# Chapter 8. Interrupts

## 8.1 INTRODUCTION

Interrupt processing is an important aspect of most microcontroller applications. Interrupts may be used to synchronize software operations with events that occur in real time. When interrupts occur, the normal flow of software execution is suspended and special functions are invoked to process the event. At the completion of interrupt processing, previous context information is restored and normal execution resumes.

The 16-bit devices support multiple interrupts from both internal and external sources. In addition, the devices allow high-priority interrupts to override any low priority interrupts that may be in progress.

The compiler provides full support for interrupt processing in C or inline assembly code. This chapter presents an overview of interrupt processing.

## 8.2 HIGHLIGHTS

Items discussed in this chapter are:

- **Writing an Interrupt Service Routine** – You can designate one or more C functions as Interrupt Service Routines (ISR's) to be invoked by the occurrence of an interrupt. For best performance in general, place lengthy calculations or operations that require library calls in the main application. This strategy optimizes performance and minimizes the possibility of losing information when interrupt events occur rapidly.
- **Writing the Interrupt Vector** – The 16-bit devices use interrupt vectors to transfer application control when an interrupt occurs. An interrupt vector is a dedicated location in program memory that specifies the address of an ISR. Applications must contain valid function addresses in these locations to use interrupts.
- **Interrupt Service Routine Context Saving** – To handle returning from an interrupt to code in the same conditional state as before the interrupt, context information from specific registers must be saved.
- **Latency** – The time between when an interrupt is called and when the first ISR instruction is executed is the latency of the interrupt.
- **Nesting Interrupts** – The compiler supports nested interrupts.
- **Enabling/Disabling Interrupts** – Enabling and disabling interrupt sources occurs at two levels: globally and individually.
- **Sharing Memory Between Interrupt Service Routines and Mainline Code** – How to mitigate potential hazards when this technique is used.
- **PSV Usage with Interrupt Service Routines** – Using ISRs with managed psv pointers and CodeGuard Security psv constant sections.

## 8.3 WRITING AN INTERRUPT SERVICE ROUTINE

Following the guidelines in this section, you can write all of your application code, including your interrupt service routines, using only C language constructs.

### 8.3.1 Guidelines for Writing ISR's

The guidelines for writing ISR's are:

- declare ISR's with no parameters and a `void` return type (mandatory)
- do not let ISR's be called by main line code (mandatory)
- do not let ISR's call other functions (recommended)

A 16-bit device ISR is like any other C function in that it can have local variables and access global variables. However, an ISR needs to be declared with no parameters and no return value. This is necessary because the ISR, in response to a hardware interrupt or trap, is invoked asynchronously to the mainline C program (that is, it is not called in the normal way, so parameters and return values don't apply).

ISR's should only be invoked through a hardware interrupt or trap and not from other C functions. An ISR uses the return from interrupt (`RETFIE`) instruction to exit from the function rather than the normal `RETURN` instruction. Using a `RETFIE` instruction out of context can corrupt processor resources, such as the Status register.

Finally, ISR's should not call other functions. This is recommended because of latency issues. See **Section 8.6 "Latency"** for more information.

### 8.3.2 Syntax for Writing ISR's

To declare a C function as an interrupt handler, tag the function with the interrupt attribute (see § 2.3 for a description of the `__attribute__` keyword). The syntax of the interrupt attribute is:

```
__attribute__((interrupt [(
                          [ save(symbol-list)]
                          [, irq(irqid)]
                          [, altirq(altirqid)]
                          [, preprologue(asm)]
                          )]
              ))
```

The `interrupt` attribute name and the parameter names may be written with a pair of underscore characters before and after the name. Thus, `interrupt` and `__interrupt__` are equivalent, as are `save` and `__save__`.

The optional `save` parameter names a list of one or more variables that are to be saved and restored on entry to and exit from the ISR. The list of names is written inside parentheses, with the names separated by commas.

You should arrange to save global variables that may be modified in an ISR if you do not want the value to be exported. Global variables modified by an ISR should be qualified `volatile`.

The optional `irq` parameter allows you to place an interrupt vector at a specific interrupt, and the optional `altirq` parameter allows you to place an interrupt vector at a specified alternate interrupt. Each parameter requires a parenthesized interrupt ID number. (See **Section 8.4 "Writing the Interrupt Vector"** for a list of interrupt ID's.)

The optional `preprologue` parameter allows you to insert assembly-language statements into the generated code immediately before the compiler-generated function prologue.

When using the `interrupt` attribute, please specify either `auto_psv` or `no_auto_psv`. If none is specified a warning will be produced and `auto_psv` will be assumed.

### 8.3.3 Coding ISR's

The following prototype declares function `isr0` to be an interrupt handler:

```
void __attribute__((__interrupt__)) isr0(void);
```

As this prototype indicates, interrupt functions must not take parameters nor may they return a value. The compiler arranges for all working registers to be preserved, as well as the Status register and the Repeat Count register, if necessary. Other variables may be saved by naming them as parameters of the `interrupt` attribute. For example, to have the compiler automatically save and restore the variables, `var1` and `var2`, use the following prototype:

```
void __attribute__((__interrupt__(__save__(var1,var2))))
isr0(void);
```

To request the compiler to use the fast context save (using the `push.s` and `pop.s` instructions), tag the function with the `shadow` attribute (see **Section 2.3.2 "Specifying Attributes of Functions"**). For example:

```
void __attribute__((__interrupt__, __shadow__)) isr0(void);
```

### 8.3.4 Using Macros to Declare Simple ISRs

If an interrupt handler does not require any of the optional parameters of the interrupt attribute, then a simplified syntax may be used. The following macros are defined in the device-specific header files:

```
#define _ISR __attribute__((interrupt))
#define _ISRFAST __attribute__((interrupt, shadow))
```

For example, to declare an interrupt handler for the timer0 interrupt:

```
#include <p30fxxxx.h>
void _ISR _INT0Interrupt(void);
```

To declare an interrupt handler for the SPI1 interrupt with fast context save:

```
#include <p30fxxxx.h>
void _ISRFAST _SPI1Interrupt(void);
```

## 8.4 WRITING THE INTERRUPT VECTOR

dsPIC30F/33F DSC and PIC24F/H MCU devices have two interrupt vector tables – a primary and an alternate table – each containing several exception vectors.

The exception sources have associated with them a primary and alternate exception vector, each occupying a program word, as shown in the tables below. The alternate vector name is used when the `ALTIVT` bit is set in the `INTCON2` register.

> **Note:** A device reset is not handled through the interrupt vector table. Instead, upon device reset, the program counter is cleared. This causes the processor to begin execution at address zero. By convention, the linker script constructs a `GOTO` instruction at that location which transfers control to the C run-time startup module.

To field an interrupt, a function's address must be placed at the appropriate address in one of the vector tables, and the function must preserve any system resources that it uses. It must return to the foreground task using a `RETFIE` processor instruction. Interrupt functions may be written in C. When a C function is designated as an interrupt handler, the compiler arranges to preserve all the system resources which the compiler uses, and to return from the function using the appropriate instruction. The compiler can optionally arrange for the interrupt vector table to be populated with the interrupt function's address.

To arrange for the compiler to fill in the interrupt vector to point to the interrupt function, name the function as denoted in the preceding table. For example, the stack error vector will automatically be filled if the following function is defined:

```
void __attribute__((__interrupt__)) _StackError(void);
```

Note the use of the leading underscore. Similarly, the alternate stack error vector will automatically be filled if the following function is defined:

```
void __attribute__((__interrupt__)) _AltStackError(void);
```

Again, note the use of the leading underscore.

For all interrupt vectors without specific handlers, a default interrupt handler will be installed. The default interrupt handler is supplied by the linker and simply resets the device. An application may also provide a default interrupt handler by declaring an interrupt function with the name `_DefaultInterrupt`.

The last nine interrupt vectors in each table do not have predefined hardware functions. The vectors for these interrupts may be filled by using the names indicated in the preceding table, or, names more appropriate to the application may be used, while still filling the appropriate vector entry by using the `irq` or `altirq` parameter of the interrupt attribute. For example, to specify that a function should use primary interrupt vector fifty-two, use the following:

```
void __attribute__((__interrupt__(__irq__(52)))) MyIRQ(void);
```

Similarly, to specify that a function should use alternate interrupt vector fifty-two, use the following:

```
void __attribute__((__interrupt__(__altirq__(52)))) MyAltIRQ(void);
```

The `irq`/`altirq` number can be one of the interrupt request numbers 45 to 53. If the `irq` parameter of the interrupt attribute is used, the compiler creates the external symbol name `__Interrupt`*n*, where n is the vector number. Therefore, the C identifiers `_Interrupt45` through `_Interrupt53` are reserved by the compiler. In the same way, if the `altirq` parameter of the interrupt attribute is used, the compiler creates the external symbol name `__AltInterrupt`*n*, where n is the vector number. Therefore, the C identifiers `_AltInterrupt45` through `_AltInterrupt53` are reserved by the compiler.

### 8.4.1 dsPIC30F DSCs (Non-SMPS) Interrupt Vectors

The dsPIC30F SMPS devices are currently dsPIC30F1010, dsPIC30F2020 and dsPIC30F2023. All other dsPIC30F devices are non-SMPS.

**TABLE 8-1: INTERRUPT VECTORS – dsPIC30F DSCs (NON-SMPS)**

| IRQ# | Primary Name | Alternate Name | Vector Function |
|---|---|---|---|
| N/A | _ReservedTrap0 | _AltReservedTrap0 | Reserved |
| N/A | _OscillatorFail | _AltOscillatorFail | Oscillator fail trap |
| N/A | _AddressError | _AltAddressError | Address error trap |
| N/A | _StackError | _AltStackError | Stack error trap |
| N/A | _MathError | _AltMathError | Math error trap |
| N/A | _ReservedTrap5 | _AltReservedTrap5 | Reserved |
| N/A | _ReservedTrap6 | _AltReservedTrap6 | Reserved |
| N/A | _ReservedTrap7 | _AltReservedTrap7 | Reserved |
| 0 | _INT0Interrupt | _AltINT0Interrupt | INT0 External interrupt 0 |
| 1 | _IC1Interrupt | _AltIC1Interrupt | IC1 Input capture 1 |
| 2 | _OC1Interrupt | _AltOC1Interrupt | OC1 Output compare 1 |
| 3 | _T1Interrupt | _AltT1Interrupt | TMR1 Timer 1 expired |
| 4 | _IC2Interrupt | _AltIC2Interrupt | IC2 Input capture 2 |
| 5 | _OC2Interrupt | _AltOC2Interrupt | OC2 Output compare 2 |
| 6 | _T2Interrupt | _AltT2Interrupt | TMR2 Timer 2 expired |
| 7 | _T3Interrupt | _AltT3Interrupt | TMR3 Timer 3 expired |
| 8 | _SPI1Interrupt | _AltSPI1Interrupt | SPI1 Serial peripheral interface 1 |
| 9 | _U1RXInterrupt | _AltU1RXInterrupt | UART1RX Uart 1 Receiver |
| 10 | _U1TXInterrupt | _AltU1TXInterrupt | UART1TX Uart 1 Transmitter |
| 11 | _ADCInterrupt | _AltADCInterrupt | ADC convert completed |
| 12 | _NVMInterrupt | _AltNVMInterrupt | NMM NVM write completed |
| 13 | _SI2CInterrupt | _AltSI2CInterrupt | Slave $I^2C$ interrupt |
| 14 | _MI2CInterrupt | _AltMI2CInterrupt | Master $I^2C$ interrupt |
| 15 | _CNInterrupt | _AltCNInterrupt | CN Input change interrupt |
| 16 | _INT1Interrupt | _AltINT1Interrupt | INT1 External interrupt 0 |
| 17 | _IC7Interrupt | _AltIC7Interrupt | IC7 Input capture 7 |
| 18 | _IC8Interrupt | _AltIC8Interrupt | IC8 Input capture 8 |
| 19 | _OC3Interrupt | _AltOC3Interrupt | OC3 Output compare 3 |
| 20 | _OC4Interrupt | _AltOC4Interrupt | OC4 Output compare 4 |
| 21 | _T4Interrupt | _AltT4Interrupt | TMR4 Timer 4 expired |
| 22 | _T5Interrupt | _AltT5Interrupt | TMR5 Timer 5 expired |
| 23 | _INT2Interrupt | _AltINT2Interrupt | INT2 External interrupt 2 |
| 24 | _U2RXInterrupt | _AltU2RXInterrupt | UART2RX Uart 2 Receiver |
| 25 | _U2TXInterrupt | _AltU2TXInterrupt | UART2TX Uart 2 Transmitter |
| 26 | _SPI2Interrupt | _AltSPI2Interrupt | SPI2 Serial Peripheral Interface 2 |
| 27 | _C1Interrupt | _AltC1Interrupt | CAN1 combined IRQ |
| 28 | _IC3Interrupt | _AltIC3Interrupt | IC3 Input capture 3 |
| 29 | _IC4Interrupt | _AltIC4Interrupt | IC4 Input capture 4 |
| 30 | _IC5Interrupt | _AltIC5Interrupt | IC5 Input capture 5 |
| 31 | _IC6Interrupt | _AltIC6Interrupt | IC6 Input capture 6 |
| 32 | _OC5Interrupt | _AltOC5Interrupt | OC5 Output compare 5 |
| 33 | _OC6Interrupt | _AltOC6Interrupt | OC6 Output compare 6 |

**TABLE 8-1: INTERRUPT VECTORS – dsPIC30F DSCs (NON-SMPS)**

| IRQ# | Primary Name | Alternate Name | Vector Function |
|---|---|---|---|
| 34 | _OC7Interrupt | _AltOC7Interrupt | OC7 Output compare 7 |
| 35 | _OC8Interrupt | _AltOC8Interrupt | OC8 Output compare 8 |
| 36 | _INT3Interrupt | _AltINT3Interrupt | INT3 External interrupt 3 |
| 37 | _INT4Interrupt | _AltINT4Interrupt | INT4 External interrupt 4 |
| 38 | _C2Interrupt | _AltC2Interrupt | CAN2 combined IRQ |
| 39 | _PWMInterrupt | _AltPWMInterrupt | PWM period match |
| 40 | _QEIInterrupt | _AltQEIInterrupt | QEI position counter compare |
| 41 | _DCIInterrupt | _AltDCIInterrupt | DCI CODEC transfer completed |
| 42 | _LVDInterrupt | _AltLVDInterrupt | PLVD low voltage detected |
| 43 | _FLTAInterrupt | _AltFLTAInterrupt | FLTA MCPWM fault A |
| 44 | _FLTBInterrupt | _AltFLTBInterrupt | FLTB MCPWM fault B |
| 45 | _Interrupt45 | _AltInterrupt45 | Reserved |
| 46 | _Interrupt46 | _AltInterrupt46 | Reserved |
| 47 | _Interrupt47 | _AltInterrupt47 | Reserved |
| 48 | _Interrupt48 | _AltInterrupt48 | Reserved |
| 49 | _Interrupt49 | _AltInterrupt49 | Reserved |
| 50 | _Interrupt50 | _AltInterrupt50 | Reserved |
| 51 | _Interrupt51 | _AltInterrupt51 | Reserved |
| 52 | _Interrupt52 | _AltInterrupt52 | Reserved |
| 53 | _Interrupt53 | _AltInterrupt53 | Reserved |

### 8.4.2 dsPIC30F DSCs (SMPS) Interrupt Vectors

The dsPIC30F SMPS devices are currently dsPIC30F1010, dsPIC30F2020 and dsPIC30F2023. All other dsPIC30F devices are non-SMPS.

**TABLE 8-2: INTERRUPT VECTORS - dsPIC30F DSCs (SMPS)**

| IRQ# | Primary Name | Alternate Name | Vector Function |
|---|---|---|---|
| N/A | _ReservedTrap0 | _AltReservedTrap0 | Reserved |
| N/A | _OscillatorFail | _AltOscillatorFail | Oscillator fail trap |
| N/A | _AddressError | _AltAddressError | Address error trap |
| N/A | _StackError | _AltStackError | Stack error trap |
| N/A | _MathError | _AltMathError | Math error trap |
| N/A | _ReservedTrap5 | _AltReservedTrap5 | Reserved |
| N/A | _ReservedTrap6 | _AltReservedTrap6 | Reserved |
| N/A | _ReservedTrap7 | _AltReservedTrap7 | Reserved |
| 0 | _INT0Interrupt | _AltINT0Interrupt | INT0 External interrupt 0 |
| 1 | _IC1Interrupt | _AltIC1Interrupt | IC1 Input capture 1 |
| 2 | _OC1Interrupt | _AltOC1Interrupt | OC1 Output compare 1 |
| 3 | _T1Interrupt | _AltT1Interrupt | TMR1 Timer 1 expired |
| 4 | _Interrupt4 | _AltInterrupt4 | Reserved |
| 5 | _OC2Interrupt | _AltOC2Interrupt | OC2 Output compare 2 |
| 6 | _T2Interrupt | _AltT2Interrupt | TMR2 Timer 2 expired |
| 7 | _T3Interrupt | _AltT3Interrupt | TMR3 Timer 3 expired |
| 8 | _SPI1Interrupt | _AltSPI1Interrupt | SPI1 Serial peripheral interface 1 |
| 9 | _U1RXInterrupt | _AltU1RXInterrupt | UART1RX Uart 1 Receiver |
| 10 | _U1TXInterrupt | _AltU1TXInterrupt | UART1TX Uart 1 Transmitter |

**TABLE 8-2: INTERRUPT VECTORS - dsPIC30F DSCs (SMPS) (CONTINUED)**

| IRQ# | Primary Name | Alternate Name | Vector Function |
|---|---|---|---|
| 11 | _ADCInterrupt | _AltADCInterrupt | ADC Convert completed |
| 12 | _NVMInterrupt | _AltNVMInterrupt | NVM write completed |
| 13 | _SI2CInterrupt | _AltSI2CInterrupt | Slave $I^2C$ interrupt |
| 14 | _MI2CInterrupt | _AltMI2CInterrupt | Master $I^2C$ interrupt |
| 15 | _Interrupt15 | _AltInterrupt15 | Reserved |
| 16 | _INT1Interrupt | _AltINT1Interrupt | INT1 External interrupt 1 |
| 17 | _INT2Interrupt | _AltINT2Interrupt | INT2 External interrupt 2 |
| 18 | _PWMSpEvent MatchInterrupt | _AltPWMSpEvent MatchInterrupt | PWM special event interrupt |
| 19 | _PWM1Interrupt | _AltPWM1Interrupt | PWM period match 1 |
| 20 | _PWM2Interrupt | _AltPWM2Interrupt | PWM period match 2 |
| 21 | _PWM3Interrupt | _AltPWM3Interrupt | PWM period match 3 |
| 22 | _PWM4Interrupt | _AltPWM4Interrupt | PWM period match 4 |
| 23 | _Interrupt23 | _AltInterrupt23 | Reserved |
| 24 | _Interrupt24 | _AltInterrupt24 | Reserved |
| 25 | _Interrupt25 | _AltInterrupt25 | Reserved |
| 26 | _Interrupt26 | _AltInterrupt26 | Reserved |
| 27 | _CNInterrupt | _AltCNInterrupt | Input Change Notification |
| 28 | _Interrupt28 | _AltInterrupt28 | Reserved |
| 29 | _CMP1Interrupt | _AltCMP1Interrupt | Analog comparator interrupt 1 |
| 30 | _CMP2Interrupt | _AltCMP2Interrupt | Analog comparator interrupt 2 |
| 31 | _CMP3Interrupt | _AltCMP3Interrupt | Analog comparator interrupt 3 |
| 32 | _CMP4Interrupt | _AltCMP4Interrupt | Analog comparator interrupt 4 |
| 33 | _Interrupt33 | _AltInterrupt33 | Reserved |
| 34 | _Interrupt34 | _AltInterrupt34 | Reserved |
| 35 | _Interrupt35 | _AltInterrupt35 | Reserved |
| 36 | _Interrupt36 | _AltInterrupt36 | Reserved |
| 37 | _ADCP0Interrupt | _AltADCP0Interrupt | ADC Pair 0 conversion complete |
| 38 | _ADCP1Interrupt | _AltADCP1Interrupt | ADC Pair 1 conversion complete |
| 39 | _ADCP2Interrupt | _AltADCP2Interrupt | ADC Pair 2 conversion complete |
| 40 | _ADCP3Interrupt | _AltADCP3Interrupt | ADC Pair 3 conversion complete |
| 41 | _ADCP4Interrupt | _AltADCP4Interrupt | ADC Pair 4 conversion complete |
| 42 | _ADCP5Interrupt | _AltADCP5Interrupt | ADC Pair 5 conversion complete |
| 43 | _Interrupt43 | _AltInterrupt43 | Reserved |
| 44 | _Interrupt44 | _AltInterrupt44 | Reserved |
| 45 | _Interrupt45 | _AltInterrupt45 | Reserved |
| 46 | _Interrupt46 | _AltInterrupt46 | Reserved |
| 47 | _Interrupt47 | _AltInterrupt47 | Reserved |
| 48 | _Interrupt48 | _AltInterrupt48 | Reserved |
| 49 | _Interrupt49 | _AltInterrupt49 | Reserved |
| 50 | _Interrupt50 | _AltInterrupt50 | Reserved |
| 51 | _Interrupt51 | _AltInterrupt51 | Reserved |
| 52 | _Interrupt52 | _AltInterrupt52 | Reserved |
| 53 | _Interrupt53 | _AltInterrupt53 | Reserved |

### 8.4.3 PIC24F MCUs Interrupt Vectors

The table below specifies the interrupt vectors for these 16-bit devices.

**TABLE 8-3: INTERRUPT VECTORS - PIC24F MCUs**

| IRQ# | Primary Name | Alternate Name | Vector Function |
|---|---|---|---|
| N/A | _ReservedTrap0 | _AltReservedTrap0 | Reserved |
| N/A | _OscillatorFail | _AltOscillatorFail | Oscillator fail trap |
| N/A | _AddressError | _AltAddressError | Address error trap |
| N/A | _StackError | _AltStackError | Stack error trap |
| N/A | _MathError | _AltMathError | Math error trap |
| N/A | _ReservedTrap5 | _AltReservedTrap5 | Reserved |
| N/A | _ReservedTrap6 | _AltReservedTrap6 | Reserved |
| N/A | _ReservedTrap7 | _AltReservedTrap7 | Reserved |
| 0 | _INT0Interrupt | _AltINT0Interrupt | INT0 External interrupt 0 |
| 1 | _IC1Interrupt | _AltIC1Interrupt | IC1 Input capture 1 |
| 2 | _OC1Interrupt | _AltOC1Interrupt | OC1 Output compare 1 |
| 3 | _T1Interrupt | _AltT1Interrupt | TMR1 Timer 1 expired |
| 4 | _Interrupt4 | _AltInterrupt4 | Reserved |
| 5 | _IC2Interrupt | _AltIC2Interrupt | IC2 Input capture 2 |
| 6 | _OC2Interrupt | _AltOC2Interrupt | OC2 Output compare 2 |
| 7 | _T2Interrupt | _AltT2Interrupt | TMR2 Timer 2 expired |
| 8 | _T3Interrupt | _AltT3Interrupt | TMR3 Timer 3 expired |
| 9 | _SPI1ErrInterrupt | _AltSPI1ErrInterrupt | SPI1 error interrupt |
| 10 | _SPI1Interrupt | _AltSPI1Interrupt | SPI1 transfer completed interrupt |
| 11 | _U1RXInterrupt | _AltU1RXInterrupt | UART1RX Uart 1 Receiver |
| 12 | _U1TXInterrupt | _AltU1TXInterrupt | UART1TX Uart 1 Transmitter |
| 13 | _ADC1Interrupt | _AltADC1Interrupt | ADC 1 convert completed |
| 14 | _Interrupt14 | _AltInterrupt14 | Reserved |
| 15 | _Interrupt15 | _AltInterrupt15 | Reserved |
| 16 | _SI2C1Interrupt | _AltSI2C1Interrupt | Slave $I^2C$ interrupt 1 |
| 17 | _MI2C1Interrupt | _AltMI2C1Interrupt | Slave $I^2C$ interrupt 1 |
| 18 | _CompInterrupt | _AltCompInterrupt | Comparator interrupt |
| 19 | _CNInterrupt | _AltCNInterrupt | CN Input change interrupt |
| 20 | _INT1Interrupt | _AltINT1Interrupt | INT1 External interrupt 1 |
| 21 | _Interrupt21 | _AltInterrupt21 | Reserved |
| 22 | _Interrupt22 | _AltInterrupt22 | Reserved |
| 23 | _Interrupt23 | _AltInterrupt23 | Reserved |
| 24 | _Interrupt24 | _AltInterrupt24 | Reserved |
| 25 | _OC3Interrupt | _AltOC3Interrupt | OC3 Output compare 3 |
| 26 | _OC4Interrupt | _AltOC4Interrupt | OC4 Output compare 4 |
| 27 | _T4Interrupt | _AltT4Interrupt | TMR4 Timer 4 expired |
| 28 | _T5Interrupt | _AltT5Interrupt | TMR5 Timer 5 expired |
| 29 | _INT2Interrupt | _AltINT2Interrupt | INT2 External interrupt 2 |
| 30 | _U2RXInterrupt | _AltU2RXInterrupt | UART2RX Uart 2 Receiver |
| 31 | _U2TXInterrupt | _AltU2TXInterrupt | UART2TX Uart 2 Transmitter |
| 32 | _SPI2ErrInterrupt | _AltSPI2ErrInterrupt | SPI2 error interrupt |
| 33 | _SPI2Interrupt | _AltSPI2Interrupt | SPI2 transfer completed interrupt |
| 34 | _Interrupt34 | _AltInterrupt34 | Reserved |
| 35 | _Interrupt35 | _AltInterrupt35 | Reserved |

**TABLE 8-3: INTERRUPT VECTORS - PIC24F MCUs (CONTINUED)**

| IRQ# | Primary Name | Alternate Name | Vector Function |
|---|---|---|---|
| 36 | _Interrupt36 | _AltInterrupt36 | Reserved |
| 37 | _IC3Interrupt | _AltIC3Interrupt | IC3 Input capture 3 |
| 38 | _IC4Interrupt | _AltIC4Interrupt | IC4 Input capture 4 |
| 39 | _IC5Interrupt | _AltIC5Interrupt | IC5 Input capture 5 |
| 40 | _Interrupt40 | _AltInterrupt40 | Reserved |
| 41 | _OC5Interrupt | _AltOC5Interrupt | OC5 Output compare 5 |
| 42 | _Interrupt42 | _AltInterrupt42 | Reserved |
| 43 | _Interrupt43 | _AltInterrupt43 | Reserved |
| 44 | _Interrupt44 | _AltInterrupt44 | Reserved |
| 45 | _PMPInterrupt | _AltPMPInterrupt | Parallel master port interrupt |
| 46 | _Interrupt46 | _AltInterrupt46 | Reserved |
| 47 | _Interrupt47 | _AltInterrupt47 | Reserved |
| 48 | _Interrupt48 | _AltInterrupt48 | Reserved |
| 49 | _SI2C2Interrupt | _AltSI2C2Interrupt | Slave $I^2C$ interrupt 2 |
| 50 | _MI2C2Interrupt | _AltMI2C2Interrupt | Slave $I^2C$ interrupt 2 |
| 51 | _Interrupt51 | _AltInterrupt51 | Reserved |
| 52 | _Interrupt52 | _AltInterrupt52 | Reserved |
| 53 | _INT3Interrupt | _AltINT3Interrupt | INT3 External interrupt 3 |
| 54 | _INT4Interrupt | _AltINT4Interrupt | INT4 External interrupt 4 |
| 55 | _Interrupt55 | _AltInterrupt55 | Reserved |
| 56 | _Interrupt56 | _AltInterrupt56 | Reserved |
| 57 | _Interrupt57 | _AltInterrupt57 | Reserved |
| 58 | _Interrupt58 | _AltInterrupt58 | Reserved |
| 59 | _Interrupt59 | _AltInterrupt59 | Reserved |
| 60 | _Interrupt60 | _AltInterrupt60 | Reserved |
| 61 | _Interrupt61 | _AltInterrupt61 | Reserved |
| 62 | _RTCCInterrupt | _AltRTCCInterrupt | Real-time clock and calender |
| 63 | _Interrupt63 | _AltInterrupt63 | Reserved |
| 64 | _Interrupt64 | _AltInterrupt64 | Reserved |
| 65 | _U1EInterrupt | _AltU1EInterrupt | UART1 error interrupt |
| 66 | _U2EInterrupt | _AltU2EInterrupt | UART2 error interrupt |
| 67 | _CRCInterrupt | _AltCRCInterrupt | Cyclic Redundancy Check |
| 68 | _Interrupt68 | _AltInterrupt68 | Reserved |
| 69 | _Interrupt69 | _AltInterrupt69 | Reserved |
| 70 | _Interrupt70 | _AltInterrupt70 | Reserved |
| 71 | _Interrupt71 | _AltInterrupt71 | Reserved |
| 72 | _Interrupt72 | _AltInterrupt72 | Reserved |
| 73 | _Interrupt73 | _AltInterrupt73 | Reserved |
| 74 | _Interrupt74 | _AltInterrupt74 | Reserved |
| 75 | _Interrupt75 | _AltInterrupt75 | Reserved |
| 76 | _Interrupt76 | _AltInterrupt76 | Reserved |
| 77 | _Interrupt77 | _AltInterrupt77 | Reserved |
| 78 | _Interrupt78 | _AltInterrupt78 | Reserved |
| 79 | _Interrupt79 | _AltInterrupt79 | Reserved |
| 80 | _Interrupt80 | _AltInterrupt80 | Reserved |

**TABLE 8-3: INTERRUPT VECTORS - PIC24F MCUs (CONTINUED)**

| IRQ# | Primary Name | Alternate Name | Vector Function |
|---|---|---|---|
| 81 | _Interrupt81 | _AltInterrupt81 | Reserved |
| 82 | _Interrupt82 | _AltInterrupt82 | Reserved |
| 83 | _Interrupt83 | _AltInterrupt83 | Reserved |
| 84 | _Interrupt84 | _AltInterrupt84 | Reserved |
| 85 | _Interrupt85 | _AltInterrupt85 | Reserved |
| 86 | _Interrupt86 | _AltInterrupt86 | Reserved |
| 87 | _Interrupt87 | _AltInterrupt87 | Reserved |
| 88 | _Interrupt88 | _AltInterrupt88 | Reserved |
| 89 | _Interrupt89 | _AltInterrupt89 | Reserved |
| 90 | _Interrupt90 | _AltInterrupt90 | Reserved |
| 91 | _Interrupt91 | _AltInterrupt91 | Reserved |
| 92 | _Interrupt92 | _AltInterrupt92 | Reserved |
| 93 | _Interrupt93 | _AltInterrupt93 | Reserved |
| 94 | _Interrupt94 | _AltInterrupt94 | Reserved |
| 95 | _Interrupt95 | _AltInterrupt95 | Reserved |
| 96 | _Interrupt96 | _AltInterrupt96 | Reserved |
| 97 | _Interrupt97 | _AltInterrupt97 | Reserved |
| 98 | _Interrupt98 | _AltInterrupt98 | Reserved |
| 99 | _Interrupt99 | _AltInterrupt99 | Reserved |
| 100 | _Interrupt100 | _AltInterrupt100 | Reserved |
| 101 | _Interrupt101 | _AltInterrupt101 | Reserved |
| 102 | _Interrupt102 | _AltInterrupt102 | Reserved |
| 103 | _Interrupt103 | _AltInterrupt103 | Reserved |
| 104 | _Interrupt104 | _AltInterrupt104 | Reserved |
| 105 | _Interrupt105 | _AltInterrupt105 | Reserved |
| 106 | _Interrupt106 | _AltInterrupt106 | Reserved |
| 107 | _Interrupt107 | _AltInterrupt107 | Reserved |
| 108 | _Interrupt108 | _AltInterrupt108 | Reserved |
| 109 | _Interrupt109 | _AltInterrupt109 | Reserved |
| 110 | _Interrupt110 | _AltInterrupt110 | Reserved |
| 111 | _Interrupt111 | _AltInterrupt111 | Reserved |
| 112 | _Interrupt112 | _AltInterrupt112 | Reserved |
| 113 | _Interrupt113 | _AltInterrupt113 | Reserved |
| 114 | _Interrupt114 | _AltInterrupt114 | Reserved |
| 115 | _Interrupt115 | _AltInterrupt115 | Reserved |
| 116 | _Interrupt116 | _AltInterrupt116 | Reserved |
| 117 | _Interrupt117 | _AltInterrupt117 | Reserved |

### 8.4.4 dsPIC33F DSCs/PIC24H MCUs Interrupt Vectors

The table below specifies the interrupt vectors for these 16-bit devices.

**TABLE 8-4: INTERRUPT VECTORS - dsPIC33F DSCs/PIC24H MCUs**

| IRQ# | Primary Name | Alternate Name | Vector Function |
|---|---|---|---|
| N/A | _ReservedTrap0 | _AltReservedTrap0 | Reserved |
| N/A | _OscillatorFail | _AltOscillatorFail | Oscillator fail trap |
| N/A | _AddressError | _AltAddressError | Address error trap |

**TABLE 8-4: INTERRUPT VECTORS - dsPIC33F DSCs/PIC24H MCUs**

| IRQ# | Primary Name | Alternate Name | Vector Function |
|---|---|---|---|
| N/A | _StackError | _AltStackError | Stack error trap |
| N/A | _MathError | _AltMathError | Math error trap |
| N/A | _DMACError | _AltDMACError | DMA conflict error trap |
| N/A | _ReservedTrap6 | _AltReservedTrap6 | Reserved |
| N/A | _ReservedTrap7 | _AltReservedTrap7 | Reserved |
| 0 | _INT0Interrupt | _AltINT0Interrupt | INT0 External interrupt 0 |
| 1 | _IC1Interrupt | _AltIC1Interrupt | IC1 Input capture 1 |
| 2 | _OC1Interrupt | _AltOC1Interrupt | OC1 Output compare 1 |
| 3 | _T1Interrupt | _AltT1Interrupt | TMR1 Timer 1 expired |
| 4 | _DMA0Interrupt | _AltDMA0Interrupt | DMA 0 interrupt |
| 5 | _IC2Interrupt | _AltIC2Interrupt | IC2 Input capture 2 |
| 6 | _OC2Interrupt | _AltOC2Interrupt | OC2 Output compare 2 |
| 7 | _T2Interrupt | _AltT2Interrupt | TMR2 Timer 2 expired |
| 8 | _T3Interrupt | _AltT3Interrupt | TMR3 Timer 3 expired |
| 9 | _SPI1ErrInterrupt | _AltSPI1ErrInterrupt | SPI1 error interrupt |
| 10 | _SPI1Interrupt | _AltSPI1Interrupt | SPI1 transfer completed interrupt |
| 11 | _U1RXInterrupt | _AltU1RXInterrupt | UART1RX Uart 1 Receiver |
| 12 | _U1TXInterrupt | _AltU1TXInterrupt | UART1TX Uart 1 Transmitter |
| 13 | _ADC1Interrupt | _AltADC1Interrupt | ADC 1 convert completed |
| 14 | _DMA1Interrupt | _AltDMA1Interrupt | DMA 1 interrupt |
| 15 | _Interrupt15 | _AltInterrupt15 | Reserved |
| 16 | _SI2C1Interrupt | _AltSI2C1Interrupt | Slave $I^2C$ interrupt 1 |
| 17 | _MI2C1Interrupt | _AltMI2C1Interrupt | Master $I^2C$ interrupt 1 |
| 18 | _Interrupt18 | _AltInterrupt18 | Reserved |
| 19 | _CNInterrupt | _AltCNInterrupt | CN Input change interrupt |
| 20 | _INT1Interrupt | _AltINT1Interrupt | INT1 External interrupt 1 |
| 21 | _ADC2Interrupt | _AltADC2Interrupt | ADC 2 convert completed |
| 22 | _IC7Interrupt | _AltIC7Interrupt | IC7 Input capture 7 |
| 23 | _IC8Interrupt | _AltIC8Interrupt | IC8 Input capture 8 |
| 24 | _DMA2Interrupt | _AltDMA2Interrupt | DMA 2 interrupt |
| 25 | _OC3Interrupt | _AltOC3Interrupt | OC3 Output compare 3 |
| 26 | _OC4Interrupt | _AltOC4Interrupt | OC4 Output compare 4 |
| 27 | _T4Interrupt | _AltT4Interrupt | TMR4 Timer 4 expired |
| 28 | _T5Interrupt | _AltT5Interrupt | TMR5 Timer 5 expired |
| 29 | _INT2Interrupt | _AltINT2Interrupt | INT2 External interrupt 2 |
| 30 | _U2RXInterrupt | _AltU2RXInterrupt | UART2RX Uart 2 Receiver |
| 31 | _U2TXInterrupt | _AltU2TXInterrupt | UART2TX Uart 2 Transmitter |
| 32 | _SPI2ErrInterrupt | _AltSPI2ErrInterrupt | SPI2 error interrupt |
| 33 | _SPI2Interrupt | _AltSPI2Interrupt | SPI2 transfer completed interrupt |
| 34 | _C1RxRdyInterrupt | _AltC1RxRdyInterrupt | CAN1 receive data ready |
| 35 | _C1Interrupt | _AltC1Interrupt | CAN1 completed interrupt |
| 36 | _DMA3Interrupt | _AltDMA3Interrupt | DMA 3 interrupt |
| 37 | _IC3Interrupt | _AltIC3Interrupt | IC3 Input capture 3 |
| 38 | _IC4Interrupt | _AltIC4Interrupt | IC4 Input capture 4 |
| 39 | _IC5Interrupt | _AltIC5Interrupt | IC5 Input capture 5 |

**TABLE 8-4: INTERRUPT VECTORS - dsPIC33F DSCs/PIC24H MCUs**

| IRQ# | Primary Name | Alternate Name | Vector Function |
|---|---|---|---|
| 40 | _IC6Interrupt | _AltIC6Interrupt | IC6 Input capture 6 |
| 41 | _OC5Interrupt | _AltOC5Interrupt | OC5 Output compare 5 |
| 42 | _OC6Interrupt | _AltOC6Interrupt | OC6 Output compare 6 |
| 43 | _OC7Interrupt | _AltOC7Interrupt | OC7 Output compare 7 |
| 44 | _OC8Interrupt | _AltOC8Interrupt | OC8 Output compare 8 |
| 45 | _Interrupt45 | _AltInterrupt45 | Reserved |
| 46 | _DMA4Interrupt | _AltDMA4Interrupt | DMA 4 interrupt |
| 47 | _T6Interrupt | _AltT6Interrupt | TMR6 Timer 6 expired |
| 48 | _T7Interrupt | _AltT7Interrupt | TMR7 Timer 7 expired |
| 49 | _SI2C2Interrupt | _AltSI2C2Interrupt | Slave $I^2C$ interrupt 2 |
| 50 | _MI2C2Interrupt | _AltMI2C2Interrupt | Master $I^2C$ interrupt 2 |
| 51 | _T8Interrupt | _AltT8Interrupt | TMR8 Timer 8 expired |
| 52 | _T9Interrupt | _AltT9Interrupt | TMR9 Timer 9 expired |
| 53 | _INT3Interrupt | _AltINT3Interrupt | INT3 External interrupt 3 |
| 54 | _INT4Interrupt | _AltINT4Interrupt | INT4 External interrupt 4 |
| 55 | _C2RxRdyInterrupt | _AltC2RxRdyInterrupt | CAN2 receive data ready |
| 56 | _C2Interrupt | _AltC2Interrupt | CAN2 completed interrupt |
| 57 | _PWMInterrupt | _AltPWMInterrupt | PWM period match |
| 58 | _QEIInterrupt | _AltQEIInterrupt | QEI position counter compare |
| 59 | _DCIErrInterrupt | _AltDCIErrInterrupt | DCI CODEC error interrupt |
| 60 | _DCIInterrupt | _AltDCIInterrupt | DCI CODEC transfer done |
| 61 | _DMA5Interrupt | _AltDMA5Interrupt | DMA channel 5 interrupt |
| 62 | _Interrupt62 | _AltInterrupt62 | Reserved |
| 63 | _FLTAInterrupt | _AltFLTAInterrupt | FLTA MCPWM fault A |
| 64 | _FLTBInterrupt | _AltFLTBInterrupt | FLTB MCPWM fault B |
| 65 | _U1ErrInterrupt | _AltU1ErrInterrupt | UART1 error interrupt |
| 66 | _U2ErrInterrupt | _AltU2ErrInterrupt | UART2 error interrupt |
| 67 | _Interrupt67 | _AltInterrupt67 | Reserved |
| 68 | _DMA6Interrupt | _AltDMA6Interrupt | DMA channel 6 interrupt |
| 69 | _DMA7Interrupt | _AltDMA7Interrupt | DMA channel 7 interrupt |
| 70 | _C1TxReqInterrupt | _AltC1TxReqInterrupt | CAN1 transmit data request |
| 71 | _C2TxReqInterrupt | _AltC2TxReqInterrupt | CAN2 transmit data request |
| 72 | _Interrupt72 | _AltInterrupt72 | Reserved |
| 73 | _Interrupt73 | _AltInterrupt73 | Reserved |
| 74 | _Interrupt74 | _AltInterrupt74 | Reserved |
| 75 | _Interrupt75 | _AltInterrupt75 | Reserved |
| 76 | _Interrupt76 | _AltInterrupt76 | Reserved |
| 77 | _Interrupt77 | _AltInterrupt77 | Reserved |
| 78 | _Interrupt78 | _AltInterrupt78 | Reserved |
| 79 | _Interrupt79 | _AltInterrupt79 | Reserved |
| 80 | _Interrupt80 | _AltInterrupt80 | Reserved |
| 81 | _Interrupt81 | _AltInterrupt81 | Reserved |
| 82 | _Interrupt82 | _AltInterrupt82 | Reserved |
| 83 | _Interrupt83 | _AltInterrupt83 | Reserved |
| 84 | _Interrupt84 | _AltInterrupt84 | Reserved |

**TABLE 8-4: INTERRUPT VECTORS - dsPIC33F DSCs/PIC24H MCUs**

| IRQ# | Primary Name | Alternate Name | Vector Function |
|---|---|---|---|
| 85 | _Interrupt85 | _AltInterrupt85 | Reserved |
| 86 | _Interrupt86 | _AltInterrupt86 | Reserved |
| 87 | _Interrupt87 | _AltInterrupt87 | Reserved |
| 88 | _Interrupt88 | _AltInterrupt88 | Reserved |
| 89 | _Interrupt89 | _AltInterrupt89 | Reserved |
| 90 | _Interrupt90 | _AltInterrupt90 | Reserved |
| 91 | _Interrupt91 | _AltInterrupt91 | Reserved |
| 92 | _Interrupt92 | _AltInterrupt92 | Reserved |
| 93 | _Interrupt93 | _AltInterrupt93 | Reserved |
| 94 | _Interrupt94 | _AltInterrupt94 | Reserved |
| 95 | _Interrupt95 | _AltInterrupt95 | Reserved |
| 96 | _Interrupt96 | _AltInterrupt96 | Reserved |
| 97 | _Interrupt97 | _AltInterrupt97 | Reserved |
| 98 | _Interrupt98 | _AltInterrupt98 | Reserved |
| 99 | _Interrupt99 | _AltInterrupt99 | Reserved |
| 100 | _Interrupt100 | _AltInterrupt100 | Reserved |
| 101 | _Interrupt101 | _AltInterrupt101 | Reserved |
| 102 | _Interrupt102 | _AltInterrupt102 | Reserved |
| 103 | _Interrupt103 | _AltInterrupt103 | Reserved |
| 104 | _Interrupt104 | _AltInterrupt104 | Reserved |
| 105 | _Interrupt105 | _AltInterrupt105 | Reserved |
| 106 | _Interrupt106 | _AltInterrupt106 | Reserved |
| 107 | _Interrupt107 | _AltInterrupt107 | Reserved |
| 108 | _Interrupt108 | _AltInterrupt108 | Reserved |
| 109 | _Interrupt109 | _AltInterrupt109 | Reserved |
| 110 | _Interrupt110 | _AltInterrupt110 | Reserved |
| 111 | _Interrupt111 | _AltInterrupt111 | Reserved |
| 112 | _Interrupt112 | _AltInterrupt112 | Reserved |
| 113 | _Interrupt113 | _AltInterrupt113 | Reserved |
| 114 | _Interrupt114 | _AltInterrupt114 | Reserved |
| 115 | _Interrupt115 | _AltInterrupt115 | Reserved |
| 116 | _Interrupt116 | _AltInterrupt116 | Reserved |
| 117 | _Interrupt117 | _AltInterrupt117 | Reserved |

## 8.5 INTERRUPT SERVICE ROUTINE CONTEXT SAVING

Interrupts, by their very nature, can occur at unpredictable times. Therefore, the interrupted code must be able to resume with the same machine state that was present when the interrupt occurred.

To properly handle a return from interrupt, the setup (prologue) code for an ISR function automatically saves the compiler-managed working and special function registers on the stack for later restoration at the end of the ISR. You can use the optional `save` parameter of the `interrupt` attribute to specify additional variables and special function registers to be saved and restored.

In certain applications, it may be necessary to insert assembly statements into the interrupt service routine immediately prior to the compiler-generated function prologue. For example, it may be required that a semaphore be incremented immediately on entry to an interrupt service routine. This can be done as follows:

```
void
__attribute__((__interrupt__(__preprologue__("inc _semaphore"))))
  isr0(void);
```

## 8.6 LATENCY

There are two elements that affect the number of cycles between the time the interrupt source occurs and the execution of the first instruction of your ISR code. These are:

- **Processor Servicing of Interrupt** – The amount of time it takes the processor to recognize the interrupt and branch to the first address of the interrupt vector. To determine this value refer to the processor data sheet for the specific processor and interrupt source being used.
- **ISR Code** – The compiler saves the registers that it uses in the ISR. This includes the working registers and the RCOUNT special function register. Moreover, if the ISR calls an ordinary function, then the compiler will save all the working registers and RCOUNT, even if they are not all used explicitly in the ISR itself. This must be done, because the compiler cannot know, in general, which resources are used by the called function.

## 8.7 NESTING INTERRUPTS

The 16-bit devices support nested interrupts. Since processor resources are saved on the stack in an ISR, nested ISR's are coded in just the same way as non-nested ones. Nested interrupts are enabled by clearing the NSTDIS (nested interrupt disable) bit in the INTCON1 register. Note that this is the default condition as the 16-bit device comes out of reset with nested interrupts enabled. Each interrupt source is assigned a priority in the Interrupt Priority Control registers (IPC*n*). If there is a pending Interrupt Request (IRQ) with a priority level equal to or greater than the current processor priority level in the Processor Status register (CPUPRI field in the ST register), an interrupt will be presented to the processor.

## 8.8 ENABLING/DISABLING INTERRUPTS

Each interrupt source can be individually enabled or disabled. One interrupt enable bit for each IRQ is allocated in the Interrupt Enable Control registers (IEC*n*). Setting an interrupt enable bit to one (1) enables the corresponding interrupt; clearing the interrupt enable bit to zero (0) disables the corresponding interrupt. When the device comes out of reset, all interrupt enable bits are cleared to zero. In addition, the processor has a disable interrupt instruction (`DISI`) that can disable all interrupts for a specified number of instruction cycles.

> **Note:** Traps, such as the address error trap, cannot be disabled. Only IRQs can be disabled.

The `DISI` instruction can be used in a C program through inline assembly. For example, the inline assembly statement:

```
__asm__ volatile ("disi #16");
```

will emit the specified `DISI` instruction at the point it appears in the source program. A disadvantage of using `DISI` in this way is that the C programmer cannot always be sure how the C compiler will translate C source to machine instructions, so it may be difficult to determine the cycle count for the `DISI` instruction. It is possible to get around this difficulty by bracketing the code that is to be protected from interrupts by `DISI` instructions, the first of which sets the cycle count to the maximum value, and the second of which sets the cycle count to zero. For example,

```
__asm__ volatile("disi #0x3FFF"); /* disable interrupts */
/* ... protected C code ... */
__asm__ volatile("disi #0x0000"); /* enable interrupts */
```

An alternative approach is to write directly to the DISICNT register to enable interrupts. The DISICNT register may be modified only after a `DISI` instruction has been issued and if the contents of the DISICNT register are not zero.

```
__asm__ volatile("disi #0x3FFF"); /* disable interrupts */
/* ... protected C code ... */
DISICNT = 0x0000; /* enable interrupts */
```

For some applications, it may be necessary to disable level 7 interrupts as well. These can only be disabled through the modification of the `COROCON IPL` field. The provided support files contain some useful preprocessor macro functions to help you safely modify the IPL value. These macros are:

```
SET_CPU_IPL(ipl)
SET_AND_SAVE_CPU_IPL(save_to, ipl)
RESTORE_CPU_IPL(saved_to)
```

For example, you may wish to protect a section of code from interrupt. The following code will adjust the current `IPL` setting and restore the `IPL` to its previous value.

```
void foo(void) {
  int current_cpu_ipl;

  SET_AND_SAVE_CPU_IPL(current_cpu_ipl, 7);  /* disable interrupts */
   /* protected code here */
  RESTORE_CPU_IPL(current_cpu_ipl);
}
```

## 8.9 SHARING MEMORY BETWEEN INTERRUPT SERVICE ROUTINES AND MAINLINE CODE

Care must be taken when modifying the same variable within a main or low-priority Interrupt Service Routine (ISR) and a high-priority ISR. Higher priority interrupts, when enabled, can interrupt a multiple instruction sequence and yield unexpected results when a low-priority function has created a multiple instruction Read-Modify-Write sequence accessing the same variable. Therefore, embedded systems must implement an *atomic* operation to ensure that the intervening high-priority ISR will not write to the same variable from which the low-priority ISR has just read, but has not yet completed its write.

An atomic operation is one that cannot be broken down into its constituent parts - it cannot be interrupted. Depending upon the particular architecture involved, not all C expressions translate into an atomic operation. On dsPIC DSC devices, these expressions mainly fall into the following categories: 32-bit expressions, floating point arithmetic, division, and operations on multi-bit bitfields. Other factors will determine whether or not an atomic operation will be generated, such as memory model settings, optimization level and resource availability.

Consider the general expression:

```
foo = bar op baz;
```

The operator (op) may or may not be atomic, based on device architecture. In any event, the compiler may not be able to generate the atomic operation in all instances - this will very much depend upon several factors:

- the availability of an appropriate atomic machine instruction
- the resource availability - special registers or other constraints
- the optimization level, and other options that affect data/code placement

Without knowledge of the architecture, it is reasonable to assume that the general expression requires two reads, one for each operand and one write to store the result. Several difficulties may arise in the presence of interrupt sequences; they very much depend on the particular application.

### 8.9.1 Development Issues

Here are some examples:

**EXAMPLE 8-1: BAR MUST MATCH BAZ**

If it is required that `bar` and `baz` match, (i.e., are updated synchronously with each other), there is a possible hazard if either `bar` or `baz` can be updated within a higher priority interrupt expression. Here are some sample flow sequences:

1. Safe:
   read `bar`
   read `baz`
   perform operation
   write back result to `foo`

2. Unsafe:
   read bar
   **interrupt modifies baz**
   read baz
   perform operation
   write back result to foo
3. Safe:
   read bar
   read baz
   **interrupt modifies bar or baz**
   perform operation
   write back result to foo

The first is safe because any interrupt falls outside the boundaries of the expression. The second is unsafe because the application demands that bar and baz be updated synchronously with each other. The third is probably safe; foo will possibly have an old value, but the value will be consistent with the data that was available at the start of the expression.

**EXAMPLE 8-2: TYPE OF FOO, BAR AND BAZ**

Another variation depends upon the type of foo, bar and baz. The operations, "read bar", "read baz", or "write back result to foo", may not be atomic, depending upon the architecture of the target processor. For example, dsPIC DSC devices can read or write an 8-bit, 16-bit, or 32-bit quantity in 1 (atomic) instruction. But, a 32-bit quantity may require two instructions depending upon instruction selection (which in turn will depend upon optimization and memory model settings). Assume that the types are long and the compiler is unable to choose atomic operations for accessing the data. Then the access becomes:

read lsw bar
read msw bar
read lsw baz
read msw baz
perform operation (on lsw and on msw)
perform operation
write back lsw result to foo
write back msw result to foo

Now there are more possibilities for an update of bar or baz to cause unexpected data.

**EXAMPLE 8-3: BIT FIELDS**

A third cause for concern are bit fields. C allows memory to be allocated at the bit level, but does not define any bit operations. In the purest sense, any operation on a bit will be treated as an operation on the underlying type of the bit field and will usually require some operations to extract the field from bar and baz or to insert the field into foo. The important consideration to note is that (again depending upon instruction architecture, optimization levels and memory settings) an interrupted routine that writes to any portion of the bit field where foo resides may be corruptible. This is particularly apparent in the case where one of the operands is also the destination.

The dsPIC DSC instruction set can operate on 1 bit atomically. The compiler may select these instructions depending upon optimization level, memory settings and resource availability.

**EXAMPLE 8-4: CACHED MEMORY VALUES IN REGISTERS**

Finally, the compiler may choose to cache memory values in registers. These are often referred to as register variables and are particularly prone to interrupt corruption, even when an operation involving the variable is not being interrupted. Ensure that memory resources shared between an ISR and an interruptible function are designated as `volatile`. This will inform the compiler that the memory location may be updated out-of-line from the serial code sequence. This will not protect against the effect of non-atomic operations, but is never-the-less important.

### 8.9.2 Development Solutions

Here are some strategies to remove potential hazards:

- Design the software system such that the conflicting event cannot occur. Do not share memory between ISRs and other functions. Make ISRs as simple as possible and move the real work to main code.
- Use care when sharing memory and, if possible, avoid sharing bit fields which contain multiple bits.
- Protect non-atomic updates of shared memory from interrupts as you would protect critical sections of code. The following macro can be used for this purpose:

```
#define INTERRUPT_PROTECT(x) {              \
   char saved_ipl;                          \
                                            \
   SET_AND_SAVE_CPU_IPL(saved_ipl,7);       \
   x;                                       \
   RESTORE_CPU_IPL(saved_ipl); } (void) 0;
```

  This macro disables interrupts by increasing the current priority level to 7, performing the desired statement and then restoring the previous priority level.

### 8.9.3 Application Example

The following example highlights some of the points discussed in this section:

```
void __attribute__((interrupt))
     HigherPriorityInterrupt(void) {
      /* User Code Here */
      LATGbits.LATG15 = 1;  /* Set LATG bit 15 */
      IPC0bits.INT0IP = 2;  /* Set Interrupt 0
                               priority (multiple
                               bits involved) to 2 */
       }

int main(void) {
    /* More User Code */
    LATGbits.LATG10 ^= 1;  /* Potential HAZARD -
                              First reads LATG into a W reg,
                              implements XOR operation,
                              then writes result to LATG */

    LATG = 0x1238;         /* No problem, this is a write
                              only assignment operation */

    LATGbits.LATG5 = 1;    /* No problem likely,
                              this is an assignment of a
                              single bit and will use a single
                              instruction bit set operation */
```

```
    LATGbits.LATG2 = 0;     /* No problem likely,
                               single instruction bit clear
                               operation probably used */

    LATG += 0x0001;         /* Potential HAZARD -
                               First reads LATG into a W reg,
                               implements add operation,
                               then writes result to LATG */

    IPC0bits.T1IP = 5;      /* HAZARD -
                               Assigning a multiple bitfield
                               can generate a multiple
                               instruction sequence */

}
```

A statement can be protected from interrupt using the INTERRUPT_PROTECT macro provided above. For this example:

```
INTERRUPT_PROTECT(LATGbits.LATG15 ^= 1); /* Not interruptible by
                                            level 1-7 interrupt
                                            requests and safe
                                            at any optimization
                                            level */
```

## 8.10 PSV USAGE WITH INTERRUPT SERVICE ROUTINES

The introduction of managed psv pointers and CodeGuard Security psv constant sections in compiler v3.0 means that Interrupt Service Routines (ISRs) cannot make any assumptions about the setting of PSVPAG. This is a migration issue for existing applications with ISRs that reference the auto_psv constants section. In previous versions of the compiler, the ISR could assume that the correct value of PSVPAG was set during program startup (unless the programmer had explicitly changed it.)

To help mitigate this problem, two new function attributes will be introduced: auto_psv and no_auto_psv. If an ISR references const variables or string literals using the constants-in-code memory model, the auto_psv attribute should be added to the function definition. This attribute will cause the compiler to preserve the previous contents of PSVPAG and set it to section .const. Upon exit, the previous value of PSVPAG will be restored. For example:

```
void __attribute__((interrupt, auto_psv)) myISR()
{
  /* This function can reference const variables and
  string literals with the constants-in-code memory model. */
}
```

The no_auto_psv attribute is used to indicate that an ISR does not reference the auto_psv constants section. If neither attribute is specified, the compiler will assume auto_psv and will insert the necessary instructions to ensure correct operation at run time. A warning diagnostic message will also be issued. The warning will help alert customers to the migration issue, and to the possibility of reducing interrupt latency by specifying the no_auto_psv attribute.

**NOTES:**