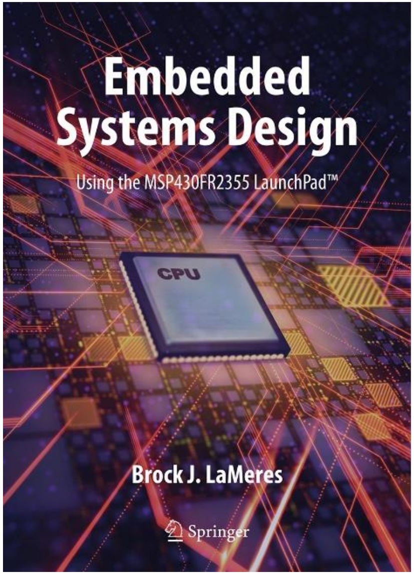
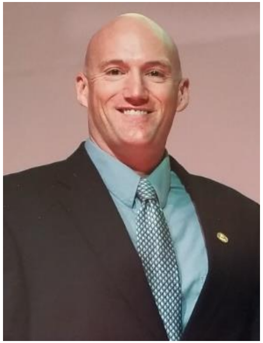
Embedded Systems Design

Chapter 11: Introduction to Interrupts

11.1 The Concept of an Interrupt

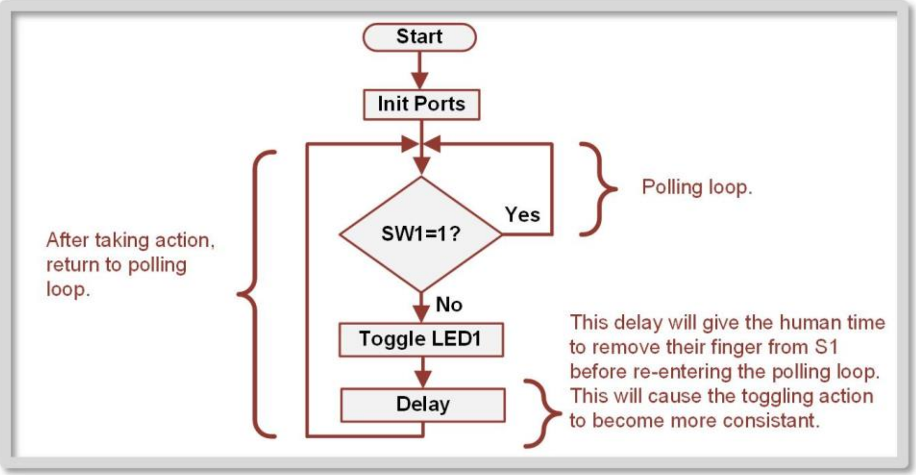




Brock J. LaMeres, Ph.D.

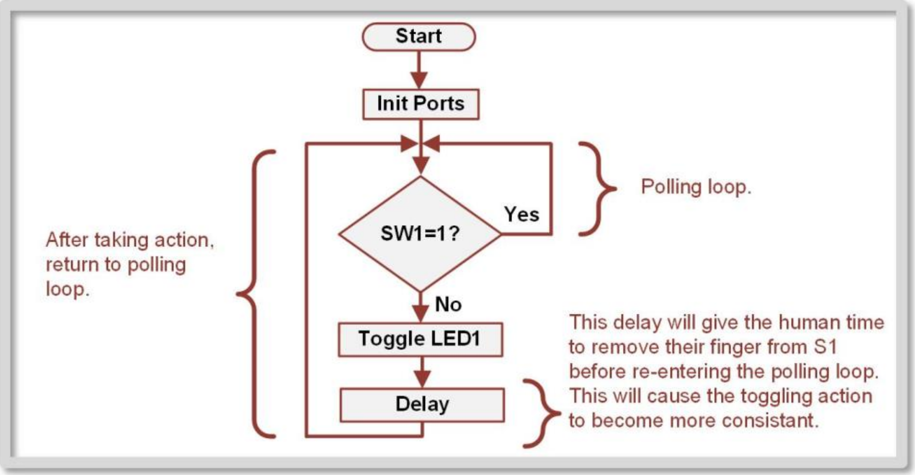
11.1 THE CONCEPT OF AN INTERRUPT

- **The Drawback of Polling** – CPU spends a lot of time executing instructions that did nothing but actively check for an infrequent event.


Start
Init Ports
SW1=1?
Yes
No
Toggle LED1
Delay
Polling loop.
After taking action, return to polling loop.
This delay will give the human time to remove their finger from S1 before re-entering the polling loop. This will cause the toggling action to become more consistant.

11.1 THE CONCEPT OF AN INTERRUPT

- **The Drawback of Polling** – CPU spends a lot of time executing instructions that did nothing but actively check for an infrequent event.



Polling loop.

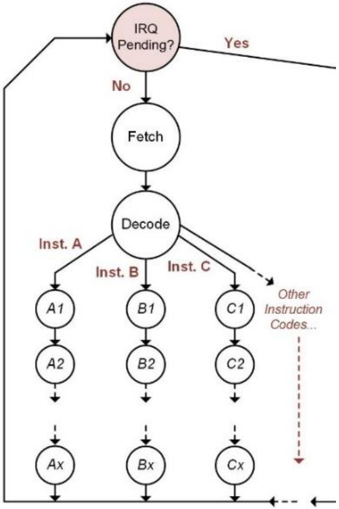
After taking action, return to polling loop.

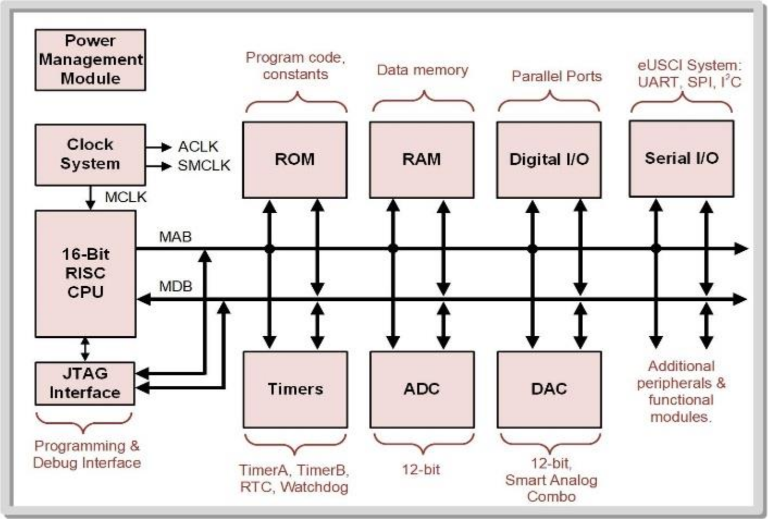
This delay will give the human time to remove their finger from S1 before re-entering the polling loop. This will cause the toggling action to become more consistant.

- **Interrupt (IRQ)** – an approach to dealing with external, asynchronous events by building hardware int o the MCU that handles identifying and prioritizing events to be serviced by the CPU; only reacts one time when a transition is observed.

11.1.1 INTERRUPT FLAGS (IFG)

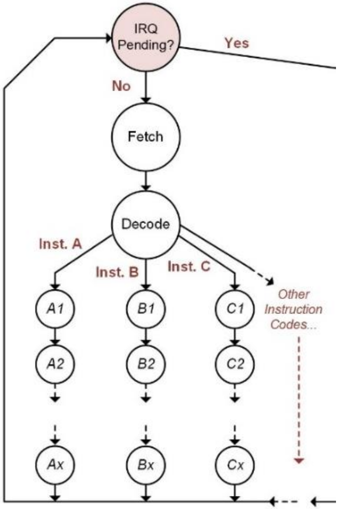
- **Flag** – notifies the CPU that an external event on a peripheral has occurred and action is requested.

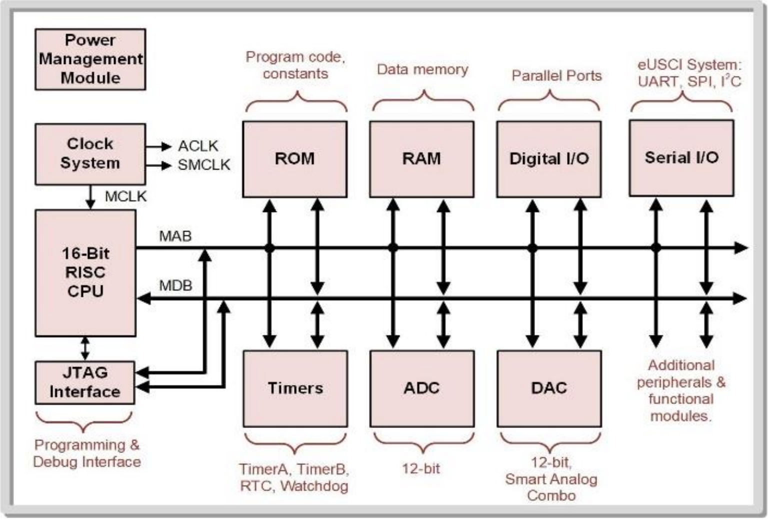




11.1.1 INTERRUPT FLAGS (IFG)

- **Flag** – notifies the CPU that an external event on a peripheral has occurred and action is requested.

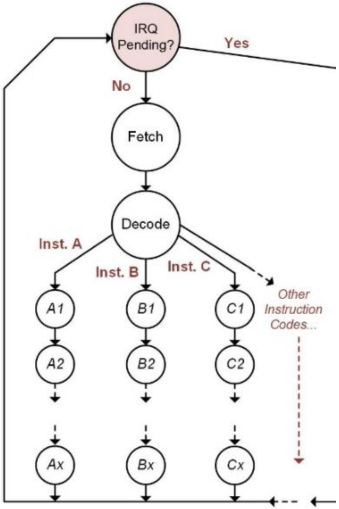


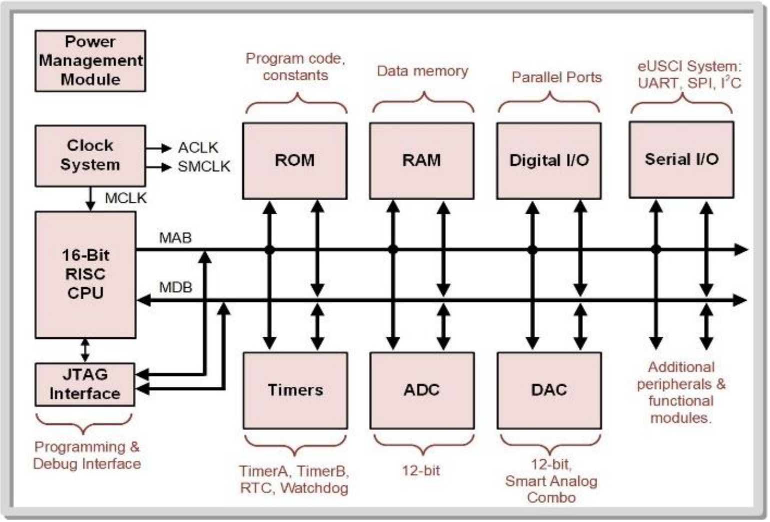


- When a flag is observed, the CPU completes its current instruction and then executes a sequence of instructions that accomplishes the desired action for the peripheral.

11.1.1 INTERRUPT FLAGS (IFG)

- **Flag** – notifies the CPU that an external event on a peripheral has occurred and action is requested.

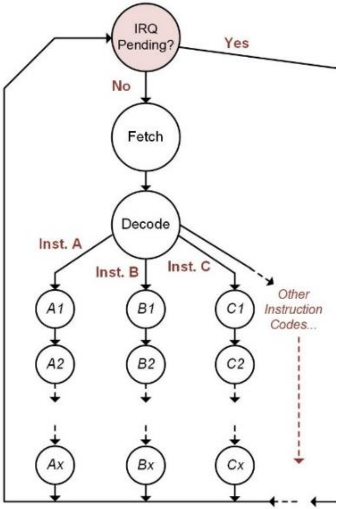


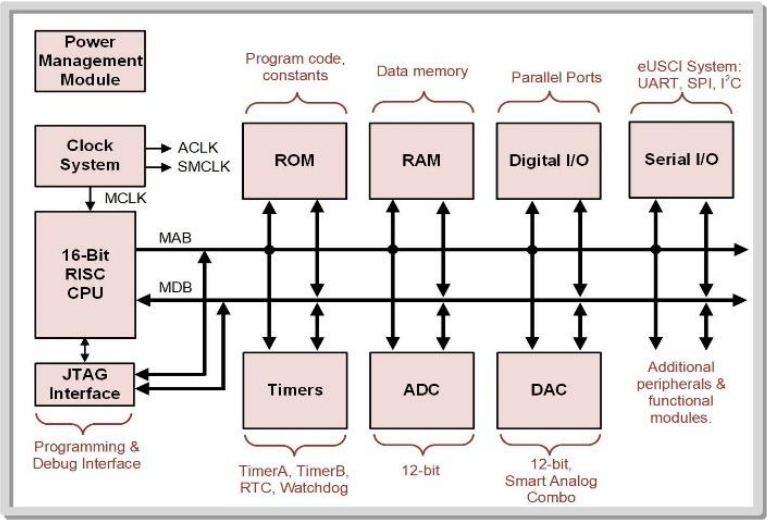


- The term *interrupt* stems from the fact that the CPU takes a break from executing the main program and instead executes instructions specifically for the peripheral event.

11.1.1 Interrupt Flags (IFG)

- **Flag** – notifies the CPU that an external event on a peripheral has occurred and action is requested.

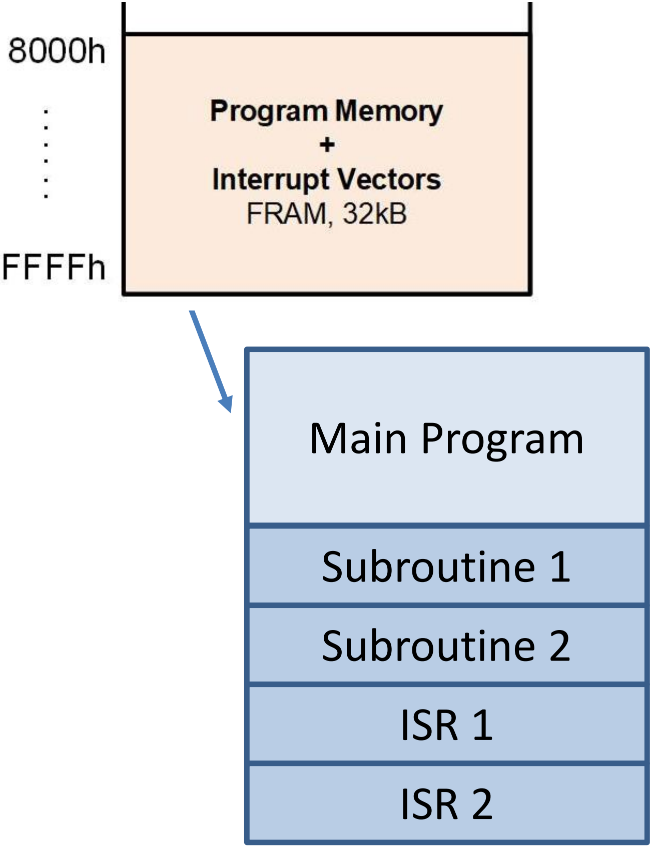




- This process is highly efficient because the CPU does not have to spend execution cycles polling each external system.

11.1.1 INTERRUPT FLAGS (IFG)

- **Interrupt service routine (ISR)** or **interrupt handler** – the code that is executed when an interrupt occurs



8000h

FFFFh

Program Memory
+
Interrupt Vectors
FRAM, 32kB

Main Program

Subroutine 1

Subroutine 2

ISR 1

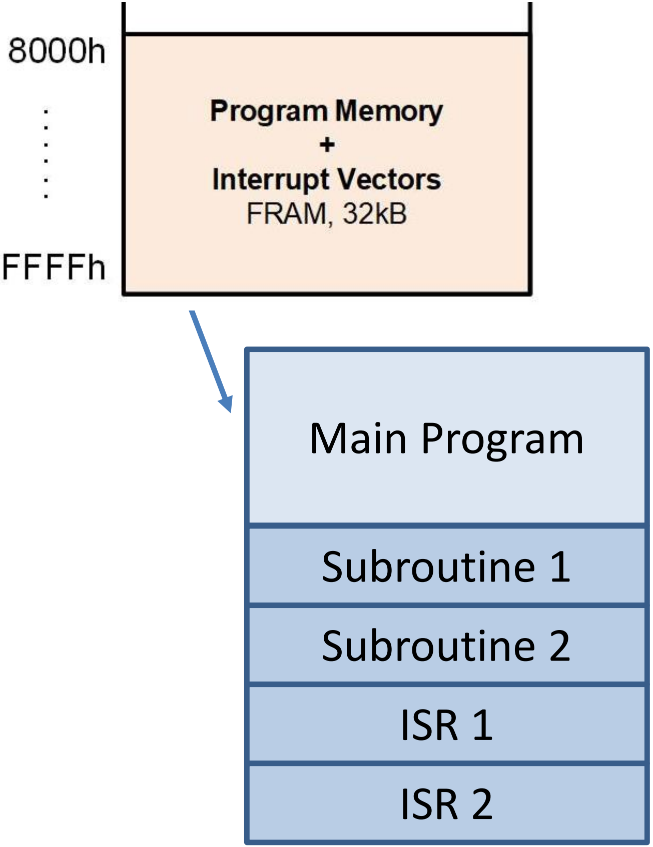
ISR 2

11.1.1 INTERRUPT FLAGS (IFG)

- **Interrupt service routine (ISR)** or **interrupt handler** – the code that is executed when an interrupt occurs

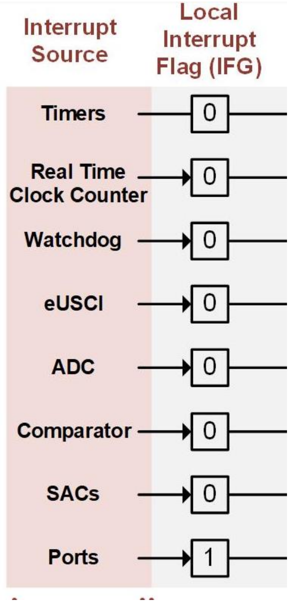
- **Servicing** – when the CPU is taking steps to handle the IRQ.

- **Pending** – when an interrupt has asserted its flag but the CPU has not had an opportunity to service it.

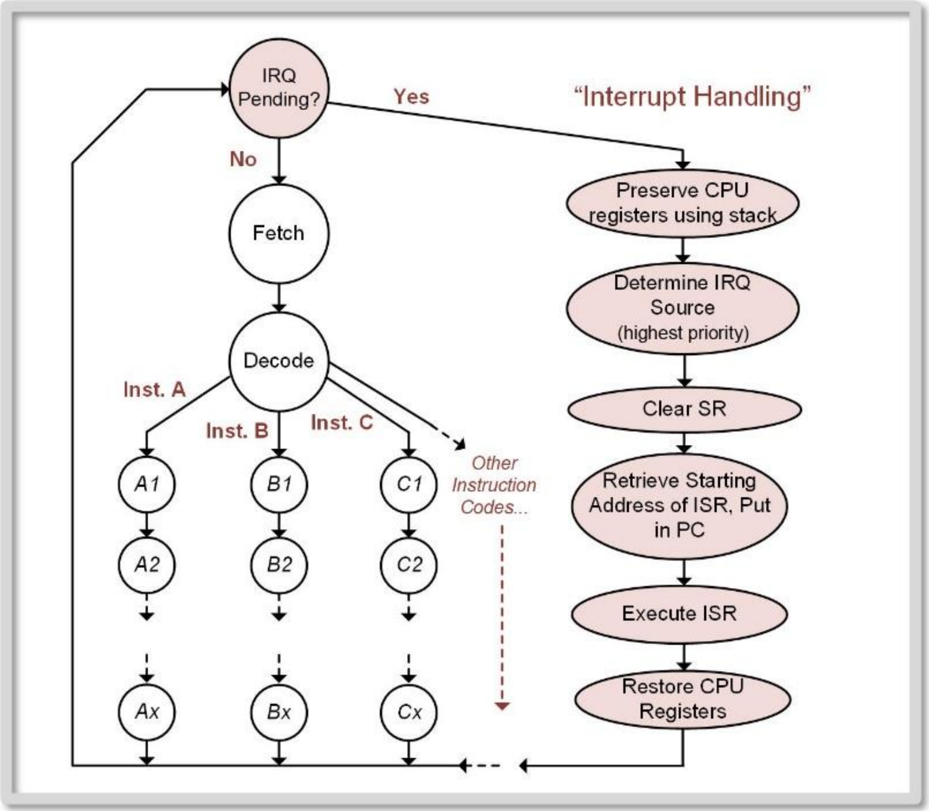


11.1.1 Interrupt Flags (IFG)

- The CPU control unit is designed to handle IRQs.

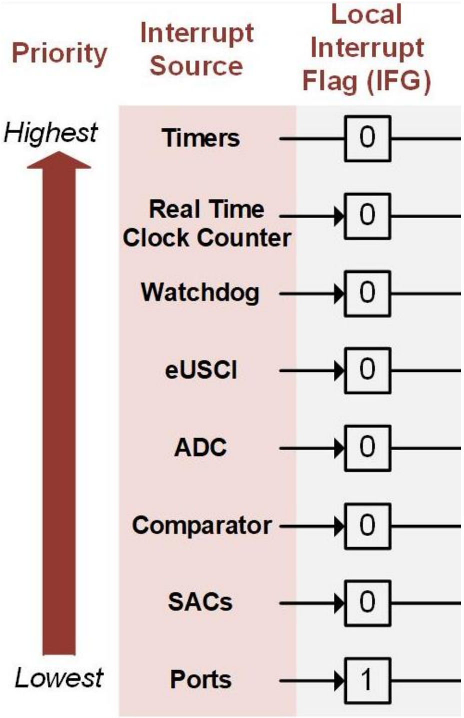


Interrupt Source	Local Interrupt Flag (IFG)
Timers	0
Real Time Clock Counter	0
Watchdog	0
eUSCI	0
ADC	0
Comparator	0
SACs	0
Ports	1


IRQ Pending?
Yes
"Interrupt Handling"
No
Fetch
Decode
Inst. A
Inst. B
Inst. C
A1
A2
Ax
B1
B2
Bx
C1
C2
Cx
Other Instruction Codes...
Preserve CPU registers using stack
Determine IRQ Source (highest priority)
Clear SR
Retrieve Starting Address of ISR, Put in PC
Execute ISR
Restore CPU Registers

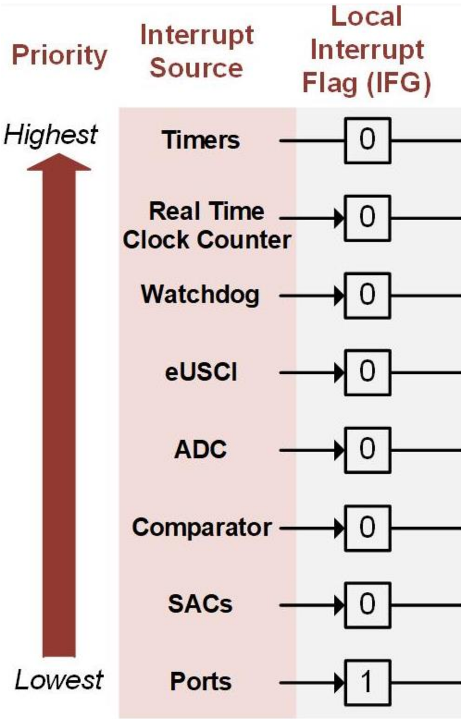
11.1.2 INTERRUPT PRIORITY AND ENABLING

- When the CPU is ready to service an interrupt, it always executes the highest priority peripheral first.



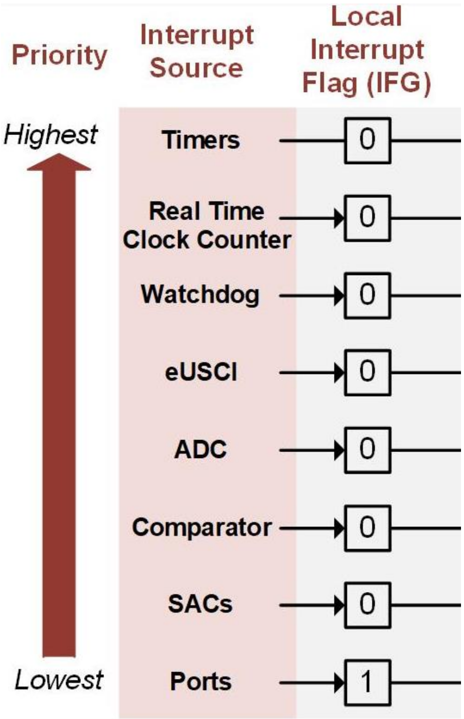
11.1.2 INTERRUPT PRIORITY AND ENABLING

- When the CPU is ready to service an interrupt, it always executes the highest priority peripheral first.
- Once that routine completes, it moves to the next highest priority peripheral that is pending, and so forth.



11.1.2 INTERRUPT PRIORITY AND ENABLING

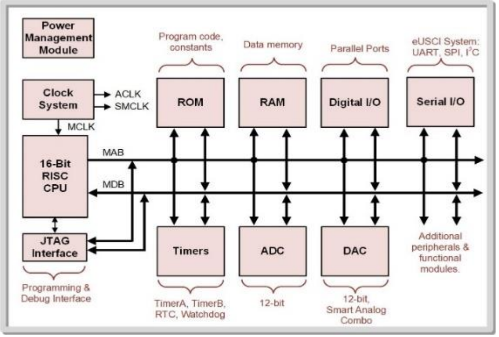
- When the CPU is ready to service an interrupt, it always executes the highest priority peripheral first.
- Once that routine completes, it moves to the next highest priority peripheral that is pending, and so forth.
- Interrupts *can* interrupt other interrupts if they have a higher priority, but some restrictions apply that are described in subsequent sections.



11.1.2 Interrupt Priority and Enabling

- Categories of interrupts:

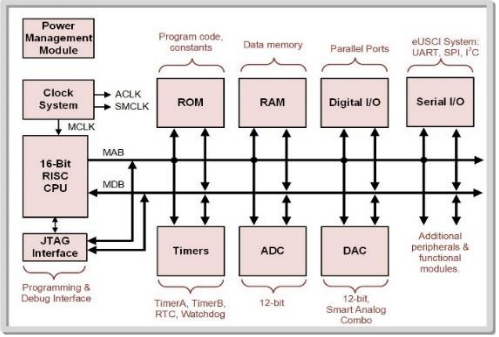
  1) System resets,
  2) Non-maskable interrupts (NMIs),
  3) Maskable interrupts



11.1.2 INTERRUPT PRIORITY AND ENABLING

- Categories of interrupts:

  1) **System resets**,
  2) Non-maskable interrupts (NMIs),
  3) Maskable interrupts



- **System Resets** – highest priority interrupts and are always enabled; causes the MCU to start its operation from the beginning.

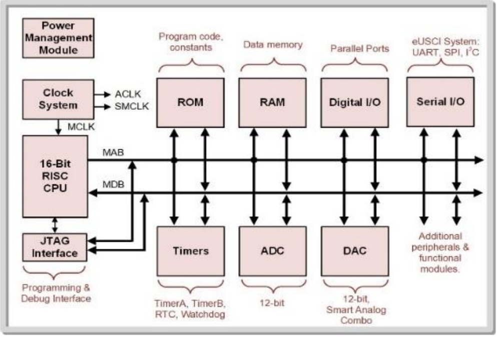
- System resets include power-on reset (POR), power-up reset (PUR), external reset, and power supply monitor violation.

- The only action needed by developer for system resets is to tell the interrupt system where the starting address of the main program is.

11.1.2 INTERRUPT PRIORITY AND ENABLING

- Categories of interrupts:

  1) System resets,
  2) Non-maskable interrupts (NMIs),
  3) Maskable interrupts

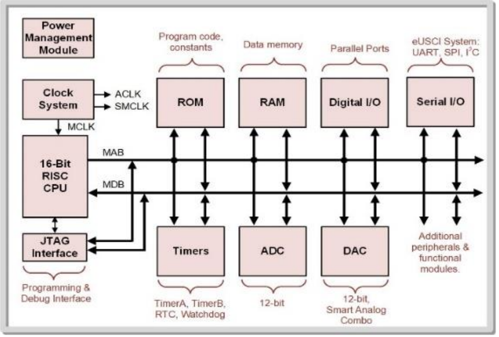


- **Non-maskable Interrupts** – second highest priority interrupts; typically handle fault conditions on the MCU.
- Examples include memory access errors and oscillator faults.
- Non-maskable interrupts are always enabled but are different from system resets in that they *do* execute developer written ISRs instead of a set of predetermined actions.

11.1.2 Interrupt Priority and Enabling

- Categories of interrupts:

  1) System resets,
  2) Non-maskable interrupts (NMIs),
  3) Maskable interrupts



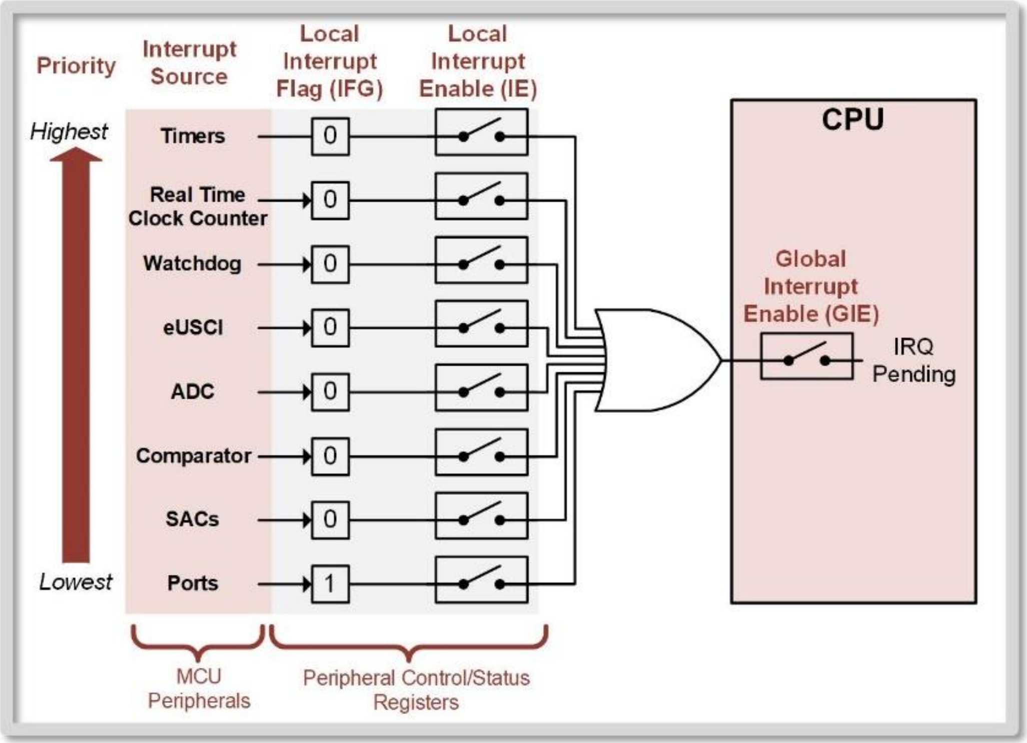
- **Maskable Interrupts** – third highest priority interrupts.

These are the peripherals!

- ports
- timers
- serial interface
- ADC
- DAC

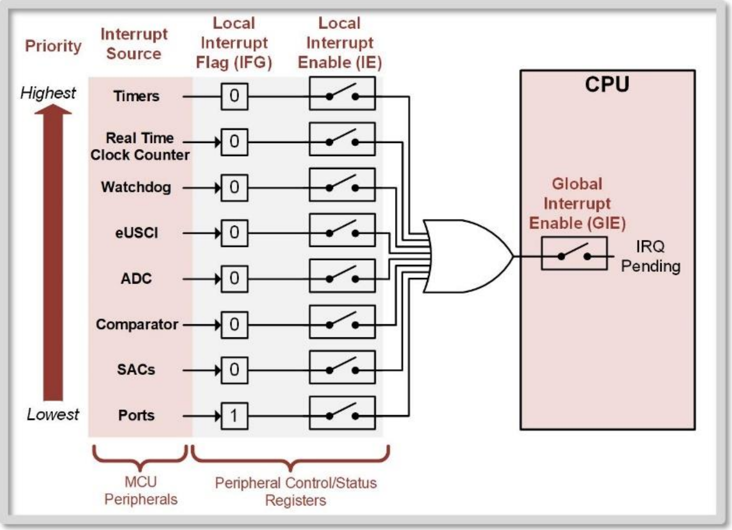
11.1.2 INTERRUPT PRIORITY AND ENABLING

- **Maskable Interrupts** - means that they can be turned on or off.



11.1.2 Interrupt Priority and Enabling

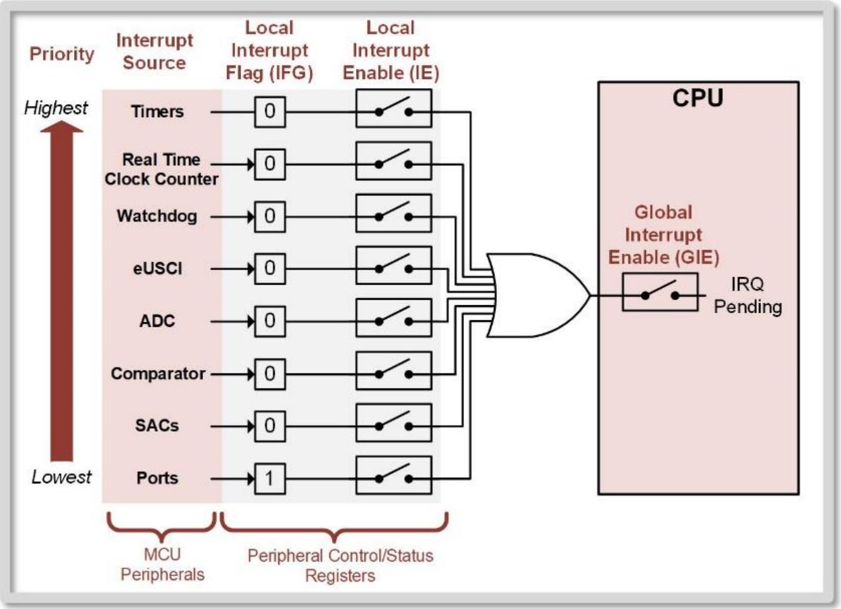
- **GIE bit** – used as the global enable for all maskable interrupts.
  - **GIE = 1** – maskable interrupts enabled
  - **GIE = 0** – maskable interrupts disabled
- Enabling and Disabling can be done by simply setting or clearing the GIE bit in the SR using **EINT** and **DINT** respectively.



11.1.2 INTERRUPT PRIORITY AND ENABLING

- **Interrupt Enable (IE)** – configured in the control/status registers within the memory map for **each** peripheral.

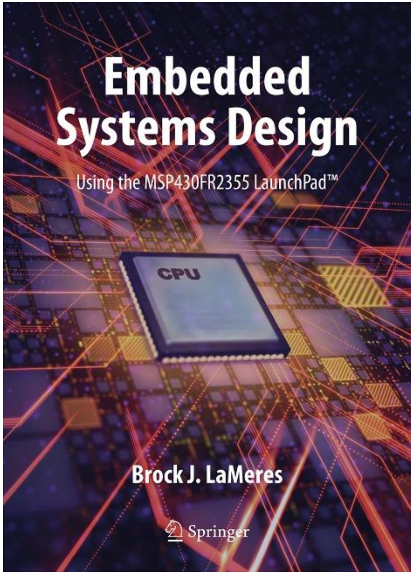
- Global and local interrupt enable bits can be thought of as gating switches that allow the peripheral's flag to be observed by the CPU when configured.



Embedded Systems Design

Chapter 11: Introduction to Interrupts

11.1.2 Interrupt Priority and Enabling


Embedded Systems Design

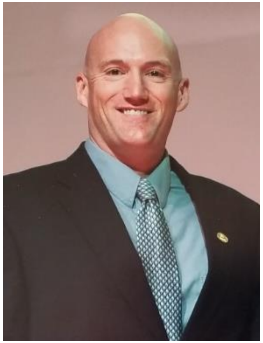
Using the MSP430FR2355 LaunchPad™

Brock J. LaMeres

Springer

Subscribe

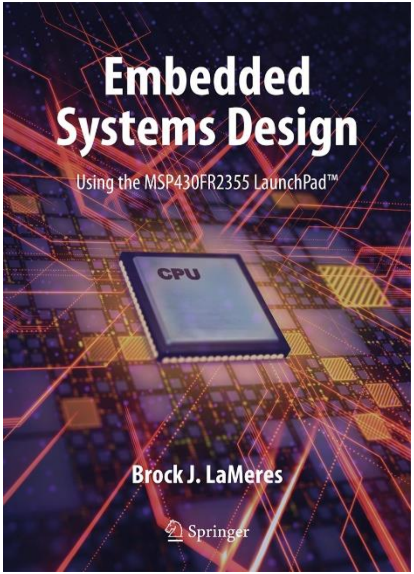
www.youtube.com/c/DigitalLogicProgramming_LaMeres

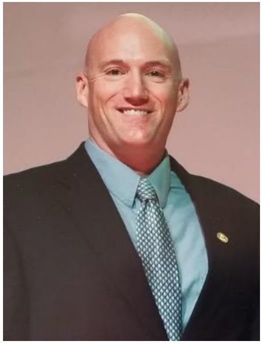


Brock J. LaMeres, Ph.D.

CHAPTER 11: INTRODUCTION TO INTERRUPTS

11.1.3 INTERRUPT VECTORS





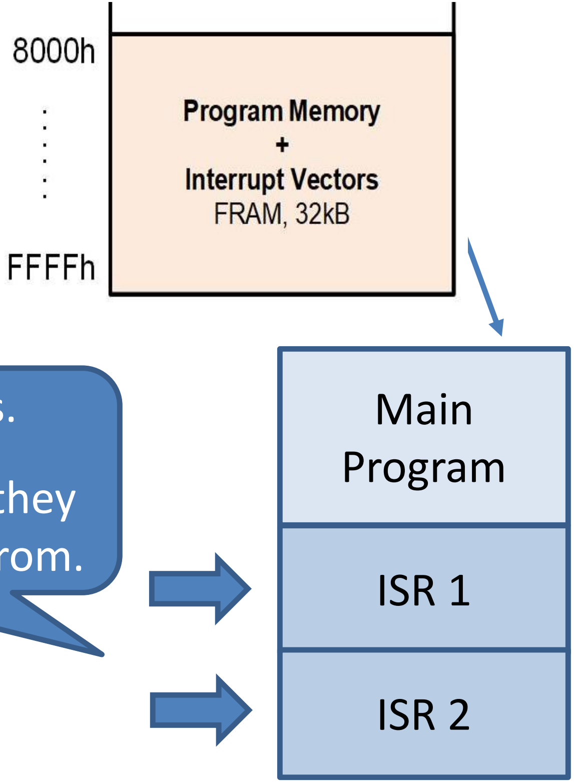
BROCK J. LAMERES, PH.D.

11.1.3 Interrupt Vectors

- An Interrupt Service Routine (ISR) is written for any peripheral that is to be handled.

Similar to subroutines.

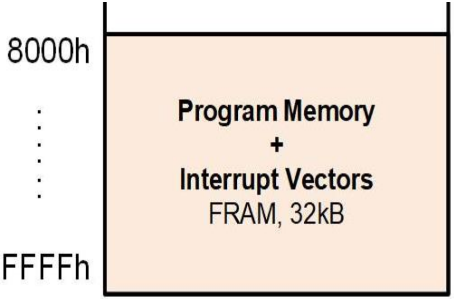
But different in the way they are called and returned from.

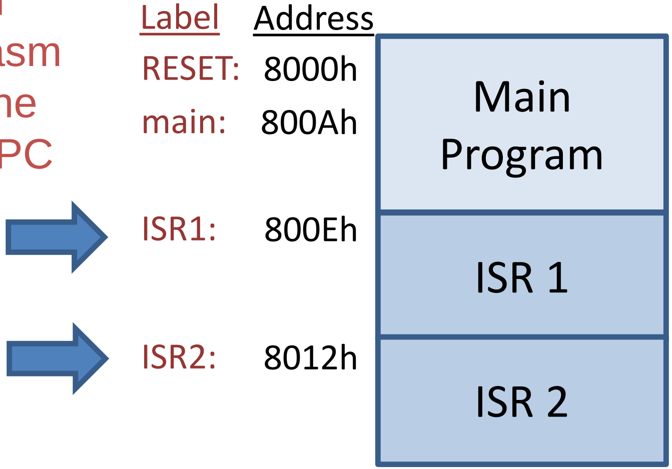


11.1.3 INTERRUPT VECTORS

- An Interrupt Service Routine (ISR) is written for any peripheral that is to be handled.

- The beginning of an ISR is marked with an address label in the main.asm file. This address label serves as the starting address to be put into the PC when the ISR is called.

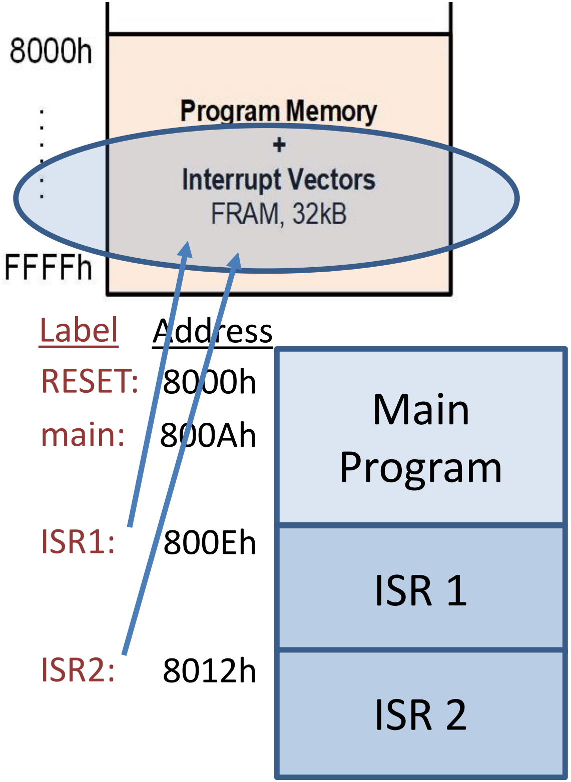




Label	Address
RESET:	8000h
main:	800Ah
ISR1:	800Eh
ISR2:	8012h

11.1.3 Interrupt Vectors

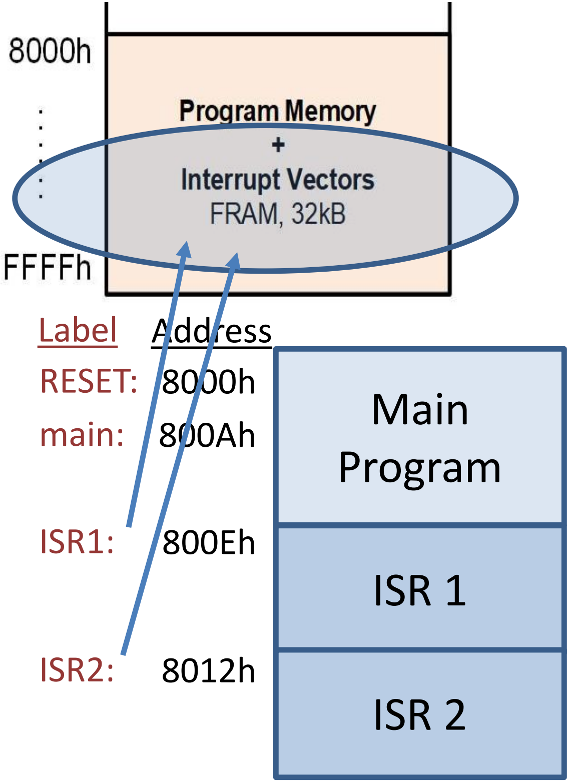
- An Interrupt Service Routine (ISR) is written for any peripheral that is to be handled.
- The beginning of an ISR is marked with an address label in the main.asm file. This address label serves as the starting address to be put into the PC when the ISR is called.
- The starting address of the ISR is then put into a dedicated **Interrupt Vector**. That is associated with the peripheral.



Label	Address
RESET:	8000h
main:	800Ah
ISR1:	800Eh
ISR2:	8012h

11.1.3 INTERRUPT VECTORS

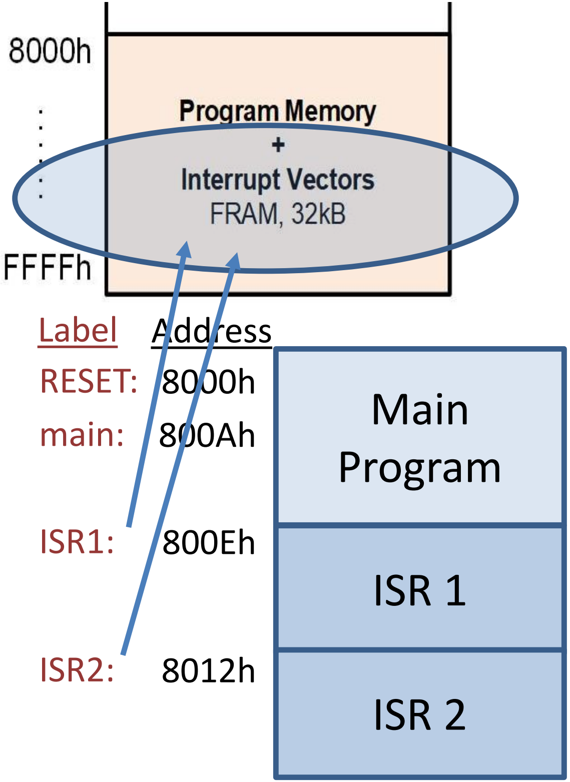
- **Interrupt Vectors** – Each peripheral has a dedicated memory location that holds the starting address of its ISR.



11.1.3 INTERRUPT VECTORS

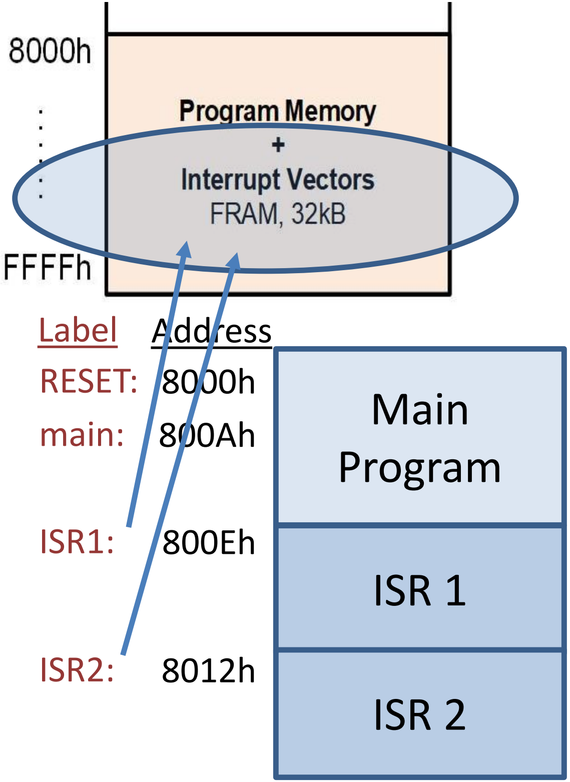
- **Interrupt Vectors** – Each peripheral has a dedicated memory location that holds the starting address of its ISR.

- This dedicated location is an "address pointer" of where the ISR starts. (aka, a vector).



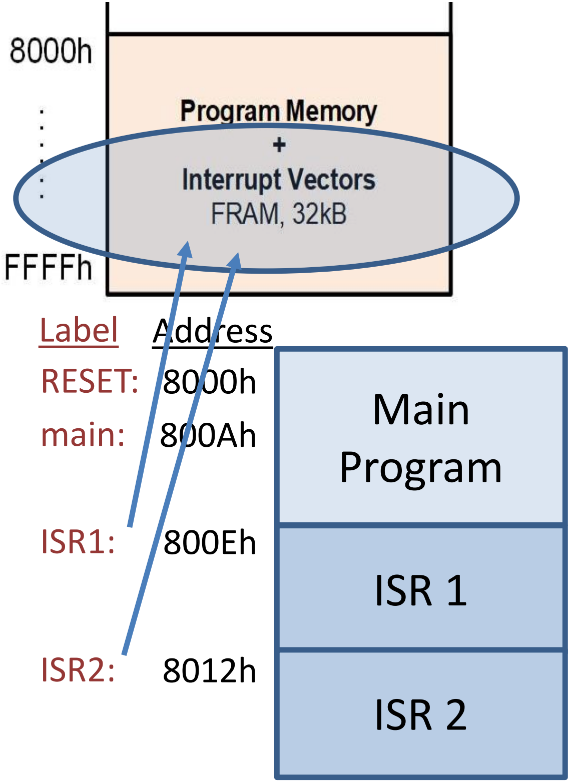
11.1.3 INTERRUPT VECTORS

- **Interrupt Vectors** – Each peripheral has a dedicated memory location that holds the starting address of its ISR.
- This dedicated location is an "address pointer" of where the ISR starts. (aka, a vector).
- Since the ISR starting address changes for every program, the developer must initialize the vector using directives when downloaded.



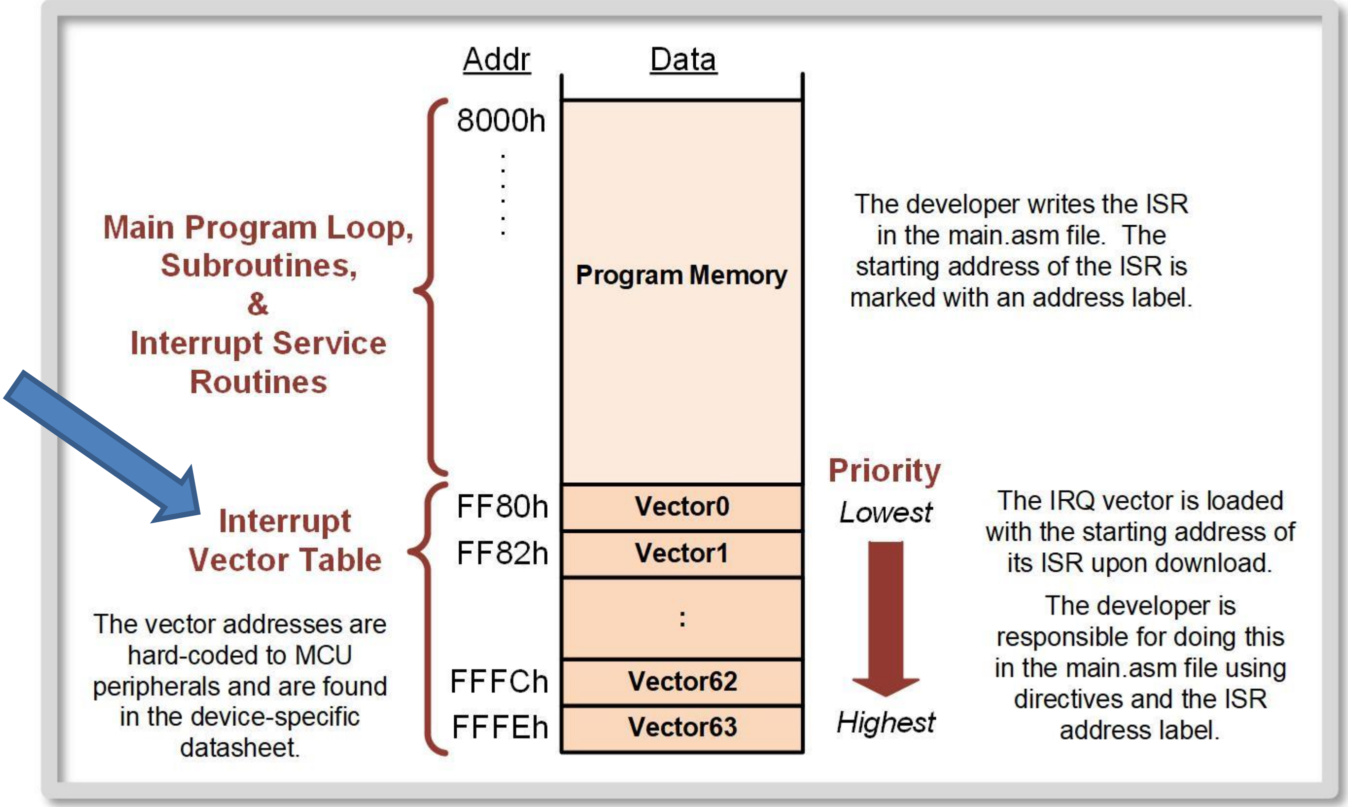
11.1.3 INTERRUPT VECTORS

- **Interrupt Vectors** – Each peripheral has a dedicated memory location that holds the starting address of its ISR.
- This dedicated location is an "address pointer" of where the ISR starts. (aka, a vector).
- Since the ISR starting address changes for every program, the developer must initialize the vector using directives when downloaded.
- The vector addresses are **hard-coded**.



Label	Address
RESET:	8000h
main:	800Ah
ISR1:	800Eh
ISR2:	8012h

11.1.3 INTERRUPT VECTORS



The developer writes the ISR in the main.asm file. The starting address of the ISR is marked with an address label.

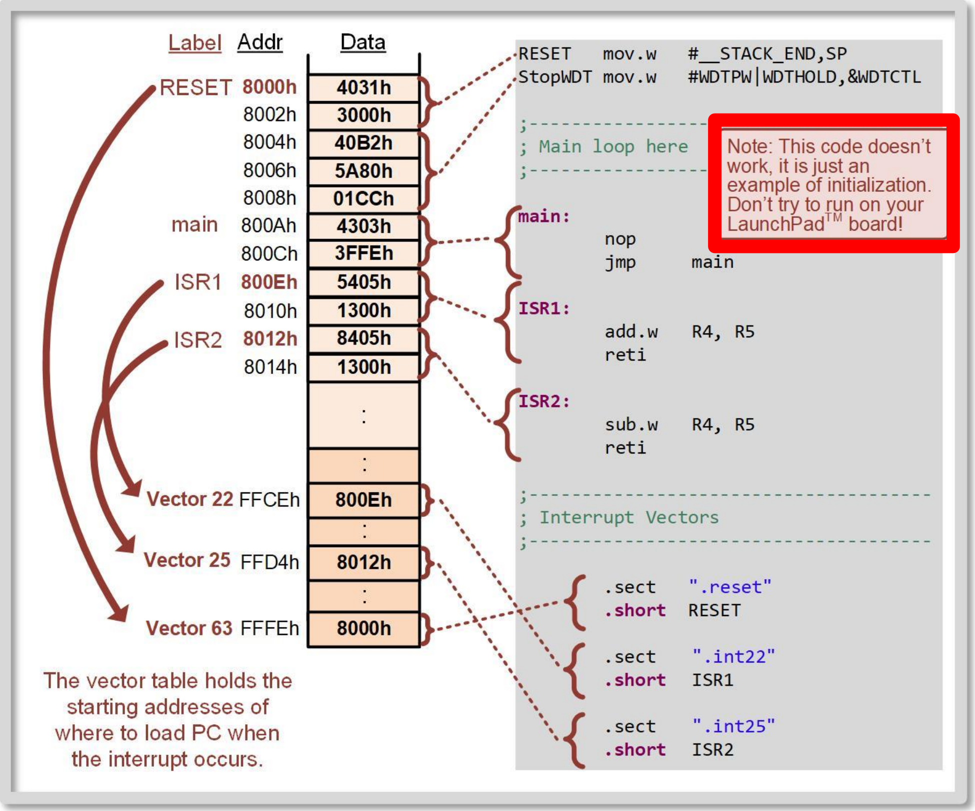
The IRQ vector is loaded with the starting address of its ISR upon download.

The developer is responsible for doing this in the main.asm file using directives and the ISR address label.

The vector addresses are hard-coded to MCU peripherals and are found in the device-specific datasheet.

11.1.3 INTERRUPT VECTORS

- Let's see how this might look:



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

```
RESET    mov.w   #__STACK_END,SP
StopWDT  mov.w   #WDTPW|WDTHOLD,&WDTCTL

;----------------
; Main loop here
;----------------
main:
         nop
         jmp     main
ISR1:
         add.w   R4, R5
         reti
ISR2:
         sub.w   R4, R5
         reti

;-----------------------------------
; Interrupt Vectors
;-----------------------------------
         .sect   ".reset"
         .short  RESET
         .sect   ".int22"
         .short  ISR1
         .sect   ".int25"
         .short  ISR2
```

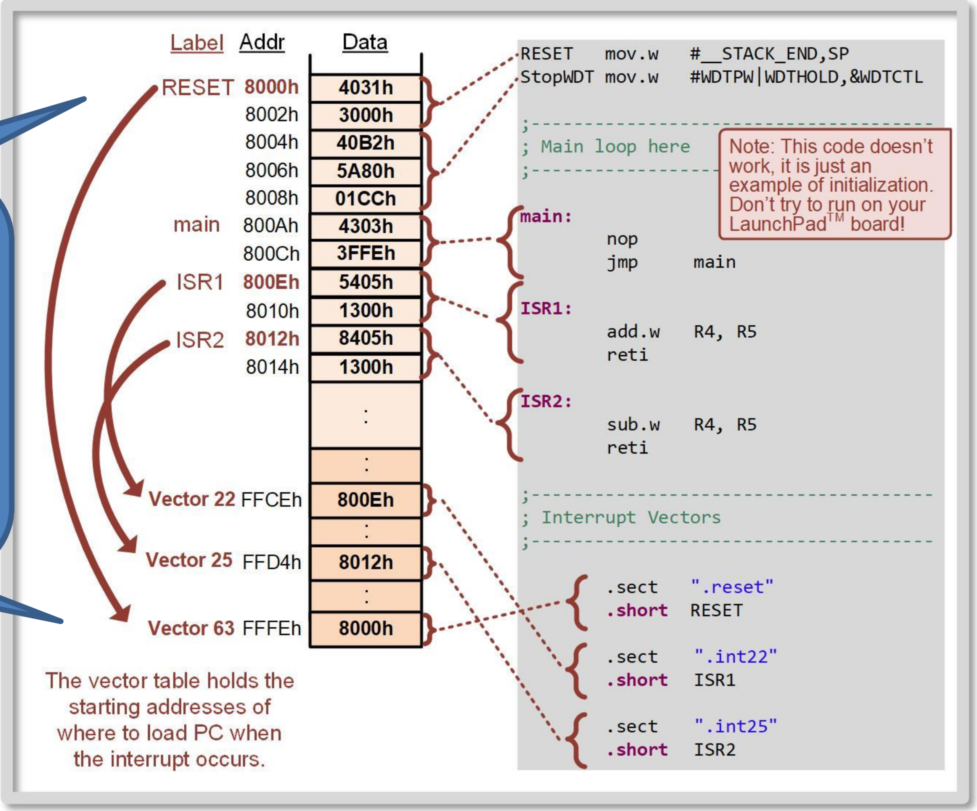
Note: This code doesn't work, it is just an example of initialization. Don't try to run on your LaunchPad™ board!

The vector table holds the starting addresses of where to load PC when the interrupt occurs.

11.1.3 Interrupt Vectors

- Let's see how this might look:

RESET (vector 63) is where the program starts. Always the highest priority.



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

The vector table holds the starting addresses of where to load PC when the interrupt occurs.

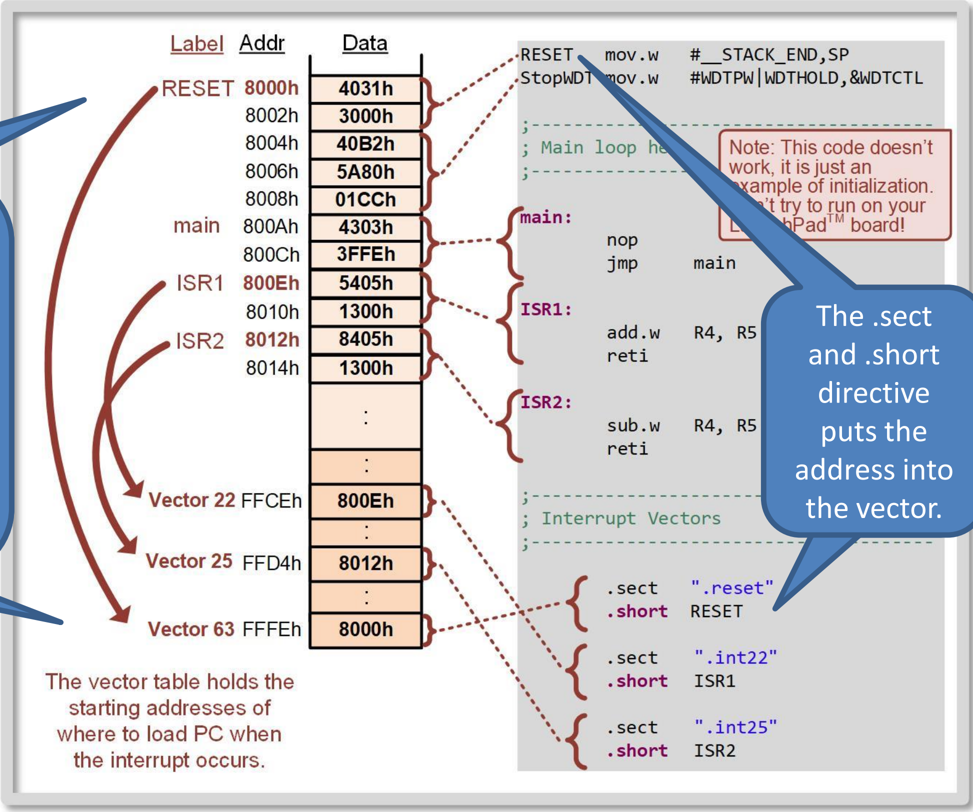
```
RESET    mov.w   #__STACK_END,SP
StopWDT  mov.w   #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------------
; Main loop here
;-------------------------------------------
main:
         nop
         jmp     main

ISR1:
         add.w   R4, R5
         reti

ISR2:
         sub.w   R4, R5
         reti

;-------------------------------------------
; Interrupt Vectors
;-------------------------------------------

         .sect   ".reset"
         .short  RESET

         .sect   ".int22"
         .short  ISR1

         .sect   ".int25"
         .short  ISR2
```

Note: This code doesn't work, it is just an example of initialization. Don't try to run on your LaunchPad^{TM} board!

11.1.3 INTERRUPT VECTORS

- Let's see how this might look:

RESET (vector 63) is where the program starts. Always the highest priority.



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

The vector table holds the starting addresses of where to load PC when the interrupt occurs.

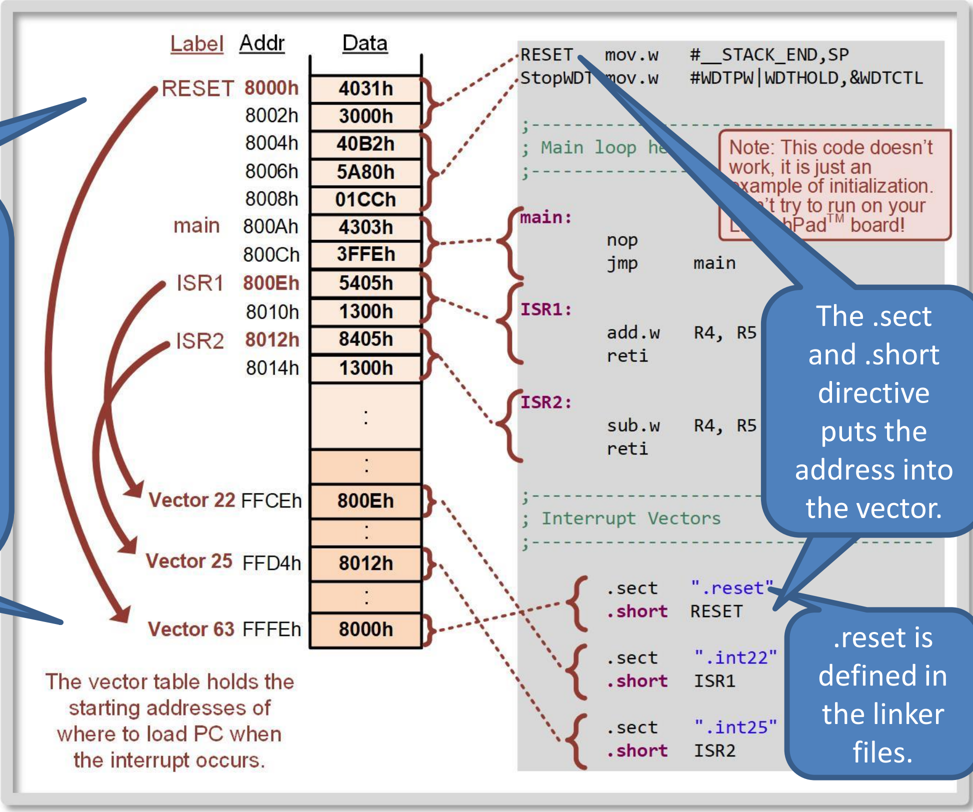
```
RESET   mov.w   #__STACK_END,SP
StopWDT mov.w   #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------------
; Main loop he
;-------------------

main:
        nop
        jmp     main

ISR1:
        add.w   R4, R5
        reti

ISR2:
        sub.w   R4, R5
        reti

;-------------------------------------------
; Interrupt Vectors
;-------------------------------------------
        .sect   ".reset"
        .short  RESET

        .sect   ".int22"
        .short  ISR1

        .sect   ".int25"
        .short  ISR2
```

Note: This code doesn't work, it is just an xample of initialization. 't try to run on your L hPad™ board!

The .sect and .short directive puts the address into the vector.

11.1.3 INTERRUPT VECTORS

- Let's see how this might look:



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		⋮
		⋮
Vector 22	FFCEh	800Eh
		⋮
Vector 25	FFD4h	8012h
		⋮
Vector 63	FFFEh	8000h

The vector table holds the starting addresses of where to load PC when the interrupt occurs.

```
RESET    mov.w    #__STACK_END,SP
StopWDT  mov.w    #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------------
; Main loop he
;-------------
main:
         nop
         jmp      main

ISR1:
         add.w    R4, R5
         reti

ISR2:
         sub.w    R4, R5
         reti

;-------------------------------------------
; Interrupt Vectors
;-------------------------------------------
         .sect    ".reset"
         .short   RESET

         .sect    ".int22"
         .short   ISR1

         .sect    ".int25"
         .short   ISR2
```

Note: This code doesn't work, it is just an xample of initialization. 't try to run on your L hPad™ board!

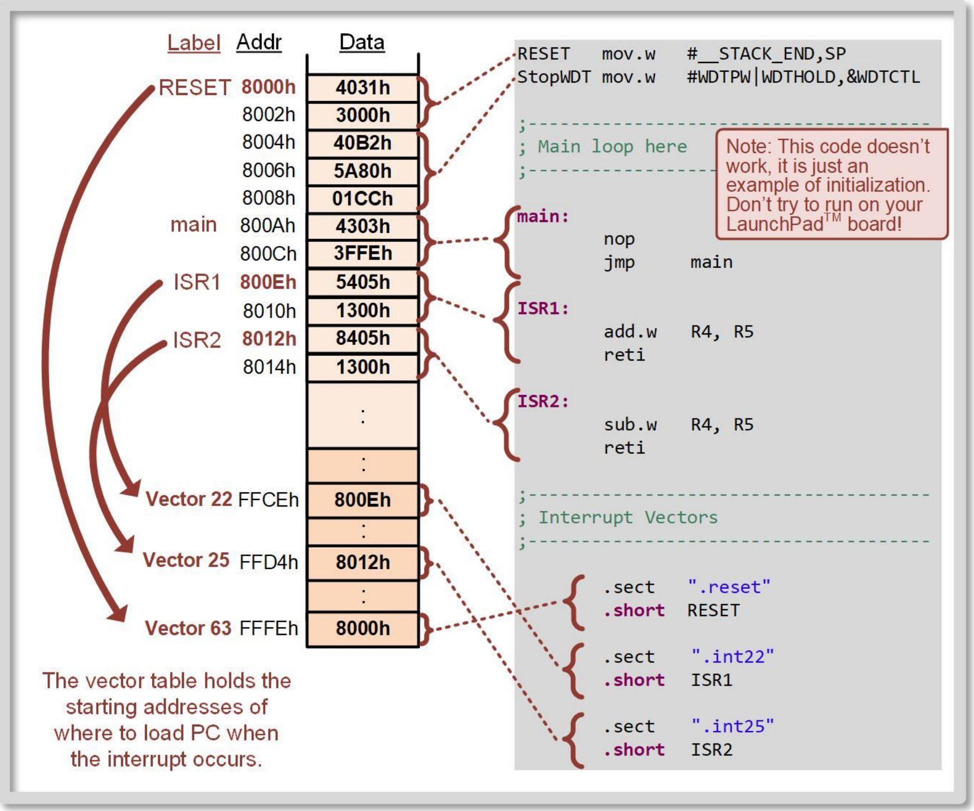
RESET (vector 63) is where the program starts. Always the highest priority.

The .sect and .short directive puts the address into the vector.

.reset is defined in the linker files.

11.1.3 INTERRUPT VECTORS

- Let's see how this might look:



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

```
RESET    mov.w   #__STACK_END,SP
StopWDT  mov.w   #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------
; Main loop here
;-------------------------------------
main:
         nop
         jmp     main

ISR1:
         add.w   R4, R5
         reti

ISR2:
         sub.w   R4, R5
         reti

;-------------------------------------
; Interrupt Vectors
;-------------------------------------
         .sect   ".reset"
         .short  RESET

         .sect   ".int22"
         .short  ISR1

         .sect   ".int25"
         .short  ISR2
```

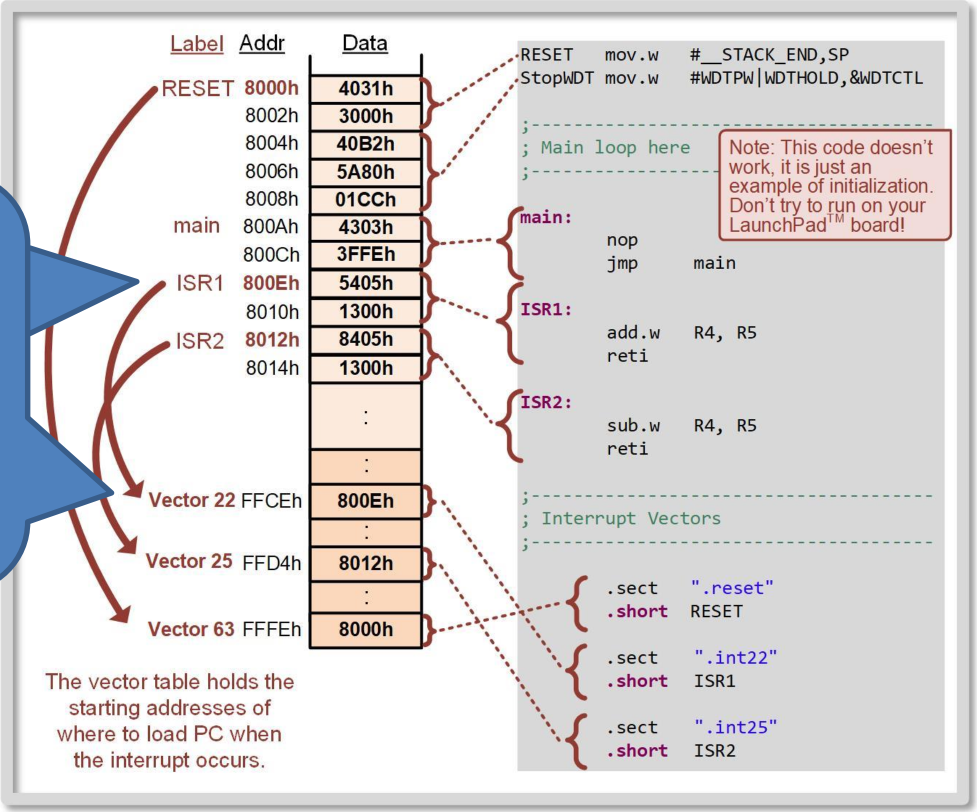
Note: This code doesn't work, it is just an example of initialization. Don't try to run on your LaunchPad™ board!

The vector table holds the starting addresses of where to load PC when the interrupt occurs.

11.1.3 INTERRUPT VECTORS

- Let’s see how this might look:

Let’s say Vector 22 is a peripheral. We need to put its ISR’s starting address in its corresponding vector address location.



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

```
RESET    mov.w   #__STACK_END,SP
StopWDT  mov.w   #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------------------------
; Main loop here
;-------------------------------------------------------
main:
         nop
         jmp     main

ISR1:
         add.w   R4, R5
         reti

ISR2:
         sub.w   R4, R5
         reti

;-------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------
         .sect   ".reset"
         .short  RESET

         .sect   ".int22"
         .short  ISR1

         .sect   ".int25"
         .short  ISR2
```

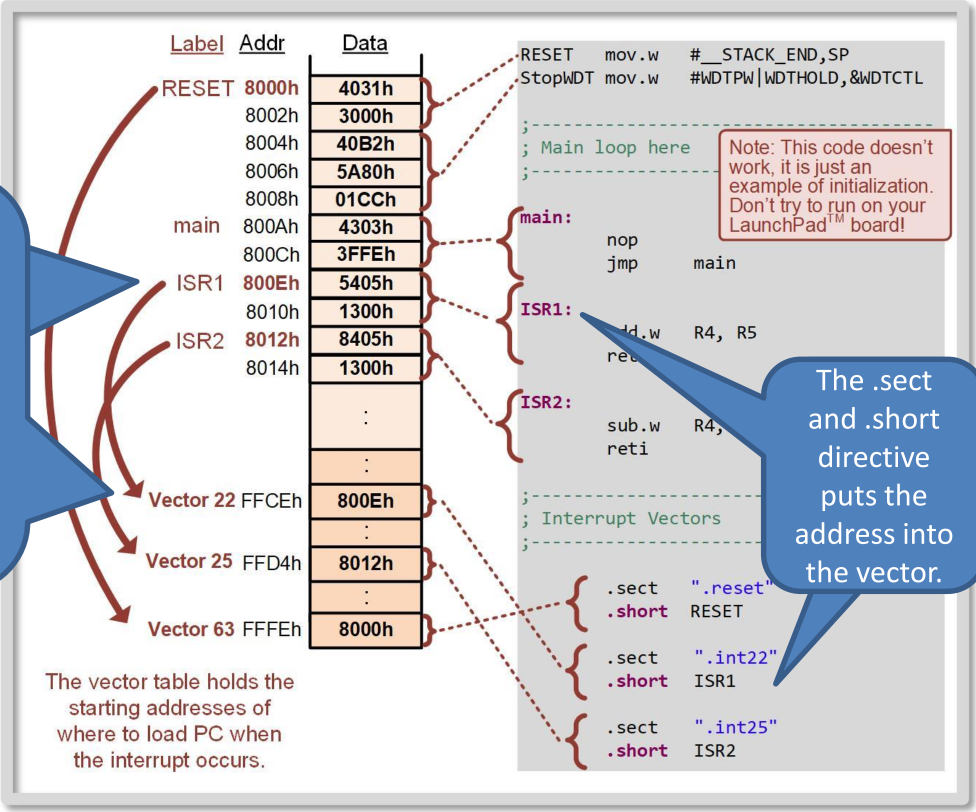
Note: This code doesn’t work, it is just an example of initialization. Don’t try to run on your LaunchPad™ board!

The vector table holds the starting addresses of where to load PC when the interrupt occurs.

11.1.3 INTERRUPT VECTORS

- Let’s see how this might look:

Let’s say Vector 22 is a peripheral. We need to put its ISR’s starting address in its corresponding vector address location.



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

```
RESET     mov.w   #__STACK_END,SP
StopWDT   mov.w   #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------
main:
          nop
          jmp     main

ISR1:
           d.w    R4, R5
          re

ISR2:
          sub.w   R4,
          reti

;-------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------
          .sect   ".reset"
          .short  RESET

          .sect   ".int22"
          .short  ISR1

          .sect   ".int25"
          .short  ISR2
```

Note: This code doesn’t work, it is just an example of initialization. Don’t try to run on your LaunchPad™ board!

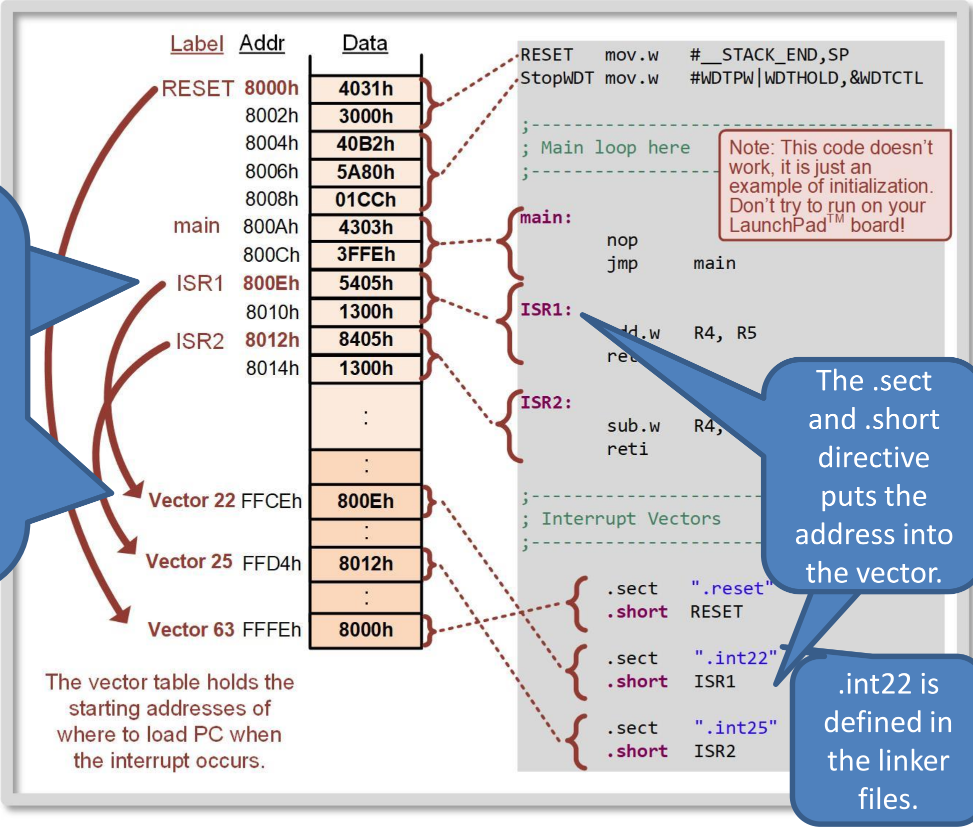
The vector table holds the starting addresses of where to load PC when the interrupt occurs.

The .sect and .short directive puts the address into the vector.

11.1.3 Interrupt Vectors

- Let’s see how this might look:

Let’s say Vector 22 is a peripheral. We need to put its ISR’s starting address in its corresponding vector address location.



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

The vector table holds the starting addresses of where to load PC when the interrupt occurs.

```
RESET   mov.w   #__STACK_END,SP
StopWDT mov.w   #WDTPW|WDTHOLD,&WDTCTL

;--------------------------------------------------
; Main loop here
;----------------
main:
        nop
        jmp     main

ISR1:
         d.w    R4, R5
        re

ISR2:
        sub.w   R4,
        reti

;-----------------------
; Interrupt Vectors
;-----------------------
        .sect   ".reset"
        .short  RESET

        .sect   ".int22"
        .short  ISR1

        .sect   ".int25"
        .short  ISR2
```

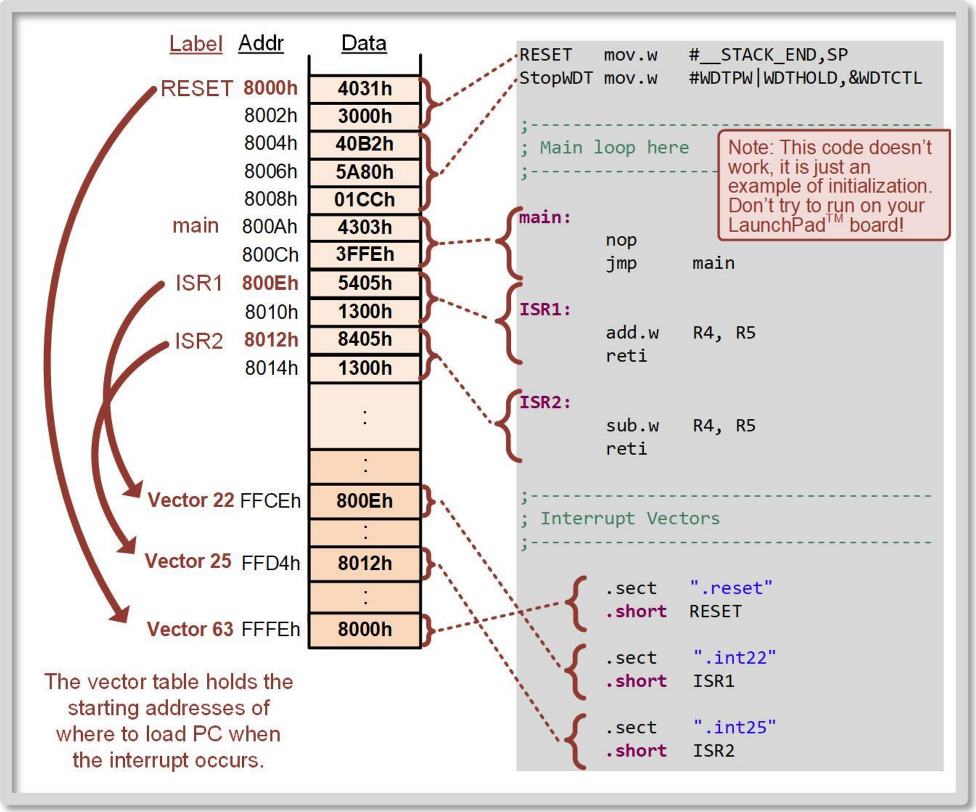
Note: This code doesn’t work, it is just an example of initialization. Don’t try to run on your LaunchPad™ board!

The .sect and .short directive puts the address into the vector.

.int22 is defined in the linker files.

11.1.3 Interrupt Vectors

- Let’s see how this might look:



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

```
RESET   mov.w   #__STACK_END,SP
StopWDT mov.w   #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------------
; Main loop here
;-------------------------------------------
main:
        nop
        jmp     main

ISR1:
        add.w   R4, R5
        reti

ISR2:
        sub.w   R4, R5
        reti

;-------------------------------------------
; Interrupt Vectors
;-------------------------------------------
        .sect   ".reset"
        .short  RESET

        .sect   ".int22"
        .short  ISR1

        .sect   ".int25"
        .short  ISR2
```

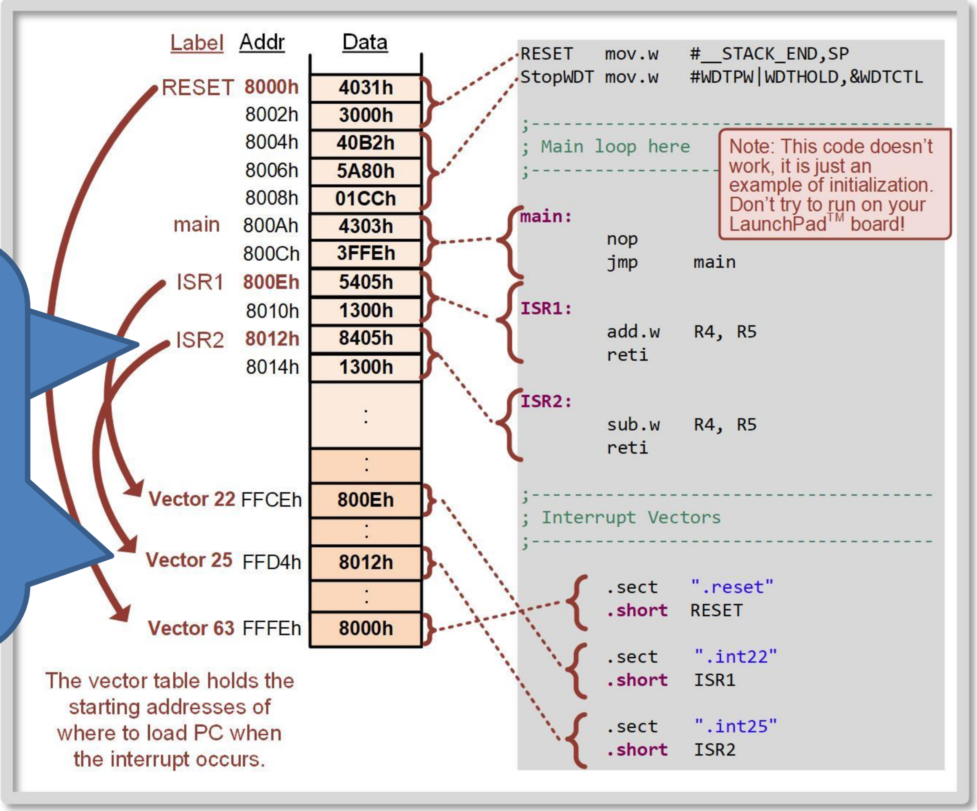
Note: This code doesn’t work, it is just an example of initialization. Don’t try to run on your LaunchPad™ board!

The vector table holds the starting addresses of where to load PC when the interrupt occurs.

11.1.3 Interrupt Vectors

- Let's see how this might look:

Let's say Vector 25 is a peripheral. We need to put its ISR's starting address in its corresponding vector address location.



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

```
RESET    mov.w   #__STACK_END,SP
StopWDT  mov.w   #WDTPW|WDTHOLD,&WDTCTL

;------------------------------------------
; Main loop here
;------------------------------------------
main:
         nop
         jmp     main

ISR1:
         add.w   R4, R5
         reti

ISR2:
         sub.w   R4, R5
         reti

;------------------------------------------
; Interrupt Vectors
;------------------------------------------
         .sect   ".reset"
         .short  RESET

         .sect   ".int22"
         .short  ISR1

         .sect   ".int25"
         .short  ISR2
```

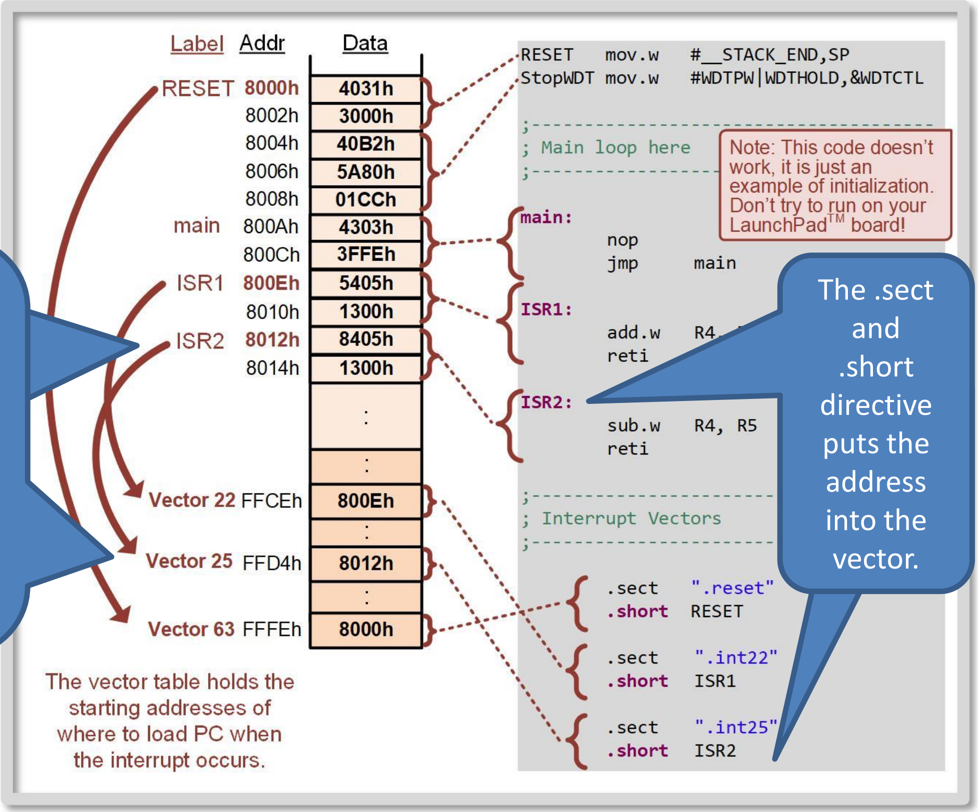
Note: This code doesn't work, it is just an example of initialization. Don't try to run on your LaunchPad^{TM} board!

The vector table holds the starting addresses of where to load PC when the interrupt occurs.

11.1.3 INTERRUPT VECTORS

- Let's see how this might look:

Let's say Vector 25 is a peripheral. We need to put its ISR's starting address in its corresponding vector address location.



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

```
RESET    mov.w    #__STACK_END,SP
StopWDT  mov.w    #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------
; Main loop here
;-------------------------------------
main:
         nop
         jmp      main

ISR1:
         add.w    R4
         reti

ISR2:
         sub.w    R4, R5
         reti

;-------------------------------------
; Interrupt Vectors
;-------------------------------------
         .sect    ".reset"
         .short   RESET

         .sect    ".int22"
         .short   ISR1

         .sect    ".int25"
         .short   ISR2
```

Note: This code doesn't work, it is just an example of initialization. Don't try to run on your LaunchPad™ board!

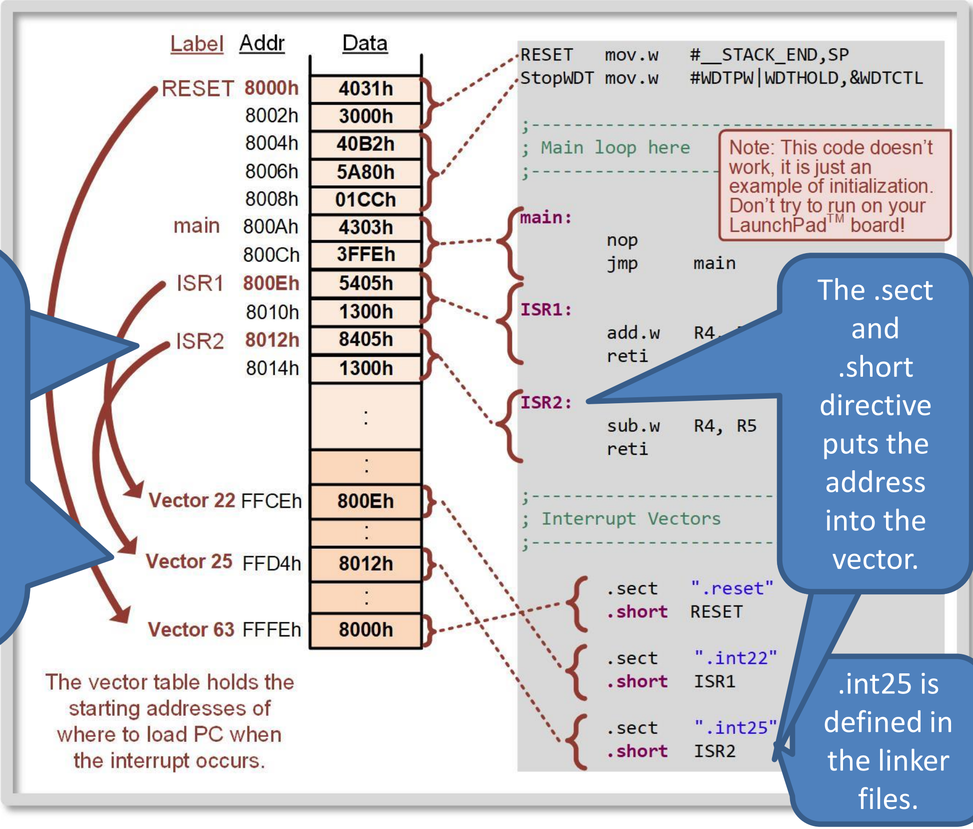
The vector table holds the starting addresses of where to load PC when the interrupt occurs.

The .sect and .short directive puts the address into the vector.

11.1.3 INTERRUPT VECTORS

- Let’s see how this might look:

Let’s say Vector 25 is a peripheral. We need to put its ISR’s starting address in its corresponding vector address location.



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

The vector table holds the starting addresses of where to load PC when the interrupt occurs.

```
RESET    mov.w   #__STACK_END,SP
StopWDT  mov.w   #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------------
; Main loop here
;-------------------------------------------
main:
         nop
         jmp     main

ISR1:
         add.w   R4,

         reti

ISR2:
         sub.w   R4, R5
         reti

;-------------------------------------------
; Interrupt Vectors
;-------------------------------------------
         .sect   ".reset"
         .short  RESET

         .sect   ".int22"
         .short  ISR1

         .sect   ".int25"
         .short  ISR2
```

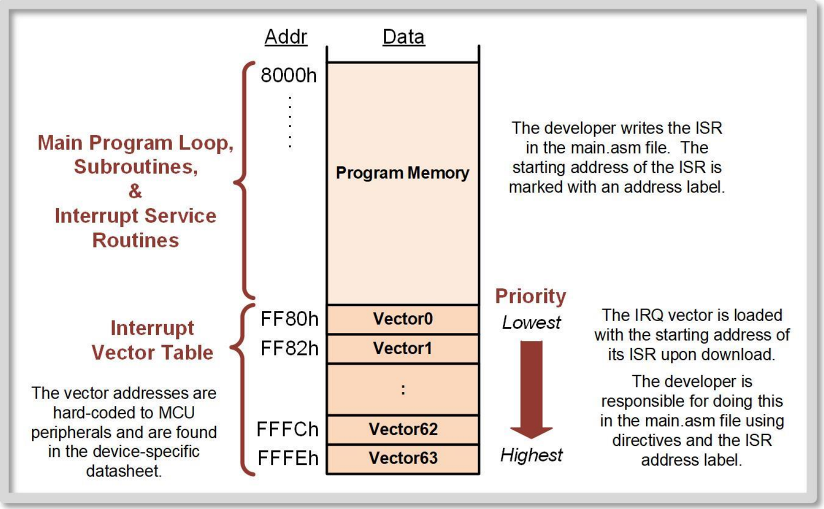
Note: This code doesn’t work, it is just an example of initialization. Don’t try to run on your LaunchPad™ board!

The .sect and .short directive puts the address into the vector.

.int25 is defined in the linker files.

11.1.3 Interrupt Vectors

- The full MSP430 architecture supports up to 64 separate interrupt vectors whose addresses are located within the range FF80h → FFFFh.
- Not all are active in every MCU.



The developer writes the ISR in the main.asm file. The starting address of the ISR is marked with an address label.

The vector addresses are hard-coded to MCU peripherals and are found in the device-specific datasheet.

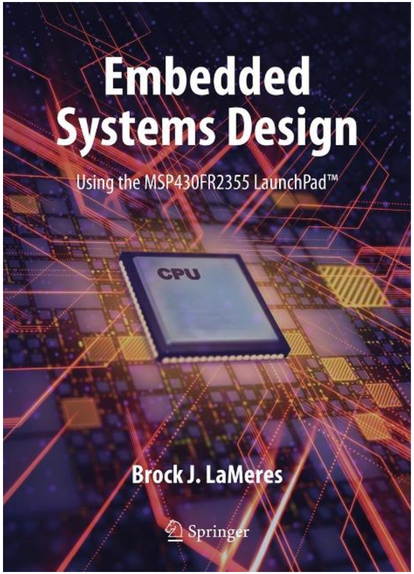
The IRQ vector is loaded with the starting address of its ISR upon download.

The developer is responsible for doing this in the main.asm file using directives and the ISR address label.

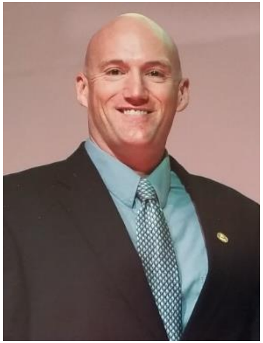
Embedded Systems Design

Chapter 11: Introduction to Interrupts

11.1.3 Interrupt Vectors



www.youtube.com/c/DigitalLogicProgramming_LaMeres

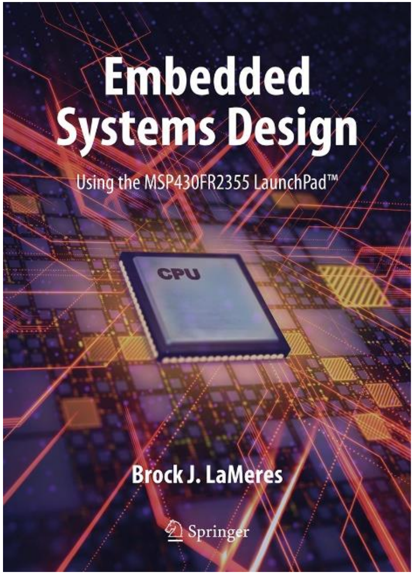


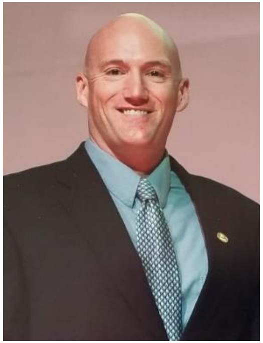
Brock J. LaMeres, Ph.D.

Embedded Systems Design

Chapter 11: Introduction to Interrupts

11.1.4 Operation of the STACK During an IRQ

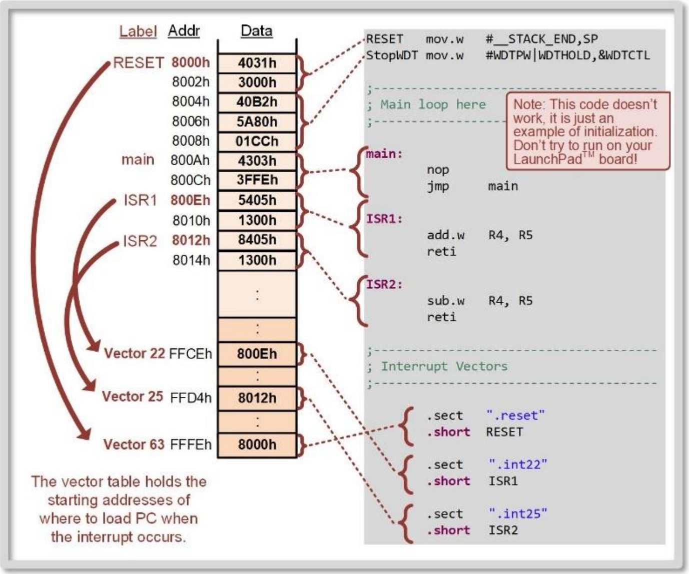




Brock J. LaMeres, Ph.D.

11.1.4 OPERATION OF THE STACK DURING AN IRQ

- When an IRQ is to be serviced:



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

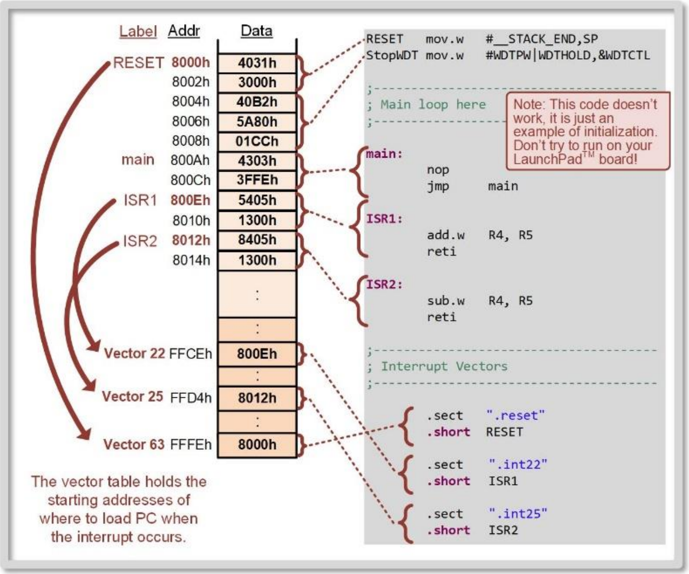
The vector table holds the starting addresses of where to load PC when the interrupt occurs.

```
RESET   mov.w   #__STACK_END,SP
StopWDT mov.w   #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------------
; Main loop here
;-------------------------------------------

main:
        nop
        jmp     main

ISR1:
        add.w   R4, R5
        reti

ISR2:
        sub.w   R4, R5
        reti

;-------------------------------------------
; Interrupt Vectors
;-------------------------------------------

        .sect   ".reset"
        .short  RESET

        .sect   ".int22"
        .short  ISR1

        .sect   ".int25"
        .short  ISR2
```

Note: This code doesn't work, it is just an example of initialization. Don't try to run on your LaunchPad™ board!

11.1.4 Operation of the STACK during an IRQ

- When an IRQ is to be serviced:
  1. CPU finishes its current instruction.



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

```
RESET    mov.w   #__STACK_END,SP
StopWDT  mov.w   #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------------
; Main loop here
;-------------------------------------------
main:
         nop
         jmp     main

ISR1:
         add.w   R4, R5
         reti

ISR2:
         sub.w   R4, R5
         reti

;-------------------------------------------
; Interrupt Vectors
;-------------------------------------------
         .sect   ".reset"
         .short  RESET

         .sect   ".int22"
         .short  ISR1

         .sect   ".int25"
         .short  ISR2
```

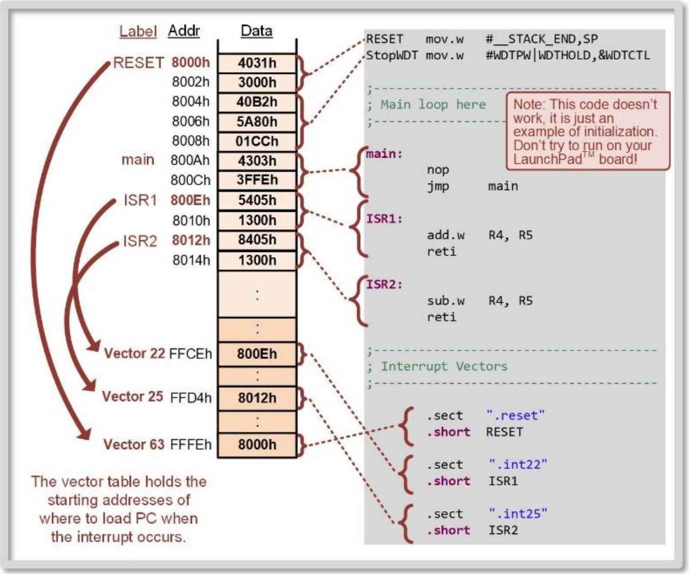
Note: This code doesn't work, it is just an example of initialization. Don't try to run on your LaunchPad™ board!

The vector table holds the starting addresses of where to load PC when the interrupt occurs.

11.1.4 Operation of the Stack during an IRQ

- When an IRQ is to be serviced:

1. CPU finishes its current instruction.
2. The next instruction in main program is put into PC.



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

The vector table holds the starting addresses of where to load PC when the interrupt occurs.

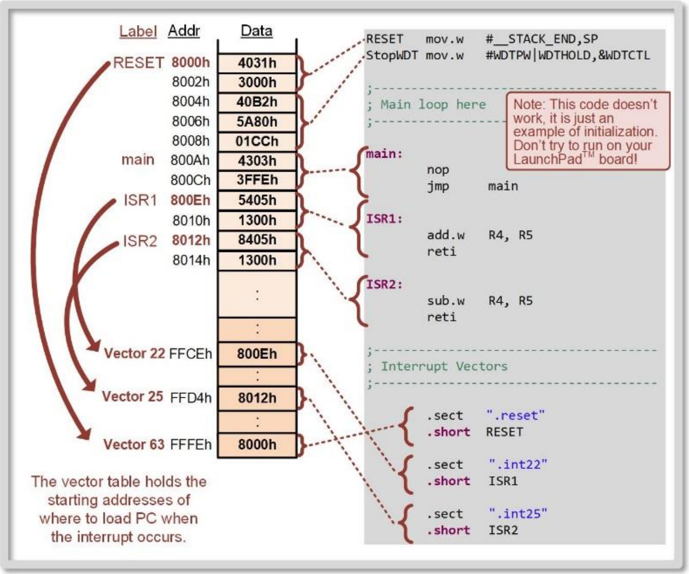
```
RESET   mov.w   #__STACK_END,SP
StopWDT mov.w   #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------
; Main loop here
;-------------------------------------

main:
        nop
        jmp     main

ISR1:
        add.w   R4, R5
        reti

ISR2:
        sub.w   R4, R5
        reti

;-------------------------------------
; Interrupt Vectors
;-------------------------------------

        .sect   ".reset"
        .short  RESET

        .sect   ".int22"
        .short  ISR1

        .sect   ".int25"
        .short  ISR2
```

Note: This code doesn't work, it is just an example of initialization. Don't try to run on your LaunchPad™ board!

11.1.4 OPERATION OF THE STACK DURING AN IRQ

- When an IRQ is to be serviced:

1. CPU finishes its current instruction.
2. The next instruction in main program is put into PC.
3. PC and SR are **pushed to stack**.



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

```
RESET   mov.w   #__STACK_END,SP
StopWDT mov.w   #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------
; Main loop here
;-------------------------------------

main:
        nop
        jmp     main

ISR1:
        add.w   R4, R5
        reti

ISR2:
        sub.w   R4, R5
        reti

;-------------------------------------
; Interrupt Vectors
;-------------------------------------

        .sect   ".reset"
        .short  RESET

        .sect   ".int22"
        .short  ISR1

        .sect   ".int25"
        .short  ISR2
```

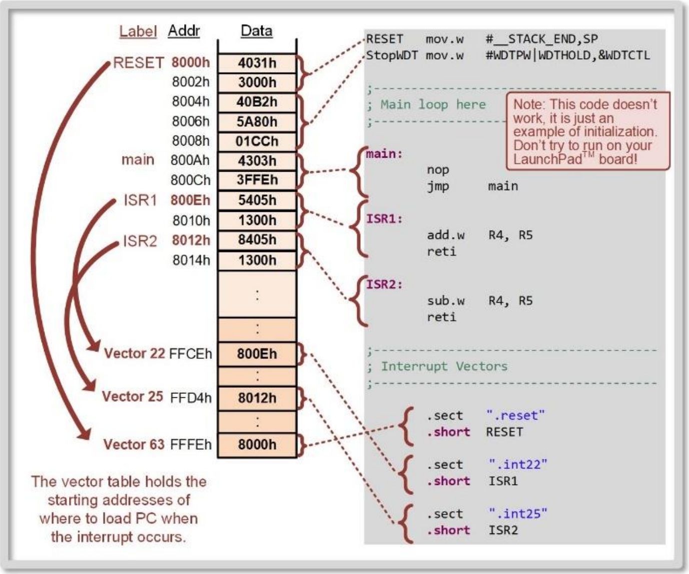
Note: This code doesn't work, it is just an example of initialization. Don't try to run on your LaunchPad™ board!

The vector table holds the starting addresses of where to load PC when the interrupt occurs.

11.1.4 OPERATION OF THE STACK DURING AN IRQ

- When an IRQ is to be serviced:

1. CPU finishes its current instruction.
2. The next instruction in main program is put into PC.
3. PC and SR are **pushed to stack**.
4. MCU clears the SR so that it can be used by the ISR.



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

The vector table holds the starting addresses of where to load PC when the interrupt occurs.

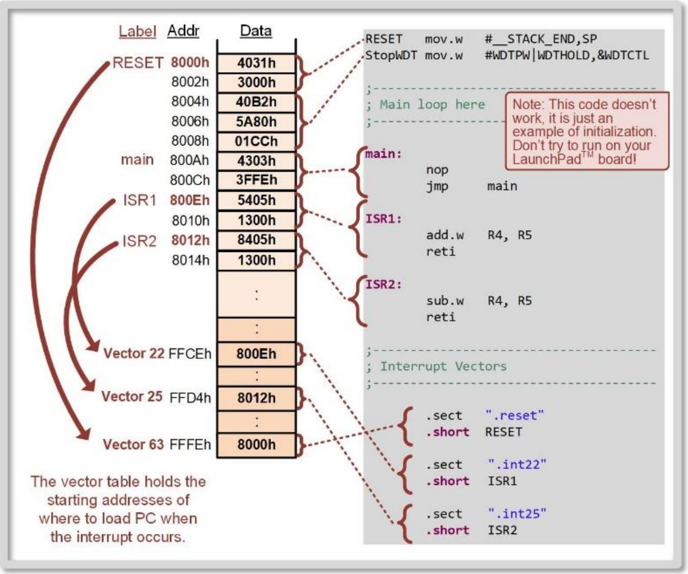
```
RESET   mov.w   #__STACK_END,SP
StopWDT mov.w   #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------
; Main loop here
;-------------------------------------
main:
        nop
        jmp     main
ISR1:
        add.w   R4, R5
        reti

ISR2:
        sub.w   R4, R5
        reti

;-------------------------------------
; Interrupt Vectors
;-------------------------------------

        .sect   ".reset"
        .short  RESET

        .sect   ".int22"
        .short  ISR1

        .sect   ".int25"
        .short  ISR2
```

Note: This code doesn't work, it is just an example of initialization. Don't try to run on your LaunchPad™ board!

11.1.4 Operation of the Stack during an IRQ

- When an IRQ is to be serviced:

1. CPU finishes its current instruction.
2. The next instruction in main program is put into PC.
3. PC and SR are **pushed to stack.**
4. MCU clears the SR so that it can be used by the ISR.
5. MCU retrieves starting address of ISR from the interrupt vector and loads it into PC.



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

The vector table holds the starting addresses of where to load PC when the interrupt occurs.

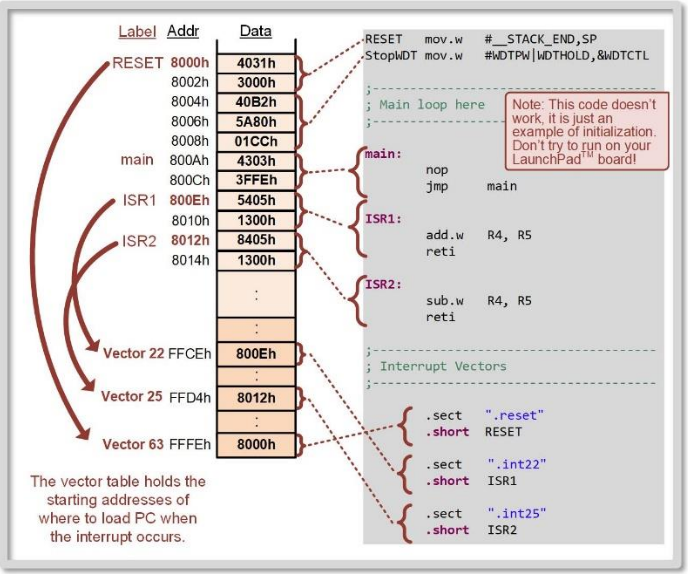
```
RESET   mov.w   #__STACK_END,SP
StopWDT mov.w   #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------------
; Main loop here
;-------------------------------------------

main:
        nop
        jmp     main

ISR1:
        add.w   R4, R5
        reti

ISR2:
        sub.w   R4, R5
        reti

;-------------------------------------------
; Interrupt Vectors
;-------------------------------------------

        .sect   ".reset"
        .short  RESET

        .sect   ".int22"
        .short  ISR1

        .sect   ".int25"
        .short  ISR2
```

Note: This code doesn't work, it is just an example of initialization. Don't try to run on your LaunchPad™ board!

11.1.4 OPERATION OF THE STACK DURING AN IRQ

- When an IRQ is to be serviced:

1. CPU finishes its current instruction.
2. The next instruction in main program is put into PC.
3. PC and SR are **pushed to stack.**
4. MCU clears the SR so that it can be used by the ISR.
5. MCU retrieves starting address of ISR from the interrupt vector and loads it into PC.
6. Execute ISR.



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

The vector table holds the starting addresses of where to load PC when the interrupt occurs.

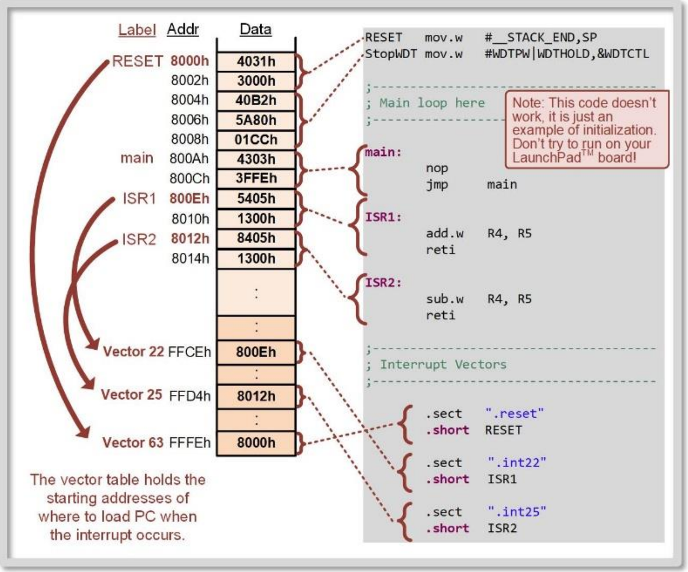
```
RESET   mov.w   #__STACK_END,SP
StopWDT mov.w   #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------------
; Main loop here
;-------------------------------------------

main:
        nop
        jmp     main

ISR1:
        add.w   R4, R5
        reti

ISR2:
        sub.w   R4, R5
        reti

;-------------------------------------------
; Interrupt Vectors
;-------------------------------------------

        .sect   ".reset"
        .short  RESET

        .sect   ".int22"
        .short  ISR1

        .sect   ".int25"
        .short  ISR2
```

Note: This code doesn't work, it is just an example of initialization. Don't try to run on your LaunchPad™ board!

11.1.4 Operation of the STACK during an IRQ

- When an IRQ is to be serviced:

1. CPU finishes its current instruction.
2. The next instruction in main program is put into PC.
3. PC and SR are **pushed to stack.**
4. MCU clears the SR so that it can be used by the ISR.
5. MCU retrieves starting address of ISR from the interrupt vector and loads it into PC.
6. Execute ISR.
7. CPU **pops the SR and PC from the stack.**



Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

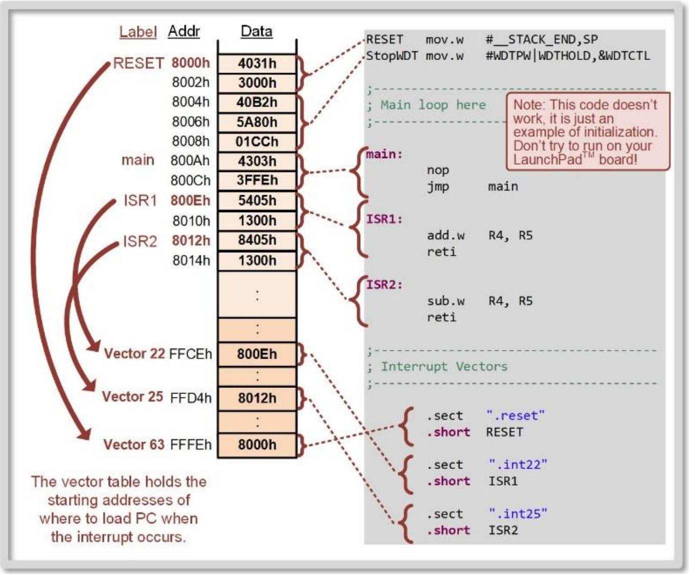
The vector table holds the starting addresses of where to load PC when the interrupt occurs.

```
RESET   mov.w   #__STACK_END,SP
StopWDT mov.w   #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------------
; Main loop here
;-------------------------------------------

main:
        nop
        jmp     main

ISR1:
        add.w   R4, R5
        reti

ISR2:
        sub.w   R4, R5
        reti

;-------------------------------------------
; Interrupt Vectors
;-------------------------------------------

        .sect   ".reset"
        .short  RESET

        .sect   ".int22"
        .short  ISR1

        .sect   ".int25"
        .short  ISR2
```

Note: This code doesn't work, it is just an example of initialization. Don't try to run on your LaunchPad™ board!

11.1.4 OPERATION OF THE STACK DURING AN IRQ

- When an IRQ is to be serviced:
  1. CPU finishes its current instruction.
  2. The next instruction in main program is put into PC.
  3. PC and SR are **pushed to stack**.
  4. MCU clears the SR so that it can be used by the ISR.
  5. MCU retrieves starting address of ISR from the interrupt vector and loads it into PC.
  6. Execute ISR.
  7. CPU **pops the SR and PC from the stack.**
  8. Main program runs from where it left off.



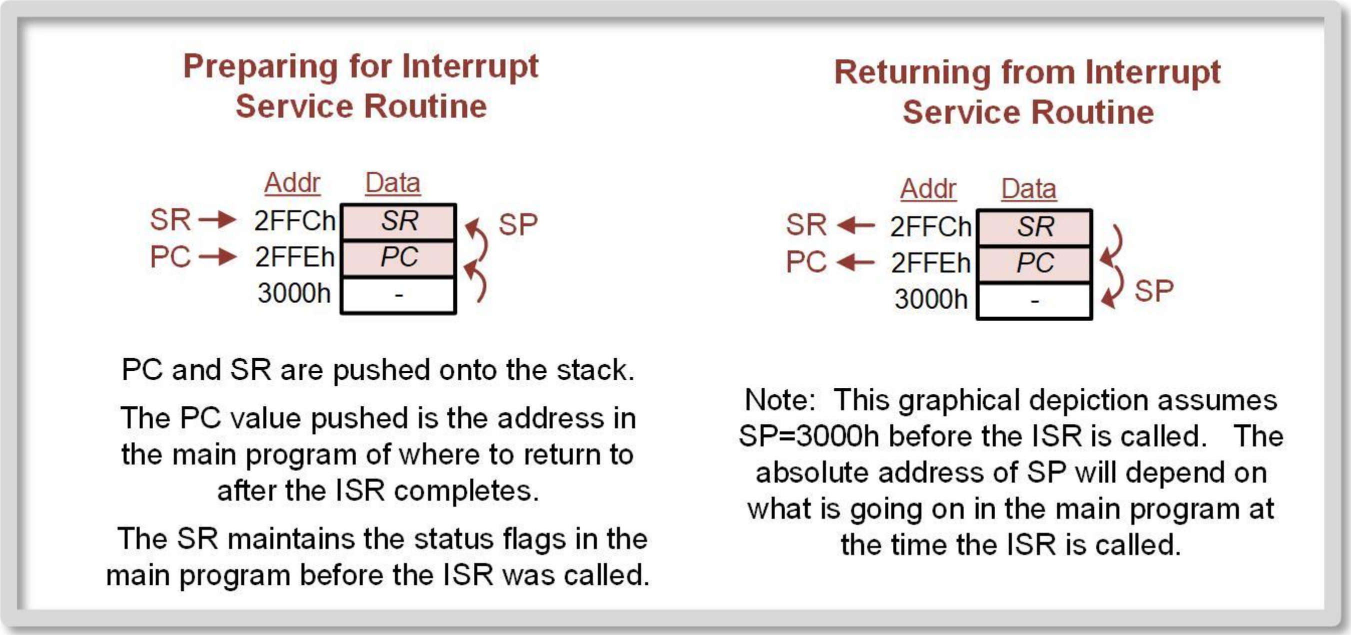
Label	Addr	Data
RESET	8000h	4031h
	8002h	3000h
	8004h	40B2h
	8006h	5A80h
	8008h	01CCh
main	800Ah	4303h
	800Ch	3FFEh
ISR1	800Eh	5405h
	8010h	1300h
ISR2	8012h	8405h
	8014h	1300h
		:
		:
Vector 22	FFCEh	800Eh
		:
Vector 25	FFD4h	8012h
		:
Vector 63	FFFEh	8000h

The vector table holds the starting addresses of where to load PC when the interrupt occurs.

```
RESET   mov.w   #__STACK_END,SP
StopWDT mov.w   #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------------
main:
        nop
        jmp     main
ISR1:
        add.w   R4, R5
        reti

ISR2:
        sub.w   R4, R5
        reti

;-------------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------------

        .sect   ".reset"
        .short  RESET

        .sect   ".int22"
        .short  ISR1

        .sect   ".int25"
        .short  ISR2
```

Note: This code doesn't work, it is just an example of initialization. Don't try to run on your LaunchPad™ board!

11.1.4 OPERATION OF THE STACK DURING AN IRQ



Preparing for Interrupt Service Routine

PC and SR are pushed onto the stack.

The PC value pushed is the address in the main program of where to return to after the ISR completes.

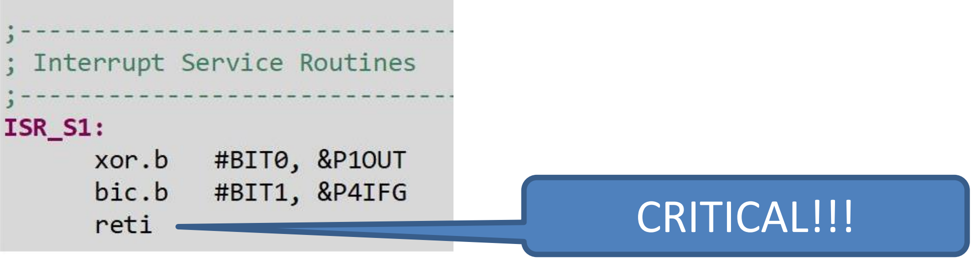
The SR maintains the status flags in the main program before the ISR was called.

Returning from Interrupt Service Routine

Note: This graphical depiction assumes SP=3000h before the ISR is called. The absolute address of SP will depend on what is going on in the main program at the time the ISR is called.

11.1.5 INTERRUPT SERVICE ROUTINES (ISR)

- **Return From Interrupt (reti)** – Always the final instruction in an ISR; pops the SR and PC off the stack in order to return the CPU execution back to the main program.

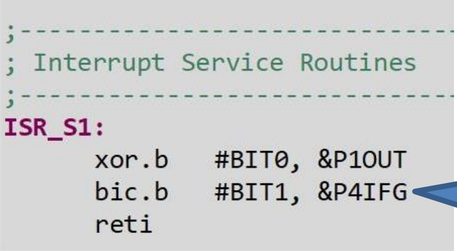


```
;------------------------------
; Interrupt Service Routines
;------------------------------
ISR_S1:
        xor.b   #BIT0, &P1OUT
        bic.b   #BIT1, &P4IFG
        reti
```

CRITICAL!!!

11.1.5 Interrupt Service Routines (ISR)

- **Return From Interrupt (reti)** – Always the final instruction in an ISR; pops the SR and PC off the stack in order to return the CPU execution back to the main program.



```
;-----------------------------
; Interrupt Service Routines
;-----------------------------
ISR_S1:
        xor.b   #BIT0, &P1OUT
        bic.b   #BIT1, &P4IFG
        reti
```

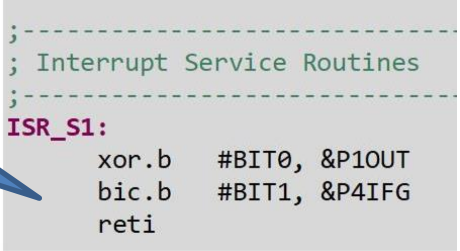
If you forget, the IRQ will continuously trigger.

- Anther critical role of a maskable ISR is that it must clear the peripheral’s local interrupt flag (IFG) that caused the interrupt in the first place. Else, the interrupt will continually run because the flag is still asserted.

11.1.5 INTERRUPT SERVICE ROUTINES (ISR)

- **Return From Interrupt (reti)** – Always the final instruction in an ISR; pops the SR and PC off the stack in order to return the CPU execution back to the main program.

Don’t put too much in an ISR



```
;-----------------------------
; Interrupt Service Routines
;-----------------------------
ISR_S1:
        xor.b   #BIT0, &P1OUT
        bic.b   #BIT1, &P4IFG
        reti
```

- Anther critical role of a maskable ISR is that it must clear the peripheral’s local interrupt flag (IFG) that caused the interrupt in the first place. Else, the interrupt will continually run because the flag is still asserted.

- ISRs should be short, fast, and dedicated to only performing the functionality needed by the peripheral at that time.

11.1.6 NESTED INTERRUPTS

- Maskable interrupts are disabled while executing an ISR because GIE = 0.

Image Courtesy of https://neodem.wp.horizon.ac.uk/

11.1.6 Nested Interrupts

- Maskable interrupts are disabled while executing an ISR because GIE = 0.

- Once the ISR completes and the SR is popped off the stack, the original value of GIE = 1 is restored and maskable interrupts are again enabled when the CPU returns to the main program.

Image Courtesy of https://neodem.wp.horizon.ac.uk/

11.1.6 NESTED INTERRUPTS

- Maskable interrupts are disabled while executing an ISR because GIE = 0.
- Once the ISR completes and the SR is popped off the stack, the original value of GIE = 1 is restored and maskable interrupts are again enabled when the CPU returns to the main program.
- System resets and non-maskable interrupts can always interrupt other lower priority ISRs because they are not enabled by GIE.

Image Courtesy of https://neodem.wp.horizon.ac.uk/

11.1.6 Nested Interrupts

- Creating *nested* maskable ISRs is **not recommended** because it can lead to stack overflow or infinite ISR loops.

Image Courtesy of https://neodem.wp.horizon.ac.uk/

11.1.6 NESTED INTERRUPTS

- Creating *nested* maskable ISRs is **not recommended** because it can lead to stack overflow or infinite ISR loops.

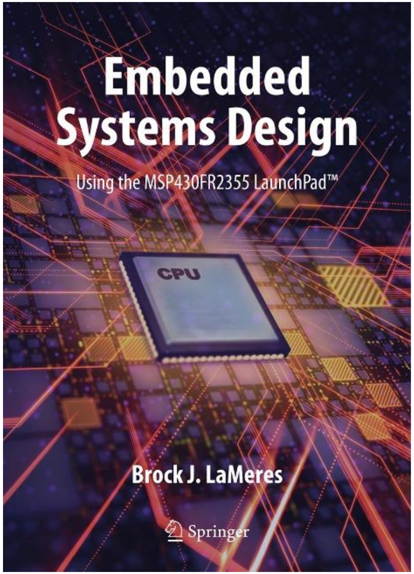
- A better approach is to create ISRs that are short and fast and then rely on the MCU’s prioritization scheme to allow higher priority interrupts to be serviced while lower priority interrupts wait in the pending state until the CPU is ready to act on them.

Image Courtesy of https://neodem.wp.horizon.ac.uk/

Embedded Systems Design

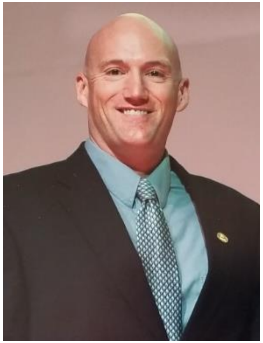
Chapter 11: Introduction to Interrupts

11.1.6 Nested IRQ



▶ Subscribe

www.youtube.com/c/DigitalLogicProgramming_LaMeres

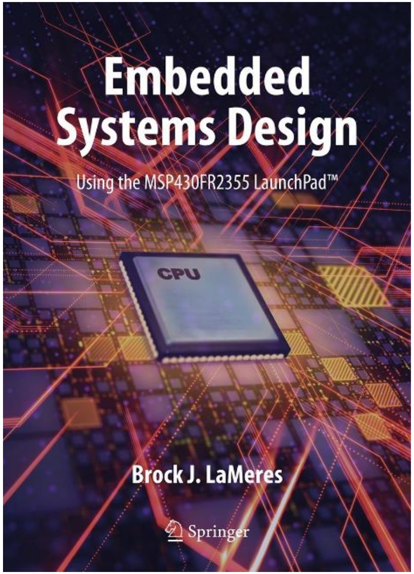


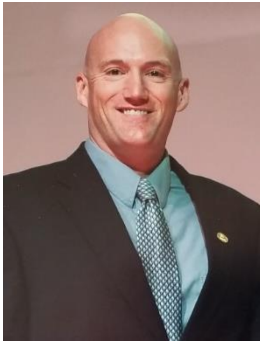
Brock J. LaMeres, Ph.D.

Embedded Systems Design

Chapter 11: Introduction to Interrupts

11.1.7 IRQ Servicing Summary & The Developer's Responsibility

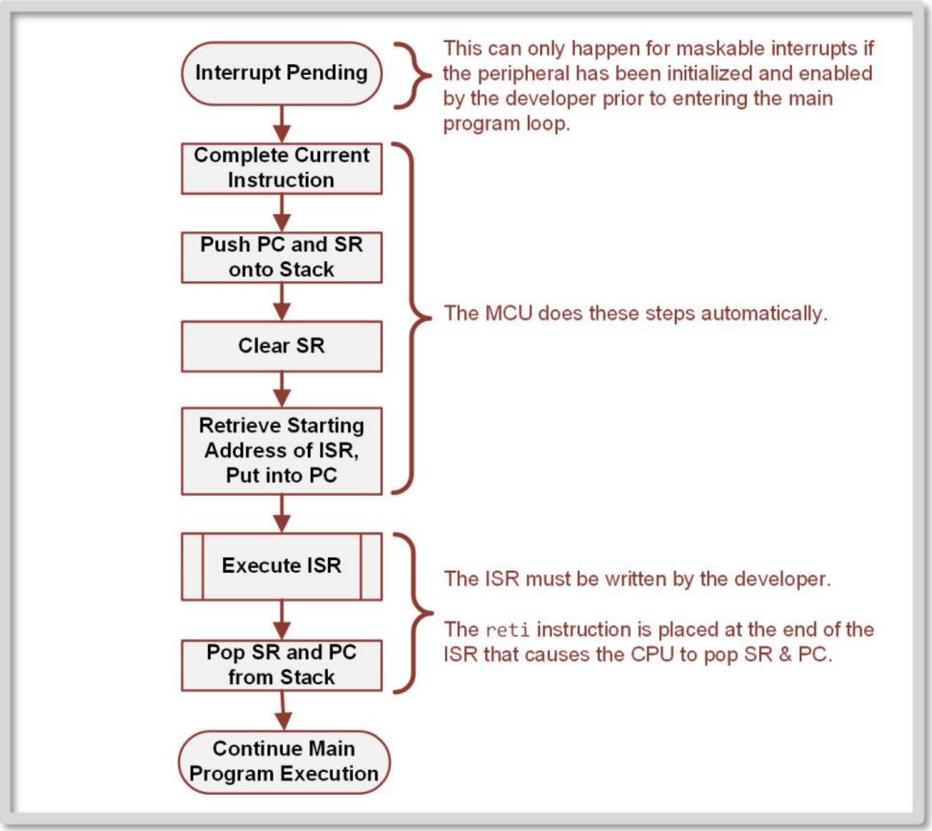




Brock J. LaMeres, Ph.D.

11.1.7 INTERRUPT SERVICING SUMMARY

- Note that some of the steps are taken automatically by the CPU while some are up to the developer.


Interrupt Pending
Complete Current Instruction
Push PC and SR onto Stack
Clear SR
Retrieve Starting Address of ISR, Put into PC
Execute ISR
Pop SR and PC from Stack
Continue Main Program Execution

This can only happen for maskable interrupts if the peripheral has been initialized and enabled by the developer prior to entering the main program loop.

The MCU does these steps automatically.

The ISR must be written by the developer.

The `reti` instruction is placed at the end of the ISR that causes the CPU to pop SR & PC.

11.1.7 INTERRUPT SERVICING SUMMARY

- When using a specific peripheral with a maskable interrupt, the developer has the following responsibilities:

11.1.7 Interrupt Servicing Summary

- When using a specific peripheral with a maskable interrupt, the developer has the following responsibilities:

  1. Configure the peripheral for the desired functionality.

11.1.7 INTERRUPT SERVICING SUMMARY

- When using a specific peripheral with a maskable interrupt, the developer has the following responsibilities:

    1. Configure the peripheral for the desired functionality.
    2. Clear the peripheral’s interrupt flag.

11.1.7 INTERRUPT SERVICING SUMMARY

- When using a specific peripheral with a maskable interrupt, the developer has the following responsibilities:

  1. Configure the peripheral for the desired functionality.
  2. Clear the peripheral’s interrupt flag.
  3. Assert the local interrupt enable (IE) for the peripheral.

11.1.7 INTERRUPT SERVICING SUMMARY

- When using a specific peripheral with a maskable interrupt, the developer has the following responsibilities:

1. Configure the peripheral for the desired functionality.
2. Clear the peripheral’s interrupt flag.
3. Assert the local interrupt enable (IE) for the peripheral.
4. Assert the global interrupt enable (GIE) in the status register.

11.1.7 INTERRUPT SERVICING SUMMARY

- When using a specific peripheral with a maskable interrupt, the developer has the following responsibilities:

1. Configure the peripheral for the desired functionality.
2. Clear the peripheral’s interrupt flag.
3. Assert the local interrupt enable (IE) for the peripheral.
4. Assert the global interrupt enable (GIE) in the status register.
5. Write the ISR with an address label to mark its starting location and the *reti* instruction to denote its end. Remember that the ISR must clear the peripheral doesn’t inadvertently trigger another IRQ.

11.1.7 Interrupt Servicing Summary

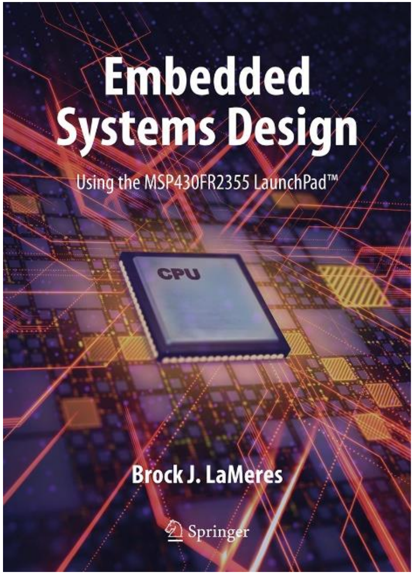
- When using a specific peripheral with a maskable interrupt, the developer has the following responsibilities:

1. Configure the peripheral for the desired functionality.
2. Clear the peripheral's interrupt flag.
3. Assert the local interrupt enable (IE) for the peripheral.
4. Assert the global interrupt enable (GIE) in the status register.
5. Write the ISR with an address label to mark its starting location and the *reti* instruction to denote its end. Remember that the ISR must clear the peripheral doesn't inadvertently trigger another IRQ.
6. Initialize the vector address for the peripheral using the ISR address label and assembler directives.

Embedded Systems Design

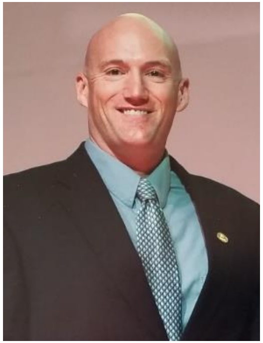
Chapter 11: Introduction to Interrupts

11.1.7 IRQ Servicing Summary & The Developer's Responsibility



▶ Subscribe

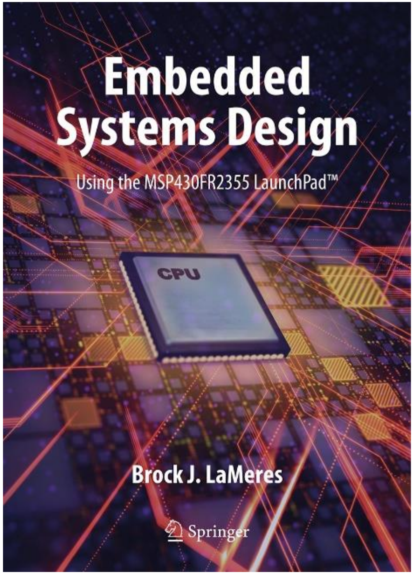
www.youtube.com/c/DigitalLogicProgramming_LaMeres

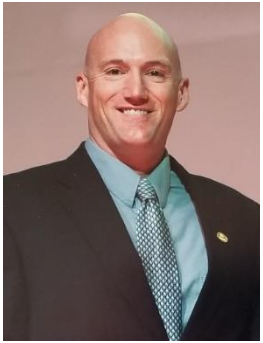


Brock J. LaMeres, Ph.D.

CHAPTER 11: INTRODUCTION TO INTERRUPTS

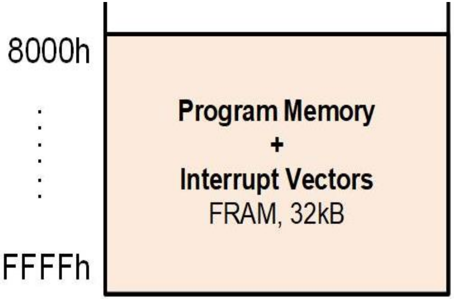
11.1.8 MSP430FR2355 INTERRUPTS

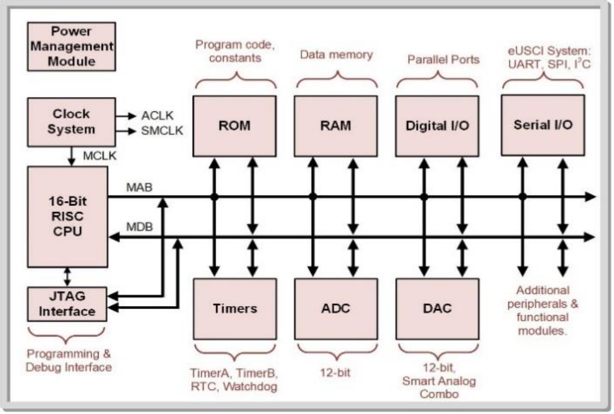




11.1.8 MSP430FR2355 INTERRUPTS

- The MSP430FR2355 implements 25 unique interrupt vector addresses. These are grouped into Resets, NMIs, and Maskable.
- Multiple peripherals will **SHARE** a vector address. If more than one is enabled, then functionality must be placed in the ISR to first determine which flag has been asserted and then execute the appropriate service routine code.



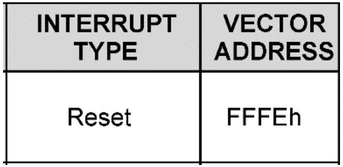


11.1.8 MSP430FR2355 INTERRUPTS

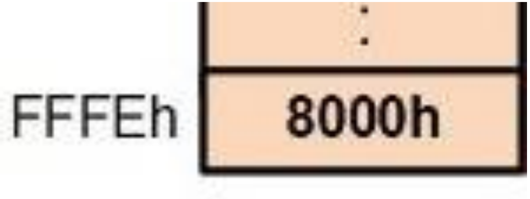
- The highest priority vector address implemented (FFFEh) is dedicated to **resets**.

- This vector is always initialized with the starting address of program memory.

- In the case of the MSP430FR2355, PC is set to **8000h**, which is the beginning of non-volatile FRAM program memory.

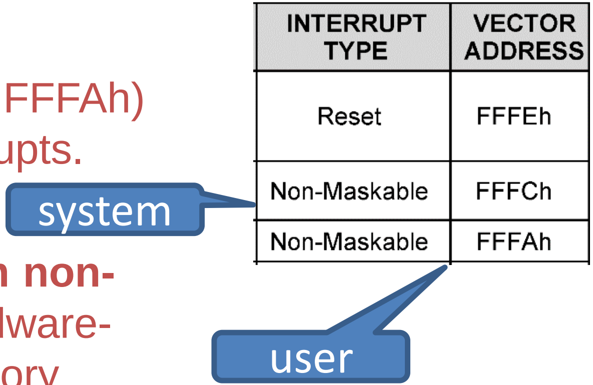


INTERRUPT TYPE	VECTOR ADDRESS
Reset	FFFEh



11.1.8 MSP430FR2355 Interrupts

- The second and third highest priority addresses implemented (FFFCh and FFFAh) are dedicated to non-maskable interrupts.



INTERRUPT TYPE	VECTOR ADDRESS
Reset	FFFEh
Non-Maskable	FFFCh
Non-Maskable	FFFAh

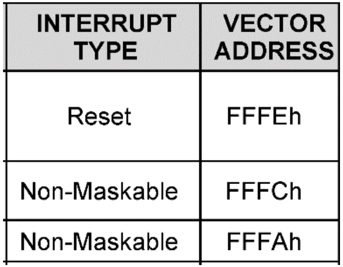
system
user

- The vector FFFCh is used for **system non-maskable interrupts**, which are hardware-level failures such as accessing memory addresses that don’t have systems mapped to them, memory access timing errors, and memory bit-error detection.

- The vector FFFAh is used for **user non-maskable interrupts**, which include an external input trigger and oscillator faults.

11.1.8 MSP430FR2355 Interrupts

- Both NMI interrupt vectors execute ISRs when triggered, so it's the developer's responsibility to ensure the ISRs exist and the vector addresses are initialized.



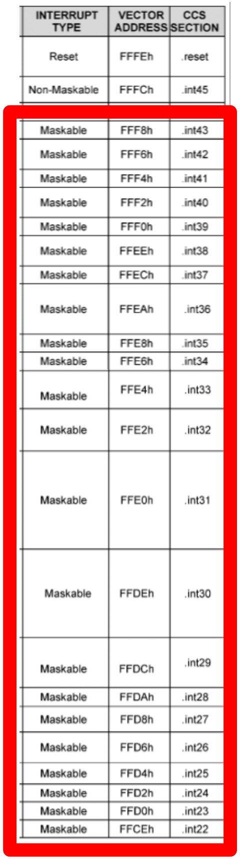
INTERRUPT TYPE	VECTOR ADDRESS
Reset	FFFEh
Non-Maskable	FFFCh
Non-Maskable	FFFAh

- The ISRs for the NMIs are often omitted due to the low likelihood of a system failure.

Proceed at your own risk!

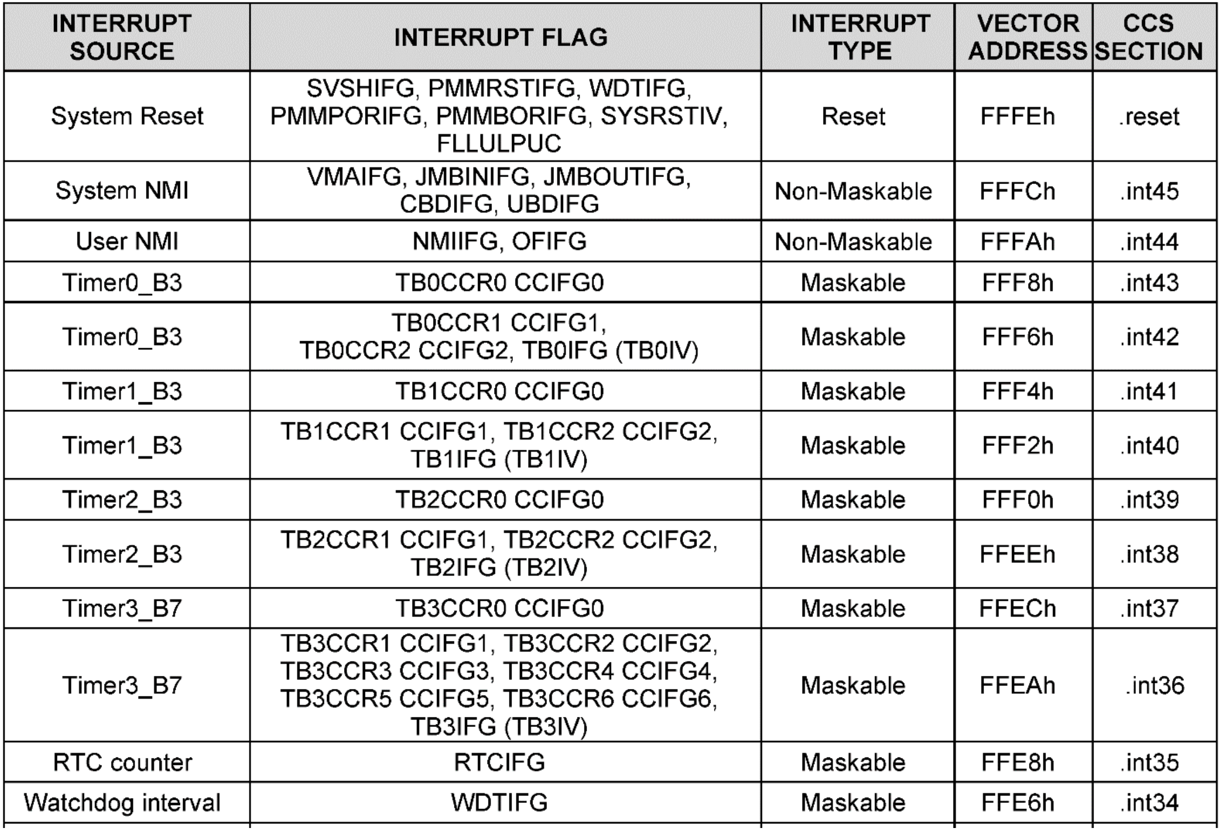
11.1.8 MSP430FR2355 Interrupts

- The remaining 22 interrupt vectors are used for maskable interrupts.
- Only enabled by the developer.
- The MSP430FR2355 assigns:
  - 8x to timers
  - 1x to the real time clock counter
  - 1x for the watchdog timer
  - 4x for the serial communication system
  - 1x for the ADC
  - 1x for the comparator
  - 2x for the smart analog combo DACs
  - 4x for the digital I/O ports.



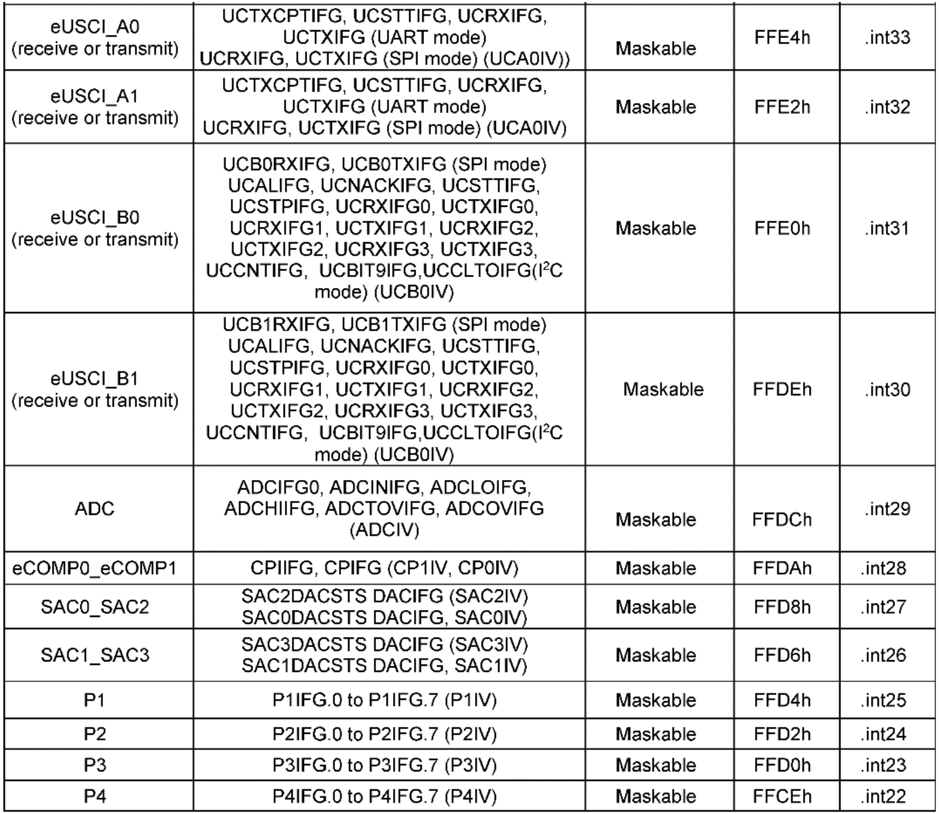
INTERRUPT TYPE	VECTOR ADDRESS	CCS SECTION
Reset	FFFEh	.reset
Non-Maskable	FFFCh	.int45
Maskable	FFF8h	.int43
Maskable	FFF6h	.int42
Maskable	FFF4h	.int41
Maskable	FFF2h	.int40
Maskable	FFF0h	.int39
Maskable	FFEEh	.int38
Maskable	FFECh	.int37
Maskable	FFEAh	.int36
Maskable	FFE8h	.int35
Maskable	FFE6h	.int34
Maskable	FFE4h	.int33
Maskable	FFE2h	.int32
Maskable	FFE0h	.int31
Maskable	FFDEh	.int30
Maskable	FFDCh	.int29
Maskable	FFDAh	.int28
Maskable	FFD8h	.int27
Maskable	FFD6h	.int26
Maskable	FFD4h	.int25
Maskable	FFD2h	.int24
Maskable	FFD0h	.int23
Maskable	FFCEh	.int22

11.1.8 MSP430FR2355 INTERRUPTS



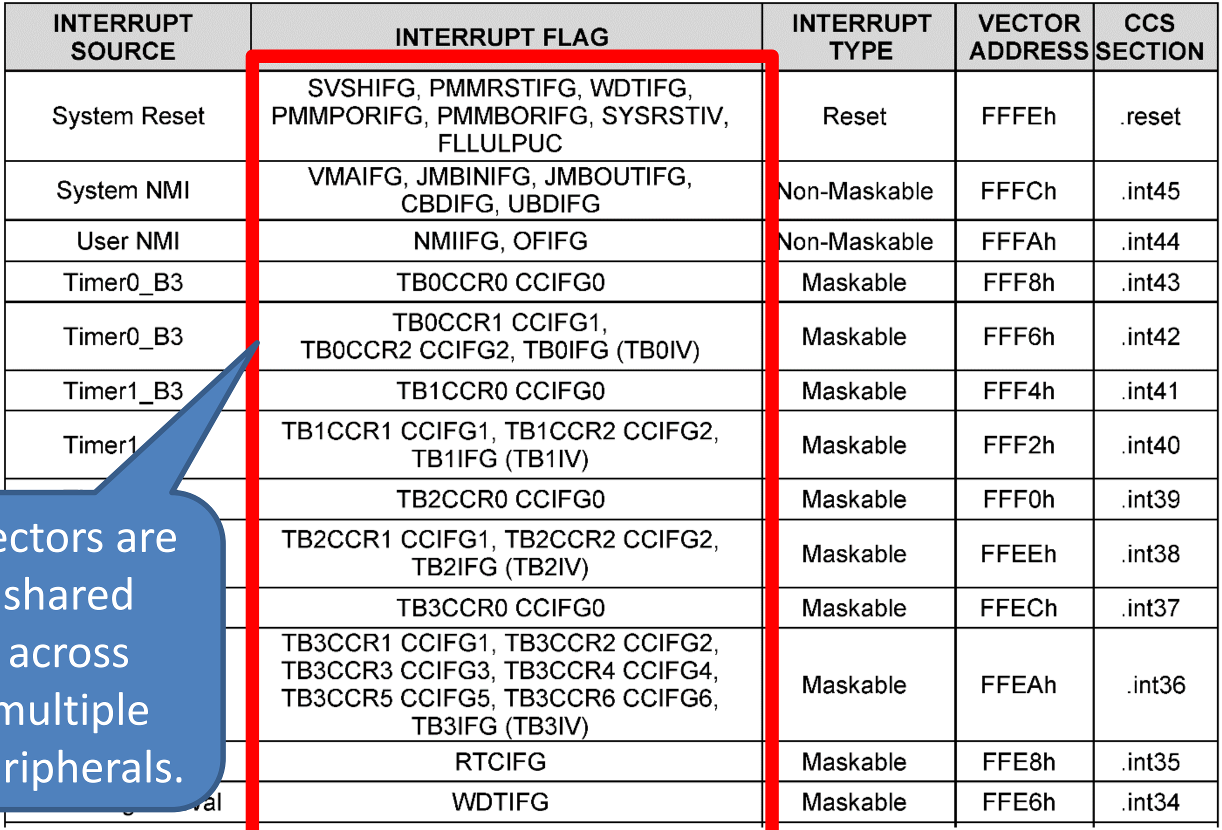
INTERRUPT SOURCE	INTERRUPT FLAG	INTERRUPT TYPE	VECTOR ADDRESS	CCS SECTION
System Reset	SVSHIFG, PMMRSTIFG, WDTIFG, PMMPORIFG, PMMBORIFG, SYSRSTIV, FLLULPUC	Reset	FFFEh	.reset
System NMI	VMAIFG, JMBINIFG, JMBOUTIFG, CBDIFG, UBDIFG	Non-Maskable	FFFCh	.int45
User NMI	NMIIFG, OFIFG	Non-Maskable	FFFAh	.int44
Timer0_B3	TB0CCR0 CCIFG0	Maskable	FFF8h	.int43
Timer0_B3	TB0CCR1 CCIFG1, TB0CCR2 CCIFG2, TB0IFG (TB0IV)	Maskable	FFF6h	.int42
Timer1_B3	TB1CCR0 CCIFG0	Maskable	FFF4h	.int41
Timer1_B3	TB1CCR1 CCIFG1, TB1CCR2 CCIFG2, TB1IFG (TB1IV)	Maskable	FFF2h	.int40
Timer2_B3	TB2CCR0 CCIFG0	Maskable	FFF0h	.int39
Timer2_B3	TB2CCR1 CCIFG1, TB2CCR2 CCIFG2, TB2IFG (TB2IV)	Maskable	FFEEh	.int38
Timer3_B7	TB3CCR0 CCIFG0	Maskable	FFECh	.int37
Timer3_B7	TB3CCR1 CCIFG1, TB3CCR2 CCIFG2, TB3CCR3 CCIFG3, TB3CCR4 CCIFG4, TB3CCR5 CCIFG5, TB3CCR6 CCIFG6, TB3IFG (TB3IV)	Maskable	FFEAh	.int36
RTC counter	RTCIFG	Maskable	FFE8h	.int35
Watchdog interval	WDTIFG	Maskable	FFE6h	.int34

11.1.8 MSP430FR2355 INTERRUPTS



eUSCI_A0 (receive or transmit)	UCTXCPTIFG, UCSTTIFG, UCRXIFG, UCTXIFG (UART mode) UCRXIFG, UCTXIFG (SPI mode) (UCA0IV))	Maskable	FFE4h	.int33
eUSCI_A1 (receive or transmit)	UCTXCPTIFG, UCSTTIFG, UCRXIFG, UCTXIFG (UART mode) UCRXIFG, UCTXIFG (SPI mode) (UCA0IV)	Maskable	FFE2h	.int32
eUSCI_B0 (receive or transmit)	UCB0RXIFG, UCB0TXIFG (SPI mode) UCALIFG, UCNACKIFG, UCSTTIFG, UCSTPIFG, UCRXIFG0, UCTXIFG0, UCRXIFG1, UCTXIFG1, UCRXIFG2, UCTXIFG2, UCRXIFG3, UCTXIFG3, UCCNTIFG, UCBIT9IFG,UCCLTOIFG(I^{2}C mode) (UCB0IV)	Maskable	FFE0h	.int31
eUSCI_B1 (receive or transmit)	UCB1RXIFG, UCB1TXIFG (SPI mode) UCALIFG, UCNACKIFG, UCSTTIFG, UCSTPIFG, UCRXIFG0, UCTXIFG0, UCRXIFG1, UCTXIFG1, UCRXIFG2, UCTXIFG2, UCRXIFG3, UCTXIFG3, UCCNTIFG, UCBIT9IFG,UCCLTOIFG(I^{2}C mode) (UCB0IV)	Maskable	FFDEh	.int30
ADC	ADCIFG0, ADCINIFG, ADCLOIFG, ADCHIIFG, ADCTOVIFG, ADCOVIFG (ADCIV)	Maskable	FFDCh	.int29
eCOMP0_eCOMP1	CPIIFG, CPIFG (CP1IV, CP0IV)	Maskable	FFDAh	.int28
SAC0_SAC2	SAC2DACSTS DACIFG (SAC2IV) SAC0DACSTS DACIFG, SAC0IV)	Maskable	FFD8h	.int27
SAC1_SAC3	SAC3DACSTS DACIFG (SAC3IV) SAC1DACSTS DACIFG, SAC1IV)	Maskable	FFD6h	.int26
P1	P1IFG.0 to P1IFG.7 (P1IV)	Maskable	FFD4h	.int25
P2	P2IFG.0 to P2IFG.7 (P2IV)	Maskable	FFD2h	.int24
P3	P3IFG.0 to P3IFG.7 (P3IV)	Maskable	FFD0h	.int23
P4	P4IFG.0 to P4IFG.7 (P4IV)	Maskable	FFCEh	.int22

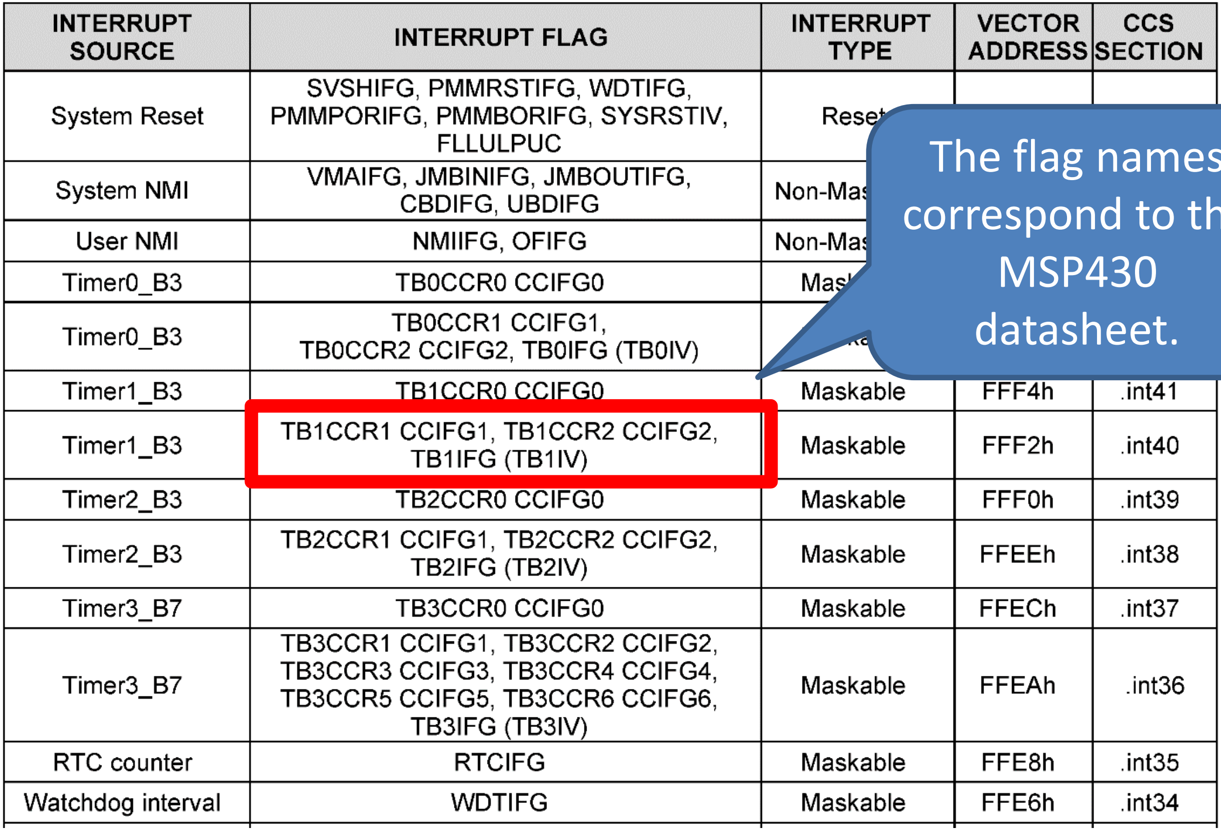
11.1.8 MSP430FR2355 INTERRUPTS (THINGS TO CONSIDER)



INTERRUPT SOURCE	INTERRUPT FLAG	INTERRUPT TYPE	VECTOR ADDRESS	CCS SECTION
System Reset	SVSHIFG, PMMRSTIFG, WDTIFG, PMMPORIFG, PMMBORIFG, SYSRSTIV, FLLULPUC	Reset	FFFEh	.reset
System NMI	VMAIFG, JMBINIFG, JMBOUTIFG, CBDIFG, UBDIFG	Non-Maskable	FFFCh	.int45
User NMI	NMIIFG, OFIFG	Non-Maskable	FFFAh	.int44
Timer0_B3	TB0CCR0 CCIFG0	Maskable	FFF8h	.int43
Timer0_B3	TB0CCR1 CCIFG1, TB0CCR2 CCIFG2, TB0IFG (TB0IV)	Maskable	FFF6h	.int42
Timer1_B3	TB1CCR0 CCIFG0	Maskable	FFF4h	.int41
Timer1[illegible]	TB1CCR1 CCIFG1, TB1CCR2 CCIFG2, TB1IFG (TB1IV)	Maskable	FFF2h	.int40
[illegible]	TB2CCR0 CCIFG0	Maskable	FFF0h	.int39
[illegible]	TB2CCR1 CCIFG1, TB2CCR2 CCIFG2, TB2IFG (TB2IV)	Maskable	FFEEh	.int38
[illegible]	TB3CCR0 CCIFG0	Maskable	FFECh	.int37
[illegible]	TB3CCR1 CCIFG1, TB3CCR2 CCIFG2, TB3CCR3 CCIFG3, TB3CCR4 CCIFG4, TB3CCR5 CCIFG5, TB3CCR6 CCIFG6, TB3IFG (TB3IV)	Maskable	FFEAh	.int36
[illegible]	RTCIFG	Maskable	FFE8h	.int35
[illegible]val	WDTIFG	Maskable	FFE6h	.int34

Vectors are shared across multiple peripherals.

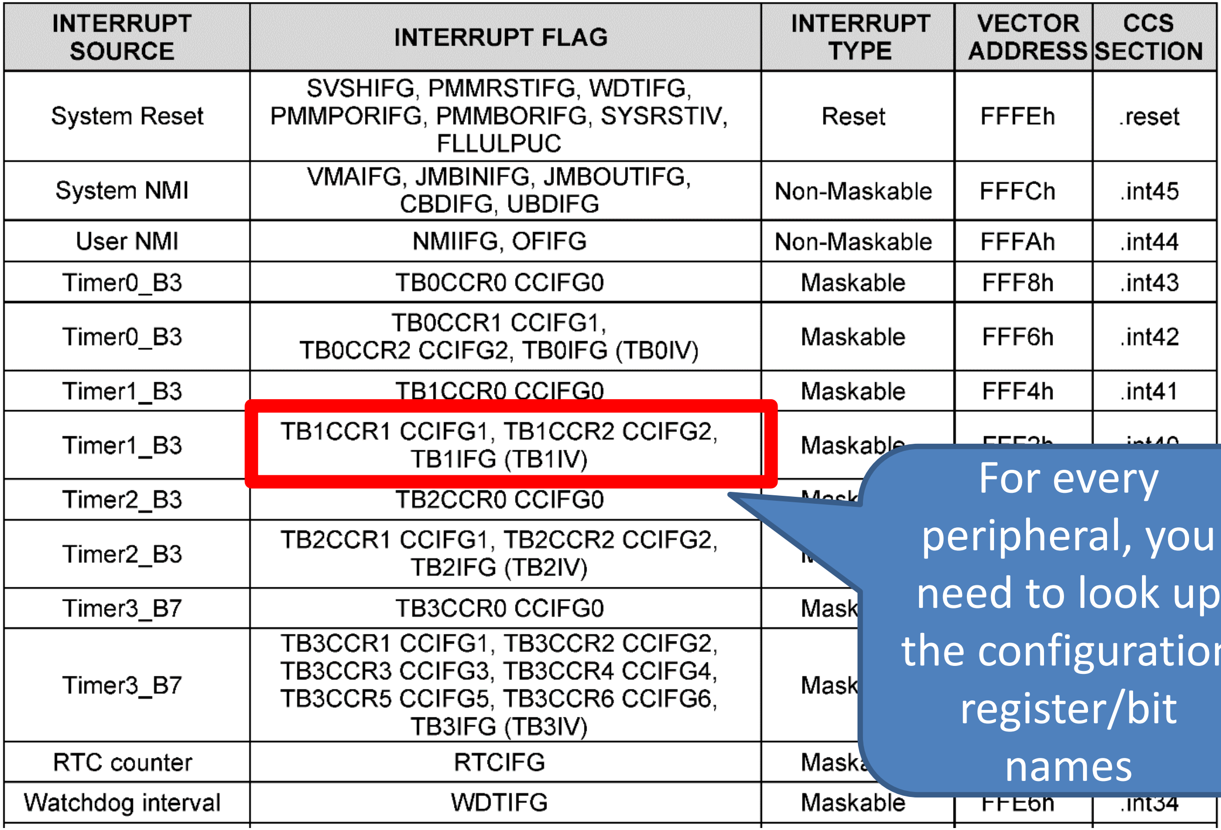
11.1.8 MSP430FR2355 Interrupts (things to consider)



INTERRUPT SOURCE	INTERRUPT FLAG	INTERRUPT TYPE	VECTOR ADDRESS	CCS SECTION
System Reset	SVSHIFG, PMMRSTIFG, WDTIFG, PMMPORIFG, PMMBORIFG, SYSRSTIV, FLLULPUC	Reset		
System NMI	VMAIFG, JMBINIFG, JMBOUTIFG, CBDIFG, UBDIFG	Non-Mas		
User NMI	NMIIFG, OFIFG	Non-Mas		
Timer0_B3	TB0CCR0 CCIFG0	Mas		
Timer0_B3	TB0CCR1 CCIFG1, TB0CCR2 CCIFG2, TB0IFG (TB0IV)			
Timer1_B3	TB1CCR0 CCIFG0	Maskable	FFF4h	.int41
Timer1_B3	TB1CCR1 CCIFG1, TB1CCR2 CCIFG2, TB1IFG (TB1IV)	Maskable	FFF2h	.int40
Timer2_B3	TB2CCR0 CCIFG0	Maskable	FFF0h	.int39
Timer2_B3	TB2CCR1 CCIFG1, TB2CCR2 CCIFG2, TB2IFG (TB2IV)	Maskable	FFEEh	.int38
Timer3_B7	TB3CCR0 CCIFG0	Maskable	FFECh	.int37
Timer3_B7	TB3CCR1 CCIFG1, TB3CCR2 CCIFG2, TB3CCR3 CCIFG3, TB3CCR4 CCIFG4, TB3CCR5 CCIFG5, TB3CCR6 CCIFG6, TB3IFG (TB3IV)	Maskable	FFEAh	.int36
RTC counter	RTCIFG	Maskable	FFE8h	.int35
Watchdog interval	WDTIFG	Maskable	FFE6h	.int34

The flag names correspond to the MSP430 datasheet.

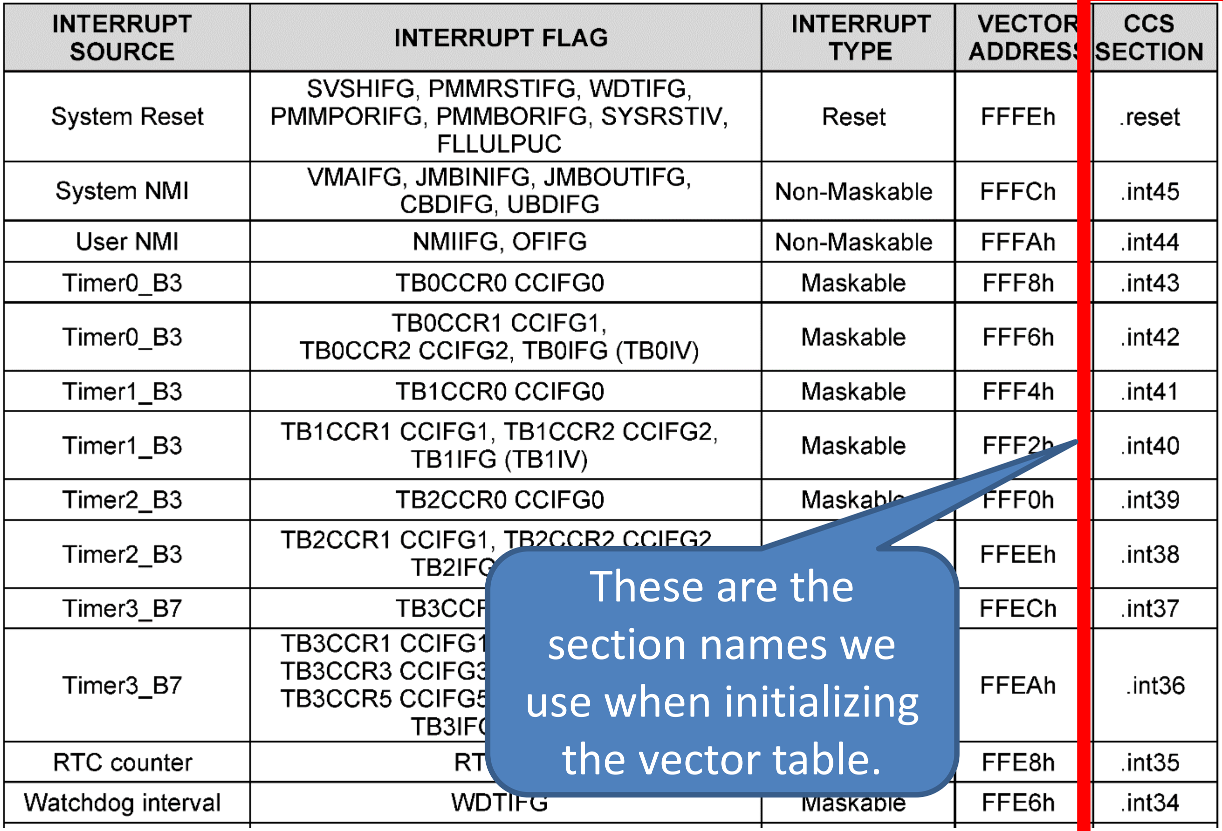
11.1.8 MSP430FR2355 INTERRUPTS (THINGS TO CONSIDER)



INTERRUPT SOURCE	INTERRUPT FLAG	INTERRUPT TYPE	VECTOR ADDRESS	CCS SECTION
System Reset	SVSHIFG, PMMRSTIFG, WDTIFG, PMMPORIFG, PMMBORIFG, SYSRSTIV, FLLULPUC	Reset	FFFEh	.reset
System NMI	VMAIFG, JMBINIFG, JMBOUTIFG, CBDIFG, UBDIFG	Non-Maskable	FFFCh	.int45
User NMI	NMIIFG, OFIFG	Non-Maskable	FFFAh	.int44
Timer0_B3	TB0CCR0 CCIFG0	Maskable	FFF8h	.int43
Timer0_B3	TB0CCR1 CCIFG1, TB0CCR2 CCIFG2, TB0IFG (TB0IV)	Maskable	FFF6h	.int42
Timer1_B3	TB1CCR0 CCIFG0	Maskable	FFF4h	.int41
Timer1_B3	TB1CCR1 CCIFG1, TB1CCR2 CCIFG2, TB1IFG (TB1IV)	Maskable	FFF2h	.int40
Timer2_B3	TB2CCR0 CCIFG0	Mask		
Timer2_B3	TB2CCR1 CCIFG1, TB2CCR2 CCIFG2, TB2IFG (TB2IV)	M		
Timer3_B7	TB3CCR0 CCIFG0	Mask		
Timer3_B7	TB3CCR1 CCIFG1, TB3CCR2 CCIFG2, TB3CCR3 CCIFG3, TB3CCR4 CCIFG4, TB3CCR5 CCIFG5, TB3CCR6 CCIFG6, TB3IFG (TB3IV)	Mask		
RTC counter	RTCIFG	Maska		
Watchdog interval	WDTIFG	Maskable	FFE6h	.int34

For every peripheral, you need to look up the configuration register/bit names

11.1.8 MSP430FR2355 INTERRUPTS (THINGS TO CONSIDER)



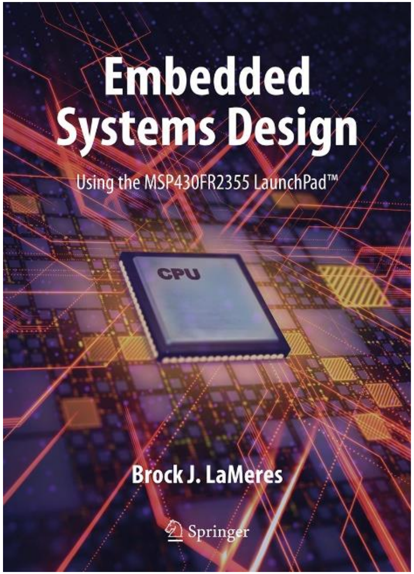
INTERRUPT SOURCE	INTERRUPT FLAG	INTERRUPT TYPE	VECTOR ADDRESS	CCS SECTION
System Reset	SVSHIFG, PMMRSTIFG, WDTIFG, PMMPORIFG, PMMBORIFG, SYSRSTIV, FLLULPUC	Reset	FFFEh	.reset
System NMI	VMAIFG, JMBINIFG, JMBOUTIFG, CBDIFG, UBDIFG	Non-Maskable	FFFCh	.int45
User NMI	NMIIFG, OFIFG	Non-Maskable	FFFAh	.int44
Timer0_B3	TB0CCR0 CCIFG0	Maskable	FFF8h	.int43
Timer0_B3	TB0CCR1 CCIFG1, TB0CCR2 CCIFG2, TB0IFG (TB0IV)	Maskable	FFF6h	.int42
Timer1_B3	TB1CCR0 CCIFG0	Maskable	FFF4h	.int41
Timer1_B3	TB1CCR1 CCIFG1, TB1CCR2 CCIFG2, TB1IFG (TB1IV)	Maskable	FFF2h	.int40
Timer2_B3	TB2CCR0 CCIFG0	Maskable	FFF0h	.int39
Timer2_B3	TB2CCR1 CCIFG1, TB2CCR2 CCIFG2 TB2IF[illegible]	[illegible]	FFEEh	.int38
Timer3_B7	TB3CC[illegible]	[illegible]	FFECh	.int37
Timer3_B7	TB3CCR1 CCIFG1[illegible] TB3CCR3 CCIFG3[illegible] TB3CCR5 CCIFG5[illegible] TB3IF[illegible]	[illegible]	FFEAh	.int36
RTC counter	RT[illegible]	[illegible]	FFE8h	.int35
Watchdog interval	WDTIFG	Maskable	FFE6h	.int34

These are the section names we use when initializing the vector table.

EMBEDDED SYSTEMS DESIGN

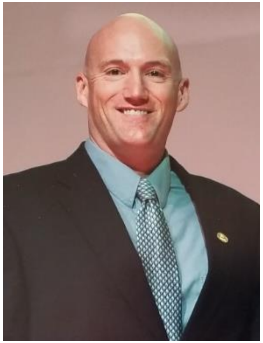
CHAPTER 11: INTRODUCTION TO INTERRUPTS

11.1.8 MSP430FR2355 INTERRUPTS



Subscribe

www.youtube.com/c/DigitalLogicProgramming_LaMeres

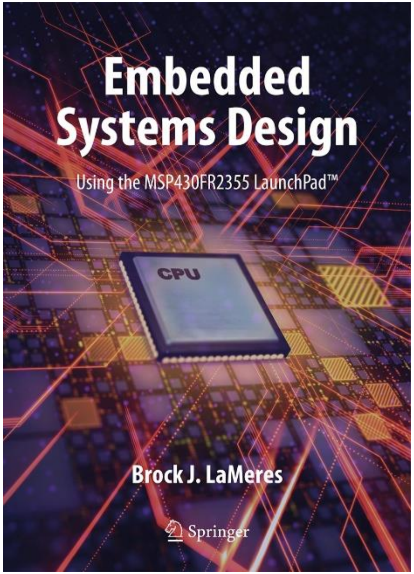


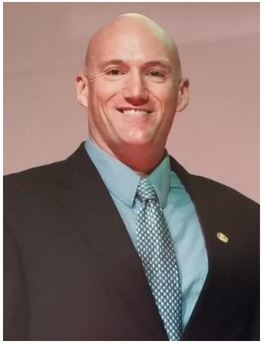
BROCK J. LAMERES, PH.D.

Embedded Systems Design

Chapter 11: Introduction to Interrupts

11.2 MSP430FR2355 Port Interrupts - Overview

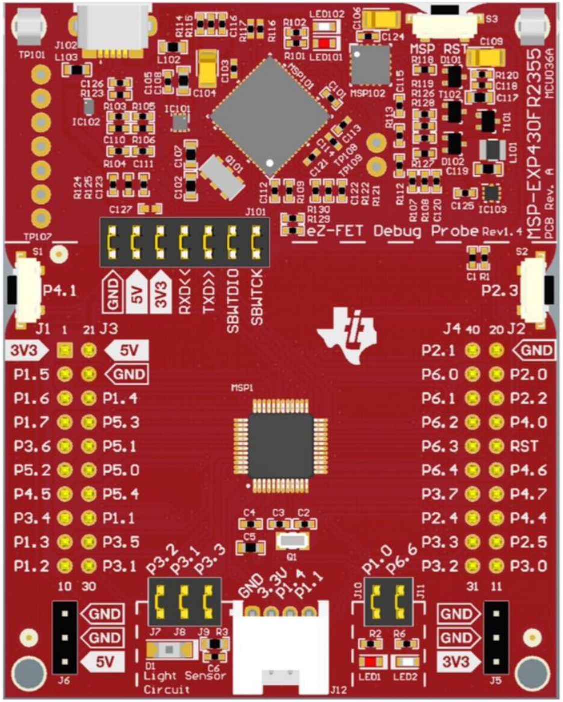




Brock J. LaMeres, Ph.D.

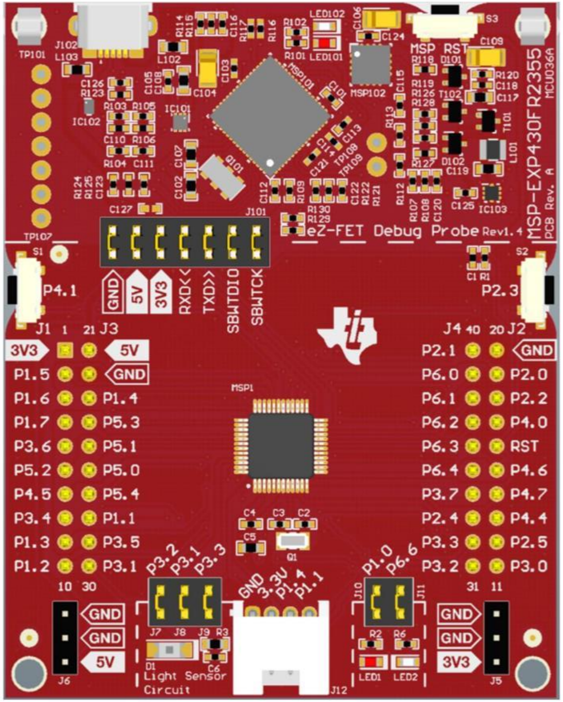
11.2 MSP430FR2355 Port Interrupts

- **Ports 1→4:**
  - Trigger an interrupt when configured as an input and there’s a transition on its pin.
  - Each bit on within each port all share the port’s vector.
  - Developer must determine which bit triggered the interrupt manually and which part of the associated ISR to execute to service that bit.



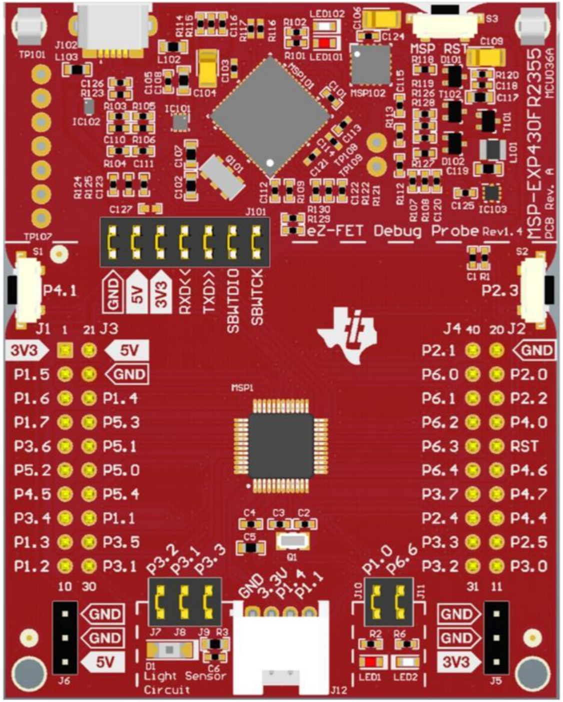
11.2 MSP430FR2355 PORT INTERRUPTS

- **Port Px Interrupt Enable (PxIE)** – local enable for the port interrupts.
    - **PxIE = 0** – interrupt disabled
    - **PxIE = 1** – interrupt enabled
- PxIE is cleared on reset, disabling all port interrupts.
- Since port interrupts are maskable, the global interrupt for all bits is the GIE in the status register.



11.2 MSP430FR2355 Port Interrupts

- **Port Px Interrupt Flag (PxIFG)** – flags for the port interrupts.
  - **PxIFG = 0** – flag unasserted
  - **PxIE = 1** – flag asserted
- PxIFG = 0 upon reset.
- Once a port IRQ is serviced, the bit’s flag needs to be cleared in PxIFG by the developer.

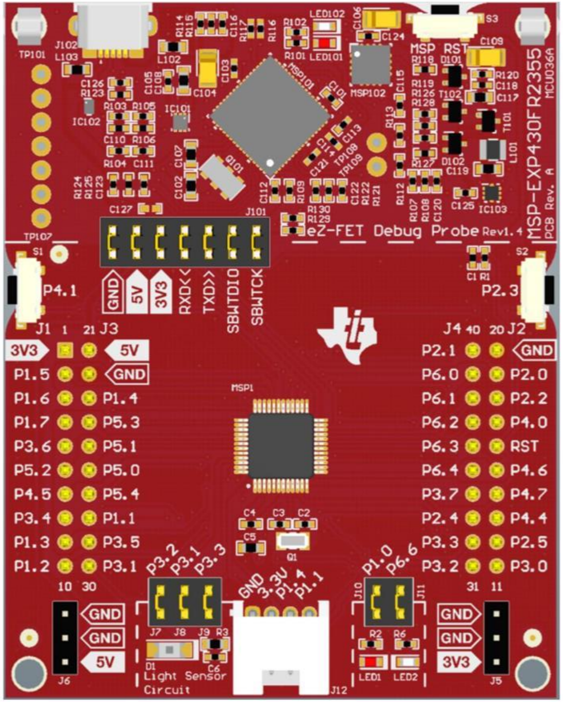


11.2 MSP430FR2355 PORT INTERRUPTS

- **Port Px Interrupt Vector Word (PxIV)** – used to indicate the priority when simultaneous port IRQs have occurred.

- PxIV is loaded with a unique number corresponding to the bit that just triggered an IRQ w/ automatic priority.

- The values it takes on represent which of the 8 inputs is pending with the highest priority: bit0 = 02h, bit1 = 04h, bit2 = 06h, bit3 = 08h, bit4 = 0Ah, bit5 = 0Ch, bit6 = 0Eh, and bit7 = 10h.

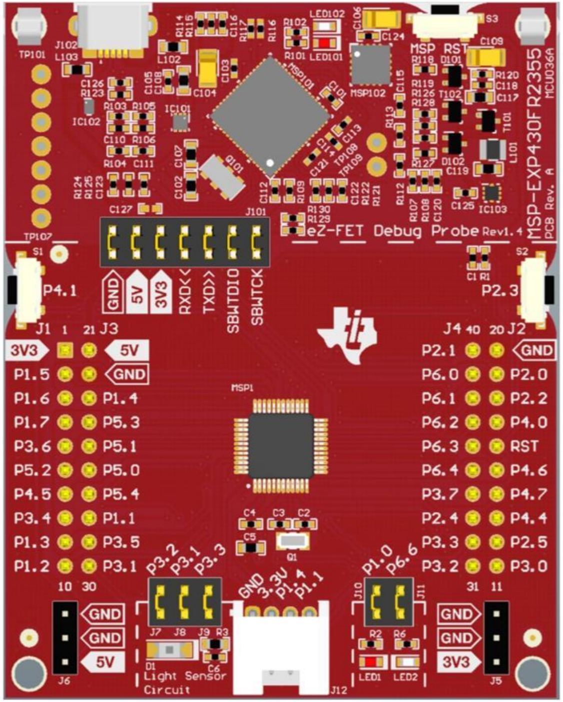


11.2 MSP430FR2355 Port Interrupts

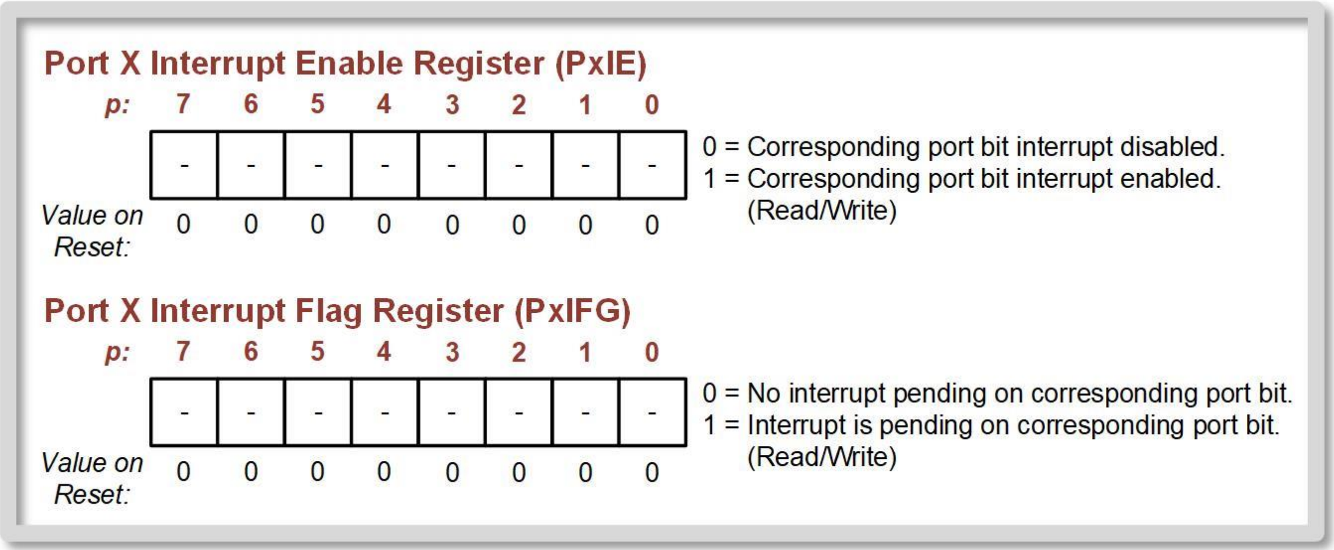
- **Port Px Interrupt Vector Word (PxIV)** cont…
- This register is read-only; however, any access to PxIV (either read or write) will clear port interrupt flag in PxIFG corresponding to the highest priority being displayed in PxIV.
- Once the highest priority flag is cleared, it’s expected that the ISR will complete and the next highest priority flag will trigger and the next highest priority bit will be serviced within the same vector address.
- Note that using this prioritization feature is optional and the developer can simply use PxIFG to determine which port bit has triggered the interrupt and which one to service first.

11.2 MSP430FR2355 PORT INTERRUPTS

- **Port Px Interrupt Edge (PxIES)** – ability to select which transition polarity triggers the interrupt.
  - **PxIES = 0** – triggers on a low-to-high transition of the pin
  - **PxIE = 1** – triggers on a high-to-low transition of the pin



11.2 MSP430FR2355 PORT INTERRUPTS



Port X Interrupt Enable Register (PxIE)

p:	7	6	5	4	3	2	1	0
	-	-	-	-	-	-	-	-
Value on Reset:	0	0	0	0	0	0	0	0

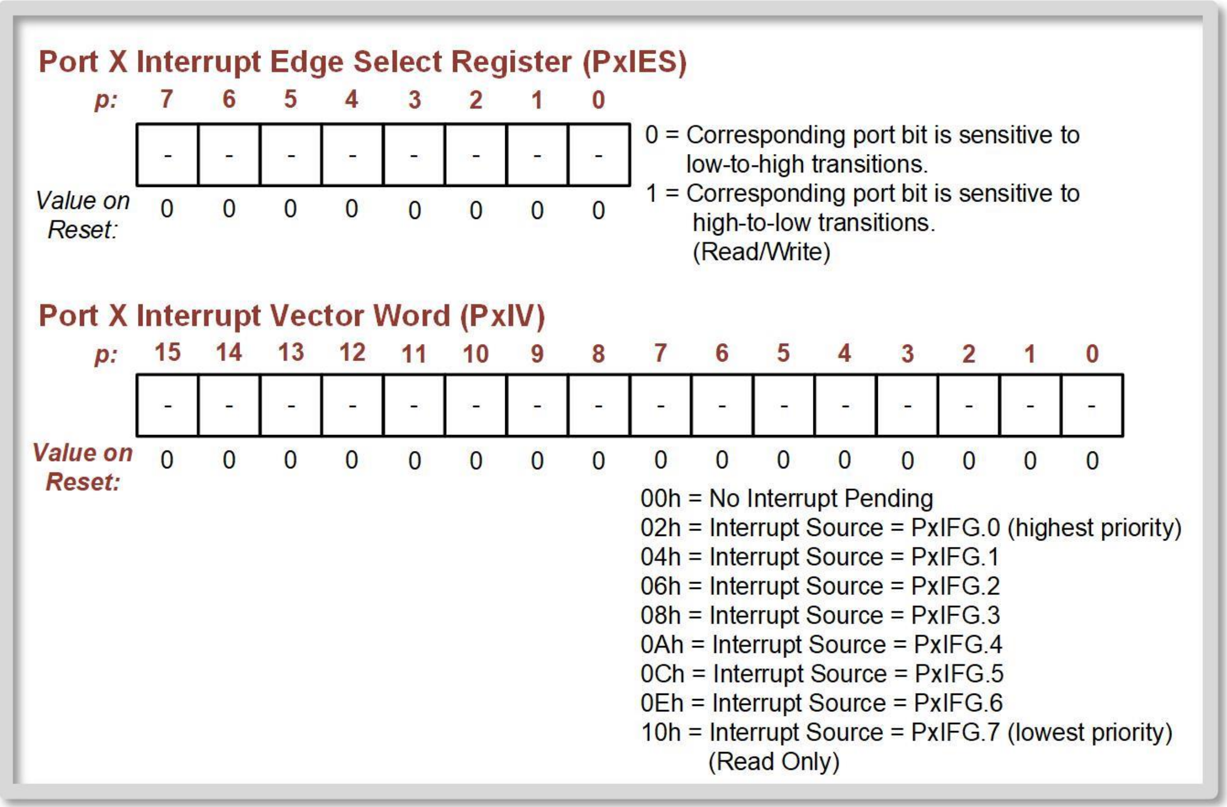
0 = Corresponding port bit interrupt disabled.
1 = Corresponding port bit interrupt enabled.
(Read/Write)

Port X Interrupt Flag Register (PxIFG)

p:	7	6	5	4	3	2	1	0
	-	-	-	-	-	-	-	-
Value on Reset:	0	0	0	0	0	0	0	0

0 = No interrupt pending on corresponding port bit.
1 = Interrupt is pending on corresponding port bit.
(Read/Write)

11.2 MSP430FR2355 PORT INTERRUPTS



Port X Interrupt Edge Select Register (PxIES)

p:	7	6	5	4	3	2	1	0
	-	-	-	-	-	-	-	-
Value on Reset:	0	0	0	0	0	0	0	0

0 = Corresponding port bit is sensitive to low-to-high transitions.
1 = Corresponding port bit is sensitive to high-to-low transitions.
(Read/Write)

Port X Interrupt Vector Word (PxIV)

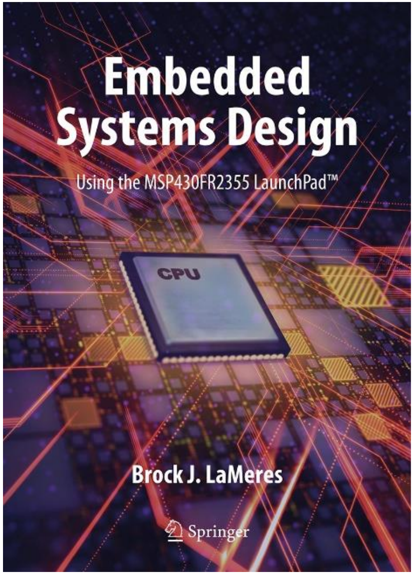
p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Value on Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

00h = No Interrupt Pending
02h = Interrupt Source = PxIFG.0 (highest priority)
04h = Interrupt Source = PxIFG.1
06h = Interrupt Source = PxIFG.2
08h = Interrupt Source = PxIFG.3
0Ah = Interrupt Source = PxIFG.4
0Ch = Interrupt Source = PxIFG.5
0Eh = Interrupt Source = PxIFG.6
10h = Interrupt Source = PxIFG.7 (lowest priority)
(Read Only)

Embedded Systems Design

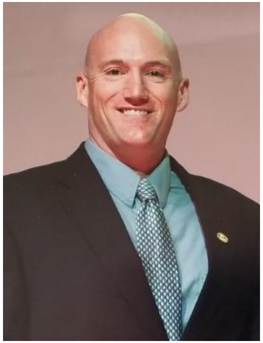
Chapter 11: Introduction to Interrupts

11.2 MSP430FR2355 Port Interrupts Overview



Subscribe

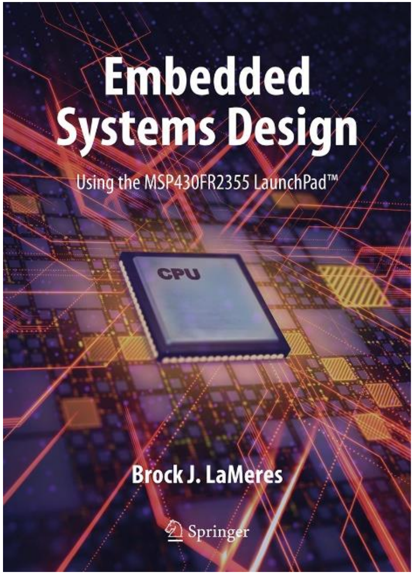
www.youtube.com/c/DigitalLogicProgramming_LaMeres

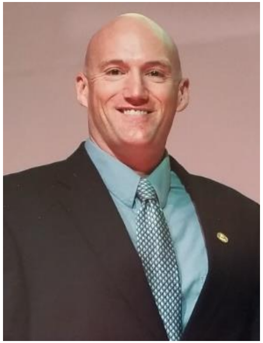


Brock J. LaMeres, Ph.D.

Chapter 11: Introduction to Interrupts

11.2 MSP430FR2355 Port Interrupts – Reading S1 Example

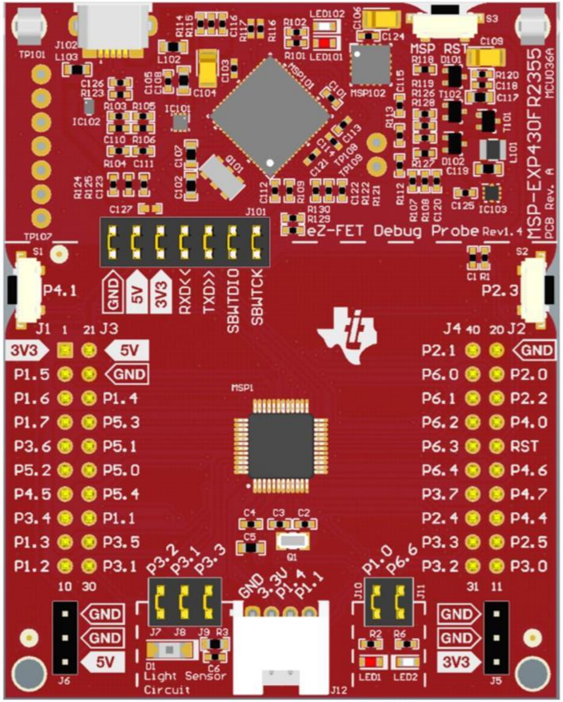




Brock J. LaMeres, Ph.D.

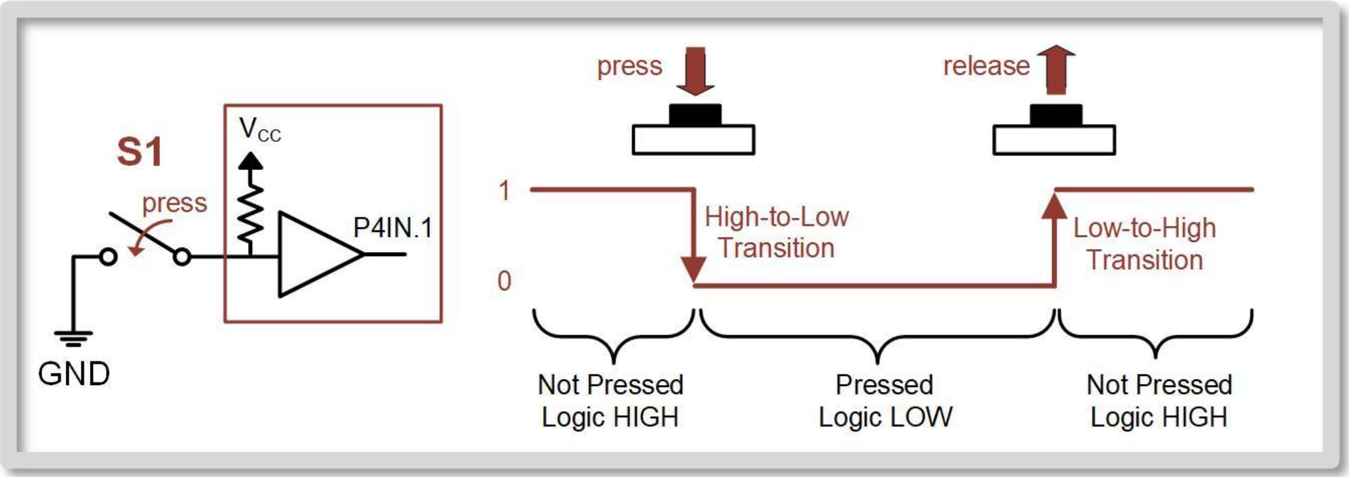
11.2 Configuring S1 to Toggle LED when Pressed

- S1 is connected to port 4 bit 1. While the logic level of S1 can be observed on P4IN.1, when using an interrupt, we don't have to look at this bit. We instead allow a transition to assert an interrupt flag and have the CPU execute an ISR accordingly.



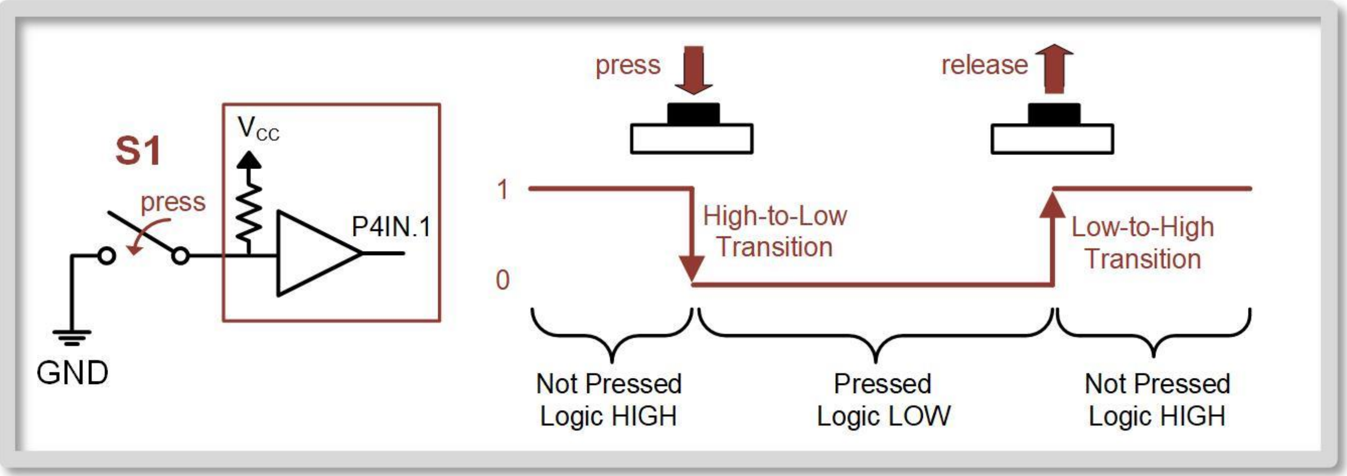
11.2 CONFIGURING S1 TO TOGGLE LED WHEN PRESSED

- S1 is an SPST switch that is connected to ground. This means we need a pull-up resistor on the MCU to provide the logic HIGH state when S1 is not pressed.



11.2 CONFIGURING S1 TO TOGGLE LED WHEN PRESSED

- If we want the IRQ to trigger immediately upon a button press, then we need to configure the interrupt edge sensitivity (**P4IES.1**) to be high-to-low. When S1 is not pressed, P4.1 is at a logic high. When the button is pressed, P4.1 goes to a logic low. If we leave the P4IES.1 sensitivity at its default value of low-to-high sensitivity, the interrupt will only trigger once S1 is released.



11.2 INITIALIZATION SEQUENCE

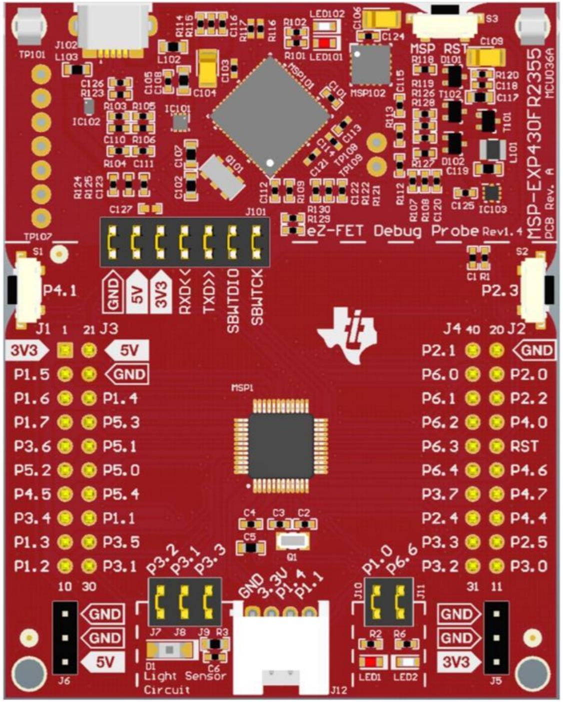
1. Configure Peripheral:
   - Initialize the port direction (PxDIR)
   - Initialize pull-up/down resistor (PxREN)
   - Initialize pull-up/down resistor polarity (PxOUT)
   - Initialize port interrupt edge select (PxIES).

2. Clear LOCKLPM5 bit.

3. Enable the Interrupt
   - Clear the port interrupt flags (PxIFG).
   - Assert the local port interrupt enable (PxIE).
   - Assert the global enable for maskable interrupts (GIE bit in SR).

11.2 CONFIGURING S1 TO TOGGLE LED WHEN PRESSED

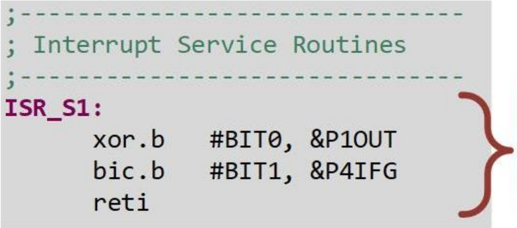
- Since we are only using one bit within P4 to trigger an IRQ, we don't need to use the P4IV register to determine the highest priority bit that caused the IRQ. We will simply use the P4IFG register knowing that we only care about bit1.



11.2 Creating the Interrupt Service Routine

- Our ISR will simply toggle LED1.

- Key things to remember
  - start the ISR with an address label (used when initializing the vector table)
  - clear the IRQ flag (P4IFG.1).
  - return with RETI

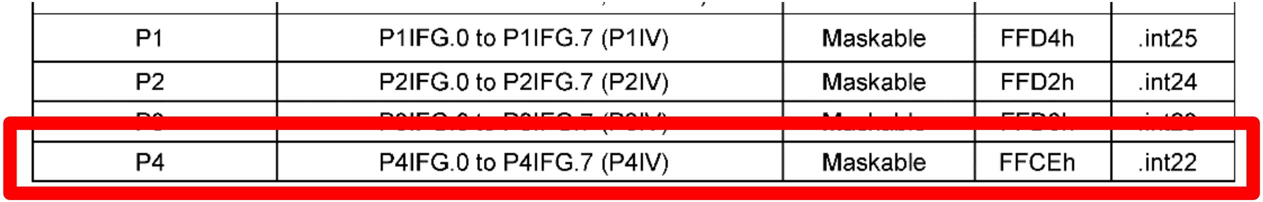


```
;-------------------------------
; Interrupt Service Routines
;-------------------------------
ISR_S1:
        xor.b   #BIT0, &P1OUT
        bic.b   #BIT1, &P4IFG
        reti
```

The ISR toggles LED1 and clears the P4IFG.1 flag. The ISR needs an address label to mark its starting address. The ISR needs to end with `reti` to return to the main program.

11.2 Initializing the Vector Table

- The port 4 interrupt vector address is FFCEh. This has a CCS section name of **.int22**. This is the name we will use when we initialize the vector address using assembler directives.



P1	P1IFG.0 to P1IFG.7 (P1IV)	Maskable	FFD4h	.int25
P2	P2IFG.0 to P2IFG.7 (P2IV)	Maskable	FFD2h	.int24
P3	P3IFG.0 to P3IFG.7 (P3IV)	Maskable	FFD0h	.int23
P4	P4IFG.0 to P4IFG.7 (P4IV)	Maskable	FFCEh	.int22

Example: Using a Port Interrupt on S1 to Toggle LED1

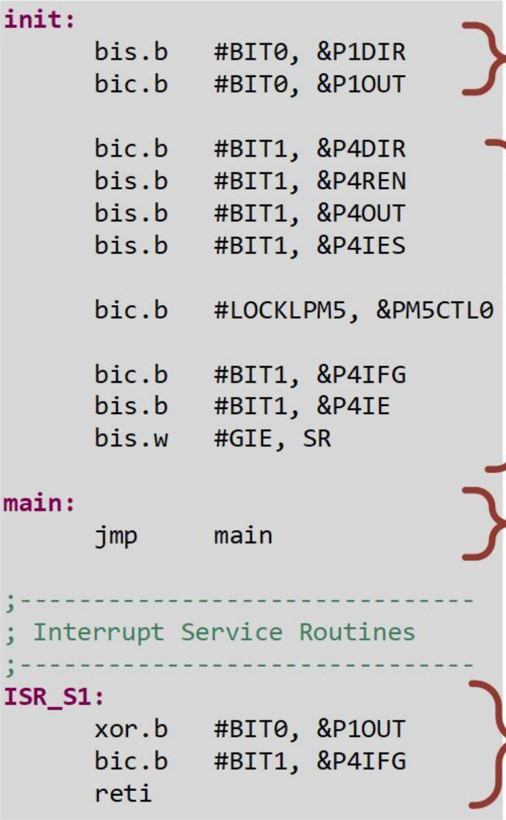
Step 1: Create a new Empty Assembly-only CCS project titled:

Asm_IRQs_Port4_S1

Step 2: Type in the following code into the main.asm file where the comments say “Main loop here.”

Image Courtesy of https://neodem.wp.horizon.ac.uk/

EXAMPLE: USING A PORT INTERRUPT ON S1 TO TOGGLE LED1



```
init:
        bis.b   #BIT0, &P1DIR
        bic.b   #BIT0, &P1OUT

        bic.b   #BIT1, &P4DIR
        bis.b   #BIT1, &P4REN
        bis.b   #BIT1, &P4OUT
        bis.b   #BIT1, &P4IES

        bic.b   #LOCKLPM5, &PM5CTL0

        bic.b   #BIT1, &P4IFG
        bis.b   #BIT1, &P4IE
        bis.w   #GIE, SR

main:
        jmp     main

;------------------------------
; Interrupt Service Routines
;------------------------------
ISR_S1:
        xor.b   #BIT0, &P1OUT
        bic.b   #BIT1, &P4IFG
        reti
```

Setup LED1 to be an output: (P1DIR.0=1)
Put LED1's initial value at zero: (P1OUT.0=0).

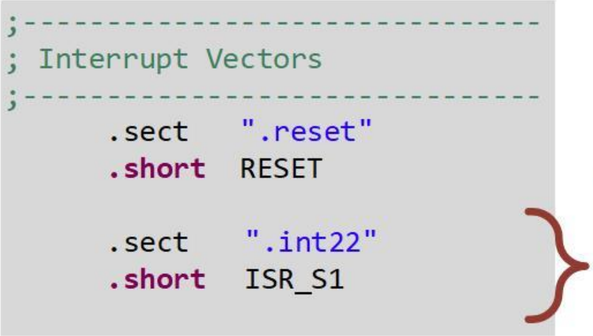
Setup S1 as a port interrupt:

- Set Port Direction to Input (P4DIR.1=0)
- Enable Pull-Up/Down Resistor (P4REN.1=1)
- Configure Resistor as Pull-Up (P4OUT.1=1)
- Set IRQ Sensitivity to High-to-Low (P4IES.1=1)
- Clear LOCKLMP5 Bit
- Clear Interrupt Flag (P4IFG.1=0)
- Assert Local Enable (P4IE.1=1)
- Assert Global Enable (GIE=1)

The main program doesn't do anything but loop forever.

The ISR toggles LED1 and clears the P4IFG.1 flag. The ISR needs an address label to mark its starting address. The ISR needs to end with `reti` to return to the main program.

EXAMPLE: USING A PORT INTERRUPT ON S1 TO TOGGLE LED1



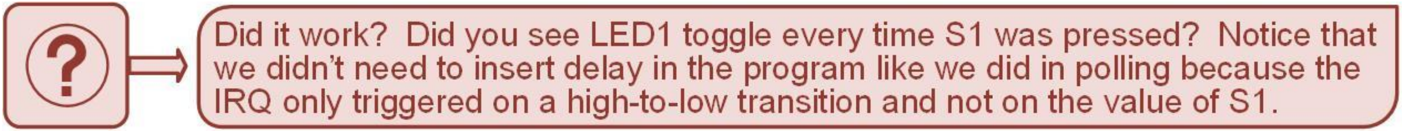
```
;-------------------------------
; Interrupt Vectors
;-------------------------------
      .sect   ".reset"
      .short  RESET

      .sect   ".int22"
      .short  ISR_S1
```

We initialize the vector table for Port4 using the CCS section name found in the linker file. We insert the starting address of the ISR using its address label (ISR_S1).

Example: Using a Port Interrupt on S1 to Toggle LED1

Step 3: Debug your program

Step 4: Run your program. You will see LED1 toggle anytime S1 is pressed.



Did it work? Did you see LED1 toggle every time S1 was pressed? Notice that we didn’t need to insert delay in the program like we did in polling because the IRQ only triggered on a high-to-low transition and not on the value of S1.

Image Courtesy of https://neodem.wp.horizon.ac.uk/

Ex: Observing Low-to-High Port IRQ Sensitivity on S1

Step 1: Create a new Empty Assembly-only CCS project titled:

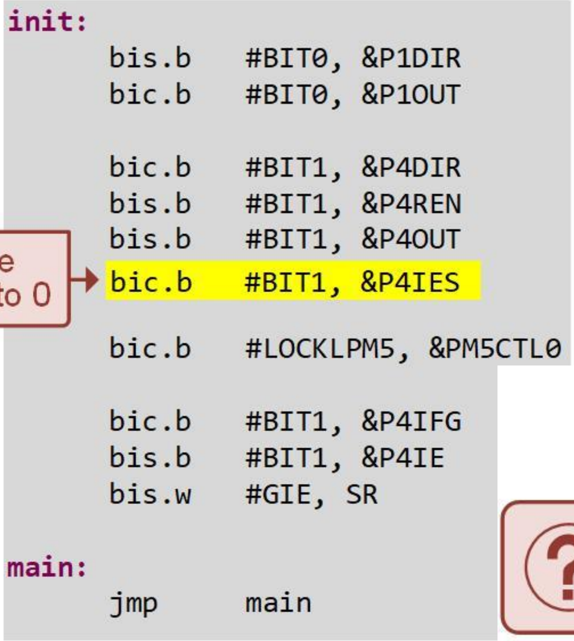
Asm_IRQs_Port4_S1n

Step 2: Copy your code from your last program “Asm_IRQs_Port4_S1” into your new project.

Step 3: Save and debug your program to verify it still works as in the last example.

Image Courtesy of https://neodem.wp.horizon.ac.uk/

Ex: Observing Low-to-High Port IRQ Sensitivity on S1



```
init:
        bis.b   #BIT0, &P1DIR
        bic.b   #BIT0, &P1OUT

        bic.b   #BIT1, &P4DIR
        bis.b   #BIT1, &P4REN
        bis.b   #BIT1, &P4OUT
        bic.b   #BIT1, &P4IES

        bic.b   #LOCKLPM5, &PM5CTL0

        bic.b   #BIT1, &P4IFG
        bis.b   #BIT1, &P4IE
        bis.w   #GIE, SR

main:
        jmp     main
```

Change P4IES.1 to 0

Step 4: Change your setting for your P4IES.1 to a zero.

Step 5: Save and debug your program. Run it and observe the new functionality.

?

Did it work? Did you see LED1 toggle every time S1 was released? Notice that you can press and hold S1 as long as you want and it won't toggle LED1. It only happens when it is released.

Which transition do you think is more responsive to the user, a high-to-low or low-to-high?

Ex: Observing the Impact of Not Clearing the IRQ Flag

Step 1: Open your last example project:

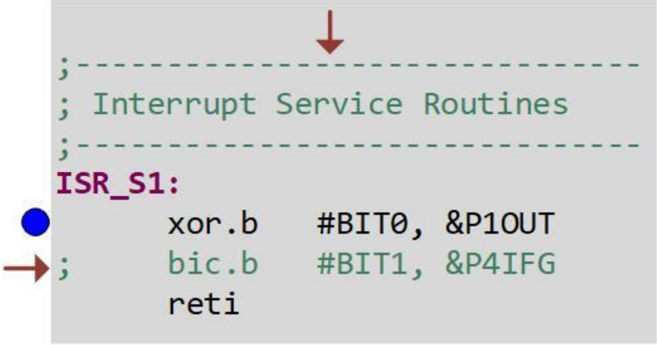
Asm_IRQs_Port4_S1n

Step 2: In your ISR, comment out the instruction that clears the P4IFG.1 flag.

Step 3: Save and debug your program. Set a breakpoint at the first instruction in your ISR. Run your program.

Image Courtesy of https://neodem.wp.horizon.ac.uk/

Ex: Observing the Impact of Not Clearing the IRQ Flag



```
;------------------------------
; Interrupt Service Routines
;------------------------------
ISR_S1:
        xor.b   #BIT0, &P1OUT
;       bic.b   #BIT1, &P4IFG
        reti
```

Step 4: Now press and release S1. Your program will run to the breakpoint in your ISR and pause.

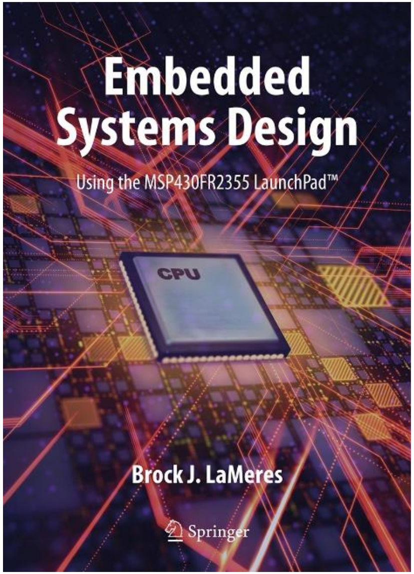
Step 5: Now step your program. You will see that the ISR continually triggers and you can never get out of it.

Do you see the problem? Not remembering to clear the flag in the ISR is one of the most common mistakes developers make when using IRQs. They are very hard to debug without setting breakpoints in the ISR.

Embedded Systems Design

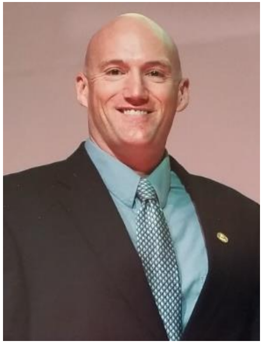
Chapter 11: Introduction to Interrupts

11.2 MSP430FR2355 Port Interrupts – Reading S1 Example



▶ Subscribe

www.youtube.com/c/DigitalLogicProgramming_LaMeres



Brock J. LaMeres, Ph.D.