      sp, -80 (sp)    ; Create stack frame and store back chain
          stw       r3, 36 (sp)     ; Save working register
          mfsrr0    r3              ; Get SRR0
          stw       r3, 12 (sp)     ; and save SRR0
          mfsrr1    r3              ; Get SRR1
          stw       r3, 16 (sp)     ; and save SRR1
```

```
                                    ; STEP 2:  MAKE MSR[RI] RECOVERABLE and SET EE
          mtspr EIE, r3             ; Set MSR[EE] bit to re-enable interrupts
                                    ; and MSR[RI] bit to indicate recoverable status

                                    ; STEP 3: SAVE OTHER APPROPRIATE CONTEXT
          mflr      r3              ; Get LR
          stw       r3, 8 (sp)      ; and save LR
          bl        finish_saving_context

                                    ; STEP 4: DETERMINE SOURCE OF INTERRUPT
                                    ; STEP 5: BRANCH TO INTERRUPT SERVICE ROUTINE
          bl        imb_irq_22_isr

                                    ; Return from C routine here
                                    ; STEP 6: RESTORE CONTEXT
          b         restore_context


finish_saving_context:
                                    ; Save other registers to allow calling a
                                    ; c function

          mfxer     r3              ; Get XER
          stw       r3, 20 (sp)     ; and save XER
          mfspr     r3, CTR         ; Get CTR
          stw       r3, 24 (sp)     ; and save CTR
          mfcr      r3              ; Get CR
          stw       r3, 28 (sp)     ; and save CR
          stw       r0, 32 (sp)     ; Save R0
          stw       r4, 40 (sp)     ; Save R4 to R12
          stw       r5, 44 (sp)
          stw       r6, 48 (sp)
          stw       r7, 52 (sp)
          stw       r8, 56 (sp)
          stw       r9, 60 (sp)
          stw       r10, 64 (sp)
          stw       r11, 68 (sp)
          stw       r12, 72 (sp)
          blr                       ; Return to interrupt routine


restore_context:
          lwz       r0, 32 (sp)     ; Restore gprs except R3
          lwz       r4, 40 (sp)
          lwz       r5, 44 (sp)
          lwz       r6, 48 (sp)
```

```
          lwz       r7, 52 (sp)
          lwz       r8, 56 (sp)
          lwz       r9, 60 (sp)
          lwz       r10, 64 (sp)
          lwz       r11, 68 (sp)
          lwz       r12, 72 (sp)
          lwz       r3, 20 (sp)         ; Get XER
          mtxer     r3                  ; and restore XER
          lwz       r3, 24 (sp)         ; Get CTR
          mtctr     r3                  ; and restore CTR
          lwz       r3, 28 (sp)         ; Get CR
          mtcrf     0xff, r3            ; and restore CR
          lwz       r3, 8 (sp)          ; Get LR
          mtlr      r3                  ; and restore LR
          mtspr     NRI, r3             ; Clear recoverable bit, MSR[RI]
                                        ; which also sets MSR[EE]=0
                                        ; Note: breakpoints CANNOT be set
                                        ; from now thru the rfi instruction
          lwz       r3, 12 (sp)         ; Get SRR0 from stack
          mtsrr0    r3                  ; and restore SRR0
          lwz       r3, 16 (sp)         ; Get SRR1 from stack
          mtsrr1    r3                  ; and restore SRR1
          lwz       r3, 36 (sp)         ; Restore R3
          addi sp,  sp, 80              ; Clean up stack


                                        ; STEP 7:  Return to Program
          rfi                           ; End of Interrupt

; ===================================================================
; External Interrupt Relocation branch table
; NOTE:  the .space directive may not be suitable for code compression
; Table reserves 96 words starting on a boundary a multiple of 0x2000

          .section .abs.00002000

EIR_table:                                              ; Branch forever if routine not written

EXT_IRQ_0_TRAP_ERROR:  ba       EXT_IRQ_0_TRAP_ERROR    ; EXT_IRQ_0 should generate NMI [1]
;EXT_IRQ_0_TRAP_ERROR: rfi                              ; For production code
                       .space 4
Level_0:               ba       Level_0
```

[1]The EXT_IRQ_0 interrupt generates NMI instead of this vector. However, improper interrupt programming might cause this vector. For debugging, the use of a trap handler that saves SRR0 and SRR1 is recommended. Saving these registers allows backtracking to find the offending code. For production code, it is recommended that an rfi instruction is used.

```
                    .space 4
IMB_IRQ_0:          ba      IMB_IRQ_0
                    .space 4
IMB_IRQ_1:          ba      IMB_IRQ_1
                    .space 4
IMB_IRQ_2:          ba      IMB_IRQ_2
                    .space 4
IMB_IRQ_3:          ba      IMB_IRQ_3
                    .space 4
EXT_IRQ_1:          ba      ext_irq_1_handler       ; Branch to handler
                    .space 4
Level_1:            ba      Level_1
                    .space 4
IMB_IRQ_4:          ba      IMB_IRQ_4
                    .space 4
IMB_IRQ_5:          ba      IMB_IRQ_5
                    .space 4
IMB_IRQ_6:          ba      imb_irq_6_handler       ; Branch to handler
                    .space 4
IMB_IRQ_7:          ba      IMB_IRQ_7
                    .space 4
EXT_IRQ_2:          ba      EXT_IRQ_2
                    .space 4
Level_2:            ba      Level_2
                    .space 4
IMB_IRQ_8:          ba      IMB_IRQ_8
                    .space 4
IMB_IRQ_9:          ba      IMB_IRQ_9
                    .space 4
IMB_IRQ_10:         ba      IMB_IRQ_10
                    .space 4
IMB_IRQ_11:         ba      IMB_IRQ_11
                    .space 4
EXT_IRQ_3:          ba      ext_irq_3_handler       ; Branch to handler
                    .space 4
Level_3:            ba      Level_3
                    .space 4
IMB_IRQ_12:         ba      IMB_IRQ_12
                    .space 4
IMB_IRQ_13:         ba      IMB_IRQ_13
                    .space 4
IMB_IRQ_14:         ba      IMB_IRQ_14
                    .space 4
IMB_IRQ_15:         ba      IMB_IRQ_15
                    .space 4
EXT_IRQ_4:          ba      EXT_IRQ_4
```

```
                    .space 4
Level_4:            ba      Level_4
                    .space 4
IMB_IRQ_16:         ba      IMB_IRQ_16
                    .space 4
IMB_IRQ_17:         ba      IMB_IRQ_17
                    .space 4
IMB_IRQ_18:         ba      IMB_IRQ_18
                    .space 4
IMB_IRQ_19:         ba      IMB_IRQ_19
                    .space 4
EXT_IRQ_5:          ba      EXT_IRQ_5
                    .space 4
Level_5:            ba      Level_5
                    .space 4
IMB_IRQ_20:         ba      IMB_IRQ_20
                    .space 4
IMB_IRQ_21:         ba      IMB_IRQ_21
                    .space 4
IMB_IRQ_22:         ba      imb_irq_22_handler ; Branch to handler
                    .space 4
IMB_IRQ_23:         ba      IMB_IRQ_23
                    .space 4
EXT_IRQ_6:          ba      EXT_IRQ_6
                    .space 4
Level_6:            ba      Level_6
                    .space 4
IMB_IRQ_24:         ba      IMB_IRQ_24
                    .space 4
IMB_IRQ_25:         ba      IMB_IRQ_25
                    .space 4
IMB_IRQ_26:         ba      IMB_IRQ_26
                    .space 4
IMB_IRQ_27:         ba      IMB_IRQ_27
                    .space 4
EXT_IRQ_7:          ba      EXT_IRQ_7
                    .space 4
Level_7:            ba      Level_7
                    .space 4
IMB_IRQ_28:         ba      IMB_IRQ_28
                    .space 4
IMB_IRQ_29:         ba      IMB_IRQ_29
                    .space 4
IMB_IRQ_30:         ba      IMB_IRQ_30
                    .space 4
IMB_IRQ_31:         ba      IMB_IRQ_31
```

7 Conclusion

Speed is typically the critical factor when servicing interrupts. Using features of the EIC, one can reduce the overall time in an interrupt service routine (ISR). The number of instructions for interrupt overhead in the examples and their relative execution times are summarized in Table 13, which includes the following metrics:

- **Measured time: interrupt request to ISR:** Time measured on an oscilloscope between events:
  - IRQOUT/ negating (indicating interrupt request)
  - Watchpoint at the first instruction of the ISR (after context is saved).
- **Total number of overhead instructions:** Includes the initial branch to the interrupt handler plus the steps listed.

While Table 13 provides some useful data points, there are other issues the system designer must consider which may improve or hurt these numbers:

- Routines written entirely in assembly language will have much less overhead as shown in Table 13. This minimal ISR is non-recoverable because SRR0 and SRR1 are not saved and MSR[RI] is not set. To make it recoverable, an additional 5 or 6 instructions would be added in saving and restoring context.
- Users who want to write interrupt routines in C can optimize interrupt responses if they use the external interrupt relocation (EIR) feature along with a compiler that makes intelligent decisions regarding how much context is saved. For example, an ISR calling multiple C functions would save a normal context, while a single-function, self-contained ISR only saves those registers accessed within the function.
- Often an operating system will save many more registers, so all overheads will significantly increase.
- Using lswx (load string word indexed) and stswx (store string word indexed) instructions in the context save and restore sections reduces the total number of instructions. However, there may not be a performance improvement.

Table 13. Summary of Interrupt Overhead for Examples

ISR Step	Regular Interrupt Controller (Level < 8)	Regular Interrupt Controller (Level > 7)	EIC (All Levels)	EIC + External Interrupt Relocation (All Levels)	EIC + External Interrupt Relocation (All Levels)	EIC + External Interrupt Relocation + Lower Priority Request Masking
	Example 1, Level 6, C ISR	Example 1, Level 22, C ISR	Example 2, C ISR	Example 3, C ISR	Example 3, non-recov. assembler ISR	Example 4 [1], non-nest C ISR
Branch to interrupt handler	1	1	1	1	1	1
1. Save machine context (SRR0:1)	6	6	6	6	0	6
2. Set MSR[RI]	1	1	1	1	0	1
3. Save other context	18	18	18	20	2	20
4. Determine interrupt source	6	15	6	0	0	0

Table 13. Summary of Interrupt Overhead for Examples

5. Branch to interrupt service routine	2	2	2	1	0	1
Total number of instructions to get to ISR	**34**	**43**	**34**	**29**	**3**	**29**
Measured time: interrupt request to ISR	**990 nsec**	**1300 nsec**	**970 nsec**	**905 nsec**	**235 nsec**	**880 nsec**
6. Restore Context	25	25	25	26	2	25
7. Return to Program	1	1	1	1	1	1
Total number of overhead instructions	**60**	**69**	**60**	**56**	**6**	**55**

[1] The interrupt used for measurement of Example 4 is not nested.

Appendix A
IRQ Pin Behavior

A.1 Edge- Vs. Level-Sensitive

IRQ pins are programmed to be edge- or level-sensitive in the SIEL register. If enabled in SIMASK, both edge- and level-sensitive interrupt pins set their corresponding SIPEND bit when their input signal goes active low. Their primary difference is in how to “clear” their interrupt.

A.1.1 Edge-Sensitive: SIEL[ED*x*] = 1

After the interrupt is recognized, it must be cleared in the interrupt handler by clearing the appropriate bit in SIPEND. This is accomplished by writing a 1 to the appropriate SIPEND bit without changing the other bits in the register.

If the interrupt bit for SIPEND is not cleared, that interrupt will immediately be taken again when interrupts are re-enabled (for example, after the rfi instruction). This could cause an endless loop.

Edge sensitivity is normally recommended for interrupt pins.

A.1.2 Level-Sensitive: SIEL[ED*x*] = 0

After the interrupt is recognized, it must be cleared by negating the signal at the interrupt input pin itself, i.e., making the logic signal release to a high state. For example, if another chip is driving the IRQ pin low, then the interrupt handler could signal the other chip to negate the signal (producing a logical high voltage).

If the signal to the IRQ pin is not negated, that interrupt also will immediately be taken again when interrupts are re-enabled, for example after the rfi instruction. This also could cause an endless loop.

A.2 IRQ0 Pin: Non-Maskable Interrupt (NMI)

The IRQ0 interrupt (also called NMI) is cleared in the same way as the edge- and level- cases for the other IRQ pins; however, it is normally cleared in the reset exception routine instead of the external interrupt exception routine. This is because NMI, though it does not cause a reset, uses the reset vector. The reset routine can test whether an NMI occurred by reading the SIPEND bit for IRQ0.

Unlike the other IRQ pins, future non-maskable interrupts will not be recognized if IRQ0 is not cleared. This is true independent of whether it is edge or level sensitive. For example, if IRQ0 is level sensitive and its input remains active low then code starts at the reset location and continues. For a subsequent NMI to be recognized, the NMI condition must be cleared.

8 Revision History

Table 14 is a revision history for this document.

Table 14. Revision History

Revision Number	Substantive Changes	Release Date
0	Initial Release.	5/2003

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor

AN2472/D, Rev. 0, 5/2003