# Application Note

# Nesting Interrupts in eZ80F91 MCU

AN017604-0608

## Abstract

This application note discusses how to nest interrupts using Zilog's eZ80F91 Flash microcontroller unit (MCU). Nesting levels are controlled by defining the required nesting depth and is dependent on the available stack size. In the eZ80F91 MCU all interrupts are considered to be at the same priority level and can preempt one another.

**Note:** *The source code file associated with this Application Note,* `AN0176-SC01.zip`*, is available for download at www.zilog.com.*

## eZ80Acclaim*Plus!*™ Flash MCU Overview

Zilog's eZ80Acclaim*Plus!*™ on-chip Flash MCUs are of an exceptional value when designing high performance, 8-bit MCU-based systems. With speeds up to 50 MHz and on-chip Ethernet MAC (eZ80F91 only), you have required performance to execute complex applications quickly and efficiently. Combining Flash and SRAM, eZ80Acclaim*Plus!* devices provide the memory required to implement communication protocol stacks and achieve flexibility when performing in-system updates of application firmware.

The eZ80Acclaim*Plus!* Flash MCU can operate in full 24-bit linear mode addressing 16 MB without a Memory Management Unit. Additionally, support for the Z80®compatible mode allows execution of Z80/Z180 legacy code within multiple 64 KB memory blocks with minimum modifications. With an external bus supporting eZ80®, Z80, Intel, Motorola bus modes, and a rich set of serial communications peripherals, there are several options when interfacing to external devices. Some of the many applications suitable for eZ80Acclaim*Plus!* devices include vending machines, point-of-sale terminals, security systems, automation, communications, industrial control, facility monitoring, and remote control.

## Discussion

This section briefly discusses what interrupts are, nesting of interrupts, and 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 the burden of polling for events that require its attention. It also allows it to 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.

For an overview of the eZ80F91 interrupts, see Appendix B—eZ80F91 MCU Interrupts on page 10.

### Nesting Interrupts

Typically, an interrupt is serviced completely before servicing the next interrupt. However, sometimes it is necessary to process an interrupt that occurs while another interrupt is being serviced. The mechanism by which one interrupt preempts another is called *nesting*.

The handling of nested interrupts can be unpredictable. Other issues can arise, such as a variable amount of delay prior to servicing a low-priority interrupt, or a higher program stack size requirement.

## Interrupt Latency

Interrupt latency is the interval of time measured from the instant an interrupt occurs, until the corresponding interrupt service routine (ISR) begins to execute. The worst-case latency for any given interrupt is a sum of a number of items, such as:

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

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 using the eZ80F91 MCU

This section explains how to nest interrupts when using the eZ80F91 MCU, followed by a demonstration that illustrates nesting up to three levels.

**Note:** *Nesting levels in the eZ80F91 MCU are dependent on the available stack space.*

## Software for Nesting Interrupts

Nesting of interrupts involves initializing the interrupts and writing the interrupt handlers. The following sections present the software for nesting interrupts with a specific instance of nesting up to three levels. (See AN0176-SC01.zip file that contains the entire demo code, available for download at www.zilog.com).

### main.c

In the main.c file appropriate functions are called to initialize ports, interrupt handlers, and so on.

The port_init() function initializes the Port D pins 4, 5, 6, and 7 as general output pins and Port B pins 0, 1, and 2 as falling edge detectable interrupt pins.

The init_default_vectors() initializes all the interrupt vector locations to a default handler that contains RETI instructions. The handler routine prevents any system failure when spurious interrupts are generated.

The init_handler() function calls the following functions: init_int_base and set_vector(). The init_int_base routine is implemented in Assembly (see isr.asm file, on page 3).

```
_init_int_base:
  im    2       Interrupt mode 2
  ld    hl,__vector_table  >>  8  &
  0fffeh
  ld    i,hl ; Load interrupt
  vector base
ret
```

The init_int_base routine loads the **I Register** with the relocated interrupt vector table address.

The set_vector function maps the address of the ISR to the required vector address. The set_vector() prototype is:

```
set_vector(PORT_B0, port_B0_ISR);
```

This routine maps the address of the ISR (port_B0_ISR) to the absolute address of PORT_B0 in the interrupt vector table. For example, if the content of the **I Register** is 0400H, the address of the ISR is written at location 0400A0H, where A0H is the absolute vector address for the PORT_B0 interrupt. When an interrupt occurs on PB0, the pro-

gram jumps to the location 0400A0H to execute the ISR.

### test.c

The interrupt_reset_PB0() and the user_routine() functions are defined in the test.c file. The user_routine() function is the actual ISR to be executed when an interrupt occurs.

### isr.asm

The isr.asm file is an assembly file containing the ISR for the interrupts that occur at Port B pins 0, 1, and 2. All the ISRs, including the ISR_Prolog() and ISR_Epilog() functions are written in the isr.asm file. The ISR_Prolog() and ISR_Epilog() functions can be used. In addition, the ISRs call the user_routine() that is defined in the test.c file.

## How to Nest Interrupts

When an interrupt occurs, the CPU performs certain activities by default. To nest interrupts, you must perform additional activities.

The operations involved when an interrupt occurs, including the steps that you must introduce to nest interrupts, are explained below:

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.

Follow the steps below to nest interrupts:

1. Reset the interrupt request.
2. Save the context of the Working Registers.
3. Call ISR_Prolog() routine.
4. Execute the application-specific user code.
5. Call ISR_Epilog() routine.
6. Restore the context from the Working Registers.
7. Enable interrupts.
8. Execute a RETI (return interrupt) instruction to return from the ISR.

The CPU performs the next operation only when an RETI instruction is encountered. By default, the CPU 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.

A sample code snippet is provided to explain the details of the steps involved in nesting interrupts when using the eZ80F91 MCU.

**Table 1. ISR Code Snippet**

```
_port_B0_ISR:
        CALL    _interrupt_
                reset_PB0
        PUSH    AF
        PUSH    BC
        PUSH    DE
        PUSH    HL
        PUSH    IX
        PUSH    IY
        CALL    ISR_Prolog
        CALL    _user_routi ;Interrupt
                ne0         specific
                            code
        CALL    ISR_Epilog
        POP     IY
        POP     IX
        POP     HL
        POP     DE
        POP     BC
        POP     AF
        EI
        RETI
```

The ISR Code Snippet is taken from the `isr.asm` file. The ISR presented above is to service an interrupt occurring on Port B pin 0. The `ISR_Prolog()` and `ISR_Epilog()` functions may be used without modification. The `isr.asm` file is available in the `AN0176-SC01.zip` file, available for download at www.zilog.com.

You must provide the `_user_routine0()` function and write the `interrupt_reset_PB0()` function according to the port pin at which interrupts are generated in the user application. The `interrupt_reset_PB0()` function in the `test.c` file (available in `AN0176-SC01.zip` file) is used as a reference to write the user-specific function.

You need to write the ISRs for interrupts generated on the General-Purpose Input/Output (GPIO) ports according to the template in the ISR Code Snippet.

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

## Resetting the Interrupt Request

As long as an interrupt request is valid, it dispatches the same interrupt to the CPU repeatedly. It is therefore necessary to clear the interrupt and the interrupt request after an interrupt occurs.

When a port pin (for example, Port B pin 0) generates an interrupt, the program jumps to the corresponding ISR. The first line in the ISR code is `CALL _interrupt_reset_PB0` (see Table 1 on page 3). The C function `interrupt_reset_PB0()` resets the interrupt request for a particular source (PB0, in this example).

Writing a 1 to the port pin clears the interrupt. However, the method to clear the interrupt request from different sources (for example, I2C, UART, SPI) are explained in the *eZ80F91 MCU Product Specification (PS0192)*, available for download at www.zilog.com.

## Saving the Context of the Working Registers

The context of the Working Registers must be saved. The Working Registers for eZ80F91 MCU are `AF`, `BC`, `DE`, `HL`, `IX`, and `IY`. During normal program execution, the CPU uses these registers to store intermediate data. When program execution jumps to an ISR, the context of these Registers must be PUSHed on the stack, and when the interrupt is completely serviced, the contents of the Working Registers are POPed back. For more details, see Table 1 on page 3.

## Calling the ISR_Prolog

Calling the `ISR_Prolog()` routine causes interrupts to be enabled or disabled. The routine `ISR_Prolog()` keeps a count of the current nesting levels for the stack. A variable, `max_level`, stores the count of the nested interrupt levels. When the `ISR_Prolog()` routine is invoked it increments the `max_level` count by one, and compares the value obtained with the actual defined value in the `#define NESTED_ISR_COUNT`.

When the values are equal, interrupts are disabled because the maximum defined nesting level is reached. The interrupts are enabled when the maximum defined value is not equal to the incremented `max_level` value.

**Note:** *Ensure that the* `ISR_Prolog()` *function is called only after critical sections of the user code are executed.*

## Executing the User Code

The user code is an interrupt specific code that is called within the ISR. For the purpose of this nesting interrupts demo, the user code `(user_routine())` is located in the isr.asm file.

### Calling the ISR_Epilog

Calling the `ISR_Epilog()` routine decrements the `max_level` variable's count by one, so that predefined nesting depth is maintained.

### Restoring the Context of the Working Registers

When the ISR executes completely, the context is restored back by POPing data from the stack. The previous values of the Working Registers are loaded back from the stack.

### Executing an EI and RETI Instruction

`EI` instruction enables the interrupt and the `RETI` instruction causes the program to exit from the interrupt routine. The CPU restores the address of the next instruction to be executed in the interrupted routine from the stack, and program execution resumes in the routine that was interrupted.

## Hardware Setup to Demonstrate Nesting of Interrupts

To demonstrate the nesting of interrupts, the eZ80F91 development board is used along with LEDs and resistors. Figure 1 displays the block diagram for the hardware implementation using the eZ80F91 MCU. Three interrupt sources are simulated using switches (on the Development Board) to allow a nesting depth of three levels. Four externally-connected LEDs to Port D are used in the demonstration setup.

For a complete schematic of the eZ80F91 Development Board, refer to *eZ80F91 Development Kit User Manual (UM0142)*.

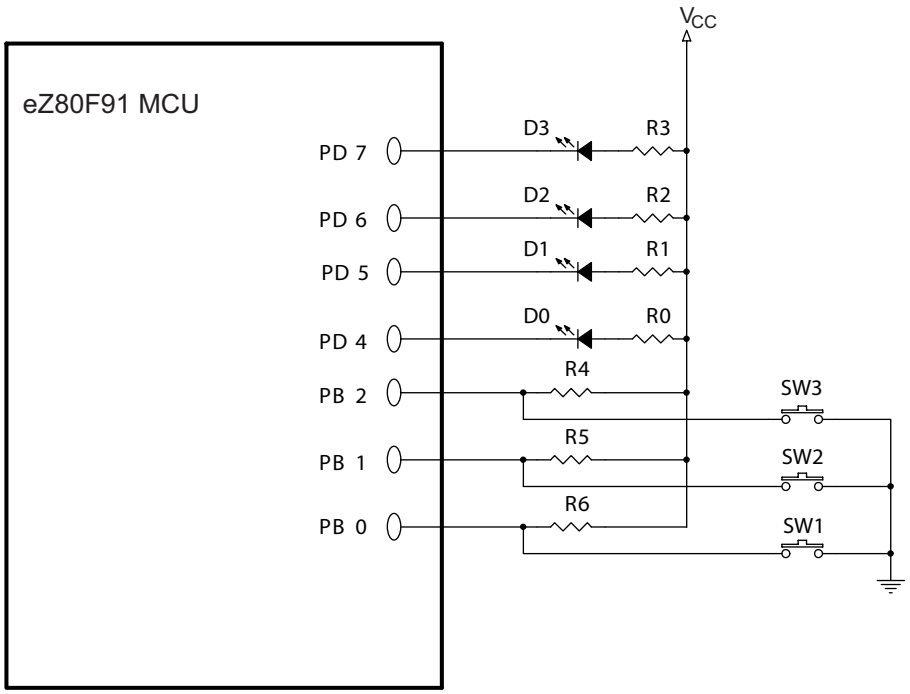


**Figure 1. LED and Switch Connections to the eZ80F91 MCU**

Figure 1 on page 5 displays a block diagram for the eZ80F91 devices, in which the following points must be noted:

- The port pins PB0, PB1, and PB2 are configured as interrupt pins. They simulate an interrupt event by pulling the Port B pins Low to generate the falling edge that is recognized as an interrupt.
- Port pins PD0, PD1, PD2, and PD3 correspond to the four LEDs (LEDs 0 to 3) on the eZ80F91 Development Board. The four Port D pins are configured as output pins.

**Caution:** *A debouncing circuit must be added to the switches to prevent bouncing that may generate multiple interrupts. The debouncing circuit is not displayed in Figure 1 on page 5.*

### Equipment Used

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

- eZ80F91 Development Kit (eZ80F910200ZCO) featuring the eZ80F91 MCU
- ZDS II IDE for eZ80Acclaim MCU, v4.11

## Demonstration Procedure

Follow the steps provided below to execute the nesting interrupts demo.

1. Download the Nesting Interrupts Demo source code (contained in the `AN0176-SC01.zip` file) to the eZ80F91 Development Board, using ZDS II IDE.
2. Execute the program. LED0 starts blinking and all other LEDs are switched on.
3. While LED0 continues to blink, simulate an interrupt by pressing switch SW1 (see Figure 2 on page 7). Observe that the LED0 stops blinking and LED1 starts blinking, indicating that the program is executing ISR1.
4. While LED1 blinks, simulate a Level 2 interrupt by pressing switch SW2. LED1 stops blinking and LED2 starts blinking, indicating that the program has jumped from ISR1 to ISR2.
5. While LED2 blinks, simulate a Level 3 interrupt by pressing switch SW3. LED2 stops blinking and LED3 starts blinking, indicating that the program has jumped from ISR2 to ISR3.

After a brief delay, LED3 stops blinking, indicating that ISR3 has completed executing and the program has jumped to ISR2, which causes LED2 to resume blinking. After a brief delay, LED2 stops blinking, indicating that ISR2 has completed execution. The program jumps to ISR1, and LED1 resumes blinking. LED1 stops blinking after a brief delay, and LED0 starts blinking, indicating that the program has jumped back to `main()`. Figure 2 on page 7 graphically displays this demonstration.

**Caution:** *Care must be taken to avoid stack overflow while nesting interrupts.*

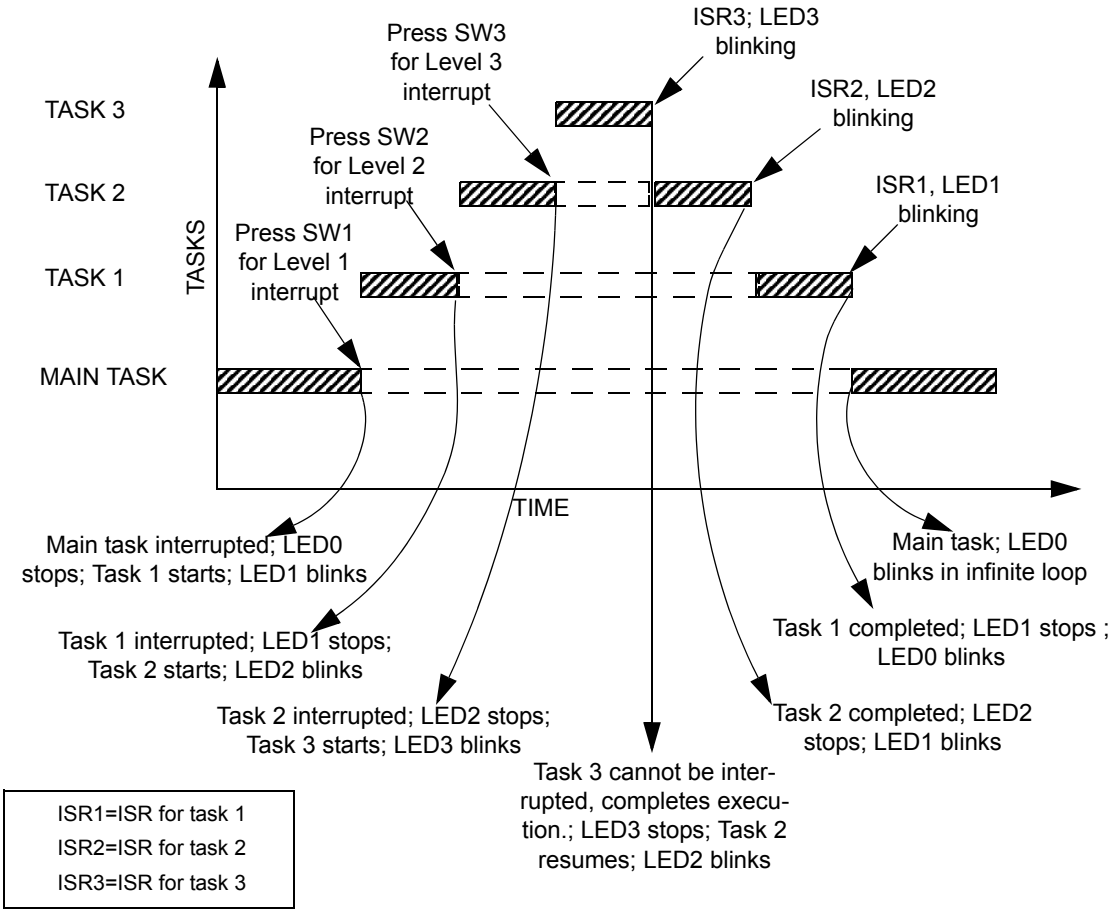


**Figure 2. Graphical Representation of the Nesting Interrupts Demo**

## Conclusion

This application note demonstrates the nesting of interrupts on a eZ80F91 MCU up to a depth of three levels. A variable defined in the ISR file `isr.asm` is used to control the level of nesting. Nesting depth is dependent only on stack size. The stack requirement for nesting interrupts up to three levels is only 27 bytes.

The eZ80F91 MCU operates at 50 MHz, and most of the instructions are one or two cycles. Thus, its architecture is ideally suited for real-time applications requiring a very fast response time. While nesting interrupts, keeping the number of instructions within the ISR to a minimum, reduces interrupt latency.

## References

The documents associated with eZ80®, eZ80Acclaim!®, and eZ80Acclaim*Plus!*™ family of products are listed below:

- eZ80F91 MCU Flash Microcontroller, Product Specification (PS0192)
- eZ80F91 Development Kit User Manual (UM0142)
- eZ80 CPU User Manual (UM0077)
- Zilog Developer Studio II (ZDS II)—eZ80Acclaim! User Manual (UM0144)

# Appendix A—Flowcharts

Flowchart for the Main Function of Nesting Interrupts (eZ80F91) is displayed in Figure 3.

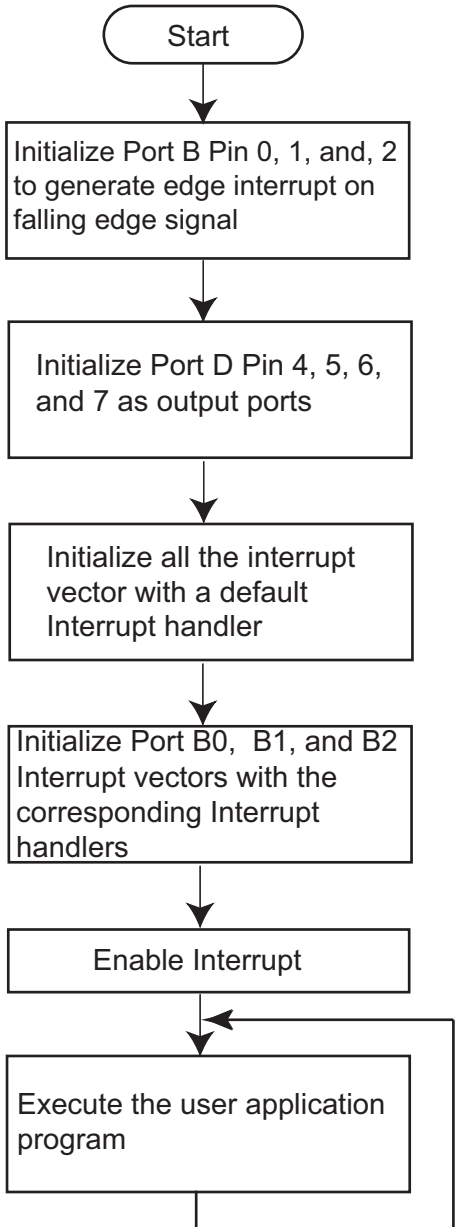


**Figure 3. Flowchart for the Main Function**

The Interrupt Service Routine for Nesting Interrupts (eZ80F91) is displayed in Figure 4.

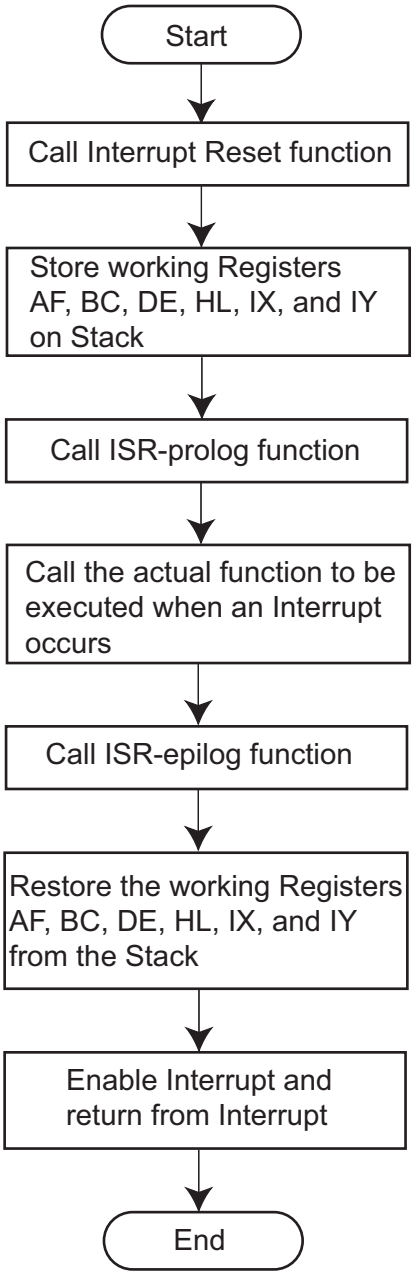


**Figure 4. Flowchart for Interrupt Service Routine**

# Appendix B—eZ80F91 MCU Interrupts

The eZ80F91 device features 32 GPIO pins. The GPIO pins are assembled as four 8-bit ports— Port A, Port B, Port C, and Port D. All port signals can be configured for use as either inputs or outputs. In addition, all the port pins can be used as vectored interrupt sources for the CPU.

The eZ80F91 MCU's GPIO ports are slightly modified from its eZ80® predecessors. Specifically, Port A pins now can source 8 mA and sink 10 mA. In addition, the Port B and Port C inputs now feature Schmitt-trigger input buffers.

## GPIO Operation

GPIO operation is same for all the four GPIO ports (Ports A, B, C, and D). Each port features eight GPIO port pins. The operating mode for each pin is controlled by four bits that are divided between four 8-bit registers. These GPIO mode control registers are:

- Port *x* Data Register (Px_DR)
- Port *x* Data Direction Register (Px_DDR)
- Port *x* Alternate Register 1 (Px_ALT1)
- Port *x* Alternate Register 2 (Px_ALT2)

where, *x* can be *A, B*, *C*, or *D* representing any of the four GPIO ports A, B, C, or D. The mode for each pin is controlled by setting each register bit pertinent to the pin to be configured. For example, the operating mode for Port B Pin 7 (PB7) is set by the values contained in PB_DR[7], PB_DDR[7], PB_ALT1[7], and PB_ALT2[7].

## GPIO Port Interrupts

All interrupts are latched. In effect, an interrupt is held even if the interrupt occurs while another interrupt is being serviced and interrupts are disabled; or if the interrupt is of a lower priority. However, before the latched ISR completes its task or reenables interrupts, the ISR must clear the interrupt. For on-chip peripherals, the interrupt is cleared when the data register is accessed. For GPIO-level interrupts, the interrupt signal must be removed before the ISR completes its task. For GPIO-edge interrupts (single and dual), the interrupt is cleared by writing a 1 to the corresponding bit position in the data register.

## Edge-Triggered Interrupts

When the port is configured for edge-triggered interrupts, the corresponding port pin is tristated. If the pin receives the correct edge from an external device, the port pin generates an interrupt request signal to the CPU. Any time a port pin is configured for an edge-triggered interrupt, writing a 1 to that pin's Port *x* Data register causes a reset of an edge-detected interrupt. You must set the bit in the Port *x* Data register to 1 before entering either single or dual edge-triggered interrupt mode for that port pin.

# eZ80F91 Interrupt Controller

The interrupt controller on the eZ80F91 device routes the interrupt request signals from the internal peripherals, external devices (via the internal port I/O), and the non-maskable interrupt (NMI) pin to the CPU.

On the eZ80F91 device, all maskable interrupts use the CPU's vectored interrupt function. The size of the I Register is modified to 16 bits in the eZ80F91 device, differing from that of previous versions of the eZ80 CPU, to allow for a 16 MB range of interrupt vector table placement. Additionally, the size of the IVECT register is increased from eight bits to nine bits to provide an interrupt vector table that can be expanded and is more easily integrated with other interrupts.

**Table 2. Interrupt Vector Sources by Priority**

| Priority | Vector | Source | Priority | Vector | Source |
|---|---|---|---|---|---|
| 0 | 040H | EMAC Rx | 24 | 0A0H | Port B 0 |
| 1 | 044H | EMAC Tx | 25 | 0A4H | Port B 1 |
| 2 | 048H | EMAC SYS | 26 | 0A8H | Port B 2 |
| 3 | 04CH | PLL | 27 | 0ACH | Port B 3 |
| 4 | 050H | Flash | 28 | 0B0H | Port B 4 |
| 5 | 054H | Timer 0 | 29 | 0B4H | Port B 5 |
| 6 | 058H | Timer 1 | 30 | 0B8H | Port B 6 |
| 7 | 05CH | Timer 2 | 31 | 0BCH | Port B 7 |
| 8 | 060H | Timer 3 | 32 | 0C0H | Port C 0 |
| 9 | 064H | unused* | 33 | 0C4H | Port C 1 |
| 10 | 068H | unused* | 34 | 0C8H | Port C 2 |
| 11 | 06CH | RTC | 35 | 0CCH | Port C 3 |
| 12 | 070H | UART 0 | 36 | 0D0H | Port C 4 |
| 13 | 074H | UART 1 | 37 | 0D4H | Port C 5 |
| 14 | 078H | $I^2C$ | 38 | 0D8H | Port C 6 |
| 15 | 07CH | SPI | 39 | 0DCH | Port C 7 |
| 16 | 080H | Port A 0 | 40 | 0E0H | Port D 0 |
| 17 | 084H | Port A 1 | 41 | 0E4H | Port D 1 |
| 18 | 088H | Port A 2 | 42 | 0E8H | Port D 2 |
| 19 | 08CH | Port A 3 | 43 | 0ECH | Port D 3 |
| 20 | 090H | Port A 4 | 44 | 0F0H | Port D 4 |
| 21 | 094H | Port A 5 | 45 | 0F4H | Port D 5 |
| 22 | 098H | Port A 6 | 46 | 0F8H | Port D 6 |
| 23 | 09CH | Port A 7 | 47 | 0FCH | Port D 7 |

Note: *The vector addresses 064H and 068H are left unused to avoid conflict with the NMI address 066H. The NMI is prioritized higher than all maskable interrupts.

The program should store the ISR starting address in the four-byte interrupt vector locations. As an example, in ADL mode, the three-byte address for the SPI interrupt service routine is stored at {I[15:1], 07CH}, {I[15:1], 07DH}, and {I[15:1], 07EH}. In Z80® mode, the two-byte address for the SPI interrupt service routine is be stored at {MBASE[7:0], I[7:1], 07CH} and {MBASE, I[7:1], 07DH}. The least-significant byte (LSB) is stored at the lower address.

When any one or more of the interrupt requests (IRQs) become active, an interrupt request is generated by the interrupt controller and sent to the CPU. The response of the CPU to a vectored interrupt on the eZ80F91 device is explained in Table 3. Interrupt sources are required to be active until the ISR starts.

**Table 3. Vectored Interrupt Operation**

| Memory Mode | ADL Bit | MADL Bit | Operation |
|---|---|---|---|
| Z80® Mode | 0 | 0 | Read the LSB of the interrupt vector placed on the internal vectored interrupt bus, IVECT [8:0], by the interrupting peripheral.<br>IEF1 ← 0<br>IEF2 ← 0<br>The Starting Program Counter is effectively {MBASE, PC[15:0]}.<br>Push the 2-byte return address PC[15:0] onto the ({MBASE,SPS}) stack.<br>The ADL mode bit remains cleared to 0.<br>The interrupt vector address is located at { MBASE, I[7:1], IVECT[8:0] }.<br>PC[23:0] ← ( { MBASE, I[7:1], IVECT[8:0] } ).<br>The interrupt service routine must end with RETI. |
| ADL Mode | 1 | 0 | Read the LSB of the interrupt vector placed on the internal vectored interrupt bus, IVECT [8:0], by the interrupting peripheral.<br>IEF1 ← 0<br>IEF2 ← 0<br>The Starting Program Counter is PC[23:0].<br>Push the 3-byte return address, PC[23:0], onto the SPL stack.<br>The ADL mode bit remains set to 1.<br>The interrupt vector address is located at { I[15:1], IVECT[8:0] }.<br>PC[23:0] ← ( { I[15:1], IVECT[8:0] } ).<br>The interrupt service routine must end with RETI. |
| Z80 Mode | 0 | 1 | Read the LSB of the interrupt vector placed on the internal vectored interrupt bus, IVECT[8:0], bus by the interrupting peripheral.<br>IEF1 ← 0<br>IEF2 ← 0<br>The Starting Program Counter is effectively {MBASE, PC[15:0]}.<br>Push the 2-byte return address, PC[15:0], onto the SPL stack.<br>Push a 00H byte onto the SPL stack to indicate an interrupt from Z80 mode (because ADL = 0).<br>Set the ADL mode bit to 1.<br>The interrupt vector address is located at { I[15:1], IVECT[8:0] }.<br>PC[23:0] ← ( { I[15:1], IVECT[8:0] } ).<br>The interrupt service routine must end with RETI. |

**Table 3. Vectored Interrupt Operation (Continued)**

| Memory Mode | ADL Bit | MADL Bit | Operation |
|---|---|---|---|
| ADL Mode | 1 | 1 | Read the LSB of the interrupt vector placed on the internal vectored interrupt bus, IVECT [8:0], by the interrupting peripheral.<br>IEF1 ← 0<br>IEF2 ← 0<br>The Starting Program Counter is PC[23:0].<br>Push the 3-byte return address, PC[23:0], onto the SPL stack.<br>Push a 01h byte onto the SPL stack to indicate a restart from ADL mode (because ADL = 1).<br>The ADL mode bit remains set to 1.<br>The interrupt vector address is located at {I[15:1], IVECT[8:0]}.<br>PC[23:0] ← ( { I[15:1], IVECT[8:0] } ).<br>The interrupt service routine must end with RETI. |

For more details, refer to *eZ80F91 MCU Product Specification document (PS0192)*, available for download at www.zilog.com.

**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

 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.