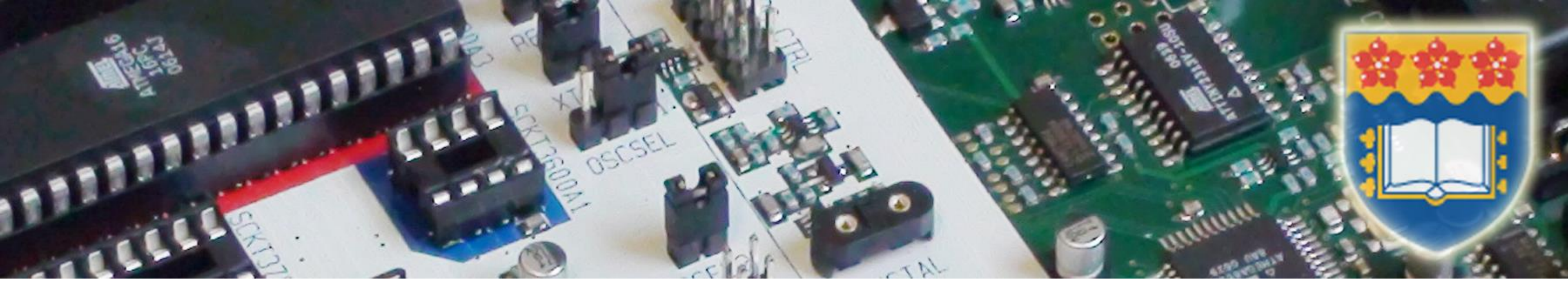


ECTE333
Lecture 4 – Interrupts and Timers

Prof. Lam Phung

School of Electrical, Computer and Telecommunications Engineering
University of Wollongong
Australia

ECTE333's schedule

Lecture (2h)	Tutorial (1h)	Lab (2h)
L1:Introduction to AVR Microcontrollers		
L2: C Programming, Digital IO	Tutorial 1	Lab 1
L3: Serial Communication		
	Tutorial 2	Lab 2
L4: Interrupts, Timers		
	Tutorial 3	Lab 3
L5: Pulse Width Modulators		
	Tutorial 4	Lab 4
L6: Analogue-to-Digital Converters		
	Tutorial 5	Lab 5
L7: Microcontroller Applications		
		Lab 6

Lecture 4’s sequence

4.1 Interrupt programming in C for ATmega16

4.2 Timers in ATmega16

4.3 Timer applications

4.1 Interrupt programming in C for ATmega16

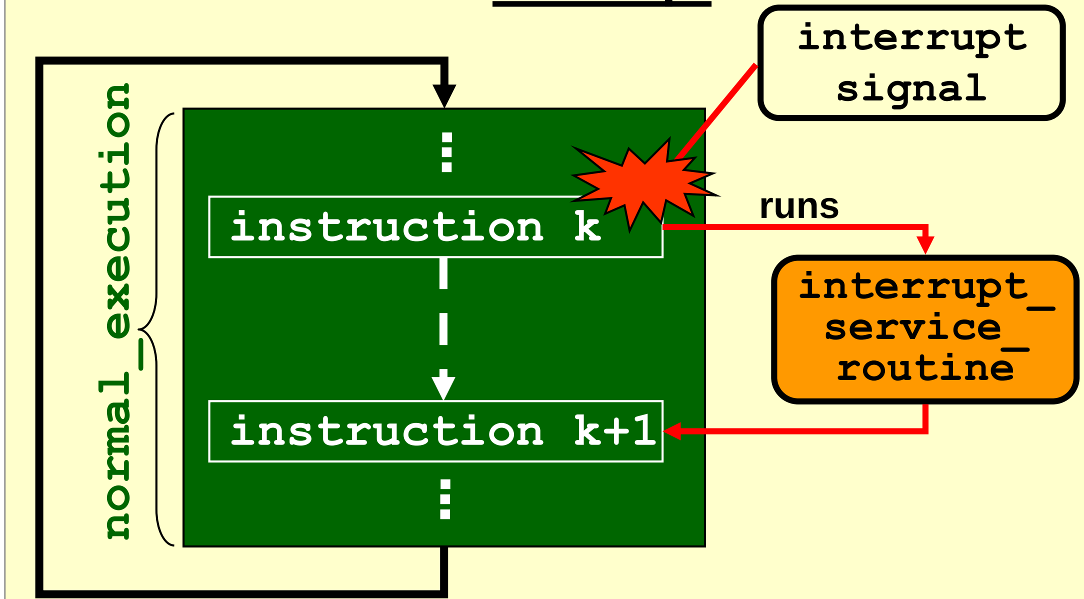
- **Compared to polling, interrupt is a more efficient approach for the CPU to handle peripheral devices.**
- **Example peripheral devices are serial port, external switches, timers, PWM, and ADC.**
- **In this lecture, we will learn the interrupt subsystem in the ATmega16.**
- **We will also learn how to write an interrupt-driven program in C.**

Polling versus Interrupt

Polling

```
while (1){
    get_device_status;
    if (service_required){
        service_routine;
    }
    normal_execution;
}
```

Interrupt



- Using polling: The CPU must continually check the device's status.
- Using interrupt:
  - When needed, a device sends an interrupt signal.
  - In response, the CPU will perform an interrupt service routine, and then resume its normal execution.

Interrupt execution sequence

1. A device issues an interrupt
2. CPU finishes the current instruction
3. CPU acknowledges the interrupt
4. CPU saves its states and PC onto stack
5. CPU loads the address of ISR onto PC
6. CPU executes the ISR
7. CPU retrieves its states and PC from stack
8. Normal execution resumes

ATmega16 interrupt subsystem

The ATmega16 has 21 interrupts:

- **3 external interrupts**
- **8 timer interrupts**
- **3 serial port interrupts**
- **1 ADC interrupt**

our focus

- **1 analogue comparator interrupt**
- **1 SPI interrupt**
- **1 TWI interrupt**
- **2 memory interrupts**
- **1 reset interrupt**

Table 4.1: Interrupts in ATmega16

Vector No.	Program Address	Interrupt vector name	Description
1	$000	RESET_vect	Reset
2	$002	INT0_vect	External Interrupt Request 0
3	$004	INT1_vect	External Interrupt Request 1
4	$006	TIMER2_COMP_vect	Timer/Counter2 Compare Match
5	$008	TIMER2_OVF_vect	Timer/Counter2 Overflow
6	$00A	TIMER1_CAPT_vect	Timer/Counter1 Capture Event
7	$00C	TIMER1_COMPA_vect	Timer/Counter1 Compare Match A
8	$00E	TIMER1_COMPB_vect	Timer/Counter1 Compare Match B
9	$010	TIMER1_OVF_vect	Timer/Counter1 Overflow
10	$012	TIMER0_OVF_vect	Timer/Counter0 Overflow
11	$014	SPI_STC_vect	Serial Transfer Complete
12	$016	USART_RXC_vect	USART, Rx Complete
13	$018	USART_UDRE_vect	USART Data Register Empty
14	$01A	USART_TXC_vect	USART, Tx Complete
15	$01C	ADC_vect	ADC Conversion Complete
16	$01E	EE_RDY_vect	EEPROM Ready
17	$020	ANA_COMP_vect	Analog Comparator
18	$022	TWI_vect	2-wire Serial Interface
19	$024	INT2_vect	External Interrupt Request 2
20	$026	TIMER0_COMP_vect	Timer/Counter0 Compare Match
21	$028	SPM_RDY_vect	Store Program Memory Ready

Table 4.1: Interrupts in ATmega16

- Vector No
  - An interrupt with a lower ‘Vector No’ has a higher priority.
  - E.g., INT0 has a higher priority than INT1 and INT2.

- Program Address
  - The fixed memory location for a given interrupt handler.
  - E.g., in response to interrupt INT0, CPU runs the instruction at $002.

- Interrupt Vector Name
  - This is the interrupt name, to be used with C macro ISR().

Steps to program an interrupt in C

- To program an interrupt, 5 steps are required.
  1. Include header file <avr\interrupt.h>.
  2. Use C macro ISR() to define the interrupt handler and update IVT.
  3. Enable the specific interrupt.
  4. Configure details of the interrupt by setting the relevant registers.
  5. Enable the interrupt subsystem globally using sei().

- Later, we'll study steps for interrupt programming in C, via 2 examples.

  4.1.1 USART RXD Complete interrupt

  4.1.2 External interrupts

Using C macro ISR()

- The C macro ISR() is used to define the handler for a given interrupt.

- Its syntax is given as

```
ISR(interrupt_vector_name){
    // … code for interrupt handler here
}
```

where `interrupt_vector_name` is given in Table 4.1.

- Example: On interrupt 'RXD Complete', to put the received character in Port B, we write

```
ISR(USART_RXC_vect){
  PORTB = UDR;  // put the received character in Port B
}
```

Learning ATmega16 interrupts

Vector No.	Interrupt vector name	Description
1	RESET_vect	Reset
2	INT0_vect	External Interrupt Request 0
3	INT1_vect	External Interrupt Request 1
4	TIMER2_COMP_vect	Timer/Counter2 Compare Match
5	TIMER2_OVF_vect	Timer/Counter2 Overflow
6	TIMER1_CAPT_vect	Timer/Counter1 Capture Event
7	TIMER1_COMPA_vect	Timer/Counter1 Compare Match A
8	TIMER1_COMPB_vect	Timer/Counter1 Compare Match B
9	TIMER1_OVF_vect	Timer/Counter1 Overflow
10	TIMER0_OVF_vect	Timer/Counter0 Overflow
11	SPI_STC_vect	Serial Transfer Complete
12	USART_RXC_vect	USART, Rx Complete
13	USART_UDRE_vect	USART Data Register Empty
14	USART_TXC_vect	USART, Tx Complete
15	ADC_vect	ADC Conversion Complete
16	EE_RDY_vect	EEPROM Ready
17	ANA_COMP_vect	Analog Comparator
18	TWI_vect	2-wire Serial Interface
19	INT2_vect	External Interrupt Request 2
20	TIMER0_COMP_vect	Timer/Counter0 Compare Match
21	SPM_RDY_vect	Store Program Memory Ready

Lecture 4.1.2

Lecture 4.2, 4.3

Lecture 5

Lecture 4.1.1

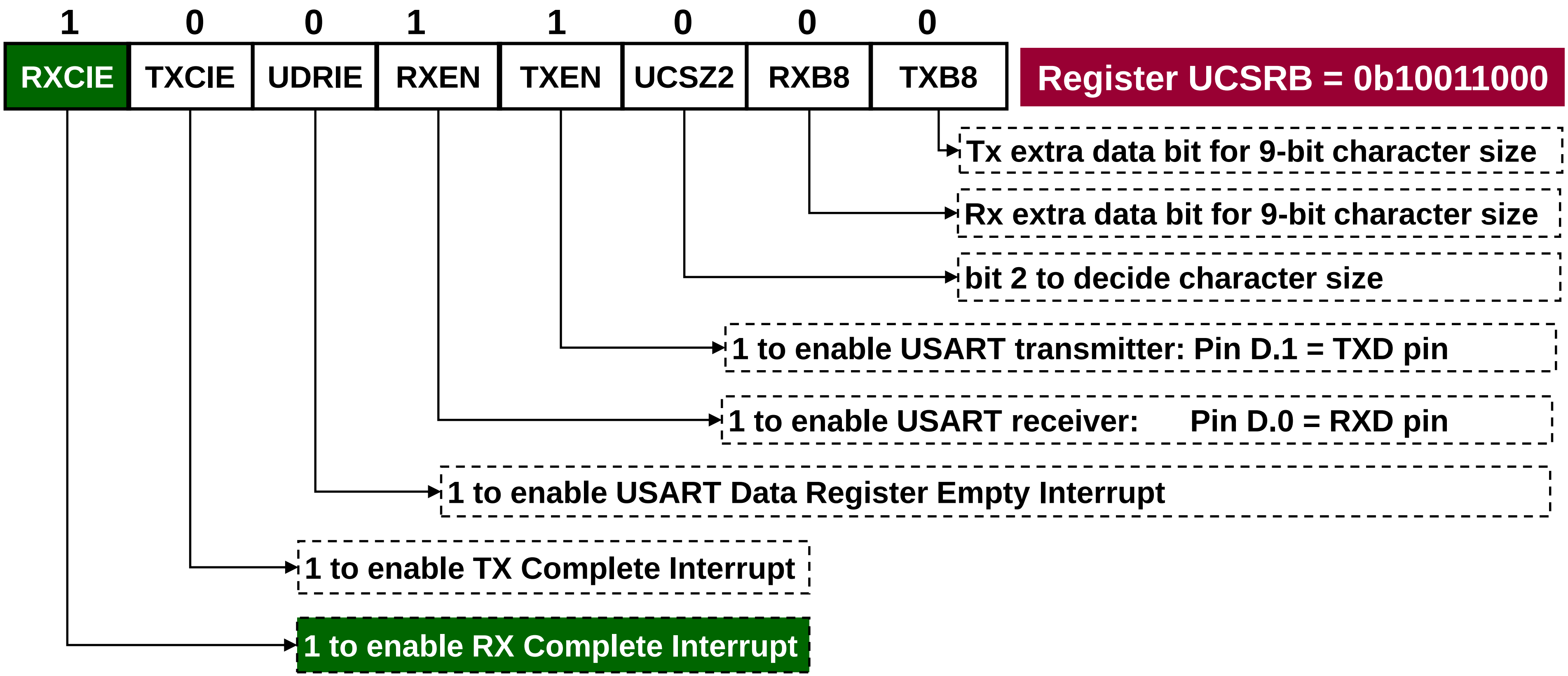
Lecture 6

4.1.1 Serial RXD interrupt

Write a C interrupt-driven program to use the serial port of ATmega16 at baud rate 1200, no parity, 1 stop bit, 8 data bits, clock speed 1MHz. Whenever a character is received, it should be sent to Port B.

- **The serial port of ATmega16 can trigger an RXD interrupt whenever a character is received [Lecture 3].**
- **We enable this interrupt by setting a flag in a serial port register.**
- **We then write the ISR, to be run whenever the interrupt is triggered.**

Serial RXD interrupt: Enabling



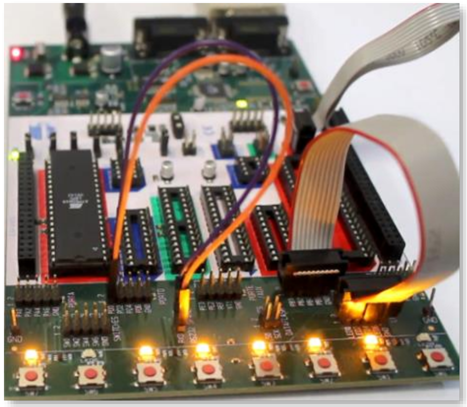
- The ATmega16 manual explains how to enable any interrupt.
- For serial RXD interrupt, we look at the ‘USART’ section.

Serial RXD interrupt: serial_int.c

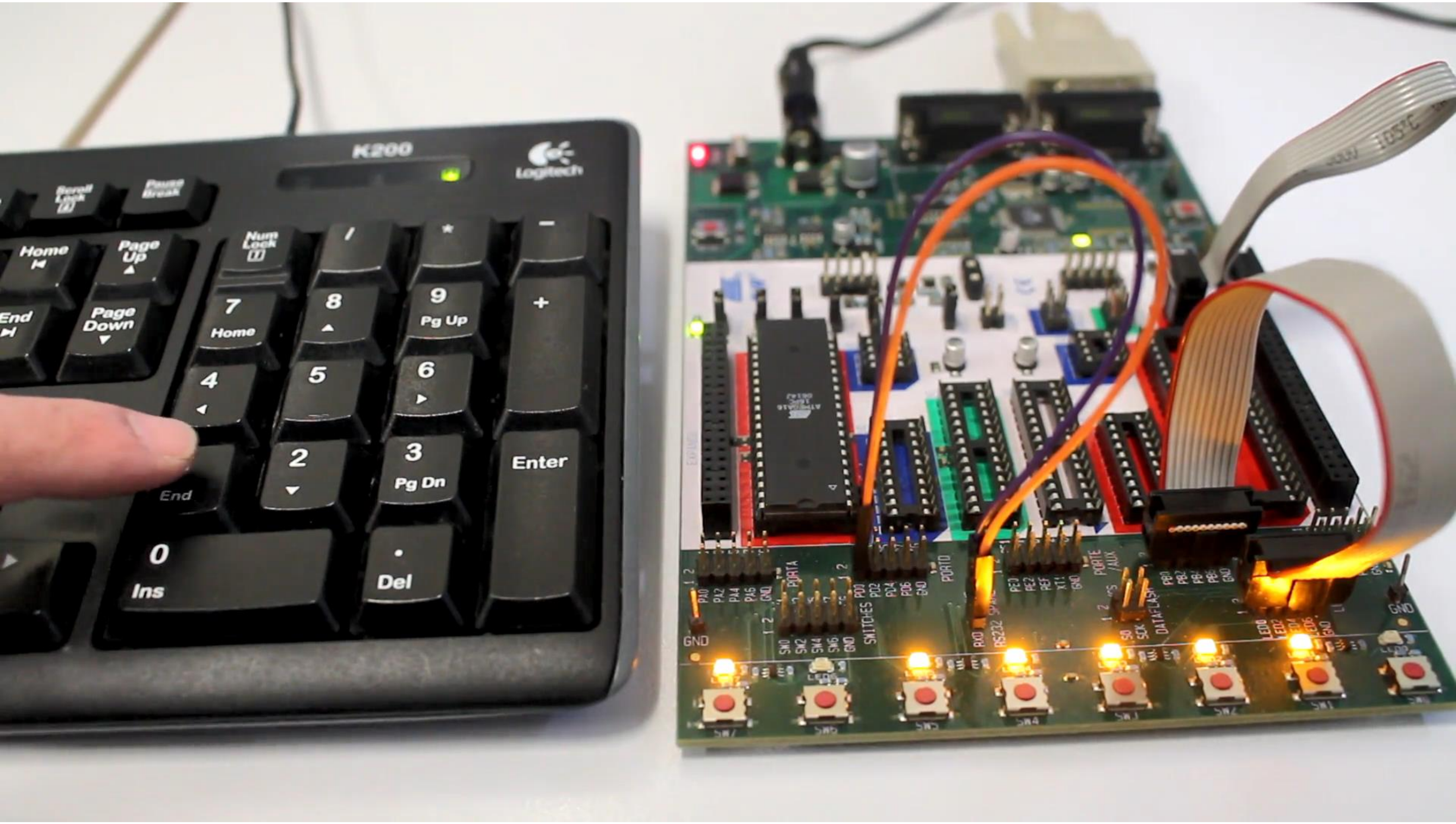
```c
#include <avr/io.h>
#include <avr/interrupt.h>                                              1

void serial_init(void){
    // Normal speed, disable multi-proc
    UCSRA = 0b00000000;

    // Enable Tx and Rx pins, enable RX interrupt                       3
    UCSRB = 0b10011000;

    // Asynchronous mode, no parity, 1 stop bit, 8 data bits
    UCSRC = 0b10000110;

    // Baud rate 1200bps, assuming 1MHz clock
    UBRRL = 0x33; UBRRH = 0x00;
}

ISR(USART_RXC_vect){ // handler for RXD interrupt
    PORTB = UDR;      // received character is displayed on port B     2
}

int main(void) {
    serial_init(); // initialise serial port                            4
    sei();         // enable interrupt subsystem globally               5
    DDRB = 0xFF;   // set port B for output
    while (1) {;}  // infinite loop
    return 0;
}
```

Serial RXD interrupt: Testing

- Connect **RXD pin (pin D.0)** to **RXD pin of RS232 Spare**.
- Connect **TXD pin (pin D.1)** to **TXD pin of RS232 Spare**.
- Connect **Port B** to **LED connector**.
- Compile, download program.
- Connect **RS232 Spare Connector** to **Serial Port of PC**.
- Configure and run Hyper Terminal and use it to send characters.



Serial RXD interrupt: Testing



Demo: serial_int.mp4

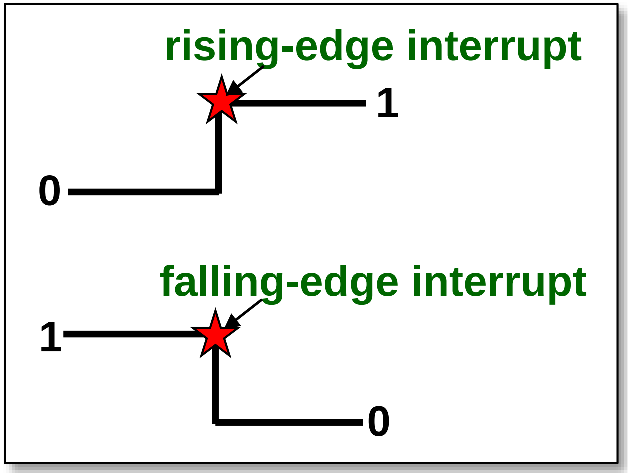
Serial RXD: Polling approach

For comparison, the program below uses polling for the same effect.

```
#include <avr/io.h>

void serial_init(void){
   // Normal speed, disable multi-proc
   UCSRA = 0b00000000;

   // Enable Tx and Rx, disable interrupts
   UCSRB = 0b00011000;

   // Asynchronous mode, no parity, 1 stop bit, 8 data bits
   UCSRC = 0b10000110;

   // Baud rate 1200bps, assuming 1MHz clock
   UBRRL = 0x33; UBRRH = 0x00;
}

int main(void) {
   serial_init(); // initialise serial port
   DDRB = 0xFF;  // set port B for output
   while (1) {   // infinite loop
       while ((UCSRA & (1<<RXC)) == 0x00){;} // poll until RXC flag = 1

       PORTB = UDR;  // received character is displayed on port B
   }
   return 0;
}
```

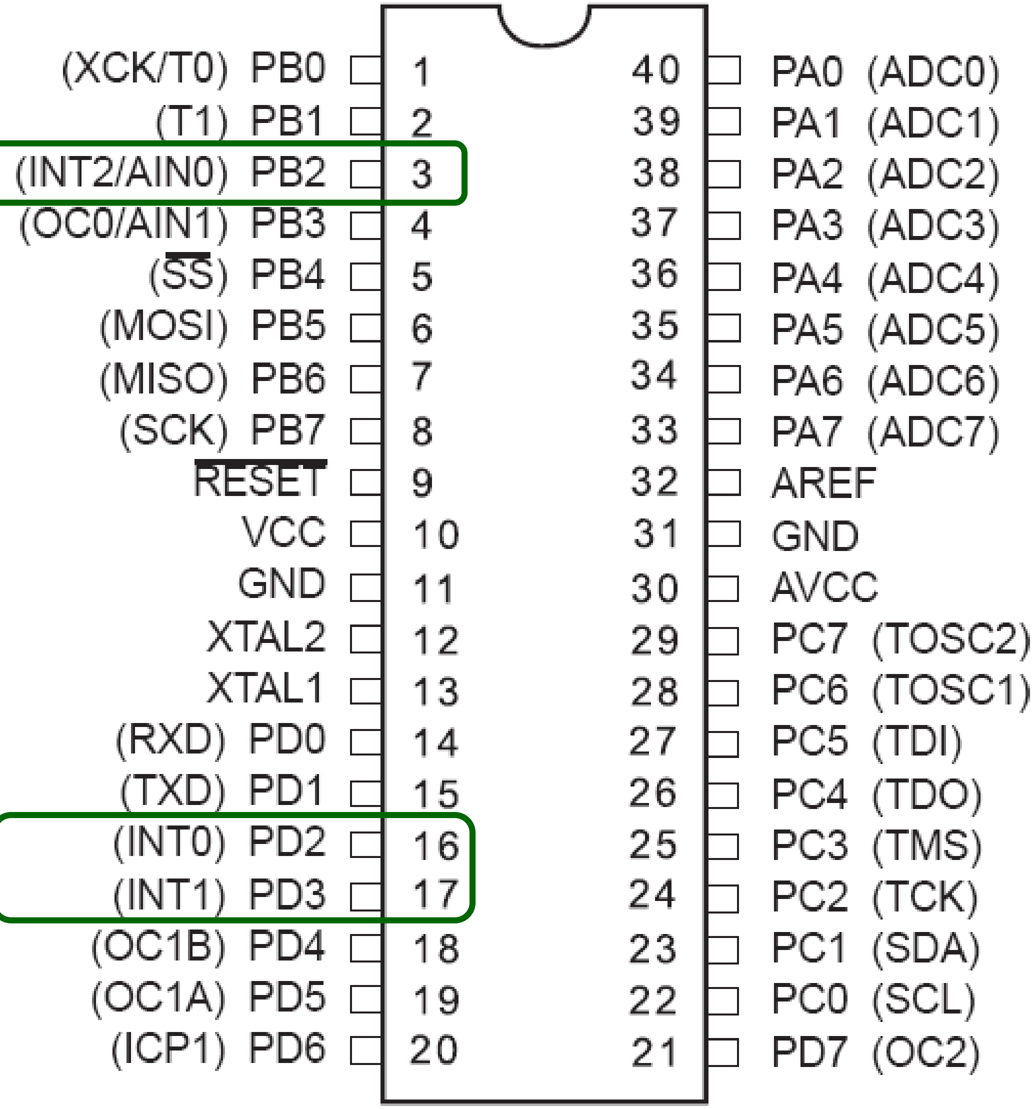
4.1.2 External interrupts

- External interrupts on ATmega16 and ATmega8515 are similar.
- Key references: ATmega16 user manual, 'External Interrupts' section.
- Three external interrupts can be triggered.
  - INT0 on pin D.2,
  - INT1 on pin D.3,
  - INT2 on pin B.2.



- Key steps in using external interrupts:
  - enable the interrupt,
  - specify what events will trigger the interrupt.

External Interrupts: Relevant pins



ATmega16 chip

External interrupts: Enabling

- To enable an external interrupt, set a flag in General Interrupt Control Register (GICR):

INT1	INT0	INT2	-	-	-	IVSEL	IVSEL
0	0	0	(initial value)				

- Example: To enable INT1 (pin D.3), we can write

```
GICR = 0b10000000;    // same as GICR = (1 << INT1);
```

Note that `INT1` and `GICR` names are already defined in <avr/io.h>.

```
#define INT1   7
```

External interrupts: Specifying events

- Specify the events that trigger an external interrupt by using
  - MCU Control Register (for INT0 and INT1), or
  - MCU Control and Status Register (for INT2).

- **For INT2**:

MCUCSR

JTD	ISC2	-	JTRF	WDRF	BORF	EXTRF	PORF

INT2

ISC2:
- 0: falling edge triggers an interrupt INT2
- 1: rising edge triggers an interrupt INT2

External interrupts: Specifying events

- **For INT0 and INT1:**

MCUCR								
MCUCR	SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00

INT1

ISC11	ISC10	An interrupt is triggered when
0	0	low level of INT1
0	1	any change of INT1
1	0	falling edge of INT1
1	1	rising edge of INT1

INT0

ISC11	ISC10	An interrupt is triggered when
0	0	low level of INT0
0	1	any change of INT0
1	0	falling edge of INT0
1	1	rising edge of INT0

External interrupts: Example

Write a C program to invert Port B whenever a switch on the STK500 board is pressed. The program should use an external interrupt.

- **Let's use interrupt INT1. This interrupt is triggered on pin D.3.**
- **To enable interrupt INT1:**

```
GICR = 0b1000000;    // same as GICR = (1 << INT1);
```

- **To specify that INT1 is triggered on any change in pin D.3:**

```
MCUCR = 0b00000100; // same as MCUCR = (1 << ISC10);
```

- **Then, we write the ISR, and enable interrupt subsystem.**

External interrupts: ext_int.c

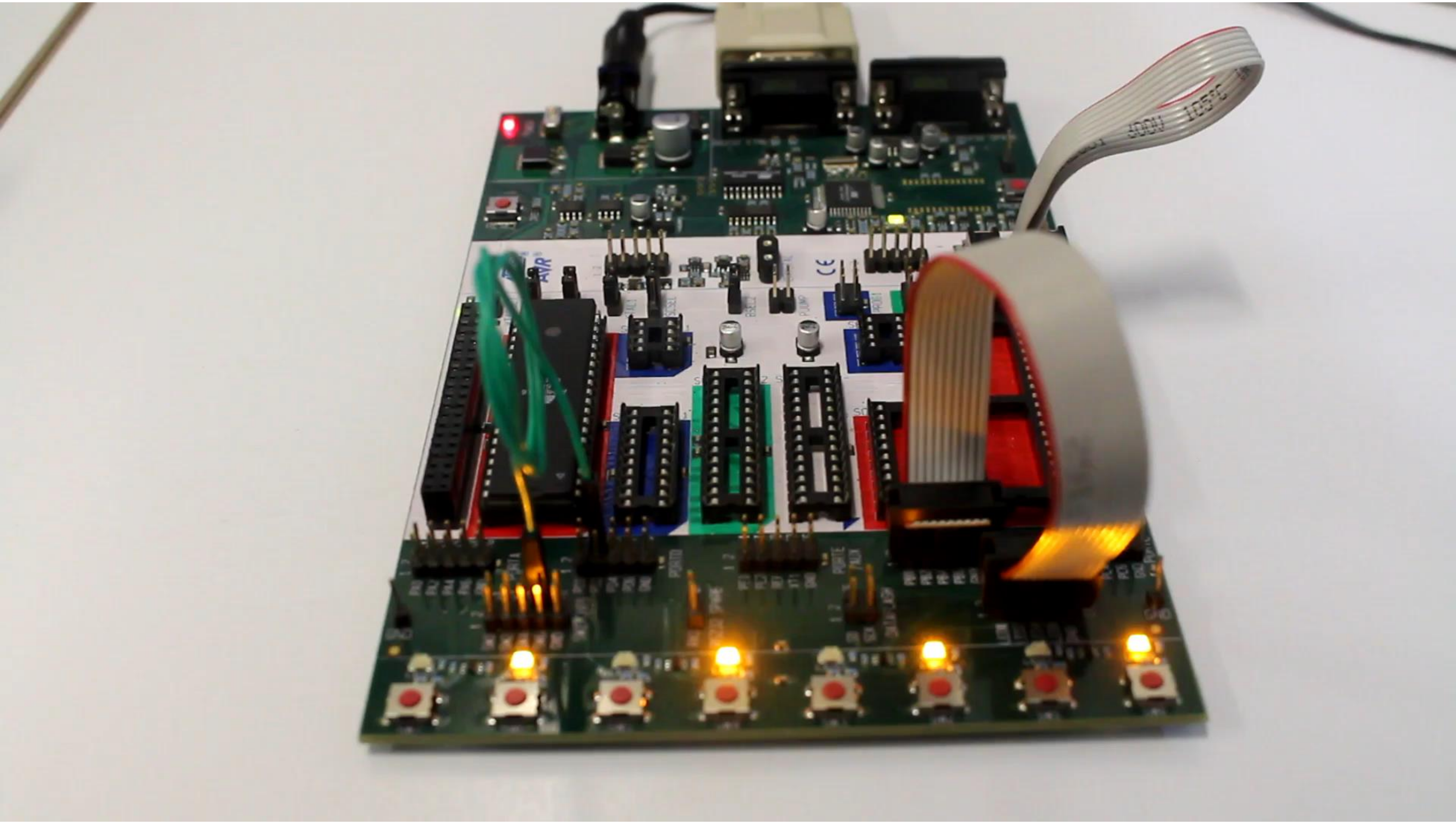
```c
#include <avr/io.h>
#include <avr/interrupt.h>

ISR(INT1_vect){           // handler for INT1 interrupt
   PORTB = (~PORTB);     // invert Port B
}

int main(void) {
   GICR =  (1 << INT1);    // enable interrupt INT1
   MCUCR = (1 << ISC10); // triggered on any change to INT1 (D.3)
   sei();                 // enable interrupt subsystem globally
   DDRB = 0xFF;           // set Port B for output
   PORTB = 0b10101010;    // initial value
   while (1) {;}          // infinite main loop
   return 0;
}
```

1 2 3 4 5

External interrupts: Testing ext_int.c

- **Connect INT1 pin (D.3) to switch SW7 (or any switch) of STK500 board.**
- **Connect GRD pin of Port D to GRD pin of SW connector.**
- **Connect Port B to LED connector.**
- **Compile, download program.**
- **Press switch SW7; LEDs will toggle.**

External interrupts: Testing ext_int.c



Demo: ext_int.mp4

Lecture 4's sequence

4.1 Interrupt programming in C for ATmega16

4.2 Timers in ATmega16

4.3 Timer applications

4.2 Timers in ATmega16

- **Many computer applications require accurate timing.**

- **Examples include:**
  - **recording the time when an event occurs,**
  - **calculating the time difference between events,**
  - **performing tasks at specific or periodic times,**
  - **creating accurate time delays,**
  - **generating waveforms of a certain shape, period, or duty cycle.**

- **Lectures 4 and 5 focus on using timers to perform time-related tasks.**

Timer terminology

- **Input Capture**:
  - **Input signal is connected to a pin, called input capture, of the timer.**
  - **When an event (rising edge, falling edge, or change) occurs on this pin, the current timer value is automatically stored in a register.**

- **Output Compare**:
  - **A timer typically has a pin, called output compare.**
  - **When the timer reaches a preset value, the output compare pin can be automatically changed to binary 0 or 1.**

Overview of Timers in ATmega16

- ATmega16 has three timers: Timer 0, Timer 1 and Timer 2.
- Each timer is associated with a counter and a clock signal.
- The counter is incremented by 1 in every clock cycle of the timer.
- The clock signal of a timer can come from
  - the internal system clock, or
  - an external clock source.

Overview of Timers in ATmega16

- When the internal system clock is used, a prescaler can be applied to make the timer count at a slower rate.

- Example:
  - Consider a system clock of 1Mhz (i.e. 1µs per cycle).
  - Suppose that a timer prescaler of 64 is used.
  - Then, timer will increment every 64µs.

Overview of Timers in ATmega16

	Timer 0	Timer 1	Timer 2
Overall	- 8-bit counter - 10-bit prescaler	- 16-bit counter - 10-bit prescaler	- 8-bit counter - 10-bit prescaler
Functions	- PWM - Frequency generation - Event counter - Output compare	- PWM - Frequency generation - Event counter - Output compare channels: 2 - Input capture	- PWM - Frequency generation - Event counter - Output compare
Operation modes	- Normal mode - Clear timer on compare match - Fast PWM - Phase correct PWM	- Normal mode - Clear timer on compare match - Fast PWM - Phase correct PWM	- Normal mode - Clear timer on compare match - Fast PWM - Phase correct PWM

Timer 1 has the most capability among the three timers.

Study plan

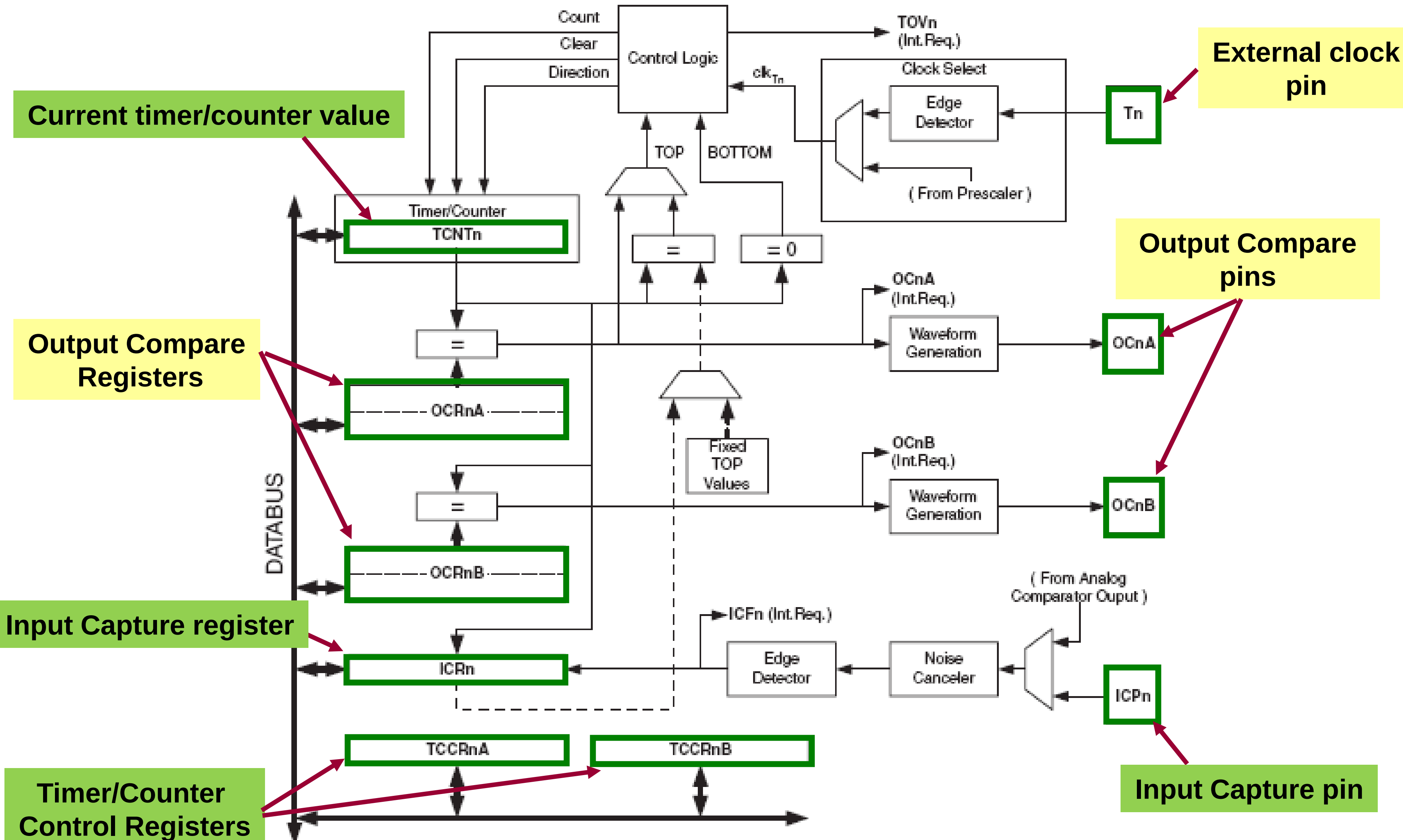
- In Lecture 4, we focus on
  - operations of Timer 1,
  - using Timer 1 overflow interrupt,
  - using Timer 1 input capture interrupt,
  - measuring time, creating time delay,
  - measuring period/duty cycle of a signal,
  - information required for Lab 3.

- In Lecture 5, we will learn
  - using Timer 1 output compare interrupt,
  - generating PWM signals,
  - information required for Lab 4.

Timer 1: An overview

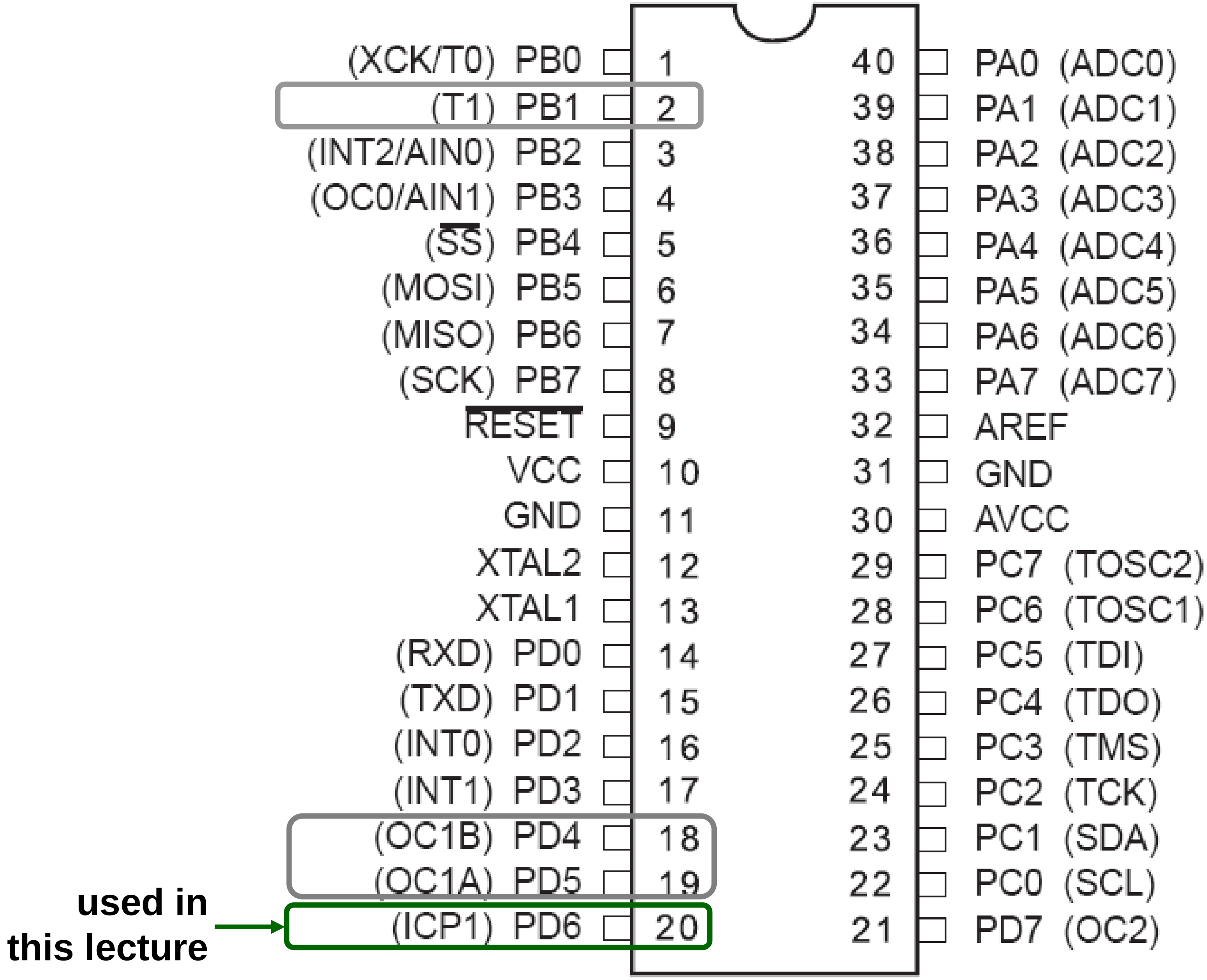
- 16-bit counter.
- 10-bit prescaler: 8, 64, 256, and 1024
- can trigger a timer overflow interrupt when counter reaches MAX.
- can trigger an input capture interrupt when an event occurs on the input capture pin.
  - Timer value is stored automatically in a register.
  - Input capture pin for Timer 1 is ICP1 (D.6).
- can trigger an output compare match interrupt when timer reaches a preset value.
  - There are two independent output compare channels A and B.

Timer 1: Block diagram



Not shown here: TIMSK and TIFR registers

Timer 1 — Relevant pins



Timer 1 — Five groups of registers

1) Timer/Counter 1
- TCNT1
- 16-bit register that stores the current value of the timer.

2) Timer/Counter 1 Control Registers
- TCCR1A and TCCR1B
- To configure the operations of Timer 1.

3) Input Capture Register
- ICR1
- to store timer value when an event occurs on input capture pin.

4) Interrupt registers
- TIMSK to enable timer interrupts
- TIFR to monitor status of timer interrupts.

5) Output Compare Registers
- OCR1A, OCR1B
- To store the preset values for output compare.

will be covered in Lecture 5.

Timer 1 — Five groups of registers

We now study the important registers for Timer 1.

4.2.1 Timer/Counter 1 Control Register A (TCCR1A)

7	6	5	4	3	2	1	0
COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10

- **WGM11, WGM10:** Specify Waveform Generation Mode used with WGM13, WGM12. WGM13..WGM10 = 0000 for normal mode.
- **FOC1B:** 1 to force output compare on channel B
- **FOC1A:** 1 to force output compare on channel A
- **COM1B1, COM1B0:** Output compare mode for channel B
- **COM1A1, COM1A0:** Output compare mode for channel A

4.2.2 Timer/Counter 1 Control Register B (TCCR1B)

7	6	5	4	3	2	1	0
ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10

clock select (next slide)

select Waveform Generation Mode, used with WGM11, WGM10
WGM13..WGM10 = 0000 for normal mode.

input capture edge select: 1 will select rising edge,
0 will select falling edge

1 to activate input capture noise canceller

Clock select

CS12	CS11	CS10	Description
0	0	0	No clock source (timer stopped)
0	0	1	$CLK_{I/O}/1$ (no prescaling)
0	1	0	$CLK_{I/O}/8$ (from prescaler)
0	1	1	$CLK_{I/O}/64$ (from prescaler)
1	0	0	$CLK_{I/O}/256$ (from prescaler)
1	0	1	$CLK_{I/O}/1024$ (from prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

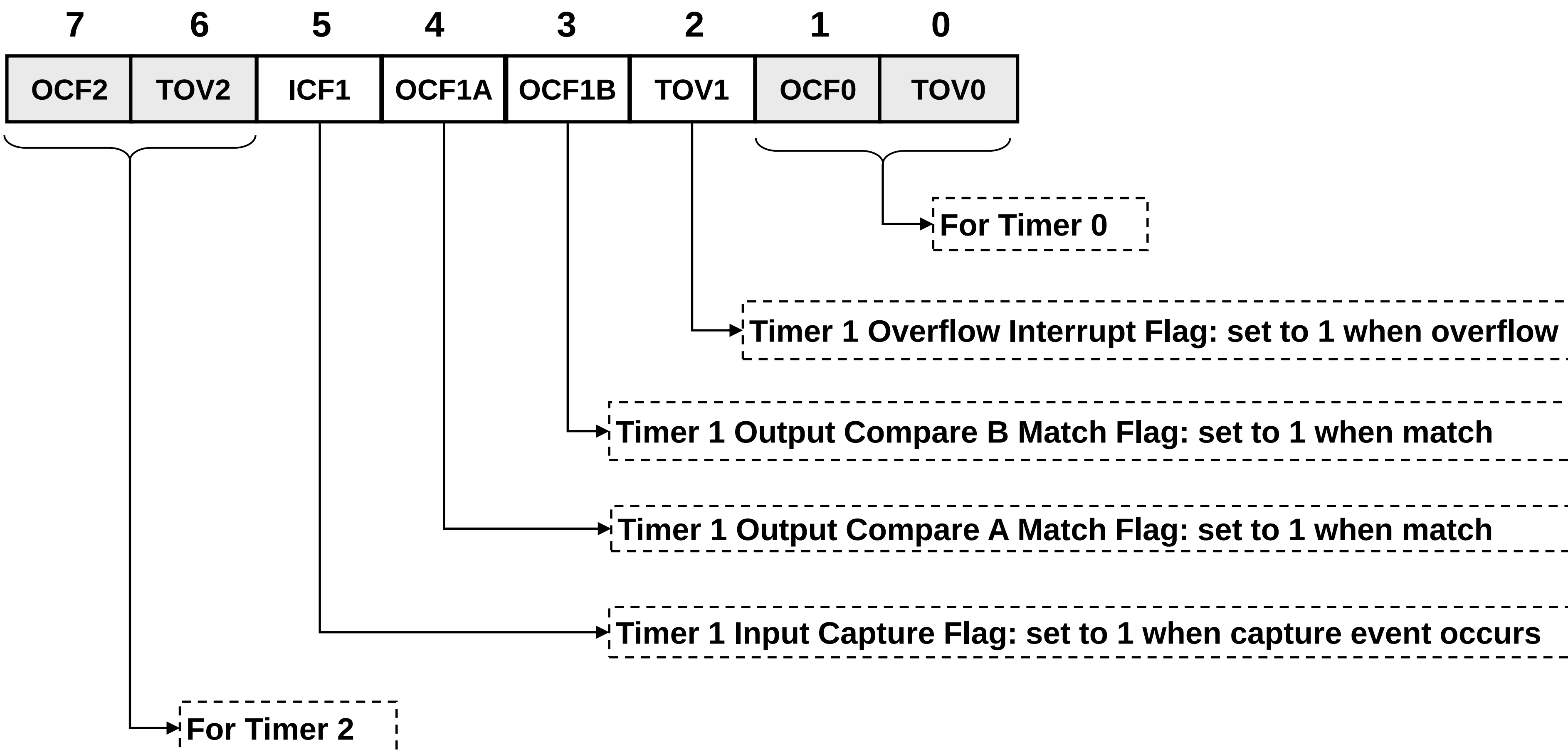
- For ATmega16, the default internal clock is $CLK_{I/O}$ = 1MHz.
- Timer 1 can use the internal or external clock.
- If using the internal clock, we can set Timer 1 to be 8, 64, 256, or 1024 times slower than the internal clock.

4.2.3 Timer/Counter Interrupt Mask Register (TIMSK)

7	6	5	4	3	2	1	0
OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0

- For Timer 0
- Timer 1 Overflow Interrupt Enable
- Timer 1 Output Compare B Match Interrupt Enable: 1 to enable
- Timer 1 Output Compare A Match Interrupt Enable: 1 to enable
- Timer 1 Input Capture Interrupt Enable: 1 to enable
- For Timer 2

4.2.4 Timer/Counter Interrupt Flag Register (TIFR)



- This register has flags that indicate when a timer interrupt occurs.
- It is not often used in ATmega16 programs.

Lecture 4's sequence

4.1 Interrupt programming in C for ATmega16

4.2 Timers in ATmega16

4.3 Timer applications

4.3 Timer applications

- **In this section, we consider three applications of Timer 1.**

  4.3.1 Creating an accurate delay using timer overflow interrupt.

  4.3.2 Measuring elapsed time between two events.

  4.3.3 Measuring the period of a square signal using input capture interrupt.

4.3.1 Creating an accurate delay

Write a C program for ATmega16 to invert PORTB every 2 seconds.

It should use Timer 1 overflow interrupt to create delays of 2s each.

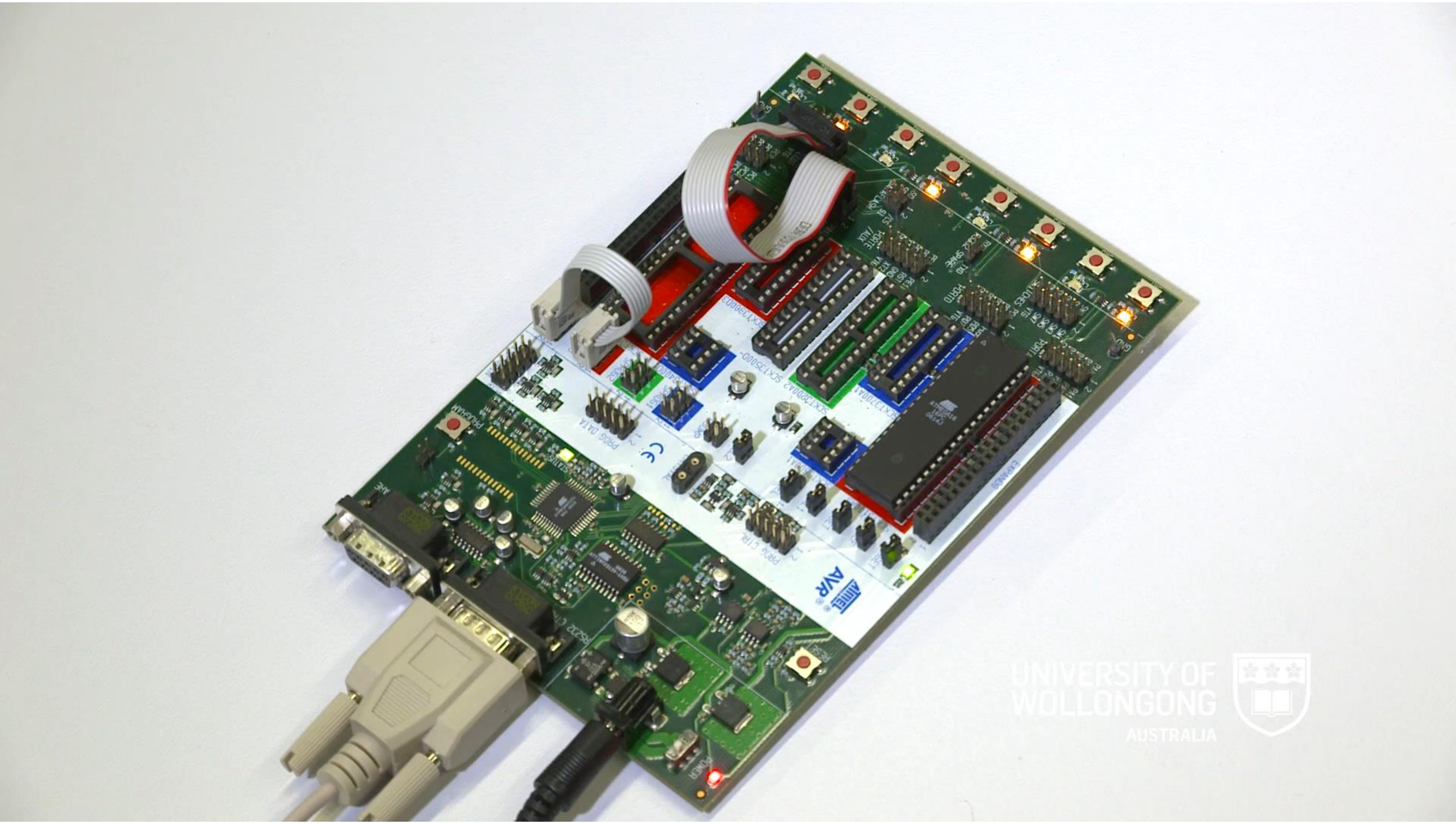
- **Analysis**
  - Internal system clock: 1MHz.
  - With no prescaler, Timer 1 increments every 1 μs.
  - Timer 1 is 16-bit counter, so it will overflow every 2^{16} μs.
  - For a 2s delay, we need Timer 1 to overflow for: 2s/2^{16} μs = 31 times.

- **Implementation**
  - Write code to enable Timer 1 overflow interrupt.
  - Use ISR to count the number of overflows.
  - When the number of overflows is 31, invert port B.

Creating an accurate delay: timer_delay.c

```
#include <avr/io.h>
#include <avr/interrupt.h>                                                   1

volatile int overflow_count; // declare a global variable

ISR(TIMER1_OVF_vect){              // ISR for Timer1 overflow interrupt
   overflow_count++;              // increment overflow count
   if (overflow_count >= 31){ // when 2s has passed
        overflow_count = 0;     // start new count                          2
        PORTB = ~PORTB;         // invert port B
   }
}

int main(void) {
   DDRB  = 0xFF;          // set port B for output
   PORTB = 0x00;          // initial value of PORTB
   overflow_count = 0;  // initialise overflow count
   TCCR1A = 0b00000000; // normal mode                                      4
   TCCR1B = 0b00000001; // no prescaler, internal clock
                                                                            3
   TIMSK  = 0b00000100; // enable Timer 1 overflow interrupt
   sei();                 // enable interrupt subsystem globally           5

   while (1){;}          // infinite loop
   return 0;
}
```

Creating an accurate delay: Demo



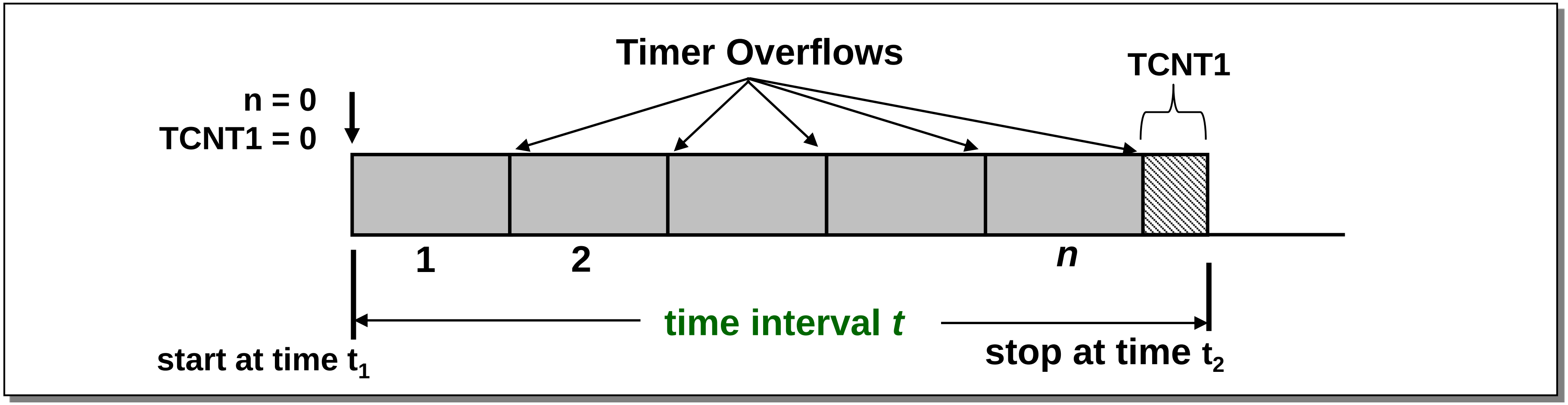
Using a timer to turn LEDs on/off periodically (Lab 3 – Task 1)

4.3.2 Measuring elapsed time

- To measure time using Timer 1, we must keep track of both
  - the number of times that Timer 1 has overflowed: n
  - the current counter value: TCNT1

- Reset n and TCNT1 at the beginning of the interval. The time elapse is (assuming no prescaler, 1MHz clock):

$$t = n \times 2^{16} + \text{TCNT1} \quad (\mu s)$$



Measuring elapsed time

Use Timer 1 to measure the execution time of some custom C code.

Approach

- **Clear Timer 1 when the custom code starts.**
- **Record Timer 1 when the custom code finishes.**
- **Use Timer 1 Overflow Interrupt to count the number of timer overflows.**

Measuring elapsed time: measure_time.c

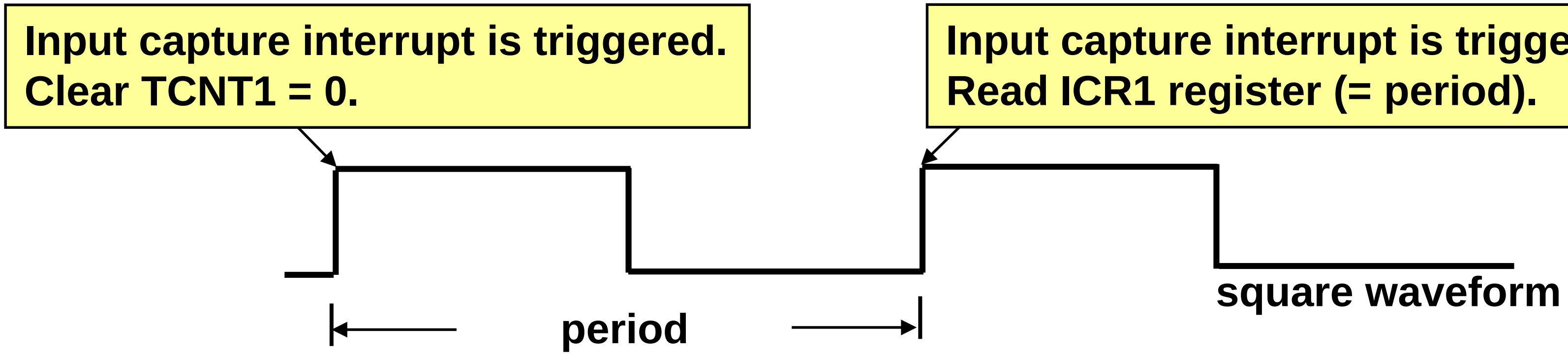
```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <inttypes.h>

volatile uint32_t n;   // uint32_t is unsigned 32-bit integer data type
ISR(TIMER1_OVF_vect){  // handler for Timer1 overflow interrupt
   n++;                // increment overflow count
}

int main(void) {
   int i, j;
   uint32_t elapse_time; // uint32_t is unsigned 32-bit integer data type

   TCCR1A = 0b00000000; // normal mode
   TCCR1B = 0b00000001; // no prescaler, internal clock
   TIMSK  = 0b00000100; // enable Timer 1 overflow interrupt

   n = 0;               // reset n
   TCNT1 = 0;           // reset Timer 1
   sei();               // enable interrupt subsystem globally
   // -----  start code --------------
   for (i = 0; i < 100; i++)
        for (j = 0; j < 1000; j++){;}
   // -----  end code -----------------
   elapse_time = (n << 16) + (uint32_t) TCNT1;
   cli();               // disable interrupt subsystem globally
   return 0;
}
```

any custom code

4.3.3 Measuring period of a square signal

Use Timer 1 input capture interrupt to measure the period of a square wave.

- **Analysis:**
  - The period of a square wave = the time difference between two consecutive rising edges.
  - Connect the square wave to Input Capture pin of Timer 1.
  - Configure input capture module to trigger on rising edges.



Measuring period of a square signal

- **Assumption**: The input signal has a high frequency, hence timer overflow can be ignored.

- **Implementation**:
  - **Select timer operations**: normal, no prescaler, internal clock 1MHz, noise canceller enabled, input capture for rising edges.

```
TCCR1A = 0b00000000;
TCCR1B = 0b11000001;
```

  - **Enable input capture interrupt**:

```
TIMSK = 0b00100000;
```

measure_period.c

```c
#include <avr/io.h>
#include <avr/interrupt.h>
#include <inttypes.h>     // header file for custom data types

// Global variable period is used to share data between ISR() and main()
volatile uint16_t period; // uint16_t is unsigned 16-bit integer

ISR(TIMER1_CAPT_vect){ // handler for Timer1 input capture interrupt
  period = ICR1;       // period = value of Timer 1 stored in ICR1
  TCNT1 = 0;           // reset Timer 1
}

int main(void) {
   DDRB   = 0xFF;         // set port B for output

   TCCR1A = 0b00000000; // normal mode
   TCCR1B = 0b11000001; // no prescaler, rising edge, noise canceller
   TIMSK  = 0b00100000; // enable Timer 1 input capture interrupt
   sei();               // enable interrupt subsystem globally
   while (1){           // infinite loop
        PORTB = ~(period >> 8);// Top 8-bit to PORT B: LED on=0, off=1
   }
   return 0;
}
```

Testing measure_period.c

- Connect Input Capture pin (D.6) to square wave output of Function Generator.
- Connect GRD pin of Port D to GRD pin of Function Generator.
- Connect Port B to LED connector.
- Compile, download program.
- Change frequency of square ware and observe output on LEDs.
- Demo: See later in Slide 58.

Extending measure_period.c

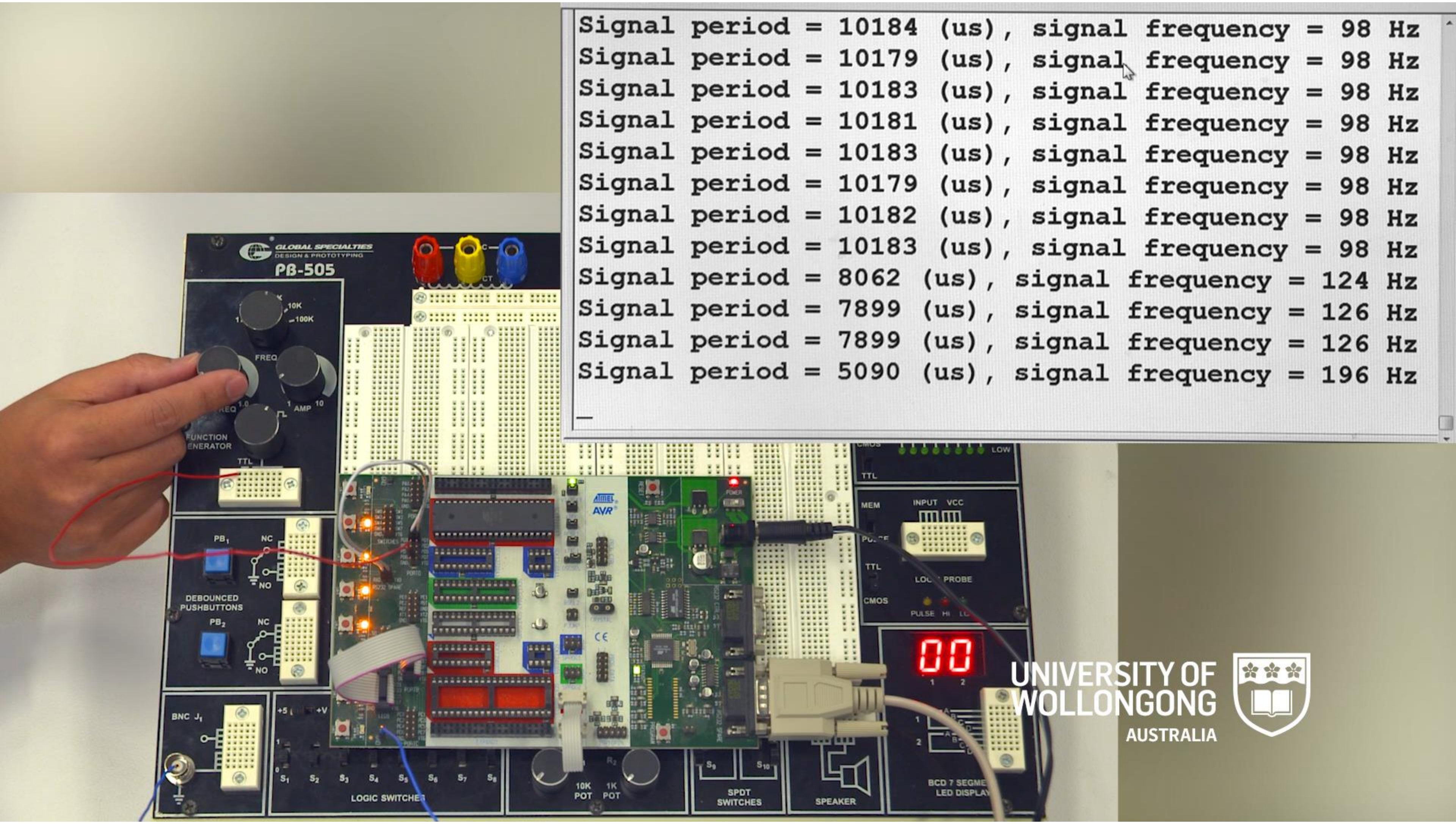
- This example assumes no timer overflow between two rising edges of the square signal.

- In Lab 3, you are required to extend the code to measure the period for low-frequency signals.

- It is necessary to intercept timer overflow (see Examples 4.3.1 & 4.3.2).

- For testing, the measured period is sent to the PC via serial port.

Extending measure_period.c



Using a timer to measure the period of a square signal (Lab 3 – Task 2)

Lecture 4's summary

- **Key points of this lecture**
  - **Writing an interrupt-driven program in C for ATmega16.**
  - **Programming serial and external interrupts in C.**
  - **Overview of timers in ATmega16.**
  - **Using Timer1 overflow and input capture interrupts in 3 applications.**

- **Next activities**
  - **Tutorial 3: 'Timers'.**
  - **Lab 3: 'Timers'**
    - **Complete the online Pre-lab Quiz for Lab 3.**
    - **Study video demos of Lab 3.**
    - **Write programs for Tasks 1 and 2 of Lab 3.**

Lecture 4's references

- M. A. Mazidi, S. Naimi, S. Naimi, AVR Microcontroller and Embedded Systems: Using Assembly and C, 2nd edition Pearson, 2015, [Chapters 9, 10, 15].

- J. Pardue, C Programming for Microcontrollers, 2005, SmileyMicros, [Chapter 7: Interrupts…].

- Atmel Corp., 8-bit AVR microcontroller with 16K Bytes In-System Programmable Flash ATmega16/ATmega16L, 2010, [Interrupts], [External Interrupts] and [Timers].

- S. F. Barrett and D. J. Pack, Atmel AVR Microcontroller Primer: Programming and Interfacing, 2008, Morgan & Claypool Publishers, [Chapter 5: Timing Subsystem].

Manual

Lecture 4's references

- M. Mazidi, J. Mazidi, R. McKinlay, "The 8051 microcontroller and embedded systems using assembly and C," 2nd ed., Pearson Prentice Hall, 2006, [Chapter 9].
- M. Mazidi and J. Mazidi, "The 8086 IBM PC and compatible computers," 4th ed., Pearson Prentice Hall, 2003, [Chapter 13].
- P. Spasov, "Microcontroller technology the 68HC11," 3rd ed., Prentice Hall, 1999, [Chapter 11].
- H. Huang, "MC68HC12 an introduction: software and hardware interfacing," Thomson Delmar Learning, 2003, [Chapter 8].