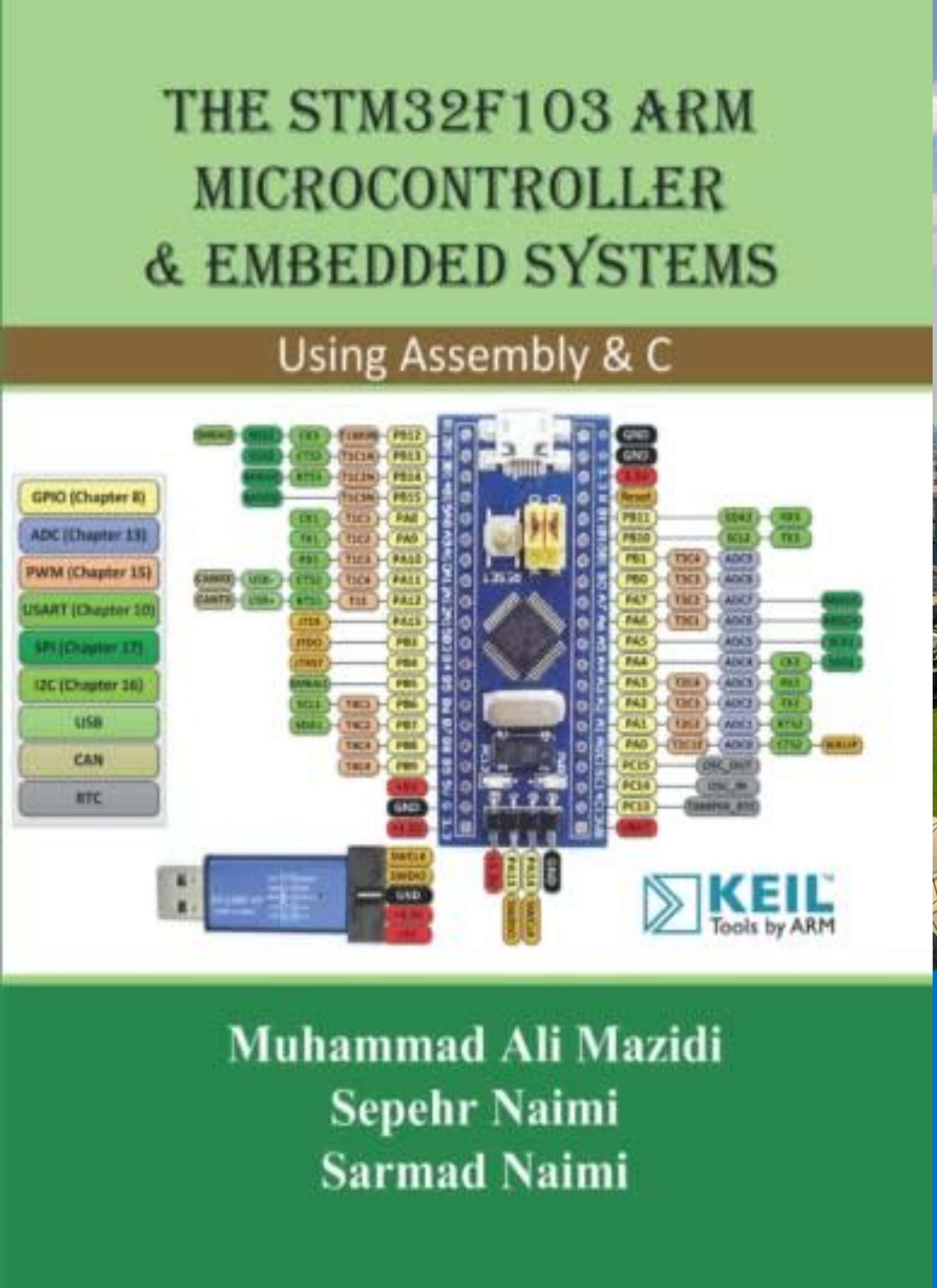


THE STM32F103 ARM MICROCONTROLLER & EMBEDDED SYSTEMS

Using Assembly & C

Muhammad Ali Mazidi
Sepehr Naimi
Sarmad Naimi

LECTURE # 03M

Arduino Interrupts

Prof. Dr. Engr. Muhibul Haque Bhuyan
Professor, Department of EEE
American International University-Bangladesh (AIUB)
Dhaka, Bangladesh

Interrupts

- An interrupt is the automatic transfer of software execution in response to a hardware event that is asynchronous with the current software execution.
- An interrupt is a signal emitted by a device attached to a computer or from a program within the computer. It requires the operating system (OS) to stop and figure out what to do next. An interrupt temporarily stops or terminates a service or a current process.
- The hardware event can either be a busy-to-ready transition in an external I/O device (like the UART input/output) or an internal event (like a bus fault, memory fault, or a periodic timer).

Introduction

- Microcontroller normally executes instructions in an orderly fetch-execute sequence as dictated by a user-written program.
- However, a microcontroller must also be ready to handle **unscheduled events** that might occur inside or outside the microcontroller.
- The interrupt system onboard a microcontroller allows it to respond to these internally and externally generated events. By definition, we do not know when these events will occur.
- When an interrupt event occurs, the microcontroller will normally complete the instruction it is currently executing and then will transfer the program control to an Interrupt Service Routine (ISR) that handles the interrupt.
- Once the ISR is complete, the microcontroller will resume processing where it left off before the interrupt event occurred.

Interrupts

- A request for the processor to ‘interrupt’ the currently executing process, so the event can be processed on time
- Also referred to as ‘trap’
- If the request is accepted: the processor suspends current processes, saves its states, and executes the Interrupt Service Routine (ISR) or, Interrupt Handler
- The concept of ‘Interrupt’ is very useful when implementing any of the switch debouncing methods.
- **Scheduled events are not interrupts.**

Introduction

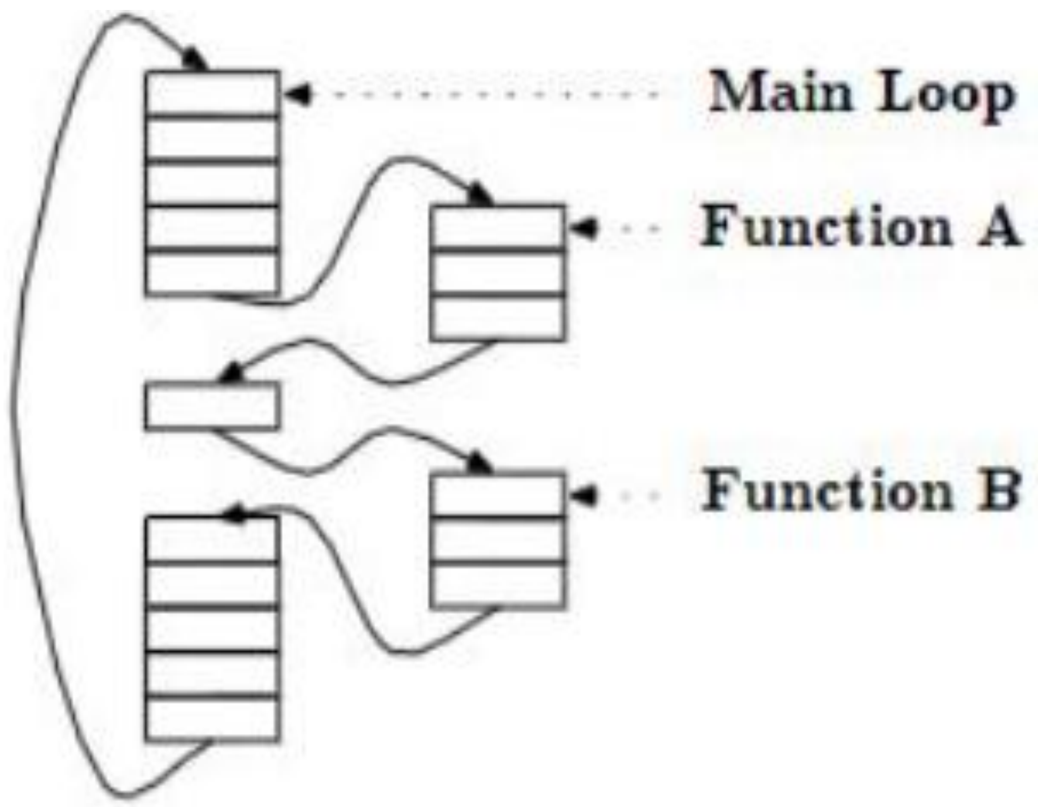


Figure: Outline of Instructions

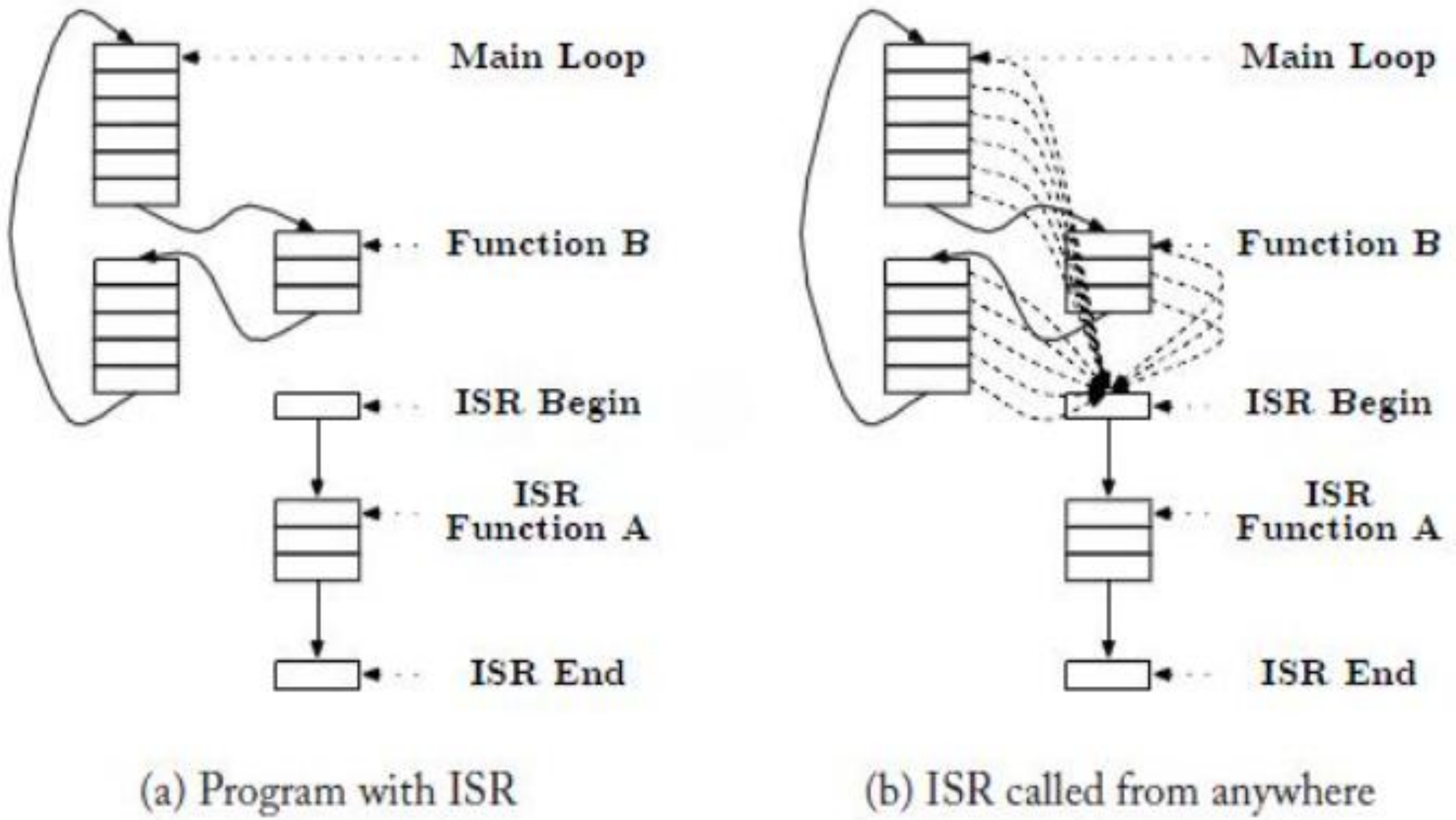


Figure: Outline of Instructions with ISR

Why do we need it?

- **Let's think about a real-life example:**
- You are heating up your food using the microwave.
- In this case, you can either:
  - Stare at the microwave while it heats up the food
  - Go about your life normally and go to the microwave when you get the signal that the food has been heated up
- **Which of the above-mentioned options do you think is more efficient?**
- Usually, we want to go about our life while the food is being heated up.
- This is because we want to make efficient use of our time.
- It is very similar for processors as well.
- We hear the ting sound from the microwave, stop what we are doing, and go to take out the heated food.
- This 'ting' sound is an interrupt.
- **Similarly, if the processor is waiting for some conditions to be fulfilled to perform a specific task (task 1):**
  - It can wait and halt other processes till the conditions for that task is fulfilled.
  - It can execute the normal instructions and halt them for a brief time once the conditions for task 1 have been fulfilled, finish task 1 and go back to executing the main routine.
- **Clearly, the 2nd option is better for maximizing the processor's resource utilization.**
- This is where interrupts come in handy.
- When the conditions for task 1 has been met, it can simply trigger an interrupt, stop the main routine to complete the task and once finished go back to the main routine.

The Main Reasons You Might Use Interrupts

- To detect pin changes (e.g., rotary encoders, button presses)
- Watchdog timer (e.g., if nothing happens after 8 seconds, interrupt me)
- **Timer interrupts - used for comparing/overflowing timers**
- SPI data transfers
- I2C data transfers
- USART data transfers
- ADC conversions (analog to digital)
- EEPROM ready for use
- Flash memory ready

Interrupt Vector

- The interrupt vectors and vector table are crucial to the understanding of hardware and software interrupts. Interrupt vectors are addresses that inform the interrupt handler as to where to find the ISR (Interrupt Service Routine, also called Interrupt Service Procedure).

- Misspelling the vector name (even wrong capitalization) will result in the ISR not being called and **it will not also result in a compiler error**.

Interrupt Service Routines

- Interrupt Service Routines are functions with no arguments. Some Arduino libraries are designed to call your own functions, so you just supply an ordinary function, e.g.,

```
// Interrupt Service Routine (ISR)
void pinChange ()
  {
   flag = true;
  }  // end of pinChange
```

- As per the datasheet, the minimal amount of time to service an interrupt is 82 clock cycles in total. Assuming a 16 MHz clock, there would be 62.5 ns time needed per clock cycle. So, the total time required is 5.125 μs.

Interrupt Service Routines

- When an ISR is entered, interrupts are disabled. Naturally, they must have been enabled in the first place, otherwise, the ISR would not be entered. However, to avoid having an ISR itself be interrupted, the processor turns interrupts off.
- When an ISR exits, then interrupts are enabled again. The compiler also generates code inside an ISR to save registers and status flags, so that whatever you were doing when the interrupt occurred will not be affected.
- However, you can turn interrupts on inside an ISR if you absolutely must.

```
// Interrupt Service Routine (ISR)
void pinChange (){
  // handle pin change here
  interrupts ();  // allow more interrupts
 }  // end of pinChange
```

Interrupt Function of Arduino

- Re-enables interrupts (after they've been disabled by noInterrupts()). Interrupts allow certain important tasks to happen in the background and are enabled by default. Some functions will not work while interrupts are disabled, and incoming communication may be ignored. Interrupts can slightly disrupt the timing of code, however, and may be disabled for particularly critical sections of code.
  - Syntax: interrupts() and noInterrupts(), all are opposite of interrupts()
  - Parameters: None
  - Returns: Nothing
- Example Code: The code enables Interrupts.

```
void setup() {}
void loop() {
  noInterrupts();
  // critical, time-sensitive code here
  interrupts();
  // other code here
}
```

Global Enable

- The main interrupt flag
- This is used to turn all the interrupts on or off
- Example: sei(); //globally enable interrupt
- Available in the Status Register (SREG): Bit 7

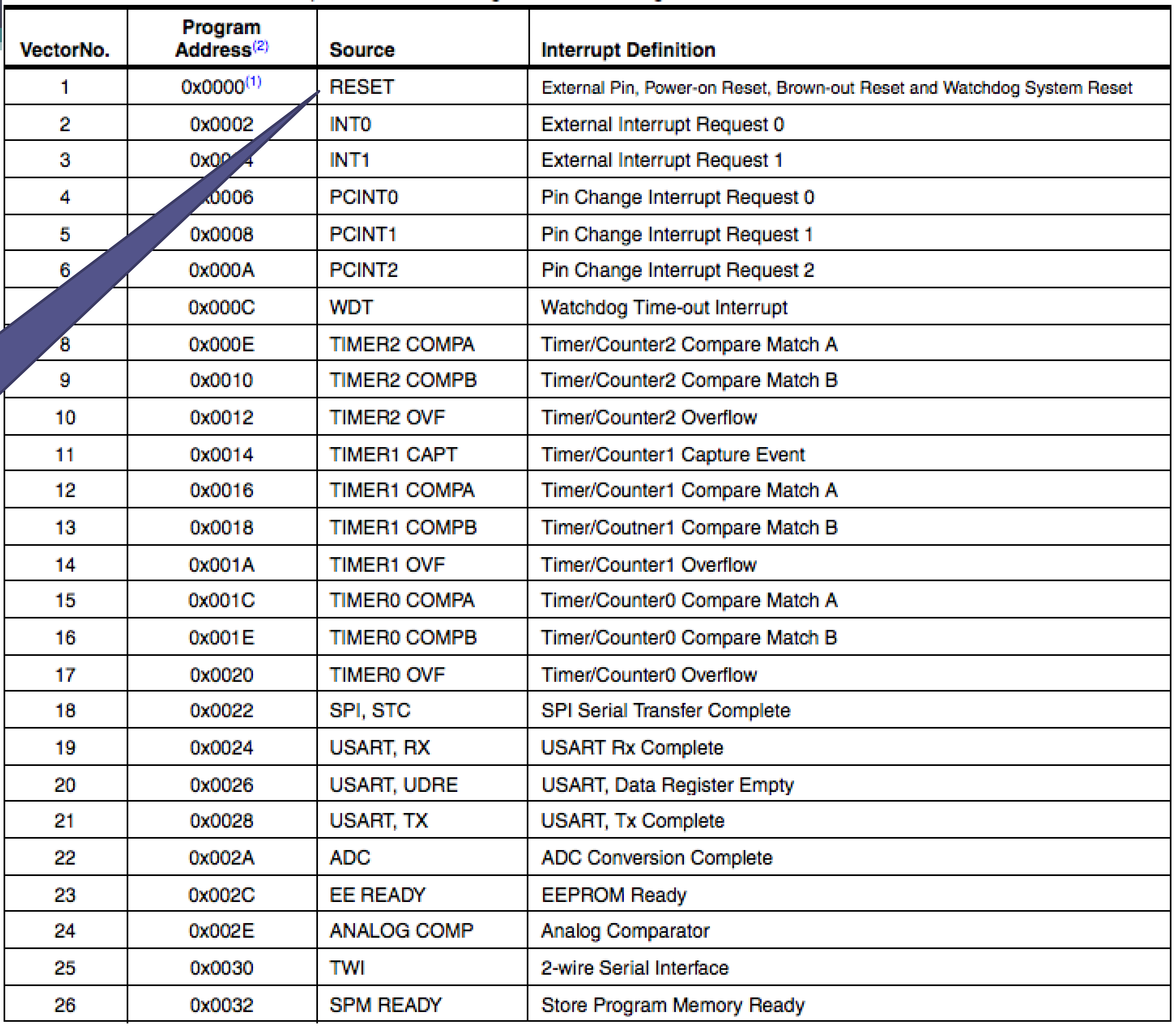
Atmega328p Interrupt Vector Table

- The ATmega328P provides support for **25 different interrupt sources**. These interrupts and the separate Reset Vector each have a separate program vector located at the lowest addresses in the Flash program memory space.

- The complete list of “Reset and Interrupt Vectors” in the ATMega328P is shown in this table. Each Interrupt Vector occupies two instruction words.

- The list also determines the priority levels of the different interrupts. The lower the address the higher the priority level. RESET has the highest priority, and next is **INT0 – the External Interrupt Request 0**.

Atmega328p Interrupt Vector Table



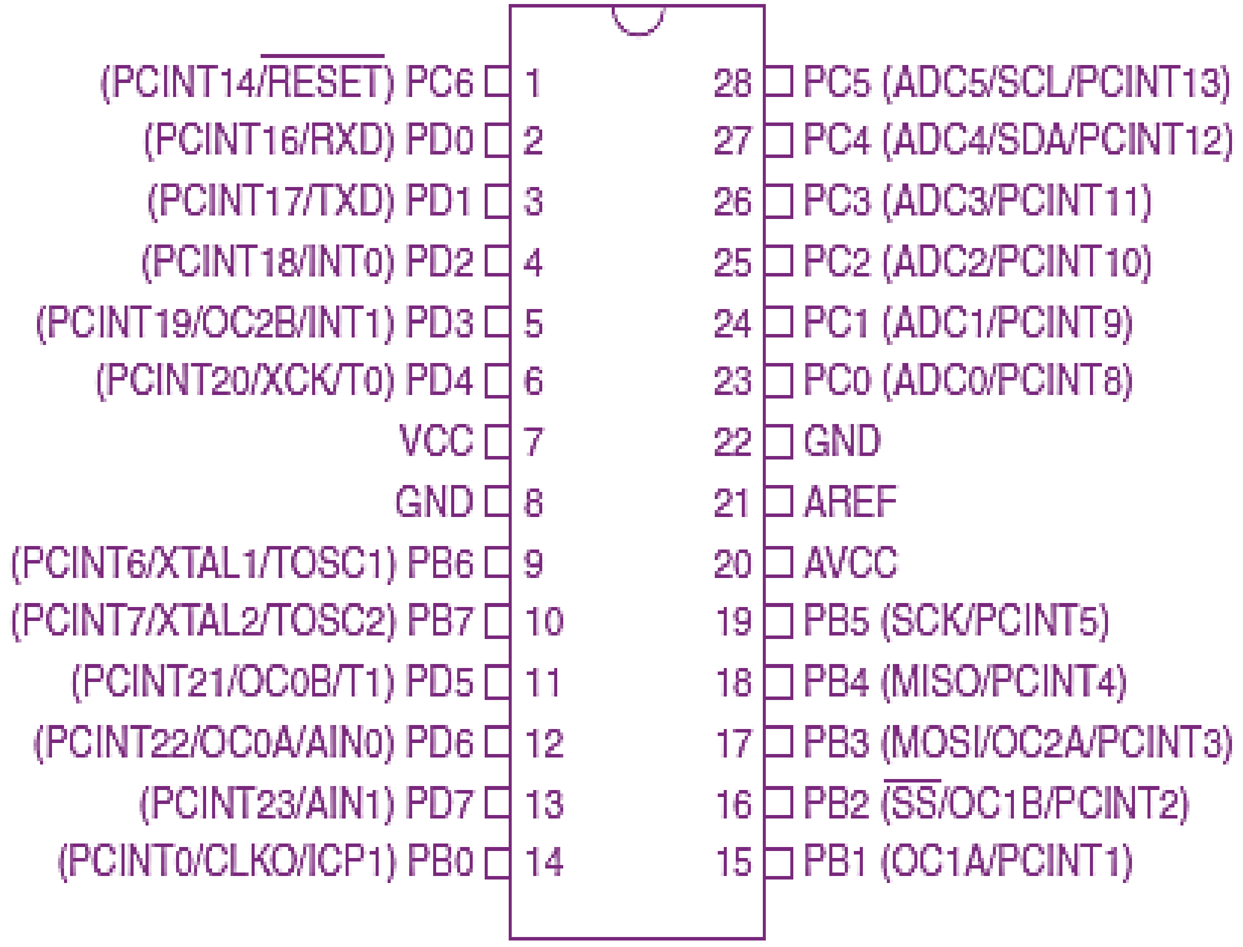
VectorNo.	Program Address[2]	Source	Interrupt Definition
1	0x0000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
[illegible]	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2 OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1 COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1 OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow
18	0x0022	SPI, STC	SPI Serial Transfer Complete
19	0x0024	USART, RX	USART Rx Complete
20	0x0026	USART, UDRE	USART, Data Register Empty
21	0x0028	USART, TX	USART, Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE READY	EEPROM Ready
24	0x002E	ANALOG COMP	Analog Comparator
25	0x0030	TWI	2-wire Serial Interface
26	0x0032	SPM READY	Store Program Memory Ready

This has the highest level of interrupt priority

List of interrupts, in priority order, for the ATmega328p:

1	Reset	
2	External Interrupt Request 0 (pin D2)	(INT0_vect)
3	External Interrupt Request 1 (pin D3)	(INT1_vect)
4	Pin Change Interrupt Request 0 (pins D8 to D13)	(PCINT0_vect)
5	Pin Change Interrupt Request 1 (pins A0 to A5)	(PCINT1_vect)
6	Pin Change Interrupt Request 2 (pins D0 to D7)	(PCINT2_vect)
7	Watchdog Time-out Interrupt	(WDT_vect)
8	Timer/Counter2 Compare Match A	(TIMER2_COMPA_vect)
9	Timer/Counter2 Compare Match B	(TIMER2_COMPB_vect)
10	Timer/Counter2 Overflow	(TIMER2_OVF_vect)
11	Timer/Counter1 Capture Event	(TIMER1_CAPT_vect)
12	Timer/Counter1 Compare Match A	(TIMER1_COMPA_vect)
13	Timer/Counter1 Compare Match B	(TIMER1_COMPB_vect)
14	Timer/Counter1 Overflow	(TIMER1_OVF_vect)
15	Timer/Counter0 Compare Match A	(TIMER0_COMPA_vect)
16	Timer/Counter0 Compare Match B	(TIMER0_COMPB_vect)
17	Timer/Counter0 Overflow	(TIMER0_OVF_vect)
18	SPI Serial Transfer Complete	(SPI_STC_vect)
19	USART Rx Complete	(USART_RX_vect)
20	USART, Data Register Empty	(USART_UDRE_vect)
21	USART, Tx Complete	(USART_TX_vect)
22	ADC Conversion Complete	(ADC_vect)
23	EEPROM Ready	(EE_READY_vect)
24	Analog Comparator	(ANALOG_COMP_vect)
25	2-wire Serial Interface (I2C)	(TWI_vect)
26	Store Program Memory Ready	(SPM_READY_vect)

Interrupt Pin Configuration for the ATmega328p:



Trigger

- An asynchronous event that causes the interrupt
- For example, the push button press during experiment 4 in our MES Lab can be a trigger.
- It can cause an interrupt that is registered by the module and is reacted to.
- What if all the conditions are not met but a trigger flag is set?
- In this case, rather than the request being dismissed, it is held pending and postponed until a later time.
- **What happens after a trigger?** Once the interrupt is triggered and processed, the interrupt **flag is cleared**.
- Clearing the interrupt flag is called **acknowledgment**.

Interrupt Service Routine (ISR)

- The module that is executed when hardware requests an interrupt.
- There may be 1 large ISR handling all the interrupt requests, or many small ISRs handling the many interrupts (interrupt vectors).

Example:

ISR(TIMER0_OVF_vect) // enabling overflow vector inside Timer0 using an ISR

ISR(TIMER0_COMPA_vect) // This is the Timer0 Compare ‘A’ interrupt service routine.

Remember, the ISR is a separate routine and requires a separate flowchart to represent.

General Rules for ISR

- The ISR should execute as fast as possible.
  - The interrupt should occur when it’s time to perform the required action
  - The ISR should perform the action
  - The ISR should end and return to the main function right away.
- Placing backward branches (busy, wait, iterations) inside the ISR should be avoided.
- The percentage of time spent executing ISR should be small when compared to the time between interrupt triggers.

General Rules while writing an Interrupt Service Routine (ISR):

- Keep it short
- Don't use delay ()
- Don't do serial prints
- Make variables shared with the main code volatile
- Variables shared with the main code may need to be protected by "critical sections" (see below)
- Don't try to turn interrupts off or on

What are "volatile" variables?

- Variables shared between ISR functions and normal functions should be declared "**volatile**". This tells the compiler that such **variables might change at any time**, and thus the **compiler must reload the variable** whenever you reference it, rather than relying upon a copy it might have in a processor register.

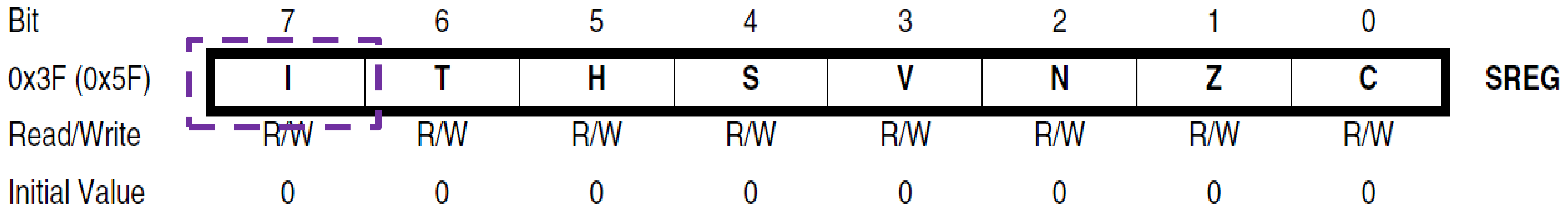
```
volatile boolean flag;

// Interrupt Service Routine (ISR)
void isr()
{
 flag = true;
}  // end of isr
```

What are "volatile" variables?

```
void setup ()
{
  attachInterrupt (digitalPinToInterrupt (2), isr, CHANGE);  // attach interrupt handler
}  // end of setup

void loop ()
{
  if (flag)
    {
    // interrupt has occurred
    }
}  // end of loop
```

Atmega328p Interrupt Processing

- **When an interrupt occurs, the microcontroller completes the current instruction and stores the address of the next instruction on the stack.**
- It also turns off the interrupt system to prevent further interrupts while one is in progress. This is done by clearing the SREG **Global Interrupt Enable I-bit**.
- The Interrupt flag bit is cleared for **Type 1 Interrupts only** (see the next Slide for Type definitions).



Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Atmega328p Interrupt Processing

- **The execution of the ISR** is performed by loading the **beginning address** of the ISR specific for that interrupt **into the program counter**. The processor starts running the ISR.
- Execution of the ISR continues until the **Return from Interrupt Instruction** (**RETI**) is encountered. The **SREG I-bit** is automatically **set** when the **RETI** instruction is executed (i.e., **Interrupts enabled**).
- When the processor exits from an interrupt, it will always return to the interrupted program and execute one more instruction before any pending interrupt is served.
- The **Status Register is not automatically stored** when entering an interrupt routine, nor restored when returning from an interrupt routine. This must be handled by software.

Atmega328p Interrupt Processing - Type 1

- The user software can write logic one to the I-bit to enable nested interrupts. All enabled interrupts can then interrupt the current interrupt routine.
  - The SREG I-bit is automatically set to logic one when a **Return from Interrupt Instruction (RETI)** is executed.
- There are basically **two types of interrupts:**
  - The **first type (Type 1)** is triggered by an event that sets the Interrupt Flag. For these interrupts, the Program Counter is vectored to the actual Interrupt Vector to execute the interrupt handling routine, and **hardware clears the corresponding Interrupt Flag**.

Atmega328p Interrupt Processing - Type 1

- If the same interrupt condition occurs while the corresponding interrupt enables bit is cleared, the Interrupt Flag will be set and remembered until the interrupt is enabled, or the flag is cleared by software (interrupt canceled).
- Interrupt Flag can be cleared by writing a logic one to the flag bit position(s) to be cleared.
    - If one or more interrupt conditions occur while the Global Interrupt Enable (SREG I) bit is cleared, the corresponding Interrupt Flag(s) will be set and remembered until the Global Interrupt Enable bit is set on return (**RETI**) and will then be executed by order of priority.

Atmega328p Interrupt Processing - Type 2

- The **second type (Type 2)** of interrupts will trigger as long as the interrupt condition is present. These interrupts do not necessarily have Interrupt Flags. If the interrupt condition disappears before the interrupt is enabled, the interrupt will not be triggered.

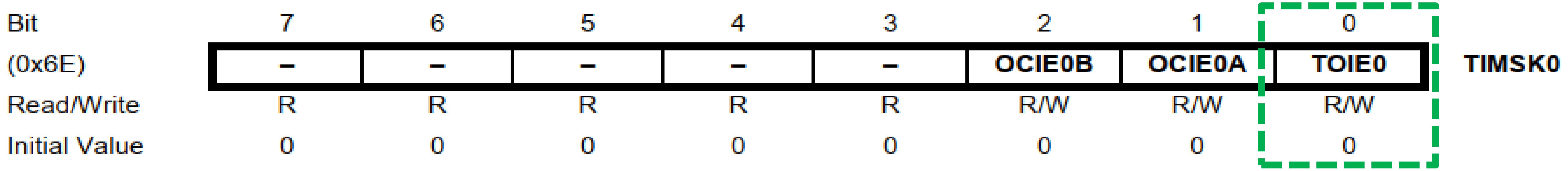
TIMSK0 - Timer/Counter0 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
(0x6E)	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0	TIMSK0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bits 7..3 – Res: Reserved Bits**
  These bits are reserved bits in the ATmega328P and will always read as zero.

- **Bit 2 – OCIE0B: Timer/Counter Output Compare Match B Interrupt Enable**
  When the OCIE0B bit is written to one, and the I-bit in the Status Register is set, the Timer/Counter Compare Match B Interrupt is Enabled. The corresponding interrupt is executed if Compare Match in Timer/Counter0 occurs, i.e., when the OCF0B bit is set in the Timer/Counter Interrupt Flag Register, TIFR0.

TIMSK0 - Timer/Counter0 Interrupt Mask Register



Bit	7	6	5	4	3	2	1	0	
(0x6E)	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0	TIMSK0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

• **Bit 1 – OCIE0A: Timer/Counter0 Output Compare Match A: Interrupt Enable**

When the OCIE0A bit is written to one, and the I-bit in the Status Register is set, the Timer/Counter0 Compare Match A interrupt is enabled. The corresponding interrupt is executed if a Compare Match in Timer/Counter0 occurs, i.e., when the OCF0A bit is set in the Timer/Counter 0 Interrupt Flag Register, TIFR0.

• **Bit 0 – TOIE0: Timer/Counter0 Overflow Interrupt Enable**

When the TOIE0 bit is written to one, and the I-bit in the Status Register is set, the Timer/Counter0 Overflow interrupt is enabled. The corresponding interrupt is executed if an overflow in Timer/Counter0 occurs, i.e., when the TOV0 bit is set in the Timer/Counter 0 Interrupt Flag Register – TIFR0

TIFR0 - Timer/Counter0 Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x15 (0x35)	–	–	–	–	–	OCF0B	OCF0A	TOV0	TIFR0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bits 7..3 – Res: Reserved Bits**

  These bits are reserved bits in the ATmega328P and will always read as zero.

- **Bit 2 – OCF0B: Timer/Counter 0 Output Compare B Match Flag**

  The OCF0B bit is set when a Compare Match occurs between the Timer/Counter and the data in OCR0B – Output Compare Register0 B. OCF0B is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, OCF0B is cleared by writing a logic one to the flag. When the I-bit in SREG, OCIE0B (Timer/Counter Compare B Match Interrupt Enable), and OCF0B are set, the Timer/Counter Compare Match Interrupt is executed.

TIFR0 - Timer/Counter0 Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x15 (0x35)	–	–	–	–	–	OCF0B	OCF0A	TOV0	TIFR0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 1 – OCF0A: Timer/Counter 0 Output Compare A Match Flag**

  The OCF0A bit is set when a Compare Match occurs between the Timer/Counter0 and the data in OCR0A – Output Compare Register0. OCF0A is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, OCF0A is cleared by writing a logic one to the flag. When the I-bit in SREG, OCIE0A (Timer/Counter0 Compare Match Interrupt Enable), and OCF0A are set, the Timer/Counter0 Compare Match Interrupt is executed.

TIFR0 - Timer/Counter0 Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x15 (0x35)	–	–	–	–	–	OCF0B	OCF0A	TOV0	TIFR0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 0 – TOV0: Timer/Counter0 Overflow Flag**

  The bit TOV0 is set when an overflow occurs in Timer/Counter0. TOV0 is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, TOV0 is cleared by writing a logic one to the flag. When the SREG I-bit, TOIE0 (Timer/Counter0 Overflow Interrupt Enable), and TOV0 are set, the Timer/Counter0 Overflow interrupt is executed.

  The setting of this flag is dependent on the WGM02:0 bit setting

TIMSK1 - Timer/Counter1 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
(0x6F)	–	–	ICIE1	–	–	OCIE1B	OCIE1A	TOIE1	TIMSK1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

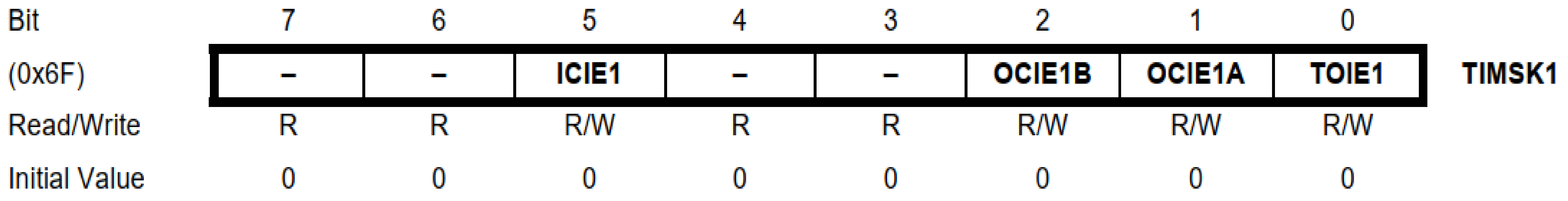
- **Bit 7, 6 – Res: Reserved Bits**

These bits are unused bits in the ATmega48P/88P/168P/328P and will always read as zero.

- **Bit 5 – ICIE1: Timer/Counter1, Input Capture Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Input Capture interrupt is enabled. The corresponding Interrupt Vector is executed when the ICF1 Flag, located in TIFR1, is set.

TIMSK1 - Timer/Counter1 Interrupt Mask Register



Bit	7	6	5	4	3	2	1	0	
(0x6F)	–	–	ICIE1	–	–	OCIE1B	OCIE1A	TOIE1	TIMSK1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 4, 3 – Res: Reserved Bits**

These bits are unused bits in the ATmega328P and will always read as 0.

- **Bit 2 – OCIE1B: Timer/Counter1, Output Compare B Match Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Output Compare B Match interrupt is enabled. The corresponding Interrupt Vector is executed when the OCF1B Flag, located in TIFR1, is set.

TIMSK1 - Timer/Counter1 Interrupt Mask Register

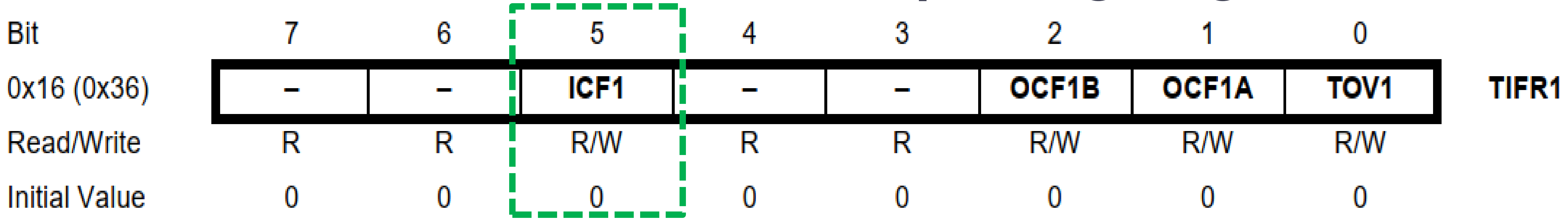
- **Bit 1 – OCIE1A: Timer/Counter1, Output Compare A Match Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Output Compare A Match interrupt is enabled. The corresponding Interrupt Vector is executed when the OCF1A Flag, located in TIFR1, is set.

- **Bit 0 – TOIE1: Timer/Counter1, Overflow Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Overflow interrupt is enabled. The corresponding Interrupt Vector is executed when the TOV1 Flag, located in TIFR1, is set.

TIFR1 - Timer/Counter1 Interrupt Flag Register



Bit	7	6	5	4	3	2	1	0	
0x16 (0x36)	–	–	ICF1	–	–	OCF1B	OCF1A	TOV1	TIFR1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7, 6 – Res: Reserved Bits**

These bits are unused bits in the ATmega328P and will always read as zero.

- **Bit 5 – ICF1: Timer/Counter1, Input Capture Flag**

This flag is set when a capture event occurs on the ICP1 pin. When the Input Capture Register (ICR1) is set by the WGM13:0 to be used as the TOP value, the ICF1 Flag is set when the counter **reaches the TOP value**. ICF1 is automatically cleared when the **Input Capture Interrupt Vector is executed**. Alternatively, ICF1 can be cleared by writing a logic one to its bit location.

TIFR1 - Timer/Counter1 Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x16 (0x36)	–	–	ICF1	–	–	OCF1B	OCF1A	TOV1	TIFR1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

• **Bit 4, 3 – Res: Reserved Bits**

These bits are unused bits in the ATmega328P and will always read as zero.

• **Bit 2 – OCF1B: Timer/Counter1, Output Compare B Match Flag**

This flag is set in the timer clock cycle after the counter (TCNT1) value matches the Output Compare Register B (OCR1B). Note that a Forced Output Compare (FOC1B) strobe will not set the OCF1B Flag. OCF1B is automatically cleared when the Output Compare Match B Interrupt Vector is executed. Alternatively, OCF1B can be cleared by writing a logic one to its bit location.

TIFR1 - Timer/Counter1 Interrupt Flag Register

• **Bit 1 – OCF1A: Timer/Counter1, Output Compare A Match Flag**

This **flag is set** in the timer clock cycle after the counter (**TCNT1**) **value matches** the Output Compare Register A (**OCR1A**). Note that a Forced Output Compare (FOC1A) strobe will not set the OCF1A Flag. **OCF1A is automatically cleared** when the **Output Compare Match A InteOCF1A can be cleared** by writing a **logic one** to its bit location**rrupt Vector** is executed. Alternatively,.

• **Bit 0 – TOV1: Timer/Counter1, Overflow Flag**

The setting of this flag is dependent on the WGM13:0 bits setting. In Normal and CTC modes, the TOV1 Flag is set when the timer overflows.

Flag behavior when using another WGM13:0 bit setting. TOV1 is automatically cleared when the Timer/Counter1 Overflow Interrupt Vector is executed. Alternatively, TOV1 can be cleared by writing a **logic one to its bit location**.

500 ms Blink Example Code using Timer1 Interrupts

Timer 1 is used here. The values of TCCR1A and 1B are reset to 0 to make sure everything is clear. TCCR1B was set equal to 00000100 for a prescalar of 256. If you want to set ones, an OR operation can be used. If you want to set zeros, an AND operation can be used. The compare match mode for the OCR1A register is enabled. For that, the OCIE1A bit was set to be a 1 and that's from the TIMSK1 register. So, we equal that to OR and this byte 00000010.
The OCR1A register was set to 31250 value so that we will have an interruption each 500ms. Each time the interrupt is triggered, we go to the related ISR vector. Since we have 3 timers, **we have 6 ISR vectors**, two for each timer and they have these names:

TIMER1_COMPA_vect,	TIMER2_COMPA_vect,	TIMER0_COMPA_vect,
TIMER1_COMPB_vect,	TIMER2_COMPB_vect,	TIMER0_COMPB_vect

500 ms Blink Example Code using Timer1 Interrupts

Since the timer count value is greater than 255, we can't use Timer0 or Timer2. So, **Timer1** and its **Output Compare Register A (OCR1A)**, where timer count value of **31,250** need be stored, was used so we must use the ISR **TIMER1_COMPA_vect**. So, below the void loop, the interrupt service routine is defined. Inside this interruption, the state of the LED was inverted, and a digital write was created. But first, **the timer value** was reset. Otherwise, it will continue to count up to its maximum value. So, for each 500 ms, this code will run and invert the LED state and that creates a blink of the LED connected to **pin PB5**, for example.

500 ms Blink Example Code using Timer1 Interrupts and Pre-scalar values of 256

- We use the **pre-scalar** value of 256 to reduce the timer clock frequency to 16 MHz/256 = 62.5 kHz, with a **new timer clock period = 1/62.5 kHz = 16 µs**. Thus, the required timer count = (500,000 µs/16 µs) – 1 = 31,250 – 1 = 31,249. So, this is the value that we should store in the OCR register.

- The program for Arduino is as follows:

```
bool LED_State = 'True';

void setup() {
        pinMode(13, OUTPUT); // An LED is connected with this port
        cli();               // stop interrupts till we make the settings
        TCCR1A = 0;          // Reset the entire A and B registers of Timer1 to make sure that
        TCCR1B = 0;          // we start with everything disabled.
```

500 ms Blink Example Code using Timer1 Interrupts and Pre-scalar values of 256

```
        TCCR1B = 0b00000100; // Set CS12 bit of TCCR1B to 1 to get a prescalar value of 256.
        TIMSK1 = 0b00000010; // Set OCIE1A bit to 1 to enable compare match mode of A reg.
        OCR1A = 31250;   // We set the required timer count value in the compare register, A
        sei();           // Enable back the interrupts
}

void loop() {
        // put your main code here, to run repeatedly.
}

// With the settings above, this ISR will trigger each 500 ms.
ISR(TIMER1_COMPA_vect){
        TCNT1 = 0;       // First, set the timer back to 0 so that it resets for the next interrupt
        LED_State = !LED_State;        // Invert the LED State
        digitalWrite(13, LED_State);   // Write this new state to the LED connected to pin PB5
}
```

Example 2: Blink Program using Timer:

An Arduino microcontroller code is to be written for a 600 ms interval LED blinking system using Timer_n Interrupts and a Pre-scalar value of 1024. LED is connected to pin 5 of the board.
Compute the value to be loaded into the OCRnB register. The clock frequency is 16 MHz. The Timer ISR vector is TIMERn_COMPx_vect. [n = 0-2; x = A/B]. **Determine the contents** of the following program.

```
bool LED_State = 'True';

void setup() {
        pinMode(5, OUTPUT);
        cli();
        TCCR__A = 0b____________;      |COMnA1|COMnA0| COMnB1| COMnB0| - | - |WGMn1|WGMn0|
        TCCR__B = 0b____________;      |FOCnA|FOCnB| - | - |WGMn2|CSn2| CSn1| CSn0|
TCCR__B = 0b____________;
        TIMSK___ = 0b____________;    | - | - | ICIEn | - | - |OCIEnB|OCIEnA|TOIEn|
        OCR_____ = _______________;   | - | - | - | - | - | - | - | - |
        sei();   }

void loop() {
        // main code here, to run repeatedly.   }

ISR(TIMER____COMP____vect) {
        TCNT__ = 0x_________________;
        LED_State = !LED_State;
        DigitalWrite(_______, _________);   }
```

Answer:

Required delay, t_d = 600 ms = 600000 μs
Given system frequency, f = 16 MHz
So, clock period, T = 1/frequency = 1/16 MHz = 0.0625 μs
As such, Timer Count, TC = Required delay/(clock period×pre-scaler value) – 1
= 600000/(0.0625×1024) – 1 = 9375 – 1 = 9374

But Timer 0 can count up to 255 and Timer 1 can count up to 65,535, so Timer 1 is suitable for this application. So, the Timer Count value (9375) should be loaded into the OCR1B register.

The given program is corrected/filled up as follows (red marked).

Answer:

```
bool LED_State = 'True';

void setup() {
        pinMode(5, OUTPUT);
        cli();
        TCCR1A = 0b00000000;        |COMnA1|COMnA0| COMnB1| COMnB0| - | - |WGMn1|WGMn0|
        TCCR1B = 0b00000000;        |FOCnA|FOCnB| - | - |WGMn2|CSn2| CSn1| CSn0|
TCCR1B = 0b00000101;
        TIMSK1 = 0b00000100;          | - | - | - | - | - |OCIE0B|OCIE0A|TOIE0|
        OCR1B = 9375;                 | - | - | - | - | - | - | - | - |
        sei();   }

void loop() {
        // main code here, to run repeatedly.    }

ISR(TIMER1_COMPB_vect){
        TCNT1 = 0x0000;
        LED_State = !LED_State;
        digitalWrite(5, LED_State);   }
```

Example 3: Blink Program using Timer:

Write an Arduino microcontroller code for a 400 ms interval LED blinking system using Timern Interrupts and a Pre-scalar value of 1024. LED is connected to pin 5 of the board. Compute the value to be loaded into the OCRnA register. [n = 0-2].

Answer:

We use the pre-scalar value of 1024 to reduce the timer clock frequency to 16 MHz/1024 = 15.625 kHz, with a new timer clock period = 1/15.625 kHz = 64 µs. Thus, the required timer count is computed as-

$$Timer\ Count = \frac{Requied\ Delay}{Clock\ peried\ after\ prescaling\ the\ frequency} - 1$$

$$= \frac{400{,}000\ \mu s}{64\ \mu s} - 1 = 6{,}250 - 1 = 6{,}249.$$

So, this is the value that we should store in the OCR1A register as it is >255.

Example 3: Blink Program using Timer:

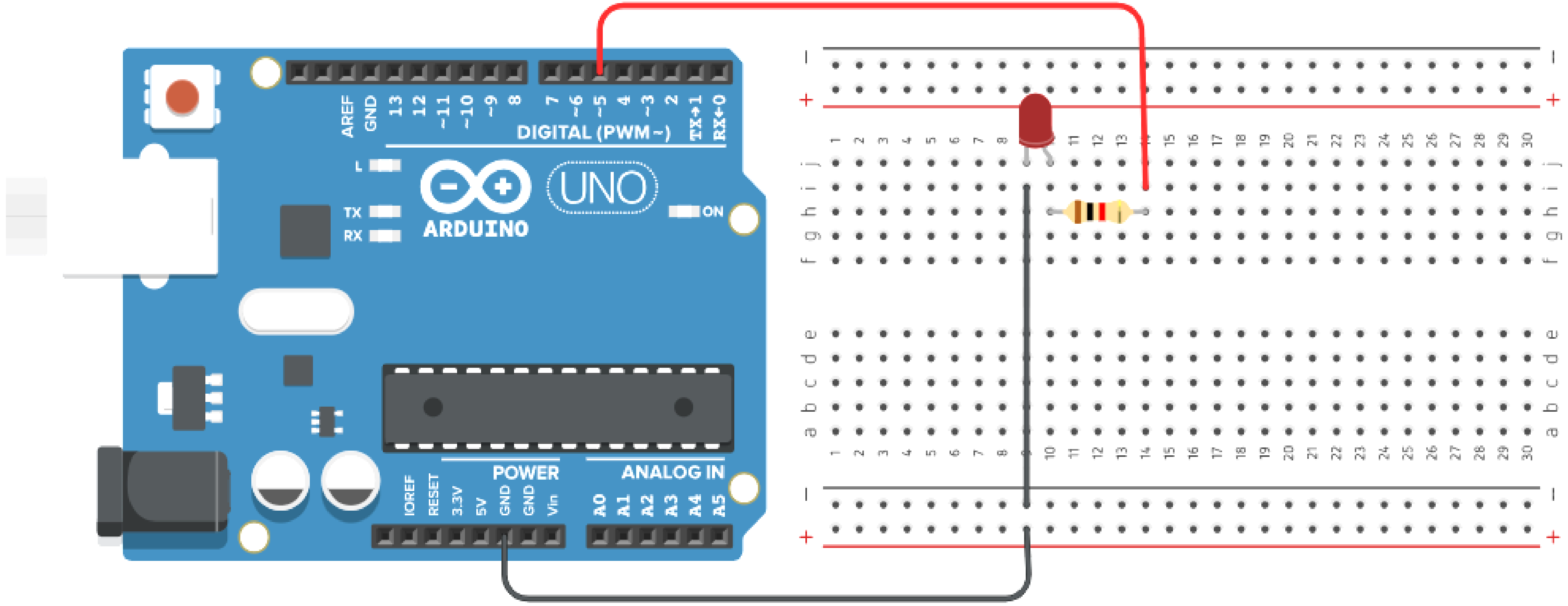
The program for Arduino is as follows:

```
boolean LED_State = HIGH;

void setup() {
        pinMode(5, OUTPUT);
        cli();                  // stop interrupts till we make the settings
        TCCR1A = 0; // Reset the entire A and B registers of Timer1 to make sure that
        TCCR1B = 0; // we start with everything disabled.
        TCCR1B = 0b00000101; // Set CS12:10 bits of TCCR1B to 101 to get a prescalar value of 1024.
        TIMSK1 = 0b00000010; // Set OCIE1A bit to 1 to enable compare match mode of A register
        OCR1A = 6250;           // We set the required timer count value in the compare register, A
        sei();                  // Enable back the interrupts
}

void loop() {
        // put your main code here, to run repeatedly.
}
```

Example 3: Blink Program using Timer:

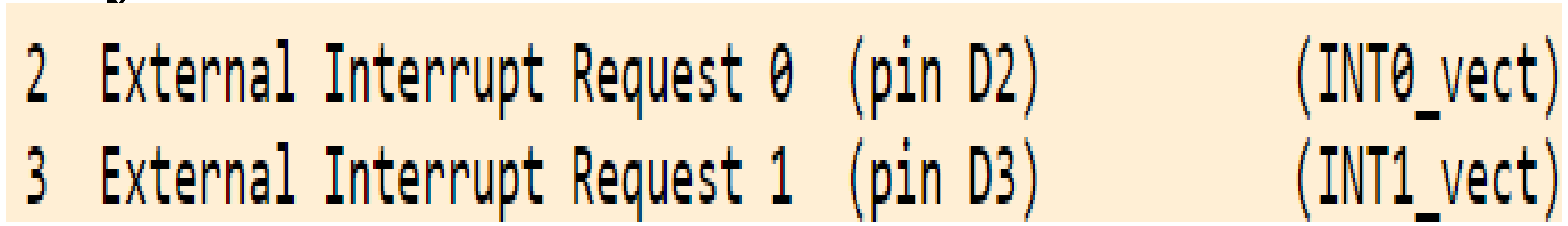
```
// With the settings above, this ISR will trigger each 400 ms.
ISR(TIMER1_COMPA_vect) {
        TCNT1 = 0;          // First, set the timer back to 0 so that it resets for the next interrupt
        LED_State = !LED_State;    // Invert the LED State
        digitalWrite(5, LED_State);  // Write this new state to the LED connected to pin 5
}
```

The circuit connection diagram is given below for this program:



External Interrupts

- External interrupts triggered by:
  - INT0 or INT1 pins
  - Any of the PCINT23:0 pins



```
2  External Interrupt Request 0  (pin D2)          (INT0_vect)
3  External Interrupt Request 1  (pin D3)          (INT1_vect)
```

- INT0 and INT1 are mapped with the pins PD2 and PD3 on ATMega328P
- PD2 and PD3 are mapped with digital pins 2 and 3 (GPIO) on the Arduino Uno board
- PCINT23:0 are mapped with ports B, C, and D on the Arduino Uno board
- Implies any of the I/O pins on the board can be configured to handle interrupts
- These interrupts can be used to wake up the processor from sleep modes

External Interrupts

- INT0 and INT1 are referred to as 'External Interrupts'
  - Can be triggered by a rising edge, falling edge, logical change, or low level
- PCINT23:0 are referred to as 'Pin Change Interrupts'
- External interrupts have a higher priority
- External interrupts have their own interrupt vectors
- Pin change interrupts share interrupt vectors
  - The interrupt handler has to decide which pin the interrupt originated from

EICRA – External Interrupt Control Register A

- Contains control bits for interrupt sense control

Bit	7	6	5	4	3	2	1	0	
(0x69)	–	–	–	–	ISC11	ISC10	ISC01	ISC00	EICRA
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bits 7:4 – Reserved bits – always read as 0
- Bits 3:2 – ISC11, ISC10 - Interrupt Sense Control 1
- Bits 1:0 – ISC01, ISC00 – Interrupt Sense Control 0
- ISC1x corresponds to INT1
- ISC0x corresponds to INT0

EICRA – External Interrupt Control Register A

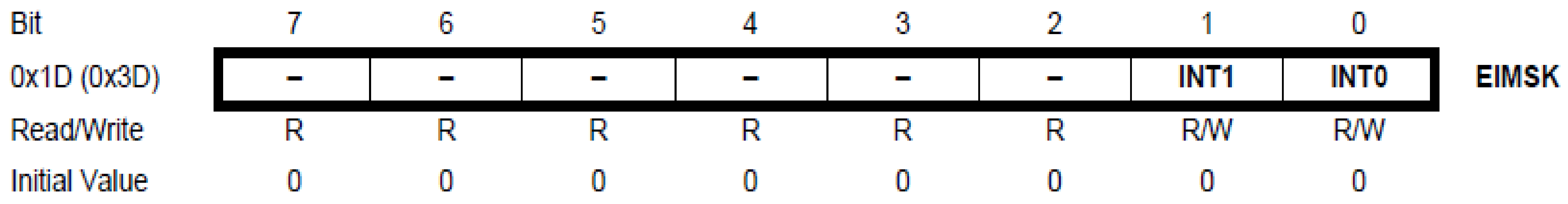
- External Interrupts activated by the external pin INT1 or INT0 if the SREG I-flag and the corresponding interrupt mask are set.
- Level/edge on INT1/INT0 pin which activates interrupts:

ISCx1	ISCx0	Description
0	0	LOW level of INT1/INT0 generates interrupt request
0	1	Any logical/level CHANGE on INT1/INT0 generates interrupt request
1	0	FALLING edge of INT1/INT0 generates interrupt request
1	1	RISING edge of INT1/INT0 generates interrupt request

EICRA – External Interrupt Control Register A

- Value of the pin is sampled before detecting the edges
- If edge or toggle interrupt is selected:
    - Pulses longer than 1 clock period will generate an interrupt
    - Shorter pulses are not guaranteed to generate an interrupt
- If low-level interrupt is selected, to generate interrupt:
    - Low level must be held until the completion of currently executing instruction

EIMSK – External Interrupt Mask Register



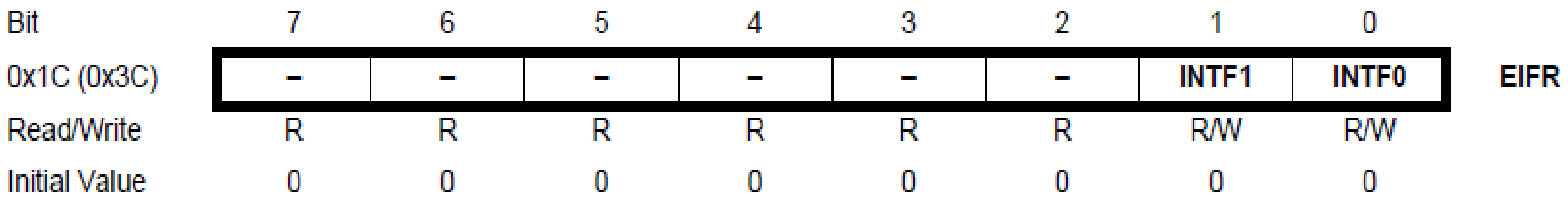
Bit	7	6	5	4	3	2	1	0	
0x1D (0x3D)	–	–	–	–	–	–	INT1	INT0	EIMSK
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bits 7:2 – Reserved – always read as 0
- Bit 1 – INT1: External Interrupt Request 1 Enable
- Bit 0 – INT0: External Interrupt Request 0 Enable

EIMSK – External Interrupt Mask Register

- When INT1/INT0 bit is enabled, and I bit (bit 7) of SREG is enabled:
  - External pin interrupt is activated
  - ISCx1, and ISCx0 pins in EICRA define whether the external interrupt will be activated on:
    - Rising edge
    - Falling edge
    - Logic change
    - Low level
- Activity on the pin will cause an interrupt even if the pin is configured as an output
- Executed from INT1/INT0 interrupt vector

EIFR – External Interrupt Flag Register



Bit	7	6	5	4	3	2	1	0	
0x1C (0x3C)	–	–	–	–	–	–	INTF1	INTF0	EIFR
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

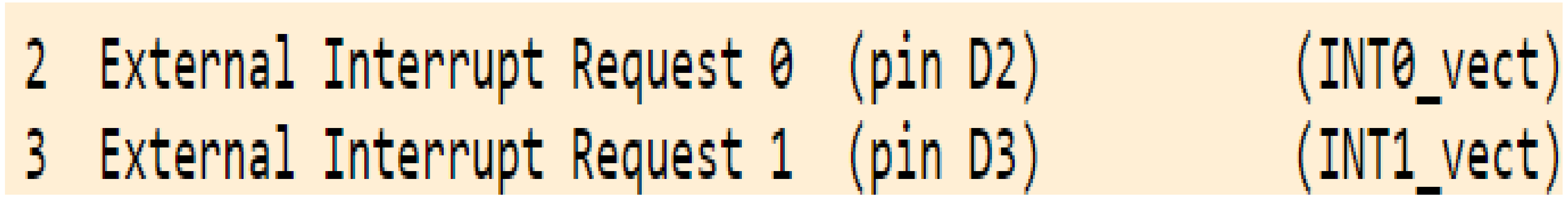
- Bits 7:2 – Reserved – always read as 0
- Bit 1 – INTF1: External Interrupt Flag 1 will be set if INT1 interrupt occurs; cleared once the ISR is executed.
- Bit 0 – INTF0: External Interrupt Flag 0 will be set if INT0 interrupt occurs; cleared once the ISR is executed.

EIFR – External Interrupt Flag Register

- When the INTx pin triggers an interrupt request, INTFx is set
- If I bit in SREG and INTx bit in EIMSK are set, MCU will jump to a relevant interrupt vector
- INTFx cleared when:
  - Interrupt routine is executed
  - A logical 1 is written (cancel any falling interrupts)
  - Always cleared when INTx is configured as a level interrupt

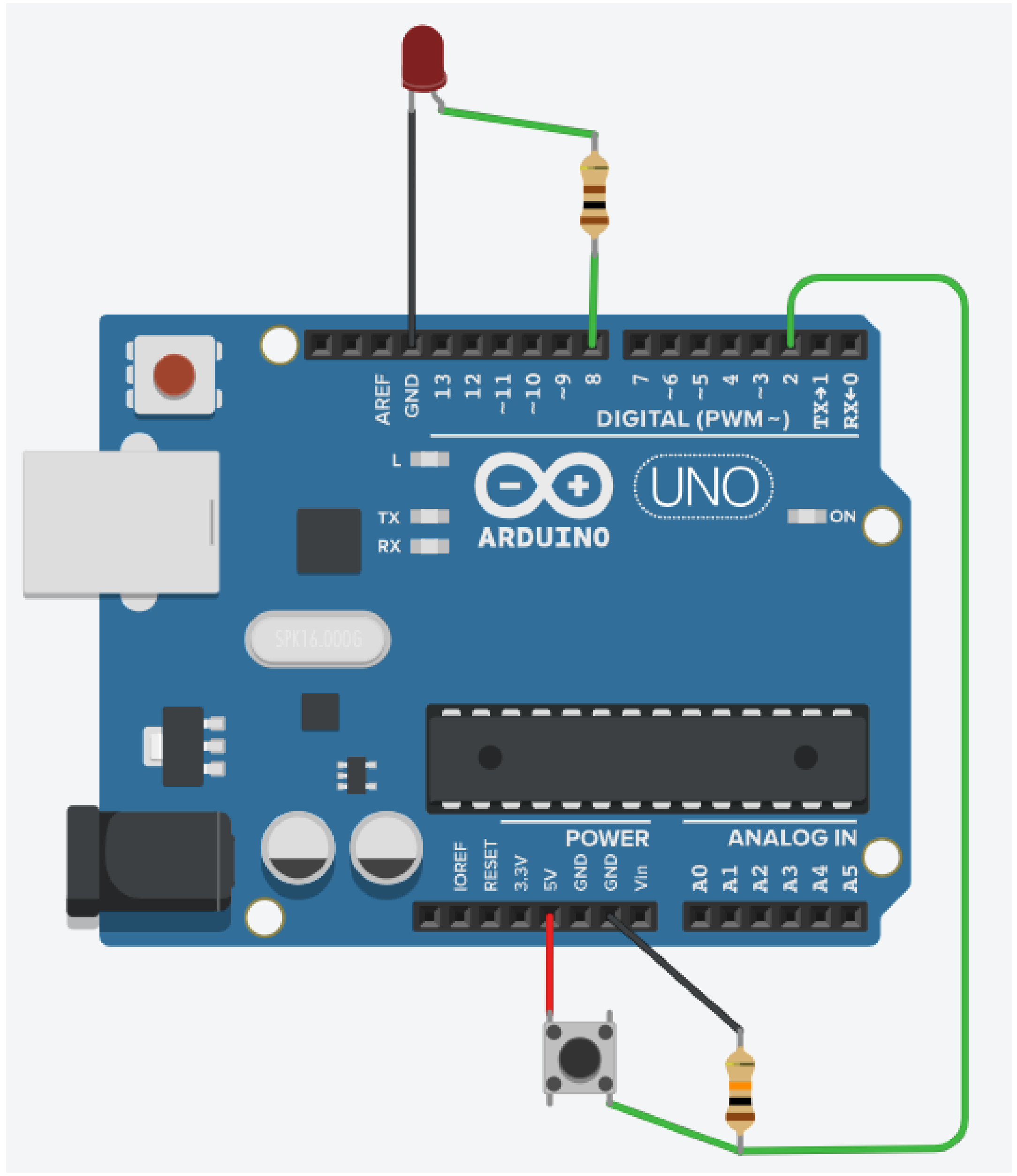
External Interrupt related Program

```
volatile boolean Output_Signal = true;

void isr() {
 Output_Signal = !Output_Signal;
 digitalWrite(8, Output_Signal);  }

void setup () {
  cli();
  pinMode(8, OUTPUT);  // an LED is connected to this pin, 8
  attachInterrupt (digitalPinToInterrupt(2), isr, RISING); // a push switch is connected to this pin, 2
  EICRA = 0b00000011;
  sei();
  EIMSK = 0b00000001;   }

void loop () {
  digitalWrite(8, Output_Signal);  }
```



```
2  External Interrupt Request 0  (pin D2)        (INT0_vect)
3  External Interrupt Request 1  (pin D3)        (INT1_vect)
```

Circuit Diagram of External Interrupt related Program



- When the INT0 pin triggers an interrupt request, INTF0 is set
- If I bit in SREG and INT0 bit in EIMSK are set, MCU will jump to the interrupt vector
- INTF0 cleared when:
  - Interrupt routine is executed
  - A logical 1 is written (cancel any falling interrupts)
  - Always cleared when INT0 is configured as a level interrupt

PCICR – Pin Change Interrupt Control Register

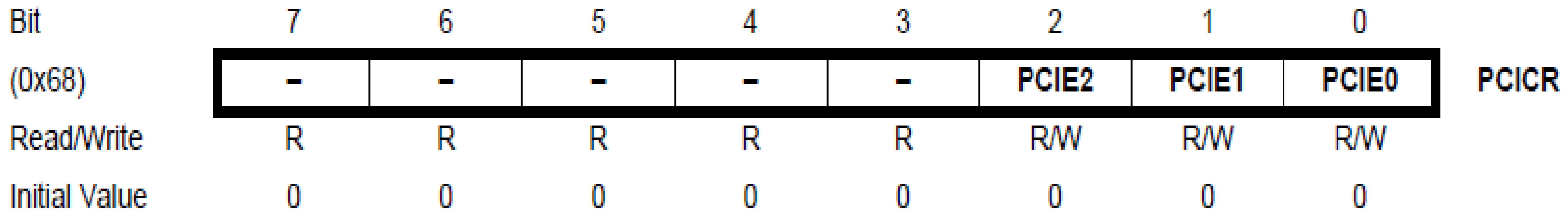
We can initialize, configure, and control Arduino PCINT (pin change interrupts) using the associated registers as stated in the datasheet. The PCINT-associated registers are as follows:

PCICR: Pin Change Interrupt Control Register, To enable/disable the 3 global PCINT interrupt signals (PCIE0, PCIE1, and PCIE2).

PCIFR: Pin Change Interrupt Flag Register, To read or clear the 3 global PCINT interrupt flag bits (PCIF0, PCIF1, and PCIF2).

PCMSK(0-2): Pin Change Mask Registers, To enable/disable pin-change interrupts for each pin individually. Each port of pins (B, C, and D) has a separate PCMSKx register (0, 1, and 2).

PCICR – Pin Change Interrupt Control Register



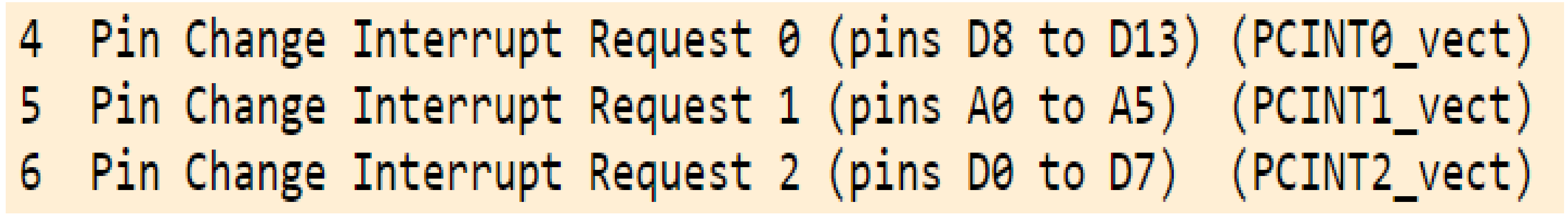
Bit	7	6	5	4	3	2	1	0	
(0x68)	–	–	–	–	–	PCIE2	PCIE1	PCIE0	PCICR
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bits 7: 3 – Reserved – always read as 0
- Bit 2 – PCIE2 – Pin Change Interrupt Enable 2
- Bit 1 – PCIE1 – Pin Change Interrupt Enable 1
- Bit 0 – PCIE0 – Pin Change Interrupt Enable 0

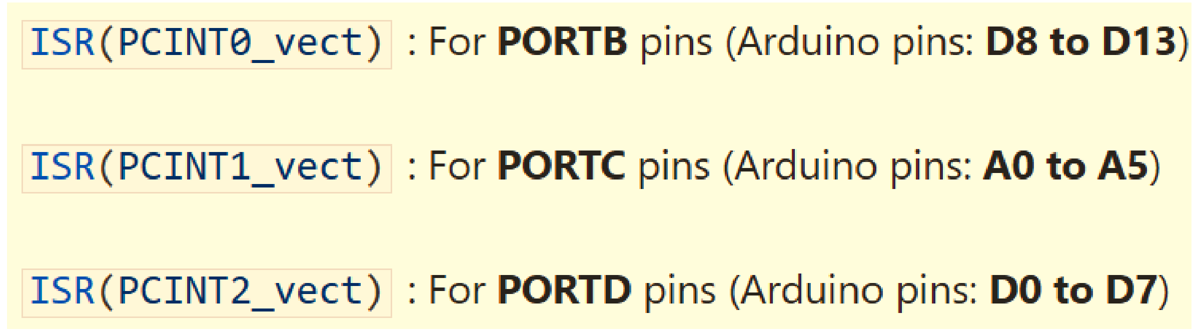
PCICR – Pin Change Interrupt Control Register

- If PCIEx is enabled and I bit in SREG is enabled, pin change interrupt x is enabled
- PCINT 23:16 corresponds to pin change interrupt 2
- PCINT 14:8 corresponds to pin change interrupt 1
- PCINT 7:0 corresponds to pin change interrupt 0
- The pins are enabled individually from PCMSKx register
- Executed from PCIx interrupt vector
- Any change in these pins will cause an interrupt

PCICR – Pin Change Interrupt Control Register



```
4   Pin Change Interrupt Request 0 (pins D8 to D13) (PCINT0_vect)
5   Pin Change Interrupt Request 1 (pins A0 to A5)  (PCINT1_vect)
6   Pin Change Interrupt Request 2 (pins D0 to D7)  (PCINT2_vect)
```



`ISR(PCINT0_vect)` : For **PORTB** pins (Arduino pins: **D8 to D13**)

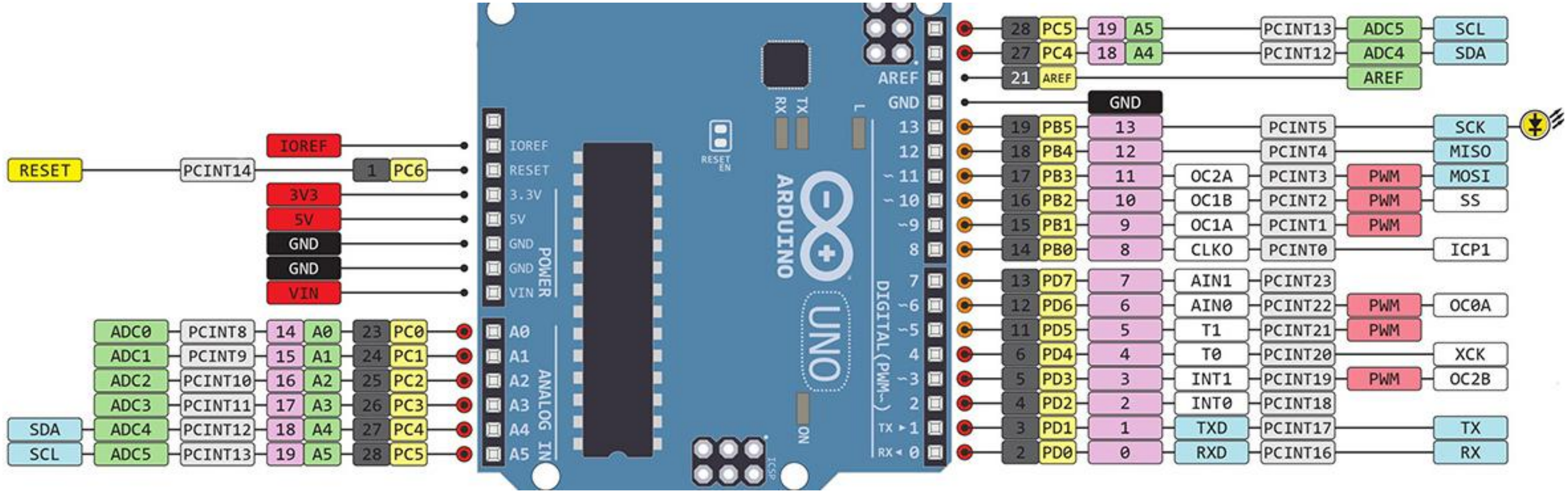
`ISR(PCINT1_vect)` : For **PORTC** pins (Arduino pins: **A0 to A5**)

`ISR(PCINT2_vect)` : For **PORTD** pins (Arduino pins: **D0 to D7**)

- PCINT 23:16 –
  - PD7:PD0 on ATMega328P
  - Pins D7:D0 (GPIO) on Arduino Uno board
- PCINT 14:8 –
  - PC6:PC0 on ATMega328P
  - PC5:0 – Pins A5:A0 on Arduino Uno board
  - PC6 – Reset pin on Arduino Uno Board
- PCINT 7:0 –
  - PB7:PB0 on ATMega328P
  - PB5:0 – Pins D13:D8 on Arduino Uno board
  - PB6 and PB7 are not available externally on the Arduino Uno board

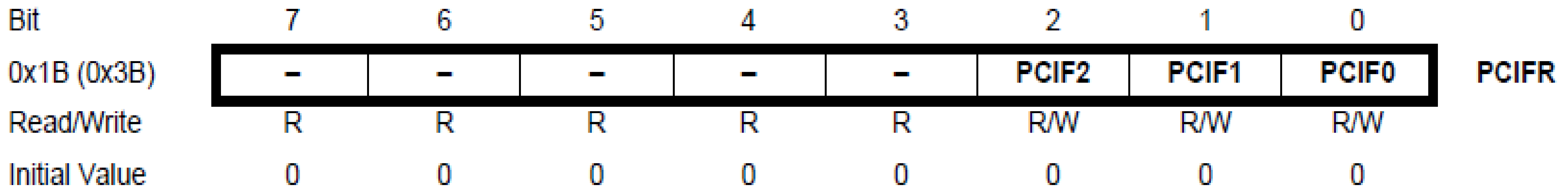
PCICR – Pin Change Interrupt Control Register

- All Arduino UNO digital pins can be used as interrupt pins using the PCINT (Pin Change Interrupts) hardware. This is not a dedicated interrupt signal source unlike the dedicated external interrupt pints (INT0/INT1).



RESET
PCINT14
IOREF
1 PC6
3V3
5V
GND
GND
VIN
ADC0 PCINT8 14 A0 23 PC0
ADC1 PCINT9 15 A1 24 PC1
ADC2 PCINT10 16 A2 25 PC2
ADC3 PCINT11 17 A3 26 PC3
SDA ADC4 PCINT12 18 A4 27 PC4
SCL ADC5 PCINT13 19 A5 28 PC5
28 PC5 19 A5 PCINT13 ADC5 SCL
27 PC4 18 A4 PCINT12 ADC4 SDA
21 AREF AREF
GND
19 PB5 13 PCINT5 SCK
18 PB4 12 PCINT4 MISO
17 PB3 11 OC2A PCINT3 PWM MOSI
16 PB2 10 OC1B PCINT2 PWM SS
15 PB1 9 OC1A PCINT1 PWM
14 PB0 8 CLKO PCINT0 ICP1
13 PD7 7 AIN1 PCINT23
12 PD6 6 AIN0 PCINT22 PWM OC0A
11 PD5 5 T1 PCINT21 PWM
6 PD4 4 T0 PCINT20 XCK
5 PD3 3 INT1 PCINT19 PWM OC2B
4 PD2 2 INT0 PCINT18
3 PD1 1 TXD PCINT17 TX
2 PD0 0 RXD PCINT16 RX
ARDUINO UNO

PCIFR – Pin Change Interrupt Flag Register



Bit	7	6	5	4	3	2	1	0	
0x1B (0x3B)	–	–	–	–	–	PCIF2	PCIF1	PCIF0	PCIFR
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bits 7:3 – Reserved – Always read as 0
- Bit 2 – PCIF2 – Pin Change Interrupt Flag 2
- Bit 1 – PCIF1 – Pin Change Interrupt Flag 1
- Bit 0 – PCIF0 – Pin Change Interrupt Flag 0

PCIFR – Pin Change Interrupt Flag Register

- Any logic change on PCINT23:0 triggers an interrupt request: PCIFx set
- If I bit in SREG and PCIEx on PCICR are set, MCU will go to the corresponding interrupt vector
- Flag is cleared when:
  - Interrupt routine is executed
  - A logical 1 is written

PCMSK2 – Pin Change Mask Register 2

Bit	7	6	5	4	3	2	1	0	
(0x6D)	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16	PCMSK2
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

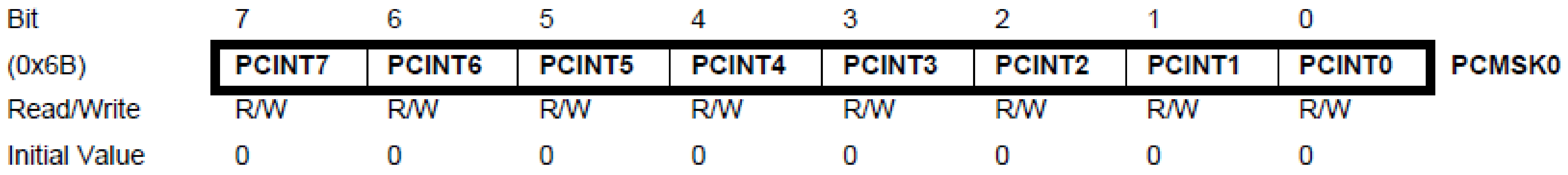
- Bits 7:0 – PCINT23:16 – Pin Change Enable Mask 23:16
- Each bit selects whether the pin change interrupt is enabled on the corresponding I/O pin
- If any bit from PCINT23:16 is set and the PCIE2 bit on PCICR is set, pin change interrupt is enabled on the corresponding I/O pin
- If PCINT23:16 is cleared, the pin change interrupt on the corresponding I/O pin is disabled

PCMSK1 – Pin Change Mask Register 1

Bit	7	6	5	4	3	2	1	0	
(0x6C)	–	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8	PCMSK1
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 7 – Reserved – Always read as 0
- Bits 6:0 – PCINT14:8 – Pin Change Enable Mask 14:8
- Each bit selects whether the pin change interrupt is enabled on the corresponding I/O pin
- If any bit from PCINT14:8 is set and the PCIE1 bit on PCICR is set, pin change interrupt is enabled on the corresponding I/O pin
- If PCINT14:8 is cleared, the pin change interrupt on the corresponding I/O pin is disabled

PCMSK0 – Pin Change Mask Register 0

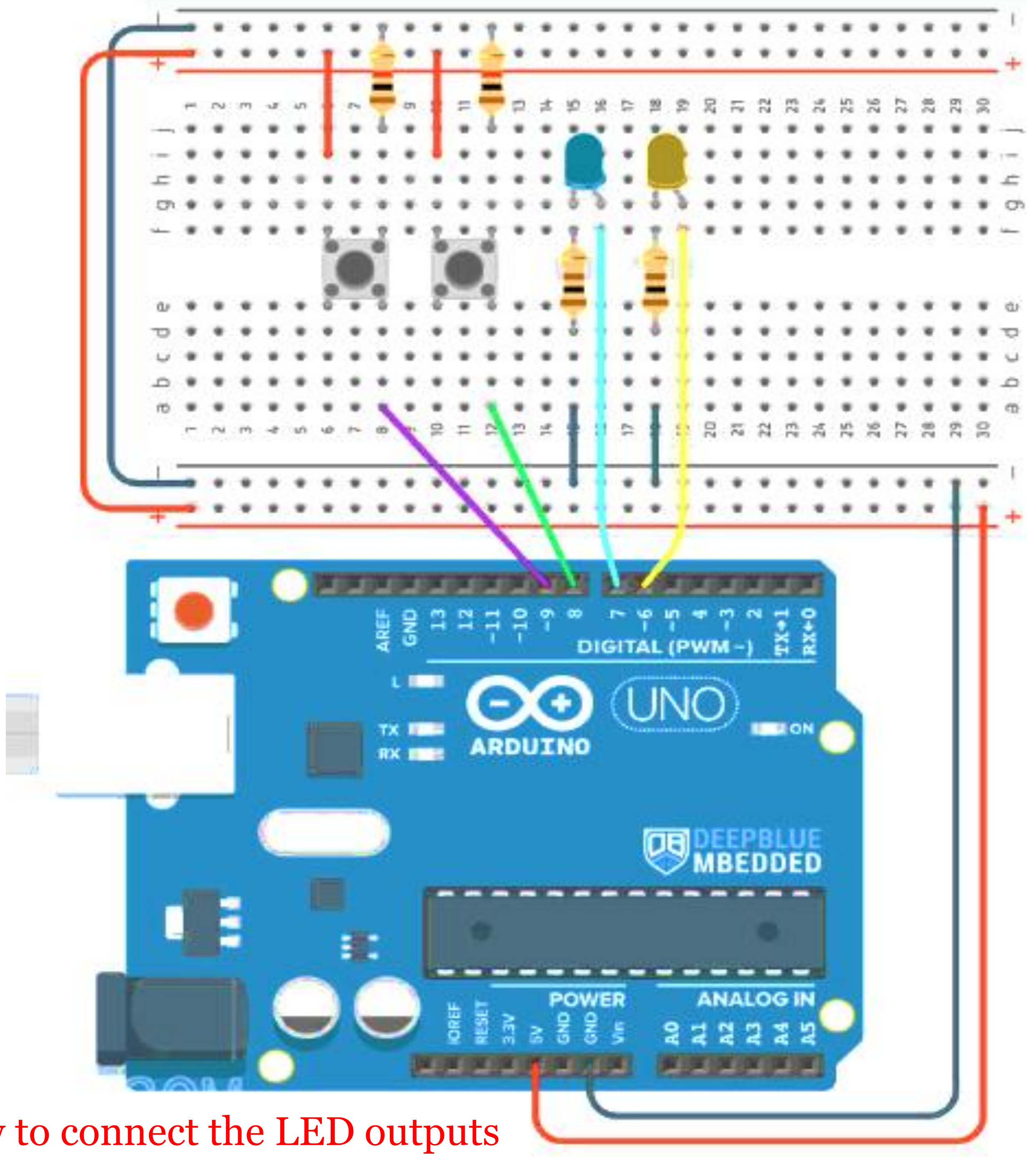


Bit	7	6	5	4	3	2	1	0	
(0x6B)	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0	PCMSK0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bits 7:0 – PCINT7:0 – Pin Change Enable Mask 7:0
- Each bit selects whether the pin change interrupt is enabled on the corresponding I/O pin
- If any bit from PCINT7:0 is set and the PCIE0 bit on PCICR is set, pin change interrupt is enabled on the corresponding I/O pin
- If PCINT7:0 is cleared, the pin change interrupt on the corresponding I/O pin is disabled

Example on Pin Change Interrupt:

Use 2 pins in the same port for pin change interrupt. Use Arduino Pin8 and Pin9, which maps to (B0 & B1 in PORTB). The 2 PCINT input pins must be connected to 2 push buttons. Use the interrupt events to toggle 2 output LEDs, connected to the pin pin6 and pin7 (mapped to D6 and D7 of PORTD) in the respective ISR handler function for each PCINT interrupt.



The wiring diagram for this example showing how to connect the LED outputs (6 & 7), and the push buttons to the PCINT interrupt input pins (8 & 9)

Example on Pin Change Interrupt:

```
#include "PinChangeInterrupt.h"

#define BTN0_PIN  8
#define BTN1_PIN  9
#define LED0_PIN  6
#define LED1_PIN  7

void BTN0_ISR(void)
{
  digitalWrite(LED0_PIN,
    !digitalRead(LED0_PIN));
}

void BTN1_ISR(void)
{
  digitalWrite(LED1_PIN,
    !digitalRead(LED1_PIN));
}
```

```
void setup() {
  pinMode(LED0_PIN, OUTPUT);
  pinMode(LED1_PIN, OUTPUT);
  pinMode(BTN0_PIN, INPUT);
  pinMode(BTN1_PIN, INPUT);
  attachPCINT(digitalPinToPCINT(BTN0_PIN), BTN0_ISR, RISING);
  attachPCINT(digitalPinToPCINT(BTN1_PIN), BTN1_ISR, RISING);
}

void loop() {
  // Do Nothing
}
```

References

- ATMega328 Datasheet
- Arduino Uno Datasheet
- Chapter 5: Microprocessors, Advanced Industrial Control Technology by Peng Zhang
- http://www.ganssle.com/debouncing-pt2.htm
- https://mansfield-devine.com/speculatrix/2018/04/debouncing-fun-with-schmitt-triggers-and-capacitors/
- https://www.arxterra.com/10-atmega328p-interrupts/
- https://www.arxterra.com/11-atmega328p-external-interrupts/
- http://www.gammon.com.au/forum/?id=11488

Thanks for attending….