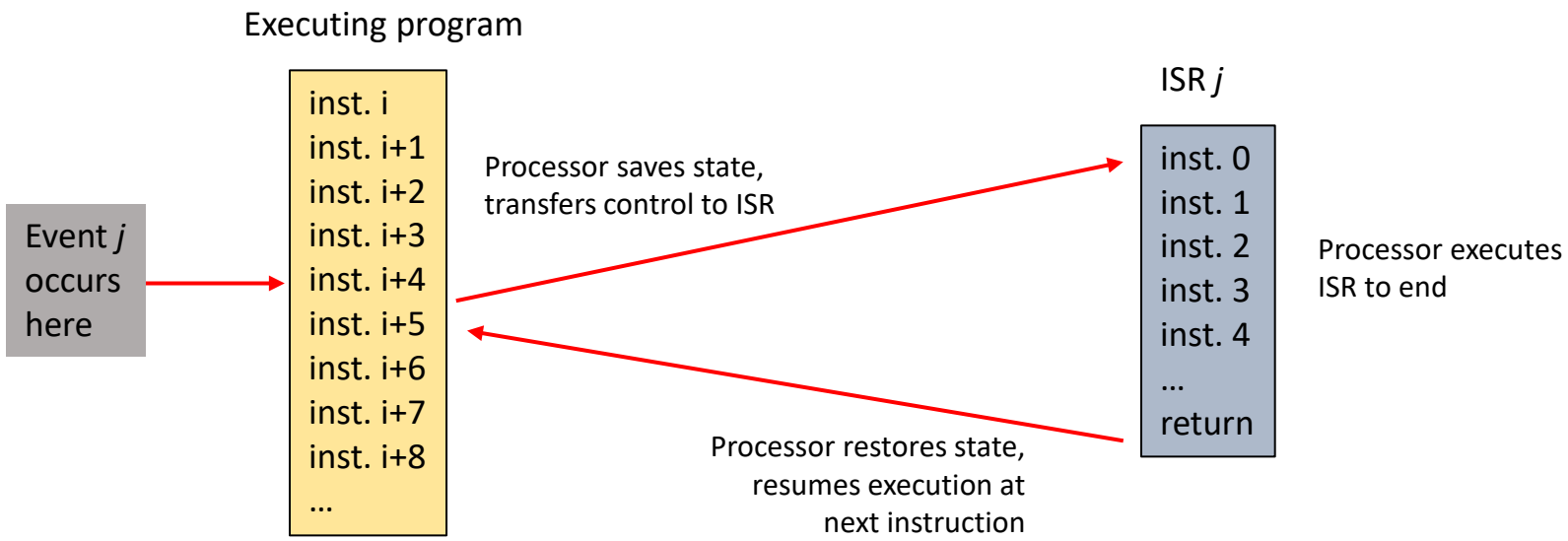
Interrupt Controller Driver

ECEN 330

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

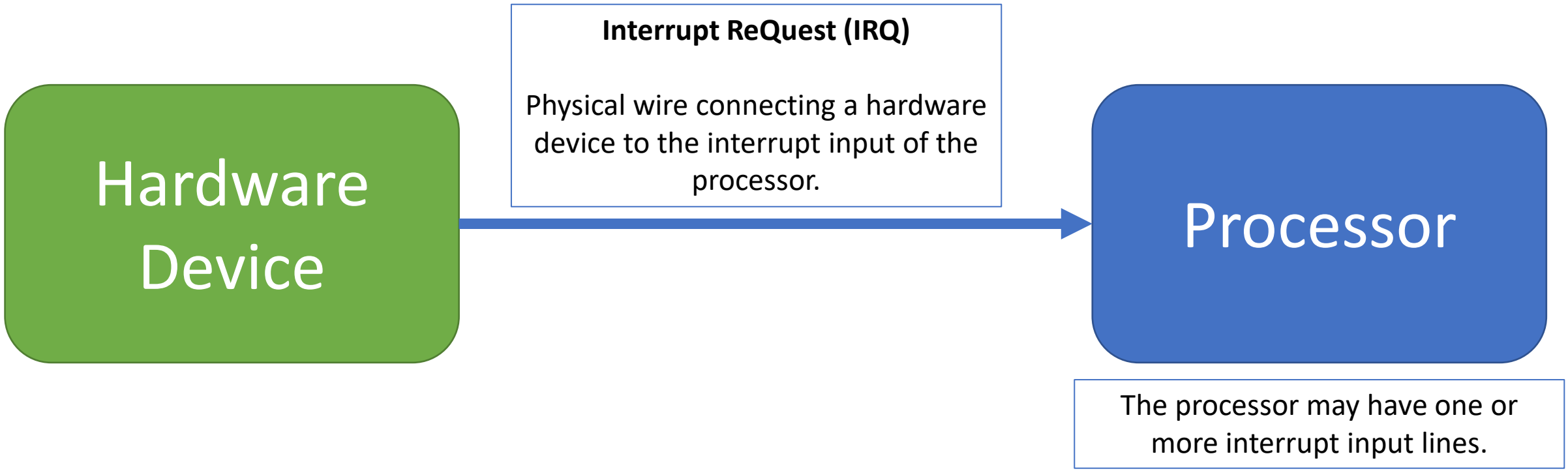
Interrupts

- 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



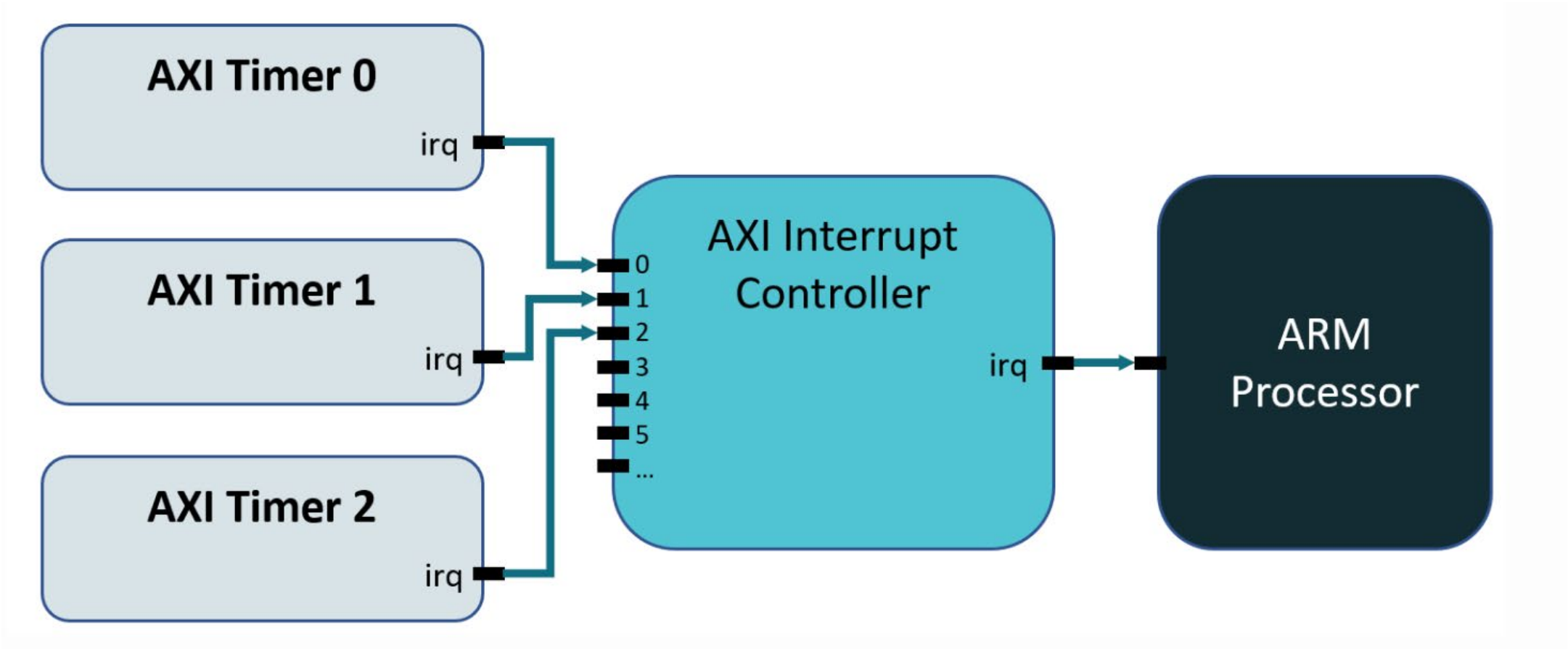
Interrupt Hardware

What does the hardware look like?



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

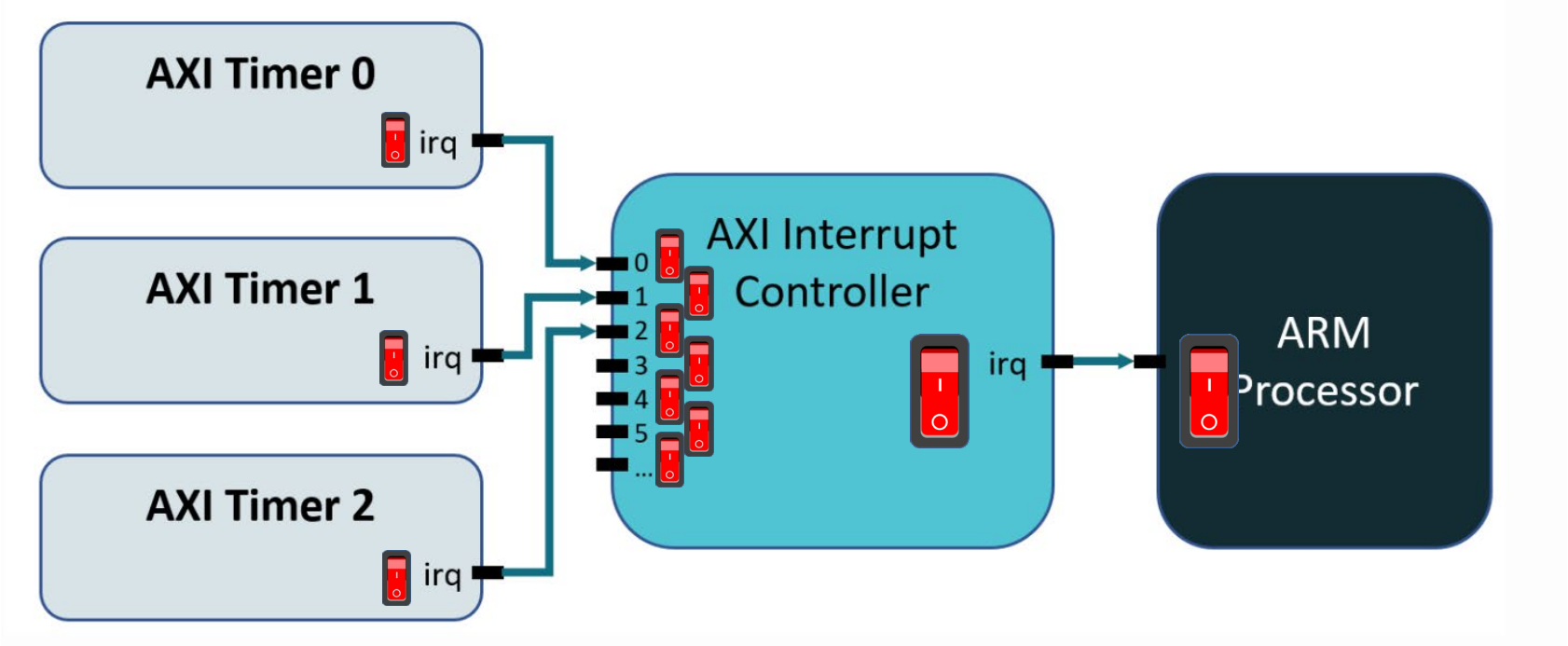
Interrupt Controller



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)

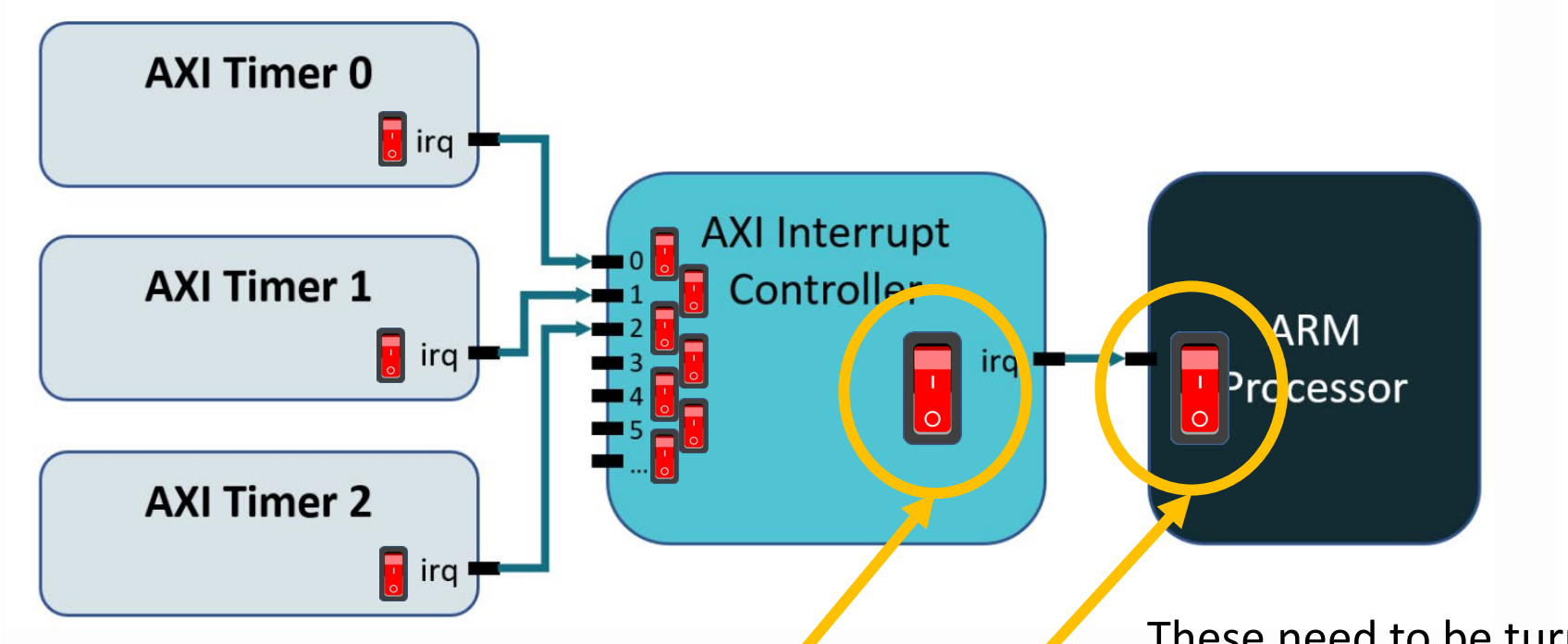
Setting Up the Interrupt Controller

For interrupts to function, they have to be enabled:



Setting Up the Interrupt Controller

For interrupts to function, they have to be enabled:



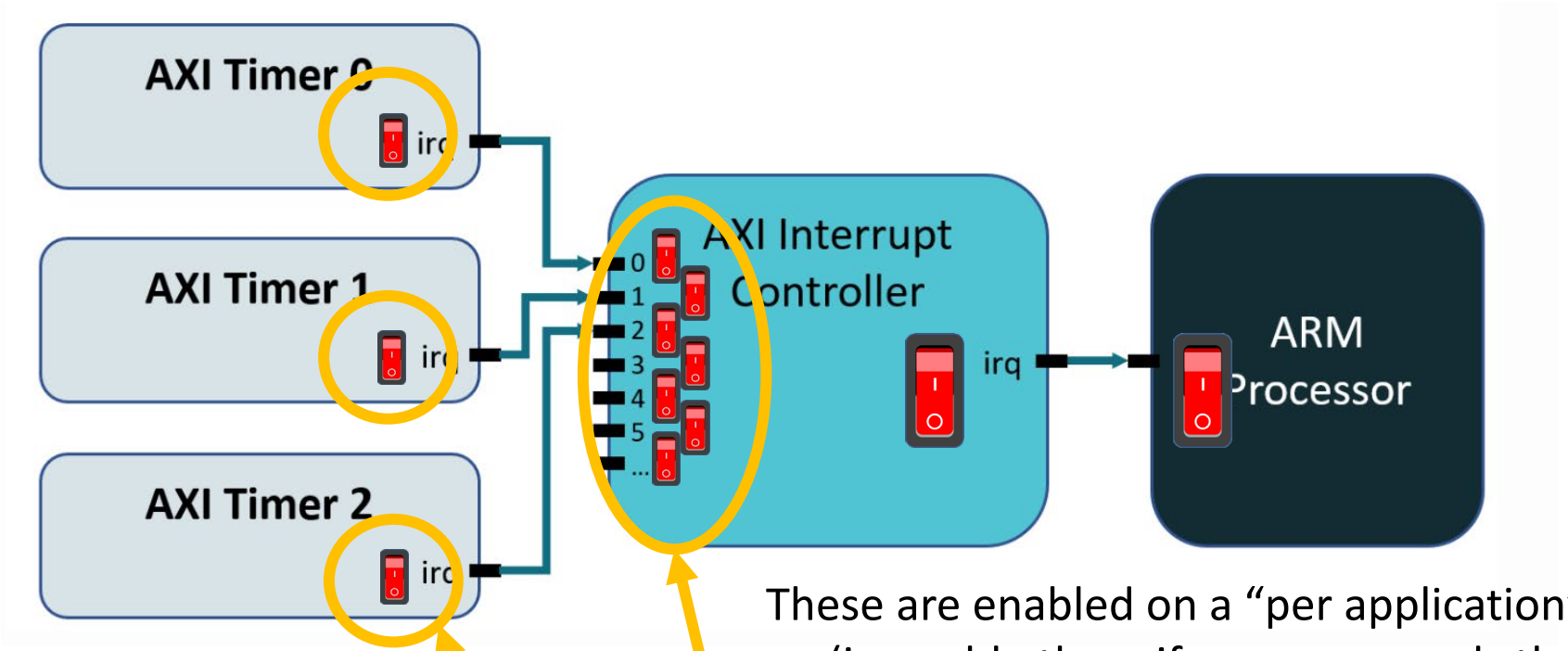
Set both bits in “Master Enable Register”

```
armInterrupts_init();
armInterrupts_enable();
```

These need to be turned on whenever you use interrupts, so turn them on in your **`interrupts_init()`** function

Setting Up the Interrupt Controller

For interrupts to function, they have to be enabled:



These are enabled on a “per application” basis
(ie enable them if your app needs them)

`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.

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)

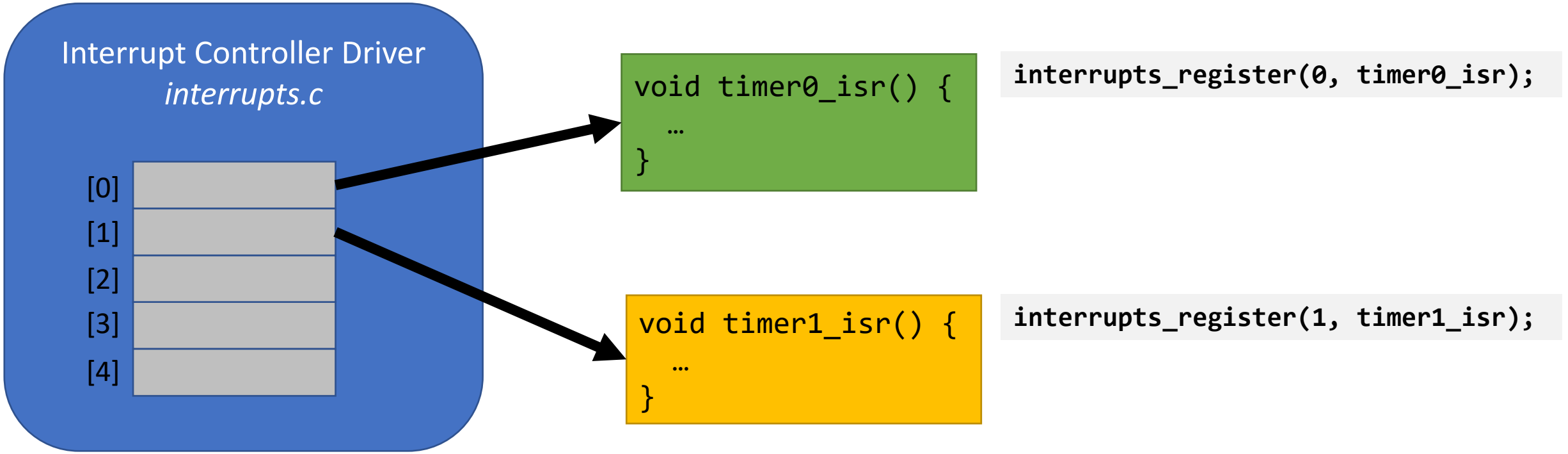
…but we want to make this driver reusable and NOT specific to one application.

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};
```

Storing function pointer in array:

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

Calling a function:

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

We now have:

An **enabled** interrupt system

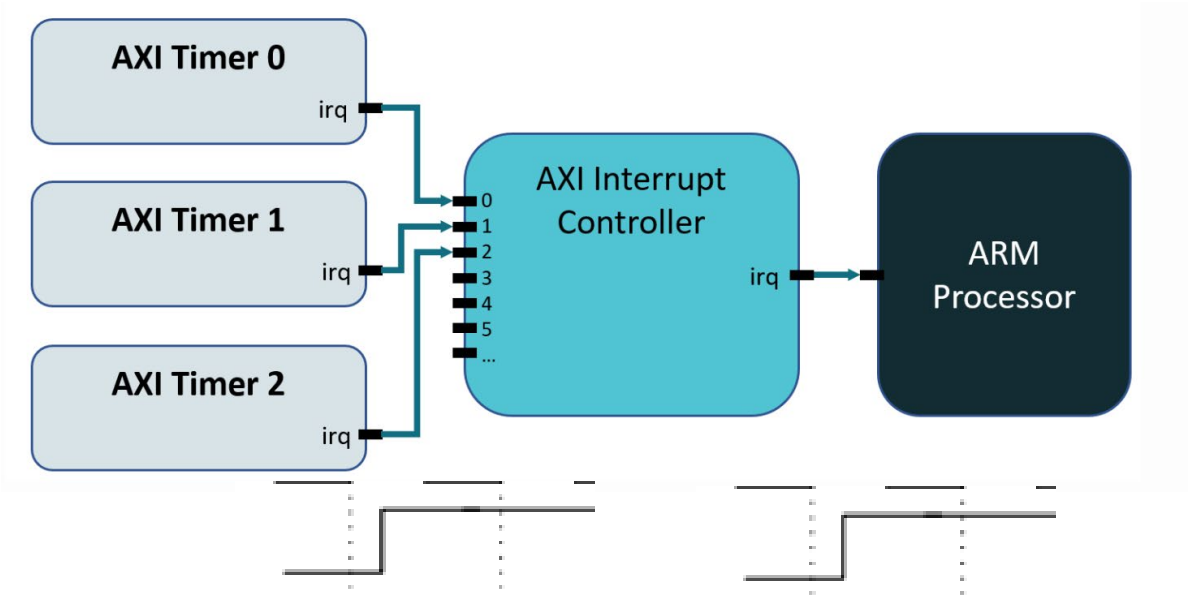
That can call application **callback** functions when an interrupt occurs.

…we are close to begin done, but missing a **critical** part…

Key Fact:

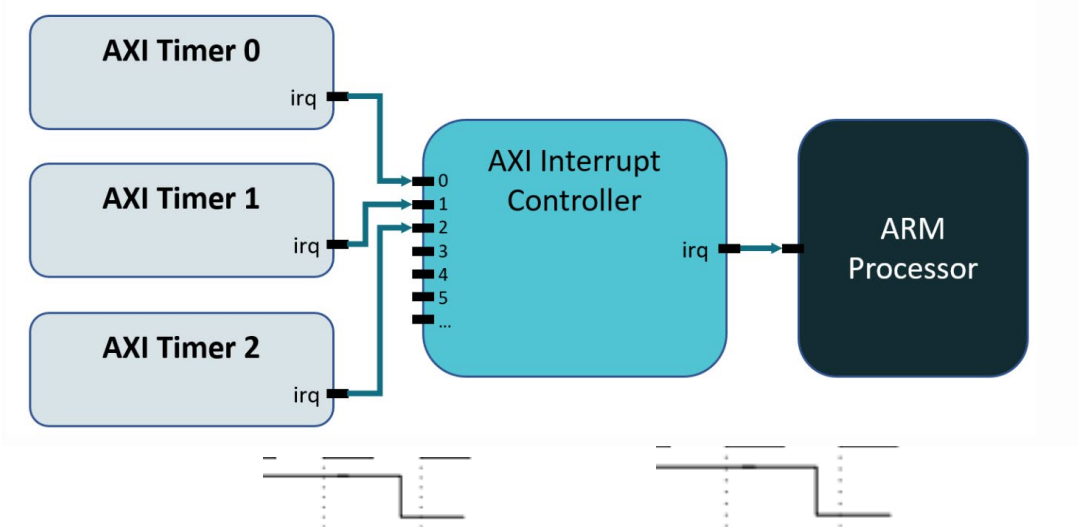
Hardware devices don't know when it's IRQ has been handled.

- 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



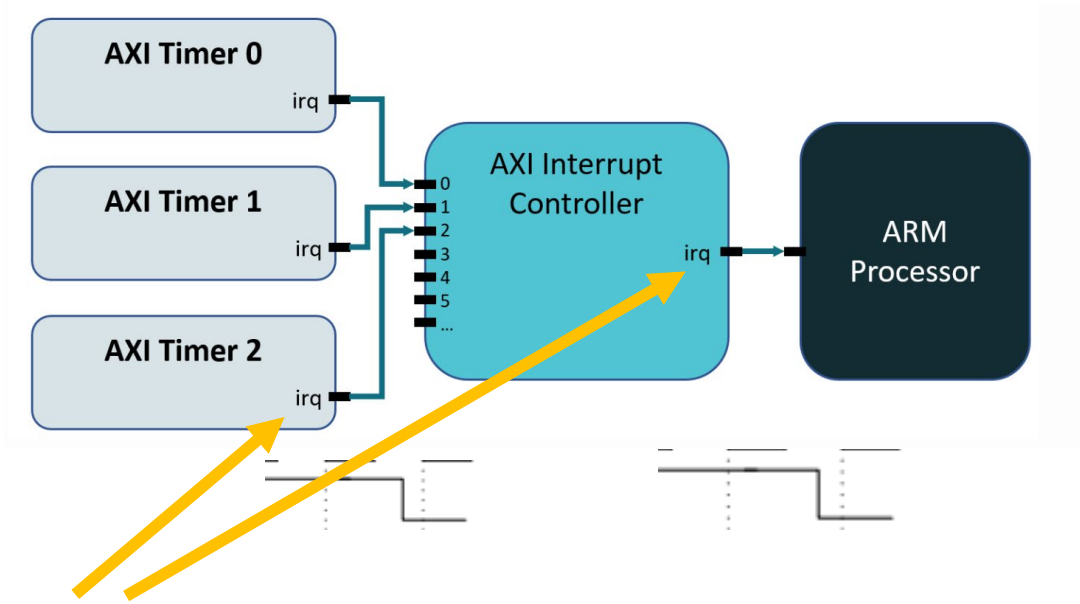
Q1: If you don't acknowledge the IRQ on the Interrupt Controller, what will happen when your ISR function completes?

- 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

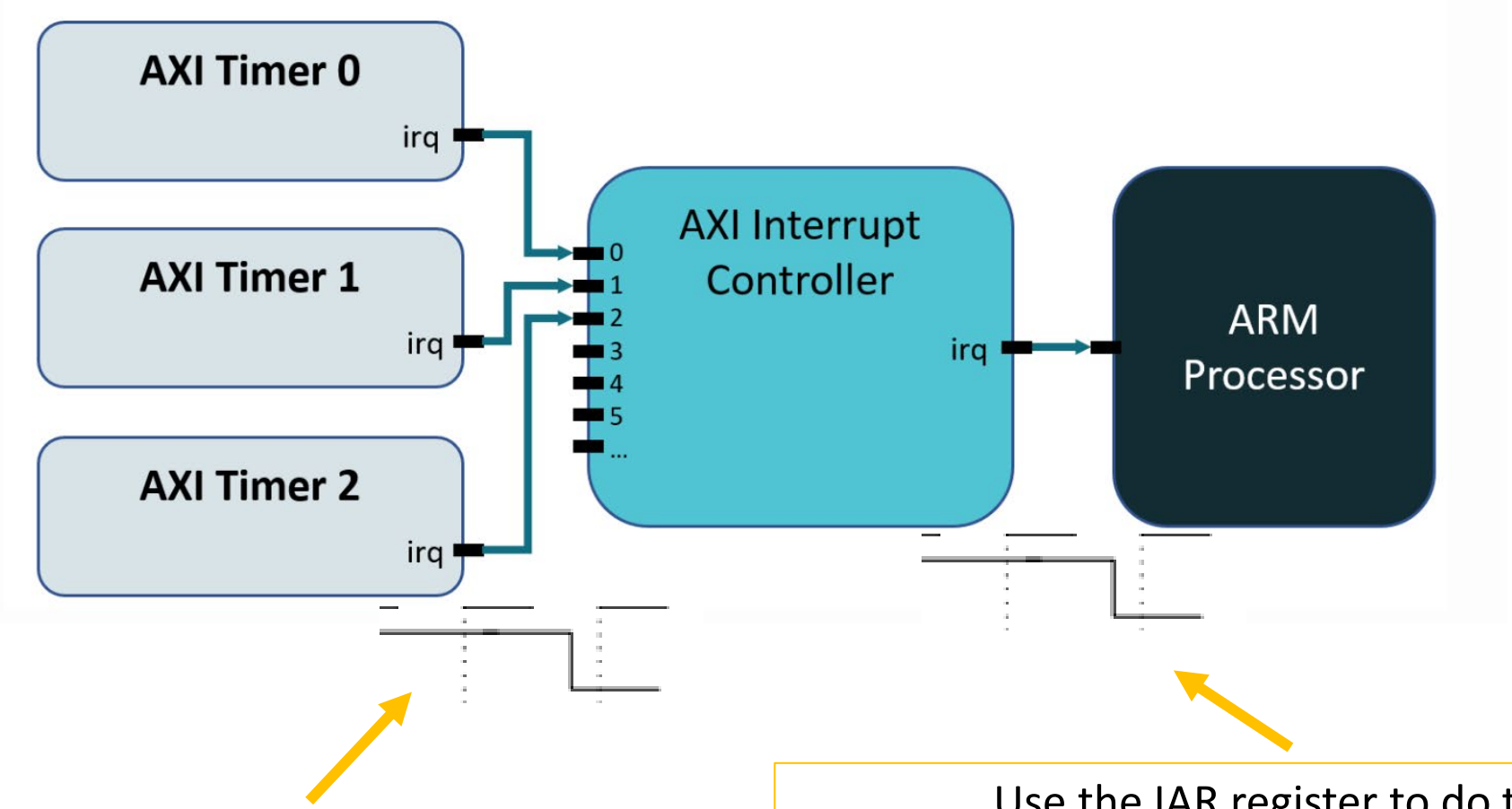


We need to acknowledge the device generating the interrupt AND the interrupt controller.

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



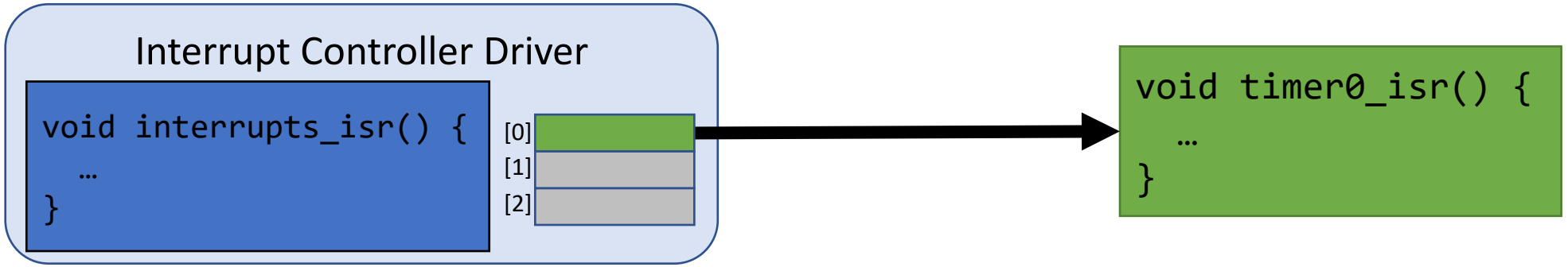
`intervalTimer_ackInterrupt()`

Call this from your application.

Use the IAR register to do this.

Do this inside your ISR function, within the driver.

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
}}}
```

```
static void timer0_isr() {

  // Acknowledge timer interrupt

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