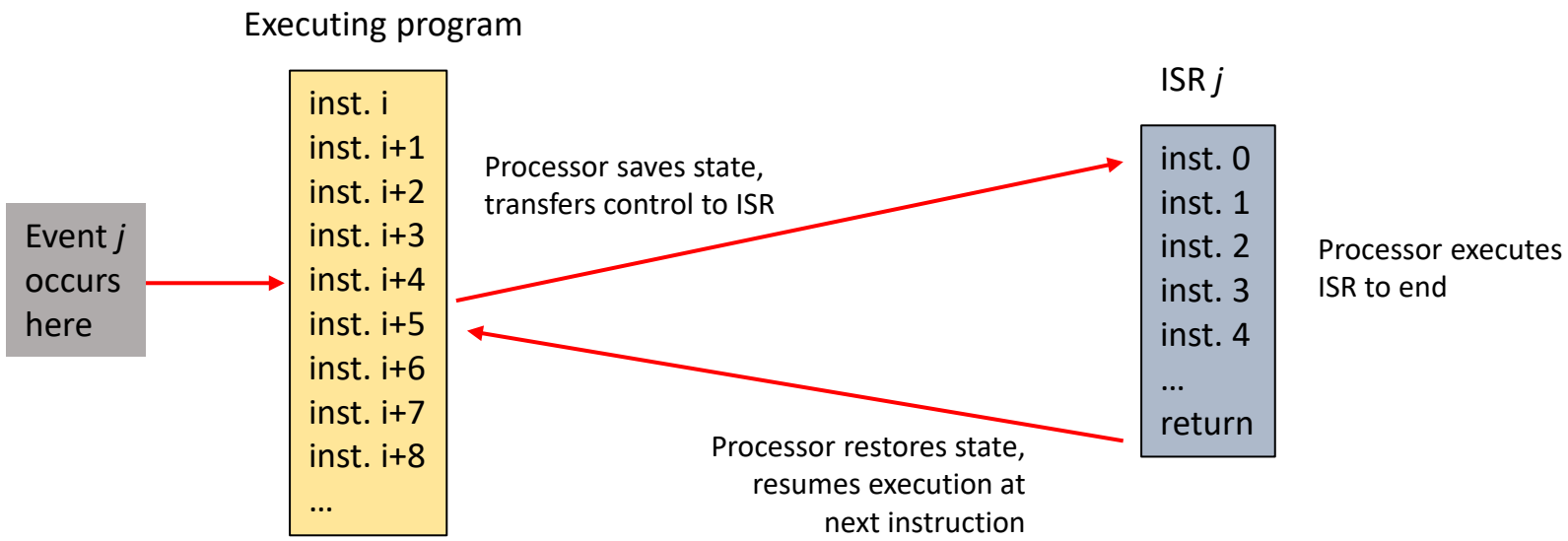
Interrupt Controller Driver

ECEN 330

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

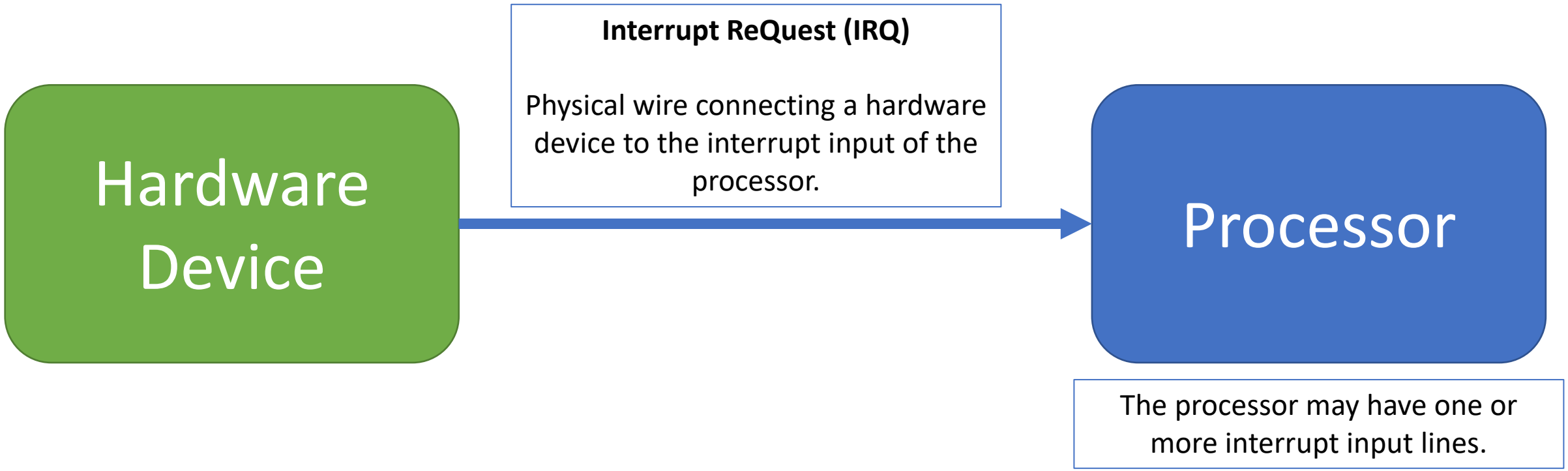
Interrupts

- A signal to the processor that an event occurred requiring a response
- The processor responds by:
  - Saving the state of the code that was running
  - Transferring control to a function written specifically to deal with that event



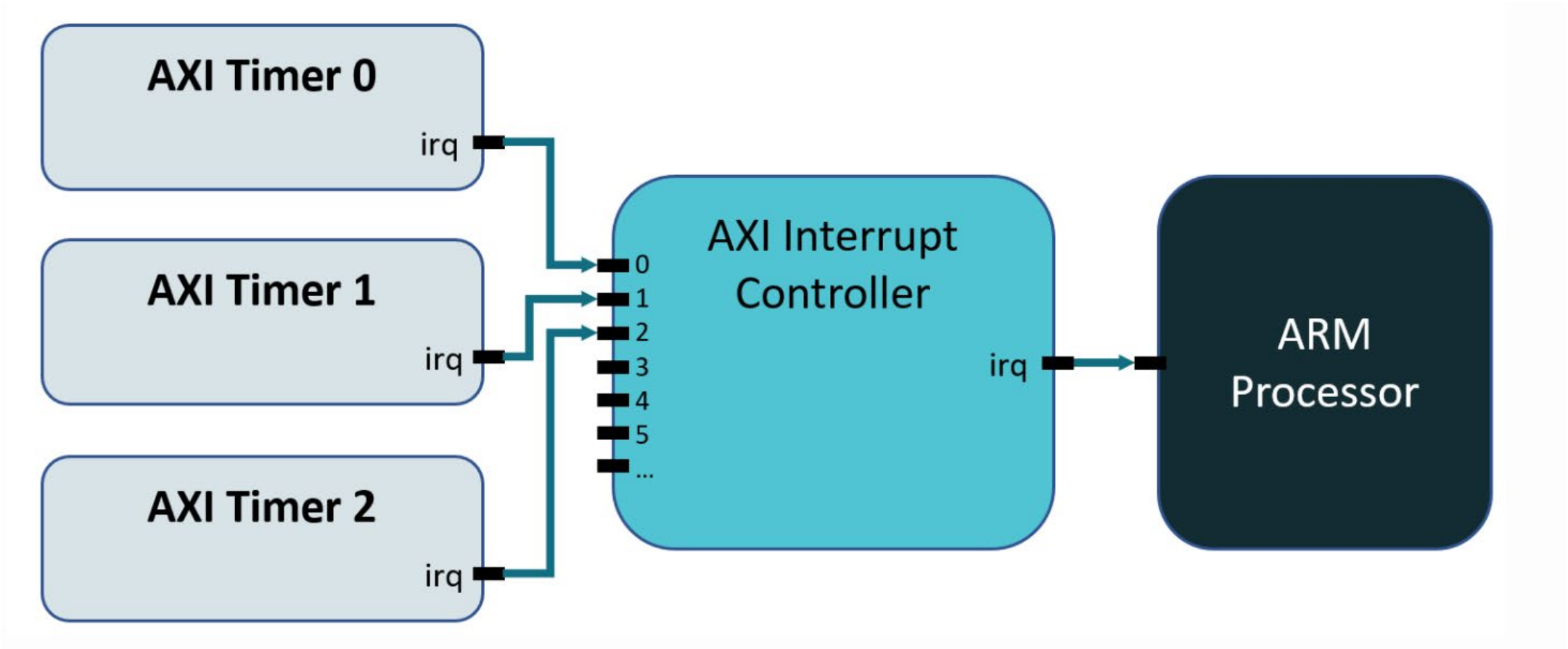
Interrupt Hardware

What does the hardware look like?



What happens if there are more devices than CPU interrupt inputs?

Interrupt Controller

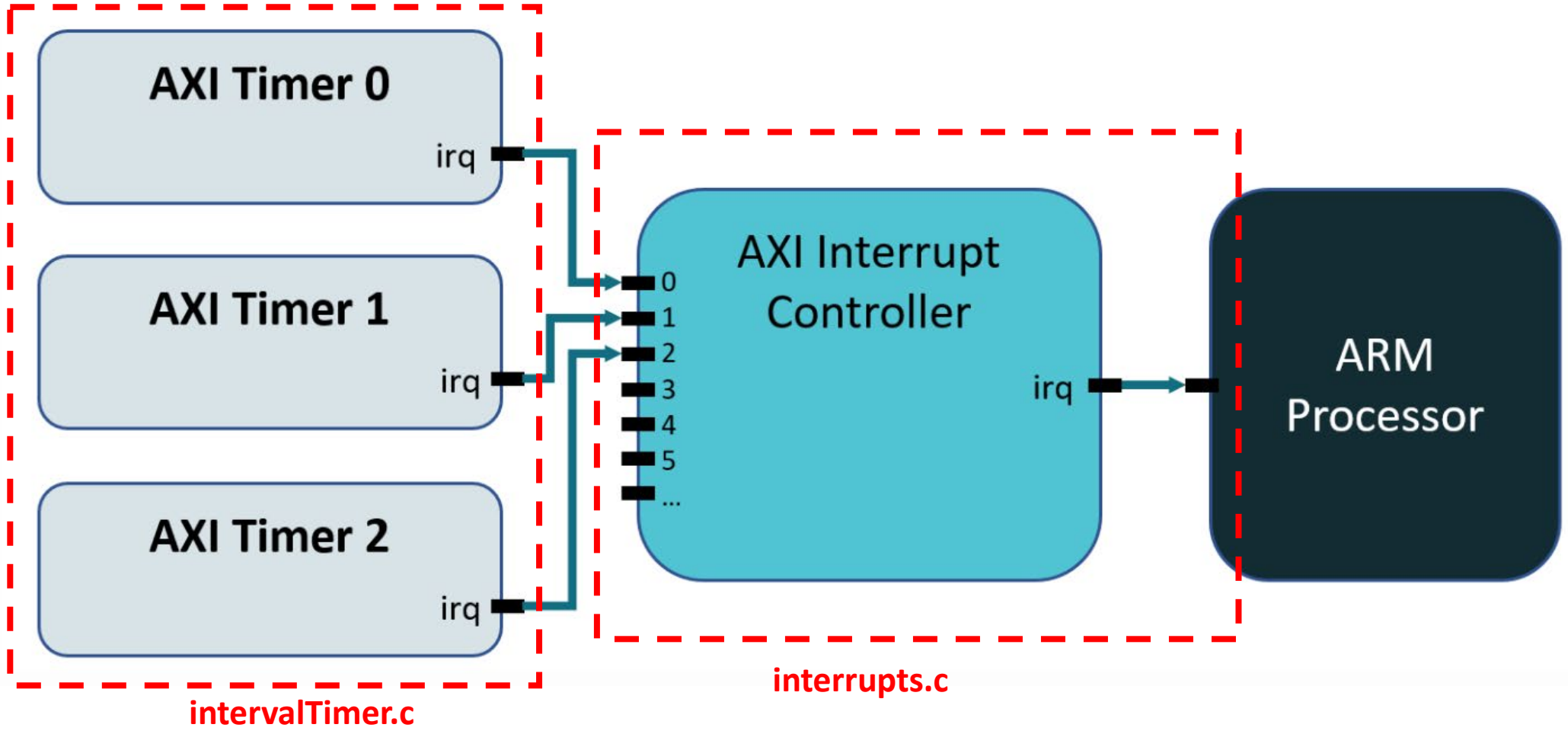


When a hardware device sends an IRQ to the interrupt controller,
the interrupt controller sends an IRQ to the processor.
(assuming appropriate things are enabled)

Interrupt Controller

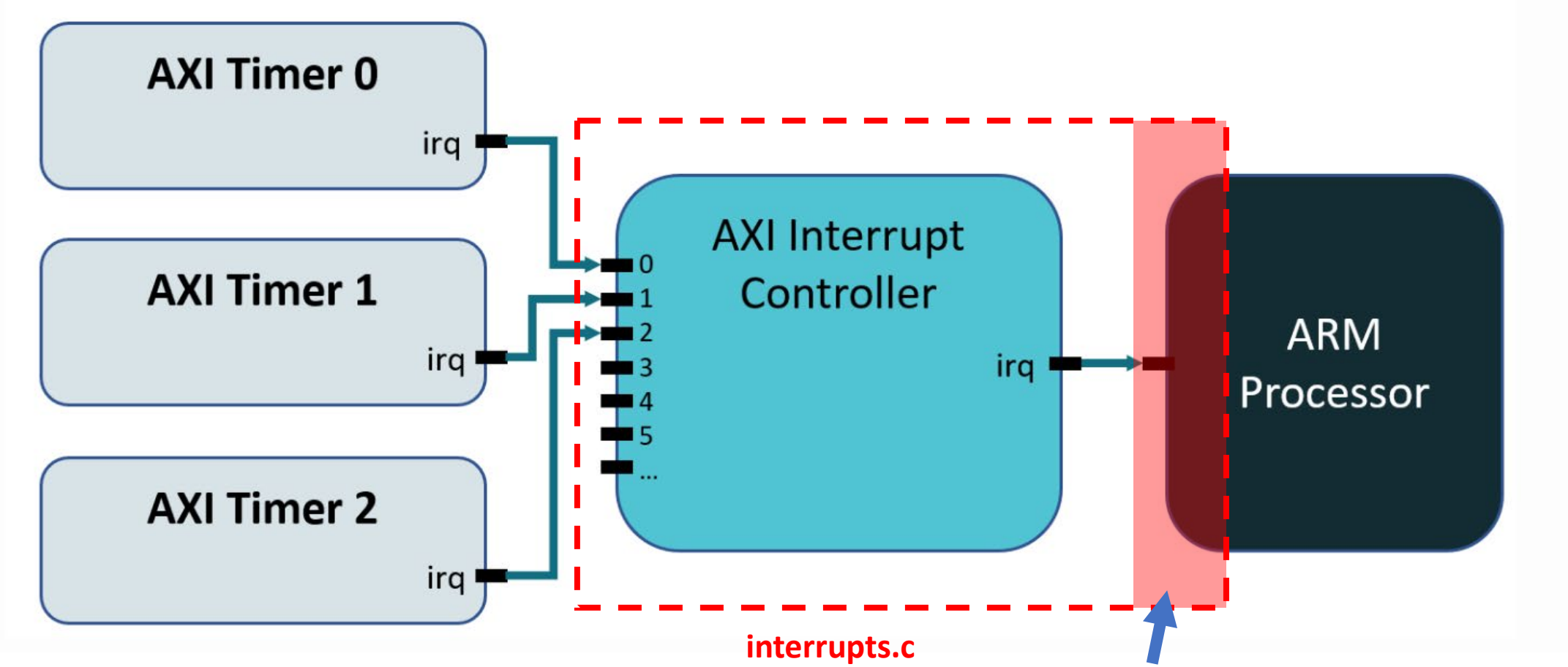
In the last lab we wrote a driver to control the AXI Timer hardware.

In this lab you will write a driver to control the AXI Interrupt Controller



Interrupt Controller

How will we "talk to" the Interrupt Controller? **Read/Write Registers!**



We will also call some **`armInterrupts*()`** functions to configure the interrupt inputs on the processor
(This will only need to be done once in intc_init())

Table 2-4: **Register Address Mapping**

Address Offset	Register Name	Description
00h	ISR	Interrupt Status Register (ISR)
04h	IPR	Interrupt Pending Register (IPR)
08h	IER	Interrupt Enable Register (IER)
0Ch	IAR	Interrupt Acknowledge Register (IAR)
10h	SIE	Set Interrupt Enables (SIE)
14h	CIE	Clear Interrupt Enables (CIE)
18h	IVR	Interrupt Vector Register (IVR)
1Ch	MER	Master Enable Register (MER)
20h	IMR	Interrupt Mode Register (IMR)
24h	ILR	Interrupt Level Register (ILR)
100h to 17Ch	IVAR	Interrupt Vector Address Register (IVAR)
200h to 2FCh	IVEAR	Interrupt Vector Extended Address Register (IVEAR)

Why Write a Driver?

If we were creating an **application** (ie. a game) we could:

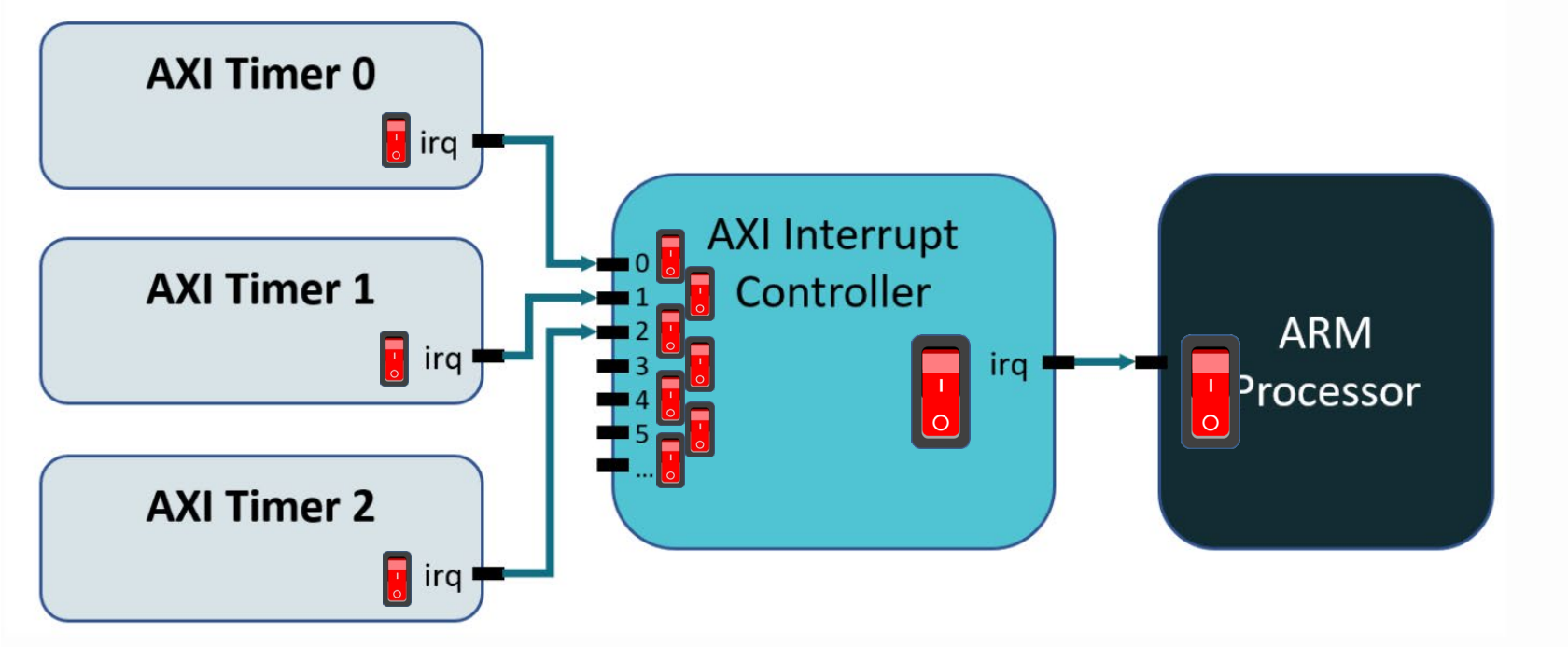
- Read/Write GPIO registers directly in our application
- Read/Write Timer registers directly in our application
- Read/Write Interrupt Controller registers directly in our applications.

What's wrong with that approach?

…its good to keep this in mind as we talk about how we are designing our interrupt controller driver this lab. It's designed to serve the needs of various applications…

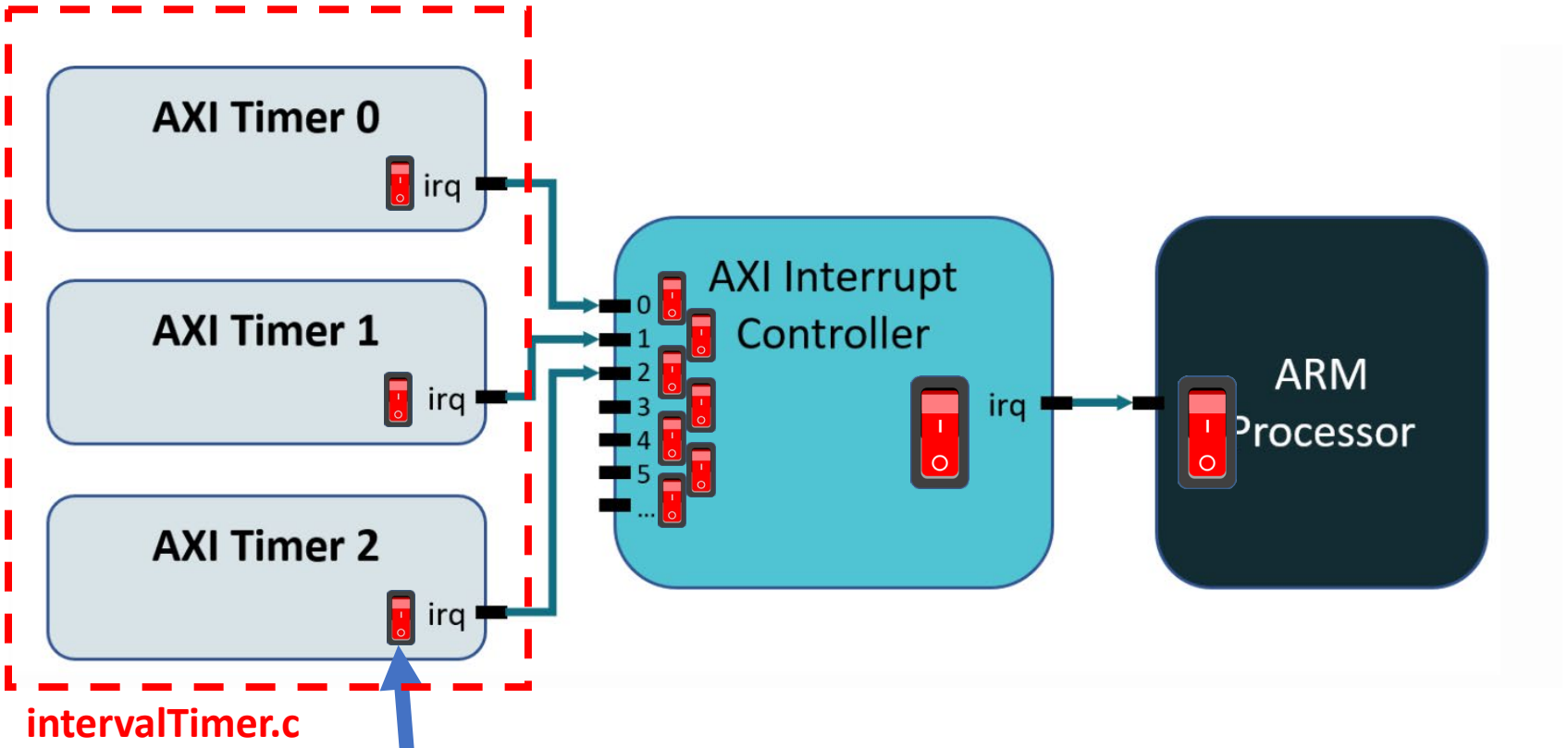
Setting Up the Interrupt Controller

For interrupts to function, they need to be enabled:



Setting Up the Interrupt Controller

For interrupts to function, they need to be enabled:



You already did this in the last lab: `intervalTimer_enableInterrupt()`

Setting Up the Interrupt Controller

For interrupts to function, they need to be enabled:

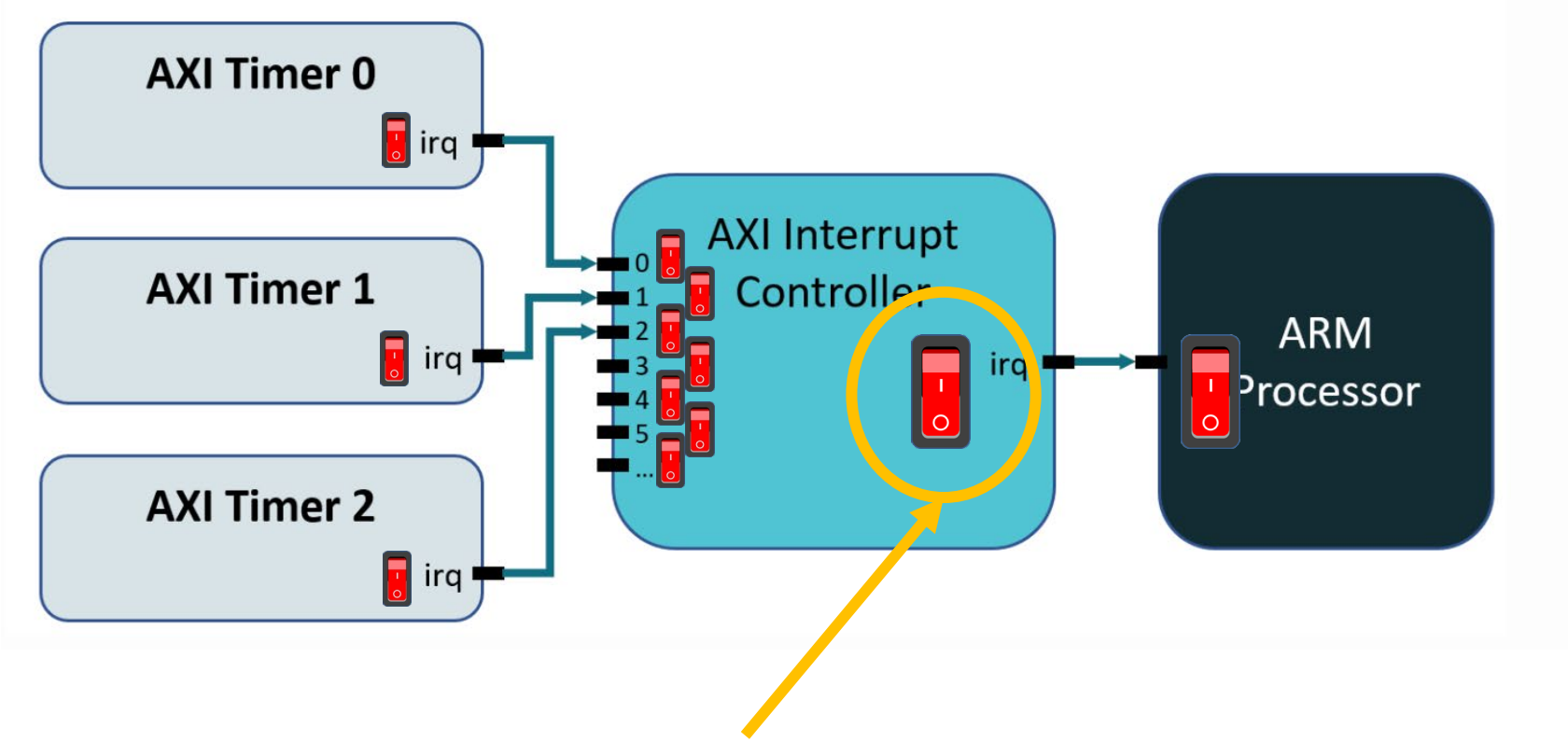


Table 2-4: **Register Address Mapping**

Address Offset	Register Name	Description
00h	ISR	Interrupt Status Register (ISR)
04h	IPR	Interrupt Pending Register (IPR)
08h	IER	Interrupt Enable Register (IER)
0Ch	IAR	Interrupt Acknowledge Register (IAR)
10h	SIE	Set Interrupt Enables (SIE)
14h	CIE	Clear Interrupt Enables (CIE)
18h	IVR	Interrupt Vector Register (IVR)
1Ch	MER	Master Enable Register (MER)
20h	IMR	Interrupt Mode Register (IMR)
24h	ILR	Interrupt Level Register (ILR)
100h to 17Ch	IVAR	Interrupt Vector Address Register (IVAR)
200h to 2FCh	IVEAR	Interrupt Vector Extended Address Register (IVEAR)

Master Enable Register (MER)

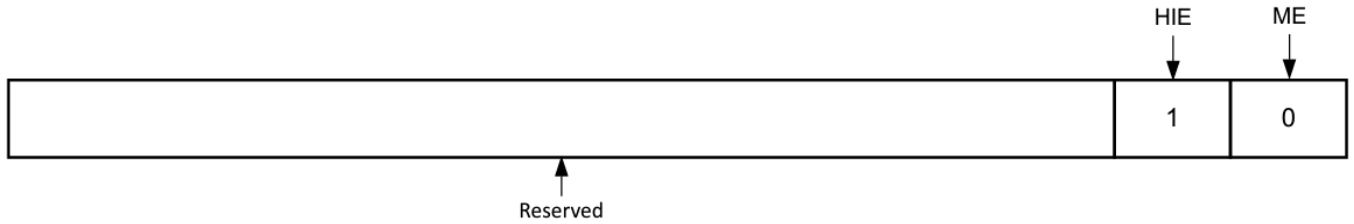


Figure 2-8: **Master Enable Register (MER)**

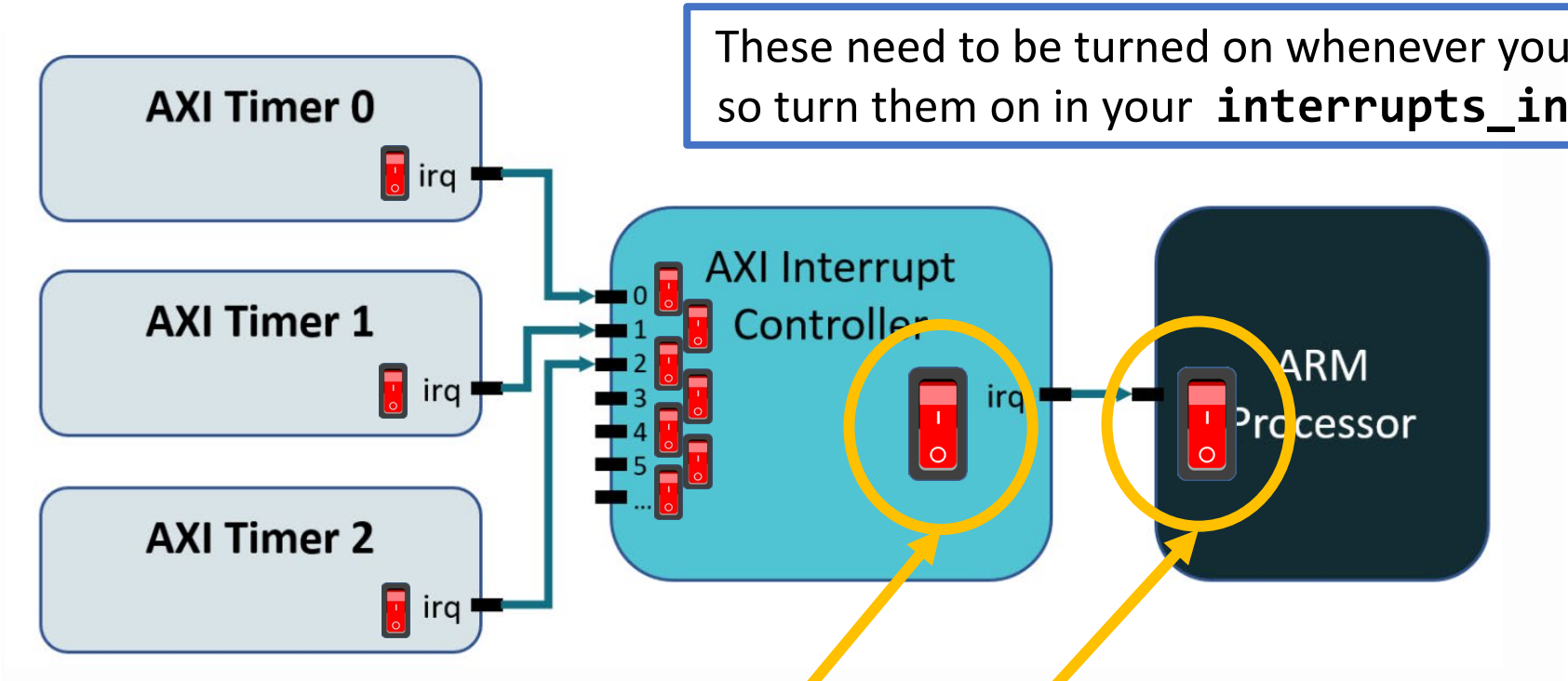
Table 2-12: **Master Enable Register Bit Definitions**

Bits	Name	Reset Value	Access	Description
(w(1)-1):2	Reserved	0x0	N/A	Reserved
1	HIE	0	Read / Write	Hardware Interrupt Enable 0 = Read - Generating HW interrupts from SW enabled Write - No effect 1 = Read - HW interrupts enabled Write - Enable HW interrupts
0	ME	0	Read / Write	Master IRQ Enable 0 = Irq disabled - All interrupts disabled 1 = Irq enabled - All interrupts can be enabled

- You need to set both of these bits

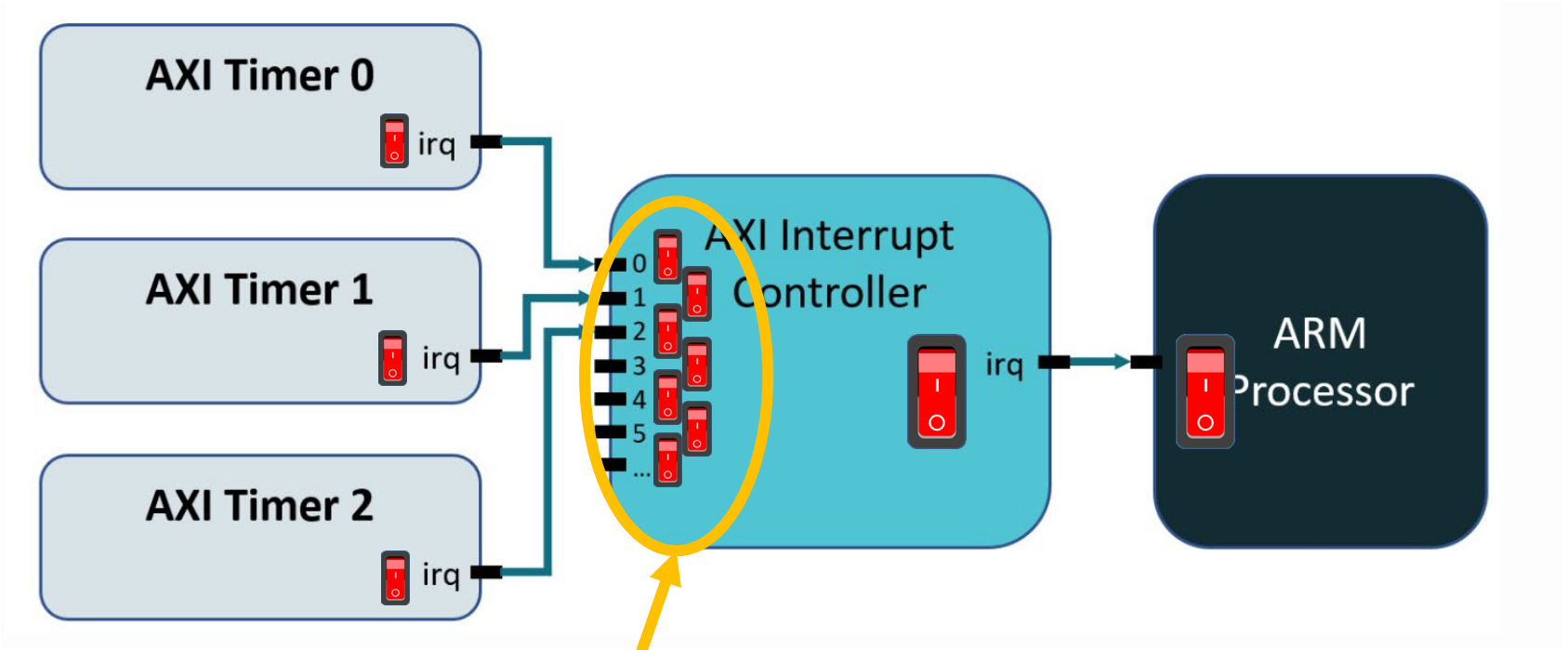
Setting Up the Interrupt Controller

For interrupts to function, they need to be enabled:



Setting Up the Interrupt Controller

For interrupts to function, they need to be enabled:




```
interrupts_irq_enable()
interrupts_irq_disable()
```

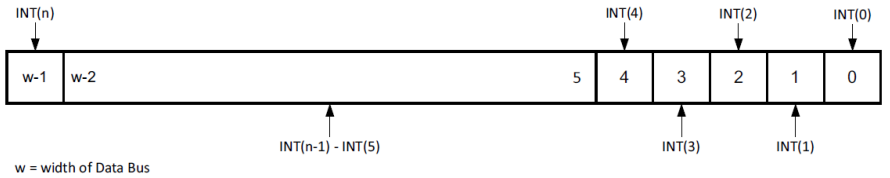
These are enabled on a “per application” basis
(ie enable them if your app needs them)
So they will be turned on/off via API functions

Table 2-4: **Register Address Mapping**

Address Offset	Register Name	Description
00h	ISR	Interrupt Status Register (ISR)
04h	IPR	Interrupt Pending Register (IPR)
08h	IER	Interrupt Enable Register (IER)
0Ch	IAR	Interrupt Acknowledge Register (IAR)
10h	SIE	Set Interrupt Enables (SIE)
14h	CIE	Clear Interrupt Enables (CIE)
18h	IVR	Interrupt Vector Register (IVR)
1Ch	MER	Master Enable Register (MER)
20h	IMR	Interrupt Mode Register (IMR)
24h	ILR	Interrupt Level Register (ILR)
100h to 17Ch	IVAR	Interrupt Vector Address Register (IVAR)
200h to 2FCh	IVEAR	Interrupt Vector Extended Address Register (IVEAR)

Interrupt Enable Register (IER)

This is a read-write register. Writing a 1 to a bit in this register enables, or unmasks, the corresponding ISR bit, allowing it to affect the `irq` output. An IER bit set to 0 does not inhibit an interrupt condition from being captured, just passing it to the processor. Writing a 0 to a bit disables, or masks, the generation of interrupt output for the corresponding interrupt.



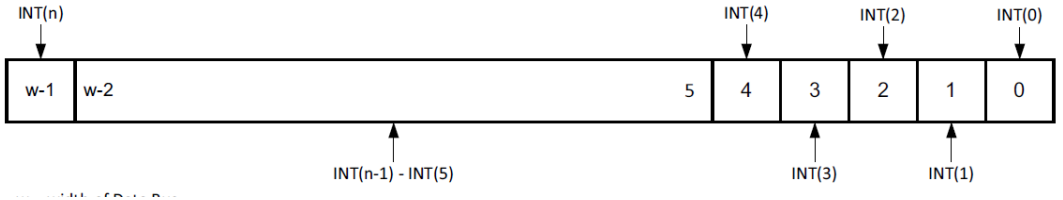
Set Interrupt Enables (SIE)

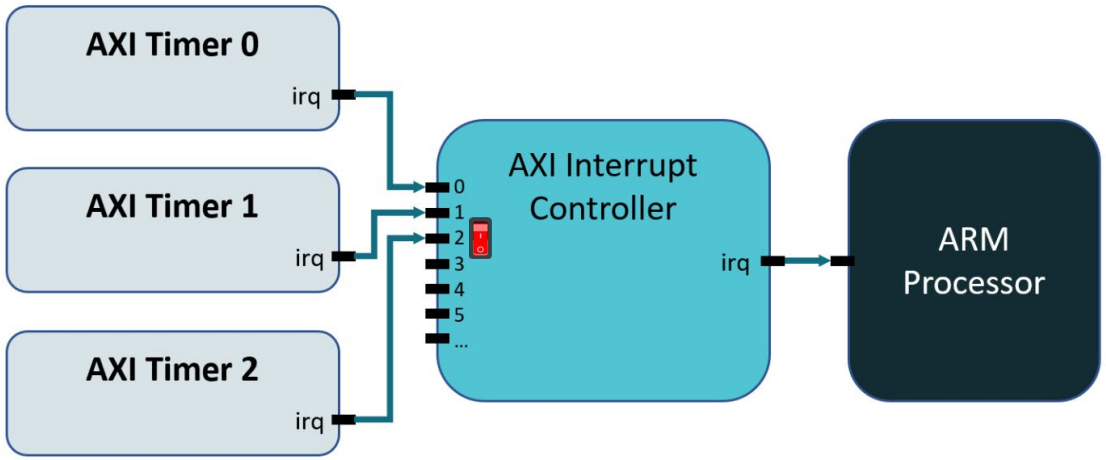
SIE is a register used to set the IER bits in a single atomic operation, rather than using a read-modify-write sequence. Writing a 1 to a bit location in SIE sets the corresponding bit in the IER. Writing 0 does nothing, as does writing 1 to a bit location that corresponds to a non-existing interrupt.

Clear Interrupt Enables (CIE)

CIE is a register used to clear IER bits in a single atomic operation, rather than using a read-modify-write sequence. Writing a 1 to a bit location in CIE clears the corresponding bit in the IER. Writing 0 does nothing, as does writing 1 to a bit location that corresponds to a non-existing interrupt.

Set Interrupt Enables (SIE)





Let's suppose the application wants to turn on input #2 (corresponds to AXI Timer 2 in the hardware design)

```
interrupts_irq_enable(2);
```

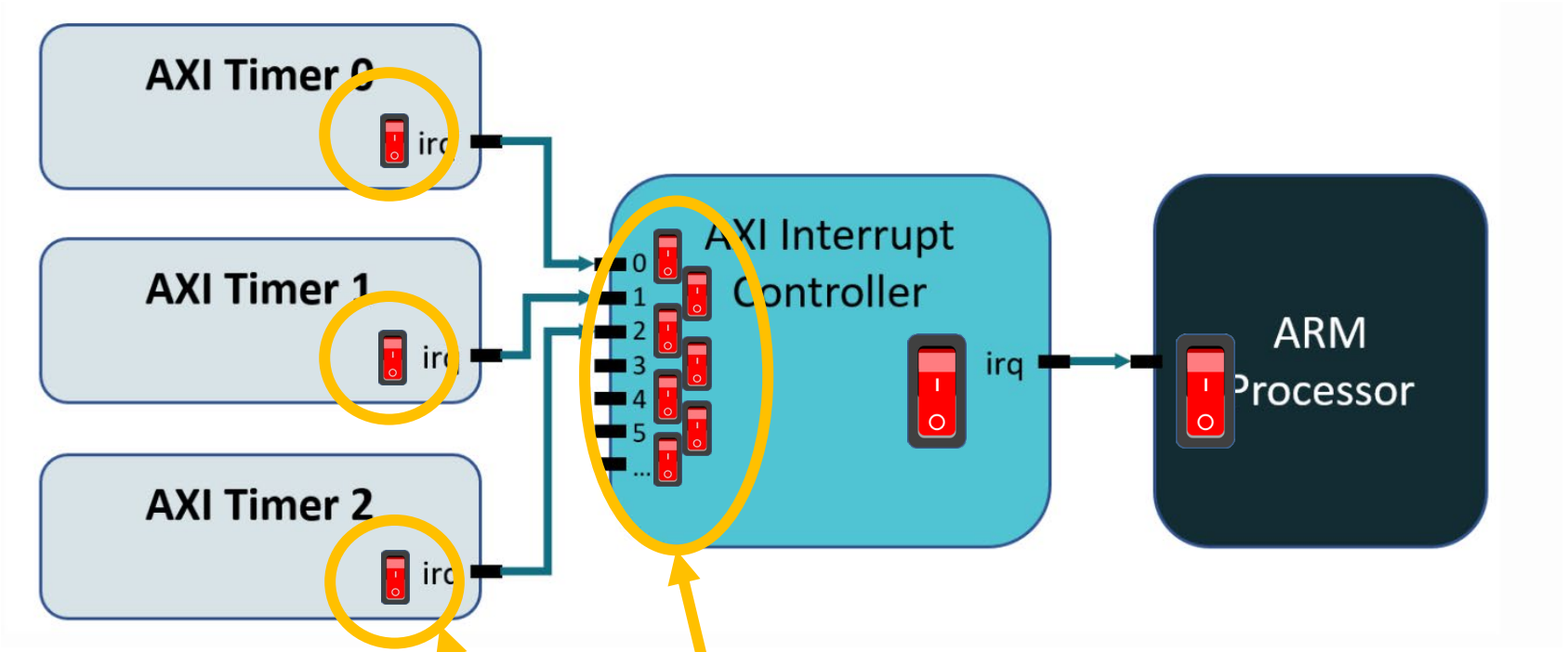
What value should you write to the SIE?

Now the general case…

```
interrupts_irq_enable(uint8_t irq) {


}
```

Setting Up the Interrupt Controller

For interrupts to function, they need to be enabled:



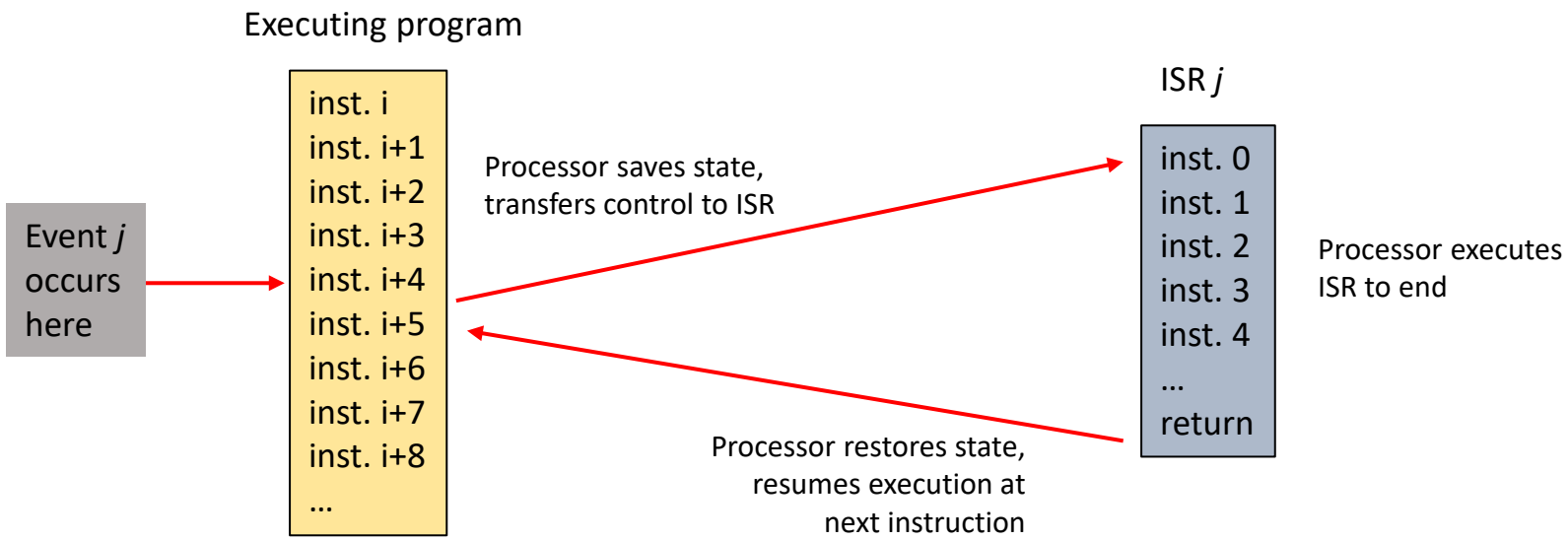
`intervalTimer_enableInterrupt()`

`interrupts_irq_enable()` – Use IER or SIE register
`interrupts_irq_disable()` – Use IER or CIE register

Inside **`interrupts_init()`**, it's a good idea to disable all of the interrupt inputs.

Interrupts

- A signal to the processor that an event occurred requiring a response
- The processor responds by:
  - Saving the state of the code that was running
  - Transferring control to a function written specifically to deal with that event



Enabling the Interrupt Controller

The last setup step:

Specify an interrupt service routine (ISR).

- This is a function in your code that is called when the processor detects an interrupt.

```
static void interrupts_isr() {
    …
    …
}
```

This function will be a helper function in your Interrupt Controller Driver (inside *interrupts.c*)

How do you do this?

- Call the following and provide a function pointer:
  `armInterrupts_setupIntc(interrupts_isr);`

- Do this inside your **`interrupts_init()`** function

Now you are done setting up your interrupt controller!

At this point you should have written these functions:

```
interrupts_init()
interrupts_irq_enable()
interrupts_irq_disable()
```

```
static void interrupts_isr() {
    …
    …
}
```

…so assuming a hardware device (like a timer), interrupts your processor

What should you do? What code should you put in your ISR function?

We could do something application-specific…
(Like in Lab 4 you need to blink some LEDs)

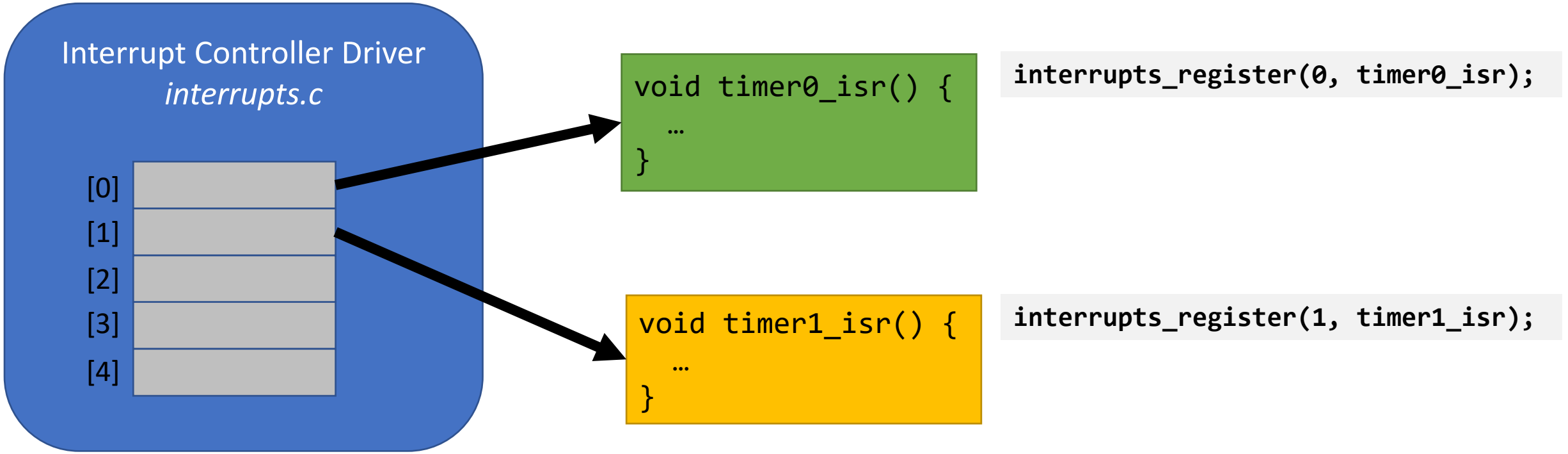
…but we want to make this driver reusable and NOT specific to one application.

ISR

Approach: Allow programs to **register** a **callback function** tied to an IRQ #.

```
void interrupts_register(uint8_t irq, void (*fcn)());
```

Your ISR can check which interrupt input fired, and call the appropriate callback function.



This allows us to run application-specific interrupt code from our driver.

Array of Function Pointers

Declaring function pointer array:

```
static void (*isrFcnPtrs[3])() = {NULL};
```

Global variable in your interrupts.c

Storing function pointer in array:

```
isrFcnPtrs[2] = fcn;
```

Do this in `interrupts_register()`

Calling a function:

```
isrFcnPtrs[2]();
```

Do this in your interrupt handler (interrupts_isr)

Interrupt Handler

This needs to check which interrupt inputs are pending, and call the appropriate function:

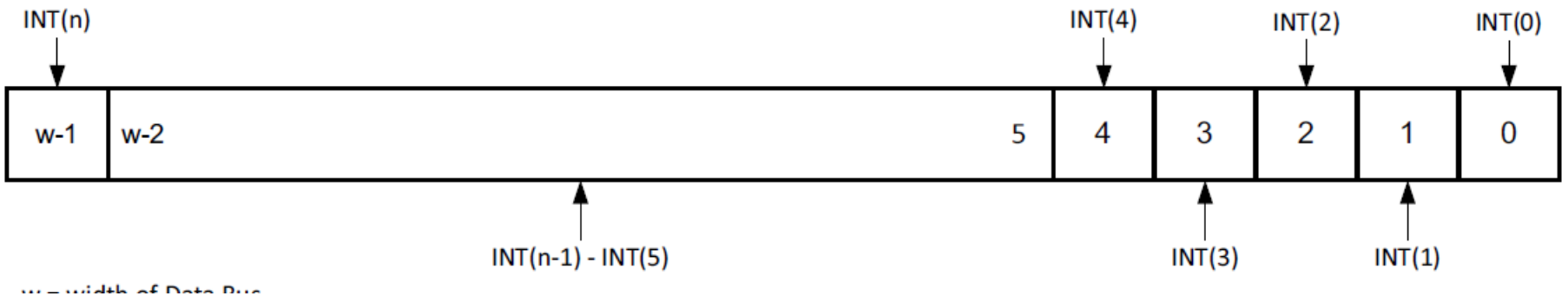
```
static void interrupts_isr() {

  // Loop through each interrupt input
  for (i, 0 to # interrupt inputs - 1) {

    // Check if it has an interrupt pending
    if (input i has pending interrupt) {

      // Call the callback function


}}}
```

Table 2-4: **Register Address Mapping**

Address Offset	Register Name	Description
00h	ISR	Interrupt Status Register (ISR)
04h	IPR	Interrupt Pending Register (IPR)
08h	IER	Interrupt Enable Register (IER)
0Ch	IAR	Interrupt Acknowledge Register (IAR)
10h	SIE	Set Interrupt Enables (SIE)
14h	CIE	Clear Interrupt Enables (CIE)
18h	IVR	Interrupt Vector Register (IVR)
1Ch	MER	Master Enable Register (MER)
20h	IMR	Interrupt Mode Register (IMR)
24h	ILR	Interrupt Level Register (ILR)
100h to 17Ch	IVAR	Interrupt Vector Address Register (IVAR)
200h to 2FCh	IVEAR	Interrupt Vector Extended Address Register (IVEAR)

Interrupt Pending Register (IPR)

This is an optional read-only register in the AXI INTC and can be set in Vivado Design Suite Customize IP dialog box by checking **Enable Interrupt Pending Register** (parameter C_HAS_IPR). Reading the contents of this register indicates the presence or absence of an active interrupt that is also enabled. This register is used to reduce interrupt processing latency by reducing the number of reads of the INTC by one.



Interrupt Handler


```
static void interrupts_isr() {

  // Loop through each interrupt input
  for (i, 0 to # interrupt inputs - 1) {

    // Check if it has an interrupt pending
    if (input i has pending interrupt) {

      // Check if there is a callback
      if (isrFcnPtrs[i])
     // Call the callback function
        isrFcnPtrs[i]();

}}}
```

Pause for class activity…

We now have:

An **enabled** interrupt system

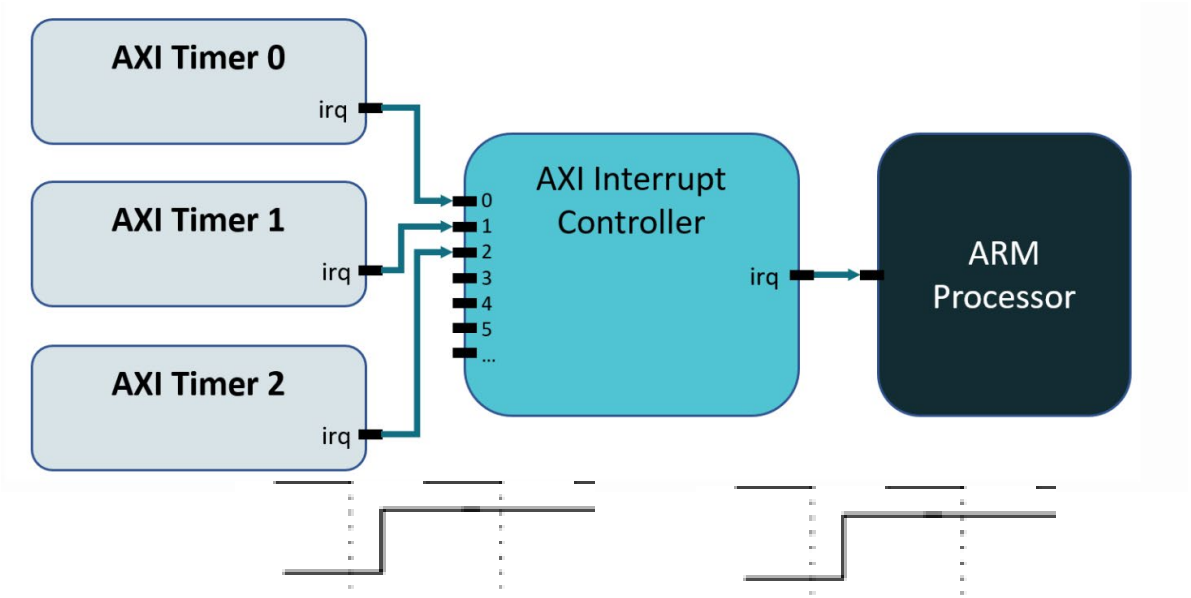
That can call application **callback** functions when an interrupt occurs.

…the last piece is to acknowledge the interrupts…

Key Fact:

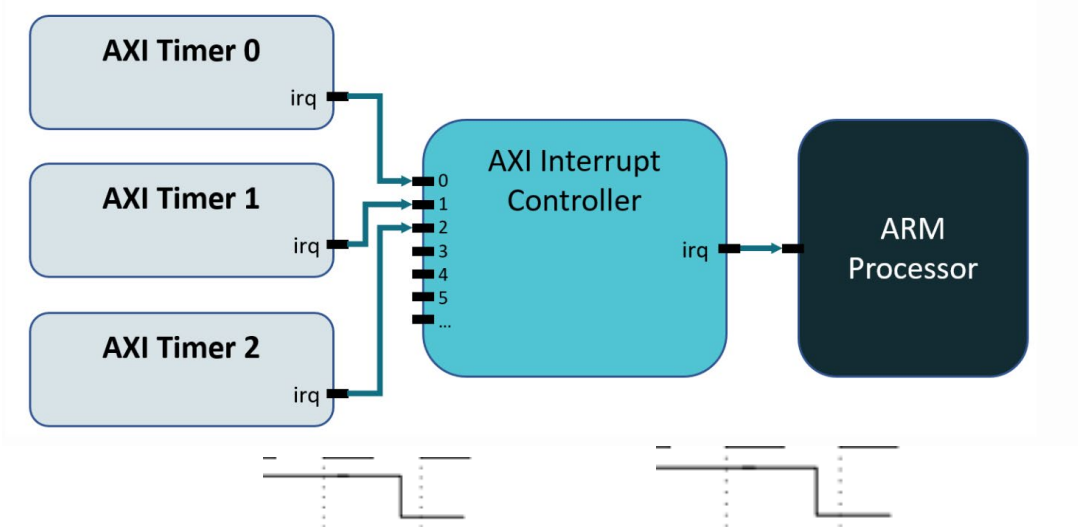
Hardware devices don't know when it's IRQ has been handled.

- So (typically) they keep sending the IRQ until the software acknowledges/clears it.



Even if the IRQ input the interrupt controller goes low, it keeps sending its interrupt signal until you acknowledge.

Acknowledging Interrupts



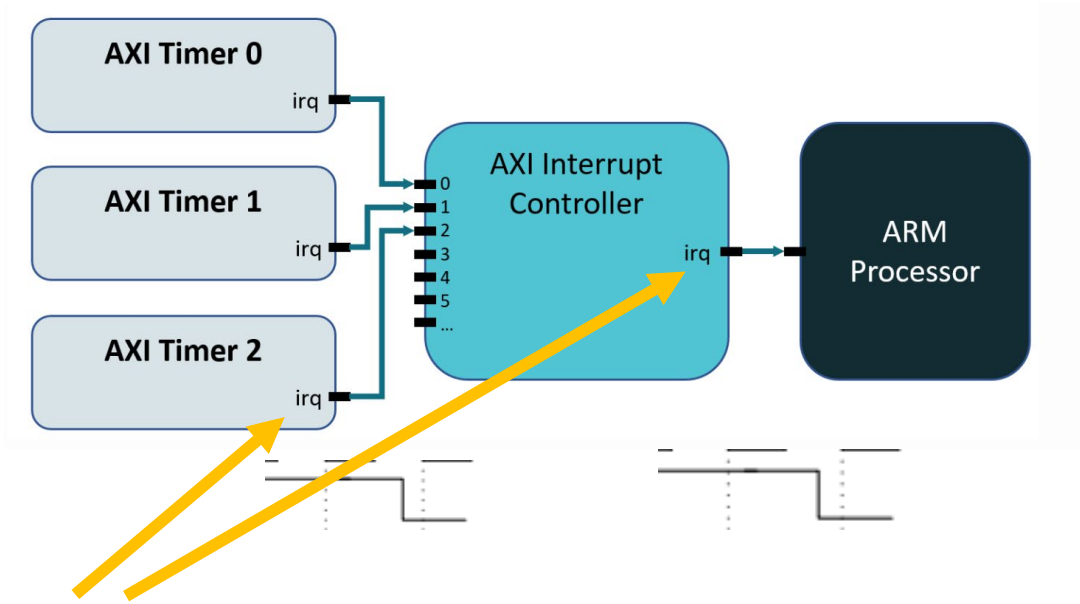
Q1: If you don't acknowledge the IRQ on the Interrupt Controller, what will happen when your ISR function completes?

- Your ISR will immediately be called again. Infinite loop! You will never return to your program…

Q2: If you don't acknowledge the IRQ from the Timer, what will happen?

Q3: Does it matter which you acknowledge first?

Acknowledging Interrupts

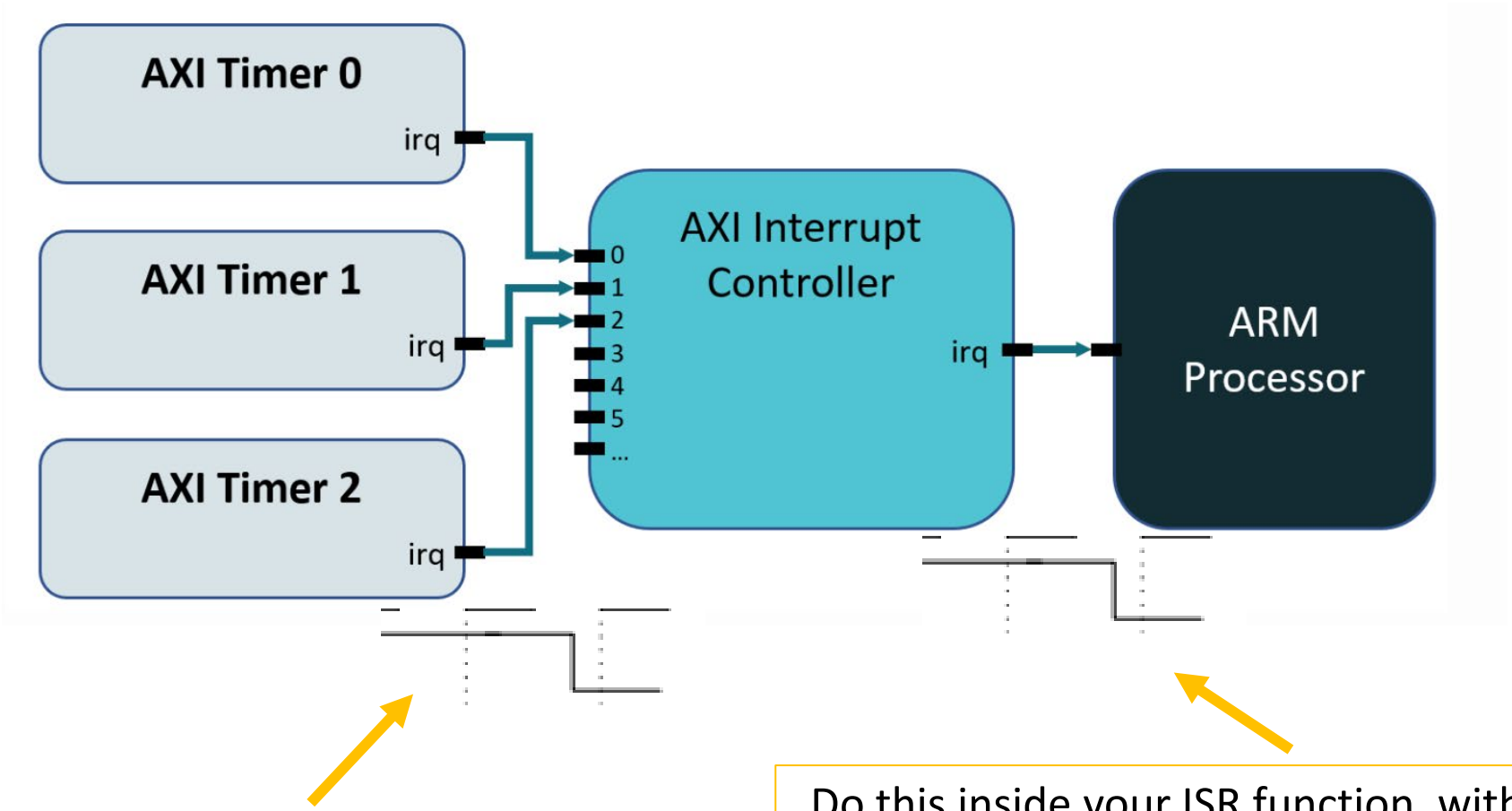


We need to acknowledge the device generating the interrupt AND the interrupt controller.

Which code should do this?

- Acknowledging the device (Interval Timer) IRQ:
  - Do this in your application code.
  - Why? The Interrupt Controller driver should be *independent* of devices that connect to it.
- Acknowledging the Interrupt Controller IRQ:
  - Do this in the driver code.
  - Why? Only need to do it one place, versus several applications.

Acknowledging Interrupts



`intervalTimer_ackInterrupt()`

Call this from your application.
(You already wrote this function last lab)

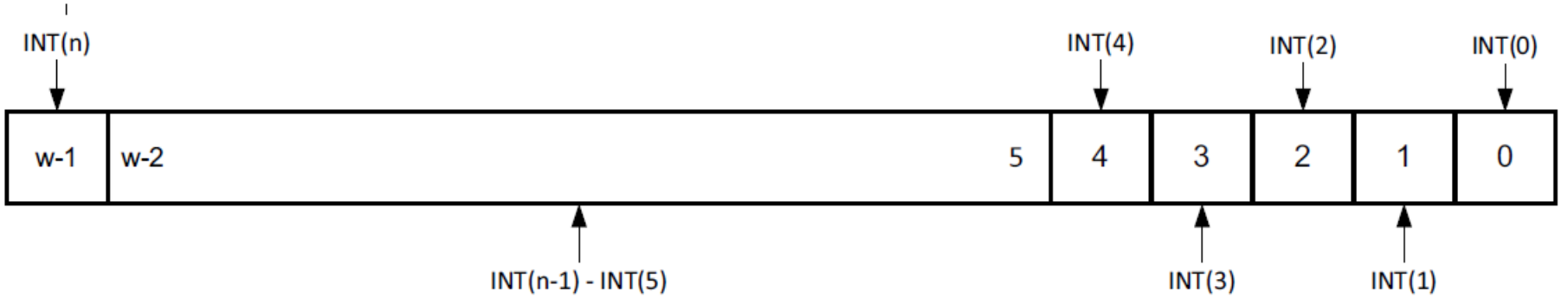
Do this inside your ISR function, within the driver.

Table 2-4: **Register Address Mapping**

Address Offset	Register Name	Description
00h	ISR	Interrupt Status Register (ISR)
04h	IPR	Interrupt Pending Register (IPR)
08h	IER	Interrupt Enable Register (IER)
0Ch	IAR	Interrupt Acknowledge Register (IAR)
10h	SIE	Set Interrupt Enables (SIE)
14h	CIE	Clear Interrupt Enables (CIE)
18h	IVR	Interrupt Vector Register (IVR)
1Ch	MER	Master Enable Register (MER)
20h	IMR	Interrupt Mode Register (IMR)
24h	ILR	Interrupt Level Register (ILR)
100h to 17Ch	IVAR	Interrupt Vector Address Register (IVAR)
200h to 2FCh	IVEAR	Interrupt Vector Extended Address Register (IVEAR)

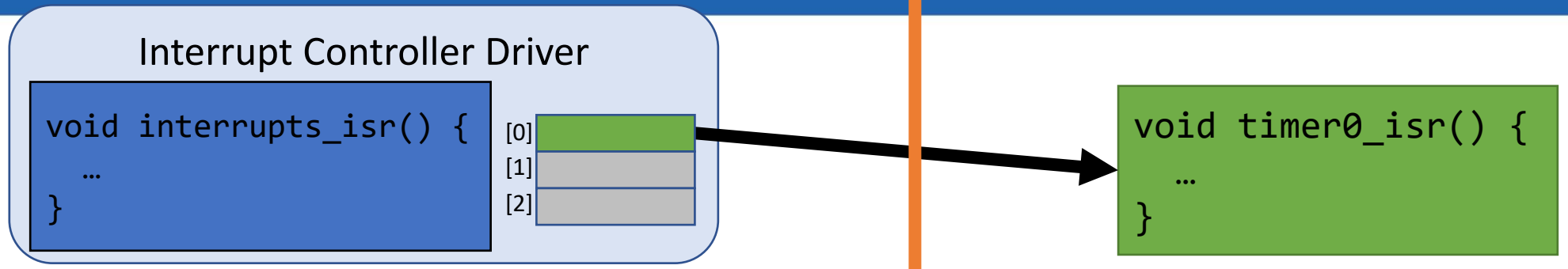
Interrupt Acknowledge Register (IAR)

The IAR is a write-only register that clears the interrupt request associated with selected interrupts. Writing 1 to a bit in IAR clears the corresponding bit in the ISR, and also clears the bit itself in IAR.



OK, let’s put it all together …

ISR




```
static void interrupts_isr() {

  // Loop through each interrupt input
  for (i, 0 to # interrupt inputs - 1) {

    // Check if it has an interrupt pending
    if (input i has pending interrupt) {

      // Check if there is a callback
      if (isrFcnPtrs[i])
        // Call the callback function
        isrFcnPtrs[i]();

    // Acknowledge interrupt
    write to IAR register
}}}
```

Driver

```
static void timer0_isr() {

  // Acknowledge timer interrupt
  intervalTimer_ack(INTERVAL_TIMER_0_INTERRUPT_IRQ );

  // Do whatever you need to do!
  // (In lab4 you need to blink an LED)
}
```

Application