MOTOROLA
SEMICONDUCTOR
APPLICATION NOTE

Order this document
by AN1281/D

MPC509 Interrupts

By Steve Mihalik

MPC509 Interrupts

by Steve Mihalik

The MPC509 interrupt controller receives interrupt requests from multiple interrupt sources and generates a single interrupt signal to the RCPU, as shown in **Figure 1**. This application note describes the function of the interrupt controller and related interrupt registers, and also provides example initialization and handler routines.

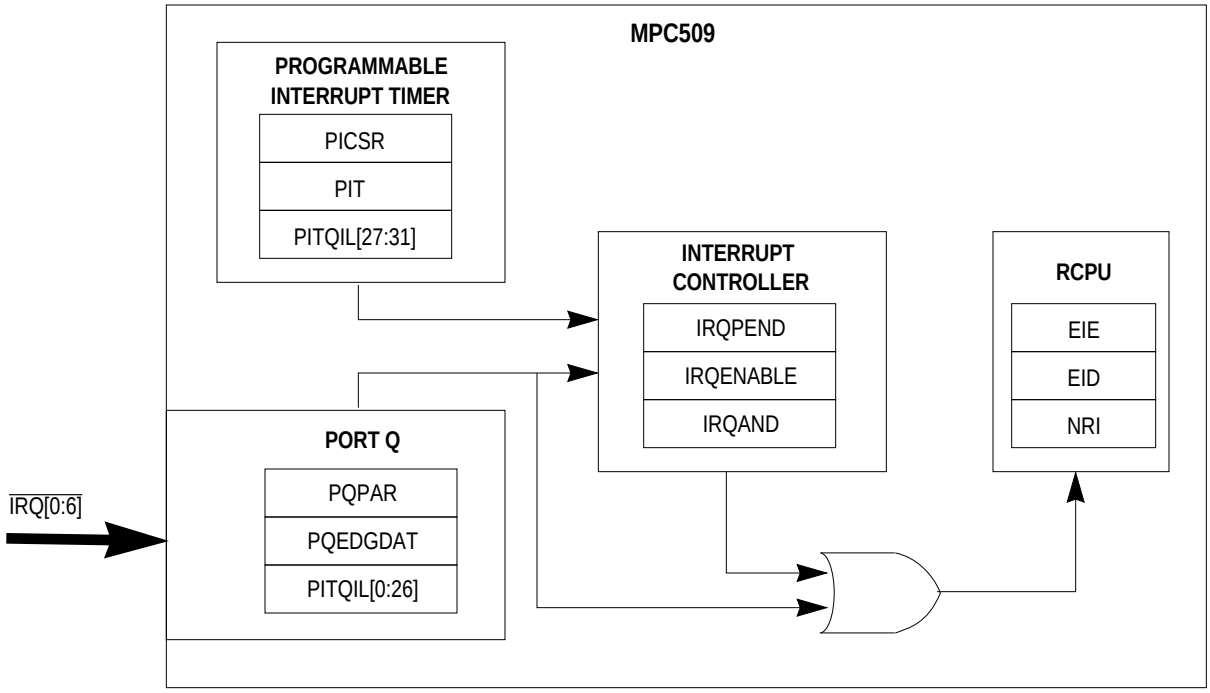

Figure 1 MPC509 Interrupt Structure Block Diagram

1 Interrupt Basics

The PowerPC™ architecture uses unique vector offsets for “exceptions” from normal processing. When an exception occurs, a hardware context switch takes place and processing branches to the appropriate exception vector address. The address is the sum of the vector offset plus a physical base address of either 0x0000 0000 or 0xFFF0 0000 as determined by the IP bit in the machine status register (MSR). In this note, use of the 0x0000 0000 base address is assumed. **Table 1** shows types of exceptions and associated vectors.

Table 1 Exceptions and Vectors

Exception Type	Vector Offset
Reserved	0x00000
System reset	0x00100
Machine check	0x00200
Reserved	0x00300
Reserved	0x00400
Interrupts	0x00500
Alignment	0x00600
etc.	etc.

There is a difference in terminology between Motorola and IBM PowerPC literature. Motorola uses the term *exception* to mean any event which causes the processor to transfer control to one of the vector addresses in the range 0xxxx0 0000 to 0xxxx0 3FFF, as well as to mean the action taken by the processor in response to such an event. The term *interrupt* is used by Motorola to refer to events and the corresponding processor responses which are associated with offset 0x00500.

IBM uses the term *exception* to refer to the event and the term *interrupt* to refer to the action taken by the processor in response to any exception, independent of the vector address. Thus, IBM literature describes floating point overflow as a floating point exception, which causes the processor to take a floating point interrupt. The specific interrupt with the vector offset of 0x0500 is called the *external interrupt* to distinguish it from other interrupts.

In Motorola terminology, interrupts are one type of exception, and all interrupts share vector offset 0x00500. Interrupt “sources” originate interrupt service requests. These sources include the external interrupt pins ($\overline{IRQ0}$ through $\overline{IRQ6}$), the periodic interrupt timer (PIT) and on-chip peripheral residing on the intermodule bus. Each source is identified by a “level”. Some interrupt sources, specifically the $\overline{IRQ3}$ through $\overline{IRQ6}$ pins, have fixed levels. Other levels can be assigned by the system designer. **Table 2** is a summary of interrupt sources and levels.

Table 2 Sources Versus Levels

Source	Level
External Pins:	
$\overline{IRQ0}$	Assignable between 0:31
$\overline{IRQ1}$	Assignable between 0:31
$\overline{IRQ2}$	Assignable between 0:31
$\overline{IRQ3}$	6
$\overline{IRQ4}$	8
$\overline{IRQ5}$	10
$\overline{IRQ6}$	12
Internal:	
PIT	Assignable between 0:31
IMB Peripheral	(Future)

When an interrupt occurs, its corresponding level is set in the 32-bit interrupt request pending (IRQPEND) register. Each bit represents one level, hence up to 32 levels (or sources) are possible. This allows for future expansion of interrupt sources in the MPC509.

1.1 Enabling and Configuring Individual Interrupts

Individual interrupt levels are enabled and disabled by the 32-bit IRQENABLE register. Setting a bit enables a source of the corresponding level to cause an actual interrupt; clearing a bit disables the corresponding level. For instance, to enable interrupt level 3 and disable all other levels, set bit 3 of IRQENABLE to one and clear all other bits to zero.

IRQENABLE

0	1	2	3	4	…	31
0	0	0	1	0	…	0

External interrupt pins can be configured for either edge or level sensitivity. An interrupt pin that is not needed for interrupt function can be configured for general-purpose input or output.

1.2 Interrupt Identification

When an interrupt occurs, software reads the IRQAND register to find out which enabled interrupt caused the exception. The IRQAND register is simply the logical AND of the IRQPEND register (which indicates pending levels) with the IRQENABLE register (which indicates enabled levels).

$$IRQAND = IRQPEND \bullet IRQENABLE$$

1.3 Priorities

If two interrupts occur at the same time, a priority scheme is needed to determine which one to handle first. MPC509 interrupt priorities are not determined by hardware, but rather by the way software responds to an interrupt. An easy and efficient way to handle priorities is to assign levels in order of priority, using level 0 for the highest priority and level 31 for the lowest priority. The instruction “count leading zeros in a word” (cntlzw) can be used to quickly identify the first bit set, i.e., the highest priority interrupt.

More general priority schemes can be implemented by means of a set of bitmasks to be ANDed with the IRQAND register. The first mask used has a one in the bit position corresponding to each level in the highest priority set, and so on. Such schemes allow multiple interrupts at the same priority, and also allow the implementation of dynamic priorities by changing the masks.

2 Initialization Example

Table 3 shows steps used to initialize interrupts during a reset routine. Keep in mind that after reset, most processor resources like the floating point unit, interrupts, etc. are disabled.

Table 3 Interrupt Initialization

Step	Action	Registers Used
1	Configure $\overline{IRQ}$ pins as interrupts and assign sensitivity	PQPAR
2	Assign interrupt levels	PITQIL
3	Initialize PIT	PIT
4	Enable individual interrupts	IRQENABLE
5	Enable external interrupts	EIE

2.1 Configure $\overline{IRQ}$ Pins as Interrupts and Assign Sensitivity

The $\overline{IRQ}$ pins can be used to make interrupt service requests or for general-purpose input/output. Pin use is determined by the port Q pin assignment register (PQPAR). PQPAR contains two 2-bit configuration fields for each pin.

The port Q pin assignment (PQPA) field determines whether a pin is used for general-purpose input, for general-purpose output, for interrupt requests to the RCPU, or for interrupt requests to the interrupt controller.

The port Q edge (PQEDGE) field determines whether an interrupt pin is level-sensitive, falling-edge sensitive, rising-edge sensitive, or rising-and-falling edge sensitive.

Pins that are configured for interrupt requests to the RCPU bypass the interrupt controller completely, and cannot be assigned an interrupt level or individually enabled and disabled. It is assumed that most users will configure pins for interrupt requests to the interrupt controller.

2.2 Assign Interrupt Levels

In this example the assignable levels are also used as priorities. The PIT/port Q interrupt levels (PITQIL) register is used to set up the levels. Five bits per interrupt are used to assign a level between 0 and 31 for the PIT and pins $\overline{\text{IRQ0}}$, $\overline{\text{IRQ1}}$, and $\overline{\text{IRQ2}}$.

2.3 Initialize PIT

An initial 16-bit PIT value is loaded into the periodic interrupt timer (PIT) register. When the PIT is enabled, the register is automatically decremented, and an interrupt service request is made each time the count passes through zero. The initial value is automatically reloaded into the register after each PIT time-out.

The periodic interrupt control and select register (PICSR) controls the PIT. Bits and fields in PICSR perform the following functions:

- Select PIT clock frequency
- Define the 16-bit count value for the PIT
- Enable the PIT counter
- Enable the PIT interrupt
- Indicate PIT status (report whether a PIT interrupt request has been asserted)

2.4 Enable Individual Interrupts

Load the IRQENABLE register with a value to enable the desired interrupt levels. A level is enabled by setting the corresponding bit to one or disabled by clearing the corresponding bit to zero. For example, setting bit 2 of IRQENABLE enables interrupt level 2.

2.5 Enable External Interrupts

Following reset, all interrupts and other exceptions are disabled. To enable interrupts, the external enable (EE) bit in the machine state register (MSR) must be set. The recoverable interrupt (RI) bit must also be set. This topic is discussed in subsequent sections. Both the EE and RI bits can quickly be set by using one of the special purpose registers (SPR) called the external interrupt enable (EIE). The only purpose of the EIE is to efficiently set the EE and RI bits, which is accomplished by loading any value to EIE.

In the PowerPC architecture, an SPR must be loaded from a general-purpose register using the move to SPR instruction. Any GPR can be used as a source. The following code enables interrupts and sets RI:

```
mtspr             EIE, gpr0
```

3 Exception Context Switch

After an interrupt or any other exception is recognized, the hardware automatically performs a context switch, which includes the following:

A. Saves the current state of the machine:
   1. The address of the next instruction is saved in register SRR0
   2. The MSR value is saved in register SRR1
B. Changes the MSR:
   1. Disables further interrupts by clearing the EE bit
   2. Sets the privilege level to supervisor but clears the PR bit

3. Clears the RI bit to indicate that the interrupt may not be able to recover if another exception immediately follows.

C. Branches to the interrupt exception vector

4 Key Handler Issues

4.1 Where to Store Local Data for Handler

Interrupt handlers require at least enough data storage to preserve the previous state of the machine. Typically storage is allocated on the stack. Special purpose registers SPRG0:3 can also be used for storage, but the registers provide only four words of storage, and using them prevents nesting of exceptions. Since this method makes the interrupt handler non-reentrant, it can only be used for the highest priority interrupt.

Assuming the stack is used for storage, care must be taken to ensure there is enough room on the stack for a worst case of nested exceptions. All stack manipulation code must also be carefully designed to ensure that an interrupt during a stack operation does not cause problems. Some programs use separate stacks for interrupt and non-interrupt code.

4.2 How Long Interrupts are Disabled

As long as interrupts are disabled (EE bit reset in the MSR), additional interrupts are not recognized. To minimize latency of the next interrupt, the interrupt handler should be as short as possible (unless the handler routines check for higher priority interrupts during its processing).

4.3 Non-Maskable Exceptions During Interrupt Handler Routines

When an exception occurs, hardware automatically saves the state of the machine in registers SRR0 and SRR1. When another exception occurs, the contents of SRR0 and SRR1 are overwritten. If the machine state of an exception is lost, the program cannot recover.

Interrupts can be masked, but most exceptions are non-maskable. To minimize the risk of being unable to recover from nested exceptions, take these precautions when writing interrupt or other exception handlers:

1. Save registers that can be altered by non-maskable exceptions early in the handler, preferably in the prologue. These registers include SRR0 and SRR1, and in certain exceptions the DAR and DSISR registers.
2. Once these registers are saved, set the recoverable interrupt (RI) bit in the machine status register.
3. Ensure that exception-generating instructions (like “system call”) are not executed during the prologue or epilogue (between restoring of these registers and execution of the “rfi” instruction).

5 Example Interrupt Handler

The steps for an interrupt handler vary by application, but the issues are similar. **Table 4** shows a general case. Some general-purpose registers must be used, so the previous contents of these registers needs to be stored (typically on the stack) and then restored before the end of the handler.

Table 4 Interrupt Handler

Step	Action	Registers Used
1	Save previous state	SRR0, SRR1, one GPR
2	Set the recoverable interrupt bit	EID
3	Identify interrupt source	IRQAND, one GPR
4	Branch to handler	None
5	Perform handler functions	Varies
6	Housekeeping	PQEDGDAT, NRI, SRR0, SRR1, GPR(s)

5.1 Save Previous State

Since a non-maskable exception (like RESET) can occur just after an interrupt takes place, the previous state of the machine (SRR0 and SRR1) must be stored on the stack as a first step.

Only general-purpose registers (GPR) can be written to memory, so a GPR is needed for temporary storage of SRR0 and SRR1. Asynchronous exceptions, like interrupts, must treat all registers as non-volatile — the contents must be preserved so they can be restored at the end of the interrupt routine. Synchronous exceptions, like system call instructions, can treat registers as in function or subroutine calls.

After initial stack functions, like creating a stack frame, the contents of the GPR used for temporary storage must be saved on the stack. After that, the code to save SRR0 and SRR1 on the stack using GPR4 as temporary storage and a predefined GPR as a stack pointer (SP) could look like:

```
mfspr    gpr4, SRR0           # move spr SRR0 to gpr4
stw      gpr4, 12 (SP)        # store SRR0 value 12 bytes above SP
mfspr    gpr4, SRR1           # move spr SRR1 to gpr4
stw      gpr4, 16 (SP)        # store SRR1 value 16 bytes above SP
```

5.2 Set the Recoverable Interrupt Bit

Once the previous state is saved or copied into memory, it is possible to recover if a non-maskable exception occurs. The recoverable interrupt bit in the MSR must be set to communicate to any non-maskable exception routine that recovery is possible.

Once again, a special purpose register has been implemented to quickly assist this operation. Loading any value to the external interrupt disable (EID) SPR sets the RI bit in the MSR while keeping interrupts disabled (EE bit 0). Again, any GPR could be used to write to that SPR, since the content of the GPR is unaltered. For example:

```
mtspr       EID, gpr0
```

5.3 Identify Interrupt Source

Read the IRQAND register (IRQAND = IRQENABLE • IRQPEND) to identify which enabled level caused the interrupt. Under the priority scheme set up during initialization, the most significant bit is the higher priority interrupt to be serviced. The PowerPC instruction “count leading zeros in the word” (cntlzw) can be used to identify the set bit in a register. This instruction counts the number of consecutive zero bits starting at bit 0, the most significant bit. The result, from 0 through 32, is put in the first register operand.

The following code identifies the source interrupt. Since a GPR is required, the same GPR is used as before for temporary storage.

```
lis      gpr4, SIUBASE_UPPER           # gpr4 now has 8007 0000
ori      gpr4, gpr4,IRQAND_OFFSET      # gpr4 has E.A. of IRQAND
lwz      gpr4, 0(gpr4)                 # gpr4 has IRQAND contents
cntlzw   gpr4, gpr4                    # Find level number
```

5.4 Branch to Appropriate Routine

There are a number of ways to branch to the individual routine once the integer value representing the interrupt level is loaded into GPR4. For example, a table of interrupt handler addresses could be constructed, and GPR4 could be used to index into that table, get the appropriate interrupt handler address for a level, and branch to that address.

5.5 Perform Handler Functions

Once in the handler, one approach would be to do all the interrupt handler functions and then leave. But since interrupts are still disabled, this would result in the longest interrupt latency. A popular approach is to use a multitasking kernel operating system which would send appropriate messages or flags and then exit.

5.6 Housekeeping Items and Return

Before exiting the interrupt service routine, a number of housekeeping items must be performed:

1. Clear the pending interrupt condition which set the bit in the IRQPEND register.
2. If the PIT caused the interrupt, clear the PIT status (PS) bit in PICSR.
3. If the interrupt was caused by an external interrupt pin in edge-detect mode, clear the corresponding status bit in the port Q edge detect/data (PQEDGDAT) register by first reading the bit as a one and then writing it to a zero.
4. Restore any previously used GPR (e.g. GPR4 used above).
5. In case a non-maskable exception occurred during this interrupt source routine,

   A. Disable the recoverable interrupt (RI) bit in the MSR by writing any value to the NRI register:

   ```
   mtspr        NRI, gpr0
   ```

   B. Restore SRR0, SRR1 (and stack pointer if it changed).

6. Execute the return from interrupt instruction (which also enables interrupts again):

   ```
   rfi
   ```

6 Summary

With proper assignment of sources to levels, the MPC509 interrupt controller provides an efficient priority mechanism. The interrupt level can also be used as an offset to provide a branch to the appropriate handler routine. Key issues a system designer must address include handling non-maskable exceptions while interrupts are disabled, handler data storage, and the length of time interrupts are disabled.

How to reach us:
USA/EUROPE/Locations Not Listed: Motorola Literature Distribution;
P.O. Box 5405, Denver Colorado 80217. 1-800-441-2447, (303) 675-2140
Mfax™: RMFAX0@email.sps.mot.com - TOUCHTONE (602) 244-6609
INTERNET: http://Design-NET.com
JAPAN: Nippon Motorola Ltd.; Tatsumi-SPD-JLDC,
6F Seibu-Butsuryu-Center, 3-14-2 Tatsumi Koto-Ku, Tokyo 135, Japan. 81-3-3521-8315
ASIA PACIFIC: Motorola Semiconductors H.K. Ltd.; 8B Tai Ping Industrial Park,
51 Ting Kok Road, Tai Po, N.T., Hong Kong. 852-26629298

MOTOROLA

AN1281/D