Interrupts

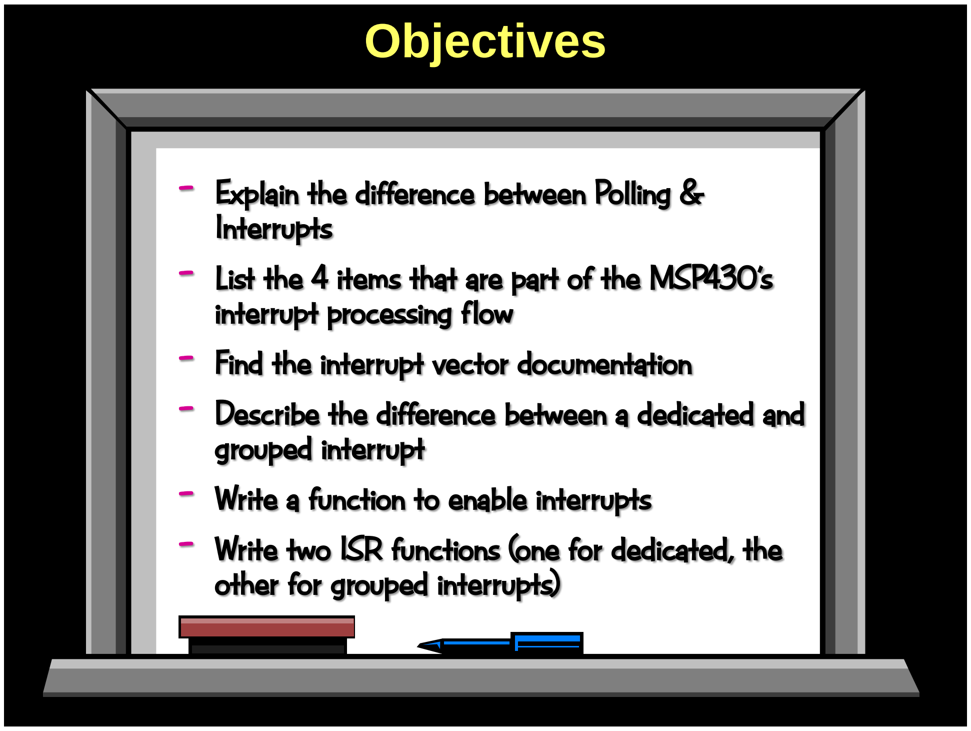
Introduction

What is an embedded system without interrupts?

If you just needed to solve a math problem you would most likely sit down and use a desktop computer. Embedded systems, on the other hand, take inputs from real-world events and then act upon them. These real-world events usually translate into ‘interrupts’ – asynchronous signals provided to the microcontroller: timers, serial ports, pushbuttons … and so on.

This chapter discusses how interrupts work; how they are implemented on the MSP430 MCU, and what code we need to write in order to harness their functionality. The lab exercises provided are relatively simple (using a pushbutton to generate an interrupt), but the skills we learn here will apply to all the remaining chapters of this workshop.

Learning Objectives



Objectives

- Explain the difference between Polling & Interrupts
- List the 4 items that are part of the MSP430’s interrupt processing flow
- Find the interrupt vector documentation
- Describe the difference between a dedicated and grouped interrupt
- Write a function to enable interrupts
- Write two ISR functions (one for dedicated, the other for grouped interrupts)

Chapter Topics

Interrupts, The Big Picture

While many of you are already familiar with interrupts, they are so fundamental to embedded systems that we wanted to briefly describe what they are all about.

From Wikipedia:

> *A **hardware interrupt** is an electronic alerting signal sent to the processor from an external device, either a part of the [device, such as an internal peripheral] or an external peripheral.*

In other words, the interrupt is a signal which notifies the CPU that an event has occurred. If the interrupt is configured, the CPU will respond to it immediately – as described later in this chapter.

Polling vs Interrupts

In reality, though, there are two methods that events can be recognized by the processor. One is called “Polling”; the other is what we just defined, “Interrupts”.

We start with a non-engineering analogy for these two methods. If you’ve ever taken a long family vacation, you’ve probably dealt with the “Are we there yet” question. In fact, kids often ask it over-and-over again. Eventually … the answer will be, “Yes, we’re there”. The alternative method is when my spouse says, “Wake me up when we get there”.

Waiting for an Event: Family Vacation

Polling

Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?

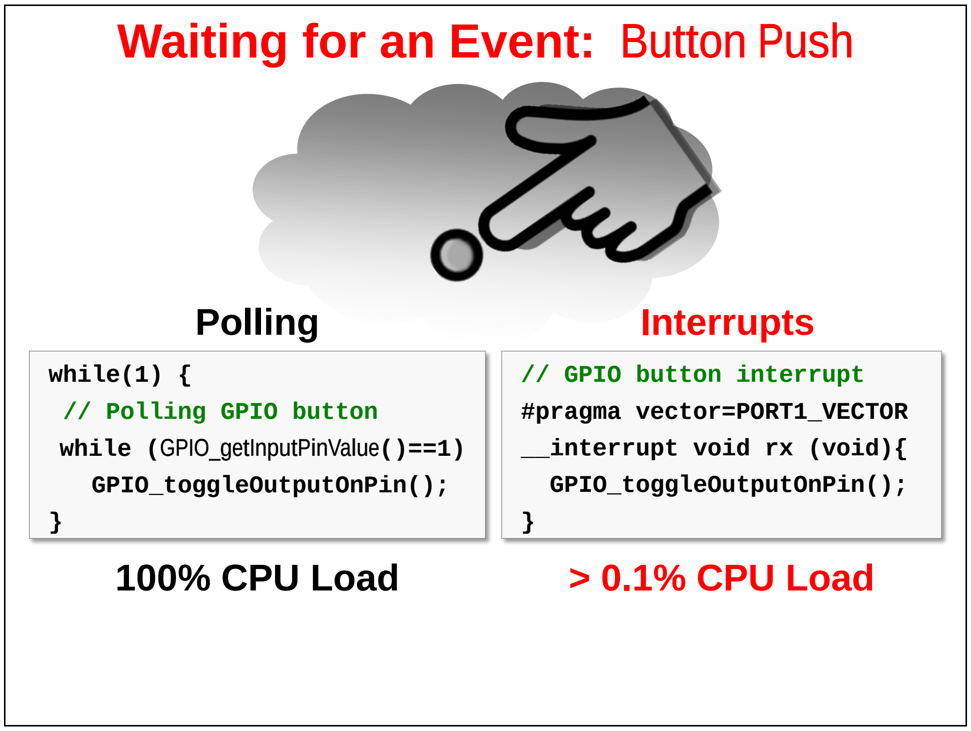
Interrupts

Wake me up when we get there...

Both methods signal that we have arrived at our destination. In most cases, though, the use of Interrupts tends to be <u>much more efficient</u>. For example, in the case of the MSP430, we often want to sleep the processor while waiting for an event. When the event happens and signals us with an interrupt, we can wake up, handle the event and then return to sleep waiting for the next event.

A real-world event might be our system responding to a push-button. Once again, the event could be handled using either **Polling** or **Interrupts**.

It is common to see “simple” example code utilize **Polling**. As you can see from the left-side example below, this can simply consist of a while{} loop that keeps repeating until a button-push is detected. The big downfall here, though, is that the processor is constantly running– asking the question, “Has the button been pushed, yet?”



Waiting for an Event: Button Push

Polling

```
while(1) {
 // Polling GPIO button
 while (GPIO_getInputPinValue()==1)
   GPIO_toggleOutputOnPin();
}
```

100% CPU Load

Interrupts

```
// GPIO button interrupt
#pragma vector=PORT1_VECTOR
__interrupt void rx (void){
  GPIO_toggleOutputOnPin();
}
```

> 0.1% CPU Load

The example on the right shows an **Interrupt** based solution. Since this code is not constantly running, as in the previous example’s while{} loop, the CPU load is very low.

Why do simple examples often ignore the use of interrupts? Because they are “simple”. Interrupts, on the other hand, require an extra three items to get them running. We show two of them in the right-hand example above.

- The #pragma sets up the interrupt vector. The MSP430 has a handy pragma which makes it easy to configure this item. (Note: we’ll cover the details of all these items later in this chapter.)
- The __interrupt keyword tells the compiler to code this function as an interrupt service routine (ISR). Interrupt functions require a context save and restore of any resources used within them.

While not shown above, we thought we’d mention the third item needed to get interrupts to work. For a CPU to respond to an interrupt, you also need to enable the interrupt. (Oh, and you may also have to setup the interrupt source; for example, we would have to configure our GPIO pin to be used as an interrupt input.)

So, in this chapter we leave the simple and inefficient examples behind and move to the real-world – where real-world embedded systems thrive on interrupts.

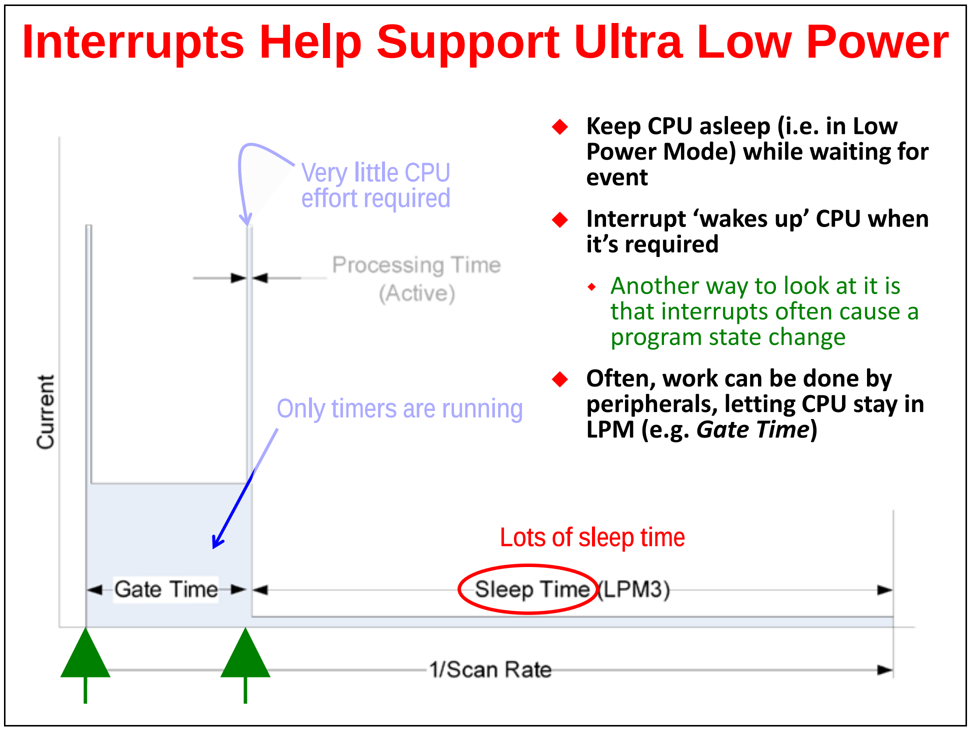
Processor States and Interrupts

In a previous chapter we covered many of the MSP430's processor states. To summarize, the MSP430 CPU can reside in: Reset, Active, or one of many Low-Power Modes (LPM). In many cases, interrupts cause the CPU to change states. For example, when sitting in Low Power Mode, an interrupt can "wake-up" the processor and return it to its active mode.

To help demonstrate this point, we stole the following slide from a discussion about Capacitive Touch. While most of this slide's content is not important for our current topic, we thought the *current vs time graph* was interesting. It tries to visually demonstrate the changing states of the device by charting power usage over time.

Notice the four states shown in this diagram:

- Notice how the current usage goes up at the beginning event – this is when the CPU is woken up so it can start a couple of peripherals (timers) needed to read the CapTouch button.
- The CPU can then go back to sleep while the sensor is being 'read' by the timers.
- When the read is complete (defined by something called "Gate" time, the CPU gets interrupted and wakes up again in order to calculate the CapTouch button's value from the sensor data.
- Finally the CPU (and CapTouch hardware) can go back to sleep and wait for another system wake-up event.



Interrupts Help Support Ultra Low Power

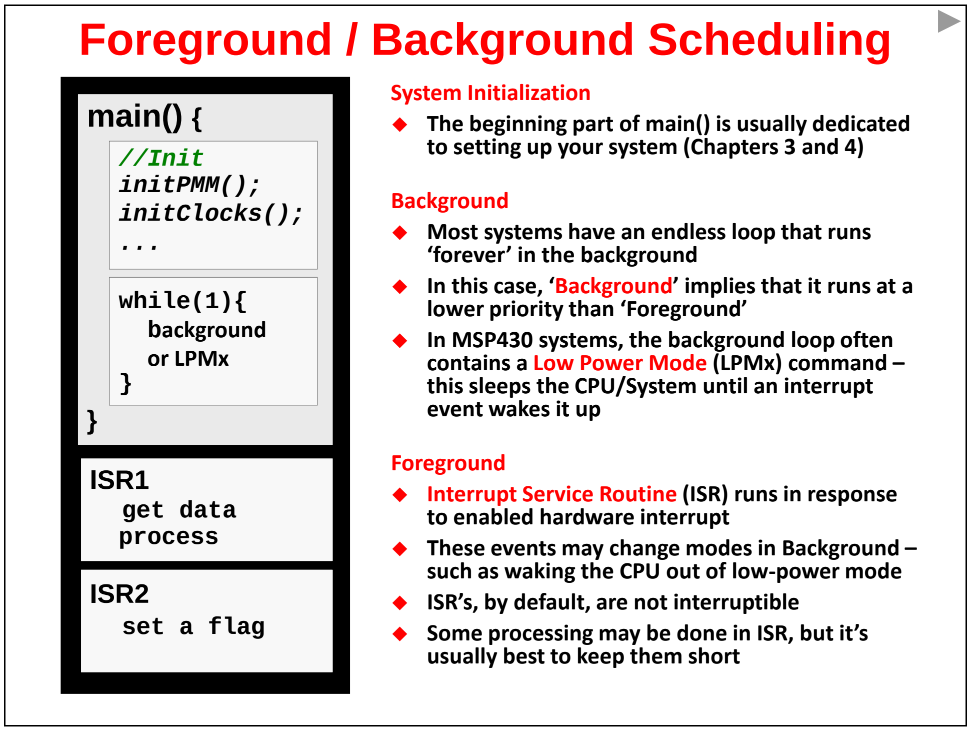
- **Keep CPU asleep (i.e. in Low Power Mode) while waiting for event**
- **Interrupt 'wakes up' CPU when it's required**
  - Another way to look at it is that interrupts often cause a program state change
- **Often, work can be done by peripherals, letting CPU stay in LPM (e.g. *Gate Time*)**

Threads: Foreground and Background

We conclude our Interrupts introduction by defining a few common terms used in interrupt-driven systems: **Thread**, **Foreground** and **Background**.

If you look at the “code” below, you will see that there are three individual – and independent – code segments below: main, ISR1, and ISR2.

We use the word *independent* because, if you were to examine the code in such a system, there are no calls between these three routines. Each one begins and ends execution without calling the others. It is common to call these separate segments of code: “Threads”.



Foreground / Background Scheduling

```
main() {
   //Init
   initPMM();
   initClocks();
   ...

   while(1){
      background
      or LPMx
   }
}

ISR1
   get data
   process

ISR2
   set a flag
```

System Initialization

- **The beginning part of main() is usually dedicated to setting up your system (Chapters 3 and 4)**

Background

- **Most systems have an endless loop that runs ‘forever’ in the background**
- **In this case, ‘Background’ implies that it runs at a lower priority than ‘Foreground’**
- **In MSP430 systems, the background loop often contains a Low Power Mode (LPMx) command – this sleeps the CPU/System until an interrupt event wakes it up**

Foreground

- **Interrupt Service Routine (ISR) runs in response to enabled hardware interrupt**
- **These events may change modes in Background – such as waking the CPU out of low-power mode**
- **ISR’s, by default, are not interruptible**
- **Some processing may be done in ISR, but it’s usually best to keep them short**

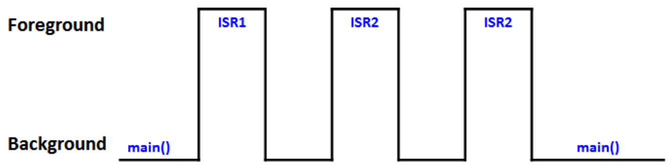
As we’ve seen in the workshop already, it is our main() thread that begins running once the processor has been started. The compiler’s initialization routine calls main() when its work is done. (In fact, this is why all C programs start with a main() function. Every compiler works the same way, in this regard.)

With the main() thread started, since it is coded with a while(1) loop, it will keep running forever. That is, unless a hardware interrupt occurs.

When an enabled interrupt is received by the CPU, it preempts the main() thread and runs the associated ISR routine – for example, ISR1. In other words, the CPU stops running main() temporarily and runs ISR1; when ISR1 completes execution, the CPU goes back to running main().

Here's where the terms **Foreground** and **Background** come into play. We call main() the **Background** thread since it is our "default" thread; that is, the program is designed such that we start running main() and go back to it whenever we're done with our other threads, such as ISR1.

Whenever an interrupt causes another thread to run, we call that a **Foreground** thread. The foreground threads preempt the Background thread, returning to the Background once completed.



The words "**Foreground**" and "**Background**" aren't terribly important. They just try to provide a bit of context that can be visualized in this common way.

It should be noted that it's important to keep your interrupt service routines short and quick. This, again, is common practice for embedded systems.

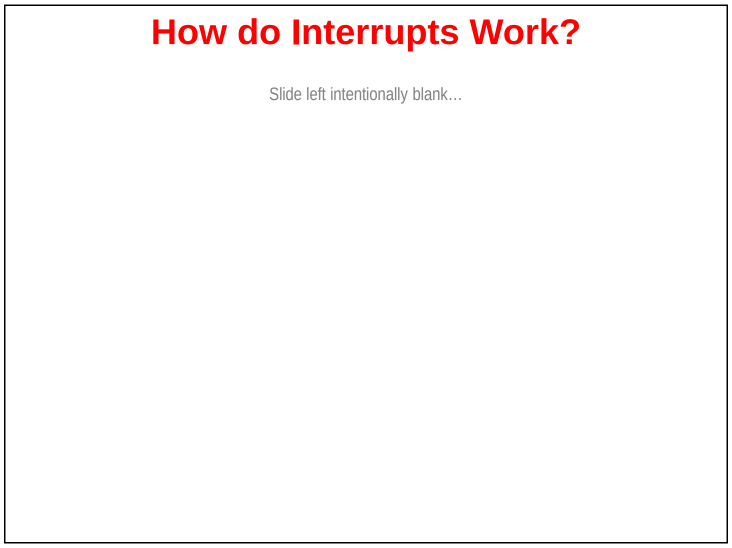
Note: We realize that our earlier definition of "Thread" was a little weak. What we said was true, but not complete. The author's favorite definition for "Thread" is as follows:

"A function or set of functions that operate independently of other code – running within their own context."

The key addtion here is that a thread runs within its own context. When switching from one thread to another, the context (register values and other resources) must be saved and restored.

How Interrupts Work

Now that we have a rough understanding of what interrupts are used for, let's discuss what mechanics are needed to make them work. Hint, there are 4 steps to getting interrupts to work…



How do Interrupts Work?

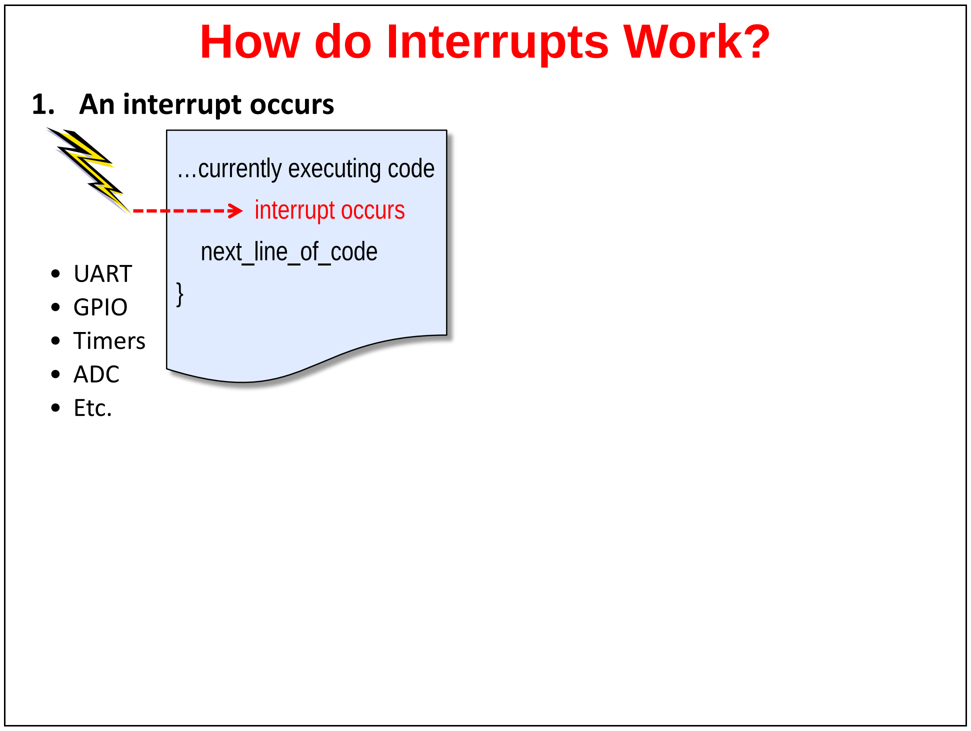
Slide left intentionally blank…

If you've been reading this chapter, you might notice that we've already covered these four items. Over the next few pages we enumerate these steps again, filling-in additional details.

Notes

1. Interrupt Must Occur

For the processor to respond to an interrupt, it must have occurred. There are many possible sources of interrupts. Later in this chapter we will delve into the MSP430 datasheet which lists all of the interrupt sources.



How do Interrupts Work?

1. **An interrupt occurs**

- UART
- GPIO
- Timers
- ADC
- Etc.

```
…currently executing code
   interrupt occurs
   next_line_of_code
}
```

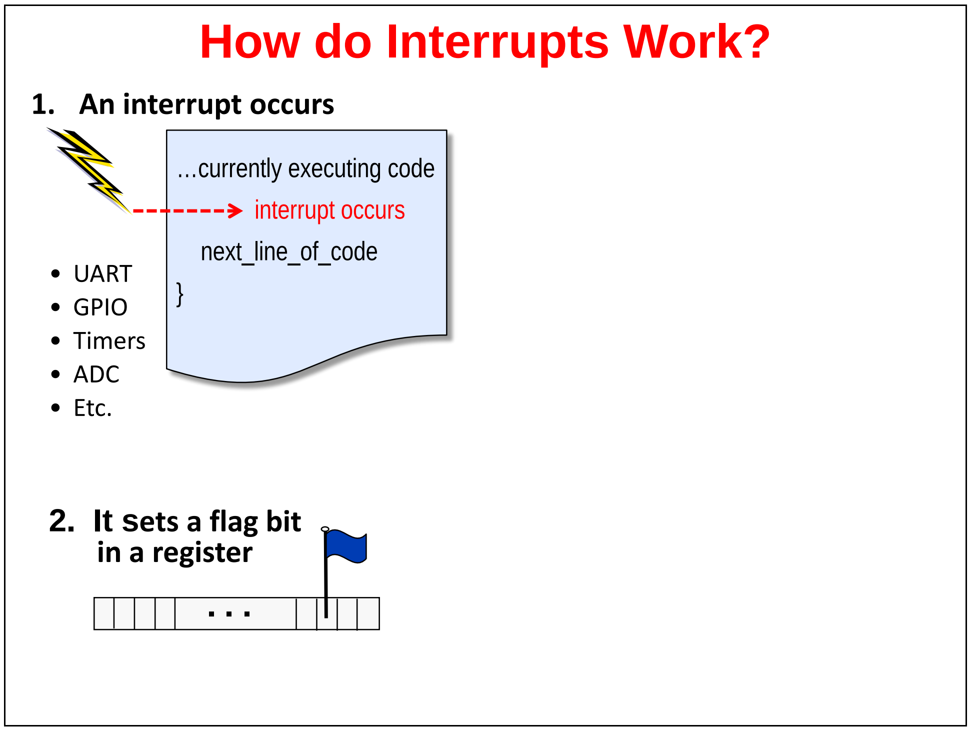
Suffice it to say that most peripherals can generate interrupts to provide status and information to the CPU. Most often, the interrupt indicates that data is available (e.g. serial port) and/or an event has occurred that needs processing (e.g. timer). In some cases, though, an interrupt may be used to indicate an error or exception in a peripheral that the CPU needs to handle.

Interrupts can also be generated from GPIO pins. This is how an external peripheral, or some other controller, can signal the MSP430 CPU. Most MSP430 devices allow the pins from the first two I/O ports (P1 and P2) to be individually configured for interrupt inputs. On the larger devices, there may be additional ports that can be configured for this, as well.

Finally, your software can often generate interrupts. The logic for some interrupts on the processor allow you to manually set a flag bit, thus ‘emulating’ a hardware interrupt. Not all interrupts provide this feature, but when available, it can be a handy way to test your interrupt service routine.

2. Interrupt is Flagged (and must be Enabled)

When an interrupt signal is received, an interrupt flag (IFG) bit is latched. You can think of this as the processor's "copy" of the signal. As some interrupt sources are only on for a short duration, it is important that the CPU registers the interrupt signal internally.



How do Interrupts Work?

1. **An interrupt occurs**

```
...currently executing code
  interrupt occurs
  next_line_of_code
}
```

- UART
- GPIO
- Timers
- ADC
- Etc.

2. **It sets a flag bit in a register**

MSP430 devices are designed with "distributed" interrupt management. That is, most IFG bits are found inside each peripheral's control registers; this is different from most processors which have a common, dedicated set of interrupt registers.

The distributed nature of the interrupts provides a number of benefits in terms of device flexibility and future feature expansion; further, it fits nicely with the low-power nature of the MSP430.

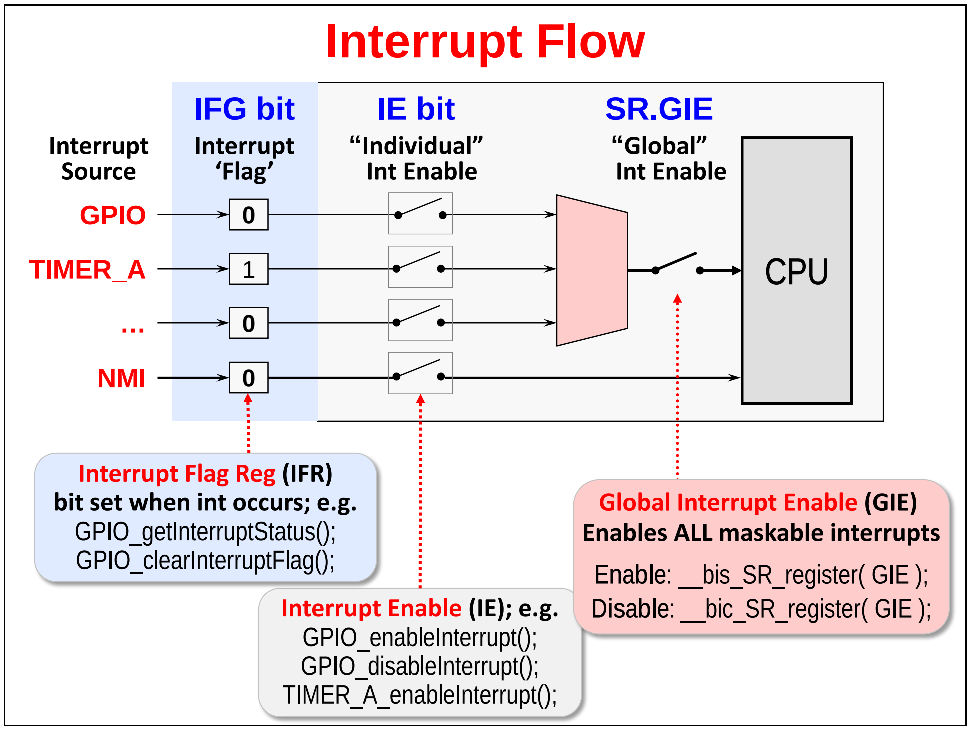
The only 'negative' of distributed interrupts might be that it's different — it's just that many of us older engineers are used to seeing all the interrupts grouped together. Bottom line, though, is that working with interrupts (enabling interrupts, clearing flags, responding to them) is the same whether the hardware is laid out centrally or in a distributed fashion.

Interrupt Flow

How does the interrupt signal reach the CPU?

We've just talked about the interrupt flag (IFG) bit – let's start there. As described on the previous page, when the interrupt source signal is received, the associated IFG bit is set. In fact, DriverLib contains functions to read the status of most IFG bits. (Handy in those few cases where you need to poll an interrupt source.)

When the IFG is set, the MSP430 *device* now sees that the signal has occurred, but the signal hasn't made its way to the *CPU*, yet. For that to happen, the interrupt must be **enabled**.


Interrupt Flow
IFG bit
IE bit
SR.GIE
Interrupt Source
Interrupt 'Flag'
"Individual" Int Enable
"Global" Int Enable
GPIO
TIMER_A
…
NMI
0
1
0
0
CPU
Interrupt Flag Reg (IFR)
bit set when int occurs; e.g.
GPIO_getInterruptStatus();
GPIO_clearInterruptFlag();
Interrupt Enable (IE); e.g.
GPIO_enableInterrupt();
GPIO_disableInterrupt();
TIMER_A_enableInterrupt();
Global Interrupt Enable (GIE)
Enables ALL maskable interrupts
Enable: __bis_SR_register(GIE);
Disable: __bic_SR_register(GIE);

Interrupt enable bits (IE) exist to protect the CPU … and thus, your program. Even with so many peripherals and interrupt sources, it's likely that your program will only care about a few of them. The enable bits provide your program with 'switches' that let you ignore all those sources you don't need.

By default, all interrupt bits are disabled (except the Watchdog Timer). It is your program's responsibility to enable those interrupt sources that are needed. To that end, once again, DriverLib provides a set of functions that make it easy for you to set the necessary IE bits.

Finally, there's a "master" switch that turns all interrupts off. This lets you turn off interrupts without having to modify all of the individual IE bits. The MSP430 calls this the global interrupt enable (GIE). It is found in the MSP430 Status Register (SR).

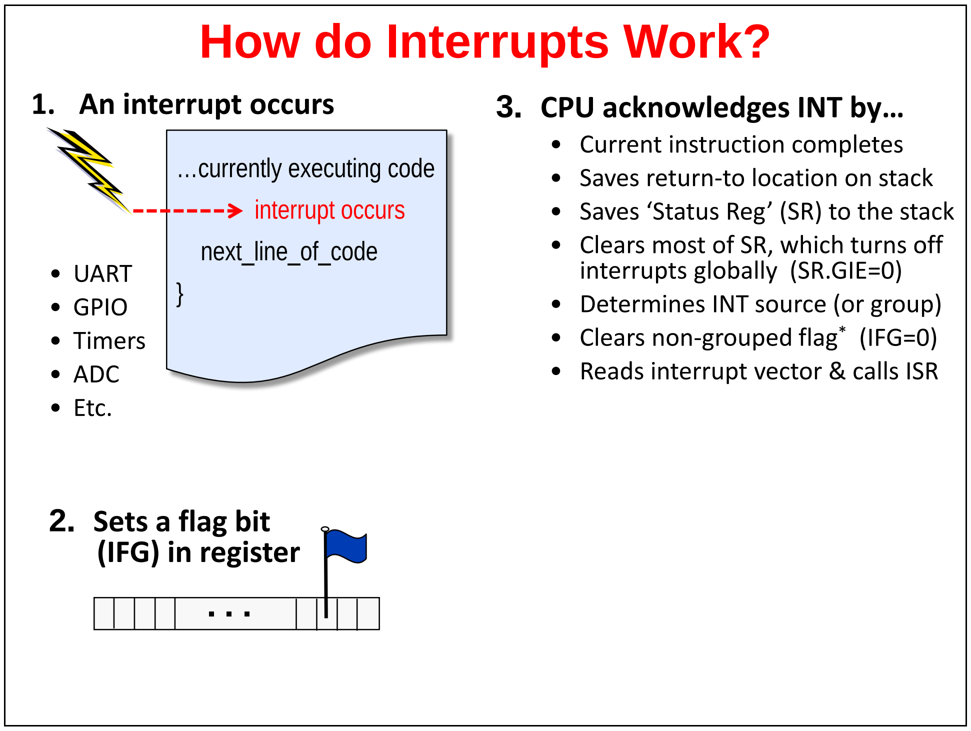
Why would you need a GIE bit? Sometimes your program may need to complete some code *atomically*; that is, your program may need to complete a section of code without the fear that an interrupt could preempt it. For example, if your program shares a global variable between two threads – say between main() and an ISR – it may be important to prevent interrupts while the main code reads and modifies that variable.

Note: There are a few non-maskable interrupts (NMI). These sources bypass the GIE bit. These interrupts are often considered critical events – i.e. 'fatal' events – that could be used to provide a *warm reset* of the CPU.

3. CPU's Hardware Response

At this point, let's assume you have an interrupt that has: occurred; been flagged; and since it was enabled, its signal has reached the CPU. What would the CPU do in response to the interrupt?

Earlier in the chapter we stated: "The interrupt preempts the current thread and starts running the interrupt service routine (ISR)." While this is true, there are actually a number of items performed by the hardware to make this happen – as shown below:



How do Interrupts Work?

1. An interrupt occurs
   - UART
   - GPIO
   - Timers
   - ADC
   - Etc.

2. Sets a flag bit (IFG) in register

3. CPU acknowledges INT by…
   - Current instruction completes
   - Saves return-to location on stack
   - Saves 'Status Reg' (SR) to the stack
   - Clears most of SR, which turns off interrupts globally (SR.GIE=0)
   - Determines INT source (or group)
   - Clears non-grouped flag* (IFG=0)
   - Reads interrupt vector & calls ISR

We hope the first 3 items are self-explanatory; the current instruction is completed while the Program Counter (PC) and Status Register (SR) are written to the system stack. (You might remember, the stack was setup for the MSP430 by the compiler's initialization routine. Please refer to the compiler user's guide for more information.)

After saving the context of SR, the interrupt hardware in the CPU clears most of the SR bits. Most significantly, it clears GIE. That means that by default, whenever you enter an ISR function, all maskable interrupts have been turned off. (We'll address the topic of 'nesting' interrupts in the next section.)

The final 3 items basically tell us that the processor figures out which interrupt occurred and calls the associated interrupt service routine; it also clears the interrupt flag bit (if it's a dedicated interrupt). The processor knows which ISR to run because each interrupt (IFG) is associated with an ISR function via a look-up table – called the Interrupt Vector Table.

Interrupt Vector Table – How is it different than other MCU's?

The MSP430 Vector Table is similar and dissimilar to other microcontrollers:

- The MSP430, like most microcontrollers, uses an Interrupt Vector Table. This is an area of memory that specifies a vector (i.e. ISR address) for each interrupt source.
- Some processors provide a unique ISR (and thus, vector) for every interrupt source. Other processors provide only 1 interrupt vector and make the user program figure which interrupt occurred. To maximize flexibility and minimize cost and power, the MSP430 falls in between these two extremes. There are some interrupts which have their own, dedicated interrupt vector – while other interrupts are logically grouped together.
- Where the MSP430 differs from many other processors is that it includes an Interrupt Vector (IV) register for each grouped interrupt; reading this register returns the highest-priority, enabled interrupt for that group of interrupt sources. As we'll see later in this chapter, all you need to do is read this register to quickly determine which specific interrupt to handle.

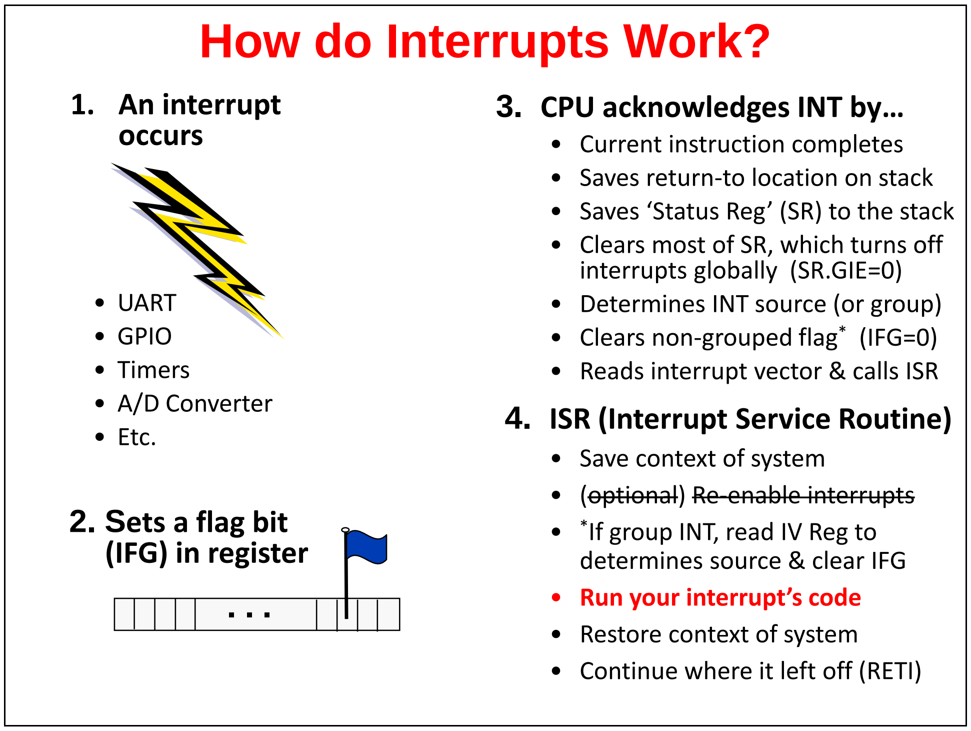
Note: We'll describe Interrupt Vector Table in more detail later in the chapter.

4. Your Software ISR

An *interrupt service routine* (ISR), also called an *interrupt handler*, is the code you write that will be run when a hardware interrupt occurs. Your ISR code must perform whatever task you want to execute in response to the interrupt, but without adversely affecting the threads (i.e. code) already running in the system.

Before we examine the details of the ISR; once again, how did we get to this point?

> Looking at the diagram below, we can see that (1) the interrupt must have occurred; (2) the processor flags the incoming interrupt; (3) if enabled, the interrupt flag signal is routed to the CPU where it saves the Status Register and Return-to address and then branches to the ISR's address found in the appropriate location in the vector table. (4) Finally, your ISR is executed.



How do Interrupts Work?

1. An interrupt occurs

- UART
- GPIO
- Timers
- A/D Converter
- Etc.

2. Sets a flag bit (IFG) in register

3. CPU acknowledges INT by…

- Current instruction completes
- Saves return-to location on stack
- Saves 'Status Reg' (SR) to the stack
- Clears most of SR, which turns off interrupts globally (SR.GIE=0)
- Determines INT source (or group)
- Clears non-grouped flag* (IFG=0)
- Reads interrupt vector & calls ISR

4. ISR (Interrupt Service Routine)

- Save context of system
- (~~optional~~) ~~Re-enable interrupts~~
- *If group INT, read IV Reg to determines source & clear IFG
- **Run your interrupt's code**
- Restore context of system
- Continue where it left off (RETI)

The crux of the ISR is doing what needs to be done in response to the interrupt; the 4th bullet (listed in red) reads:

- **Run your interrupt's code**

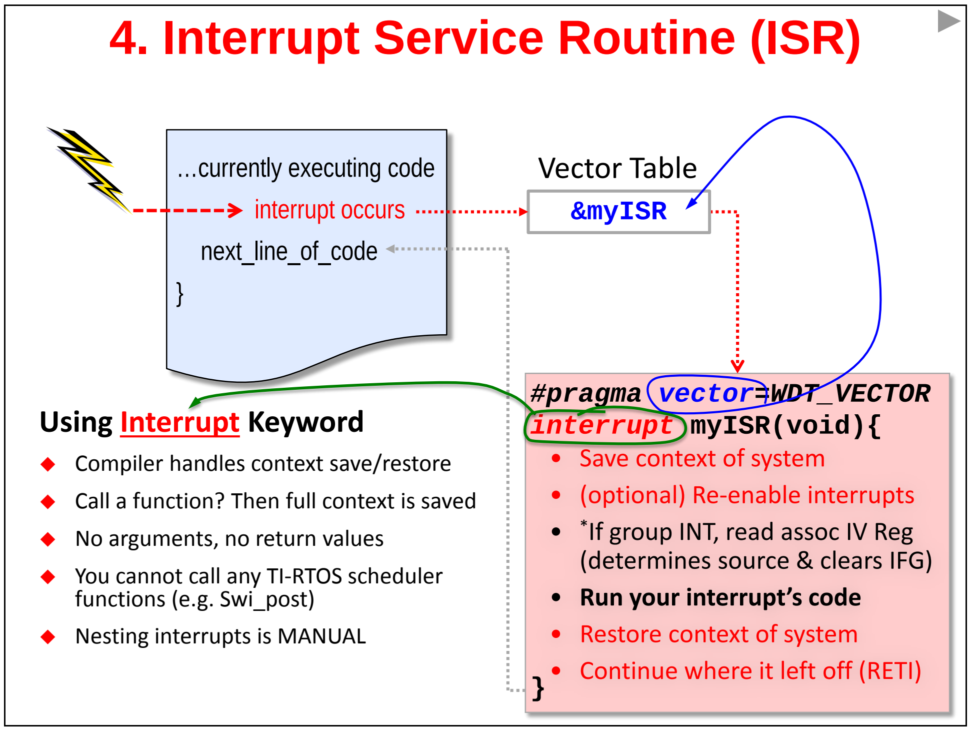
This is meant to describe the code you write to handle the interrupt. For example, if it's a UART interrupt, your code might read an incoming byte of data and write it to memory.

We'll discuss the 2nd (optional) bullet on the next page.

The 3rd bullet indicates that if this is a "grouped" interrupt, you have to add code to figure out which interrupt, in the group, needs to be handled. This is usually done by reading the group's IV register. (This bullet was in red because it is code you need to write.)

The other bullets listed under "4. ISR" are related to saving and restoring the context of the system. This is required so that the condition mentioned earlier can be met: *"without adversely affecting the code threads already running in the system."*

We show the interrupt flow in a slightly different fashion in the following diagram. As you can see, when an enabled interrupt occurs, the processor will look up the ISR's branch-to address from a specific address in memory (called the *interrupt vector*). For the MSP430, this address is defined using the *vector* pragma.



4. Interrupt Service Routine (ISR)

```
#pragma vector=WDT_VECTOR
interrupt myISR(void){
  • Save context of system
  • (optional) Re-enable interrupts
  • *If group INT, read assoc IV Reg
    (determines source & clears IFG)
  • Run your interrupt's code
  • Restore context of system
  • Continue where it left off (RETI)
}
```

Using Interrupt Keyword

- Compiler handles context save/restore
- Call a function? Then full context is saved
- No arguments, no return values
- You cannot call any TI-RTOS scheduler functions (e.g. Swi_post)
- Nesting interrupts is MANUAL

The context of the system – for example, the CPU registers used by the ISR – must be saved before running your code and restored afterwards. Thankfully, the compiler handles this for you when the function is declared as an *interrupt*. (As part of the "context restore", the compiler will return to running the previous thread of code by using the RETI instruction).

Please note the bullets under "Using the Interrupt Keyword" from the preceding diagram.

Using this keyword, the compiler handles all of the context save/restore for you and knows how to return to your previous code – even restoring the original value for the Status Register (SR).

Hint: If you call a function within your ISR, the compiler will have to save/restore every CPU register, not just the ones that it uses to implement your C code. This is because it doesn't know what resources the function call may end up using.

Since the interrupt occurs asynchronously to the background thread, you cannot pass arguments to and receive return values from the ISR. You must communicate between threads using global variables (or other appropriate data objects).

TI's real-time operating system (TI-RTOS) provides a rich set of scheduling functions that are often used within interrupt service routines. Be aware, though, that some of these functions can only be used with RTOS "managed" interrupts. In fact, it's actually easier to let TI-RTOS manage your interrupts; it automatically handles plugging the interrupt vector as well as context save/restore. (All you have to do is write a standard C function.) But, the details of TI-RTOS are outside the scope of this workshop. While we provide a brief discussion of TI-RTOS at the end of this chapter, please refer to the *Introduction to TI-RTOS Kernel* workshop for more details.

Nesting Interrupts (not recommended)

Finally, while the MSP430 allows nesting of interrupts, it is not recommended.

- *Nesting* interrupts means one interrupt can interrupt another interrupt.
- You must manually configure nesting. That is, before running your interrupt handling code you must:
  - Disable any interrupts that you do not want to occur during your ISR. In other words, you must first save, then disable, any IE bit that correlates to an interrupt that you do not want to interrupt your ISR.
  - Then, turn on interrupts globally by setting GIE = 1.
  - At this point you can run your code that responds to the original interrupt. It may end up being interrupted by any source that you left enabled.
  - When you've completed your original interrupt code, you need to disable interrupts before returning from the function. That is, set GIE = 0. (This is the state GIE was in when entering your ISR code.
  - You can now restore the IE bits that you saved before enabling GIE.
  - At this point, you can return from the ISR and let the compiler's code handle the remaining context save and return branch back to the original thread.
- In general, it's considered better programming practice to keep interrupt service routines very short – i.e. lean-and-mean. Taking this further, with low-power and efficiency in mind, the MSP430 team recommends you follow the no-nesting general principle.

Hint: We encourage you to avoid nesting, if at all possible. Not only is it difficult, and error prone, it often complicates your programs ability to reach low-power modes.

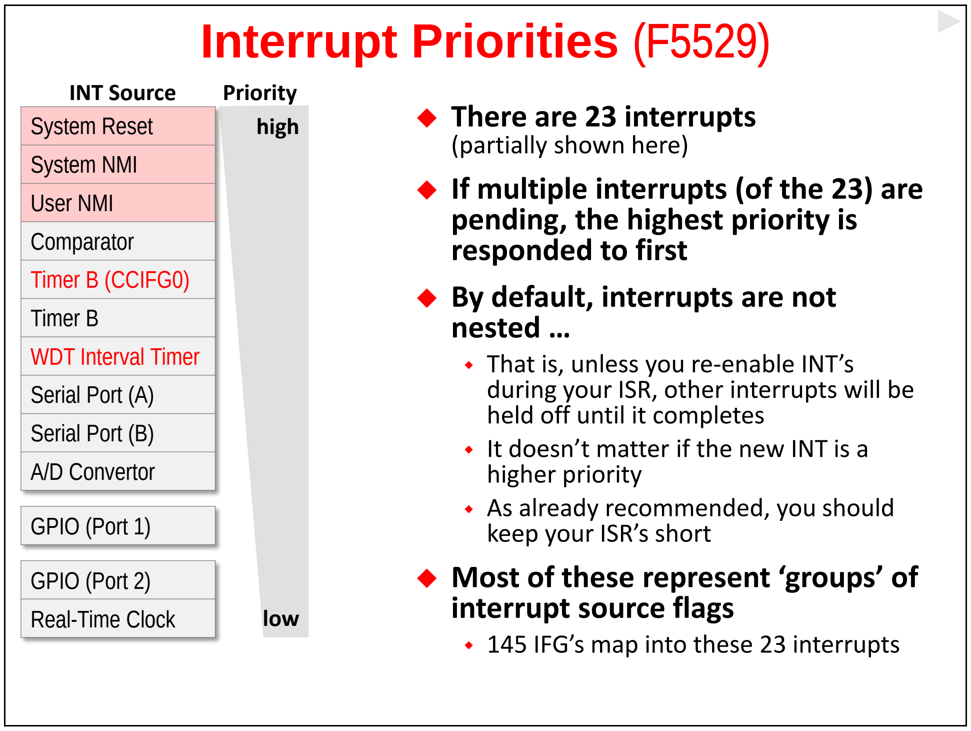
Interrupts: Priorities & Vectors

Interrupts and Priorities

Each MSP430 device datasheet defines the *pending* priority for each of its hardware interrupts. In the case of the MSP430F5529, there are 23 interrupts shown listed below in decreasing priority.

In the previous paragraph we used the phrase “pending priority” deliberately. As you might remember from the last topic in this chapter, interrupts on the MSP430 do not nest within each other by default. This is because the global interrupt (GIE) bit is disabled when the CPU acknowledges and processes an interrupt. Therefore, if an interrupt occurs while an ISR is being executed, it will have to wait for the current ISR to finish before it can be handled … even if the new interrupt is of higher priority.

On the other hand, if two interrupts occur at the same time – that is, if there are two interrupts currently pending – then the highest priority interrupt is acknowledged and handled first.



Interrupt Priorities (F5529)

INT Source	Priority
System Reset	high
System NMI	
User NMI	
Comparator	
Timer B (CCIFG0)	
Timer B	
WDT Interval Timer	
Serial Port (A)	
Serial Port (B)	
A/D Convertor	
GPIO (Port 1)	
GPIO (Port 2)	
Real-Time Clock	low

- **There are 23 interrupts**
  (partially shown here)
- **If multiple interrupts (of the 23) are pending, the highest priority is responded to first**
- **By default, interrupts are not nested …**
  - That is, unless you re-enable INT’s during your ISR, other interrupts will be held off until it completes
  - It doesn’t matter if the new INT is a higher priority
  - As already recommended, you should keep your ISR’s short
- **Most of these represent ‘groups’ of interrupt source flags**
  - 145 IFG’s map into these 23 interrupts

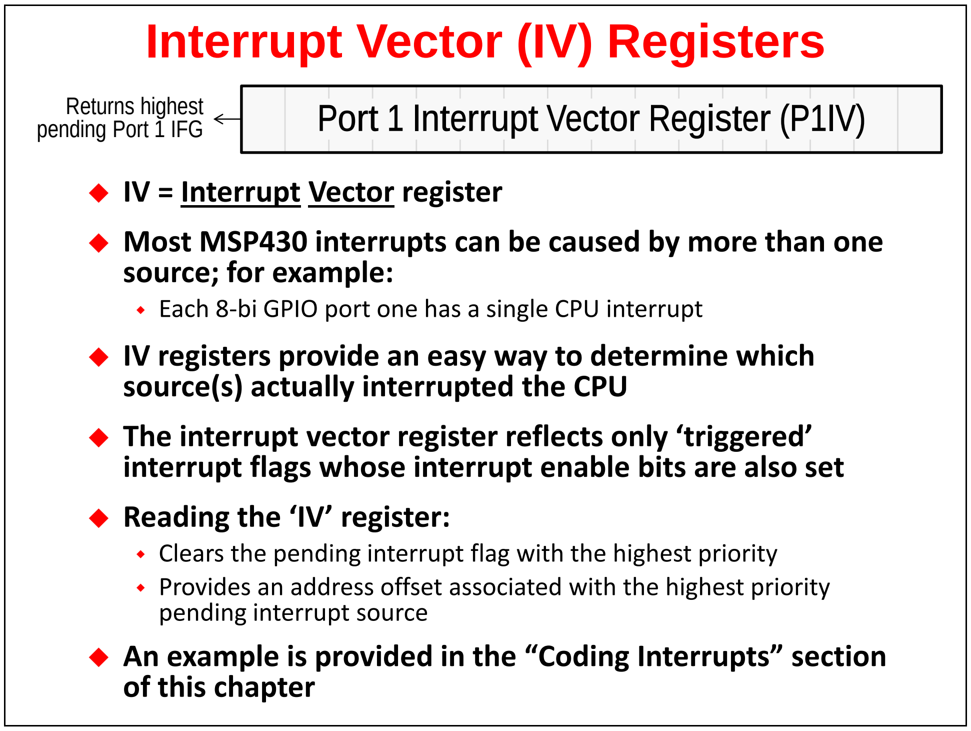
Most of the 23 interrupts on the ‘F5529 represent ‘groups’ of interrupts. There are actually 145 interrupt sources – each with their own interrupt flag (IFG) – that map into these 23 interrupts.

For example, the “Timer B (CCIFG0)” interrupt represents a single interrupt signal. When the CPU acknowledges it, it will clear its single IFG flag.

On the other hand, the next interrupt in line, the “Timer B” interrupt, represents all the rest of the interrupts that can be initiated by Timer0_B. When any one of the interrupts in this group occurs, the ISR will need to determine which specific interrupt source occurred and clear its flag (along with executing whatever code you want to associate with it).

Interrupt Vector (IV) Registers

As has been mentioned a couple of times in this chapter, to make responding to *grouped* interrupts easier to handle, the MSP430 team created the concept of *Interrupt Vector (IV) Registers*. Reading an IV register will return the *highest-priority, pending interrupt* in that group; it will also clear that interrupts associated flag (IFG) bit.



Interrupt Vector (IV) Registers

Returns highest pending Port 1 IFG ← Port 1 Interrupt Vector Register (P1IV)

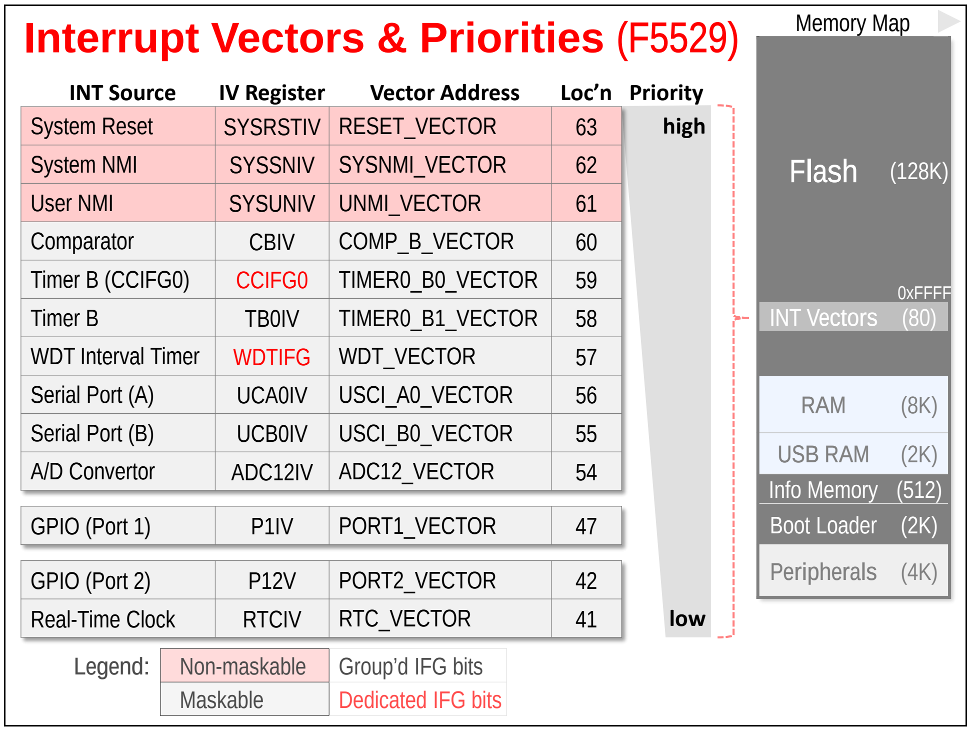
- **IV = <u>Interrupt</u> <u>Vector</u> register**
- **Most MSP430 interrupts can be caused by more than one source; for example:**
  - Each 8-bi GPIO port one has a single CPU interrupt
- **IV registers provide an easy way to determine which source(s) actually interrupted the CPU**
- **The interrupt vector register reflects only 'triggered' interrupt flags whose interrupt enable bits are also set**
- **Reading the 'IV' register:**
  - Clears the pending interrupt flag with the highest priority
  - Provides an address offset associated with the highest priority pending interrupt source
- **An example is provided in the "Coding Interrupts" section of this chapter**

For grouped interrupts, most users read the IV register at the beginning of the ISR and use the return value to pick the appropriate code to run. This is usually implemented with a Switch/Case statement. (We will explore an example of this code later in the chapter.)

Interrupt Vector Table

We can expand the previous interrupt source & priority listing to include a few more items. First of all, we added a column that provides the IV register associated with each interrupt. (Note, the two names shown in red text represent the IFG bits for dedicated/individual interrupts.)

Additionally, the first 3 rows (highlighted with red background fill) indicate that these interrupt groups are non-maskable; therefore, they bypass the GIE bit.



Interrupt Vectors & Priorities (F5529)

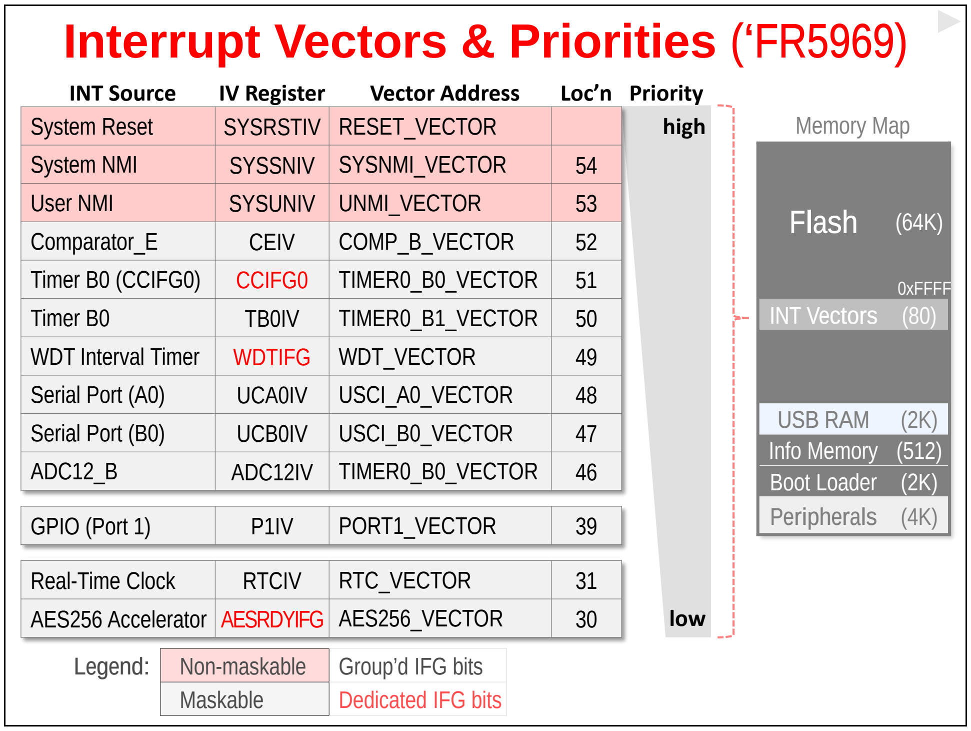
INT Source	IV Register	Vector Address	Loc'n	Priority
System Reset	SYSRSTIV	RESET_VECTOR	63	high
System NMI	SYSSNIV	SYSNMI_VECTOR	62	
User NMI	SYSUNIV	UNMI_VECTOR	61	
Comparator	CBIV	COMP_B_VECTOR	60	
Timer B (CCIFG0)	CCIFG0	TIMER0_B0_VECTOR	59	
Timer B	TB0IV	TIMER0_B1_VECTOR	58	
WDT Interval Timer	WDTIFG	WDT_VECTOR	57	
Serial Port (A)	UCA0IV	USCI_A0_VECTOR	56	
Serial Port (B)	UCB0IV	USCI_B0_VECTOR	55	
A/D Convertor	ADC12IV	ADC12_VECTOR	54	
GPIO (Port 1)	P1IV	PORT1_VECTOR	47	
GPIO (Port 2)	P12V	PORT2_VECTOR	42	
Real-Time Clock	RTCIV	RTC_VECTOR	41	low

Legend:	
Non-maskable	Group'd IFG bits
Maskable	Dedicated IFG bits

The final column in the above diagram hints at the location of each interrupts address vector in the memory map. For example, when using the WDT as an interval timer, you would put the address of your appropriate ISR into location "57". As we saw in a previous topic, this can easily be done using the *vector* pragma.

The MSP430 devices reserve the range 0xFFFF to 0xFF80 for the interrupt vectors. This means that for the 'F5529, the address for the System Reset interrupt service routine will sit at addresses 0xFFFE – 0xFFFF. (A 16-bit address requires two 8-bit memory locations.) The remaining interrupt vectors step down in memory from this point. The map to the right of the table shows where the interrupt vectors appear within the full MSP430 memory map.

Here's a quick look at the same table showing the MSP430FR5969 interrupt vectors and priorities. The list is very similar to the 'F5529; the main differences stem from the fact that the two devices have a slightly different mix of peripherals.

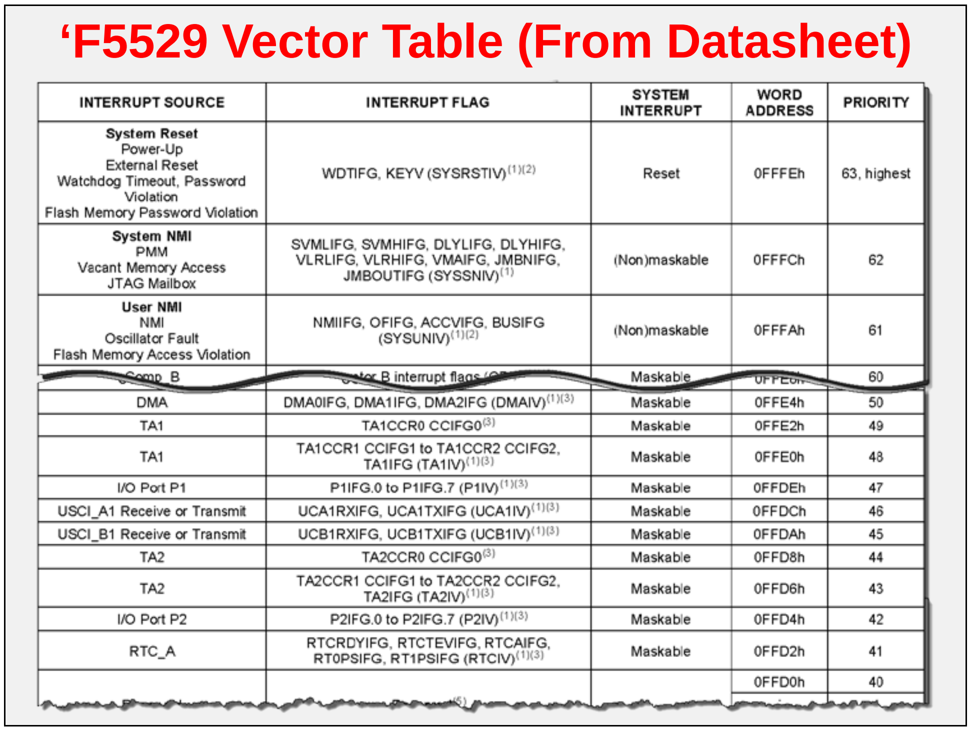


Interrupt Vectors & Priorities ('FR5969)

INT Source	IV Register	Vector Address	Loc'n	Priority
System Reset	SYSRSTIV	RESET_VECTOR		high
System NMI	SYSSNIV	SYSNMI_VECTOR	54	
User NMI	SYSUNIV	UNMI_VECTOR	53	
Comparator_E	CEIV	COMP_B_VECTOR	52	
Timer B0 (CCIFG0)	CCIFG0	TIMER0_B0_VECTOR	51	
Timer B0	TB0IV	TIMER0_B1_VECTOR	50	
WDT Interval Timer	WDTIFG	WDT_VECTOR	49	
Serial Port (A0)	UCA0IV	USCI_A0_VECTOR	48	
Serial Port (B0)	UCB0IV	USCI_B0_VECTOR	47	
ADC12_B	ADC12IV	TIMER0_B0_VECTOR	46	
GPIO (Port 1)	P1IV	PORT1_VECTOR	39	
Real-Time Clock	RTCIV	RTC_VECTOR	31	
AES256 Accelerator	AESRDYIFG	AES256_VECTOR	30	low

Legend:		
	Non-maskable	Group'd IFG bits
	Maskable	Dedicated IFG bits

The preceding interrupt tables were re-drawn to make them easier to view when projected during a workshop. The following slide was captured from ‘F5529 datasheet. This is what you will see if you examine the MSP430 documentation.



‘F5529 Vector Table (From Datasheet)

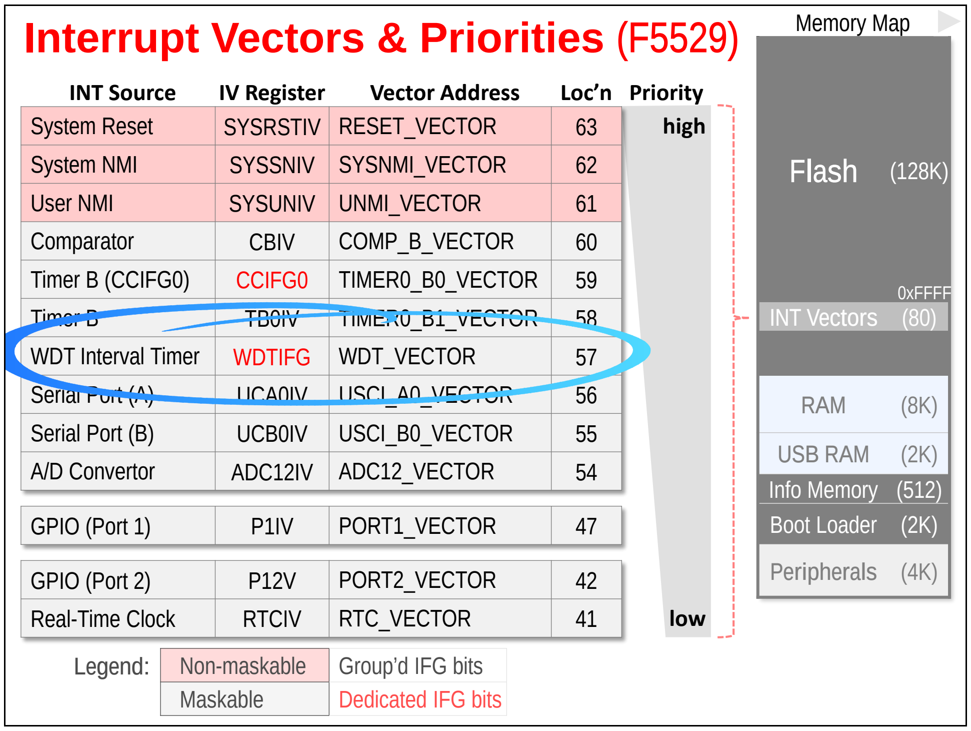
INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
System Reset Power-Up External Reset Watchdog Timeout, Password Violation Flash Memory Password Violation	WDTIFG, KEYV (SYSRSTIV)(1)(2)	Reset	0FFFEh	63, highest
System NMI PMM Vacant Memory Access JTAG Mailbox	SVMLIFG, SVMHIFG, DLYLIFG, DLYHIFG, VLRLIFG, VLRHIFG, VMAIFG, JMBNIFG, JMBOUTIFG (SYSSNIV)(1)	(Non)maskable	0FFFCh	62
User NMI NMI Oscillator Fault Flash Memory Access Violation	NMIIFG, OFIFG, ACCVIFG, BUSIFG (SYSUNIV)(1)(2)	(Non)maskable	0FFFAh	61
Comp_B	[illegible] B interrupt flags [illegible]	Maskable	[illegible]	60
DMA	DMA0IFG, DMA1IFG, DMA2IFG (DMAIV)(1)(3)	Maskable	0FFE4h	50
TA1	TA1CCR0 CCIFG0(3)	Maskable	0FFE2h	49
TA1	TA1CCR1 CCIFG1 to TA1CCR2 CCIFG2, TA1IFG (TA1IV)(1)(3)	Maskable	0FFE0h	48
I/O Port P1	P1IFG.0 to P1IFG.7 (P1IV)(1)(3)	Maskable	0FFDEh	47
USCI_A1 Receive or Transmit	UCA1RXIFG, UCA1TXIFG (UCA1IV)(1)(3)	Maskable	0FFDCh	46
USCI_B1 Receive or Transmit	UCB1RXIFG, UCB1TXIFG (UCB1IV)(1)(3)	Maskable	0FFDAh	45
TA2	TA2CCR0 CCIFG0(3)	Maskable	0FFD8h	44
TA2	TA2CCR1 CCIFG1 to TA2CCR2 CCIFG2, TA2IFG (TA2IV)(1)(3)	Maskable	0FFD6h	43
I/O Port P2	P2IFG.0 to P2IFG.7 (P2IV)(1)(3)	Maskable	0FFD4h	42
RTC_A	RTCRDYIFG, RTCTEVIFG, RTCAIFG, RT0PSIFG, RT1PSIFG (RTCIV)(1)(3)	Maskable	0FFD2h	41
			0FFD0h	40

Each device’s datasheet provides a similar *vector table* listing. If you are using the ‘G2553 or ‘FR5969 devices, for example, you will find a similar table in each of their respective datasheets.

Coding Interrupts

As previously discussed, the code within your interrupt service routine will vary slightly based on whether it handles a dedicated, single interrupt or if it handles a grouped interrupt. We will cover both cases; starting with the easier, dedicated case.

Dedicated ISR (Interrupt Service Routine)



Interrupt Vectors & Priorities (F5529)

INT Source	IV Register	Vector Address	Loc'n	Priority
System Reset	SYSRSTIV	RESET_VECTOR	63	high
System NMI	SYSSNIV	SYSNMI_VECTOR	62	
User NMI	SYSUNIV	UNMI_VECTOR	61	
Comparator	CBIV	COMP_B_VECTOR	60	
Timer B (CCIFG0)	CCIFG0	TIMER0_B0_VECTOR	59	
Timer B	TB0IV	TIMER0_B1_VECTOR	58	
WDT Interval Timer	WDTIFG	WDT_VECTOR	57	
Serial Port (A)	UCA0IV	USCI_A0_VECTOR	56	
Serial Port (B)	UCB0IV	USCI_B0_VECTOR	55	
A/D Convertor	ADC12IV	ADC12_VECTOR	54	
GPIO (Port 1)	P1IV	PORT1_VECTOR	47	
GPIO (Port 2)	P12V	PORT2_VECTOR	42	
Real-Time Clock	RTCIV	RTC_VECTOR	41	low

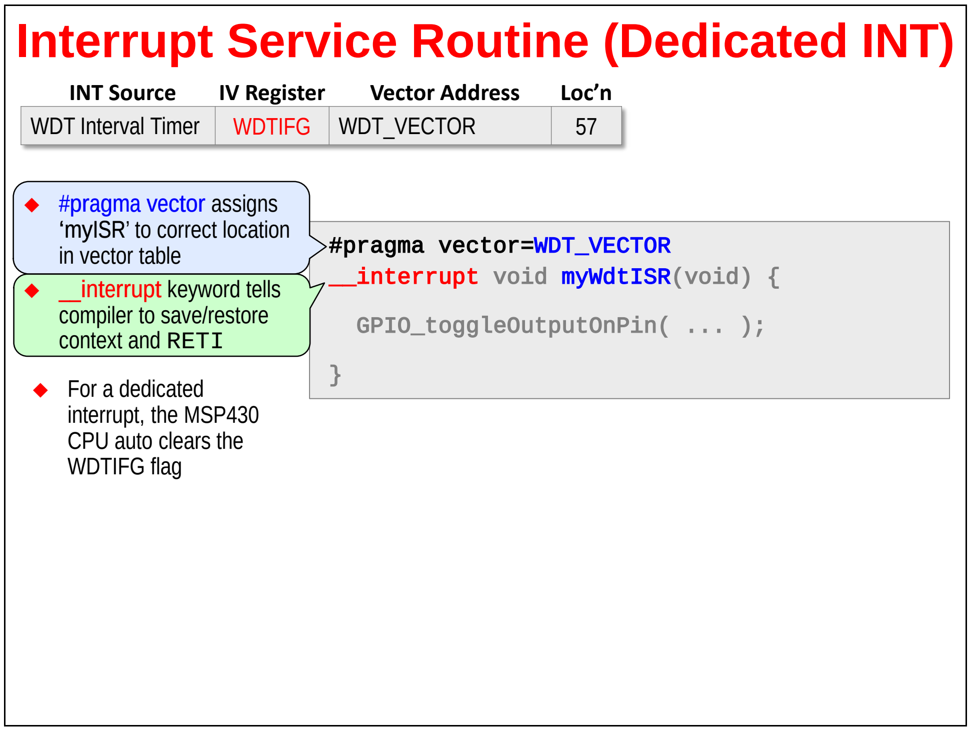
Legend:		
	Non-maskable	Group'd IFG bits
	Maskable	Dedicated IFG bits

The watchdog interrupt flag vector (WDTIFG) is a dedicated interrupt; therefore, your ISR code only needs to respond to the single interrupt condition. Additionally, because it is a dedicated interrupt, the CPU hardware automatically clears the WDTIFG bit when responding to the interrupt and branching to your ISR.

When writing an ISR for dedicated interrupts, you code must address three items:

1. Put the ISR address into the vector table (using the vector #pragma)
2. Save/Restore the CPU context (using the __interrupt keyword)
3. Write your interrupt handler code (in other words, "Do what needs doing")

We will use the following code example to demonstrate these three items.



Interrupt Service Routine (Dedicated INT)

INT Source	IV Register	Vector Address	Loc'n
WDT Interval Timer	WDTIFG	WDT_VECTOR	57

- #pragma vector assigns 'myISR' to correct location in vector table
- __interrupt keyword tells compiler to save/restore context and `RETI`
- For a dedicated interrupt, the MSP430 CPU auto clears the WDTIFG flag

```
#pragma vector=WDT_VECTOR
__interrupt void myWdtISR(void) {

  GPIO_toggleOutputOnPin( ... );

}
```

Plug the Vector Table (#pragma vector)

In our example, the following line of code:

```
#pragma vector=WDT_VECTOR
```

tells the compiler to associate the function (on the following line) with the WDT_VECTOR. Looking in the MSP430F5529 device-specific linker command file, you should find this vector name ("WDT_VECTOR") associated with vector #57. This matches with the datasheet documentation we looked at earlier in the chapter.

Save/Restore CPU context (__interrupt keyword)

The __*interrupt* keyword tells the compiler that this function is an interrupt service routine and thus it needs to save (and then restore) the context of the processor (i.e. CPU registers) before (and after) executing the function's code.

Don't forget, functions using the __*interrupt* keyword cannot accept arguments or return values.

Hint: Empirical analysis shows that "__interrupt" and "interrupt" are both accepted by the compiler.

Your Interrupt Code

In this example, the output of a GPIO pin is toggled every time the watchdog timer interrupt event occurs. Not all ISR's will be this short, but we hope this gives you a good starting example to work from.

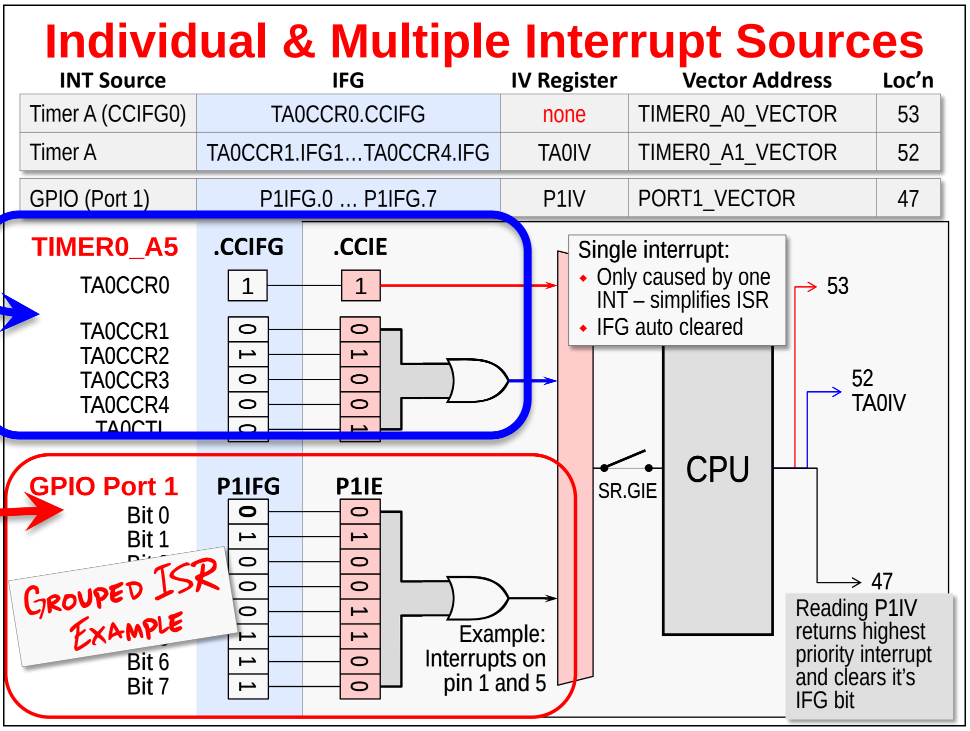
Grouped ISR (Interrupt Service Routine)

Logical Diagram for Grouped Interrupts

Before examining the code for a grouped ISR, let's first examine the grouped interrupt using a logical diagram.

As we briefly mentioned earlier in the chapter (and will discuss in full detail in a later chapter), the Timer_A and Timer_B peripherals are provided with two interrupts. For example, when looking at Timer0_A5, there is a dedicated interrupt for TA0CCR0 (which stands for Timer0_A Capture/Compare Register 0). Notice below how this is routed directly to the GIE input mux.

The remaining five Timer0_A5 interrupts are logically AND'd together; this combination provides a second interrupt signal from Timer0_A5 to the GIE input mux.



Individual & Multiple Interrupt Sources

INT Source	IFG	IV Register	Vector Address	Loc'n
Timer A (CCIFG0)	TA0CCR0.CCIFG	none	TIMER0_A0_VECTOR	53
Timer A	TA0CCR1.IFG1…TA0CCR4.IFG	TA0IV	TIMER0_A1_VECTOR	52
GPIO (Port 1)	P1IFG.0 … P1IFG.7	P1IV	PORT1_VECTOR	47

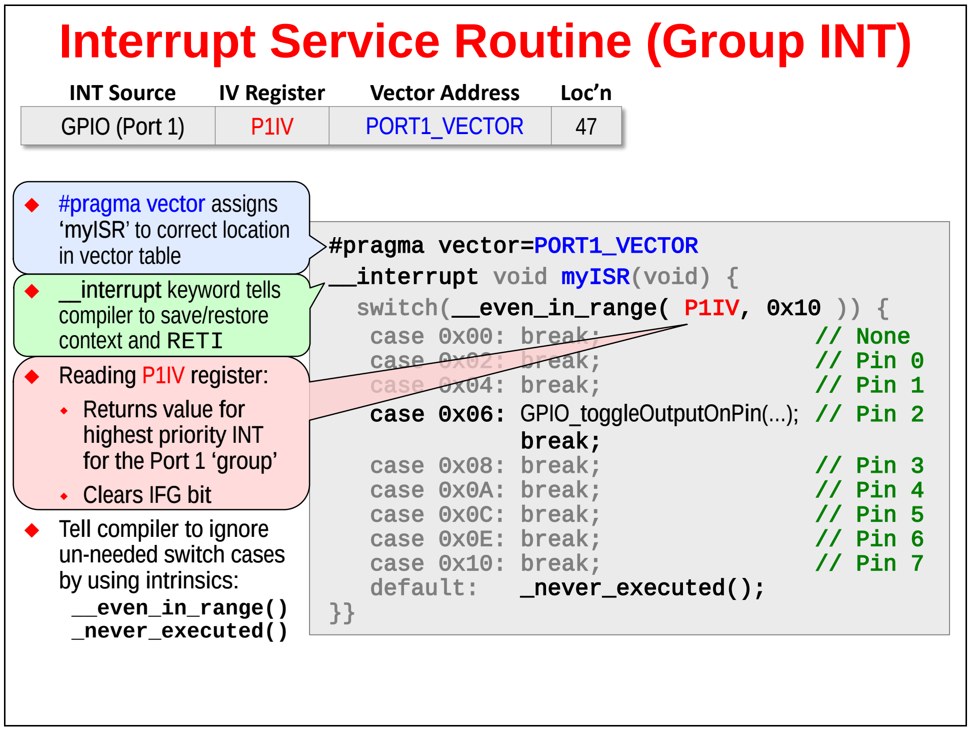
This diagram also shows that all of the input pins for GPIO port 1 (P1) share a single, grouped interrupt. This means your GPIO ISR must always verify which pin actually caused an interrupt whenever the ISR is executed.

The interrupt logic within the CPU recognizes each of these interrupt sources, therefore:

- If the first interrupt (TA0CCR0) occurs, it will cause the code at vector address 53 (TIMER_A0_VECTOR) to be executed.
- Similarly, the remaining Timer0 interrupts are associated with vector 52.
- Finally, the GPIO port (P1) was assigned (by the chip designer) to vector 47.

ISR Example for Grouped Interrupts

The code for a grouped ISR begins similar to any MPS430 interrupt service routine; you should use the #pragma vector and __interrupt keyword syntax.



Interrupt Service Routine (Group INT)

INT Source	IV Register	Vector Address	Loc'n
GPIO (Port 1)	P1IV	PORT1_VECTOR	47

- #pragma vector assigns 'myISR' to correct location in vector table
- __interrupt keyword tells compiler to save/restore context and RETI
- Reading P1IV register:
  - Returns value for highest priority INT for the Port 1 'group'
  - Clears IFG bit
- Tell compiler to ignore un-needed switch cases by using intrinsics:
  `__even_in_range()`
  `_never_executed()`

```
#pragma vector=PORT1_VECTOR
__interrupt void myISR(void) {
  switch(__even_in_range( P1IV, 0x10 )) {
   case 0x00: break;                      // None
   case 0x02: break;                      // Pin 0
   case 0x04: break;                      // Pin 1
   case 0x06: GPIO_toggleOutputOnPin(...); // Pin 2
              break;
   case 0x08: break;                      // Pin 3
   case 0x0A: break;                      // Pin 4
   case 0x0C: break;                      // Pin 5
   case 0x0E: break;                      // Pin 6
   case 0x10: break;                      // Pin 7
   default:   _never_executed();
}}
```

For grouped interrupts, though, we also need to determine which specific source caused the CPU to be interrupted. As we've described, the Interrupt Vector (IV) register is an easy way to determine the highest-priority, pending interrupt source. In the case of GPIO port 1, we would read the P1IV register.

It's common to see the IV register read within the context of a switch statement. In the above case, if the P1IV register returns "6", it means that pin 2 was our highest-priority, enabled interrupt on Port 1; therefore, its case statement is executed. (Note, the return values for each IV register are detailed in the *F5xx device Users Guide* and the *F5xx DriverLib User's Guide*. You will find similar documentation for all MSP430 devices..)

If our program was using Pin 2 on Port 1, you should see the code for case 0x06 executed if the GPIO interrupt occurs.

By the way, there are two items in the above code example which help the compiler to produce better, more optimized, code. While these intrinsic functions are not specific to interrupt processing, they are useful in creating optimized ISR's.

- The *__even_in_range()* intrinsic function provides the compiler a bounded range to evaluate. In other words, this function tells the compiler to only worry about even results that are lower or equal to 10.
- Likewise the *_never_executed()* intrinsic tells the compiler that, in this case, "default" will never occur.

Enabling Interrupts

Earlier in the chapter we learned that for the CPU to recognize an interrupt two enable bits must be set:

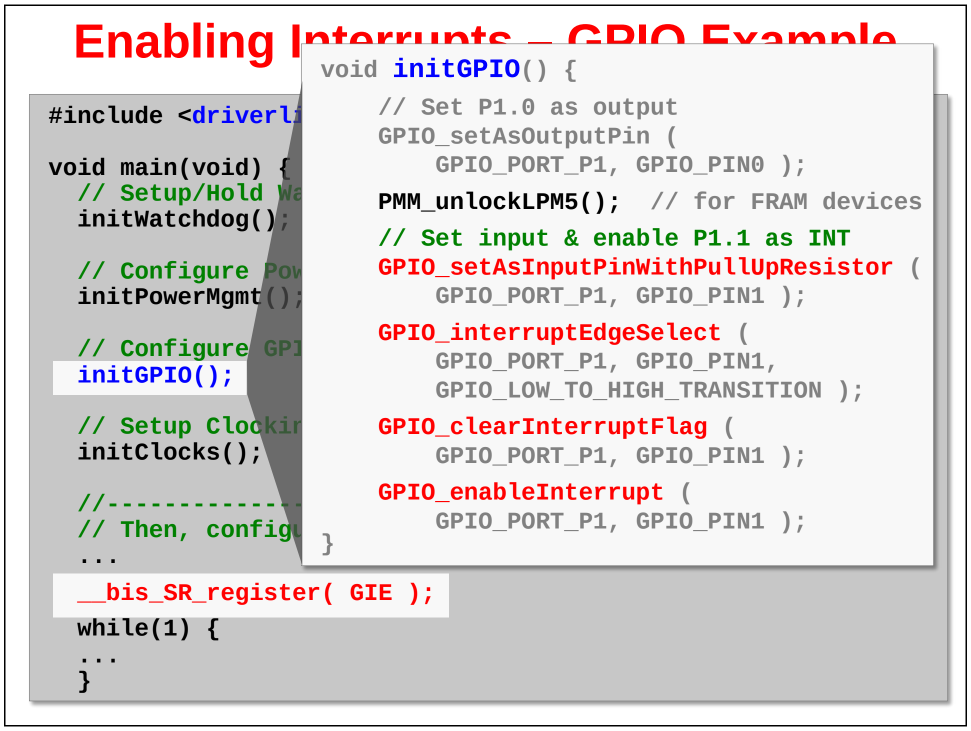
- **Individual Enable** – one IE bit for each interrupt source
- **Global Interrupt Enable** – GIE is a common "master" enable bit for all interrupts (except those defined as non-maskable)

In the example below we show the code required to setup a GPIO pin as an interrupt. We chose to enable the interrupt, as well as configuring the other GPIO pins, in a function called initGPIO(); implementing your code in this way is not required, but it's how we decided to organize our code.

The key DriverLib function which enables the external interrupt is:

```
GPIO_enableInterrupt()
```

You will find that most of the MSP430ware DriverLib interrupt enable functions take a similar form: `<module>_enableInterrupt()`.



Enabling Interrupts – GPIO Example

```
#include <driverli

void main(void) {
  // Setup/Hold Wa
  initWatchdog();

  // Configure Pow
  initPowerMgmt();

  // Configure GPI
  initGPIO();

  // Setup Clockin
  initClocks();

  //-------------
  // Then, config
  ...
  __bis_SR_register( GIE );
  while(1) {
  ...
  }
```

```
void initGPIO() {
    // Set P1.0 as output
    GPIO_setAsOutputPin (
        GPIO_PORT_P1, GPIO_PIN0 );
    PMM_unlockLPM5();  // for FRAM devices
    // Set input & enable P1.1 as INT
    GPIO_setAsInputPinWithPullUpResistor (
        GPIO_PORT_P1, GPIO_PIN1 );
    GPIO_interruptEdgeSelect (
        GPIO_PORT_P1, GPIO_PIN1,
        GPIO_LOW_TO_HIGH_TRANSITION );
    GPIO_clearInterruptFlag (
        GPIO_PORT_P1, GPIO_PIN1 );
    GPIO_enableInterrupt (
        GPIO_PORT_P1, GPIO_PIN1 );
}
```

Within initGPIO() we highlighted three other related functions in **Red**:

- **GPIO_setAsInputPinWithPullUpResistor()** is required to configure the pin as an input. On the Launchpad, the hardware requires a pull-up resistor to complete the circuit properly. Effectively, this function configures our interrupt "source".
- **GPIO_interruptEdgeSelect()** should be used to configure what edge transition (low-to-high or high-to-low) will trigger an interrupt. This configures bits in the port's IES register – which are left uninitialized after reset.

- **GPIO_clearInterruptFlag()** clears the IFG bit associated with our pin (e.g. P1.1). This is not required but is commonly used right before a call to "enable" an interrupt. You would clear the IFG before setting IE when you want to ignore any prior interrupt event; in other words, clear the flag first if you only care about interrupts that will occur now – or in the future.

Finally, once you have enabled each individual interrupt, the global interrupt needs to be enabled. This can be done in a variety of ways. The two most common methods utilize compiler intrinsic functions:

- **__bis_SR_register(GIE)** instructs the compiler to set the GIE bit in the Status Register
  - bis = bit set
  - SR = Status Register
  - GIE = which bit to set in the SR
- **__enable_interrupts(void)** tells the compiler to enable interrupts. The compiler uses the EINT assembly instruction which pokes 1 into the GIE bit.

Sidebar – Where in your code should you enable GIE?

The short answer, "Whenever you need to turn on interrupts".

A better answer, as seen in our code example, is "right before the while{} loop".

Conceptually, the main() function for most embedded systems consists of two parts:

- Setup
- Loop

That is, the first part of the main() function is where we tend to setup our I/O, peripherals, and other system hardware. In our example, we setup the watchdog timer, power management, GPIO, and finally the system clocks.

The second part of main() usually involves an infinite loop – in our example, we coded this with an endless while{} loop. An infinite loop is found in almost all embedded systems since we want to run forever after the power is turned on.

The most common place to enable interrupts globally (i.e. setting GIE) is right between these two parts of main(). Looking at the previous code example, this is right where we placed our function that sets GIE.

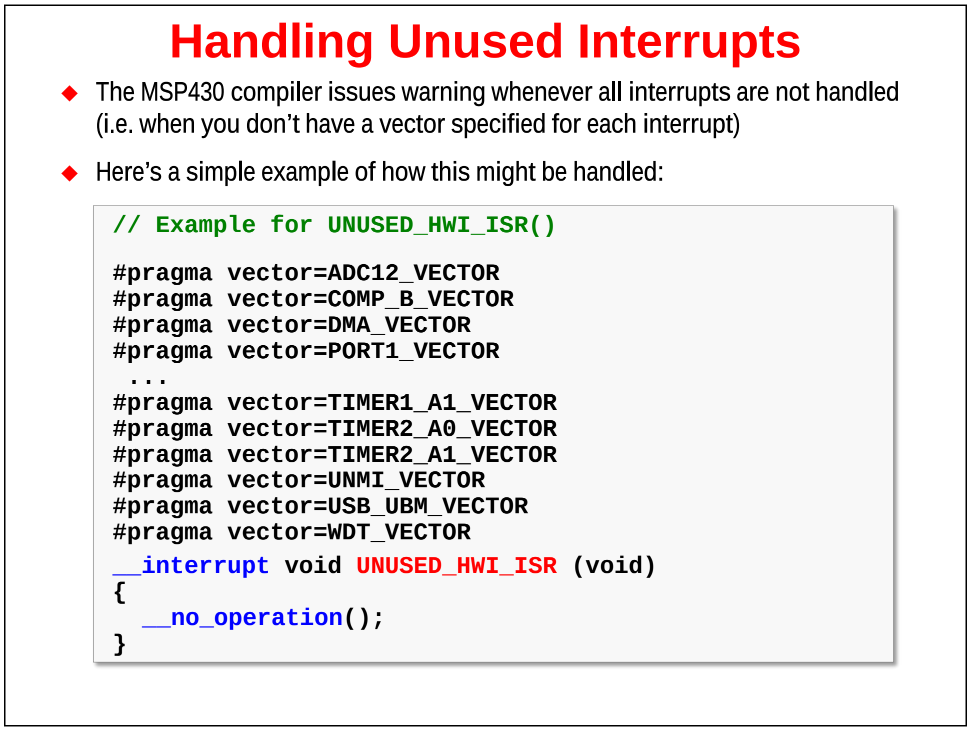
As a product example, think of the A/C power adaptor you use to charge your computer; most of these, today, utilize an inexpensive microcontroller to manage them. (In fact, the MSP430 is very popular for this type of application.) When you plug in your power adapter, we're guessing that you would like it to run as long as it's plugged in. In fact, this is what happens; once plugged in, the first part of main() sets up the required hardware and then enters an endless loop which controls the adaptor. What makes the MSP430 such a good fit for this application is: (1) it's inexpensive; and (2) when a load is not present and nothing needs to be charged, it can turn off the external charging components and put itself to sleep – until a load is inserted and wakes the processor back up.

Miscellaneous Topics

Handling Unused Interrupts

While you are not required to provide interrupt vectors – or ISR's – for every CPU interrupt, it's considered good programming practice to do so. To this end, the MSP430 compiler issues a warning whenever there are "unhandled" interrupts.

The following code is an example that you can include in all your projects. Then, as you implement an interrupt and write an ISR for it, just comment the associated #pragma line from this file.



Handling Unused Interrupts

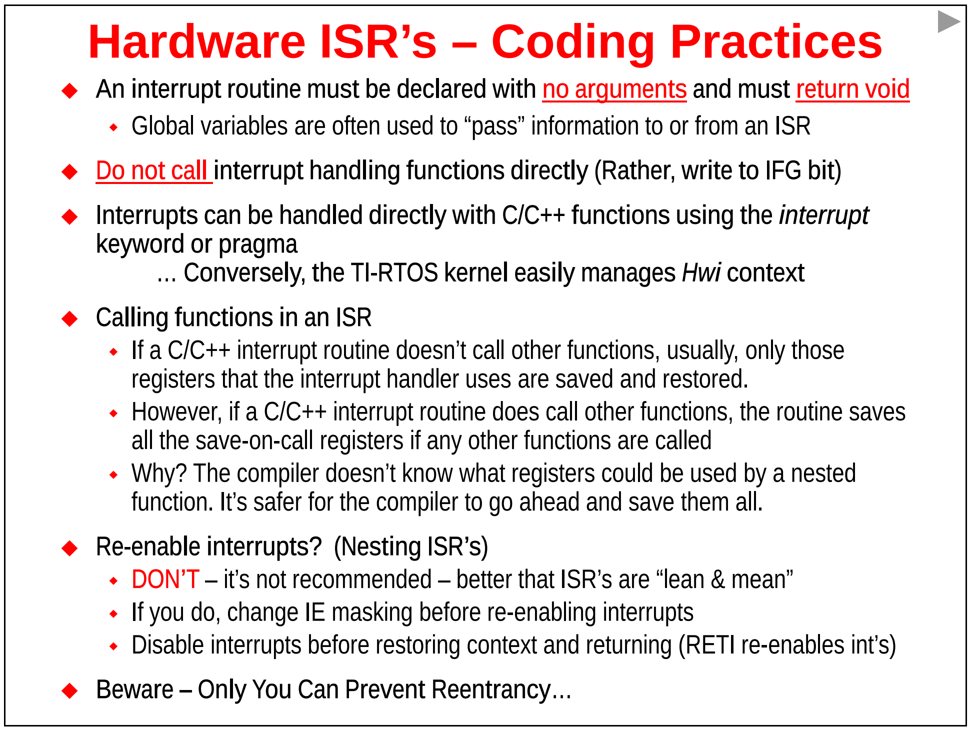
- The MSP430 compiler issues warning whenever all interrupts are not handled (i.e. when you don't have a vector specified for each interrupt)
- Here's a simple example of how this might be handled:

```
// Example for UNUSED_HWI_ISR()

#pragma vector=ADC12_VECTOR
#pragma vector=COMP_B_VECTOR
#pragma vector=DMA_VECTOR
#pragma vector=PORT1_VECTOR
  ...
#pragma vector=TIMER1_A1_VECTOR
#pragma vector=TIMER2_A0_VECTOR
#pragma vector=TIMER2_A1_VECTOR
#pragma vector=UNMI_VECTOR
#pragma vector=USB_UBM_VECTOR
#pragma vector=WDT_VECTOR
__interrupt void UNUSED_HWI_ISR (void)
{
  __no_operation();
}
```

Note: The TI code generation tools distinguish between "warnings" and "errors". Both represent issues found during compilation and build, but whereas a *warning* is issued and code building continues … when an *error* is encountered, an *error* statement is issued and the tools stop before creating a final executable.

Interrupt Service Routines – Coding Suggestions

Listed below are a number of required and/or good coding practices to keep in mind when writing hardware interrupt service routines. Many of these have been discussed elsewhere in this chapter.



Hardware ISR's – Coding Practices

- An interrupt routine must be declared with no arguments and must return void
  - Global variables are often used to "pass" information to or from an ISR
- Do not call interrupt handling functions directly (Rather, write to IFG bit)
- Interrupts can be handled directly with C/C++ functions using the *interrupt* keyword or pragma
  ... Conversely, the TI-RTOS kernel easily manages *Hwi* context
- Calling functions in an ISR
  - If a C/C++ interrupt routine doesn't call other functions, usually, only those registers that the interrupt handler uses are saved and restored.
  - However, if a C/C++ interrupt routine does call other functions, the routine saves all the save-on-call registers if any other functions are called
  - Why? The compiler doesn't know what registers could be used by a nested function. It's safer for the compiler to go ahead and save them all.
- Re-enable interrupts? (Nesting ISR's)
  - DON'T – it's not recommended – better that ISR's are "lean & mean"
  - If you do, change IE masking before re-enabling interrupts
  - Disable interrupts before restoring context and returning (RETI re-enables int's)
- Beware – Only You Can Prevent Reentrancy…

We wrote the last bullet, regarding reentrancy, in a humorous fashion. That said, it speaks to an important point. If you decide to enable interrupt nesting, you need to be careful that you either prevent reentrancy - or that your code is capable of reentrancy.

Wikipedia defines reentrancy as:

> In *computing*, a *computer program* or *subroutine* is called **reentrant** if it can be interrupted in the middle of its execution and then safely called again ("re-entered") before its previous invocations complete execution.

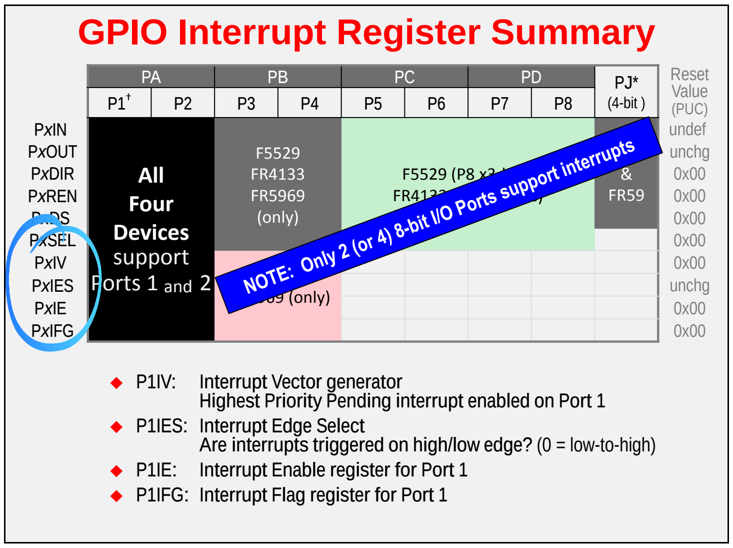
This type of program/system error can be very difficult to debug (i.e. find and fix). This is especially true if you call functions within your interrupt service routines. For example, the C language's malloc() function is not reentrant. If you were to call this function from an ISR and it was interrupted, and then it is called again by another ISR, your system would most likely fail – and fail in a way that might be very difficult to detect.

So, we stated this humorously, but it is very true. We recommend that:

- You shouldn't nest interrupts
- If you do, verify the code in your ISR is reentrant
- Never call malloc() – or similar functions - from inside an ISR

GPIO Interrupt Summary

The diagram used to summarize the GPIO control registers in a previous chapter is a good way to visualize the GPIO interrupt capabilities of our devices. From the diagram below we can see that most MSP430 processors allow ports P1 and P2 to be used as external interrupt sources; we see this from the fact that these ports actually have the required port interrupt registers.



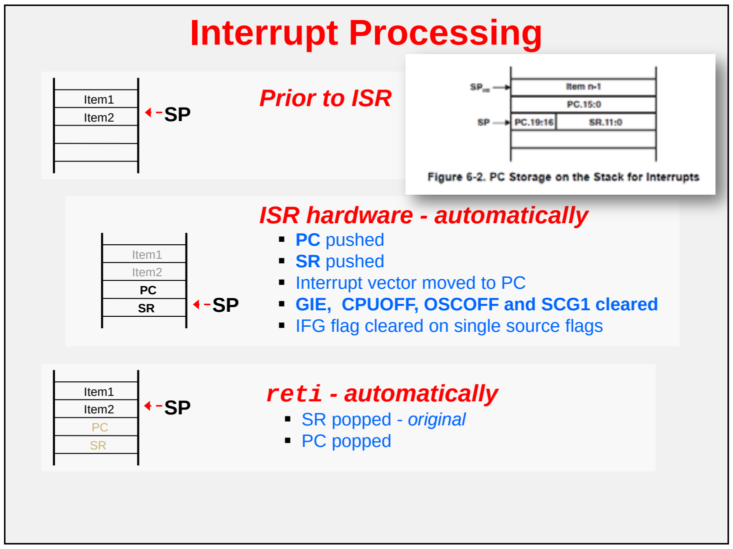
GPIO Interrupt Register Summary

◆ P1IV:	Interrupt Vector generator Highest Priority Pending interrupt enabled on Port 1
◆ P1IES:	Interrupt Edge Select Are interrupts triggered on high/low edge? (0 = low-to-high)
◆ P1IE:	Interrupt Enable register for Port 1
◆ P1IFG:	Interrupt Flag register for Port 1

There are other devices in the MSP430 family that support interrupts on more than 2 ports, but of the three example processors we use throughout this course, only the FR5969 (FRAM) devices support interrupt inputs on additional ports (P3 and P4).

Interrupt Processing Flow

The following information was previously covered in this chapter, but since the slide is a good summary of the interrupt processing flow, we have included it anyway.



Interrupt Processing

Prior to ISR

Figure 6-2. PC Storage on the Stack for Interrupts

ISR hardware - automatically

- **PC** pushed
- **SR** pushed
- Interrupt vector moved to PC
- **GIE, CPUOFF, OSCOFF and SCG1 cleared**
- IFG flag cleared on single source flags

`reti` - automatically

- SR popped - *original*
- PC popped

Interrupts and TI-RTOS Scheduling

When embedded systems start to become more complex – that is, when you need to juggle more than a handful of events – using a Real-Time Operating System (RTOS) can greatly increase your system's reliability … while decreasing your time-to-market, frustration and costs.

The Texas Instruments RTOS (TI-RTOS) – also known as SYS/BIOS – provides many functions that you can use within your program; for example, the TI-RTOS kernel includes: *Scheduling*, *Instrumentation*, and *Memory Management*. You can choose which parts of TI-RTOS are needed and discard the rest (to saves memory).

Think of TI-RTOS as a library and toolset to help you build and maintain robust systems. If you're doing just "one" thing, it's probably overkill. As you end up implementing more and more functionality in your system, though, the tools and code will save you time and headaches.

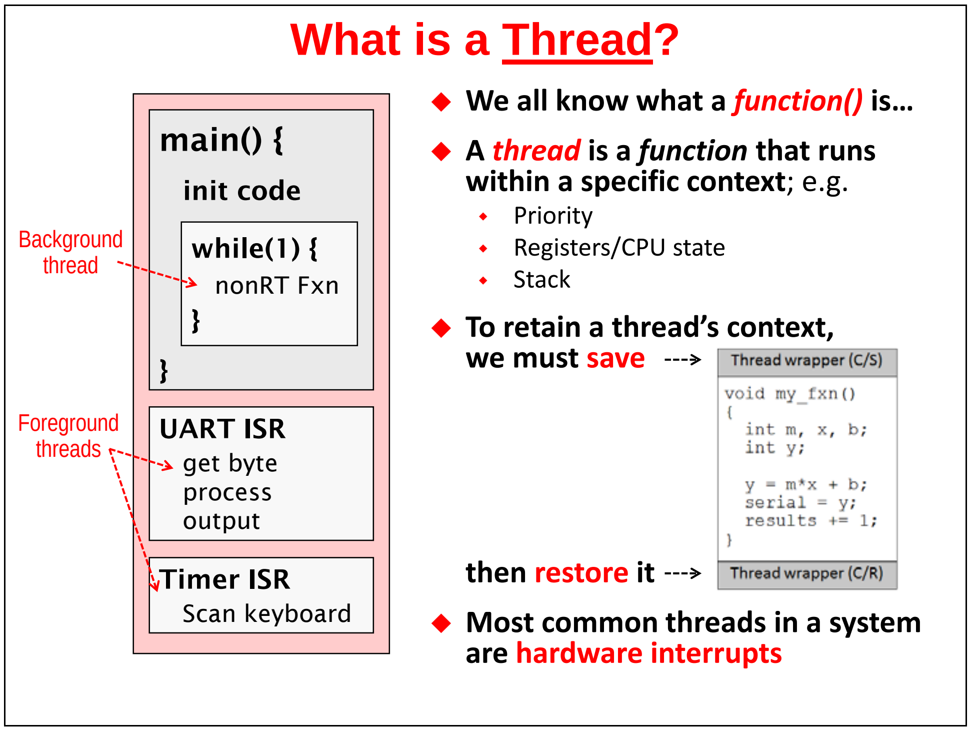
The only part of TI-RTOS discussed in this chapter is "Scheduling". We talk about this because it is very much related to the topics covered throughout this chapter – interrupts and threads. In many cases, if you're using an RTOS, it will manage much of the interrupt processing for you; it will also provide additional options for handling interrupts – such as post-processing of interrupts.

As a final note, we will only touch on the topics of scheduling and RTOS's. TI provides a 2-day workshop where you can learn all the details of the TI-RTOS kernel. You can view a video version of the TI-RTOS course or take one live. Please check out the following wiki page for more information:

http://processors.wiki.ti.com/index.php/Introduction_to_the_TI-RTOS_Kernel_Workshop

Threads – Foreground and Background

Our quick introduction to TI-RTOS begins with a summary of threads. While we discussed these concepts earlier in the chapter, they are very important to how a RTOS scheduler works.



What is a Thread?

```
main() {
  init code
  while(1) {
    nonRT Fxn
  }
}
UART ISR
  get byte
  process
  output
Timer ISR
  Scan keyboard
```

- We all know what a *function()* is…
- A *thread* is a ***function*** that runs within a specific context; e.g.
  - Priority
  - Registers/CPU state
  - Stack
- To retain a thread's context, we must save ---> Thread wrapper (C/S)

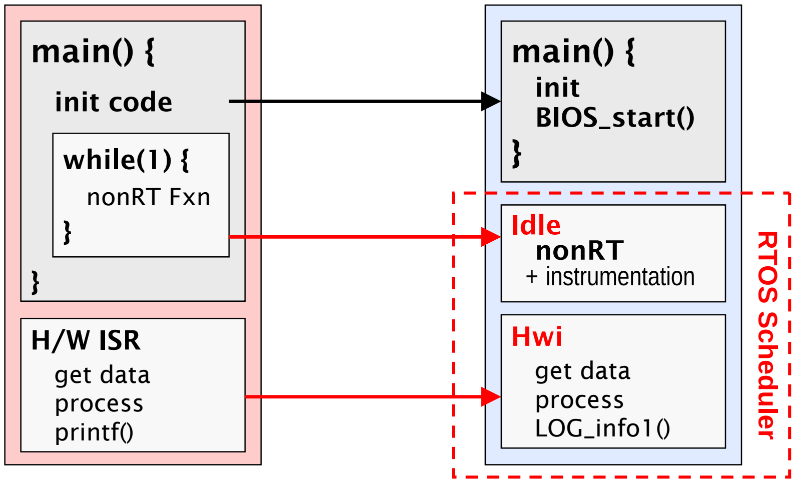
```
void my_fxn()
{
  int m, x, b;
  int y;

  y = m*x + b;
  serial = y;
  results += 1;
}
```

then restore it ---> Thread wrapper (C/R)

- Most common threads in a system are hardware interrupts

We also discussed the idea of foreground and background threads as part of the interrupts chapter. In the case shown below (on the left), the endless loop in main() will run forever and be pre-empted by higher-priority hardware interrupts.

Foreground / Background Scheduling



- **Idle events run in sequence when no Hwi's are posted**
- **Hwi is ISR with automatic vector table generation + context save/restore**
- **Hwi performs "process" – typical use is to perform HRT need, then post "follow-up activity"**

TI-RTOS utilizes these same concepts … only the names and threads change a little bit.

Rather than main() containing both the *setup* and *loop* code as described earlier, TI-RTOS creates an ***Idle*** thread that operates in place of the while{} loop found previously in main(). In other words, rather than adding your functions to a while{} loop, TI-RTOS has you add them to ***Idle***. (TI-RTOS includes a GUI configuration tool that makes this very easy to do.)

Since interrupts are part of the MSP430's hardware, they essentially work the same way when using TI-RTOS. What changes when using RTOS are:

- TI-RTOS calls them ***Hwi*** threads … for Hardware Interrupts
- Much of the coding effort is handled automatically for you by TI-RTOS (very nice)

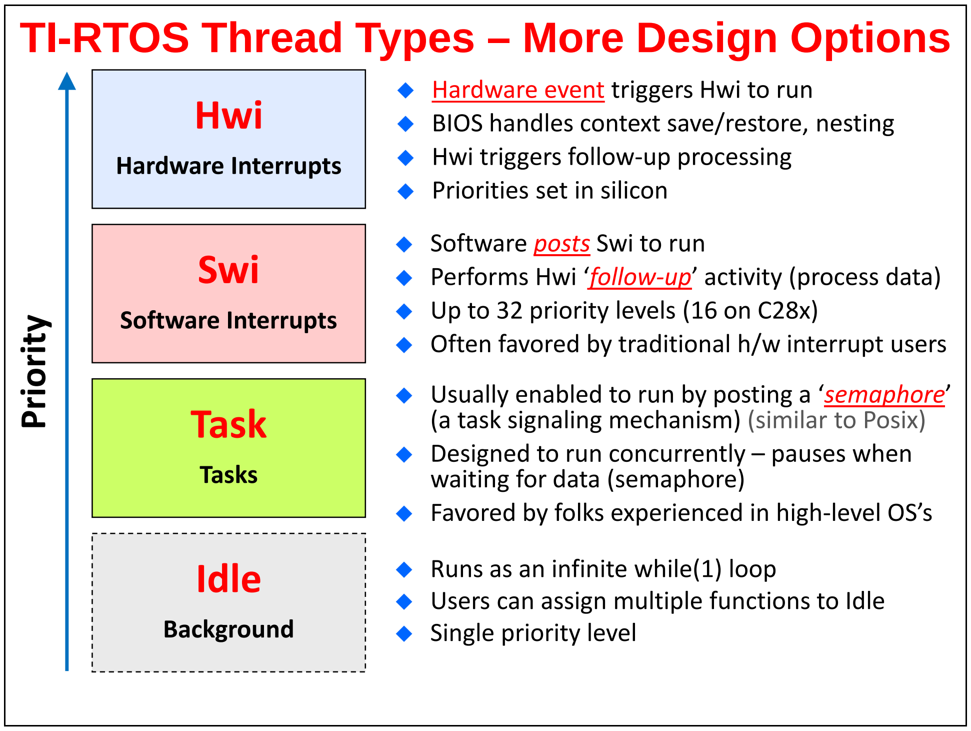
Don't worry, though, you're not locked into anything. You can mix-and-match how you handle interrupts. Let TI-RTOS manage some of your interrupts while handling others in your own code, just as we described in this chapter.

Hint: When using TI-RTOS, the author prefers to let it manage all of the interrupts because it's easier that way. Only

Only in a rare case – like to save a few CPU cycles – would there be a need to managed an interrupt outside of TI-RTOS. Thusfar, the only reason I've actually done this is to provde that it works.

TI-RTOS Thread Types

We already described two types of threads: ***Hwi*** and ***Idle***. Using these two threads is very similar to what we described throughout this chapter.



TI-RTOS Thread Types – More Design Options

Hwi — **Hardware Interrupts**

- Hardware event triggers Hwi to run
- BIOS handles context save/restore, nesting
- Hwi triggers follow-up processing
- Priorities set in silicon

Swi — **Software Interrupts**

- Software *posts* Swi to run
- Performs Hwi '*follow-up*' activity (process data)
- Up to 32 priority levels (16 on C28x)
- Often favored by traditional h/w interrupt users

Task — **Tasks**

- Usually enabled to run by posting a '*semaphore*' (a task signaling mechanism) (similar to Posix)
- Designed to run concurrently – pauses when waiting for data (semaphore)
- Favored by folks experienced in high-level OS's

Idle — **Background**

- Runs as an infinite while(1) loop
- Users can assign multiple functions to Idle
- Single priority level

TI-RTOS provides two additional thread types: ***Software Interrupts*** (***Swi***) and ***Tasks*** (***Task***). As you can see above, these thread types fall between ***Hwi*** and ***Idle*** in terms of priority.

Each of these threads can be used to extend your system's processing organization.

What do we mean by this?

You might remember that we HIGHLY recommended that you DO NOT nest interrupts. But what happens if you want to run an algorithm based on some interrupt event? For example, you want to run a filter whenever you receive a value from an A/D converter or from the serial port.

Without an RTOS, you would need to organize your main while{} loop to handle all of these interrupt, follow-up tasks. This is not a problem for one or two events; but for lots of events, this can become very complicated – especially when they all run at different rates. This way of scheduling your processing is called a *SuperLoop*.

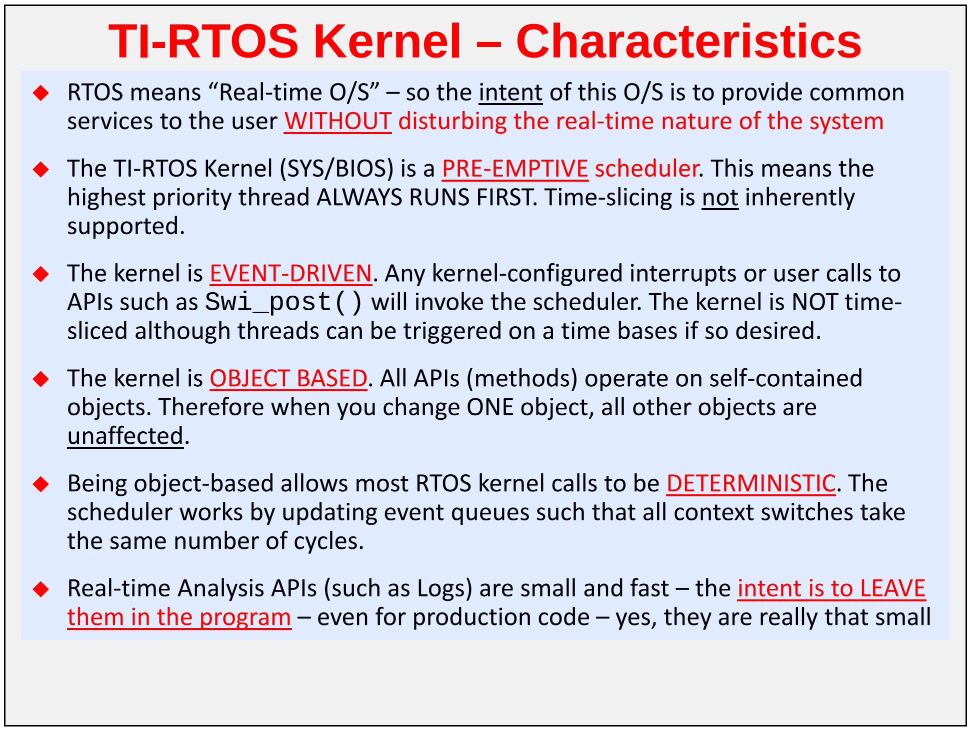
With an RTOS, we can post follow-up activity to a Swi or Task. A ***Swi*** acts just like a software triggered interrupt service routine. ***Tasks***, on the other hand, run all the time (*have you heard the term multi-tasking before?*) and utilize ***Semaphores*** to signal when to run or when to *block* (i.e. pause).

In other words, ***Swi's*** and ***Task's*** provide two different ways to schedule follow-up processing code. They let us keep our hardware interrupts (Hwi's) very short and simple – for example, all we need to do is read our ADC and then post an associated ***Swi*** to run.

If all of this sounds complicated, it really isn't. While outside the scope of this course, the TI-RTOS course will have you up-and-running in no time. Once you experience the effective organization provided by an RTOS, you may never build another system without one.

TI-RTOS Details

The following slide provides some “characteristics” of the TI-RTOS kernel. The bottom-line here is that it is a priority-based scheduler. The highest priority thread gets to run, period. (Remember, hardware interrupts are always the highest priority.)



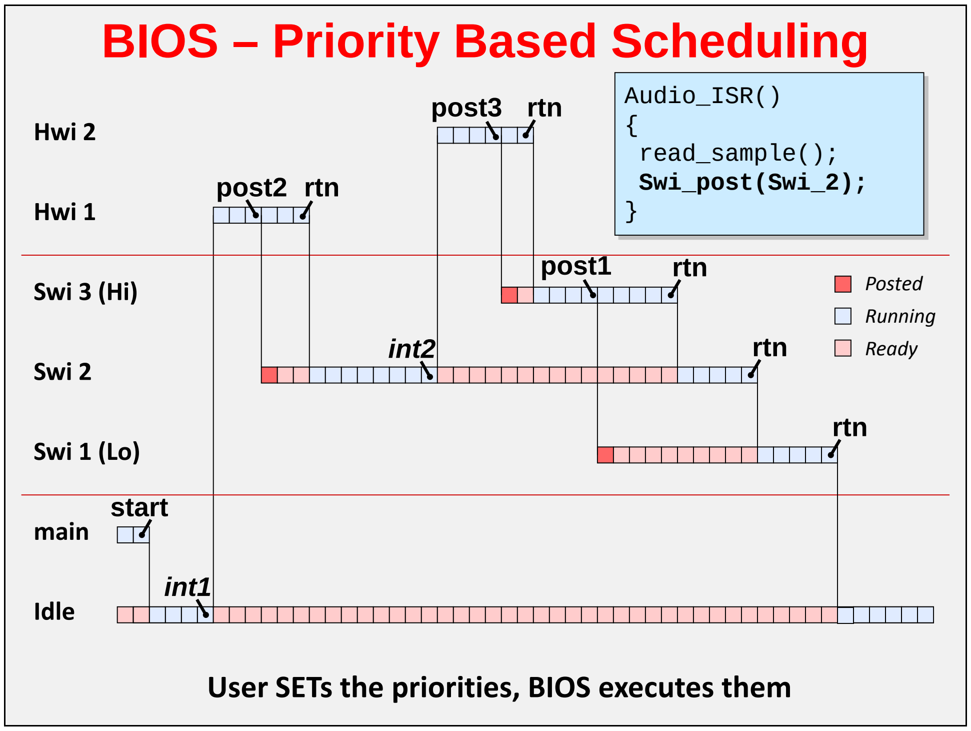
TI-RTOS Kernel – Characteristics

- RTOS means “Real-time O/S” – so the intent of this O/S is to provide common services to the user WITHOUT disturbing the real-time nature of the system
- The TI-RTOS Kernel (SYS/BIOS) is a PRE-EMPTIVE scheduler. This means the highest priority thread ALWAYS RUNS FIRST. Time-slicing is not inherently supported.
- The kernel is EVENT-DRIVEN. Any kernel-configured interrupts or user calls to APIs such as `Swi_post()` will invoke the scheduler. The kernel is NOT time-sliced although threads can be triggered on a time bases if so desired.
- The kernel is OBJECT BASED. All APIs (methods) operate on self-contained objects. Therefore when you change ONE object, all other objects are unaffected.
- Being object-based allows most RTOS kernel calls to be DETERMINISTIC. The scheduler works by updating event queues such that all context switches take the same number of cycles.
- Real-time Analysis APIs (such as Logs) are small and fast – the intent is to LEAVE them in the program – even for production code – yes, they are really that small

While you can construct a time-slicing system using TI-RTOS, this is not commonly done. While time-slicing can be a very effective technique in host operating systems (like Windows or Linux), it is not a common method for scheduling threads in an embedded system.

Hwi – Swi – Idle Scheduling

Here’s a simple, visual example of what real-time scheduling might look like in an RTOS based system.



```
Audio_ISR()
{
 read_sample();
 Swi_post(Swi_2);
}
```

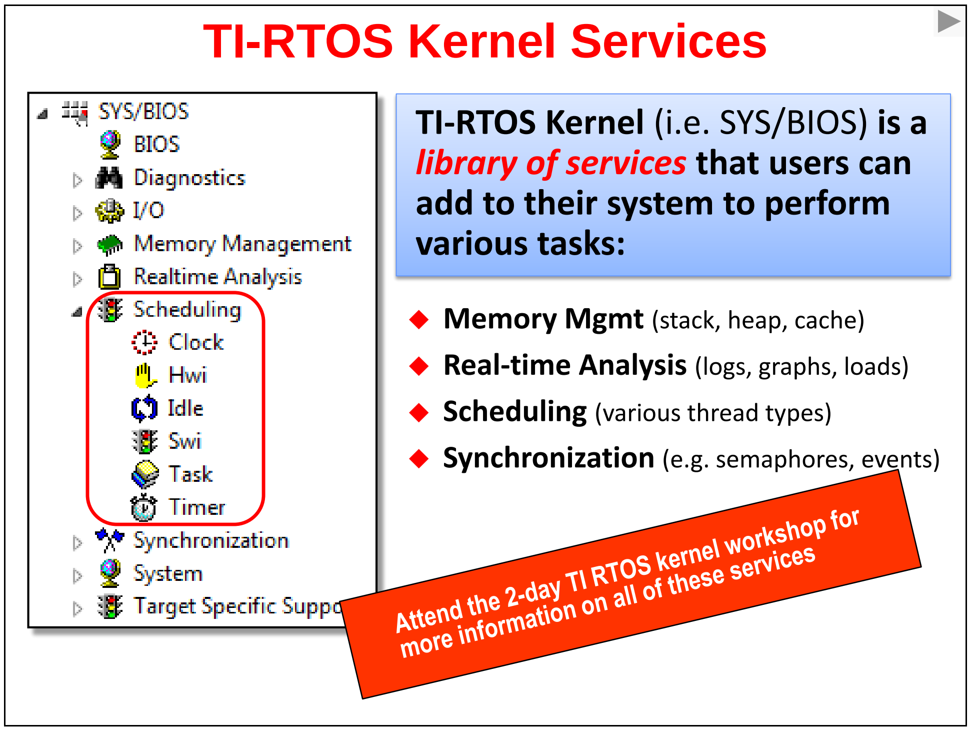
Notice how the system enters Idle from main(). Idle is always ready to run (just as our old while{} loop was always ready to run).

When a hardware interrupt (Hwi) occurs, we leave Idle and execute the Hwi thread’s code. Since it appears the Hwi posted a Swi, that’s where the TI-RTOS scheduler goes to once the Hwi finishes.

We won’t go through the remaining details in this course, though we suspect that you can all follow the diagram. For this slide, and a lot more information, please refer to the TI-RTOS Kernel Workshop.

Summary: TI-RTOS Kernel

The following slide summarizes much of the functionality found in the TI-RTOS kernel. In this chapter we've only touched on the scheduling features.



TI-RTOS Kernel Services

SYS/BIOS
BIOS
Diagnostics
I/O
Memory Management
Realtime Analysis
Scheduling
Clock
Hwi
Idle
Swi
Task
Timer
Synchronization
System
Target Specific Suppo

TI-RTOS Kernel (i.e. SYS/BIOS) **is a *library of services* that users can add to their system to perform various tasks:**

- **Memory Mgmt** (stack, heap, cache)
- **Real-time Analysis** (logs, graphs, loads)
- **Scheduling** (various thread types)
- **Synchronization** (e.g. semaphores, events)

Attend the 2-day TI RTOS kernel workshop for more information on all of these services

The TI-RTOS product includes the kernel, shown above, along with a number of additional drivers and stacks. Oh, and the kernel comes with complete source code – nothing is hidden from you.

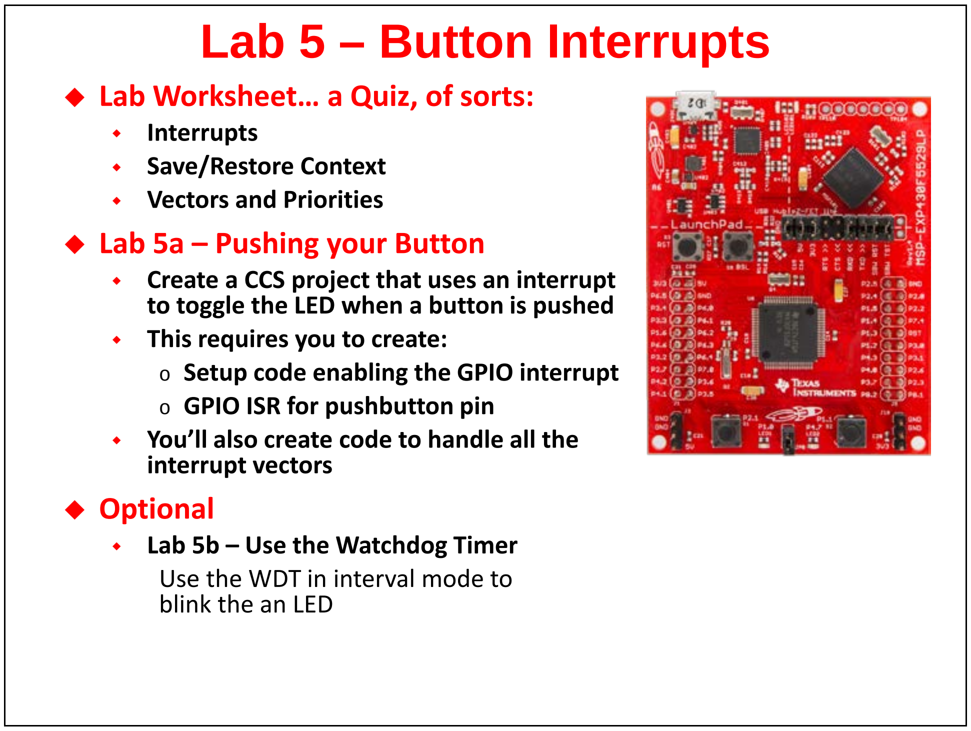
For many, though, one of the compelling features of TI-RTOS is that it's FREE*.

Remember, we make our money selling you devices. Our code and tools are there to help you get your programs put together – and your systems to market – more quickly.

* That is, it's free for use on all Texas Instruments processors.

Lab 5 – Interrupts

This lab introduces you to programming MSP430 interrupts. Using interrupts is generally one of the core skills required when buiding embedded systems. If nothing else, it will be used extensively in later chapters and lab exercises.



Lab 5 – Button Interrupts

- **Lab Worksheet… a Quiz, of sorts:**
  - **Interrupts**
  - **Save/Restore Context**
  - **Vectors and Priorities**
- **Lab 5a – Pushing your Button**
  - **Create a CCS project that uses an interrupt to toggle the LED when a button is pushed**
  - **This requires you to create:**
    - **Setup code enabling the GPIO interrupt**
    - **GPIO ISR for pushbutton pin**
  - **You'll also create code to handle all the interrupt vectors**
- **Optional**
  - **Lab 5b – Use the Watchdog Timer**

    Use the WDT in interval mode to blink the an LED

Lab 5a covers all the essential details of interrupts:

- Setup the interrupt vector
- Enable interrupts
- Create an ISR

When complete, you should be able to push the SW1 button and toggle the Red LED on/off.

Lab 5b is listed as optional since, while these skills are valuable, you should know enough at the end of Lab 5a to move on and complete the other labs in the workshop.

Lab Topics

Lab 5 Worksheet

General Interrupt Questions

Hint: You can look in the Chapter 5 discussion for the answers to these questions

1. When your program is not in an interrupt service routine, what code is it usually executing? And, what ‘name’ do we give this code?

   ____

2. Why keep ISR’s short? That is, why shouldn’t you do a lot of processing in them)?

   ____

   ____

3. What causes the MSP430 to exit a Low Power Mode (LPMx)?

   ____

4. Why are *interrupts* generally preferred over *polling*?

   ____

   ____

Interrupt Flow

5. Name 4 sources of interrupts? *(Well, we gave you one, so name 3 more.)*
   Hint: Look at the chapter discussion, datasheet or User's Guide for this answer.

   TIMER_A ______________________________________

   ____

   ____

   ____

6. What signifies that an interrupt has occurred?
   Hint: Look at the "Interrupt Flow" part of this chapter discussion.

   A __________ bit is set

   What's the acronym for these types of 'bits" ___________

Setting up GPIO Port Interrupts

Next, let's review the code required to setup one of the Launchpad buttons for GPIO input.
(Hint: Look in the Chapter 5 "Enabling Interrupts" discussion for help on the next two questions.)

7. Write the code to enable a GPIO interrupt for the listed Port.Pin?

```
// GPIO pin to use:   F5529 = P1.1,   FR4133 = P1.2,  FR5969 = P1.1

____________________________________________ // setup pin as input

____________________________________________ // set edge select

____________________________________________ // clear individual flag

____________________________________________ // enable individual interrupt
```

8. Write the line of code required to turn on interrupts globally:

```
______________________________________ // enable global interrupts (GIE)
```

   Where, in our programs, is the most common place we see GIE enabled?
   (Hint: you can look back at the sidebar discussion where we showed how to do this.)

   ____

Interrupt Priorities & Vectors

9. Check the interrupt that has higher priority. (*Hint: Look at the chapter discussion or device datasheet for the answer.)*

- ❑ GPIO Port 2
- ❑ WDT Interval Timer

10. Where do you find the name of an "interrupt vector" (e.g. PORT1_VECTOR)?

Hint: Which header file defines symbols for each device?

___

11. Write the code to set the interrupt vector? *(To help, we've provided a simple ISR to go with the line of code we're asking you to complete. Finish the #pragma statement...)*

```
// Put's the ISR function's address into the Port 1 vector location

#pragma ____________________________________________

__interrupt void pushbutton_ISR (void)
{
    // Toggle the LED on/off
    GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
}
```

What is wrong with this GPIO port ISR?

___

___

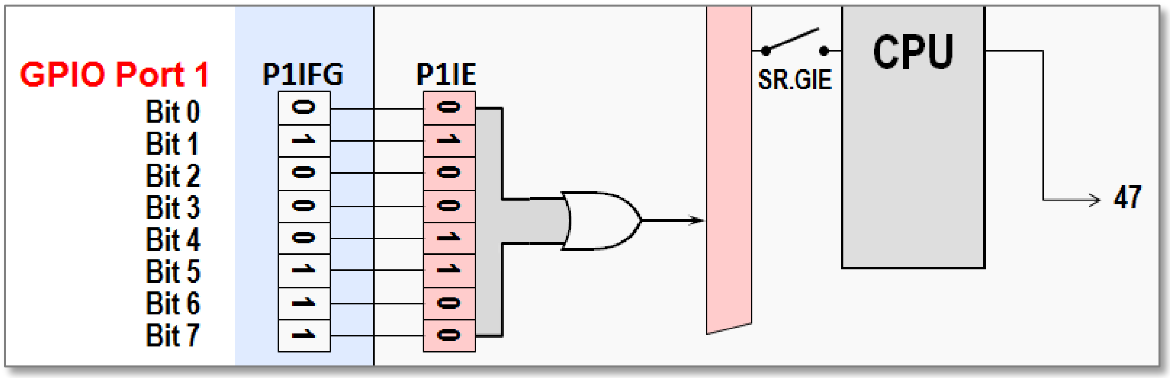
12. How do you pass a value into (or out from) and interrupt service routine (ISR)?

Hint: Look at the chapter topic "Interrupt Service Routines – Coding Suggestions".

___

ISR's for Group Interrupts

As we learned earlier, most MSP430 interrupts are grouped. For example, the GPIO port interrupts are all grouped together. *(Hint: To answer these last two questions, look at the discussion titled "Grouped ISR" in this chapter's discussion.)*



13. For dedicated interrupts (such as WDT interval timer) the CPU clears the IFG flag when responding to the interrupt. How does an IFG bit get cleared for group interrupts?

____

____

14. Creating ISR's for grouped interrupts is as easy as following a 'template'. The following code represents a grouped ISR template.
    - Fill in the appropriate blank line to respond to the Port 1 pin used for the pushbutton on your Launchpad. (F5529/FR5969 = P1.1; FR4133 = P1.2)
    - Add the code needed to toggle the LED (on `P1.0`) in response to the button interrupt.

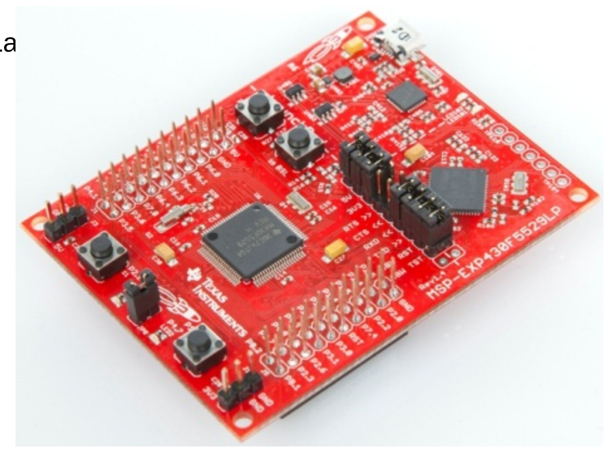
```
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void) {
   switch(__even_in_range( ____________________, 0x10 )){
        case 0x00: break;         // None
        case 0x02: break;         // Pin 0

             _____________________________________________
             break;
        case 0x04:                // Pin 1

             _____________________________________________
             break;
        case 0x06:                // Pin 2

             _____________________________________________
             break;
        case 0x08:                // Pin 3

             _____________________________________________
             break;
        case 0x0A:                // Pin 4

             _____________________________________________
             break;
        case 0x0C:                // Pin 5

             _____________________________________________
             break;
        case 0x0E:                // Pin 6

             _____________________________________________
             break;
        case 0x10:                // Pin 7

             _____________________________________________
        default:
             _never_executed();
   }
```

Lab 5a – Push Your Button

When Lab 5a is complete, you should be able to push the S2 button and toggle the Red LED on/off.

We will begin by importing the solution to La
code and add the following.

- Setup the interrupt vector
- Enable interrupts
- Create an ISR



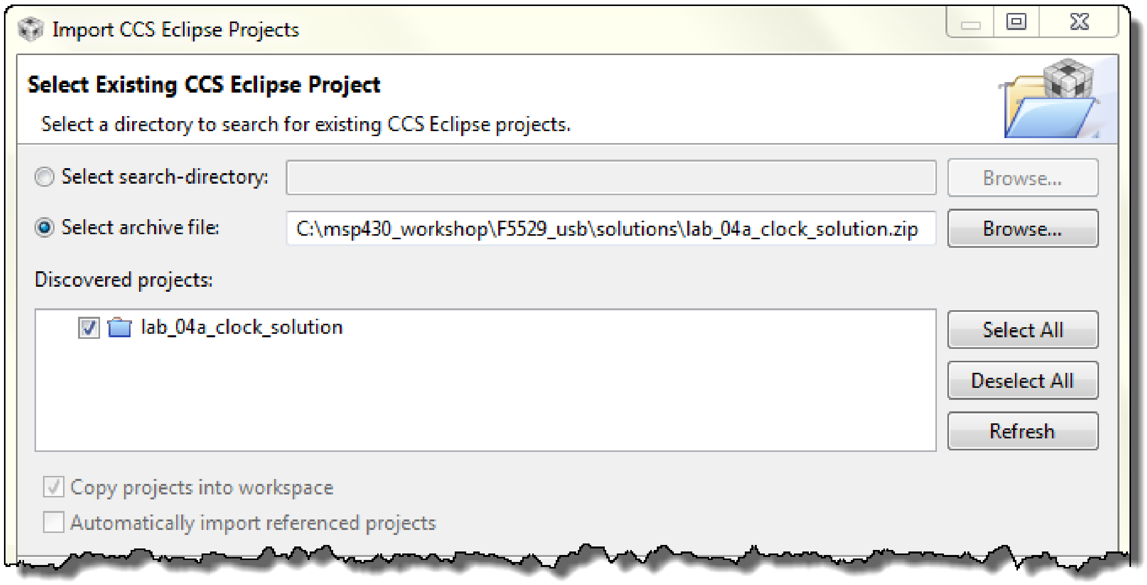
Launchpad	Pin	Button
F5529	`P1.1`	S2
FR5969	`P1.1`	S2
FR4133	`P2.2`	S1

File Management

1. **Close all previous projects. Also, close any remaining open files.**
2. **Import the solution for Lab 4a from: `lab_04a_clock_solution`**

   Select import previous CCS project from the *Project* menu:

   `Project → Import CCS Projects…`



Import CCS Eclipse Projects

Select Existing CCS Eclipse Project

Select a directory to search for existing CCS Eclipse projects.

○ Select search-directory: [ ] Browse...

● Select archive file: C:\msp430_workshop\F5529_usb\solutions\lab_04a_clock_solution.zip Browse...

Discovered projects:

☑ lab_04a_clock_solution

Select All
Deselect All
Refresh

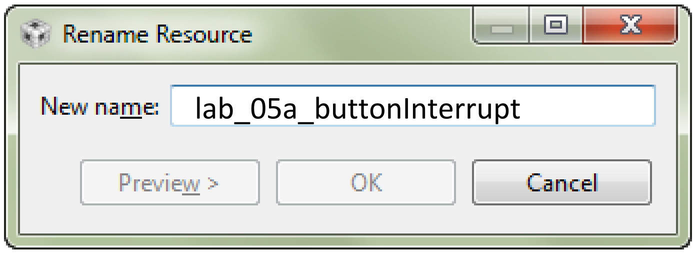
☑ Copy projects into workspace
☐ Automatically import referenced projects

3. **Rename the imported project to: `lab_05a_buttonInterrupt`**

   You can right-click on the project name and select *Rename*, though the easiest way to rename a project is to:

   ```
   Select project in Project Explorer → hit F2
   ```

   When the following dialog pops up, fill in the new project name:



New name: lab_05a_buttonInterrupt

4. **Verify the project is active, then check that it builds and runs.**

   Before we change the code, let's make sure the original project is working. Build and run the project – you should see the LED flashing once per second.

   When complete, **terminate** the debugger.

5. **Add `unused_interrupts.c` file to your project.**

   To save a lot of typing (and probably typos) we already created this file for you. You'll need to add it to your project.

   ```
   Right-click project → Add Files…
   ```

   Find the file in:

   ```
   C:\msp430_workshop\<target>\lab_05a_buttonInterrupt\unused_interrupts.c

   "Copy" the file into your project
   ```

   You can take a quick look at this file, if you'd like. Notice that we created a single ISR function that is associated with all of the interrupts on your device – since, at this point, all of the interrupts are unused. As you add each interrupt to the project, you will need to modify this file.

6. **Before we start adding new code … comment out the old code from while{} loop.**

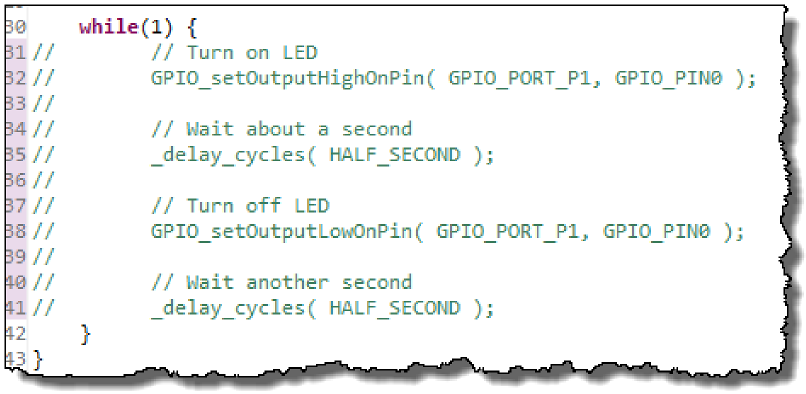
   Open `main.c` and comment out the code in the while{} loop. This is the old code that flashes the LED using the inefficient __delay_cycles() function.

   The easiest way to do this is to:

   `Select all the code in the while{} loop`

   Ctrl - / (This toggles the line comments on/off)

   Once commented, the loop should look similar to that below:



```
    while(1) {
//        // Turn on LED
//        GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 );
//
//        // Wait about a second
//        _delay_cycles( HALF_SECOND );
//
//        // Turn off LED
//        GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
//
//        // Wait another second
//        _delay_cycles( HALF_SECOND );
    }
}
```

After commenting out the while code, just double-check for errors by clicking the build button. (Fix any error that pops up.)

Configure/Enable GPIO Interrupt … Then Verify it Works

Add Code to Enable Interrupts

7. **Open `main.c` and modify initGPIO() to enable the interrupt for your push-button.**

   If you need a hint on what three lines are required, refer back to the Lab 5 Worksheet, question # 7 (see page 5-40).

   Note that the pin numbers are the same, but the switch names differ for these Launchpads:

   – For the 'F5529 Launchpad, we're using pushbutton S2 (P1.1)

   – For the 'FR5969 Launchpad, we're using pushbutton S2 (P1.1)

   – For the 'FR4311 Launchpad, we're using pushbutton S1 (P1.2)

8. **Add the line of code needed to enable interrupts globally (i.e GIE).**

   This line of code should be placed right before the while{} loop in main(). Refer back to the Lab 5 Worksheet, question # 8 (see page 5-40).

9. **Build your code.**

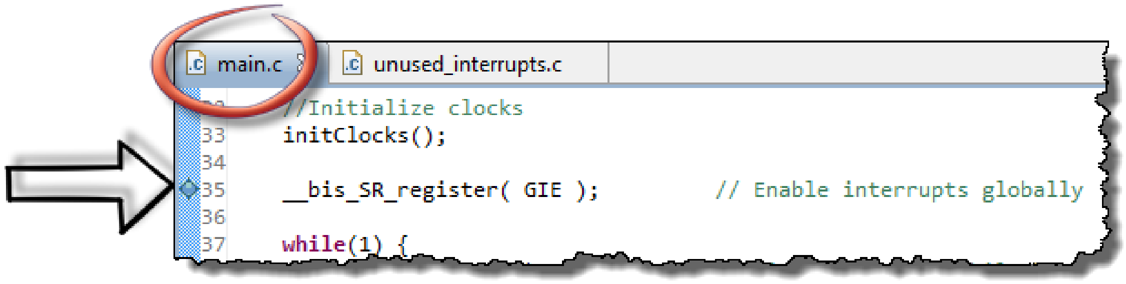
   Fix any typos or errors.

Start the Debugger and Set Breakpoints

Once the debugger opens, we'll setup two breakpoints. This allows us to verify the interrupts were enabled, as well as trapping the interrupt when it occurs.

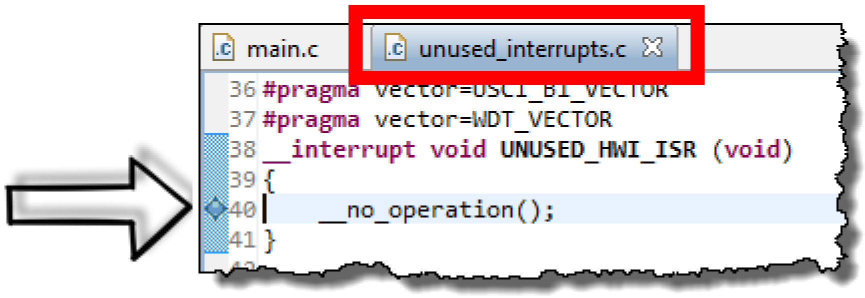
10. **Launch the debugger.**

11. **Set a breakpoint on the "enable GIE" line of code in `main.c`.**



```
    //Initialize clocks
    initClocks();

    __bis_SR_register( GIE );          // Enable interrupts globally

    while(1) {
```

12. **Next, set a breakpoint inside the ISR in the `unused_interrupts.c` file.**



```
#pragma vector=USCI_B1_VECTOR
#pragma vector=WDT_VECTOR
__interrupt void UNUSED_HWI_ISR (void)
{
      __no_operation();
}
```

Run Code to Verify Interrupts are Enabled

13. **Click Resume … the program should stop at your first breakpoint.**

14. **Open the Registers window in CCS (or show it, if it's already open).**

    If the Registers window isn't open, do so by:

    `View → Registers`

15. **Verify Port1 bits: DIR, OUT, REN, IE, IFG.**

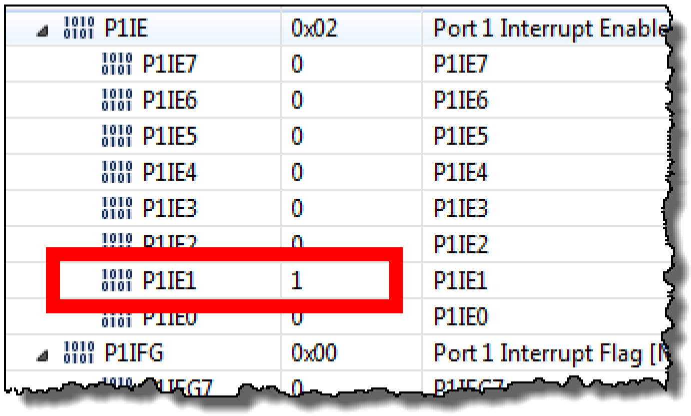
    The first breakpoint halts the processor right before setting the GIE bit. Before turning on the interrupts, let's view the GPIO Port 1 settings. Scroll/expand the registers to verify:

    - `P1DIR.0 = 1` (pin in output direction)
    - `P1DIR.1 = 0` (input direction – to be used for generating an interrupt)
    - `P1REN.1 = 1` (we enabled the resistor for our input pin)
    - `P1OUT.0 = 0` (we set it low to turn off LED)
    - `P1IE.1  = 1` (our button interrupt is enabled)
    - `P1IES.1 = 0` (configured to generate an interrupt on a low-to-high transition)
    - `P1IFG.1 = 0` (at this point, we shouldn't have received an interrupt – unless you already pushed the button…)

    Here's a snapshot of the P1IE register as an example …

FR4133

Remember, the FR4133 uses pin P1.2. Take this into account when looking at the P1 GPIO registers.

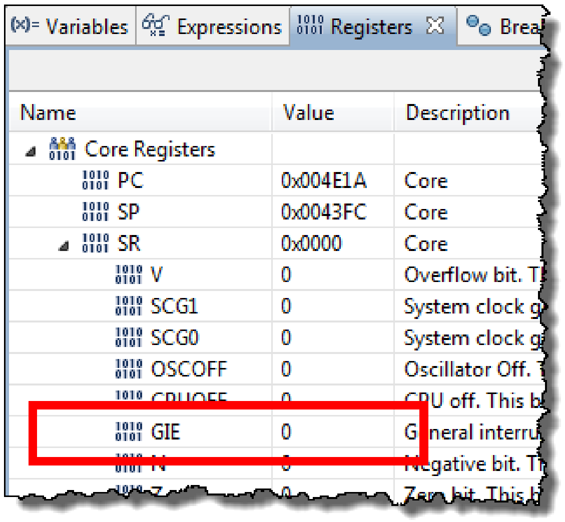


P1IE	0x02	Port 1 Interrupt Enable
P1IE7	0	P1IE7
P1IE6	0	P1IE6
P1IE5	0	P1IE5
P1IE4	0	P1IE4
P1IE3	0	P1IE3
P1IE2	0	P1IE2
P1IE1	1	P1IE1
P1IE0	0	P1IE0
P1IFG	0x00	Port 1 Interrupt Flag [
P1IFG7	0	P1IFG7

16. **Next, let's look at the *Status Register* (SR).**

    You can find it under the *Core Registers* at the top of the *Registers* window.

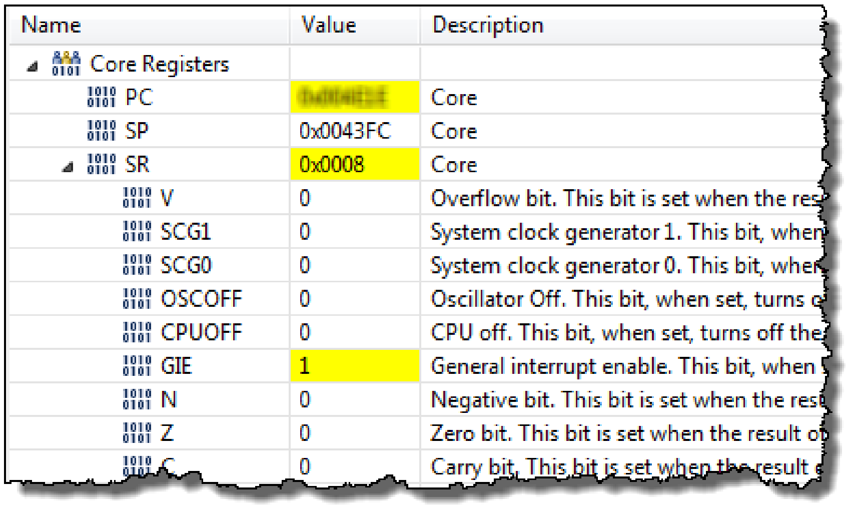
    You should notice that the GIE bit equals 0, since we haven't executed the line of code enabling interrupts globally, yet.



Name	Value	Description
Core Registers		
PC	0x004E1A	Core
SP	0x0043FC	Core
SR	0x0000	Core
V	0	Overflow bit. T
SCG1	0	System clock g
SCG0	0	System clock g
OSCOFF	0	Oscillator Off.
CPUOFF	0	CPU off. This b
GIE	0	General interru
N	0	Negative bit. T
Z	0	Zero bit. This b

17. **Single-step the processor (i.e. Step-Over) and watch GIE change.**

    Click the toolbar button or tap the F6 key. Either way, the *Registers* window should update:



Name	Value	Description
Core Registers		
PC	[redacted]	Core
SP	0x0043FC	Core
SR	0x0008	Core
V	0	Overflow bit. This bit is set when the res
SCG1	0	System clock generator 1. This bit, when
SCG0	0	System clock generator 0. This bit, whe
OSCOFF	0	Oscillator Off. This bit, when set, turns o
CPUOFF	0	CPU off. This bit, when set, turns off the
GIE	1	General interrupt enable. This bit, when
N	0	Negative bit. This bit is set when the res
Z	0	Zero bit. This bit is set when the result o
C	0	Carry bit. This bit is set when the result

Testing your Interrupt

With everything set up properly, let's try out our code.

18. **Click *Resume* (i.e. Run) … and nothing should happen.**

    In fact, if you *Suspend* (i.e. Halt) the processor, you should see that the program counter is sitting in the while{} loop, as expected.

19. **Press the appropriate pushbutton on your board.**

    Did that cause the program to stop at the breakpoint we set in the ISR?

    If you hit *Suspend* in the previous step, did you remember to hit *Resume* afterwards?

    (If it didn't stop, and you cannot figure out why, ask a neighbor/instructor for help.)

Add a Simple Interrupt Service Routine (ISR)

Thus far we have used the HWI_UNUSED_ISR. We will now add an ISR specifically for our push-button's GPIO interrupt.

20. Add the Port 1 ISR to the bottom of `main.c`.

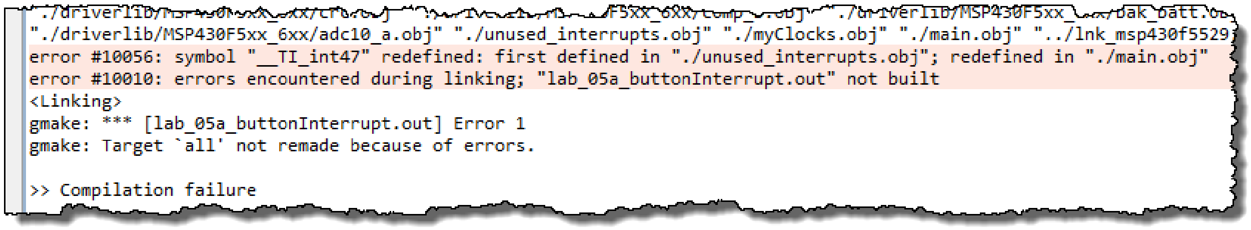
Here's a simple ISR routine that you can copy/paste into your code.

```
//*********************************************************************
// Interrupt Service Routines
//*********************************************************************
#pragma vector= ?????
__interrupt void pushbutton_ISR (void)
{
    // Toggle the LED on/off
    GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
}
```

Don't forget to fill in the ???? with your answer from question #11 from the worksheet (see page 5-41).

21. Build the program to test for any errors.

You should have gotten the error …



```
 "./driverlib/[illegible]F5xx_6xx/[illegible] "./driverlib/MSP430F5xx_[illegible]/bak_batt.[illegible]
"./driverlib/MSP430F5xx_6xx/adc10_a.obj" "./unused_interrupts.obj" "./myClocks.obj" "./main.obj" "../lnk_msp430f5529
error #10056: symbol "__TI_int47" redefined: first defined in "./unused_interrupts.obj"; redefined in "./main.obj"
error #10010: errors encountered during linking; "lab_05a_buttonInterrupt.out" not built
<Linking>
gmake: *** [lab_05a_buttonInterrupt.out] Error 1
gmake: Target `all' not remade because of errors.

>> Compilation failure
```

This error tells us that the linker cannot fit the PORT1_VECTOR into memory because the interrupt vector is defined twice. (INT47 on the 'F5529 and 'FR4133; INT39 on the 'FR5969)

We just created one of these vectors, where is the other one coming from?

____

Sidebar – Vector Error

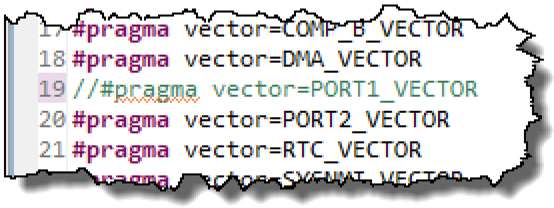
First, how did we recognize this error?

1. It says, *"errors encountered during linking"*. This tells us the complilation was fine, but there was a problem in linking.

2. Next, *"symbol "__TI_int47"" redefined"*. Oops, too many definitions for this symbol. It also tells us that this symbol was found in both `unused_interrupts.c` as well as `main.c`. (OK, it says that the offending files were `.obj`, but these were directly created from their `.c` source counterparts.

3. Finally, what's with the name, "__TI_int47"? Go back and look at the Interrupt Vector Location (sometimes it's also called Interrupt Priority) in the Interrupt Vector table. You can find this in the chapter discussion or the datasheet. Once you've done so, you should see the correlation with the PORT1_VECTOR.

22. Comment out the PORT1_VECTOR from `unused_interrupts.c`.



```
#pragma vector=COMP_B_VECTOR
#pragma vector=DMA_VECTOR
//#pragma vector=PORT1_VECTOR
#pragma vector=PORT2_VECTOR
#pragma vector=RTC_VECTOR
#pragma vector=SYSNMI_VECTOR
```

23. Try building it again

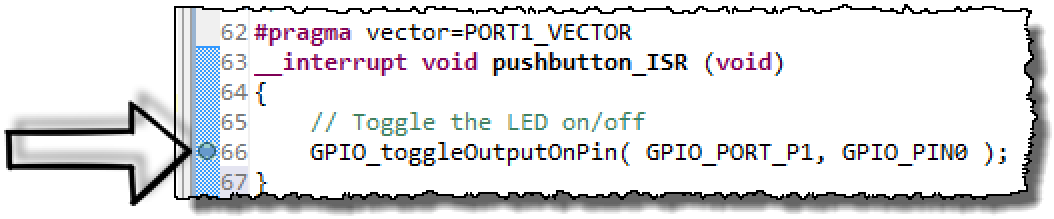
It should work this time… *our fingers are crossed for you.*

24. Launch the debugger.

25. Remove all breakpoints.

```
View → Breakpoints
Click the Remove All button
```

26. Set a breakpoint inside your new ISR.



```
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void)
{
    // Toggle the LED on/off
    GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
}
```

27. Run your code … once the code is running, push the button to generate an interrupt.

The processor should stop at your ISR (location shown above). Breakpoints like this can make it easier to see that we reached the interrupt. (A good debugging trick.)

28. *Resuming* once again, at this point inside the ISR should toggle-on the LED.

If it works, call out "Hooray!"

29. Push the button again.

Hmmm… did you get another interrupt? We didn't appear to.

We didn't see the light toggle-off – and we didn't stop at the breakpoint inside the ISR.

Some of you may have already known this was going to happen. If you're still unsure, go back to Step #0 from our worksheet (page 5-43). We discussed it there.

Upgrade Your Interrupt Service Routine (ISR)

If you hadn't already guessed what the problem was, we can deduce that since the IFG bit never got cleared, the CPU never realized that new interrupts were being applied.

For grouped interrupts, if we use the appropriate Interrupt Vector (IV) register, we can easily decipher the highest priority interrupt of the group; and, it clears the correct IFG bit for us.

30. Replace the code inside your ISR with the code that uses the P1IV register.

Once again, we have already created the code as part of the worksheet; refer to the Worksheet, Step 14 (page 5-43).

To make life easier, here's a copy of the original template from the worksheet. You may want to cut/paste this code, then tweak it with answers from your worksheet. *(Note: this is the code for the 'F5529 and 'FR5969. Remember that the 'FR4133 uses a different pin on Port 1.)*

```
//***********************************************************************
// Interrupt Service Routines
//***********************************************************************
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void) {
        switch(__even_in_range( ????, 0x10 )) {
           case 0x00: break;              // None
           case 0x02: break;              // Pin 0
           case 0x04:                     // Pin 1
                ???????????????????????;
                break;
           case 0x06: break;              // Pin 2
           case 0x08: break;              // Pin 3
           case 0x0A: break;              // Pin 4
           case 0x0C: break;              // Pin 5
           case 0x0E: break;              // Pin 6
           case 0x10: break;              // Pin 7
           default:   _never_executed();
       }
}
```

Hint: The syntax indentation often gets messed up when pasting code. If/when this occurs, the CCS editor provides a way to correct this using (<ctrl>-I).

`Select the 'ugly' code and press` Ctrl - I

31. Build the code.

If you correctly inserted the code and replaced all the questions marks, hopefully it built correctly the first time.

32. Launch the debugger. Run/Resume. Push the button. Verify the light toggles.

Run the program. Push the button and verify that the interrupt is taken every time you push the button. If the breakpoint in the ISR is still set, you should see the processor stop for each button press (and then you'll need to click *Resume*).

You're welcome to explore further by single-stepping thru code, using breakpoints, suspending (halting) the processor and exploring the various registers.

(Optional) Lab 5b – Can You Make a Watchdog Blink?

The goal of this lab is to blink the LED. Rather than using a _delay_cycles() function, we'll use a timer to tell us when to toggle the LED.

In Lab 4 we used the Watchdog timer as a … well, a watchdog timer. In all other exercises, thus far, we just turned it off with `WDT_A_hold()`.

In this lab exercise, we're going to use it as a standard timer (called 'interval' timer) to generate a periodic interrupt. In the interrupt service routine, we'll toggle the LED.

As we write the ISR code, you may notice that the Watchdog Interval Timer interrupt has a dedicated interrupt vector. (Whereas the GPIO Port interrupt had 8 grouped interrupts that shared one vector.)

Import and Explore the WDT_A Interval Timer Example

1. **Import the *wdt_a_ex2_intervalACLK* project from the MSP430 DriverLib examples.**

   We're going to "cheat" and use the example provided with MSP430ware to get the WDT_A timer up and running.

   As we discussed in Chapter 3, there are two ways we can import an example project:

   – Use the Project→Import CCS Projects (as we've done before)

   – Utilize the TI Resource Explorer (which is what we'll do again)

   a) **Open the TI Resource Explorer window, if it's not already open**

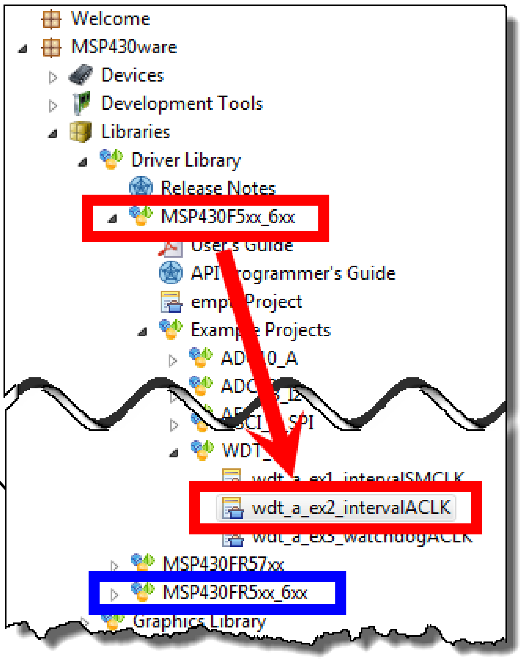
   ```
   View → Resource Explorer (Examples)
   ```

   b) **Locate the *wdt_a_ex2_intervalACLK* example for your processor.**

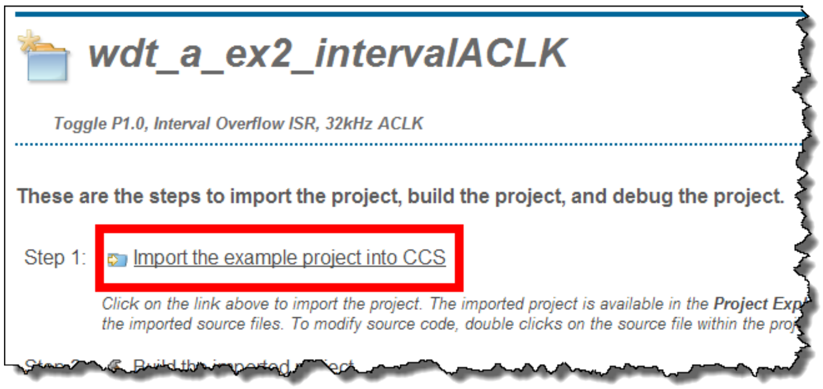
   Look for it as shown here under: Example Projects → WDT_A

If you're using the **FR5969**, follow the same path starting from the *MSP430**FR**5xx_6xx* heading

Likewise, pick the *MSP430**FR**2xx_4xx is you're using the* **FR4311**



c) Click the link to *"Import the example project into CCS"*.



wdt_a_ex2_intervalACLK

Toggle P1.0, Interval Overflow ISR, 32kHz ACLK

These are the steps to import the project, build the project, and debug the project.

Step 1: Import the example project into CCS

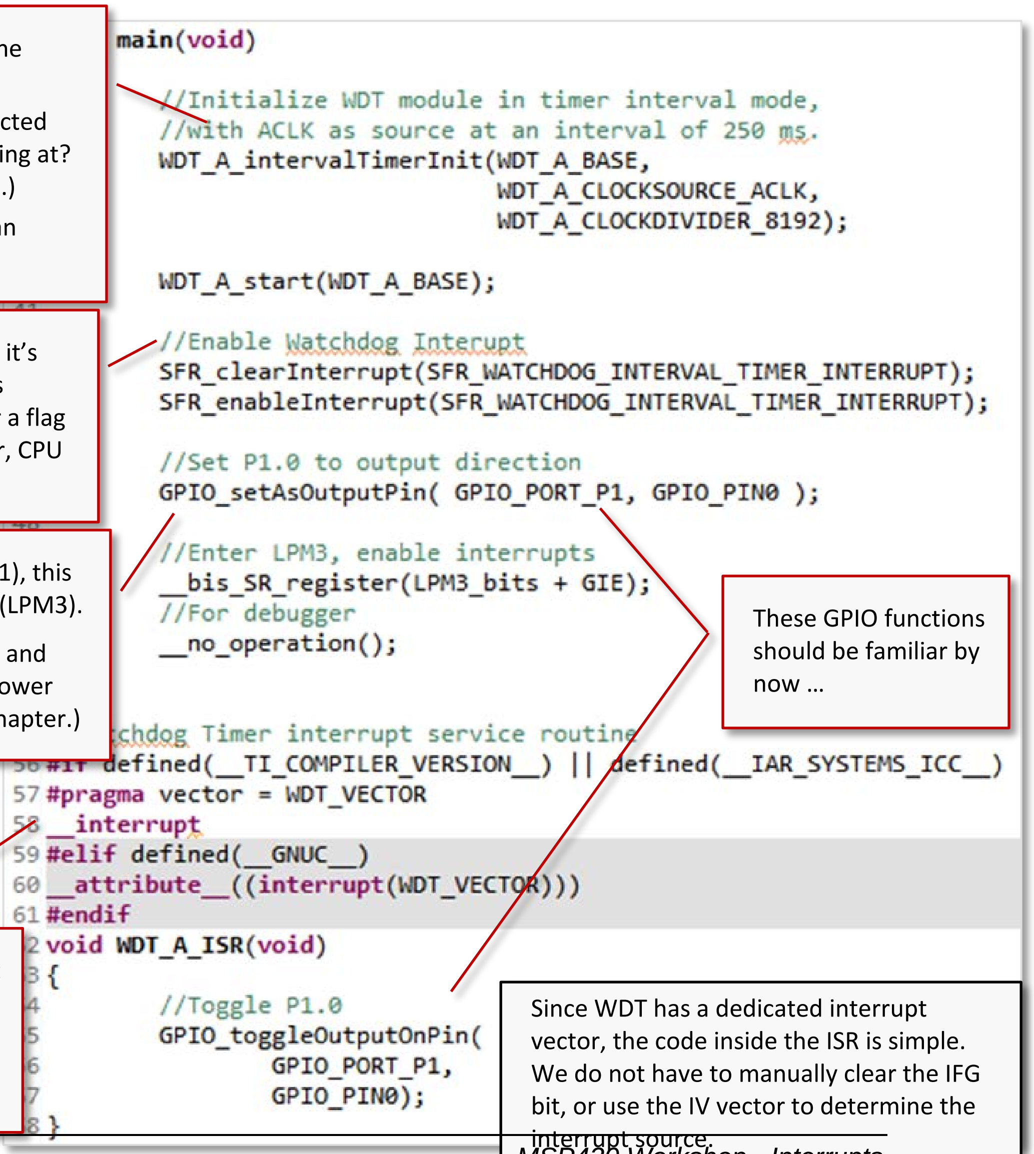
*Click on the link above to import the project. The imported project is available in the **Project Exp** the imported source files. To modify source code, double clicks on the source file within the proj*

Once imported you can close the TI Resource Explorer, if you want to get it out of the way.

d) Rename the imported project to: `lab_05b_wdtBlink`

While not required, this should make it easier to match the project to our lab files later on.

2. Open the `lab_05b_wdtBlink.c` file. Review the following points:



```
main(void)

    //Initialize WDT module in timer interval mode,
    //with ACLK as source at an interval of 250 ms.
    WDT_A_intervalTimerInit(WDT_A_BASE,
                            WDT_A_CLOCKSOURCE_ACLK,
                            WDT_A_CLOCKDIVIDER_8192);

    WDT_A_start(WDT_A_BASE);

    //Enable Watchdog Interrupt
    SFR_clearInterrupt(SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT);
    SFR_enableInterrupt(SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT);

    //Set P1.0 to output direction
    GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );

    //Enter LPM3, enable interrupts
    __bis_SR_register(LPM3_bits + GIE);
    //For debugger
    __no_operation();

chdog Timer interrupt service routine
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector = WDT_VECTOR
__interrupt
#elif defined(__GNUC__)
__attribute__((interrupt(WDT_VECTOR)))
#endif
void WDT_A_ISR(void)
{
    //Toggle P1.0
    GPIO_toggleOutputOnPin(
            GPIO_PORT_P1,
            GPIO_PIN0);
}
```

Notice the DriverLib function that sets up the WDT_A for interval timing.

You can choose which clock to use; we selected ACLK. By the way, what speed is ACLK running at? (This example uses ACLK at the default rate.)

As described, dividing ACLK/8192 gives us an interval of ¼ second.

The WDT_A is a system (SYS) interrupt, so it's IFG and IE bits are in the Special Functions Register. It's always good practice to clear a flag before enabling the interrupt. (Remember, CPU won't be interrupted until we set GIE.)

Along with enabling interrupts globally (GIE=1), this example puts the CPU into low power mode (LPM3).

When the interrupt occurs, the CPU wake up and handles it, then goes back into LPM3. (Low Power modes will be discussed further in a future chapter.)

These GPIO functions should be familiar by now …

They got a little bit fancy with the interrupt vector syntax. This code has been designed to work with 3 different compilers:

TI, IAR, and GNU C compiler.

Since WDT has a dedicated interrupt vector, the code inside the ISR is simple. We do not have to manually clear the IFG bit, or use the IV vector to determine the interrupt source.

Run the code

3. Build and run the example.

You should see the LED blinking…

Change the LED blink rate

4. Terminate the debug session.

5. Modify the example to blink the LED at about 1 second intervals.

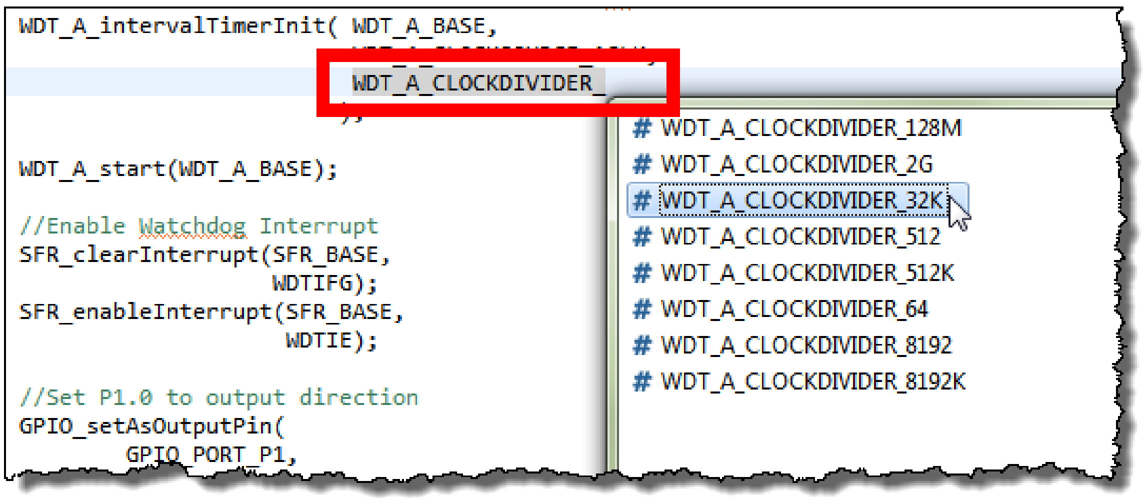
Tip: If you want help with selecting and typing function arguments, you can you the autocomplete feature of CCS. Just type part of the test, such as:

```
WDT_A_CLOCKDIVER_
```

and then hit:

```
Control-TAB
```

and a popup box appears providing you with choices – select the one you want. In this case, we suggest you divide by 32K.



```
WDT_A_intervalTimerInit( WDT_A_BASE,
                         WDT_A_CLOCKDIVIDER_
                         );

WDT_A_start(WDT_A_BASE);

//Enable Watchdog Interrupt
SFR_clearInterrupt(SFR_BASE,
                   WDTIFG);
SFR_enableInterrupt(SFR_BASE,
                    WDTIE);

//Set P1.0 to output direction
GPIO_setAsOutputPin(
        GPIO_PORT_P1,
```

```
# WDT_A_CLOCKDIVIDER_128M
# WDT_A_CLOCKDIVIDER_2G
# WDT_A_CLOCKDIVIDER_32K
# WDT_A_CLOCKDIVIDER_512
# WDT_A_CLOCKDIVIDER_512K
# WDT_A_CLOCKDIVIDER_64
# WDT_A_CLOCKDIVIDER_8192
# WDT_A_CLOCKDIVIDER_8192K
```

6. Build and run the example again.

If you want, you can experiment with other clock divider rates to see their affect on the LED's blink rate.

Appendix

Lab 05 Worksheet (1)

General Interrupt Questions

1. When your program is not in an interrupt service routine, what code is it usually executing? And, what 'name' do we give this code?

   main functions while{} loop. We often call this 'background' processing.

2. Why keep ISR's short (i.e. not do a lot of processing in them)?

   We don't want to block other interrupts. The other option is nesting interrupts, but this is INEFFICIENT. Do interrupt follow-up processing in while{} loop … or use TI-RTOS kernel.

3. What causes the MSP430 to exit a Low Power Mode (LPMx)?

   Interrupts

4. Why are *interrupts* generally preferred over *polling*?

   They are a lot more efficient. Polling ties up the CPU – even worse it consumes power waiting for an event to happen.

Lab 05 Worksheet (2)

Interrupt Flow

5. Name 3 more sources of interrupts?

   TIMER_A

   GPIO

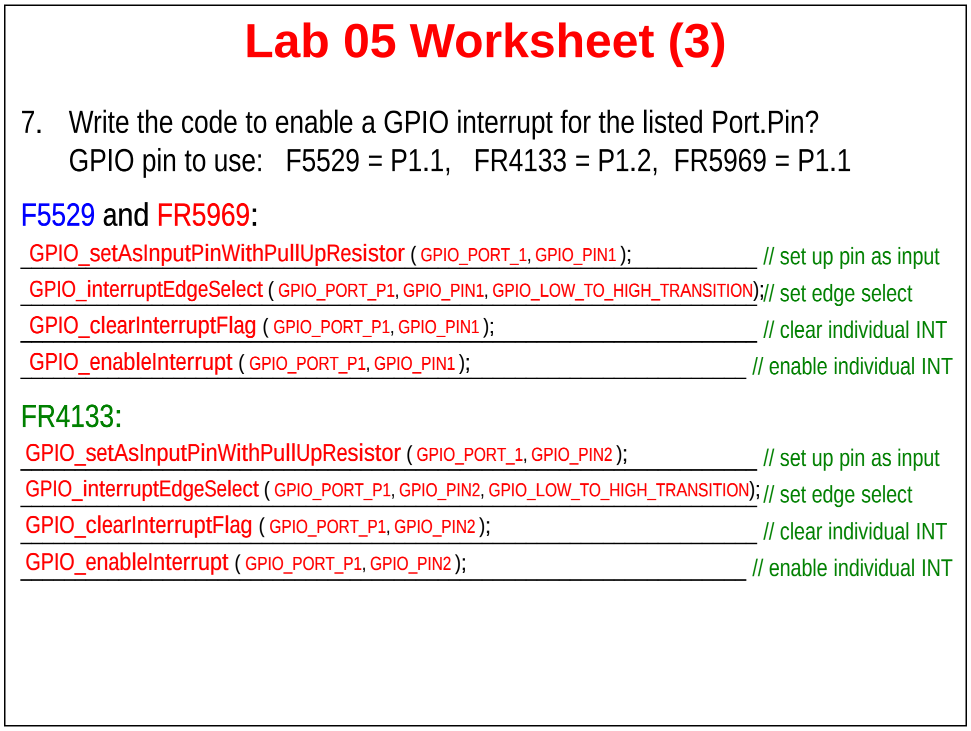
   Watchdog Interval Timer

   Analog Converter … and many more

6. What signifies that an interrupt has occurred?

   A **flag** bit is set

   What's the acronym for these types of 'bits" **IFG**



Lab 05 Worksheet (3)

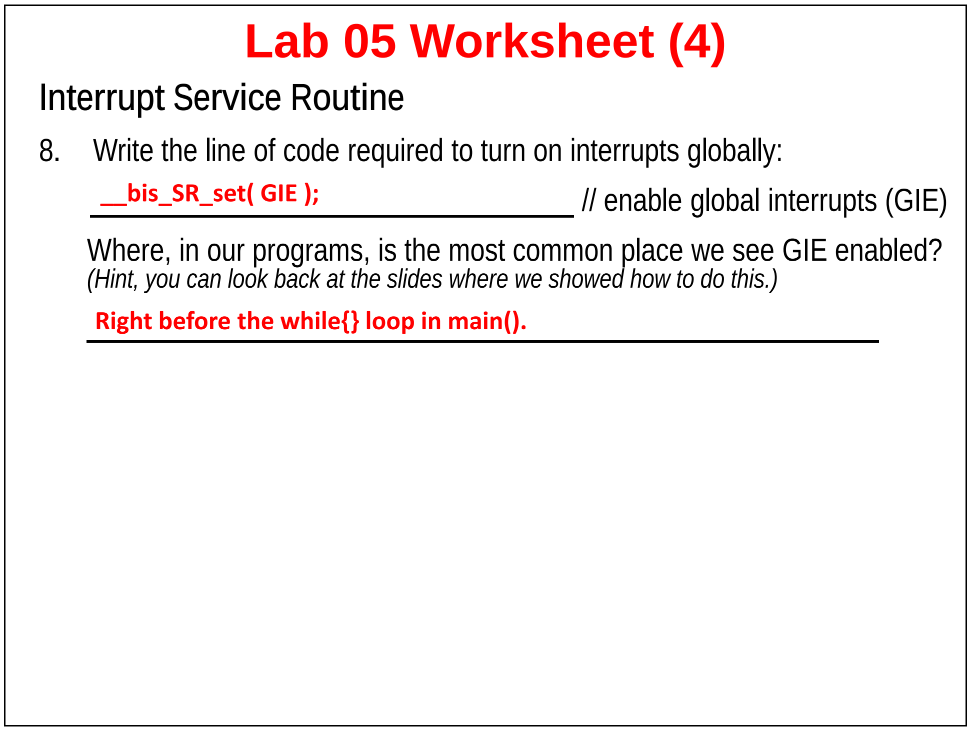
7. Write the code to enable a GPIO interrupt for the listed Port.Pin?
   GPIO pin to use: F5529 = P1.1, FR4133 = P1.2, FR5969 = P1.1

F5529 and FR5969:

GPIO_setAsInputPinWithPullUpResistor (GPIO_PORT_1, GPIO_PIN1); // set up pin as input
GPIO_interruptEdgeSelect (GPIO_PORT_P1, GPIO_PIN1, GPIO_LOW_TO_HIGH_TRANSITION); // set edge select
GPIO_clearInterruptFlag (GPIO_PORT_P1, GPIO_PIN1); // clear individual INT
GPIO_enableInterrupt (GPIO_PORT_P1, GPIO_PIN1); // enable individual INT

FR4133:

GPIO_setAsInputPinWithPullUpResistor (GPIO_PORT_1, GPIO_PIN2); // set up pin as input
GPIO_interruptEdgeSelect (GPIO_PORT_P1, GPIO_PIN2, GPIO_LOW_TO_HIGH_TRANSITION); // set edge select
GPIO_clearInterruptFlag (GPIO_PORT_P1, GPIO_PIN2); // clear individual INT
GPIO_enableInterrupt (GPIO_PORT_P1, GPIO_PIN2); // enable individual INT



Lab 05 Worksheet (4)

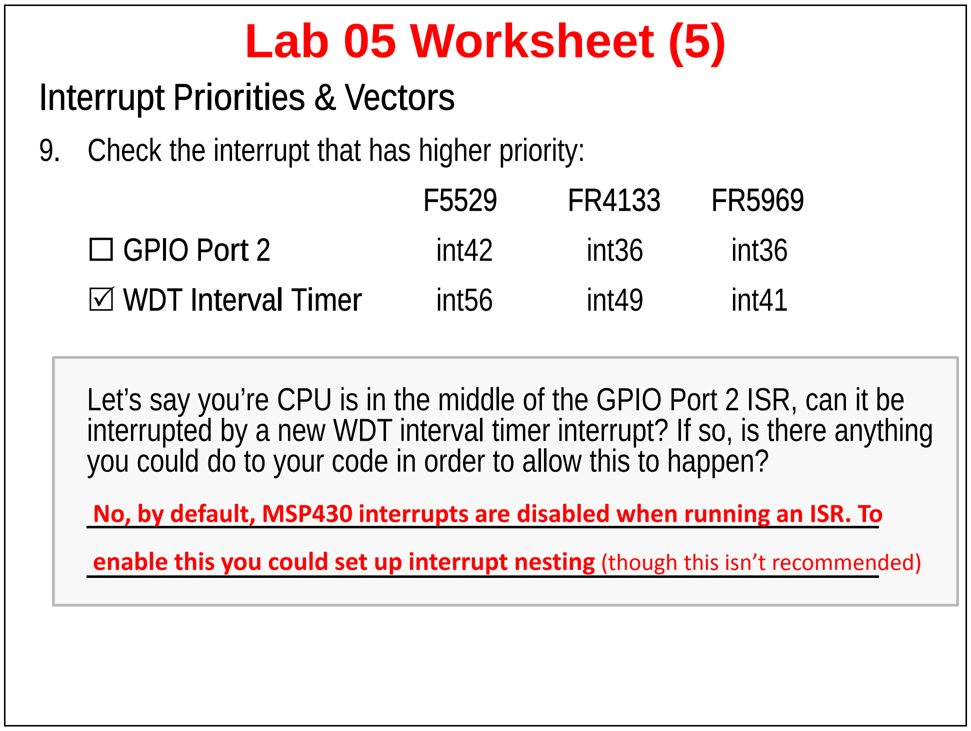
Interrupt Service Routine

8. Write the line of code required to turn on interrupts globally:

   __bis_SR_set(GIE); // enable global interrupts (GIE)

   Where, in our programs, is the most common place we see GIE enabled?
   (Hint, you can look back at the slides where we showed how to do this.)

   Right before the while{} loop in main().



Lab 05 Worksheet (5)

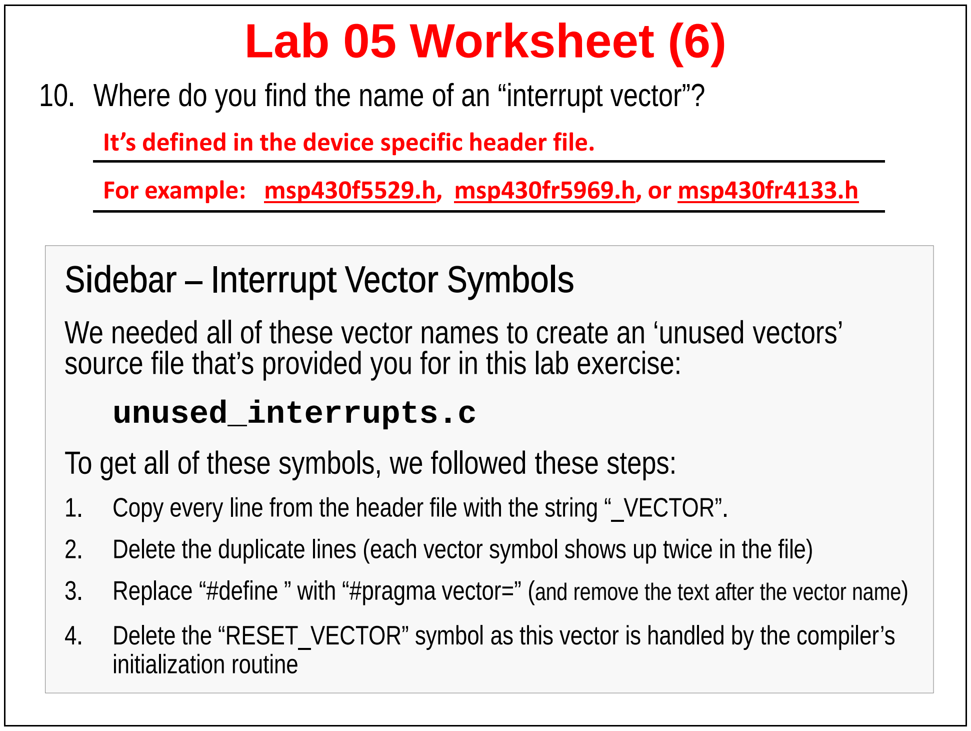
Interrupt Priorities & Vectors

9. Check the interrupt that has higher priority:

	F5529	FR4133	FR5969
☐ GPIO Port 2	int42	int36	int36
☑ WDT Interval Timer	int56	int49	int41

Let’s say you’re CPU is in the middle of the GPIO Port 2 ISR, can it be interrupted by a new WDT interval timer interrupt? If so, is there anything you could do to your code in order to allow this to happen?

No, by default, MSP430 interrupts are disabled when running an ISR. To enable this you could set up interrupt nesting (though this isn’t recommended)



Lab 05 Worksheet (6)

10. Where do you find the name of an “interrupt vector”?

It’s defined in the device specific header file.

For example: msp430f5529.h, msp430fr5969.h, or msp430fr4133.h

Sidebar – Interrupt Vector Symbols

We needed all of these vector names to create an ‘unused vectors’ source file that’s provided you for in this lab exercise:

`unused_interrupts.c`

To get all of these symbols, we followed these steps:

1. Copy every line from the header file with the string “_VECTOR”.
2. Delete the duplicate lines (each vector symbol shows up twice in the file)
3. Replace “#define ” with “#pragma vector=” (and remove the text after the vector name)
4. Delete the “RESET_VECTOR” symbol as this vector is handled by the compiler’s initialization routine

Lab 05 Worksheet (7)

11. How do you write the code to set the interrupt vector?

```
// Sets ISR address in the vector for Port 1
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void)
{
    // Toggle the LED on/off
    GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
}
```

What is wrong with this GPIO port ISR?

GPIO ports are grouped interrupts. It's better to read the P1IV register

so you can handle multiple pin interrupts using switch/case statement

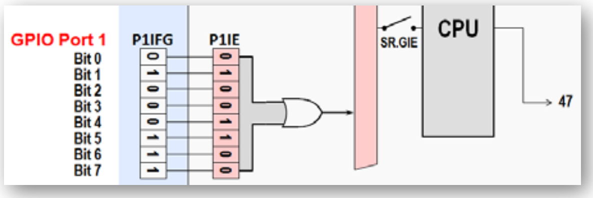
Lab 05 Worksheet (8)

12. How do you pass a value into (or out from) and interrupt service routine (ISR)?

Interrupts cannot pass arguments, we need to use global variables

ISR's for Group Interrupts

As we learned earlier, most MSP430 interrupts are grouped. For example, the GPIO port interrupts are all grouped together.



13. For dedicated interrupts (such as WDT interval timer) the CPU clears the IFG flag when responding to the interrupt. How does an IFG bit get cleared for group interrupts?

Either manually; or when you read the IV register (such as P1IV).

Lab 05 Worksheet (9)

14. Creating ISR's for grouped interrupts is as easy as following a 'template'. Toggle P1.0 when button is pressed. F5529/FR5969 uses P1.1;

```
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void) {
   switch(__even_in_range( ______P1IV_____, 0x10 )) {

// F5529 and FR5969 use P1.1 for button:
      case 0x02: break;        // Pin 0
      case 0x04:               // Pin 1
                 GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
                 break;
      case 0x06: break;        // Pin 2

// FR4311 uses P1.2 for button:
      case 0x02: break;        // Pin 0
      case 0x04: break;        // Pin 1
      case 0x06:               // Pin 2
                 GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
                 break;
```