Application Note

Nesting Interrupts with Z8 Encore! XP® MCUs

AN014105–1207

Abstract

This application note describes how to nest interrupts when developing a project with Zilog's Z8 Encore! XP® MCU. It explains a method of nesting interrupts which includes how to initialize interrupts, set interrupt priorities, and initialize interrupt vectors. This application note also provides a brief introduction of the Z8 Encore! XP interrupt controller.

Three source code files (in WinZip format) associated with this application note are listed below:

- AN0141-SC01—For 64 KB Z8 Encore! XP MCU (Z8F640x)
- AN0141-SC02—For 8 KB Z8 Encore! XP MCU (Z8F082x)
- AN0141-SC03—For 64 KB Z8 Encore! XP MCU (Z8F642x)

Z8 Encore! XP® Flash MCU Overview

Z8 Encore! XP products are based on new eZ8 CPU and introduce Flash memory to Zilog's extensive line of 8-bit microcontrollers. Flash memory in-circuit programming capability allows faster development time and program changes in the field. The high-performance register-to-register based architecture of the eZ8 core maintains backward compatibility with Zilog's popular Z8® MCU.

The new Z8 Encore! XP microcontrollers combine a 20 MHz core with Flash memory, linear-register SRAM, and an extensive array of on-chip peripherals. These peripherals make the Z8 Encore! XP suitable for various applications including motor control, security systems, home appliances, personal electronic devices, and sensors.

Discussion

This section briefly discusses following topics associated with interrupts:

- Interrupts
- Prioritizing Interrupts
- Nesting Interrupts
- Interrupt Latency

Interrupts

An interrupt is an external/internal stimulus that informs a processor that an event has occurred. Interrupts can be hardware- or software-generated. The use of interrupts frees the processor from polling for events that require attention, and from service events as and when required in real time.

The processor's architecture provides a method to selectively enable or disable an interrupt source. When an interrupt source is identified, the interrupt is serviced by vectoring to a software routine designed to handle the interrupt. Program flow then returns to the original point of interruption.

Prioritizing Interrupts

A processor can manage a single interrupt at any time. However, it is possible that several interrupts can be simultaneously generated from several enabled interrupt sources. In such instance, setting an interrupt priority schema assumes importance. Some microcontrollers and microprocessors provide an option to set priorities for interrupts.

Copyright ©2007 by Zilog®, Inc. All rights reserved.
www.zilog.com

In MCU/MPUs that do not feature this option, the interrupts are handled either by the order of their occurrence or by their position in an interrupt vector priority table (see Table 2 in Appendix C—Z8 Encore! XP® Interrupt Controller Overview on page 15).

An interrupt controller is a hardware device that feeds interrupts to the microprocessor/controller based on a set priority level.

The Z8 Encore! XP® MCU features an on-chip interrupt controller (for an overview of Z8 Encore! XP interrupt controller, see Appendix C—Z8 Encore! XP® Interrupt Controller Overview on page 15).

Nesting Interrupts

An interrupt is serviced completely before the next interrupt is serviced. However, sometimes it is necessary to process a higher-priority interrupt that can occur while a lower priority interrupt is being serviced. The mechanism by which a higher-priority interrupt preempts a lower-priority interrupt is called *nesting*.

The handling of nested interrupts can be unpredictable and therefore leads to variable amount of delay prior to servicing a low-priority interrupt, or a higher program stack size requirement.

Interrupt Latency

Interrupt latency is the time interval measured from the instant an interrupt occurs until the corresponding ISR begins to execute. The worst-case latency for any given interrupt is a sum of the following:

- Time taken to finish program instructions in progress, save a current program context, and begin an ISR
- Longest time elapsed before an Enable Interrupt (`EI`) instruction is encountered
- Time taken to execute all higher-priority interrupts if they occur simultaneously

It follows that higher-priority interrupts exhibit much lower latencies. In simple cases, latency can be calculated from instruction cycle times. Interrupt latency must be considered at design time, along with nesting interrupts, whenever responsiveness matters.

Nesting Interrupts with Z8 Encore! XP MCUs

This section describes how to nest interrupts with Z8 Encore! XP MCU and also demonstrates nesting up to three levels.

When an interrupt occurs, the CPU performs certain activities by default. To nest interrupts, some additional activities need to be performed. Follow the steps below to nest the interrupts to perform the operations involved when an interrupt occurs:

1. By default, the CPU completes the instruction currently being executed.
2. By default, the CPU saves the address of the next instruction to be executed in the interrupted routine by pushing Low and High bytes of the Program Counter onto the stack. It also pushes the Flags Register on the stack and branches to the interrupt vector address pointed to by the instruction pointer.

Follow the steps below to nest the interrupts:

1. If Assembly code[1] is used then save the context of the Working Register Group
2. Save the priority for the set interrupts

1. The software implementation to nest interrupts in this application note is written in C; Assembly language is used as necessary.

3. Mask the interrupts that are at the same and lower priority levels
4. Enable interrupts
5. Execute the application-specific user code
6. Disable the interrupts
7. Restore the priority for the set interrupts

On completing the above tasks, the CPU can perform an operation when an Interrupt Return (IRET) instruction is encountered. By default, it restores the address of the next instruction in the routine that was interrupted, and enables the interrupts. Program execution resumes in the routine that was interrupted.

The steps you must perform for nesting interrupts are discussed in detail in the following sections.

Saving Context of Working Register Group

During normal program execution, the CPU uses the Working Register Group (WRG) to store intermediate data. The WRG is selected by loading the appropriate value to the register pointer (RP). When program execution jumps to an ISR, the context of the existing WRG must be stored and another WRG must be selected. Save the intermediate data by loading a new WRG address to the RP. When the execution of ISR completes, select the original WRG.

If the program is written in C language, then the compiler automatically takes care of switching between WRGs. Therefore, saving the context of the WRG must be done only when programming completely in Assembly language.

Saving Priority for Set Interrupts

Based on the application, only the priority registers of the required interrupt must be stored on the stack.

The function `_store_priority`, provided in this application note, illustrates saving the Interrupt Priority Control Registers on the stack.

This function is written in Assembly language and called from within a C routine. For one level of nesting depth, a stack size of six bytes is required to store the priorities set for all the interrupts, two bytes are required to store the Program Counter (PC) that holds the address of the next instruction to be executed in the interrupted routine, and one byte is required to store the Register Pointer (RP).

```
_store_priority:
POP R1; Store return address to Register
R1
POP R2; Store return address to Register
R2
PUSHX  IRQ0ENH; Push IRQ0ENH to stack
PUSHX  IRQ0ENL; Push IRQ0ENL to stack
PUSHX  IRQ1ENH; Push IRQ1ENH to stack
PUSHX  IRQ1ENL; Push IRQ1ENL to stack
PUSHX  IRQ2ENH; Push IRQ2ENH to stack
PUSHX  IRQ2ENL; Push IRQ2ENL to stack
PUSH R2; Push return address on the stack
PUSH R1; Push return address on the stack
RET;
```

In the above Assembly program, Program Counter (PC) that holds the address of the next instruction in the calling routine (that is on the stack) is popped onto registers R1 and R2. Next, all the Interrupt Priority Registers are pushed onto the stack. Next, the PC is placed back on top of the stack using the `PUSH R2` and `PUSH R1` instructions so that the stack pointer points to the valid address of the next instruction in the interrupted routine.

Masking Interrupts of the Same and Low Priority

Masking interrupts of the same and lower priority ensures that only higher-priority interrupts are serviced within the lower-priority tasks. The API `void mask_equal_low_priority(unsigned char vector_no)`, provided in this application note, performs this operation.

The interrupt source and the priority of the interrupts must be determined before masking. The priority for interrupts must be decided carefully based on application requirements.

An array of unsigned characters called `priority` is used to store the priority of all the selected interrupts. See Table 1 in Appendix C—Z8 Encore! XP® Interrupt Controller Overview on page 15 to disable selected interrupts.

Enabling Interrupts to Allow Nesting

The `EI` instruction is a macro that enables all the set interrupts by setting the Interrupt Request Bit in the Interrupt Control Register to 1 (see Appendix C—Z8 Encore! XP® Interrupt Controller Overview on page 15).

Executing Application-Specific User Code

The application-specific user code is inserted at this point. If there are any critical sections to be executed in the user code, the interrupt must be enabled after the critical section.

Disabling Interrupts

The interrupts are disabled at this point, using the `DI` macro to restore the contents of the Interrupt Priority Control Registers.

Restoring Set Priority for Interrupts

The function `_restore_priority_status`, provided in this application note, restores the priority of the set interrupts. This function is written in Assembly language and called from within a C routine.

```
_restore_priority_status:

POP R1; Store the Return address to
Register R1
POP R2; Store the Return address to
Register R2
POPX  IRQ2ENL; Pop IRQ2ENL from stack
POPX  IRQ2ENH; Pop IRQ2ENH from stack
POPX  IRQ1ENL; Pop IRQ1ENL from stack
POPX  IRQ1ENH; Pop IRQ1ENH form Stack
POPX  IRQ0ENH; Pop IRQ0ENH from stack
POPX  IRQ0ENH; Pop IRQ0ENH from stack
PUSH R2; Push the return address on the
stack
PUSH R1; Push the return address on the
stack
RET;
```

Finally, before returning from the ISR, the CPU restores the PC, and program execution resumes in the routine that was interrupted.

Figure 1 on page 5 displays the graphical representation of interrupt nesting up to 3 levels. In this figure, Task 3 holds the highest priority, Task 2 holds a medium priority, and Task 1 holds a low priority.

Hardware Implementation to Demonstrate Nesting of Interrupts

To demonstrate the nesting of interrupts, Z8 Encore! XP Development Board is used along with a number of LEDs and resistors.

Figure 2 on page 5 displays the block diagram for the hardware implementation using Z8F64xx MCU.

Figure 3 on page 6 displays the block diagram for the hardware implementation using Z8F0822 MCU.

For a complete schematic of the Z8 Encore! XP Development Board, refer to *Z8 Encore! XP® Flash Microcontroller Development Kit User Manual (UM0146)*.

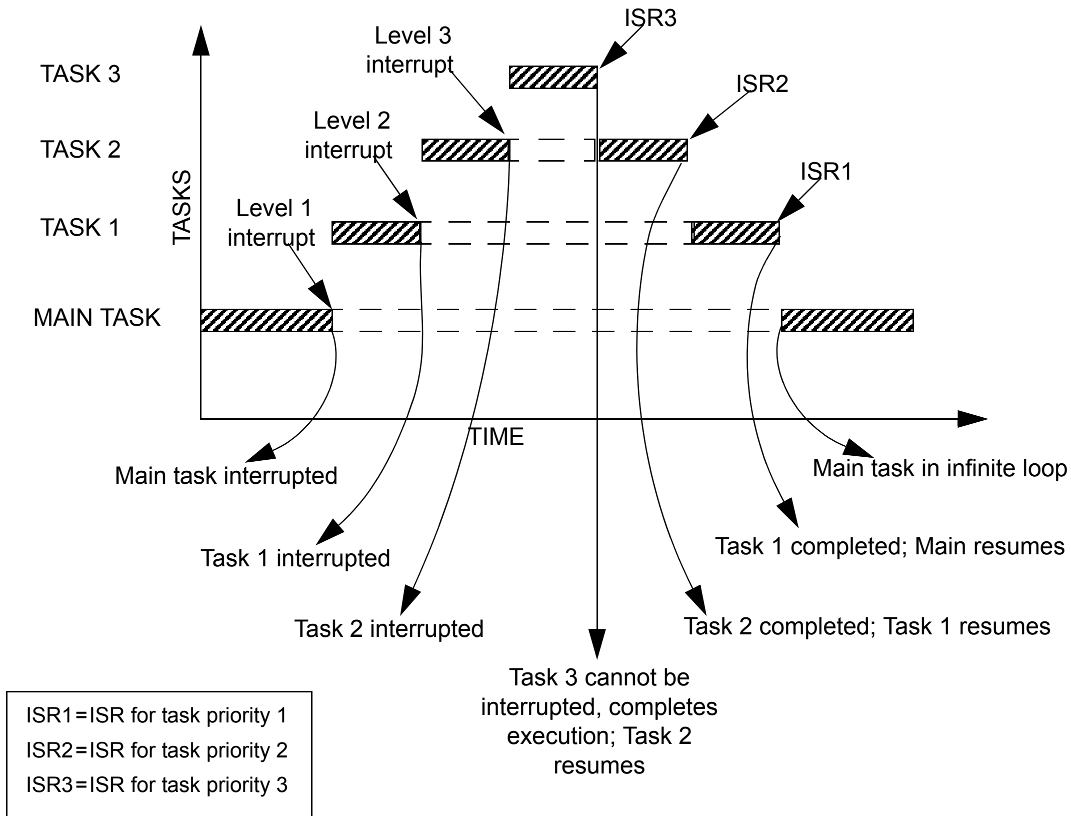

Figure 1. Graphical Representation of Nesting Interrupts up to Three Levels

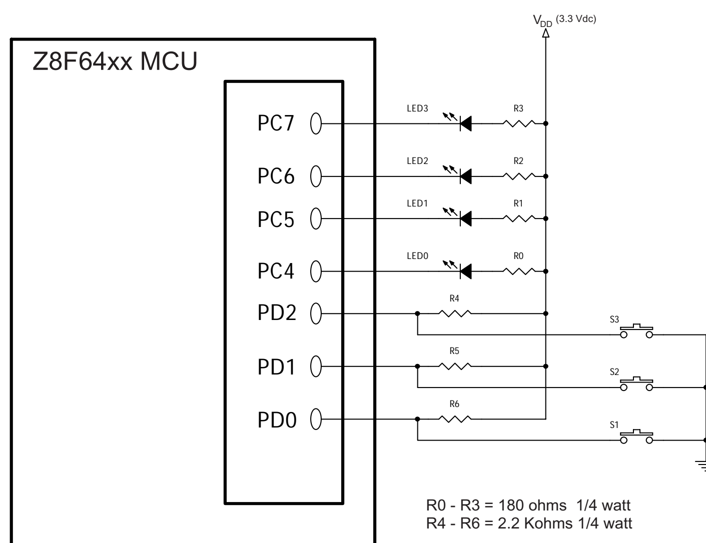

Figure 2. LED and Switch Connections to Z8F64xx MCU

Figure 2 on page 5 displays a block diagram for the Z8F64xx devices, in which the port pins PD0, PD1, and PD2 are configured as interrupt pins.

In Figure 3, which displays the Z8F0822 device, port pins PA4, PA6, and PA7 are configured as interrupt pins. In both setups, switches S1, S2, and S3 are used to generate interrupts.

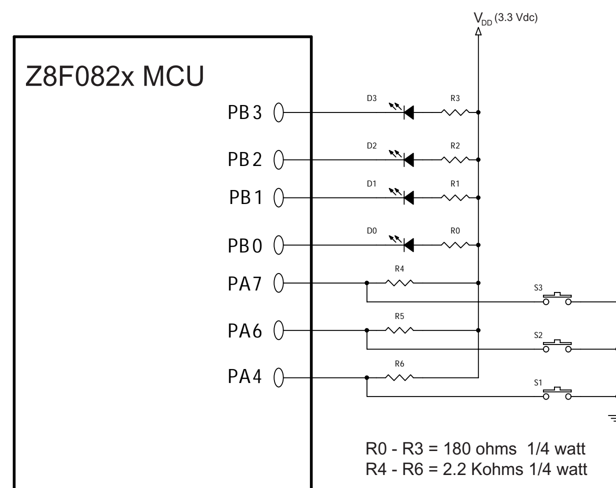

Figure 3. LED and Switch Connections to Z8F0822 MCU

Demonstration Setup

Three interrupt sources are simulated using switches to allow a nesting depth of three levels. Additionally, four externally-connected LEDs to Port C are used in the demonstration setup.

Equipment Used

The equipment used to demonstrate the nesting of interrupts are listed below:

- Z8 Encore! XP Development Kit (Z8ENCORE000ZC0-D) featuring Z8F640x MCU
- Z8 Encore! XP Development Kit (Z8F08200100) featuring Z8F082x MCU
- Z8 Encore! XP Development Kit (Z8F64200100KIT) featuring Z8F642x MCU
- ZDS II IDE for Z8F640x MCU v4.5.1
- ZDS II IDE for Z8F08xx MCU v4.5.1
- ZDS II IDE for Z8F642x MCU v4.5.1

Demonstration Procedure

The demonstration procedure includes configuring the ZDS II–Z8 Encore! IDE and then executing the Nesting Interrupts demonstration.

Configuring ZDS II

Follow the steps below to configure ZDS II:

1. Launch ZDS II[2]. Navigate to **Project Settings** dialog box.

2.Refer to `readme.txt` file included with the ZDS II–Z8 Encore! IDE for more details. This file is also available within the ZDS II–Z8 Encore! User Manual (UM0130).

2. Click the **Target** tab. Set **RData** to `40h-FFh` for three levels of nesting.

For the remaining project settings, refer to *ZDS II–Z8 Encore!® User Manual (UM0130).*

Executing Nesting Interrupts Demonstration

Follow the steps below to nest the interrupts:

1. The Z8 Encore! XP Target Board is labelled as either a Z8F64xx MCU-based board, a Z8F0822 MCU-based board, or a Z8F642x MCU-based board. Download the appropriate source code to the appropriate target board.
2. Execute the program. LED0 starts blinking and all other LEDs are switched ON.
3. While LED0 continues to blink, simulate an interrupt of Priority Level 1 by pressing switch `S1` (see Figure 4 on page 8). Observe that the LED0 stops blinking and LED1 starts blinking, indicating that the program is executing the ISR for Priority Level 1, ISR1.
4. While LED1 blinks, simulate a Level 2 interrupt by pressing switch `S2`. LED1 stops blinking and LED2 starts blinking, indicating that the program has moved from ISR1 to the ISR for Priority Level 2, ISR2.
5. While LED2 blinks, simulate a Level 3 interrupt by pressing switch S3. LED2 stops blinking and LED3 starts blinking, indicating that the program has moved from ISR2 to the ISR for Priority Level 3, ISR3.

After a brief delay, LED3 stops blinking, indicating that ISR3 has completed executing and the program has moved to ISR2, which causes LED2 to resume blinking. After a brief delay, LED2 stops blinking, indicating that ISR2 has completed execution. The program moves to ISR1, and LED1 resumes blinking. LED1 stops blinking after a brief delay, and LED0 starts blinking, indicating that the program has moved back to `main()`.

Figure 4 on page 8 graphically displays this demonstration.

> **Note:** *The C Compiler in ZDS II automatically generates a Disable Interrupt instruction at the beginning of an ISR and generates an Enable Interrupt instruction when exiting an ISR.*

> **Caution:** *Ensure to avoid stack overflow while nesting interrupts.*

Summary

This application note demonstrates the nesting of interrupts on Z8 Encore! XP MCU up to a depth of three levels. The stack requirement for nesting interrupts up to three levels is only 27 bytes. Nesting up to four levels is also possible when any nonmaskable interrupt preempts a Priority Level 3 interrupt. When nesting up to four levels, the stack requirement is 36 bytes.

As Z8 Encore! XP MCU operates at 20 MHz and most of the instructions are one or two cycles, its architecture is ideally suited for real-time applications requiring fast response time. While nesting interrupts, keeping the number of instructions within the ISR to a minimum reduces interrupt latency. The source code contains ISRs for all the 24 vectors. You must insert application-specific user code in the area specified within these ISRs.

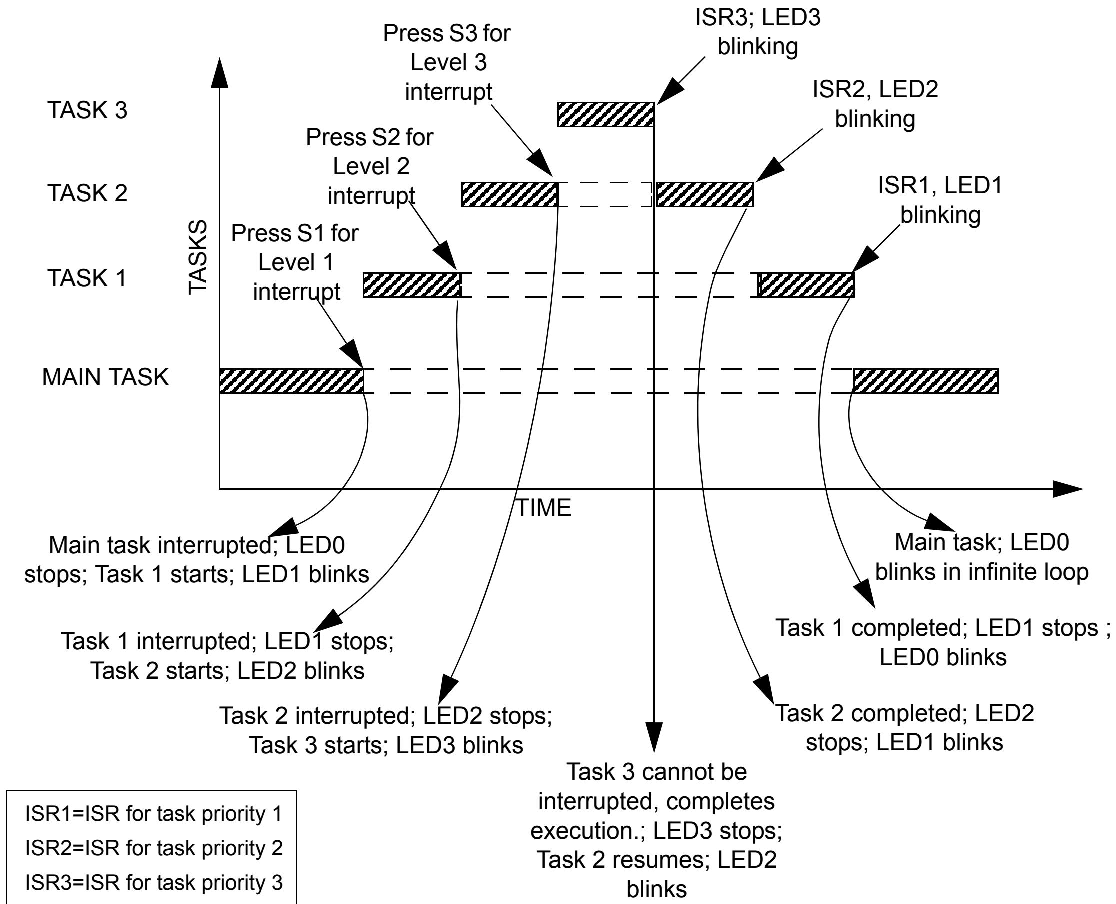

Figure 4. Graphical Representation of Nesting Interrupts Demo

References

The documents associated with Z8 Encore! XP Flash Microcontrollers, eZ8 CPU, and ZDS II Development Tool available at www.zilog.com are provided below:

- Z8 Encore! XP® 64K Series Flash Microcontrollers Product Specification (PS0199)
- Z8 Encore!® 8K and 4K Series Product Specification (PS0243)
- Z8 Encore! XP® Flash Microcontroller Development Kit User Manual (UM0146)
- eZ8 CPU User Manual (UM0128)
- Zilog Developer Studio II–Z8 Encore!® User Manual (UM0130)

Appendix A—Flowcharts

Figure 5 displays the initialization routine for nested interrupts.

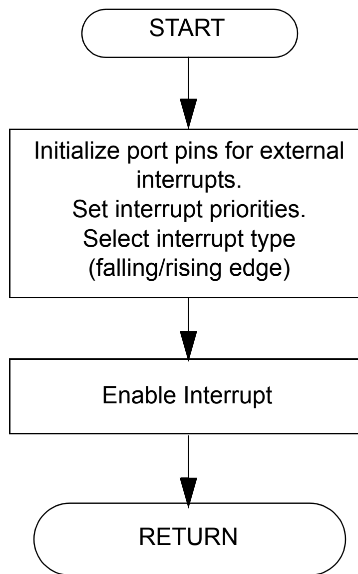

Figure 5. Initialization Routine with Interrupt Nesting

Figure 6 displays the Interrupt Service Routine (ISR) for nested interrupts.

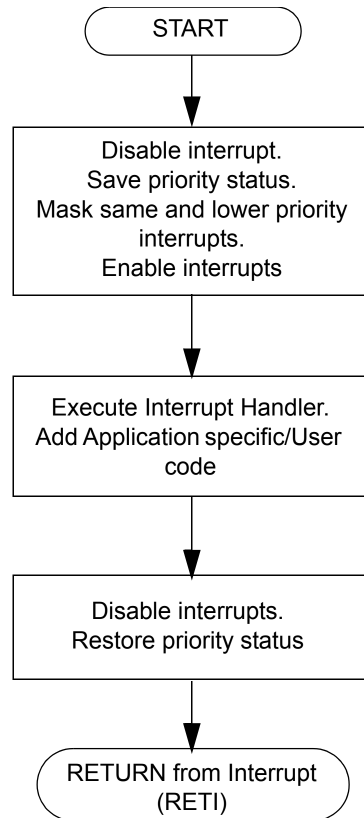

Figure 6. ISR Routine with Interrupt Nesting

Figure 7 displays the ISR without nested interrupts.

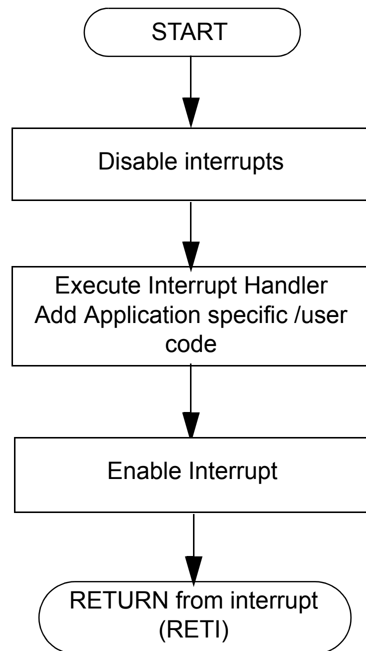

Figure 7. ISR Routine without Interrupt Nesting

Appendix B—Support Functions

This appendix describes two support functions for nesting interrupts using Z8 Encore! XP MCU.

set_priority (VECTOR, level)

Description

This routine sets the required interrupt `VECTOR` to the user-defined priority level. Three levels of priorities can be set with Z8 Encore! XP MCU. For more details, see Interrupts on page 1.

Argument(s)

`VECTOR`	The interrupt vector to be assigned a priority.
`level`	Priority to be set for interrupt.

Example

`set_priority (TIMER1, 03)`–In this example, this API sets the priority of the TIMER1 interrupt to Priority Level 3, which is the highest priority.

In Demonstration Setup on page 6, the GPIO pins D0, D1, and D2 are used for demonstrating the nested interrupts in Z8F6403 device. D0 is set to Priority Level 1 (lowest priority), D2 is set to Level 2 (medium priority), and D3 is set to Level 3 (highest priority).

`set_priority (P0AD, 1)`–In this example, this API sets the priority of PORTD pin 0 to priority level 1, which is the lowest priority.

SET_VECTOR (VECTOR, interrupt handler)

Description

Interrupts are set in C using the `SET_VECTOR` macro specific to the Z8 Encore! XP MCU, which embeds assembly language to C source code to specify the address of an interrupt handler.

The interrupt vectors of the Z8 Encore! XP MCU exist in Flash memory and cannot be initialized at run time. The `SET_VECTOR` macro places the address of the interrupt handler at the proper `VECTOR` address.

Argument(s)

`VECTOR`	Interrupt vector
`interrupt handler`	ISR routine to be used

Example

The following code illustrates the use of `SET_VECTOR` macro:

```
SET_VECTOR (P0AD, isr_PORTD0);
SET_VECTOR (P1AD, isr_PORTD1);
SET_VECTOR (P2AD, isr_PORTD2);
```

In the above example, the interrupt vectors are `P0AD`, `P1AD`, and `P2AD`. The interrupt handlers for these vector interrupts are `isr_PORTD0`, `isr_PORTD1`, and `isr_PORTD2`, respectively.

The address of the interrupt handler `isr_PORTD0` is written to Flash memory location `0026h`, which is the vector address for the `POAD` interrupt.

The address of the interrupt handler `isr_PORTD1` is written to Flash memory location `0024h`, which is the vector address for the `P1AD` interrupt.

The address of the interrupt handler `isr_PORTD2` is written to Flash memory location `0022h`, which is the vector address for the `P2AD` interrupt.

The `SET_VECTOR` macro can be added anywhere in the C program and does not generate any code. It can be placed within any function of the user program where the interrupt handler function is in scope. However, for clarity, it is preferred to place the `SET_VECTOR` macro within the interrupt handler itself.

The following code segment provides an example:

```
#pragma interrupt
void isr_TIMER1 (void)
{
  SET_VECTOR (TIMER1, isr_TIMER1);  // Set up the vector
                                    // for the TIMER1
                                    // interrupt;
                                    // enter user code
}
```

During compile time, the compiler resolves the address of function `isr_TIMER1` and maps it to the `TIMER1` vector address space.

Appendix C—Z8 Encore! XP® Interrupt Controller Overview

The Z8 Encore! XP interrupt controller includes the following features:

- 24 unique interrupt vectors:
  - 12 GPIO port pin interrupt sources
  - 12 on-chip peripheral interrupt sources
- Flexible GPIO interrupts:
  - 8 selectable rising- and falling-edge GPIO interrupts
  - 4 dual-edge interrupts
- 3 levels of individually-programmable interrupt priority
- A Watchdog Timer that can be configured to generate an interrupt

Interrupt requests (IRQs) allow peripheral devices to request a CPU operation in an orderly manner to force the CPU to start an Interrupt Service Routine (ISR) ahead of its current activity.

Usually, an ISR is involved with the exchange of data, status information, or control information between the CPU and the interrupting peripheral. When the ISR is completed, the CPU returns to the operation from which it was interrupted.

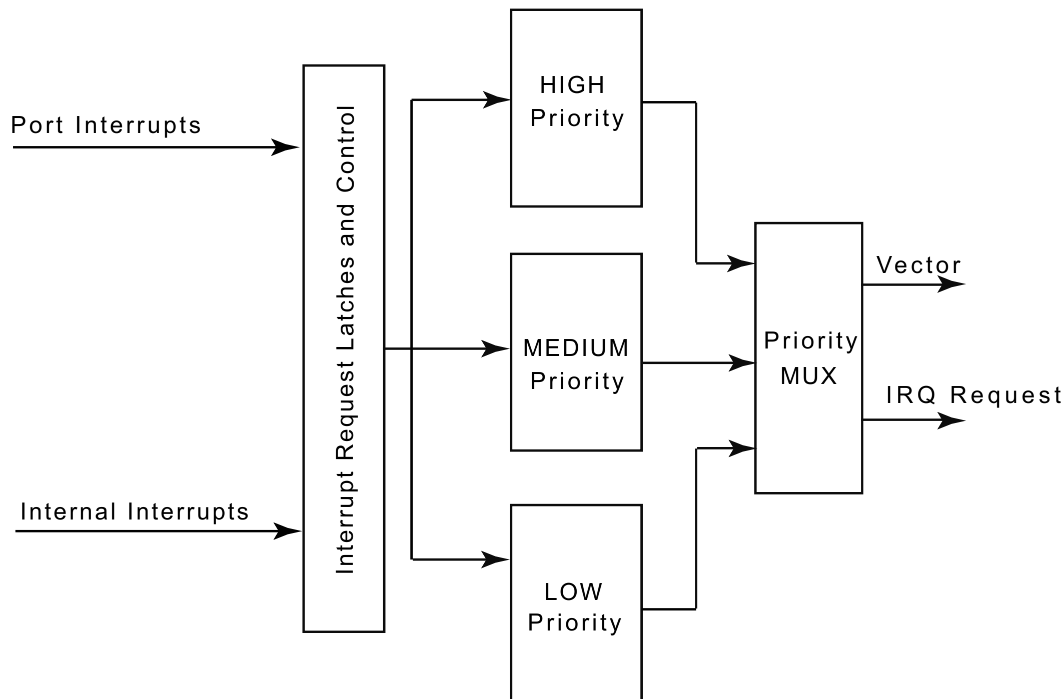

Figure 8. Interrupt Controller Block Diagram

The eZ8 CPU supports both vectored and polled interrupt handling. For polled interrupts, the interrupt control has no effect on the operation.

For more information regarding interrupt servicing by the eZ8 CPU, refer to *eZ8 CPU User Manual (UM0128)*. Figure 8 displays a block diagram of the Z8 Encore! XP interrupt controller.

Interrupt Vectors and Priority

The Z8 Encore! XP interrupt controller supports three levels of interrupt priority. Level 3 holds the highest priority and Level 1 holds the lowest priority. If all the interrupts are enabled with identical interrupt priority (all as Level 2 interrupts, for example), then priority is assigned from highest to lowest, as specified in Table 2 on page 17. The Reset, Illegal Instruction Trap, and Watchdog Timer (if enabled) interrupts always hold Level 3 priority.

Interrupt Control Registers

For all interrupts other than the Watchdog Timer interrupt, the interrupt control registers enable individual interrupts, set interrupt priorities, and indicate interrupt requests.

Interrupt Request Registers

There are three Interrupt Request Registers, which include:

- Interrupt Request 0 Register (IRQ0)
- Interrupt Request 1 Register (IRQ1)
- Interrupt Request 2 Register (IRQ3)

These registers hold a pending interrupt request. When an interrupt occurs, the relevant bit in the corresponding register is set to 1. This bit is reset to 0 when the CPU acknowledges the interrupt controller after executing the first instruction in the ISR that corresponds to the interrupt.

Interrupt Enable High and Low Bit Registers.

There are six Interrupt Enable High and Low Bit Registers, which include:

- IRQ0 Enable High Bit Register (IRQ0ENH)
- IRQ0 Enable Low Bit Register (IRQ0ENL)
- IRQ1 Enable High Bit Register (IRQ0ENH)
- IRQ1 Enable Low Bit Register (IRQ0ENL)
- IRQ2 Enable High Bit Register (IRQ0ENH)
- IRQ2 Enable Low Bit Register (IRQ0ENL)

These registers form a priority-encoded enabling for interrupts in the corresponding Interrupt Request Registers. Setting the bits in the Interrupt Enable High and Low bit registers generates priority. Table 1 describes the priority control for IRQ0, where *x* indicates the register bits 0 through 7.

Table 1. IRQ0 Enable and Priority Encoding

IRQ0ENH [*x*]	IRQ0ENL [*x*]	Priority	Description
0	0	Disabled	Disabled
0	1	Level 1	Low
1	0	Level 2	Nominal
1	1	Level 3	High

For further details on setting interrupt priority, enabling interrupts, and other interrupt control registers, refer to *Z8 Encore! XP® 64K Series Flash Microcontrollers Product Specification (PS0199)*.

Interrupt Vector Listing

Table 2 lists all the interrupts supported on the Z8 Encore! XP in the order of their priority. The interrupt vector is stored with the most significant byte (MSB) at the even Program Memory address and the least significant byte (LSB) at the following odd Program Memory address.

Table 2. Z8 Encore! XP Interrupt Vectors in Order of Priority

Priority	Program Memory Vector Address	Interrupt Source	Interrupt Assertion Type
Highest	0002h	Reset (not an interrupt)	Not applicable
	0004h	Watchdog Timer	Continuous assertion
	0006h	Illegal Instruction Trap (not an interrupt)	Not applicable
	0008h	Timer 2	Single assertion (pulse)
	000Ah	Timer 1	Single assertion (pulse)
	000Ch	Timer 0	Single assertion (pulse)
	000Eh	UART 0 receiver	Continuous assertion
	0010h	UART 0 transmitter	Continuous assertion
	0012h	I^2C	Continuous assertion
	0014h	SPI	Continuous assertion
	0016h	ADC	Single assertion (pulse)
	0018h	Port A7 or Port D7, rising or falling input edge	Single assertion (pulse)
	001Ah	Port A6 or Port D6, rising or falling input edge	Single assertion (pulse)
	001Ch	Port A5 or Port D5, rising or falling input edge	Single assertion (pulse)
	001Eh	Port A4 or Port D4, rising or falling input edge	Single assertion (pulse)
	0020h	Port A3 or Port D3, rising or falling input edge	Single assertion (pulse)
	0022h	Port A2 or Port D2, rising or falling input edge	Single assertion (pulse)
	0024h	Port A1 or Port D1, rising or falling input edge	Single assertion (pulse)
	0026h	Port A0 or Port D0, rising or falling input edge	Single assertion (pulse)
	0028h	Timer 3 (not available in 40-/44-pin packages)	Single assertion (pulse)
	002Ah	UART 1 receiver	Continuous assertion
	002Ch	UART 1 transmitter	Continuous assertion
	002Eh	DMA	Single assertion (pulse)
	0030h	Port C3, both input edges	Single assertion (pulse)

Table 2. Z8 Encore! XP Interrupt Vectors in Order of Priority (Continued)

Priority	Program Memory Vector Address	Interrupt Source	Interrupt Assertion Type
	0032h	Port C2, both input edges	Single assertion (pulse)
	0034h	Port C1, both input edges	Single assertion (pulse)
Lowest	0036h	Port C0, both input edges	Single assertion (pulse)

Warning: DO NOT USE IN LIFE SUPPORT

LIFE SUPPORT POLICY

ZILOG'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS PRIOR WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF ZILOG CORPORATION.

As used herein

Life support devices or systems are devices which (a) are intended for surgical implant into the body, or (b) support or sustain life and whose failure to perform when properly used in accordance with instructions for use provided in the labeling can be reasonably expected to result in a significant injury to the user. A critical component is any component in a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system or to affect its safety or effectiveness.

Document Disclaimer

©2007 by Zilog, Inc. All rights reserved. Information in this publication concerning the devices, applications, or technology described is intended to suggest possible uses and may be superseded. ZILOG, INC. DOES NOT ASSUME LIABILITY FOR OR PROVIDE A REPRESENTATION OF ACCURACY OF THE INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED IN THIS DOCUMENT. ZILOG ALSO DOES NOT ASSUME LIABILITY FOR INTELLECTUAL PROPERTY INFRINGEMENT RELATED IN ANY MANNER TO USE OF INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED HEREIN OR OTHERWISE. The information contained within this document has been verified according to the general principles of electrical and mechanical engineering.

Z8, Z8 Encore!, and Z8 Encore! XP are registered trademarks of Zilog, Inc. All other product or service names are the property of their respective owners.