MICROCHIP

AN576

Techniques to Disable Global Interrupts

Author: *Mark Palmer*
Microchip Technology Inc.
Contributions by Martin Burghardt
Manager Applications (Central Europe)

INTRODUCTION

This application note discusses four methods for disabling global interrupts. The method best suited for the application may then be used. All discussion will be specific to the PIC16CXXX family of products, but these concepts are also applicable to the PIC17C42, and are shown in the even numbered examples. Note that the PIC17C42's global interrupt bit is called GLINTD and has an inverse sense compared to the GIE bit of the PIC16CXXX family.

To disable all interrupts, either the Global Interrupt Enable (GIE) bit must be cleared or all the individual interrupt enable bits must be cleared. An issue arises when an instruction clears the GIE bit and an interrupt occurs "simultaneously". For example, when a program executes the instruction `BCF INTCON, GIE` (at address PC), there is a possibility that an interrupt will occur during this instruction. If an interrupt occurs during this instruction, the program would complete execution of this instruction, and then immediately branch to the user's interrupt service routine. This occurs because the GIE bit was not clear (disabled) when the interrupt occurred. Normally at the end of the interrupt service routine is the `RETFIE` instruction. This instruction causes the program to return to the instruction at PC + 1, but also sets the GIE bit (enabled). Therefore the GIE bit is not cleared as expected, and unintended program execution may occur.

One method to ensure that the GIE bit is cleared is shown in Example 1 and Example 2, as well as in the PIC16CXXX data sheets. This method tests the state of the GIE bit, after clearing, to ensure that it was not accidentally set in the user's interrupt service routine by the `RETFIE` instruction. If the GIE bit was accidentally set, the program branches back to the instruction that clears the GIE bit.

In this method, the time to ensure that the GIE bit is cleared is indeterminate. Depending on the frequency of the enabled interrupts during this code segment, unexpected delays into the following code segment may occur. For some applications, this may be undesirable. The following three methods address this issue.

EXAMPLE 1: CLEARING THE GIE BIT (DISABLING INTERRUPTS, METHOD 1, PIC16CXXX)

```
LOOP    BCF     INTCON, GIE     ; Disable Global Interrupt
        BTFSC   INTCON, GIE     ; Global Interrupt Disabled?
        GOTO    LOOP            ; NO, try again
        :                       ; YES, continue with program flow
        :
        BSF     INTCON, GIE     ; Re-enable Global Interrupt
```

EXAMPLE 2: SETTING THE GLINTD BIT (DISABLING INTERRUPTS, METHOD 1, PIC17C42)

```
        :
LOOP    BSF     CPUSTA, GLINTD ; Disable Global Interrupt
        BTFSS   CPUSTA, GLINTD ; Global Interrupt Disabled?
        GOTO    LOOP           ; NO, try again
        :                      ; YES, continue with program flow
        :
        BCF     CPUSTA, GLINTD ; Re-enable Global Interrupt
```

The second method is to disable the individual interrupt enable bits. If it is known which bits are enabled at this point, it can easily be done. Example 3 and Example 4 show the disabling of interrupts, where it is known which sources are enabled (some peripheral interrupts and the T0CKI pin interrupt).

This method also requires the same number of instructions for the disabling/enabling of interrupts, as method 1, but requires a knowledge of which individual interrupt enable bits need to be disabled and (more importantly) re-enabled. The major advantage of this method is that it can minimize the time delay entering the code segment which follows the point where interrupts are disabled.

EXAMPLE 3: CLEARING KNOWN INDIVIDUAL INTERRUPT ENABLE BITS (METHOD 2, PIC16CXXX)

```
:
MOVLW   b'10011111'     ; Disable  Peripheral and T0CKI pin interrupts,
ANDWF   INTCON, F       ; All other bits unchanged
:
:
:
MOVLW   b'01100000'     ; Re-enable Peripheral and T0CKI pin interrupts,
IORWF   INTCON, F       ; All other bits unchanged
:
```

EXAMPLE 4: CLEARING KNOWN INDIVIDUAL INTERRUPT ENABLE BITS (METHOD 2, PIC17C42)

```
:
MOVLW   b'11110011'     ; Disable Peripheral and T0CKI pin interrupts,
ANDWF   INTSTA, F       ; All other bits unchanged
:
:
:
MOVLW   b'00001100'     ; Re-enable Peripheral and T0CKI pin interrupts,
IORWF   INTSTA, F       ; All other bits unchanged
:
```

Method 3 can be used if the states of the individual interrupt enable bits are unknown. A temporary byte of data RAM is required to store the value of the INTCON register. This method is shown in Example 5 and Example 6.

This method also requires more instructions for the disabling/enabling of interrupts than in method 1 or method 2, and also a byte of data RAM to temporarily store the value of the INTCON register. The major advantage of this method is that it minimizes the time delay into the code segment which follows the point where interrupts are disabled.

EXAMPLE 5: CLEARING THE INDIVIDUAL INTERRUPT ENABLE BITS (METHOD 3, PIC16CXXX)

```
    :

    MOVF    INTCON, W          ; Move the value in INTCON to
    MOVWF   S_INTCON           ; a shadow register
    MOVLW   b'10000111'        ; Disable all individual interrupts,
    ANDWF   INTCON, F          ; All other bits unchanged
    :
    :
    :
    MOVF    S_INTCON, W        ; Restore the INTCON register
    IORWF   INTCON, F          ;
    :
```

EXAMPLE 6: CLEARING THE INDIVIDUAL INTERRUPT ENABLE BITS (METHOD 3, PIC17C42)

```
    :
    MOVPF   INTSTA, S_INTSTA   ; Move the value in INTSTA to a shadow register
    MOVLW   b'11110000'        ; Disable all individual interrupts,
    ANDWF   INTSTA, F          ; All other bits unchanged
    :
    :
    :
    MOVFP   S_INTSTA, W        ; Restore the INTSTA register
    IORWF   INTSTA, F          ;
    :
```

The final method is to use a RAM location to "shadow" the value of the GIE bit. This shadow bit can then be used in the interrupt service routine to determine which return instruction to use. That is, either the RETURN or the RETFIE (which enables the GIE bit) instruction. Example 7 and Example 8 show this implementation, which require that a general purpose bit be available to hold the "shadow" GIE value. In these examples, the shadow GIE (S_GIE) bit is contained in the register FLAG_REG. If an interrupt occurs during the clearing of the shadow GIE, the interrupt is responded to. At the end of the interrupt service routine, the shadow GIE bit is cleared so the RETURN instruction is executed. The GIE bit remains disabled and program execution returns to the instruction which tries to clear the GIE bit (disable). No interrupts can occur during this instruction since the GIE bit was not re-enabled after the interrupt service routine.

This method also requires more instructions for the disabling/enabling of interrupts than in method 1 or method 2, and a single bit of data RAM to temporarily store the value of the desired GIE value, and increases the interrupt service routine execution time by one instruction cycle, for most occurrences of interrupts (two cycles worst case). The major advantage of this method is that it minimizes the time delay into the code segment which follows the point where interrupts are disabled. Also, the individual interrupt enable bits need not be modified.

EXAMPLE 7: THE "SHADOW" GIE BIT (METHOD 4, PIC16CXXX)

```
    :
            org     0x004
INT_SERVICE_ROUTINE
    :
    :
    BTFSC   FLAG_REG, S_GIE         ; Is the S_GIE bit enabled?
    RETFIE                          ; YES, the GIE should be enabled
    RETURN                          ; NO, the GIE should be disabled
END_INT_SERVICE_ROUTINE
;
MAIN:
    :
    :
    BCF     FLAG_REG, S_GIE         ; Clear the shadow GIE bit
    BCF     INTCON, GIE             ; Disable interrupts by clearing the GIE bit
    :
    :
    :
    BSF     FLAG_REG, S_GIE         ; Set the shadow GIE bit
    BSF     INTCON, GIE             ; Enable interrupts by setting the GIE bit
    :
    :
    END
```

EXAMPLE 8: THE "SHADOW" GLINTD BIT (METHOD 4, PIC17C42)

```
    :
            org     0x004
INT_SERVICE_ROUTINE
    :
    :
    BTFSS   FLAG_REG, S_GLINTD      ; Is the S_GLINTD bit enabled?
    RETFIE                          ; YES, the GLINTD should be enabled
    RETURN                          ; NO, the GLINTD should be disabled
END_INT_SERVICE_ROUTINE
;
MAIN:
    :
    :
    BSF     FLAG_REG, S_GLINTD      ; Set the shadow GLINTD bit
    BSF     CPUSTA, GLINTD          ; Disable interrupts by setting the GLINTD bit
    :
    :
    :
    BCF     FLAG_REG, S_GLINTD      ; Clear the shadow GLINTD bit
    BCF     CPUSTA, GLINTD          ; Enable interrupts by clearing the GLINTD bit
    :
    :
    END
```

CONCLUSION

In conclusion, different methods exist to ensure that all interrupts are disabled. The requirement(s) of the application determines which of the methods is the best fit. A comparison of the different methods is shown in Table 1.

TABLE 1: COMPARISON OF DIFFERENT METHODS

	Program Memory	Data Memory	Cycle Delay (TCY)	
			Best Case	Worst Case
Method 1	2 words * N	—	2	Indeterminate
Method 2	2 words * N	—	1	1 + TISR
Method 3 - PIC16CXXX	4 words * N	1 byte	3	3 + TISR
- PIC17C42	3 words * N	1 byte	2	2 + TISR
Method 4	2 words * N + 2 words	1 bit	1†	1 + (TISR + 2)

Legend: N - Number of occurrences to disable / re-enable interrupts.
TISR - Time to execute the interrupt service routine.
† This method increases the interrupt service routine time (TISR) by 1 cycle for most occurrences (2 cycles worst case).

NOTES: