Interrupt Programming for the HCS12

Why program with Interrupts?

Microcontrollers have the ability to do several things in an on-going basis. One program in a microcontroller may be tasked with the job of driving motors connected to the wheels of a robot to move it forward or backward. If the robot encounters an obstacle it will need to respond appropriately to this challenge by stopping, reversing direction or turning for example. The obstacle would likely be detected by a sensor(s). A programmer can code the software to detect sensor signals in two ways – by polling or by the use of Interrupts.

Polling Method

Polling involves having the software check on a regular basis whether any of the sensors have detected anything typically by reading a value (usually a voltage) produced by the sensor. This continual monitoring of sensors is not an efficient use of time. Also using a polling methodology to assign a priority to a device can be difficult to do as it often checks all devices in a regular schedule and the code cannot be programmed to ignore a device's request for service.

Interrupt Method

In the Interrupt method, devices signal the microcontroller that they require service by the use of an interrupt signal. The appearance of a signal from a sensor triggers an interrupt request. The microcontroller interrupts whatever it is doing and services the interrupt request. This is usually a much more efficient use of the microcontroller's time. Interrupts can be prioritized and the microcontroller can also be programmed to ignore (mask) certain interrupts.

Interrupt Programming Terminology

The main program that the microcontroller is running is often called the foreground program. In the example above it has the job of moving the robot. The code associated with the interrupt is called the Interrupt Service Routine (ISR), the Interrupt Handler or the background program.

For every interrupt there needs to be an Interrupt Service Routine (ISR). When the interrupt occurs the microcontroller stops whatever it is doing and runs the ISR. When the ISR is done the microcontroller resumes the foreground program.

The terms traps and exceptions are also used to refer to interrupts.

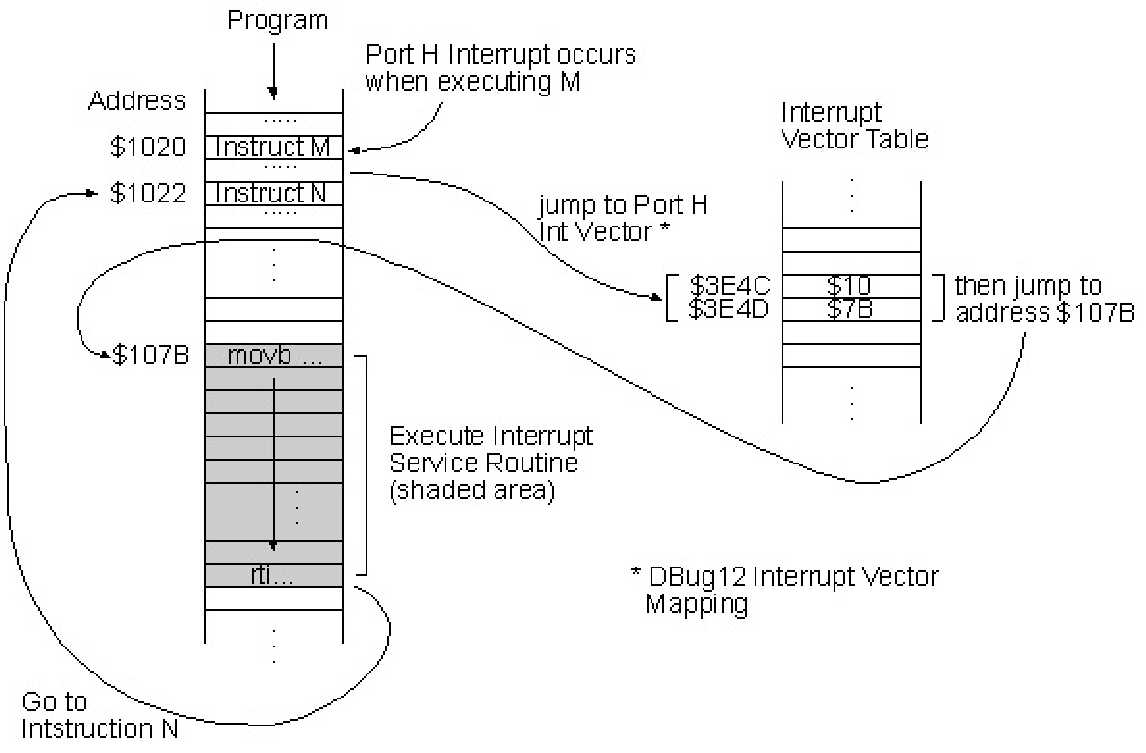
Executing an Interrupt

When the controller receives an interrupt request (software or hardware) it follows these steps.

- The controller finishes executing the current instruction

- The current values of PC, Y, X, D and the CCR registers are pushed on the stack resulting in the Stack Pointer being decremented a total of 9 memory locations from its original value. The current value of PC is referred to as the Return Address.
- The CPU function of the controller fetches the address of the ISR (Interrupt Service Routine) from the interrupt vector table and places this address into the PC register
- The I bit in the CCR is set high to block any further interrupts that might occur while the controller is processing the current interrupt
- The controller executes the ISR starting at the address loaded into the PC register
- The ISR must contain an RTI (Return from Interrupt) as the last instruction in the ISR code. When RTI is executed the controller pulls from the Stack the stored values of CCR, D, X, Y and PC and loads these values into their appropriate registers and the controller resumes running the program code from where it left off.

Note: The service routine may or not return to the interrupted program. Depending on the cause of the interrupt returning to the interrupted program caused by a software error would not resolve this error so the program would likely return to the operating system or monitor program.


Program
Port H Interrupt occurs when executing M
Address
$1020
Instruct M
$1022
Instruct N
Interrupt Vector Table
jump to Port H Int Vector *
$3E4C
$3E4D
$10
$7B
then jump to address $107B
$107B
movb ...
Execute Interrupt Service Routine (shaded area)
rti...
* DBug12 Interrupt Vector Mapping
Go to Intstruction N

Interrupt Maskability

Some types of interrupts can be ignored or not recognized so they called maskable. Masking can be done for interrupts that are not-desired or needed. There are also a few interrupts that cannot be ignored/masked and the controller must recognize them. These are referred to as non-maskable interrupts such as SWI and $\overline{\text{XIRQ}}$. To make a system more flexible the ability to mask an interrupt can usually be done at a global and local level.

Condition Code Register - CCR

The CCR in the HCS12 has 2 bits - I and X - that relate to interrupts.

	Bit 7	6	5	4	3	2	1	Bit 0
Read: Write:	S	X	H	I	N	Z	V	C
Reset:	1	1	0	1	0	0	0	0

X - $\overline{\text{XIRQ}}$ mask bit
This bit enables or disables the XIRQ interrupt

X = 0	$\overline{\text{XIRQ}}$ interrupt is enabled
X = 1	$\overline{\text{XIRQ}}$ interrupt is disabled

I - Interrupt mask bit
This bit can block (globally) all maskable interrupts

I = 0	Maskable interrupt requests enabled
I = 1	Maskable interrupt requests disabled

The commands SEI and CLI in assembler can be used to set or clear the I bit value. These commands have no effect on the $\overline{\text{XIRQ}}$ interrupt as it is non-maskable and is controlled by the X bit in the CCR.

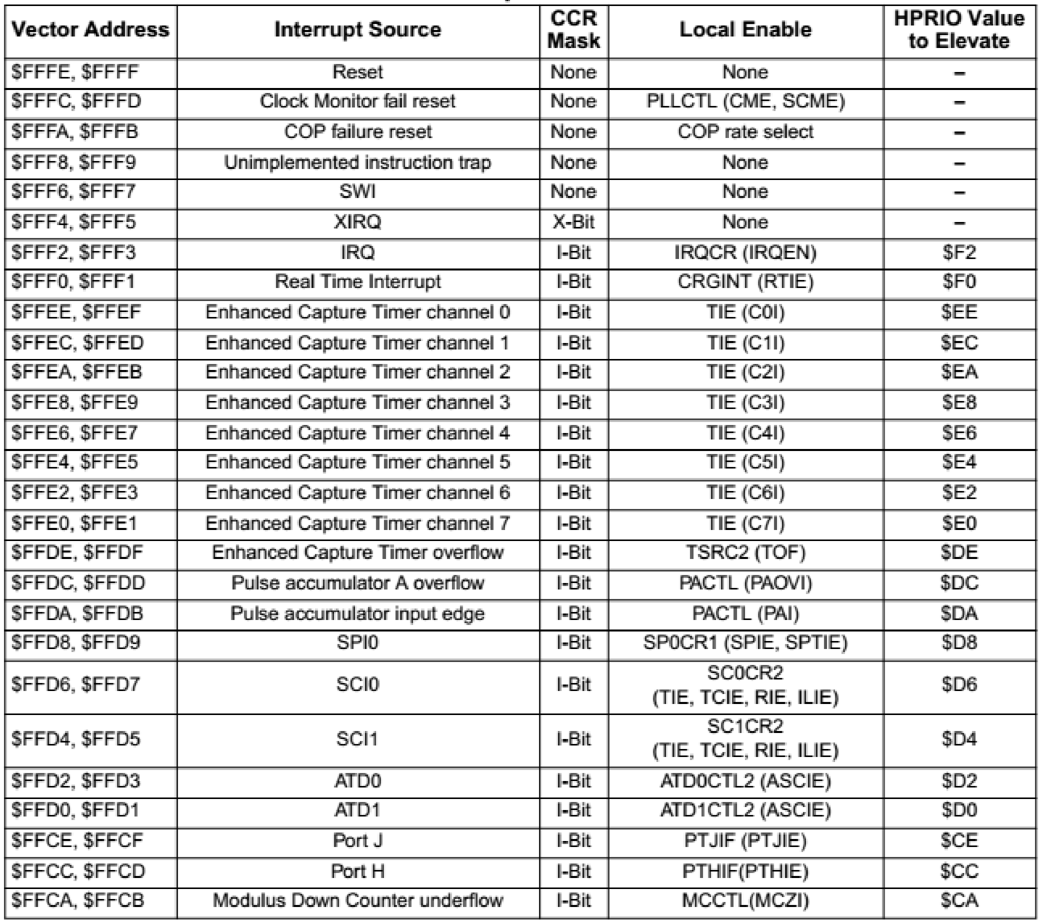
Interrupt Processing Time

Processing an interrupt takes time-based. For example saving the registers on the stack takes at least 9 clock cycles, the RTI instruction takes 8 to 11 clock cycles. And then the ISR takes time to execute depending on the number of instructions. The total time to process the interrupt can then be a minimum of 17-20 clock cycles which can be almost 1 µsec for a 24 MHz E-clock. If this time becomes excessive then an interrupt programming solution may not be the best choice.

Interrupt Vector Table

The table following shows the default address in memory of some of the most commonly used interrupts for the HCS12. In the HCs12 these addresses are located in the upper area of the memory map in ROM. Each entry in the Vector Table holds a 2 byte value referred to as the Interrupt Vector that is the starting address of the interrupt service routine.

Default Interrupt Vector Table for the HCS12



Vector Address	Interrupt Source	CCR Mask	Local Enable	HPRIO Value to Elevate
$FFFE, $FFFF	Reset	None	None	–
$FFFC, $FFFD	Clock Monitor fail reset	None	PLLCTL (CME, SCME)	–
$FFFA, $FFFB	COP failure reset	None	COP rate select	–
$FFF8, $FFF9	Unimplemented instruction trap	None	None	–
$FFF6, $FFF7	SWI	None	None	–
$FFF4, $FFF5	XIRQ	X-Bit	None	–
$FFF2, $FFF3	IRQ	I-Bit	IRQCR (IRQEN)	$F2
$FFF0, $FFF1	Real Time Interrupt	I-Bit	CRGINT (RTIE)	$F0
$FFEE, $FFEF	Enhanced Capture Timer channel 0	I-Bit	TIE (C0I)	$EE
$FFEC, $FFED	Enhanced Capture Timer channel 1	I-Bit	TIE (C1I)	$EC
$FFEA, $FFEB	Enhanced Capture Timer channel 2	I-Bit	TIE (C2I)	$EA
$FFE8, $FFE9	Enhanced Capture Timer channel 3	I-Bit	TIE (C3I)	$E8
$FFE6, $FFE7	Enhanced Capture Timer channel 4	I-Bit	TIE (C4I)	$E6
$FFE4, $FFE5	Enhanced Capture Timer channel 5	I-Bit	TIE (C5I)	$E4
$FFE2, $FFE3	Enhanced Capture Timer channel 6	I-Bit	TIE (C6I)	$E2
$FFE0, $FFE1	Enhanced Capture Timer channel 7	I-Bit	TIE (C7I)	$E0
$FFDE, $FFDF	Enhanced Capture Timer overflow	I-Bit	TSRC2 (TOF)	$DE
$FFDC, $FFDD	Pulse accumulator A overflow	I-Bit	PACTL (PAOVI)	$DC
$FFDA, $FFDB	Pulse accumulator input edge	I-Bit	PACTL (PAI)	$DA
$FFD8, $FFD9	SPI0	I-Bit	SP0CR1 (SPIE, SPTIE)	$D8
$FFD6, $FFD7	SCI0	I-Bit	SC0CR2 (TIE, TCIE, RIE, ILIE)	$D6
$FFD4, $FFD5	SCI1	I-Bit	SC1CR2 (TIE, TCIE, RIE, ILIE)	$D4
$FFD2, $FFD3	ATD0	I-Bit	ATD0CTL2 (ASCIE)	$D2
$FFD0, $FFD1	ATD1	I-Bit	ATD1CTL2 (ASCIE)	$D0
$FFCE, $FFCF	Port J	I-Bit	PTJIF (PTJIE)	$CE
$FFCC, $FFCD	Port H	I-Bit	PTHIF(PTHIE)	$CC
$FFCA, $FFCB	Modulus Down Counter underflow	I-Bit	MCCTL(MCZI)	$CA

Interrupt Priority

All HCS12 interrupts except resets and nonmaskable Interrupts can be prioritized by configuring the HPRIO register. The priority of one of the maskable interrupts can be raised to the highest level but the relative priorities of the others remain the same. To raise the priority of a maskable interrupt to the highest level write the low byte of the vector address of this interrupt to the HPRIO register.

HPRIO Register – address $1F

PSEL7	PSEL6	PSEL5	PSEL4	PSEL3	PSEL2	PSEL1	Not used

Types of Interrupts

1. Maskable Interrupts

These include the $\overline{\text{IRQ}}$ interrupt and all peripheral function interrupts such as those associated with the Timer functions.

2. **Nonmaskable Interrupts**

   These include the $\overline{\text{XIRQ}}$ interrupt, the swi instruction interrupt and the unimplemented opcode trap.

3. **Resets**

   These are the power-on, the manual RESET pin, the COP (Computer Operating Normally), and the clock monitor resets.

Hardware Interrupts

There are 2 externally accessible pins in the HCS12 that are associated with hardware interrupts. These are $\overline{\text{IRQ}}$ and $\overline{\text{XIRQ}}$. $\overline{\text{IRQ}}$ is maskable and $\overline{\text{XIRQ}}$ is non-maskable.

Software Interrupt (SWI)

Then SWI interrupt is executed as a result of a program executing the SWI instruction in a program. When this instruction is executed the controller will go to its appropriate address in the vector table to find the starting address of the ISR for the SWI interrupt.

Peripheral Interrupts

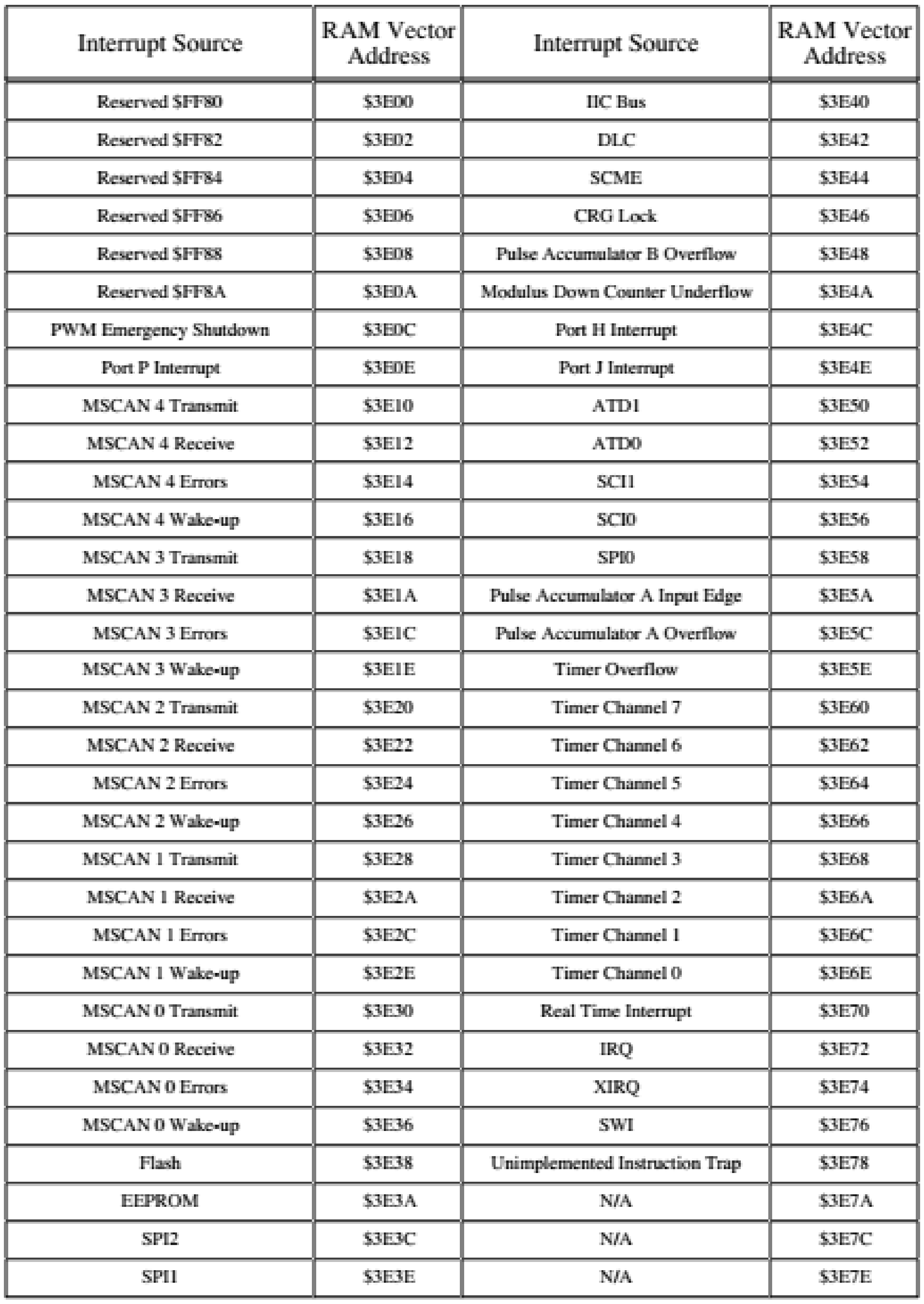
An ISR can be invoked as a result of a condition determined by a HCS12 peripheral device such as a timer function or the A/D converter. The largest number of HCS12 interrupts is in this category. To use these interrupts the programmer must enable them by clearing the I flag in the CCR and as well as enabling the interrupt flag for the particular interrupt of interest.

D-Bug 12 Interrupt Table

The D-Bug 12 monitor's EVB mode does not allow the user to modify the on-chip flash memory. This prevents the use of the default interrupt vector table. To allow a programmer to utilize peripherals in an interrupt driven manner, a RAM based interrupt vector table is provided by D-Bug12. Each of the 64 entries in the table consists of a two byte address with the table beginning at $3E00. Initially, all entries in the table have a value of $0000. To program an interrupt the user must store a value other than $0000 (the starting address of the ISR) in the required RAM interrupt vector table entries. This causes the execution of the interrupt service routine pointed to by the address when an associated interrupt occurs.

If an unmasked interrupt occurs and a table entry contains the default address of $0000, program execution is returned to D-Bug12 where a message is displayed indicating the source of the interrupt and displays the CPU registers at the point where the program was interrupted.

D-Bug 12 RAM-Based Interrupt Table



Interrupt Source	RAM Vector Address	Interrupt Source	RAM Vector Address
Reserved $FF80	$3E00	IIC Bus	$3E40
Reserved $FF82	$3E02	DLC	$3E42
Reserved $FF84	$3E04	SCME	$3E44
Reserved $FF86	$3E06	CRG Lock	$3E46
Reserved $FF88	$3E08	Pulse Accumulator B Overflow	$3E48
Reserved $FF8A	$3E0A	Modulus Down Counter Underflow	$3E4A
PWM Emergency Shutdown	$3E0C	Port H Interrupt	$3E4C
Port P Interrupt	$3E0E	Port J Interrupt	$3E4E
MSCAN 4 Transmit	$3E10	ATD1	$3E50
MSCAN 4 Receive	$3E12	ATD0	$3E52
MSCAN 4 Errors	$3E14	SCI1	$3E54
MSCAN 4 Wake-up	$3E16	SCI0	$3E56
MSCAN 3 Transmit	$3E18	SPI0	$3E58
MSCAN 3 Receive	$3E1A	Pulse Accumulator A Input Edge	$3E5A
MSCAN 3 Errors	$3E1C	Pulse Accumulator A Overflow	$3E5C
MSCAN 3 Wake-up	$3E1E	Timer Overflow	$3E5E
MSCAN 2 Transmit	$3E20	Timer Channel 7	$3E60
MSCAN 2 Receive	$3E22	Timer Channel 6	$3E62
MSCAN 2 Errors	$3E24	Timer Channel 5	$3E64
MSCAN 2 Wake-up	$3E26	Timer Channel 4	$3E66
MSCAN 1 Transmit	$3E28	Timer Channel 3	$3E68
MSCAN 1 Receive	$3E2A	Timer Channel 2	$3E6A
MSCAN 1 Errors	$3E2C	Timer Channel 1	$3E6C
MSCAN 1 Wake-up	$3E2E	Timer Channel 0	$3E6E
MSCAN 0 Transmit	$3E30	Real Time Interrupt	$3E70
MSCAN 0 Receive	$3E32	IRQ	$3E72
MSCAN 0 Errors	$3E34	XIRQ	$3E74
MSCAN 0 Wake-up	$3E36	SWI	$3E76
Flash	$3E38	Unimplemented Instruction Trap	$3E78
EEPROM	$3E3A	N/A	$3E7A
SPI2	$3E3C	N/A	$3E7C
SPI1	$3E3E	N/A	$3E7E

Setting-Up an Interrupt Vector

As an example the following code sets up the interrupt vector for the $\overline{\text{IRQ}}$ interrupt at address $3E72 using the D-Bug 12 interrupt vector value. Assume that the label **irqISR** is used at the start of the interrupt service routine for the $\overline{\text{IRQ}}$ interrupt.

```
UserIRQ:        dc.w    $3E72

        ldd     #irqISR
        std     UserIRQ
```

IRQ Interrupt Example

In this example a 1Hz clock signal is connected to the $\overline{\text{IRQ}}$ interrupt pin. Each time a pulse arrives at this pin an interrupt is generated. The main program sets up the interrupt, initializes PORTB and initializes a counter in memory. The ISR increments the count and sends the count value to PORT B.

The $\overline{\text{IRQ}}$ pin is the only maskable external interrupt signal. The $\overline{\text{IRQ}}$ pin interrupt can be edge or level triggered and also has a local enable bit. Both of these issues are controlled by the Interrupt Control register (INTCR).

Interrupt Control register (INTCR) - address $1E

IRQE	IRQEN	0	0	0	0	0	0

Reset value $40

IRQE bit – can be written once only in normal mode

0 - $\overline{\text{IRQ}}$ pin responds to low level
1 - $\overline{\text{IRQ}}$ pin responds only to falling edge

IRQEN bit – can be written any time in all modes

0 - $\overline{\text{IRQ}}$ pin interrupt disabled
1 - $\overline{\text{IRQ}}$ pin interrupt enabled

```
;*****************************************************************
;*  Program Name: IRQInterruptAssembler
;*  This Assembly program uses the IRQ interrupt on the HCS 12.
;*  A 1 Hz TTL wave form is connected to the IRQ pin.
;*  Each edge causes an interrupt that increments a counter connected to Port B.
;*
;*****************************************************************

;program variables
stack:          equ  $3C00
PORTB:          equ  $01        ;Port B address
DDRB:           equ  $02        ;Port B Data Direction control register
INTCR:          equ  $001E      ;Interrupt control register address
UserIRQ:        equ  $3E72      ;IRQ interrupt vector at address $3E72 for DeBUG 12.

                ORG  $1000
count:  ds.b  1         ;reserve 1 byte for count
```

```
            ORG  $2000
        lds  #stack                ;set up stack pointer
        movw #IRQISR ,UserIRQ     ;set up Interrupt vector in RAM
        clr  count                ;zero counter
        movb  #$FF, DDRB           ;configure port B as output
        movb count, PORTB         ;send count to Port B
        movb #$C0, INTCR          ;enable IRQ pin interrupt, set edge triggering
        cli                        ;clear I bit in CCR - enables all interrupts

;main program - essentially does nothing
again:
        nop
        bra again
        swi                          ;code never reaches here

IRQISR:     ;Interrupt Service Routine
        inc count
        movb count, PORTB            ;send count to Port B
        rti

  end
```

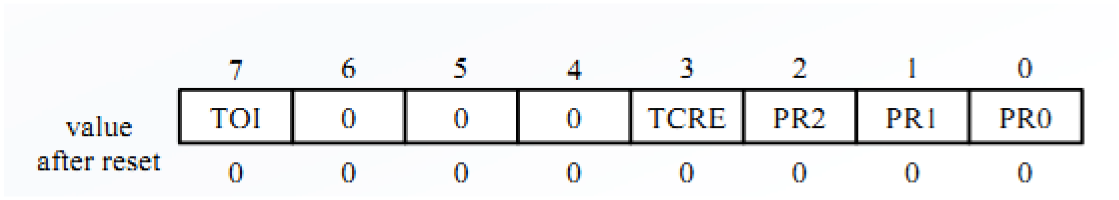
Programming Timer Interrupts

When working with the Timer module it can be seen that the TCNT Timer Overflow Flag is set whenever an overflow of this timer occurs. To use interrupts for programming the timer overflow, TOI, the timer overflow interrupt enable bit in the Timer System Control Register 2 (TSCR2) must be set.

To detect the TOF interrupt (the TCNT overflow flag becoming a 1) the TOI must be set to 1 and the I flag in the CCR must be set to 0.

Timer System Control Register 2 (TSCR2) - address $004D

The main use of this register is to set the pre-scale value for the timer clock frequency.



	7	6	5	4	3	2	1	0
	TOI	0	0	0	TCRE	PR2	PR1	PR0
value after reset	0	0	0	0	0	0	0	0

TOI - timer overflow interrupt enable bit

0	- disable interrupt
1	- enable interrupt when TOF flag is set

The Timer Interrupt Flag Register is used to detect the occurrence of an overflow condition. Bit 7 of this register will be set when an overflow occurs.

Timer Interrupt Flag Register 2 (TFLG2) - address $004F

7	6	5	4	3	2	1	0
TOF	0	0	0	0	0	0	0

Only bit 7 is used. TOF (Bit 7) will be set when overflow of TCNT occurs

0 – Timer has not overflowed
1 - Timer has overflowed

To reset the TOF bit in this register write a 1 to it.

```
bset  TFLG2, $80
```

Application Note
Written by David Lloyd
Computer Engineering Program
Humber College