06- Interrupt Handling

Ahmad Abdullah, PhD
abdullah@uvic.ca
https://web.uvic.ca/~abdullah/csc230

Lectures: MR 10:00 – 11:20 am
Location: ECS 125

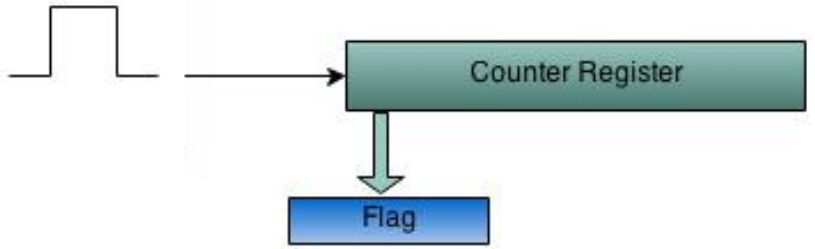
Outline

- Before interrupts: polling
- The interrupt concept
- Implementing interrupt handlers
- Nested interrupts
- More implementation issues

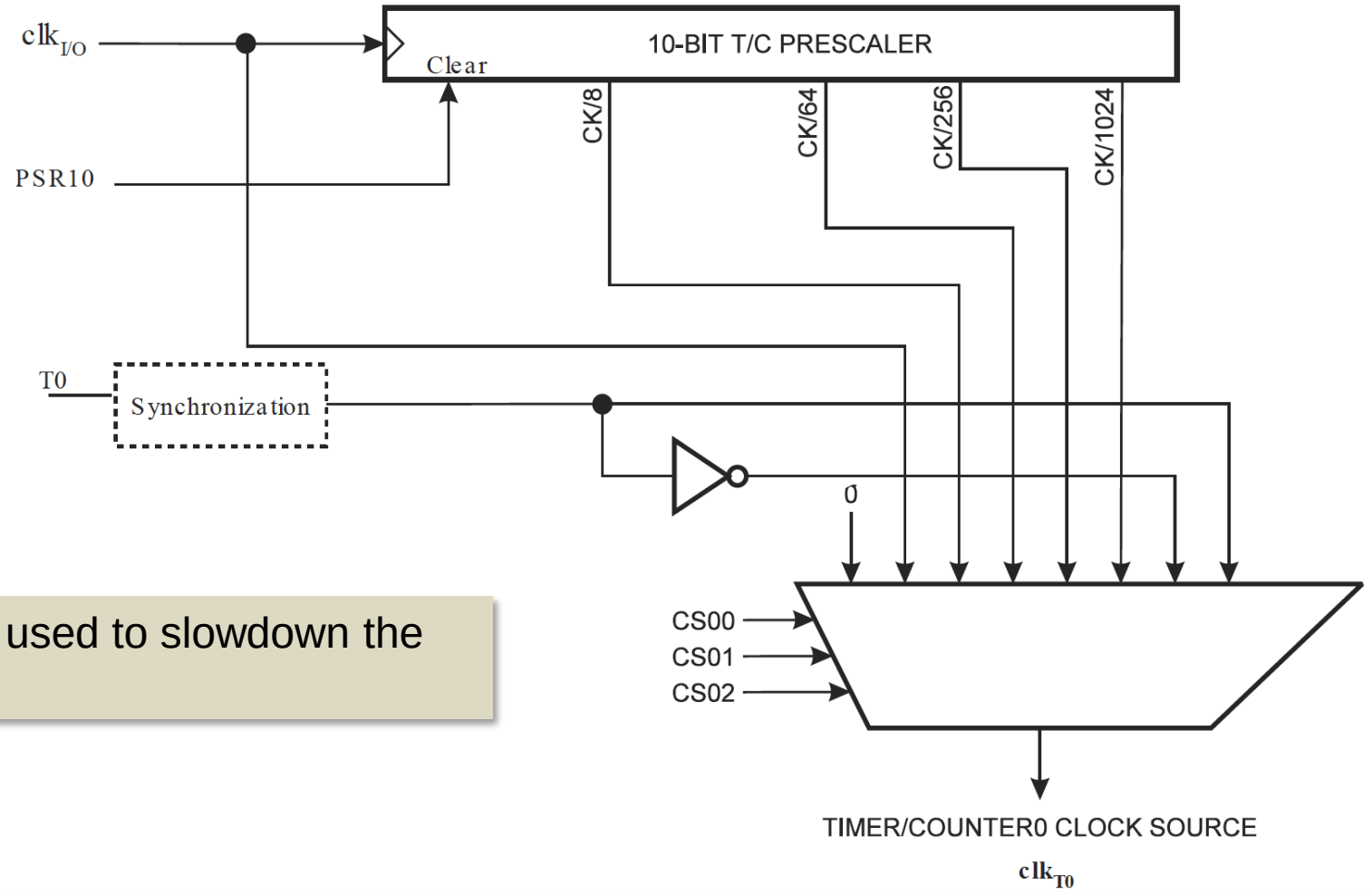
Timers

- A timers is a hardware device that keeps a track of an elapsed time between two events.
- A timer is simply a counter that counts the number of clock pulses and signals the CPU when it reaches a predefined count.

- The Atmel AT2560 processor supports several timers
  - Some of the timers are eight-bit timers (counters 0 & 2)
  - Most, however, are 16-bit timers (counters 1, 3, 4, and 5)
- In effect, these timers are bits of logic on the CPU that act independently of the ALU
  - Although they depend upon the CPU clock for their source of their oscillations...
  - ... they continue to tick independently of other code written by the programmer.

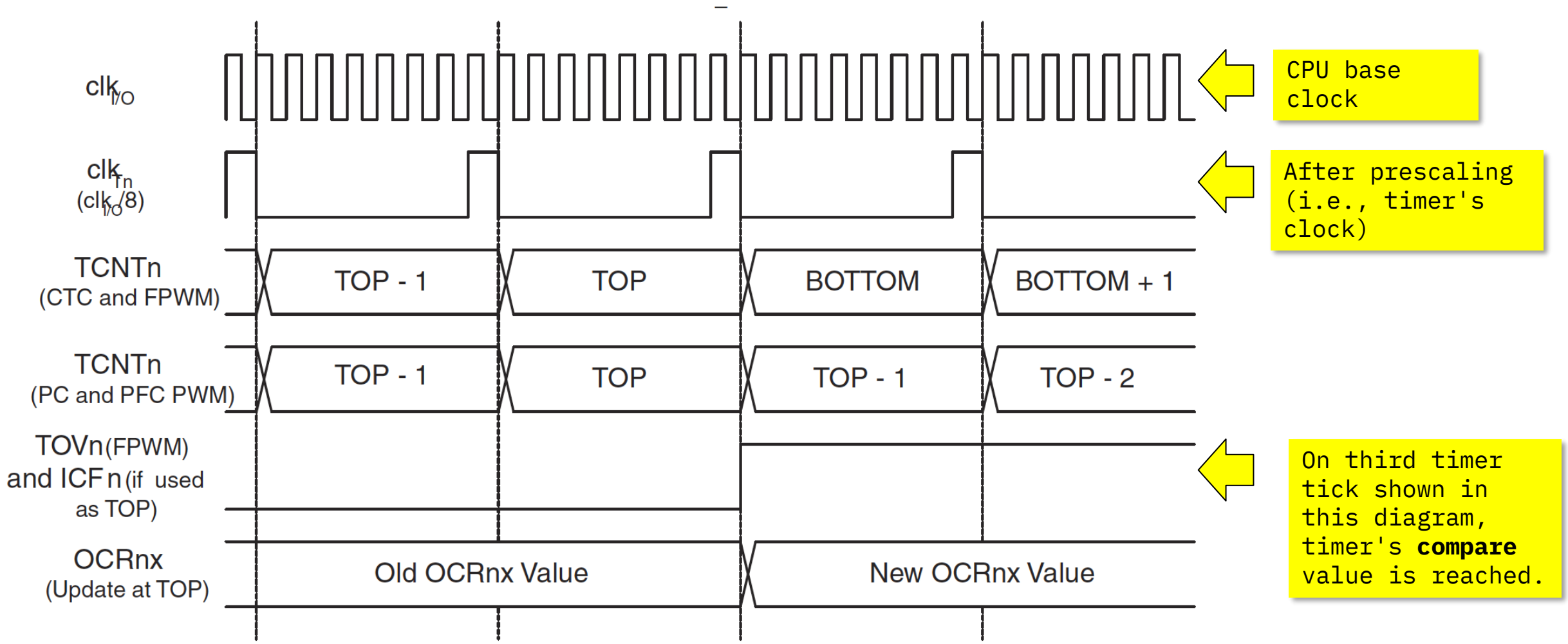


Timer diagrams



Prescalers are often used to slowdown the main clock.

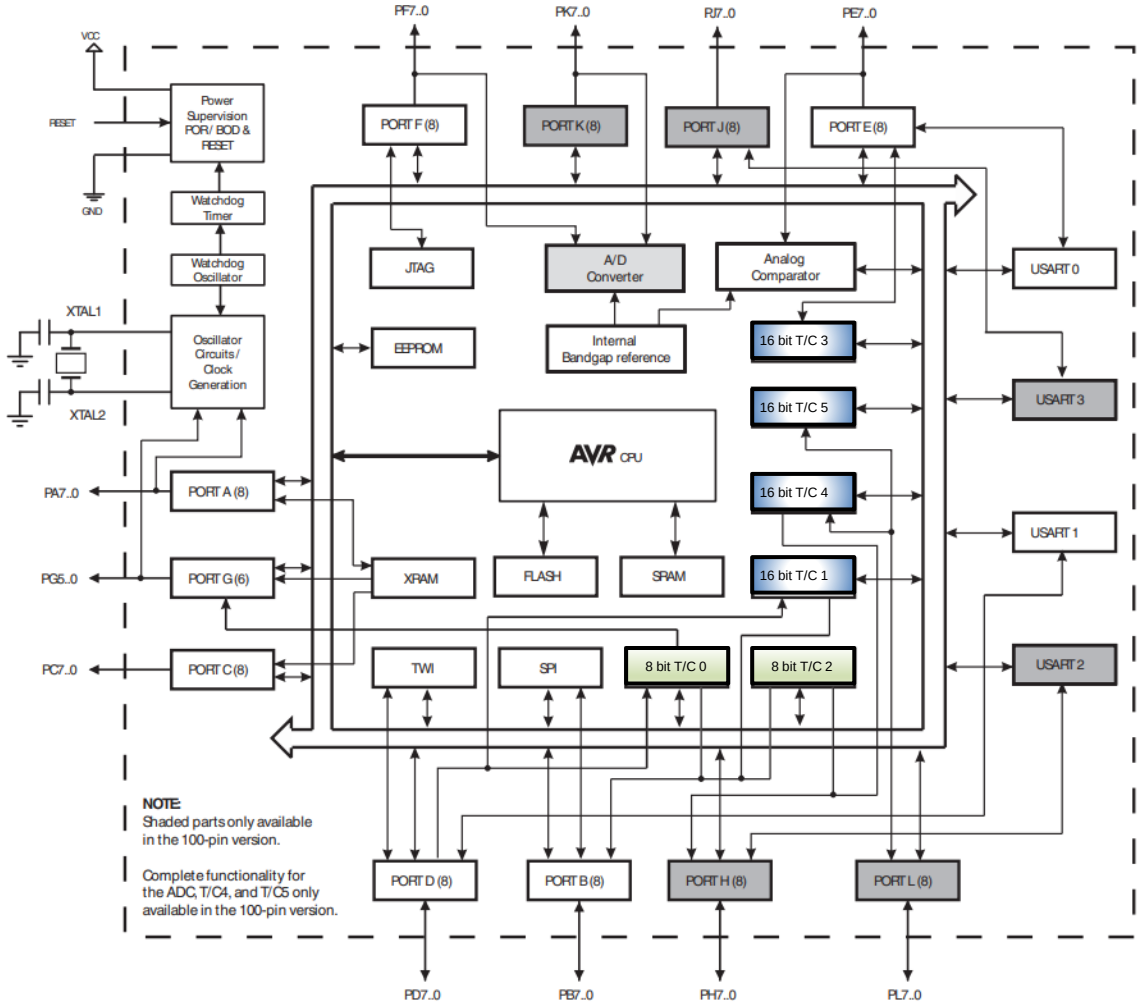
Timer diagrams (Cont.)



Atmega 2560 Timers

- The ATmega2560 has 6 timers/counters.
- Two 8-bit: T/C 0 and T/C 2.
- Four 16-bit: T/C 1, T/C 3, T/C 4 and T/C 5.
- Can be clocked by an internal or an external clock source.

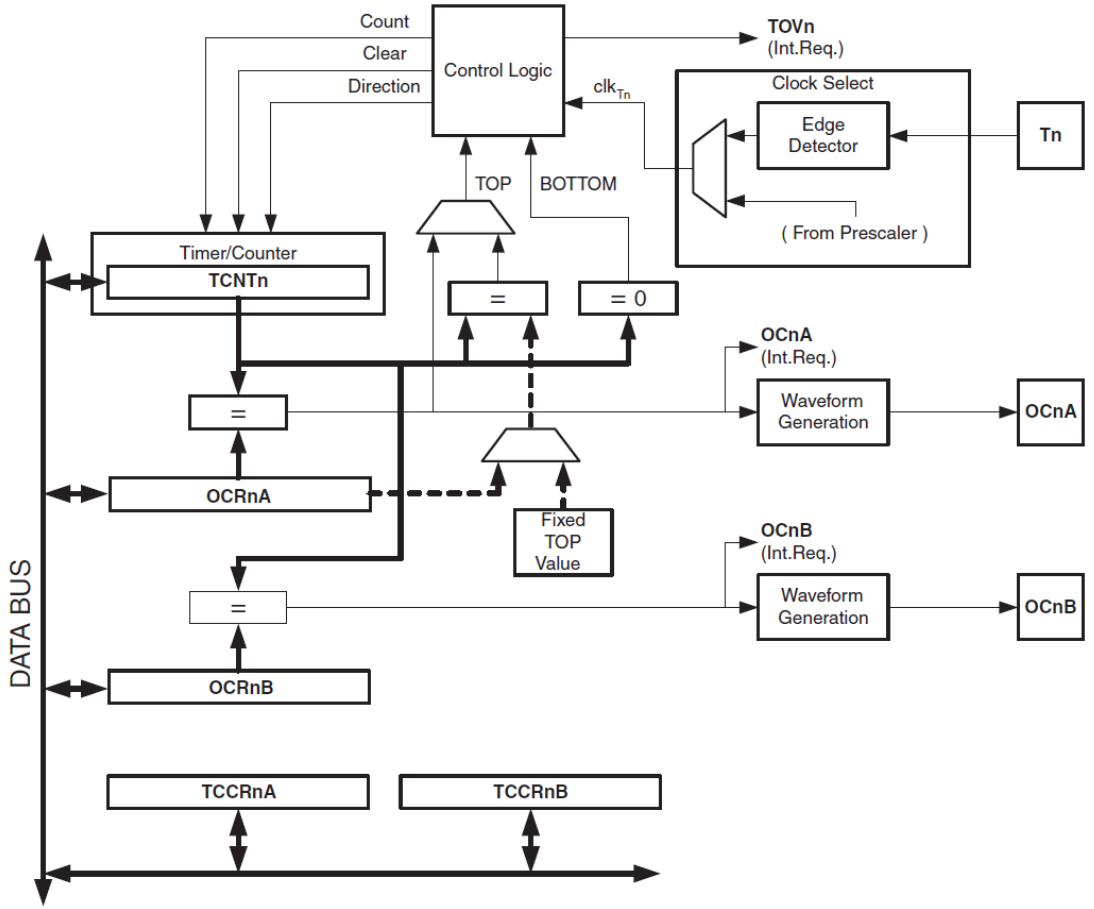
Atmega 2560 BLOCK DIAGRAM



8-bit Timer/Counter

Registers

- TCNT: Timer CouNTer register.
- OCR: Output Compare Registers.
- TCCR: Timer/Counter Control Register.
- OC: Output Compare Register.



TCCR2A – Timer/Counter Control Register A

7	6	5	4	3	2	1	0
COM2A1	COM2A0	COM2B1	COM2B0	–	–	WGM21	WGM20
R/W	R/W	R/W	R/W	R	R	R/W	R/W

Mode	WGM2	WGM1	WGM0	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	–	–	–
5	1	0	1	PWM, Phase Correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	–	–	–
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

COM2A1	COM2A0	Description
0	0	Normal port operation, OC2A disconnected
0	1	Toggle OC2A on Compare Match
1	0	Clear OC2A on Compare Match
1	1	Set OC2A on Compare Match

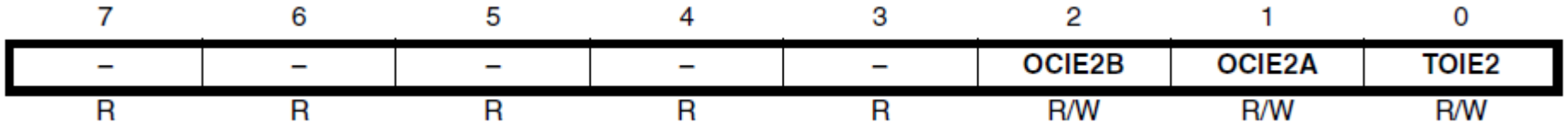
TCCR2B – Timer/Counter Control Register B

7	6	5	4	3	2	1	0
FOC2A	FOC2B	–	–	WGM22	CS22	CS21	CS20
W	W	R	R	R/W	R/W	R/W	R/W

CS22	CS21	CS20	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	clk I/O/(No prescaling)
0	1	0	clk I/O/8 (From prescaler)
0	1	1	clk I/O/32 (From prescaler)
1	0	0	clk I/O/64 (From prescaler)
1	0	1	clk I/O/128 (From prescaler)
1	1	0	clk I/O/256 (From prescaler)
1	1	1	clk I/O/1024 (From prescaler)

TIMSK2 – Timer/Counter Interrupt Mask Register

- Bit 2 – OCIE2B: Timer/Counter Output Compare Match B Interrupt Enable
- Bit 1 – OCIE2A: Timer/Counter0 Output Compare Match A Interrupt Enable
- Bit 0 – TOIE2: Timer/Counter0 Overflow Interrupt Enable



7	6	5	4	3	2	1	0
–	–	–	–	–	OCIE2B	OCIE2A	TOIE2
R	R	R	R	R	R/W	R/W	R/W

Timers (Cont.)

- Our current method of using timers depends upon **polling**
- Polling is where:
  - Some device **external to the main CPU** (i.e., low-level hardware, timer, I/O interface) ...
  - ... has **some kind of state** that **must be checked by the CPU** for readiness ...
  - ... and for which **some action must be taken** .. (i.e., a value is ready to be input from a device; a device is ready to receive a value for output; the timer has reached its maximum value; etc.s) ...
  - ... and that state is checked -- and actions taken -- **via code contained within an infinite loop**.

Polling loop

```
# ...

codeToSetUpDevices()

# Polling loop
while True:
   if device[0].isReady():
      doSomethingForDevice(0)
   else if device[1].isReady():
      doSomethingForDevice(1)
   # etc etc
   else:
      someOtherNullStepPerhaps()

# We never expect to reach this point ...
```

Polling: pluses, minuses

- Polling is a relatively straightforward technique
  - We must still figure out how to write code to detect when devices are ready for activity ...
  - ... but that is not hard.
- As we add more devices or device events that must be managed by our code, we just add more else if clauses to our loop
  - In our timer example, we had two separate timers, each blinking at a different rate.
- **Two big downsides:**
  - Even if no device is ready for action, the CPU is still busy going around and around and around in the loop consuming CPU cycle
  - **Mixing polling code with non-polling code** can be difficult.

We can do better

- Working with low-level devices is a very important part of any computer system
- Writing code for such devices is always tricky
- So rather than polling...
  - i.e., our code constantly checking for a condition to be true ...
- ... we instead inform the computer that **events of interest to our program** should cause **one of our own functions** to be executed ...
- ... and that this function execution **must happen automatically**, regardless of what the rest of our code is doing at the time of the event!

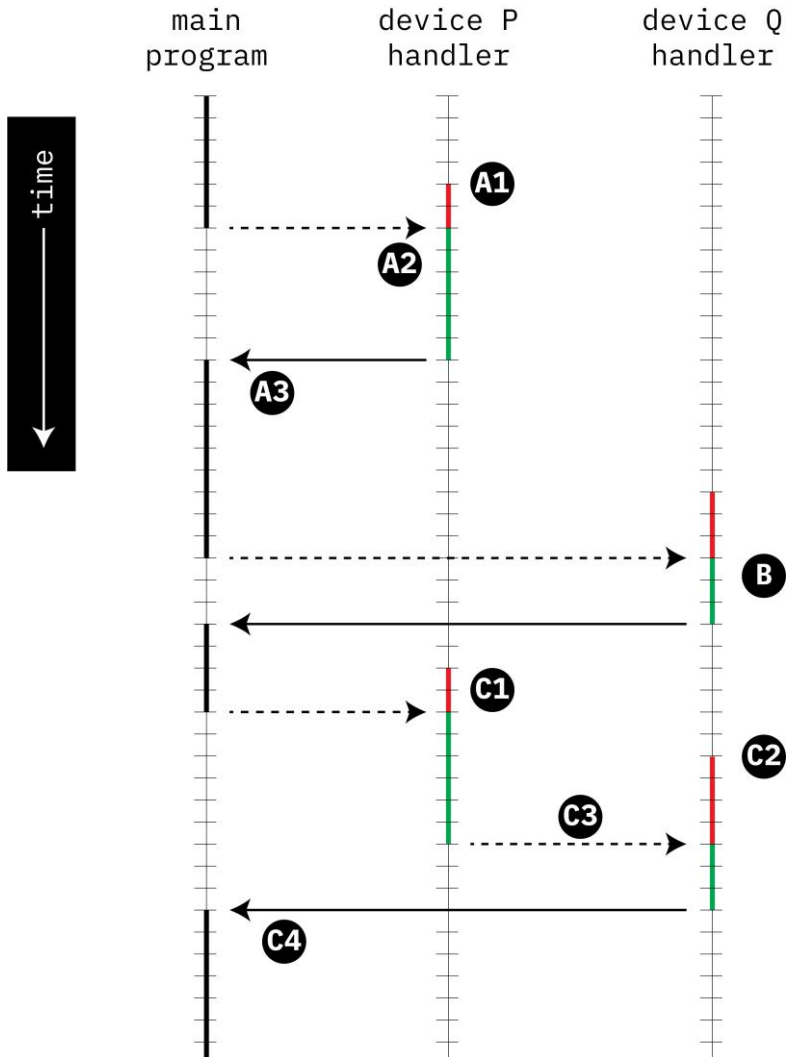
What does it mean to be interrupted?

From: https://www.youtube.com/watch?v=Mh4f9AYRCZY

Interrupt

- **An event (signal) that can temporarily halt (“interrupt”) the control flow of the currently running program in order to execute a task on behalf of that event (signal).**
- These events can be triggered by:
  - external signals (e.g., external to the CPU such as I/O devices)
  - internal signals (e.g., internal to the CPU such as timers, CPU errors)
- The running program is temporarily suspended while the task is completed
  - This task usually is some function specially written for handling the event (i.e., interrupt handler)
  - The CPU takes care of transferring control to the interrupt handler and back to the interrupted running program.

Interrupt (Cont.)



A:
Device P generates an event, it is handled after a short delay

B:
Device Q generates an event, it is handled after a short delay

C:
Device P generates an event, while device Q generates an event when P's handle is executing.

Notice that any any one point in time, only a thick black line or thick green line is active.

Implicit / explicit

- The main code whose control-flow is suspended **does not know** an interrupt has occurred
  - That is, the main code does not call the interrupt-handler function
  - Transfer of control to handler is managed by the CPU
- However, the interrupt handler function **does explicitly cause control to return** to interrupted code
  - Using **reti** instead of ret

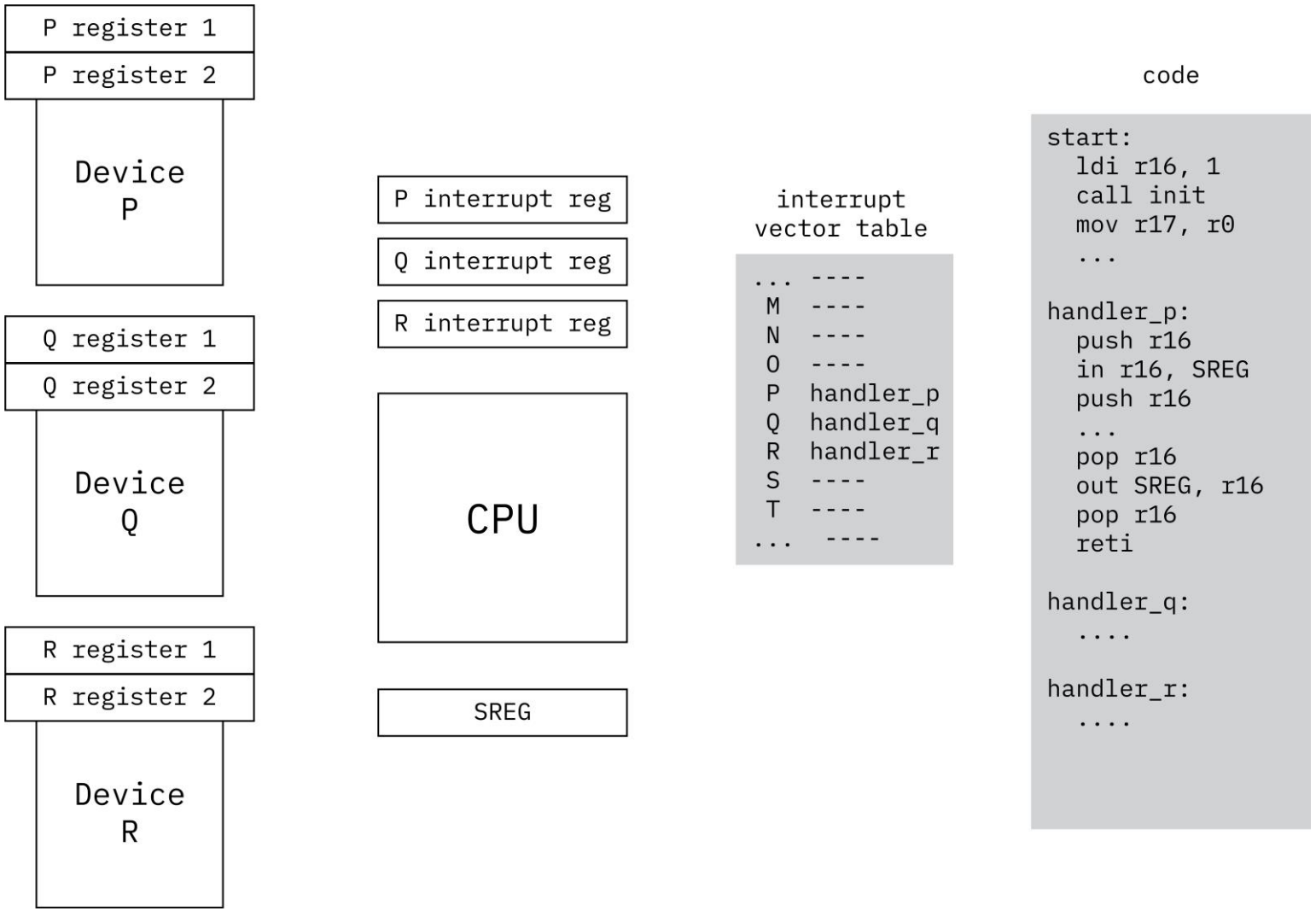
Interrupts can feel complicated

- Understanding how to program with interrupts can seem a bit tricky at first
- Some actions occur automatically, others require explicit actions on our part, and ultimately **many things will be happening simultaneously in the hardware** at runtime.
- Regardless:
  - Everything ultimately depends upon something we ourselves indicate in our code!
  - As long as we are clear about where we write code to configure interrupt handlers...
  - **... and as long as we very carefully write those handlers ...**
  - ... then coding can proceed in a straightforward fashion.

Overall idea

1. **With interrupts disabled on the CPU**, code that **configures the interrupt handler mappings** is itself executed
2. With this configuration complete, **enable interrupts**
3. Start to execute our main code
4. If an event occurs that is associated with an enabled interrupt...
   a) ... the CPU pauses where we are in our code ...
   b) ... transfers control to the correct handler for that event ...
   c) ... and that handler disables interrupts ...
   d) ... after which the handler does its work (i.e., code within the interrupt handler) ...
   e) ... and when the handler is finished, the second-to-last thing it does in re-enable interrupts ...
   f) ... and the last thing it does is causes the CPU to return control to where we originally paused.

Overall idea (Cont.)



Important

- Interrupts are not arbitrary programmatic things
  - They are not the same as threads in multithreaded programming
  - They are not used to implement multicore algorithms
- The CPU architecture will usually define the set of interrupts it can recognize
  - These interrupts are usually coupled to hardware devices the architects expect to be used
  - Some are to internal resources (such as timers)
- Programming for interrupts therefore means knowing something about what interrupt set the CPU supports
- ATmega2560: 57 interrupts (including the RESET interrupt)

Programming for interrupts

- **Interrupt vector entry**: entry in a special table that associates a device's interrupt with its interrupt-handler function
- **Interrupt handler**: A function that must be correctly implemented for the device
  - Requires knowledge of the device (obtained from a datasheet)
- Enabling some **specific interrupt flag**
  - This is for the actual device to permit it to raise interrupts
- Once all devices / interrupts are properly established...
  - ... only then enable global interrupts.
  - That is, until then these devices can be generating events and attempting to raise interrupts, but the CPU ignores them.
  - Only when **global interrupts are enabled** will the CPU transfer control to and from our handlers.

Interrupt Vectors

- The ATmega2560 **uses the first portion of program memory as the interrupt-vector table**
- Because we have not used interrupts so far in this course, **we haven't worried about this table...**
- **... but now we must!**
- To correctly make an entry we need to know:
  - The name of the interrupt
  - The interrupt-vector address at which the CPU expects the handler label
  - The handler label itself

Interrupt Vectors (Cont.)

14. Interrupts

This section describes the specifics of the interrupt handling as performed in ATmega640/1280/1281/2560/2561. For a general explanation of the AVR interrupt handling, refer to “Reset and Interrupt Handling” on page 17.

14.1 Interrupt Vectors in ATmega640/1280/1281/2560/2561

Table 14-1. Reset and Interrupt Vectors

Vector No.	Program Address(2)	Source	Interrupt Definition
1	$0000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
5	$0008	INT3	External Interrupt Request 3
6	$000A	INT4	External Interrupt Request 4
7	$000C	INT5	External Interrupt Request 5
8	$000E	INT6	External Interrupt Request 6
9	$0010	INT7	External Interrupt Request 7
10	$0012	PCINT0	Pin Change Interrupt Request 0
11	$0014	PCINT1	Pin Change Interrupt Request 1
12	$0016(3)	PCINT2	Pin Change Interrupt Request 2
13	$0018	WDT	Watchdog Time-out Interrupt

Interrupt Vectors (Cont.)

Table 14-1. Reset and Interrupt Vectors

Vector No.	Program Address(2)	Source	Interrupt Definition
1	$0000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
15	$001C	TIMER2 COMPB	Timer/Counter2 Compare Match B
16	$001E	TIMER2 OVF	Timer/Counter2 Overflow
17	$0020	TIMER1 CAPT	Timer/Counter1 Capture Event
18	$0022	TIMER1 COMPA	Timer/Counter1 Compare Match A
19	$0024	TIMER1 COMPB	Timer/Counter1 Compare Match B
20	$0026	TIMER1 COMPC	Timer/Counter1 Compare Match C
21	$0028	TIMER1 OVF	Timer/Counter1 Overflow

```
.cseg
.org 0
    jmp start

.org 0x22
    jmp timer1_handler

...

start:
    ....

timer1_handler:
    ....
```

Interrupt Vectors (Cont.)

- We must be very careful here:
  - If we have turned on a device's interrupts...
  - ... but have not properly placed an entry in the vector table ...
  - ... then the ATmega2560 will use whatever values it finds at the table location (possibly random!) as the interrupt handler.
- Put differently:
  - The assembler **does not warn us if we have been sloppy with our programming.**
  - We must manually match interrupts used in our solution to handler labels written in the vector table.

Interrupt Handler

- Handlers are meant to be the **smallest amount of code necessary** to deal with the interrupt
  - For I/O devices, this would be reading bytes from, or writing bytes to, the registers of the device.
  - For timers, this would involve setting status variables to important values
- Code within an interrupt-handler function very rarely calls other functions
  - The time spent handling one interrupt is unavailable for handling those from other devices...
  - ... and given the time sensitivity of device behavior, missing another device's interrupt could be disastrous.
- Finally: handlers must ensure the interrupted code can safely resume operation after handler is finished

Example: reading button value

```
.....

timer1_handler:
    push r16
    in r16, SREG
    push r16

    push XH
    push XL
    push r17
    mov XH, high(BUTTON_COUNT)
    mov XL, low(BUTTON_COUNT)
    ld r17, X
    inc r17
    st X, r17
    pop r17
    pop XL
    POP XH

    pop r16
    out SREG, r16
    pop r16

    reti
```

Save r16 for later restoration, and also save the current value of the status register.

Actual task meant for the handler.

Restore the status-register value, and also restore r16

Return from the interrupt.

Example: LED ON/OFF

```
.cseg
.def leds = r16 ;current LED state
.def temp = r18 ;a temporary register

.org 0; RESET
   jmp start

.org 0x1A; Timer 2 COMPA
   jmp LED_ON
.org 0x1C; Timer 2 COMPB
   jmp LED_OFF

.cseg
start:
   ldi temp, 0xFF ; set PORTB as output
   out DDRB, temp
   clr leds; To turn OFF all LEDs
   out PORTB, leds

   ; Clear general interrupts
   Cli

   ; Clear timer 1 configurations
   clr temp
   sts TCCR2A, temp
   sts TCCR2B, temp
```

```
   ; Set to CTC (mode 2)
   lds temp, TCCR2A
   ori temp, (1<<WGM21)
   sts TCCR2A, temp

   ; Set prescalar to /1024
   lds temp, TCCR2B
   ori temp, (1<<CS22) | (1<<CS21) |
                                 (1<<CS20)
   sts TCCR2B, temp

   ; Set values for OCCR2A & OCR2B
   ldi temp, 0xFF
   sts OCR2A, temp

   ldi temp, 0x7F
   sts OCR2B, temp

   ; Enable interrupt A & B for timer 2
   lds temp, TIMSK2
   ldi temp, (1 << OCIE2A) | (1 << OCIE2B)
   sts TIMSK2, temp

   ; Reset timer 2 (optional)
   clr temp
   sts TCNT2, temp
   Sei     ; Enable general interrupts
```

```
end:
    rjmp end

LED_ON:
    push leds
    push temp
    in temp, SREG
    push temp
    ldi leds, 0xFF; To turn ON all LEDs
    out PORTB, leds
    pop temp
    out SREG, temp
    pop temp
    pop leds
    reti
LED_OFF:
    push leds
    push temp
    in temp, SREG
    push temp
    clr leds; To turn OFF all LEDs
    out PORTB, leds
    pop temp
    out SREG, temp
    pop temp
    pop leds
    reti
```

Enabling interrupts

- There are two levels here:
  - Enabling the interrupts **for the device itself**
  - Enabling the CPU **to respond to interrupts in general.**
- Sometimes the former is the same thing as “turning on the device”
  - We see this with the 16-bit timers (i.e., timer-interrupt mask register, such as TIMSK1)
  - Setting a bit on this register has the same effect as enabling interrupts to occur for that timer
- The former is the main switch and is a single operation: **sei** (set global interrupt flag)
- Therefore, usual format of code:
  - Set up all handlers, enable interrupts on all devices
  - Once this is done, then (and only then!) perform **sei**

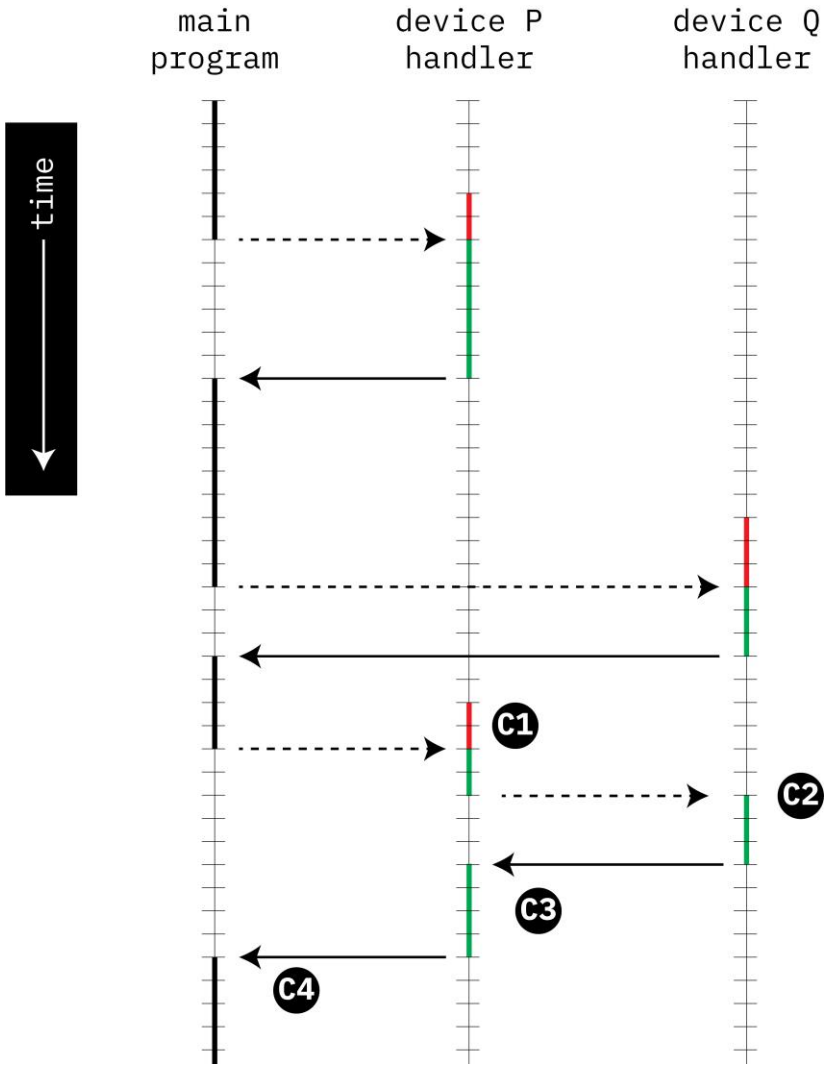
NMI

- In our handler-vector example, we saw one at location 0x00
- This vector is for the **RESET** signal
  - This is a special interrupt...
  - **... which can never be ignored!**
  - It is also never meant something from which we return
  - Sometimes referred to as a “non-maskable interrupt” or “**NMI**”
- In our case, the vector-table entry just referred to the first location of our main program (the label start that we would have written)

Multi-level interrupts

- Our examples have also assumed that only one interrupt can be handled at any one time
  - In fact, when the CPU dispatches to an interrupt handler...
  - ... it clears the global interrupt flag in **SREG** ...
  - ... and only restores it after an **RETI**
- The ATmega2560 therefore supports **single-level interrupts**
  - All interrupts have the same priority or importance
- Other computer architectures support **multi-level interrupts**
  - Here a device and its interrupt has a priority...
  - ... such that interrupts from higher-priority devices are permitted to interrupt the handlers for lower-priority devices

Multi-level interrupts (Cont.)



If device Q had a higher priority than device P...

C:
Device P generates an event, but during its handling there is an event from device Q -- and so device Q (with higher priority) has its interrupt serviced.

Once that is done, the handlers for P is resumed.

Multi-level interrupts (Cont.)

- Hardware support is normally needed for multi-level interrupts
- Motivation for such an architecture:
  - Some devices generate many more events than others (i.e., SSD interface card vs. keyboard interface)
  - Some devices have data or state that is very volatile and must be quickly read (or written) within very closer tolerances.

Critical section code

- Sometimes we may write code for which interrupts might cause a problem
- Example:
  - Timer interrupts are often used by operating systems to force switching between processes.
  - However, code in the OS may want to avoid such a switch when executing some sensitive code within the OS kernel itself
  - Therefore, the OS will disable interrupts before that sensitive code, and re-enable them afterwards
- On ATmega2560: **cli** disables interrupts, **sei** enables interrupts.

Summary

- Interrupts permit us:
  - Write code that responds to devices running in parallel with the CPU ...
  - ... in such a way that servicing those devices does not require a polling loop.
- Concepts / vocabulary:
  - interrupt / interrupt event
  - interruption of main-program control flow (and need to push and pop the status registers)
  - handlers
  - interrupt vectors
  - setting up interrupts
  - enabling / disabling interrupts
  - nested interrupts

Any Questions?