Chapter 8

Interrupts

Jacki is a professor of Computer Engineering. She is teaching Microcomputers I right now. Jacki knows that Ben, one of her students from a different class, will come and see her anytime during her lecture. How can Jacki become aware of Ben (or anybody else) waiting outside the classroom? Here are two techniques to make this happen:

- Jacki opens the door, say, every 20 minutes, and looks outside to see if anybody is waiting for her. If so, she talks to Ben, returns to the classroom, and continues her lecture.
- Ben knocks on the door as soon as he gets there; Jacki stops her lecture, talks to Ben, and then returns to the classroom to continue the lecture.

In the first scenario, Jacki must stop her lecture frequently; but will she find somebody waiting for her on every occasion? Probably not. This means some waste of her class time. Additionally, Ben has to wait for 10 minutes on average before Jacki opens the door and sees him waiting. These two issues are resolved in the second scenario, are they not? In return, it needs a mechanism to knock on the door.

The CPU in a microcomputer system looks like the instructor in the above classroom example: The instructor interacts with the visitors outside the classroom, while the CPU interacts with the devices outside the CPU.

Example 1. Using the Dragon 12+ board, design a digital system to toggle LED2 when pushbutton 2 (PB2) is pressed.

The hardware is shown in Figure 1. The CPU needs to find out if a pushbutton is pressed. Similar to the above classroom example, there are two approaches to this problem:

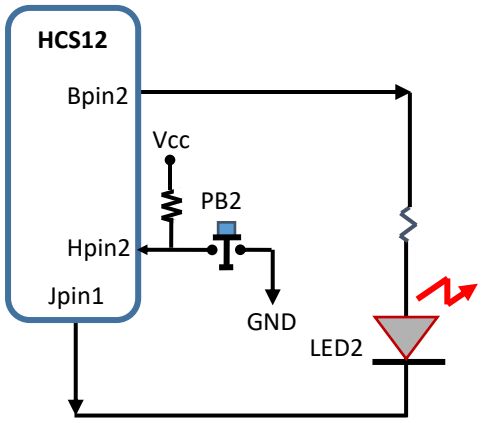

Figure 1. The hardware used in Example 1.

- The CPU stops its normal work frequently and reads Hpin2 driven by the PB2 to see if it is pressed. If it is, the CPU flips the LED by inverting the logic value on Bpin2 and then continues its normal work.

This technique is analogous to the following:

- Jacki opens the door, say, every 20 minutes, and looks outside to see if anybody is waiting for her. If so, she talks to Ben, returns to the classroom, and continues her lecture.

or

- When the pushbutton is pressed, the logic level on Hpin2 changes from 1 to 0, *notifying* the CPU that the pushbutton is pressed. As a result, the CPU toggles the LED by inverting the logic value on Bpin2 and then returns to the interrupted program (its main job). Soon, you will see more about this notification.

This technique is analogous to the following:

- Ben knocks on the door as soon as he gets there; Jacki stops her lecture, talks to Ben, and then returns to the classroom to continue the lecture.

The first technique is called *I/O Polling*, and the second is *Interrupting the CPU* or simply *Interrupt*.

I/O polling: The CPU works on its main job, stops it frequently, and reads (checks) the pushbutton; if pressed, the CPU toggles the LED. The CPU keeps watching the pushbutton in an infinite loop if there is no main job. This is equivalent to the classroom example, where Jacki is not lecturing; she sits there and checks the hallway frequently. You learned this technique in Chapter 7.

Interrupt: The CPU works on its main job until the pushbutton is pressed, *interrupting* the CPU. The CPU (*usually*) completes the execution of the current instruction[1] and then jumps to a special subroutine called the *Interrupt Handler* or the *Interrupt Service Routine* (ISR). The appropriate service is provided in this subroutine, i.e., the LED is toggled. And then, the CPU returns to the interrupted program. If there is no main job, the CPU stays in an infinite empty loop until it is interrupted; this is equivalent to the classroom example, where Jacki is not lecturing; she sits there until Ben knocks on the door. A big picture of the interaction between the interrupted program and the ISR is graphically shown in Figure 2. In this chapter, we will go over the details of this interaction:

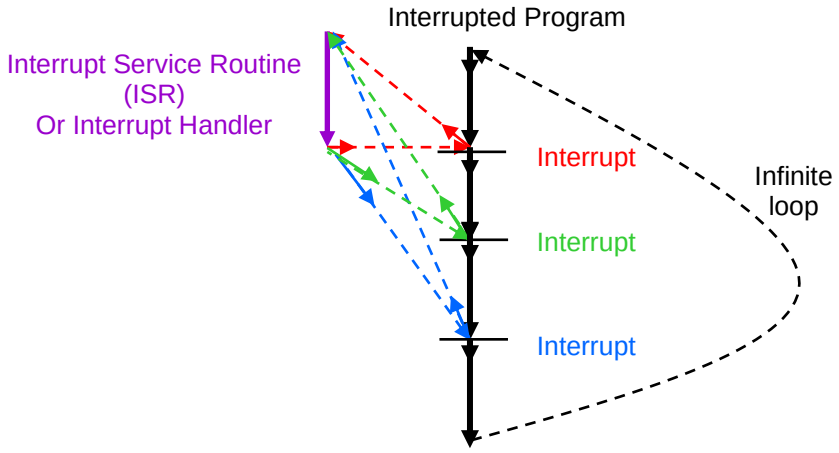

Figure 2. Graphical representation of the interaction between the interrupted program and ISR

Unlike regular subroutine calls[2],
Interrupts are *asynchronous*, i.e., they may happen at any time (at unknown times from your program's point of view), and
You cannot call ISRs in your programs, so ISRs communicate with their outside world differently than regular subroutines, as you will see in this chapter.

Back to the classroom example, Jacki can be interrupted by different interrupt sources, such as:

- A visitor behind the door (as you saw above),

[1] This order of steps may be misleading to a hardware designer. It makes more sense if they imagine it this way: the hardware checks to see if there is an interrupt request when the CPU is done with the current instruction.

[2] Software interrupts are an exception. They are not covered in this edition of the textbook.

- A phone call,
- A phone alarm.

Similarly, the CPU may also be interrupted by different *sources*. In the HCS12, PORT H, as an example, is one interrupt *source*.

Definition: An interrupt *source* has one or more members. Call each member one interrupt *cause*. Each casue can interrupt the CPU. All the causes of the same source share the same ISR. (i.e., an interrupt *source* has only one ISR.)

The eight pins of PORT H are the members of source PORT H. So, they share the same ISR. **Each Hpin can cause an interrupt when a *signal transition* occurs on that pin.** A signal transition means a 1 to 0 (↓) or 0 to 1 (↑) change in the logic level. For example, when you push the pushbutton in Figure 1, a 1 to 0 (↓) transition occurs; when you release it, a 0 to 1 (↑) transition happens, but which one interrupts the CPU? *The choice is yours*: using a register called PPSH, you can decide which transition (↓ or ↑) should cause an interrupt on each pin as explained here:

Register PPSH, **P**ORT **P**olarity **S**elect Register of PORT **H**, is located at $0265. Each bit in this register is associated with one pin of PORT H. If bit i ($0 \le i \le 7$) of this register is set to 1, then the rising edges on pin i will cause an interrupt; otherwise, the falling edges.

Example 2. The following two instructions make Hpin0 and Hpin1 sensitive to rising and falling edges, respectively. The sensitivity of the rest of the pins will not be affected because the PPSH bits associated with these pins remain unchanged. See Figure 3:

```
bset PPSH, 1   ; PH0 becomes sensitive to the rising edges
bclr  PPSH, 4  ; PH2 becomes sensitive to the falling edges
```

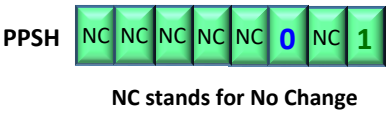

Figure 3. PPSH in Example 2

How the CPU differentiates between the interrupts caused by two pins of PORT H

Let us say one of the Hpins interrupts the CPU. No matter which pin it is, the CPU will jump to the same ISR as all the Hpins are members of the same interrupt source. Then, how can the CPU find out which pin was the cause of the interrupt? The question is answered by another 8-bit control register called **P**ORT **I**nterrupt **F**lag Register **H** or PIFH, located at $0267. When pin i ($0 \le i \le 7$) of PORT H interrupts the CPU, bit i of register PIFH is set to logic 1. In the shared ISR, the CPU has to check this register to figure out who interrupted the CPU and then take the necessary actions accordingly. This looks like a tiny I/O polling happening in the ISR.

Note that in the ISR, the programmer must reset the interrupt flag of the pin that is being serviced (in the ISR); otherwise, the asserted flag will be considered another interrupt request by the same pin right after the CPU exits the ISR (although that pin has already received the service). Therefore, the CPU will be interrupted again and again! To reset an asserted bit in the PIFH, you need to *send a logic 1* to that bit:

Example 3. The following instruction **resets** the interrupt flag of Hpin1 in register PIFH:

```
bset    PIFH, %00000010          ; reset interrupt flag of PH1
```

Figure 4 shows the PIFH after this instruction executes:

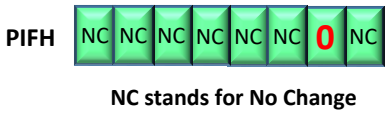

Figure 4. PIFH in Example 3

Remember:
The peripheral devices (hardware) set the bits of PIFH.
The ISR (software) resets the bits of PIFH.

Can the programmer prevent the CPU from interruption?
The answer is Yes: The fourth bit from the left in the CCR is called the I flag (Interrupt-Enable Flag), as shown in Figure 5. For an interrupt to be accepted, this *active-low* flag must be asserted, i.e., reset to 0. So, I = 0 is a necessary condition to interrupt the CPU; in other words, it is the master or global enable flag for (almost) every interrupt cause.

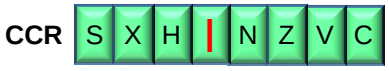

Figure 5. The I flag in the CCR must be 0 to recognize an interrupt

Using the following two pseudo instructions, you can reset or set the I flag; in other words, let the interrupt requests be accepted or ignored:

- cli ; Clear (assert) the I flag (enable interrupts). The true instruction is andcc #$EF.
- sei ; Set (deassert) the I flag (disable interrupts). The true instruction is orcc #$10.

Exception: When the HCS12 accepts an interrupt and jumps to the ISR, the I flag is automatically set to 1 by the hardware, disabling the interrupt system, which remains disabled until the CPU returns to the interrupted program. Then the I flag is reset by the hardware. This means that the HCS12 does not support *nested* interrupts. Although no interrupt source can interrupt the running (current) ISR, interrupt requests will still be *recorded* (in register PIFH as explained above) and hence called *pending*.

The second necessary condition to let an Hpin interrupt the CPU
Suppose that you want to allow Hpin4 to interrupt the CPU but not Hpin7. Here is how you can make it happen:

PORT H has another 8-bit control register called **P**ORT **I**nterrupt **E**nable Register **H** (PIEH), located at $0266. Bit i (0 ≤ i ≤ 7) of this register is the active-high *local* interrupt-enable bit for pin i. So, if you reset bit i of this register, pin i would not be able to interrupt the CPU even if the I flag (in the CCR) was asserted (although the interrupt request is still recorded in register PIFH). To let pin i of PORT H interrupt the CPU, you must set bit i of register PIEH and reset the I flag in the CCR.

Example 4. The following instruction will place binary value 0000 0101 in register PIEH as shown in Figure 6:

```
movb  #%00000101, PIEH;
```

Figure 6. Interrupt Enable Register of PORT H (PIEH)

This would allow Hpin2 and Hpin0 to interrupt the CPU (provided that the I flag in the CCR was also asserted) while the rest of the pins would be interrupt-disabled.

What happens when the CPU is interrupted
The following steps (listed in no specific order) are taken by the *hardware* when the CPU is interrupted:

- The I flag in the CCR is deasserted, i.e., it is set to 1, as shown in Figure 7; this way, the interrupt system is disabled while the CPU is running the ISR, and remains disabled until the ISR returns to the interrupted program.

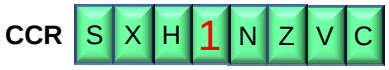

Figure 7. Hardware sets I flag to logic 1 disabling interrupt system when CPU jumps to ISR.

- The return address and the CPU's registers are saved in the stack: Figure 8a shows the stack before the CPU jumps to the ISR, and Figure 8b shows the stack after that. The registers will be restored upon the return to the interrupted program, as you will see shortly.

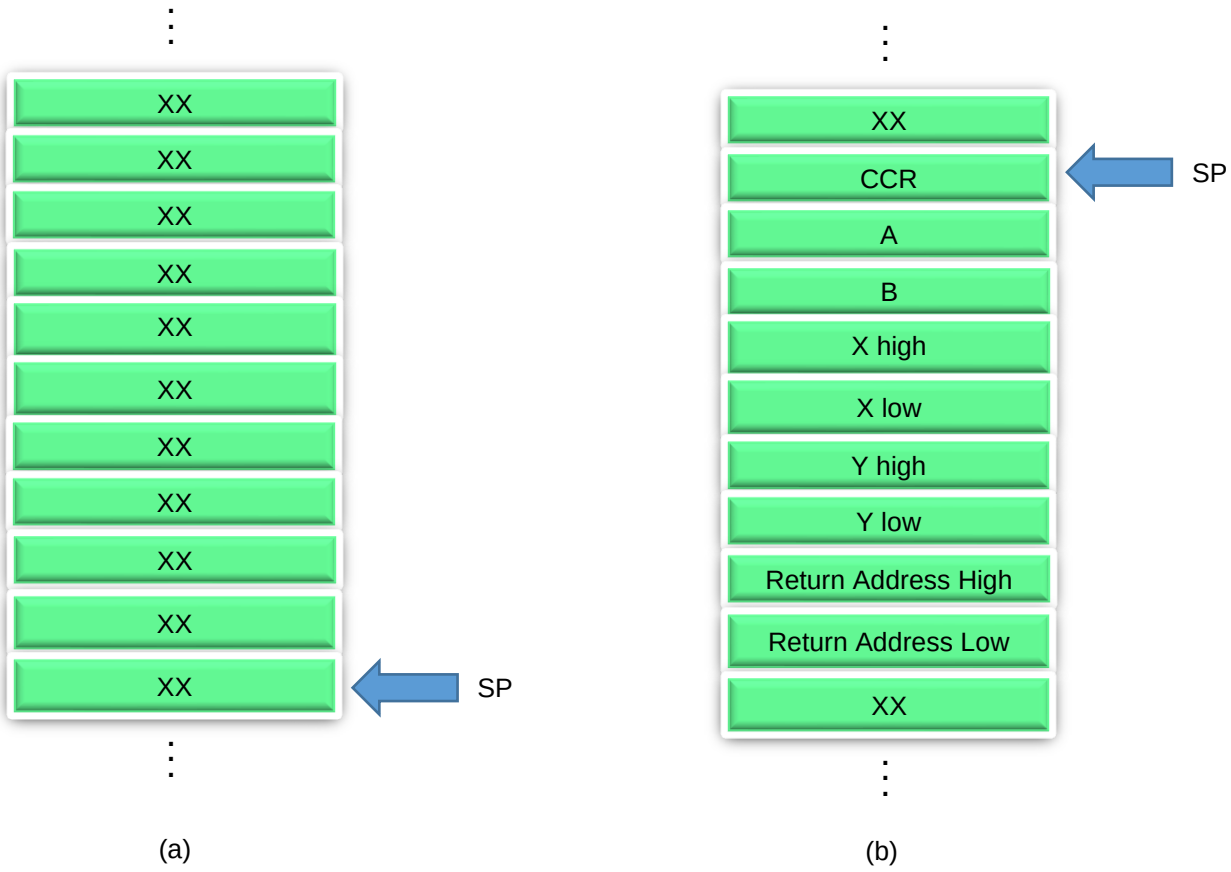

Figure 8. Stack (a) before the interrupt happens, (b) after the interrupt happens.

- CPU jumps to the ISR, i.e., PC <= Address of ISR.

Question: How does the CPU get the address of ISR?
The following section answers this question:

Interrupt vectors and vector addresses

Similar to regular subroutines, you should label the first instruction of each ISR. The associated address (the numeric value of this label) is called the *Interrupt Vector* of the corresponding interrupt source. In the HCS12, each interrupt source, such as PORT H, has its specific *Vector Address* (or address of address); this is the address of the Interrupt Vector. When an interrupt occurs by a member of an interrupt source, the CPU grabs the ISR address from the corresponding Vector Address, i.e., the PC becomes the contents of the Vector Address. The Vector Address of PORT H is $FFCC. So, location $FFCC *points* to the first intrusion of PORTH's ISR. Figure 9 shows the Vector Address, Interrupt Vector, and ISR location of PORT H graphically. Here we *assume* that the ISR is sitting at $5000:

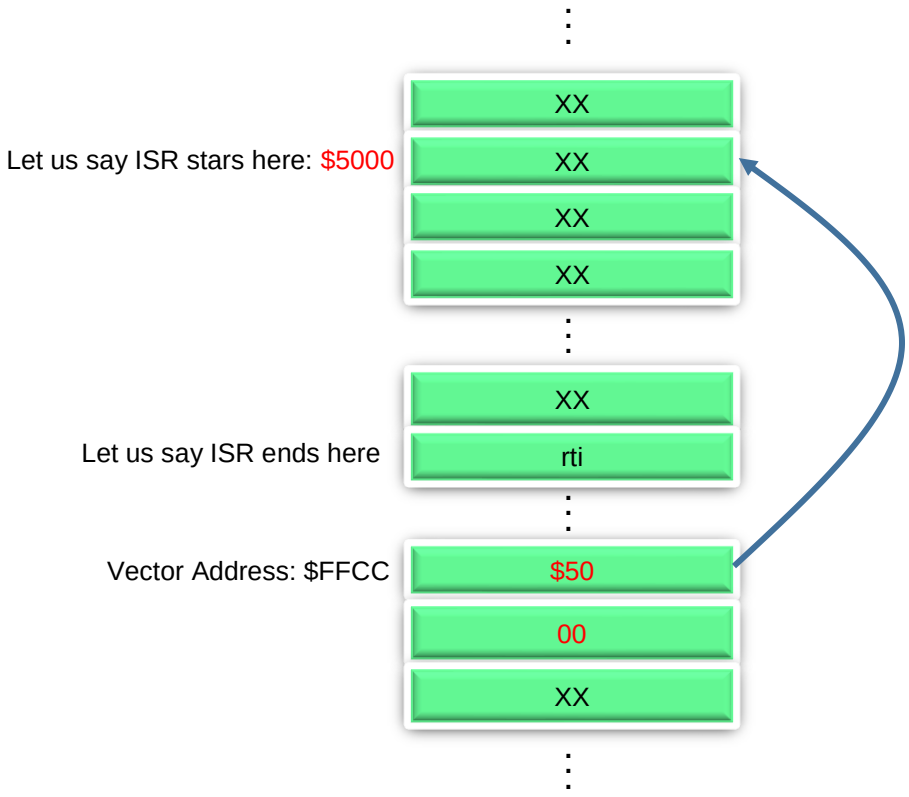

Figure 9. Vector Address, Interrupt Vector, and ISR location for PORT H.

Example 5. Let us say the name of a hypothetical ISR for PORT H is myISR. The first instruction of this ISR, as an example, is shown below:

```
myISR:          ; ISR starts here
    ldx  ...    ; this is the first instrument of myISR (just an example)

    ...
```

Use the following two lines at the **end** of your code or at the **end** of the data segment to place the Interrupt Vector (the address of myISR) at address $FFCC:

```
    org    $FFCC    ; choose vector address

    dc.w   myISR    ; seat the interrupt vector at vector address: $FFCC is the address of the address of ISR
```

Note: If you place any data or instruction after the above two lines, you must properly re-initialize the location counter. ◊

In some processors, unlike the HCS12, the CPU knows each interrupt source's Interrupt Vector (instead of the Vector Address). Now, you must start your ISR at this known address.

How to return from the ISR to the interrupted program
Use the instruction rti (**R**e**T**urn from **I**nterrupt) at the end of your ISR. This instruction will take the following steps listed in no specific order:

- Restores the CPU's registers (stored in the stack when the interrupt was accepted),
- Loads the PC with the return address (stored in the stack when the interrupt was accepted),
- Restores the SP, as shown in Figure 8a,
- Enables the interrupt system by resetting the I flag (located in the CCR).

In summary, all Hpins share the same Vector Address and, therefore, the same Interrupt Vector, hence the same ISR. In the ISR, check register PIFH and see which bit is set to 1. This tells you which Hpin interrupted the CPU. Then reset the asserted bit of PIFH to avoid multiple interrupts by the same Hpin. In the case of multiple asserted bits in PIFH, hence multiple requesting Hpins, choose the one with the highest priority, and provide service accordingly. And finally, use instruction **rti** at the end of your ISR to properly return to the interrupted program.

Interrupt priority and pending interrupts
Although (almost) no cause can interrupt the running (current) ISR, as mentioned before, interrupt requests are still recorded, hence called *pending*. Let us say the HCS12 is servicing Hpin2 in the ISR, during which pins 6 and 4, for example, make an interrupt request. The ISR is not interrupted (as the I flag = 1); however, bits 6 and 4 of resister PIFH will still be asserted (set to 1) due to interrupt requests made by pins 6 and 4. When the CPU is done with the current ISR, the CPU will be interrupted by the pending interrupts. Here is the question: which pending interrupt (the interrupt caused by pin 4 or pin 6 in this example) will be serviced first? The answer is it is your choice: the pin you check first (in the ISR) is serviced first.

You should distinguish between "interrupt is *requested* (by an interrupt cause)" and "interrupt is *accepted* (by the CPU)". When an interrupt is requested, the interrupt flag of the interrupt cause (in register PIFH) is asserted; however, the CPU will not accept the interrupt unless both of the following bits are asserted:

- The Interrupt Enable bit (in PIEH) of the interrupt cause (see Figure 6), and
- The I flag in the CCR.

The interrupt is called *pending* when requested but not yet accepted. The CPU jumps to the ISR should the interrupt request be accepted and vice versa. When we say "an interrupt *occurs*", or "the CPU or a program is *interrupted*", we mean the interrupt is requested and accepted.

Now consider a different scenario: Let us say while the CPU is servicing interrupt source A, two interrupt sources, B and C, make an interrupt request. The question is, which one, B or C, will be serviced first when the CPU is done with source A? Here is the answer: the interrupt sources in the HCS12 have fixed priority levels: the higher the vector address, the higher the priority level, with one exception explained soon. *Therefore, the pending interrupt that has the highest priority among all the pending interrupts will be serviced first*.

You may choose *one* of the interrupt sources, call it X, and provide it with the highest priority. To do so, you must seat the lower byte of the Vector Address of source X in register HPRIO of the HCS12, the **H**ighest **P**riority **R**egister located at $001F.

Example 6. Write an ISR to toggle LED0 on the Dragon 12+ board every time pushbutton 0 is pressed. The hardware is shown in Figure 10:

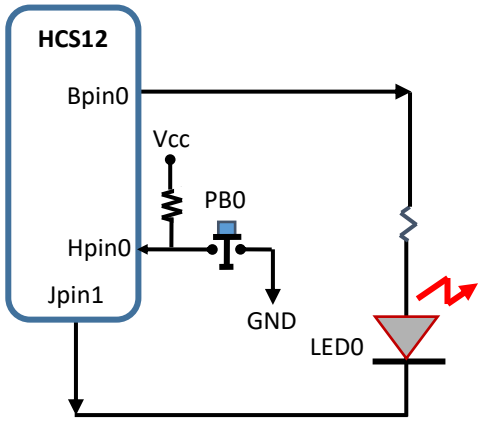

Figure 10. The hardware used in Example 6.

The initialization, as well as the infinite loop, are shown below. The only new instruction is **cli**, which enables the interrupt systems, i.e., resets the I flag in the CCR:

```
    bset DDRP, $F       ; Config. lower PORT P as output
    bset PTP,  $F       ; turn off 7-segment displays
    bset DDRJ, 2        ; configure Jpin1 as output
    bclr  PTJ,  2       ; Enable LEDs
    bset DDRB, 1        ; configure PB0 as output
    bset PORTB, 1       ; trun LED0 ON
    bclr  DDRH, 1       ; configure Hpin0 as input
    bclr  PPSH, 1       ; interrupt on the falling edges
    bset PIEH, 1        ; enable interrupt on Hpin0
    cli                 ; enable interrupts (in CCR) This is a new instruction
forever:
    bra   forever       ; this is the empty infinite loop
```

Remember from Chapter 7 that a typical microcontroller program is made up of an initialization portion followed by an infinite loop. In the current chapter (8), we learned that the infinite loop may be interrupted by different interrupt causes, i.e., now there are one or more ISRs (in the code you develop), out of which the right one will be chosen and executed when the infinite loop is interrupted.

It is important to note that in this example, the CPU has no job to do other than controlling the LED. *This is why the infinite loop is empty*. Here is the only ISR in this example, which controls the LED:

```
ledISR:                 ; ISR starts here
    ldx     #9000       ; wait to debounce pushbutton
wait:
    dbne    X, wait     ; "wait"  here (3 cycles)
    brset   PTH, 1, done ; "and see" here
    ldab    PORTB       ; read PORT B (read)
```

```
    eorb    #1              ; toggle bit 0 (modify)
    stab    PORTB           ; toggle LED0 (write)
done:
    bset    PIFH, 1         ; reset Hpin0 interrupt flag
    rti                     ; return from ISR, this is a new instruction
    org     $FFCC           ; choose vector address
    dc.w    ledISR          ; $FFCC is the address of the address of ISR
```

The only new instruction here is **rti**, by which the CPU returns to the interrupted program. Also, in the last two lines, two assembler directives are used to initialize the Vector Address of PORT H.

Vectored interrupts virus non-vectored interrupts

In a *vectored* interrupt system, each interrupt source (such as PORT H in the HCS12) has its own Vector Address and, therefore, its own Interrupt Vector, hence its own ISR. When an interrupt occurs in such a system, the interrupt source introduces itself to the CPU. This way, the CPU (the hardware) identifies the source (without any help from the software), jumps directly to the associated Vector Address, grabs its content, which is the Interrupt Vector, and then takes the next instruction (which is the first instruction of the ISR) from there.

In a non-vectored system, on the other hand, there is only one Vector Address, therefore, one Interrupt Vector, hence only one ISR. In such a system, the CPU would jump to the *same* ISR when an interrupt was accepted, regardless of the cause. Note that in non-vectored systems, the interrupting source still introduces itself to the CPU; however, the hardware by itself cannot identify the cause; a piece of code is required to figure out who interrupted the CPU. You (the programmer) should place this code in the only ISR you have in such a system. This code checks all the interrupt flags, identifies all the interrupt causes, and decides which cause should be serviced first in case of two or more interrupt requests. This search looks like I/O polling inside the ISR; however, now there is definitely at least one interrupt request, unlike regular I/O polling, where the CPU does not know whether or not an I/O device is waiting for service. Based on this definition, the HCS12 is not 100% vectored nor 100% non-vectored.

Maskable interrupts (MI) versus non-maskable interrupts (NMI)

The I flag in the CCR is the master (or global) interrupt-enable flag for the HCS12, meaning that if it was deasserted (I = 1), no interrupts would be accepted (when they were requested) but remember they are still recorded in the associated interrupt flags such as register PIFH for PORT H; see Figure 4. These interrupts are called *maskable* interrupts, as they can be ignored (or masked) under your program control by simply deasserting the I flag in the CCR (I = 1). On the other hand, microprocessors usually have pins for the *non-maskable interrupts*. The master interrupt-enable flag (I) can no longer prevent interruptions caused by these pins. This is why these interrupts are called *non-maskable*. Therefore, the ISR of a maskable interrupt source (such as PORT H in the HCS12) CAN be interrupted by a non-maskable interrupt. Generally called XIRQ, the non-maskable interrupt pin in the MC9S12DG256B (the microcontroller on the Dragon 12+ board) is PE0. A non-maskable interrupt is requested with logic zero at this pin. The X flag, bit 6 in the CCR, is the active-low interrupt-enable flag for this pin. When a non-maskable interrupt is being serviced, the X flag is pulled up to disable more non-maskable interrupts (otherwise, the low-level voltage on PE0 will keep interrupting the CPU). During the non-maskable ISR, the I flag is also pulled up or remains up to ignore maskable interrupts as well. Once the non-maskable ISR ends (by an rti instruction), both X and I flags are restored, i.e., X is asserted (X = 0), and I returns to the value it had before the non-maskable interrupt occurred. We use non-maskable interrupts when the response time is *critical*.

While the HCS12 microcontroller is being reset, and to disable all types of interrupts, X and I flags are deasserted and remain deasserted until the reset process is over. Then you can enable the non-maskable

interrupt by resetting the X flag (X = 0), but once you reset it, you cannot set (deassert) it under your program control anymore.

Different causes may *reset* the HCS12. Resets are not maskable either. The reset process is not covered in this edition of the textbook. The future editions will also cover more subtopics, such as software interrupts, exceptions, multitasking/multithreading, and more.

Real-Time Interrupt (RTI)

The second interrupt source covered in this textbook edition is called the **R**eal-**T**ime **I**nterrupt, or RTI for short. Unlike PORT H, the RTI does not need an external cause to interrupt the CPU. When it is set up properly, the RTI interrupts the CPU *periodically*. The *timeout period*, or simply, the *period*, is programmable, as you will see in this section.

Example 7. The Multiplexing/Demultiplexing algorithm (introduced in Chapter 7) to display four digits on the four 7-segment displays (on the Dagon 12+ board) is shown here again for convenience:

Do this forever:

- Turn off all the displays.
- Send the next digit to Port B.
- Turn on the next display.
- Wait for, say, two milliseconds

Here is the translation of the above algorithm into assembly language. The number of E-cycles for each instruction is also shown next to that instruction:

```
forever:                ; display
 ldy     #buffer        ; 2 cycles, initialize buffer pointer
 ldab    #%11101110     ; 1 cycle, initialize cathode
 sec                    ; 1 cycle, carry bit is not affected by load/store
nextDigit:
 movb    #$F , PTP      ; 4 cycles, Turn off all 7-segment displays
 movb    1, Y+, PORTB   ; 5 cycles, read next buffer location and send it to display
 stab    PTP            ; 3 cycles, turn the current display on
 jsr     delay2
 rolb                   ; 1 cycle, get ready for the next digit
 bcs     nextDigit      ; 3/1 cycles, move on to next digit if any
 bra     forever        ; 3 cycles
```

Inner loop delay with successful jump = 4 + 5 + 3 + 1 + 3 = 16 cycles

Inner loop delay with unsuccessful jump = 4 + 5 + 3 + 1 + 1 = 14 cycles

Outer loop delay = 2 + 1 + 1 + 3 + successful inner delay = 7 + 16 = 23 cycles

Outer loop delay = 2 + 1 + 1 + 3 + unsuccessful inner delay = 7 + 14 = 21 cycles

Average outer loop delay = (3 x successful outer delay + 1 x unsuccessful outer delay)/4 = (3 x 23 + 21)/4 = 22.5 cycles = 22.5 x 41.7 ns ≈ 1 microsecond.

In each iteration, the CPU spends around 23 E-cycles or under 1 microsecond to take the first three steps of the algorithm and then waits for two milliseconds, i.e., the wasted time (2 milliseconds) is 2000 times as large as the useful time (1 microsecond), a big waste of time! This is graphically shown in Figure 11:

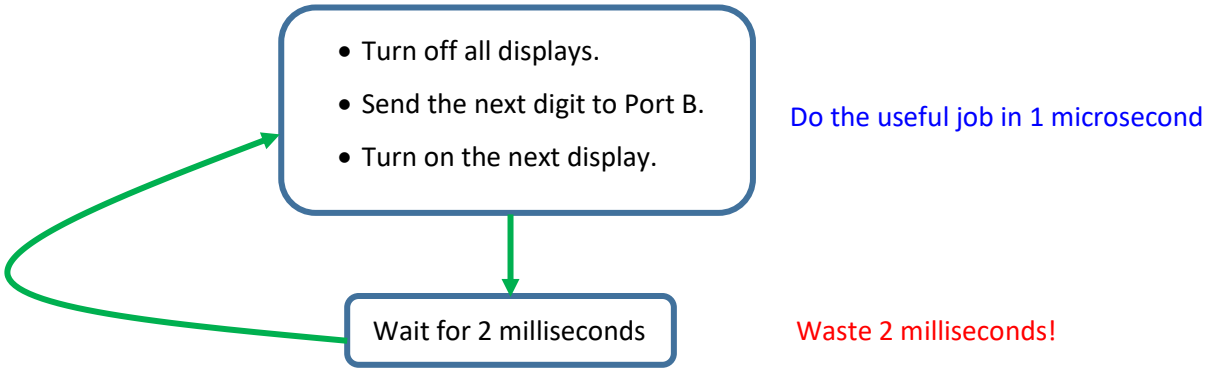

Figure 11. Graphical representation of wasted time in Example 7

Now let us set up the RTI to interrupt the CPU every two milliseconds and take the first three steps of the Multiplexing/Demultiplexing algorithm in the ISR (which lasts for only 1 microsecond every 2 milliseconds). Then the CPU can be scheduled to perform another useful job, if any, in the infinite loop during the rest of the 2 milliseconds that were entirely wasted in Figure 11. This scenario is graphically illustrated in Figure 12:

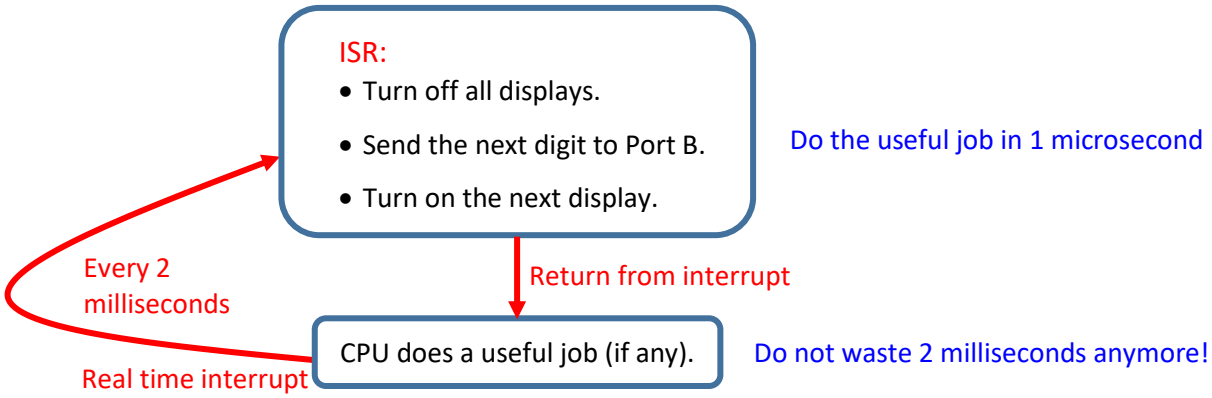

Figure 12. Graphical representation of the scenario in Example 7 when RTI is used

There are three 8-bit control registers playing the main roles in setting up the RTI:

CRGINT ($0038): CRG Interrupt-Enable Register

CRGFLG ($0037): CRG Flag Register

RTICTL ($003B): CRG RTI Control Register (this is specifically called the "**Control**" register" of RTI)

Where CRG stands for **C**lock and **R**eset **G**eneration block. In this section, you will learn about the relevant bits in these three registers:

The MSb of CRGINT is the active-high local interrupt-enable bit of RTI. See Figure 13:

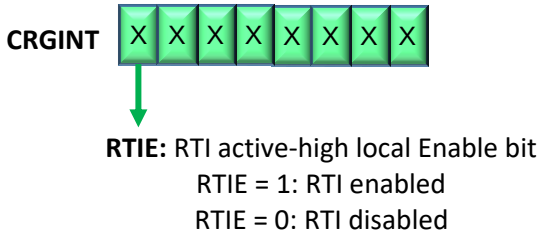

Figure 13. CRGINT register and the RTI's local interrupt-enable bit

The MSb of CRGFLG is the RTI's active-high flag bit. See Figure 14:

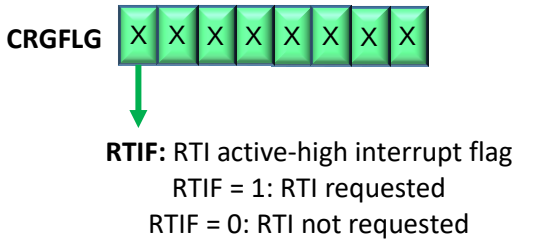

Figure 14. CRGFLG register and the RTI's interrupt flag

The flag bit is pulled up at the end of every 2 milliseconds indicating an RTI request. This flag must be reset in the ISR (similar to the interrupt flags of PORT H,); otherwise, the RTI will keep interrupting the CPU. As you see, the flag bit of RTI is set to 1 by the hardware, and it is reset to 0 with **logic 1** in the ISR (by the software).

Register RTICTL is used to specify the timeout period of RTI. Let us split this register into the 4-bit blue field (the 4 LSbs) and the 3-bit red field (bits 6 : 4), as shown in Figure 15:

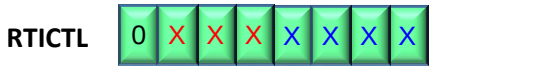

Figure 15. RTICTL control register

Note that the range of the blue field is 0 through 15, and the range of the red field is 0 through 7. The timeout period of RTI is determined as follows:

$$\text{RTI period} = (\text{Blue number} + 1) \times 2^{\text{red number} + 9} \times \text{period of OSCCLK cycle time (0.125 microseconds)} \quad (1)$$

OSCCLK is another onboard clock signal with a frequency of 8 MHz or a period of 0.125 microseconds on the Dragon 12+ board. As you see in Equation 1, (the blue number + 1) is the RTI period multiplier; in other words, the RTI frequency divider.

Example 8. Obtain the largest possible timeout period of RTI:

According to Equation (1), the timeout period is the largest when the blue and red numbers are the largest, i.e., the blue number = 15 and the red number = 7. So, if these two values are plugged in (1):

Larget RTI timeout period = $(15 + 1) \times 2^{7+9} \times 0.125$ microseconds = $16 \times 2^{16} \times 0.125$ microseconds =

$2^{20} \times 0.125$ microseconds = 131072 microseconds ≈ 131 milliseconds

So, to create a 1-second time interval, for example, we need eight back-to-back RTIs while its timeout period has been set up to 131 milliseconds, the largest possible:

8 x 131 milliseconds = 1048 milliseconds ≈ 1 second.

Example 9. Use RTI to turn LED0 ON and OFF periodically with a 2-second cycle time and a duty cycle of 50%. The Vector Address of RTI is $FFF0:

This code has three sections: the initialization, an infinite loop, and the ISR:

```
; Code segment
; ********* Initialization
        movb    #$F, DDRP       ; configure PORT P3-0 as output
        movb    #$F, PTP        ; disable 7-segment displays
        bset    DDRJ, #2        ; configure PJ1 as output
        bclr    PTJ, #2         ; enable LEDs
        movb    #1, DDRB        ; configure PORT B0 as output
        bset    PORTB, 1        ; Turn LED0 ON
        movb    #$7F, RTICTL    ; get max RTI timeout period
        ldab    #8              ; initialize interrupt counter (Reg B)
        bset    CRGINT, #$80    ; enable RTI
        cli                     ; enable interrupt system (master enable)
; ********* infinite loop
forever:
        bra forever
;********* ISR starts here
rti_isr:
        dbne    B, return       ; update interrupt counter, return if eight interrupts not occurred yet
        ldab    #8              ; initialize interrupt counter
; toggle LED0 (Read, Modify, Write)
        ldaa    PORTB
        eora    #1
        staa    PORTB
return:
        bset    CRGFLG, $80     ; reset RTI interrupt flag (send 1)
        rti
        org     $FFF0           ; vector address
        dc.w    rti_isr         ; initialize vector address
```

The infinite loop in this example is empty, as the CPU's only job is to control the LED, which is carried out in the ISR.

The RTI timeout period is set to max (≈131 milliseconds) using the {movb #$7F, RTICTL} instruction. So, we need to toggle the LED every eight interrupts. (8 x 131 microseconds ≈ 1 second). In the above code,

register B is the interrupt counter. It is initialized to 8 and then decremented by 1 in the ISR every time an RTI occurs. When B hits a zero, the occurrence of the eighth RTI is signified, meaning that 1 second has passed, and therefore, the LED should be toggled.

There is a major issue with the above code: Remember, when the RTI is accepted, register B (along with other registers) is stored in the stack and restored at the end of the ISR. More specifically, when the RTI happens for the first time, B = 8 (the initial value); therefore, value 8 is stored in the stack. In the ISR, register B is decremented and becomes 7, the value we need when the next interrupt occurs. Value 7, however, will be overwritten and hence lost when the ISR ends and register B is restored (B ← 8)!

To solve this problem, we use a global variable (call it "counter") as the interrupt counter instead of register B, as shown here:

```
; data segment
counter:
        dc.b    8               ; global location accessible to ISR and the main program
        org     $FFF0           ; vector address
        dc.w    rti_isr         ; initialize vector address

; Code segment
; ********* Initialization, delete the initialization of register B; otherwise, same as before,
            movb    #$F, DDRP       ; configure PORT P3-0 as output
            movb    #$F, PTP        ; disable 7-segment displays
            bset    DDRJ, #2        ; configure PJ1 as output
            bclr    PTJ, #2         ; enable LEDs
            movb    #1, DDRB        ; configure PORT B0 as output
            bset    PORTB, 1        ; Turn LED0 ON
            movb    #$7F, RTICTL    ; get max RTI period
            ;;; delete this line   ldab    #8              ; initialize interrupt counter (Reg B)
            bset    CRGINT, #$80 ; enable RTI
            cli                     ; enable interrupt system (master enable)
    ; ********* infinite loop
    forever:
            bra forever
    ;******** ISR
    rti_isr:
            dec     counter         ; update interrupt counter
            bne     return          ; if 8 interrupts not occurred yet, return
            movb    #8, counter     ; otherwise, initialize interrupt counter
    ; toggle LED0 (Read, Modify, Write)
            ldaa    PORTB
            eora    #1
```

```
        staa    PORTB
return:
        bset    CRGFLG, $80   ; reset RTI interrupt flag (send 1)
        rti
```

***

In the above implementation, the time spent to run the ISR (around 1 microsecond) is part of the ISR timeout period. When register RTICTL is initialized or reinitialized, the RTI timeout period restarts:

Example 10. Subroutine wait500 generates a 500-millisecond delay using RTI. Let us assume that the frequency of OSCCLK is 10.24 MHz. Then RTICTL = $19 will generate a 1-millisecond timeout period:

```
wait500:    ; 500-ms delay
        pshx
        pshc
        movb    #$19, RTICTL    ; restart RTI timeout period
        clr     msec            ; msec is a global variable; it is incremented in the ISR of RTI, so it
                                ; is incremented every 1 ms
wait:
        ldx     msec
        cpx     #500        ; has 500 ms passed?
        bne     wait        ; if not, continue to wait; the subroutine is interrupted by RTI
                            ; while waiting here
        pulc                ; otherwise, prepare to return
        pulx
        rts
```

***

The following is the RTI's ISR. The RTI requests an interrupt every 1 millisecond. The global variable msec is incremented by 1 in the ISR. The wait500 subroutine checks variable msec; when it hits 500, the subroutine returns because the 500-millisecond delay is complete:

```
rtiISR:
        ldx     msec            ; (read) msec is a global variable used as the RTI counter.
        Inx                     ; modify
        stx     msec            ; write
        bset    CRGFLG, $80     ; reset the interrupt flag
        rti
```

***

When the subroutine wait500 starts, we do not know how much time is left to complete the current timeout period of the RTI; it could be anything between 0 and 1 millisecond. To avoid this inaccuracy, we reinitialize RTICTL at the subroutine's beginning, restarting the timeout period; therefore, the first interrupt will occur in 1 millisecond.