CAB202 Topic 9 – Timers and Interrupts

Authors:

- Luis Mejias, Lawrence Buckingham QUT (2020)

Contents

- Roadmap
- References
- De-bouncing (a problem that can be solved with interrupts)
  - The problem
  - Delay-based de-bouncing (which is not very good)
    - Idea
    - Implementation
    - Pros and Cons
  - Non-blocking de-bouncing (which is much better but not quite ideal)Æ’
    - Idea
    - Implementation
    - Pros and Cons
  - De-bouncing conclusion
- ATMega328P Timers
  - Timer Introduction
  - Timer0 registers (Datasheet, Section 15.9)
  - Case study: Set Up and Read Time From Timer0
- The Timer Overflow Interrupt
  - Caveat

    - What’s an interrupt?
    - Implementing Timer Overflow ISR
- Appendices
    - Appendix 1: Interrupt-based UART
    - Appendix 2: Bit-packed boolean arrays
    - Appendix 3: Digital I/O Cheat Sheet

Roadmap

Previously:

7. AVR ATMega328P Introduction to Microcontrollers; Digital Input/Output
8. Serial Communication – communicating with another computer/microcontroller

This week:

9. **Debouncing, Timers and Interrupts. Asynchronous programming.**

Still to come:

10. Analogue to Digital Conversion; Pulse Width Modulation (PWM); Assignment 2 Q&A..
11. LCD Display, sending digital signals to a device.

References

Recommended reading:

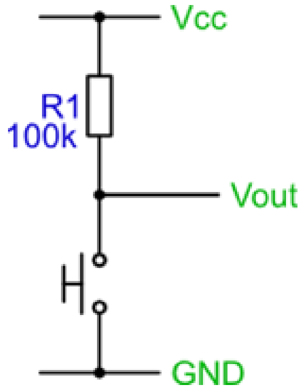
- Blackboard→Learning Resources→Microcontrollers→atmega328P_datasheet.pdf.

De-bouncing (a problem that can be solved with interrupts)

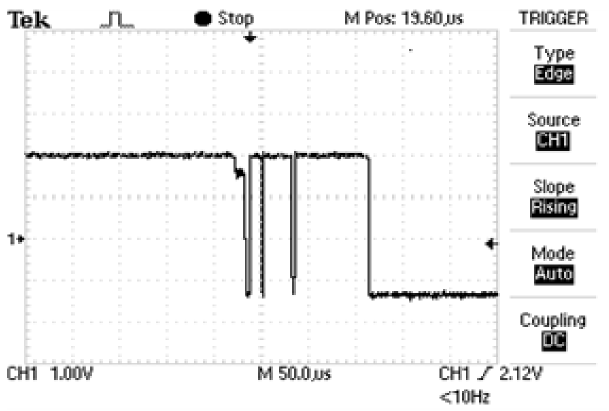
The problem

Switches are prone to a phenomenon known as *bouncing*, in which spurious “click” events are detected.

- Consider a pull-up resistor:



- When the switch opens and closes, current flows in an interrupted pattern while the connection is made, then settles to either on or off.



- Bouncing matters when we want precise recognition of “click” events.
  - A button click is recognised when a switch is *pressed* and then *released* as part of a single gesture.
  - This corresponds to a switch transitioning from *open* to *closed* and then back to *open*.

The **example1.c, BounceDemo** program illustrates bouncing.

```
#define F_CPU (16000000UL)

#include <avr/io.h>

/*
 *  Setting data directions in a data direction register (DDR)
 *
 *
 *  Setting, clearing, and reading bits in registers.
 *      reg is the name of a register; pin is the index (0..7)
 *  of the bit to set, clear or read.
 *  (WRITE_BIT is a combination of CLEAR_BIT & SET_BIT)
 */


#define SET_BIT(reg, pin)                   (reg) |= (1 << (pin))
#define CLEAR_BIT(reg, pin)               (reg) &= ~(1 << (pin))
#define WRITE_BIT(reg, pin, value)   (reg) = (((reg) & ~(1 << (pin))) | ((value) << (pin)))
#define BIT_VALUE(reg, pin)               (((reg) >> (pin)) & 1)
#define BIT_IS_SET(reg, pin)         (BIT_VALUE((reg),(pin))==1)


//Functions declaration
void setup(void);
void process(void);
void uart_init(unsigned int ubrr);
char uart_getchar(void);
void uart_putchar(unsigned char data);
void uart_putstring(unsigned char* s);

//uart definitions
#define BAUD (9600)
#define MYUBRR (F_CPU/16/BAUD-1)


//to store button press count
uint16_t counter = 0;


void setup(void) {

        uart_init(MYUBRR);

        // Enable B5 as output, led on B5
        SET_BIT(DDRB, 5);


      // Enable D6 and D7 as inputs
       CLEAR_BIT(DDRD,6);
       CLEAR_BIT(DDRD,7);

}

void process(void) {
```

```
    //define a buffer to be sent
     unsigned char temp_buf[64];

     snprintf( (char *)temp_buf, sizeof(temp_buf), "%d\n", counter );


   //detect pressed switch on D7
   if (BIT_IS_SET(PIND,7)){

     while ( BIT_IS_SET(PIND, 7) ) {
           // Block until switch released.
         }

    //increment count
         counter++;

   }

     //detect presed switch on D6
   if (BIT_IS_SET(PIND,6)){

    //send serial data
         uart_putstring(temp_buf);

     while ( BIT_IS_SET(PIND, 6) ) {
                 // Block until switch released.
                 }

     }

    //flash LED every time increments of 5 occur
     if(counter%5 == 0)
       PORTB^=(1<<PINB5);


}

int main(void) {

        setup();

        for ( ;; ) {
                process();

        }
}



/*  ****** serial uart definitions ************ */

// Initialize the UART
void uart_init(unsigned int ubrr) {

        UBRR0H = (unsigned char)(ubrr>>8);
    UBRR0L = (unsigned char)(ubrr);
```

```
        UCSR0B = (1 << RXEN0) | (1 << TXEN0);
        UCSR0C = (3<<UCSZ00);

}

// Transmit a data
void uart_putchar(unsigned char data) {

    while (!( UCSR0A & (1<<UDRE0))); /* Wait for empty transmit buffer*/

        UDR0 = data;            /* Put data into buffer, sends the data */

}


// Receive data
char uart_getchar(void) {


/* Wait for data to be received */
  while ( !(UCSR0A & (1<<RXC0)));


/* Get and return received data from buffer */
return UDR0;

}

// Transmit a string
void uart_putstring(unsigned char* s)
{
    // transmit character until NULL is reached
    while(*s > 0) uart_putchar(*s++);
}
```

TinkerCad implementation of the program:

https://www.tinkercad.com/things/4mpGeci84Mj

This program uses a polling approach to detect click events on the left switch .

- Click-detection is done in the **process** function. The logic is very simple:
  - If the left switch is closed, wait for it to become open, and then register a *click*.
  - Waiting with a loop in this way is called *busy waiting* – the CPU repeatedly executes the test in the while loop until the condition becomes false, at which point the program can move on.

- Busy-waiting is avoided wherever possible because it locks the CPU up doing nothing, and because it may result in unpredictable delays.

- Bouncing happens on a very short timescale relative to the physical act of pressing and releasing the switch, so it is quite a challenge to demonstrate on purpose.
  - If we introduce any significant delay, the effect is obscured.
  - To make the effect visible, this program performs absolutely minimal processing.
  - When a left button-click is detected, a counter increments silently.
  - To view the current value of the counter, push the right switch (it also send this value via uart).
  - Try multiple clicks on the left button and then display them by pressing right switch. Do the clicks and display values match?

Delay-based de-bouncing (which is not very good)

Idea

About the simplest work-around is to try to take advantage of the short duration of the transient behaviour.

- This click-detection algorithm is almost identical to the polling algorithm above.
- We introduce a short delay just after the button-press is detected, but before the busy-wait that detects the button-release.
- The hope is that the contact has closed properly by the time we enter the wait loop.

Implementation

Simple delay-based de-bouncing is demonstrated in **`example2.c, DelayDebounceDemo`**:

```
#define F_CPU (16000000UL)

#include <avr/io.h>

/*
 *  Setting data directions in a data direction register (DDR)
 *
 *
```

```
 *  Setting, clearing, and reading bits in registers.
 *      reg is the name of a register; pin is the index (0..7)
 *  of the bit to set, clear or read.
 *  (WRITE_BIT is a combination of CLEAR_BIT & SET_BIT)
 */


#define SET_BIT(reg, pin)                   (reg) |= (1 << (pin))
#define CLEAR_BIT(reg, pin)               (reg) &= ~(1 << (pin))
#define WRITE_BIT(reg, pin, value)   (reg) = (((reg) & ~(1 << (pin))) | ((value) << (pin)))
#define BIT_VALUE(reg, pin)               (((reg) >> (pin)) & 1)
#define BIT_IS_SET(reg, pin)         (BIT_VALUE((reg),(pin))==1)


//Functions declaration
void setup(void);
void process(void);
void uart_init(unsigned int ubrr);
char uart_getchar(void);
void uart_putchar(unsigned char data);
void uart_putstring(unsigned char* s);


//uart definitions
#define BAUD (9600)
#define MYUBRR (F_CPU/16/BAUD-1)

//delay using for debouncing
#define DEBOUNCE_MS (50)


//to store button press count
uint16_t counter = 0;


void setup(void) {

    //init uart
        uart_init(MYUBRR);

        // Enable B5 as output, led on B5
        SET_BIT(DDRB, 5);


      // Enable D6 and D7 as inputs
       CLEAR_BIT(DDRD,6);
       CLEAR_BIT(DDRD,7);
```

```
}

void process(void) {

   //define a buffer to be sent
    unsigned char temp_buf[64];

    snprintf( (char *)temp_buf, sizeof(temp_buf), "%d\n", counter );


  //detect pressed switch on D7
  if (BIT_IS_SET(PIND,7)){

   _delay_ms(DEBOUNCE_MS);

    while ( BIT_IS_SET(PIND, 7) ) {
                        // Block until switch released.
        }

   //increment counter
        counter++;

  }

    //detect presed switch on D6
  if (BIT_IS_SET(PIND,6)){

   //send serial data
        uart_putstring(temp_buf);

    while ( BIT_IS_SET(PIND, 6) ) {
                        // Block until switch released.
                }

    }

   //flash LED every time increments of 5 occur
    if(counter%5 == 0)
      PORTB^=(1<<PINB5);


}

int main(void) {
        setup();

        for ( ;; ) {
                process();
```

```
        }
}




/*  ****** serial definitions ************ */

// Initialize the UART
void uart_init(unsigned int ubrr) {

        UBRR0H = (unsigned char)(ubrr>>8);
    UBRR0L = (unsigned char)(ubrr);
        UCSR0B = (1 << RXEN0) | (1 << TXEN0);
        UCSR0C = (3 << UCSZ00);

}

// Transmit a data
void uart_putchar(unsigned char data) {

    while (!( UCSR0A & (1<<UDRE0))); /* Wait for empty transmit buffer*/

        UDR0 = data;            /* Put data into buffer, sends the data */

}

// Receive data
char uart_getchar(void) {

/* Wait for data to be received */
  while ( !(UCSR0A & (1<<RXC0)));


/* Get and return received data from buffer */
return UDR0;

}

// Transmit a string
void uart_putstring(unsigned char* s)
{
    // transmit character until NULL is reached
    while(*s > 0) uart_putchar(*s++);
}
```

TinkerCad implementation of the program:

https://www.tinkercad.com/things/g78AzAs2sgJ

Pros and Cons

Introducing a delay between detection of button-press and button-release "kind of" works. But it has some problems.

- If the delay is too short, it just doesn't work.
- If the delay is too long, then the user may click faster than we can detect.
- It is still unreliable because it still depends heavily on how fast the user do button-presses.
- Regardless of the delay duration, synchronous click detection in this way is a bad idea.
  - When we block the main event loop, everything stops until the click is detected.

Non-blocking de-bouncing (which is much better but not quite ideal)

Idea

Both algorithms above rely on a two-phase procedure to detect a click:

1. Detect that the switch is pressed.
2. Wait for the switch to be released.

The busy wait is the main problem: the event loop stalls while we wait for the switch to be released. In the present section we examine an elegant non-blocking solution.

- This algorithm hinges on the idea that at any given time the switch may be undergoing rapid on/off transitions due to bouncing, but that eventually the switch will settle to a stable configuration, at which point the button is either *definitely pressed* or *definitely not pressed*.
  - Initially, we have no opinion as to the configuration.
  - As time passes we accumulate a log of evidence which sways between the two options: *definitely pressed* or *definitely not pressed*.

- If enough evidence accumulates one way or the other, we accept the option.
- We then start again.

- To decide if the button is in a stable configuration, we repeatedly (at high frequency) sample the button state.
  - Every time we see the switch is closed, our opinion moves toward the conclusion that the button *may be* definitely pressed.
  - If we see the switch closed many times without ever seeing it open, we conclude that the button *is* definitely pressed.
  - Every time we see the switch open, we conclude that the button cannot possibly be definitely pressed, and our opinion moves toward the opposite conclusion, namely that the button *may not be* definitely pressed.
  - If we see the switch open many times without ever seeing it closed, we conclude that the button *is not* definitely pressed.

Implementation

An implementation of this algorithm is provided in **`example3.c, NonblockingDebounceDemo`**:

```
#define F_CPU (16000000UL)

#include <avr/io.h>

/*
 *  Setting data directions in a data direction register (DDR)
 *
 *
 *  Setting, clearing, and reading bits in registers.
 *      reg is the name of a register; pin is the index (0..7)
 *  of the bit to set, clear or read.
 *  (WRITE_BIT is a combination of CLEAR_BIT & SET_BIT)
 */


#define SET_BIT(reg, pin)                    (reg) |= (1 << (pin))
#define CLEAR_BIT(reg, pin)                (reg) &= ~(1 << (pin))
#define WRITE_BIT(reg, pin, value)   (reg) = (((reg) & ~(1 << (pin))) | ((value) << (pin)))
#define BIT_VALUE(reg, pin)                (((reg) >> (pin)) & 1)
#define BIT_IS_SET(reg, pin)         (BIT_VALUE((reg),(pin))==1)
```

```
//Functions declaration
void setup(void);
void process(void);
void uart_init(unsigned int ubrr);
char uart_getchar(void);
void uart_putchar(unsigned char data);
void uart_putstring(unsigned char* s);


//uart definitions
#define BAUD (9600)
#define MYUBRR (F_CPU/16/BAUD-1)

//threshold used for debouncing
#define THRESHOLD (1000)

// State machine for button pressed
bool pressed = false;
uint16_t closed_num = 0;
uint16_t open_num = 0;
uint16_t counter = 0;

void setup(void) {

    // initialise uart
      uart_init(MYUBRR);

        // Enable B5 as output, led on B5
        SET_BIT(DDRB, 5);
       // Enable D6 and D7 as inputs
       CLEAR_BIT(DDRD,6);
       CLEAR_BIT(DDRD,7);

}



bool left_button_clicked(void){

  bool was_pressed = pressed;

  if ( BIT_IS_SET(PIND, 7) ) {
                closed_num++;
                open_num = 0;

                if ( closed_num  > THRESHOLD ) {
```

```
                        if ( !pressed ) {
                                closed_num = 0;
                        }

                        pressed = true;
                }
        }
        else {
                open_num++;
                closed_num = 0;

                if ( open_num > THRESHOLD ) {
                        if ( pressed ) {
                                open_num = 0;
                        }

                        pressed = false;
                }
        }

        return was_pressed && !pressed;


}



void process(void) {

   //define a buffer sent
    unsigned char temp_buf[64];

    snprintf( (char *)temp_buf, sizeof(temp_buf), "%d\n", counter );


  //detect pressed switch on D7
  if (left_button_clicked() ){

   //increment count
        counter++;

  }

    //detect pressed switch on D6
  if (BIT_IS_SET(PIND,6)){

   //send serial data
        uart_putstring(temp_buf);
```

```
    while ( BIT_IS_SET(PIND, 6) ) {
    // Block until switch released.
        }

  }

   //flash LED every time increments of 5 occur
    if(counter%5 == 0)
    PORTB^=(1<<PINB5);


}

int main(void) {
        setup();

        for ( ;; ) {
           process();
        }
}




/*  ****** serial definitions ************ */

// Initialize the UART
void uart_init(unsigned int ubrr) {

        UBRR0H = (unsigned char)(ubrr>>8);
    UBRR0L = (unsigned char)(ubrr);
        UCSR0B = (1 << RXEN0) | (1 << TXEN0);
        UCSR0C = (3 << UCSZ00);

}

// Transmit a data
void uart_putchar(unsigned char data) {

    while (!( UCSR0A & (1<<UDRE0))); /* Wait for empty transmit buffer*/

        UDR0 = data;            /* Put data into buffer, sends the data */

}

// Receive data
char uart_getchar(void) {
```

```
/* Wait for data to be received */
  while ( !(UCSR0A & (1<<RXC0)));


/* Get and return received data from buffer */
return UDR0;

}

// Transmit a string
void uart_putstring(unsigned char* s)
{
    // transmit character until NULL is reached
    while(*s > 0) uart_putchar(*s++);
}
```

[TinkerCad](https://www.tinkercad.com/things/a2LDCDOPByq) implementation of the program:

https://www.tinkercad.com/things/a2LDCDOPByq

The algorithm is implemented in the **`left_button_clicked`** function.

- We use a boolean variable called **`pressed`** to record whether the button is *definitely pressed* or *definitely not pressed*.
- Along with **`pressed`**, we keep a pair of counters.
  - **`open_num`** is the number of consecutive times the switch has been observed to be open.
  - **`closed_num`** is the number of consecutive times the switch has been observed to be closed.
- Every time we poll the switch state, we update one or more of these three variables.
  - If the switch is closed, we increment **`closed_num`** and restore **`open_num`** to 0. If **`open_num`** passes a threshold (in this case, **`1000`**), we assign **`pressed = true`** –; *definitely pressed*.
  - If the switch is open, we increment **`open_num`** and restore **`closed_num`** to 0. If **`closed_num`** passes the threshold, we assign **`pressed = false`** –; not *definitely pressed*.

- Each time we settle on a switch state, we reset the counters and the accumulation process starts again.

Pros and Cons

This non-blocking de-bounce method is pretty good.

- The boolean `pressed` variable is a reliable indication of the true state of the button.
- Click tests based on transitions between `pressed == true` and `pressed == false` are very reliable.

Perceived problems:

- The decision depends on `THRESHOLD`, which must be just right.
  - The correct value will depend not only on the microcontroller clock speed, but also on the time between calls to `left_button_clicked`.
  - Performance may not be consistent due to factors elsewhere in the program.
  - *The algorithm polls the switch in the main event loop, which means we can never guarantee reliable performance*.

De-bouncing conclusion

- We have demonstrated switch bounce, and examined a two ways to address the problem.
- A non-blocking algorithm has been developed which is very good, but still relies on polling.
- To perfect the algorithm, we need a way to sample the physical state of the switch at a fairly high and constant frequency. **Hint: Timers and Interrupts**.

ATMega328P Timers

Timer Introduction

A timer is a semi-autonomous subsystem which runs alongside the CPU.

- *Refer: ATMega328P Datasheet, Chapters 15–17*.

- The timer executes a very simple program which listens to a clock signal.
    - By default, ATMega328P timers use use the built-in system clock which runs at 16,000,000 cycles per second (16MHz).
    - Clock signal can also come from external source.
    - After a fixed number of clock cycles, the timer updates a counter which occupies one or two 8-bit register.
    - The counter updates continuously as long as the timer is enabled.
    - When the counter reaches its maximum value, it wraps back to zero.
        - The timer can also trigger an interrupt when the counter overflows.
        - This is covered in the next section.
    - In addition to timekeeping, timers can be used to generate waveforms which in turn control external devices.
        - Removing processing load from CPU.
        - Pulse Width Modulation (PWM) will be covered in Topic 11.

ATMega328P has three timers:

- Timer 0: 8 bit timer (counter ranges from 0 to 255)
- Timer 1: 16 bit timer (counter range from 0 to 65,535)
- Timer 2: 8 bit timer (range from 0 to 255)

Each timer has a set of dedicated registers.

Timer0 registers (Datasheet, Section 15.9)

Each timer has a set of dedicated control and counter registers. Details are shown for Timer 0; you can look up the datasheet to find the corresponding registers for Timers 1 and 2.

- `TCCR0A` – Timer/Counter Control Register 0 A:

Bits	7	6	5	4	3	2	1	0

Name	**COM0A1**	**COM0A0**	**COM0B1**	**COM0B0**	-	-	**WGM01**	**WGM00**
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W
initially	0	0	0	0	0	0	0	0

- **COM0Ax** = Compare Match Output A Mode: leave this at 0
- **COM0Bx** = Compare Match Output B Mode: leave this at 0
- **WGM0x** = Waveform Generation Mode: leave this at 0
- **TL;DR – For our current purposes either ignore, or assign 0, to this register**

- **TCCR0B** – Timer/Counter Control Register 0 B:

Bits	7	6	5	4	3	2	1	0
Name	**FOC0A**	**FOC0B**	-	-	**WGM02**	**CS02**	**CS01**	**CS00**
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W
initially	0	0	0	0	0	0	0	0

  - **FOC0x** = Force output compare: leave these at 0.
  - **WGM02** = Waveform Generation Mode: leave this at 0.
  - **CS02,CS01,CS00** = Clock Select.
    Datasheet P 117, Table 15-9.
    These bits taken together form a 3-bit number which tells the timer how frequently to update the counter.
    The system clock speed is *pre-scaled* by dividing by the designated factor for each Clock Select combination.
    Figures in this table assume that the CPU speed is set to 16MHz in the **setup** *phase*.
    Values are:

CS02:0	Counter updates…
0b000	Never (Timer/Counter stopped)
0b001	Every clock cycle (No pre-scaling) == 16,000,000 ticks/sec
0b010	Every 8 clock cycles == 2,000,000 ticks/sec
0b011	Every 64 clock cycles == 250,000 ticks/sec
0b100	Every 256 clock cycles == 62,500 ticks/sec
0b101	Every 1024 clock cycles == 15,625 ticks/sec

0b110	External clock source on T0 pin. Clock on falling edge.
0b111	External clock source on T0 pin. Clock on rising edge.

- We will not be using CS02:0 == 6 or CS02:0 == 7.

- **`TCNT0`** – Timer/Counter Register 0: an 8-bit numeric value. **Where the count is stored.**
- **`OCR0A`** – Output Compare Register 0 A: an 8-bit numeric value. We will not be using this.
- **`OCR0B`** – Output Compare Register 0 B: an 8-bit numeric value. We will not be using this.
- **`TIMSK0`** – Timer/Counter Interrupt Mask Register 0:

Bits	7	6	5	4	3	2	1	0
Name	-	-	-	-	-	**OCIE0B**	**OCIE0A**	**TOIE0**
Read/Write	R	R	R	R	R	R/W	R/W	R/W
initially	0	0	0	0	0	0	0	0

  - **`OCIE0B`** = Force output compare: leave these at 0
  - **`OCIE0A`** = Force output compare: leave these at 0
  - **`TOIE0`** = Enable Timer Overflow Interrupt.

Case study: Set Up and Read Time From Timer0

In the present section we set up Timer 0, and see how to read the value of the clock.

- First, decide how fast we want the Timer/Counter register to update.
- Timer 0 is an 8 bit timer, so the Timer/Counter register will overflow every 256 ticks.
- We know the number of ticks per second from the datasheet, so we can calculate how long it will take for the timer to count from 0 to 255 (the overflow period) and how many times the counter will overflow per second (the overflow frequency).
  Definition: Frequency = 1 / Period.
  *Figures in this table assume that the CPU speed is set to 16MHz in the **`setup`** phase.*

CS02:0	Pre-scaler	Counter frequency	Overflow period = 256/freq	Overflow frequency
0b000	0	0	n/a	n/a
	1	16MHz	0.000016s	62.5kHz

0b001				
0b010	8	2MHz	0.000128s	7.8125kHz
0b011	64	250kHz	0.001024s	976.56Hz
0b100	256	62.500kHz	0.004096s	244.14Hz
0b101	1024	15.625kHz	0.016384s	61.035Hz

- Using the table, and balancing the update speed against our needs, we choose a pre-scaler.
- To set up Timer 0 to overflow about 61 times per second, we choose **`CS02:0 == 0b101 == 5`**, which corresponds to a pre-scaler of 1024.
- Starting the timer then consists of:
  - Set **`TCCR0A = 0;`**
  - Set **`TCCR0B = 5;`**
- To read the timer, we access **`TCNT0`**.
  - The value of the counter is a number of ticks.
  - To convert from ticks back to seconds, we multiply by the pre-scaler and divide by clock speed.
  - ```
    #define FREQ 16000000.0
    #define PRESCALE 1024.0
    double time = TCNT0 * PRESCALE / FREQ;
    ```

This procedure is demonstrated in **`example4.c, ReadTimer0`**

```
#define F_CPU (16000000UL)

#include <stdint.h>
#include <stdio.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

/*
 *  Setting data directions in a data direction register (DDR)
```
```

```
 *
 *
 *  Setting, clearing, and reading bits in registers.
 *      reg is the name of a register; pin is the index (0..7)
 *  of the bit to set, clear or read.
 *  (WRITE_BIT is a combination of CLEAR_BIT & SET_BIT)
 */


#define SET_BIT(reg, pin)                   (reg) |= (1 << (pin))
#define CLEAR_BIT(reg, pin)               (reg) &= ~(1 << (pin))
#define WRITE_BIT(reg, pin, value)   (reg) = (((reg) & ~(1 << (pin))) | ((value) << (pin)))
#define BIT_VALUE(reg, pin)               (((reg) >> (pin)) & 1)
#define BIT_IS_SET(reg, pin)         (BIT_VALUE((reg),(pin))==1)


//Functions declaration
void setup(void);
void process(void);
void uart_init(unsigned int ubrr);
char uart_getchar(void);
void uart_putchar(unsigned char data);
void uart_putstring(unsigned char* s);
void ftoa(float n, char * res, int afterpoint);
int intToStr(int x, char str[], int d);
void reverse(char * str, int len);


//uart definitions
#define BAUD (9600)
#define MYUBRR (F_CPU/16/BAUD-1)


//timer definitions
#define FREQ (16000000.0)
#define PRESCALE (1024.0)

void setup(void) {


     // initialise uart
      uart_init(MYUBRR);


        // Timer 0 in normal mode, with pre-scaler 1024 ==> ~61Hz overflow.
        TCCR0A = 0;
        TCCR0B = 5;
```

```
        /*
        Alternatively:
                CLEAR_BIT(TCCR0B,WGM02);
                SET_BIT(TCCR0B,CS02);
                CLEAR_BIT(TCCR0B,CS01);
                SET_BIT(TCCR0B,CS00);
        */
}


void process(void) {

        char temp_buf[64];
    char *ch;

        double time = (double) TCNT0 * PRESCALE / FREQ;

    //convert float to a string
    ftoa(time, temp_buf, 4);

         //send serial data
         uart_putstring((unsigned char *)temp_buf);
     uart_putchar('\n');
}

int main(void) {
        setup();

        for ( ;; ) {
                process();
       _delay_ms(500);
        }
}



/*  ****** auxiliary functions ************ */

// Reverses a string 'str' of length 'len'
void reverse(char * str, int len) {
  int i = 0, j = len - 1, temp;
  while (i < j) {
    temp = str[i];
    str[i] = str[j];
    str[j] = temp;
    i++;
    j--;
  }
```

```
}

// Converts a given integer x to string str[].
// d is the number of digits required in the output.
// If d is more than the number of digits in x,
// then 0s are added at the beginning.
int intToStr(int x, char str[], int d) {
  int i = 0;
  while (x) {
    str[i++] = (x % 10) + '0';
    x = x / 10;
  }

  // If number of digits required is more, then
  // add 0s at the beginning
  while (i < d)
    str[i++] = '0';

  reverse(str, i);
  str[i] = '\0';
  return i;
}

// Converts a floating-point/double number to a string.
void ftoa(float n, char * res, int afterpoint) {
  // Extract integer part
  int ipart = (int) n;

  // Extract floating part
  float fpart = n - (float) ipart;

  // convert integer part to string
  int i = intToStr(ipart, res, 0);

  // check for display option after point
  if (afterpoint != 0) {
    res[i] = '.'; // add dot

    // Get the value of fraction part upto given no.
    // of points after dot. The third parameter
    // is needed to handle cases like 233.007
    fpart = fpart * pow(10, afterpoint);

    intToStr((int) fpart, res + i + 1, afterpoint);
  }
}
```

```
/*  ****** serial definitions ************ */

// Initialize the UART
void uart_init(unsigned int ubrr) {

        UBRR0H = (unsigned char)(ubrr>>8);
    UBRR0L = (unsigned char)(ubrr);
        UCSR0B = (1 << RXEN0) | (1 << TXEN0);
        UCSR0C = (3 << UCSZ00);

}

// Transmit a data
void uart_putchar(unsigned char data) {

    while (!( UCSR0A & (1<<UDRE0))); /* Wait for empty transmit buffer*/

        UDR0 = data;            /* Put data into buffer, sends the data */

}

// Receive data
char uart_getchar(void) {

/* Wait for data to be received */
  while ( !(UCSR0A & (1<<RXC0)));


/* Get and return received data from buffer */
return UDR0;

}

// Transsmit a string
void uart_putstring(unsigned char* s)
{
    // transsmit character until NULL is reached
    while(*s > 0) uart_putchar(*s++);
```

```
}
```

TinkerCad implementation of the program:

https://www.tinkercad.com/things/cXJt5X0OLiP

The Timer Overflow Interrupt

Caveat

Today we introduce interrupts in a superficial manner. We will encounter them again in subsequent topics.

What’s an interrupt?

Refer: Datasheet chapter 12, Section 12.4.

An *interrupt* is a signal which is generated in response to an internal or external event (or change of state).

Examples:

- Pin change.
- Serial transfer complete
- Timer overflow
- + plenty more.

Special functions called *Interrupt Service Routines* (also referred to as *interrupt handlers*) can be set up to process interrupts and are called in response to an event.

- A list of interrupt vectors may be found on datasheet, p74.

- Implementing an ISR is much the same as any other function.
- The main difference is that we use one of the pre-defined macros to declare our interrupt.
- In the present section, we will implement an ISR for the Timer 0 Overflow interrupt.
- The ISR will be called automatically every time the Timer/Counter 0 register overflows (this is the event that triggers the interrupt).

When an interrupt occurs and an ISR is implemented for that interrupt:

1. The CPU temporarily stops whatever it is doing, but keeps a record of the state of the computation.
2. It then turns off interrupts so the ISR can run unimpeded.
3. The ISR is called, like a regular function.
4. After the ISR finishes, the CPU re-enables interrupts and then continues where it left off.

ISRs must use special global variables to transfer data.

- ISRs cannot accept parameters, and cannot return a value.
- Variables that may be changed by an ISR must be marked with the **`volatile`** keyword.
- **`volatile`** ensures that the compiler generates the right instructions to let other non-ISR code read the variables.

Implementing Timer Overflow ISR

This subsection shows how to implement a Timer Overflow ISR for Timer 0.

- Timer setup is much the same as the previous example.
- We add two more instructions:
    - **`TIMSK0 = 1;`** enables the Timer Overflow interrupt for Timer 0. The same result would be obtained in this program by writing **`TIMSK0 |= (1<<TOIE0);`** – clarity is in the eye of the beholder.
    - **`sei();`** enables interrupts.
- The ISR is defined as a function with the signature **`ISR(TIMER0_OVF_vect)`**.

This procedure is demonstrated in **`example5.c, TimerOverflow0`**

```
#define F_CPU (16000000UL)

#include <stdint.h>
#include <stdio.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

/*
 *  Setting data directions in a data direction register (DDR)
 *
 *
 *  Setting, clearing, and reading bits in registers.
 *      reg is the name of a register; pin is the index (0..7)
 *  of the bit to set, clear or read.
 *  (WRITE_BIT is a combination of CLEAR_BIT & SET_BIT)
 */


#define SET_BIT(reg, pin)                   (reg) |= (1 << (pin))
#define CLEAR_BIT(reg, pin)                 (reg) &= ~(1 << (pin))
#define WRITE_BIT(reg, pin, value)   (reg) = (((reg) & ~(1 << (pin))) | ((value) << (pin)))
#define BIT_VALUE(reg, pin)                 (((reg) >> (pin)) & 1)
#define BIT_IS_SET(reg, pin)         (BIT_VALUE((reg),(pin))==1)

//Functions declaration
void setup(void);
void process(void);
void uart_init(unsigned int ubrr);
char uart_getchar(void);
void uart_putchar(unsigned char data);
void uart_putstring(unsigned char* s);
void ftoa(float n, char * res, int afterpoint);
int intToStr(int x, char str[], int d);
void reverse(char * str, int len);


//uart definitions
#define BAUD (9600)
#define MYUBRR (F_CPU/16/BAUD-1)


//timer definitions
#define FREQ (16000000.0)
```

```
#define PRESCALE (1024.0)

void setup(void) {

    // initialise uart
      uart_init(MYUBRR);

        // Timer 0 in normal mode, with pre-scaler 1024 ==> ~60Hz overflow.
        // Timer overflow on.
        TCCR0A = 0;
        TCCR0B = 5;
        TIMSK0 = 1;

        /*
        Alternatively:
                CLEAR_BIT(TCCR0B,WGM02);
                SET_BIT(TCCR0B,CS02);
                CLEAR_BIT(TCCR0B,CS01);
                SET_BIT(TCCR0B,CS00);
                SET_BIT(TIMSK0, TOIE0);
        */

    // Enable B5 as output, led on B5
        SET_BIT(DDRB, 5);

        // Enable timer overflow, and turn on interrupts.
        sei();
}


volatile int overflow_counter = 0;

ISR(TIMER0_OVF_vect) {
        overflow_counter ++;
}

void process(void) {

    char temp_buf[64];
    char *ch;

    //compute elapsed time
        double time = ( overflow_counter * 256.0 + TCNT0 ) * PRESCALE  / FREQ;

    //convert float to a string
    ftoa(time, temp_buf, 4);

         //send serial data
```

```
        uart_putstring((unsigned char *) temp_buf);
        uart_putchar('\n');

    //flash LED every cycle
         PORTB^=(1<<PINB5);


}

int main(void) {
        setup();

        for ( ;; ) {
                process();
                _delay_ms(1000);
        }
}


/********** auxiliary functions *************/


// Reverses a string 'str' of length 'len'
void reverse(char * str, int len) {
  int i = 0, j = len - 1, temp;
  while (i < j) {
    temp = str[i];
    str[i] = str[j];
    str[j] = temp;
    i++;
    j--;
  }
}

// Converts a given integer x to string str[].
// d is the number of digits required in the output.
// If d is more than the number of digits in x,
// then 0s are added at the beginning.
int intToStr(int x, char str[], int d) {
  int i = 0;
  while (x) {
    str[i++] = (x % 10) + '0';
    x = x / 10;
  }

  // If number of digits required is more, then
  // add 0s at the beginning
  while (i < d)
```

```
    str[i++] = '0';

  reverse(str, i);
  str[i] = '\0';
  return i;
}

// Converts a floating-point/double number to a string.
void ftoa(float n, char * res, int afterpoint) {
  // Extract integer part
  int ipart = (int) n;

  // Extract floating part
  float fpart = n - (float) ipart;

  // convert integer part to string
  int i = intToStr(ipart, res, 0);

  // check for display option after point
  if (afterpoint != 0) {
    res[i] = '.'; // add dot

    // Get the value of fraction part upto given no.
    // of points after dot. The third parameter
    // is needed to handle cases like 233.007
    fpart = fpart * pow(10, afterpoint);

    intToStr((int) fpart, res + i + 1, afterpoint);
  }
}





/********* serial definitions ***************/

// Initialize the UART
void uart_init(unsigned int ubrr) {

        UBRR0H = (unsigned char)(ubrr>>8);
    UBRR0L = (unsigned char)(ubrr);
        UCSR0B = (1 << RXEN0) | (1 << TXEN0);
        UCSR0C = (3 << UCSZ00);

}
```

```
// Transmit a data
void uart_putchar(unsigned char data) {

    while (!( UCSR0A & (1<<UDRE0))); /* Wait for empty transmit buffer*/

        UDR0 = data;            /* Put data into buffer, sends the data */

}

// Receive data
char uart_getchar(void) {

  /* Wait for data to be received */
  while ( !(UCSR0A & (1<<RXC0)) );


/* Get and return received data from buffer */
return UDR0;

}

// Transmit a string
void uart_putstring(unsigned char* s)
{
    // transmit character until NULL is reached
    while(*s > 0) uart_putchar(*s++);
}
```

TinkerCad implementation of the program:

https://www.tinkercad.com/things/dmEso1BGcrW

- In the ISR, we increment a counter to record how many times the timer has overflowed.
- In process, we multiply the counter by 256 (the number of ticks per overflow) and add the residual value of **TCNT0** to get the total number of elapsed ticks. This is then multiplied by the scaling factor to convert to the number of elapsed seconds since the program started.
- Note that in this implementation the counter wraps around and becomes negative when it passes 32,767. We could address this by counting with some wider numeric type.

Additional exercices:

1. Use timers to flash three LEDs at 1Hz, 3Hz, 10Hz, respectively.
2. Use a switch to trigger an interrupt. This interrupt will perform an action, for example toggle the state of an LED.

Appendices

Appendix 1: Interrupt-Based UART

- This example provides an alternative to perform UART communications using interrupts. It does work in a non-blocking mode. Link to example here: https://www.tinkercad.com/things/kcQ9NqHHy3Q

Appendix 2: Bit-packed boolean arrays

- Microcontrollers typically have limited RAM, so when writing complex programs we take every chance to economise on memory use.
- This subsection shows how we can use bit-level operations in an orderly way to pack multiple boolean values into simple variables, emulating a packed array of boolean.
- The key idea is as follows:
  - A variable of type **`uint8_t`** has 8 bits, so it can be used to remember up to 8 independent YES/NO values.
  - A variable of type **`uint16_t`** has 16 bits, so it can be used to remember up to 16 independent YES/NO values.
  - A variable of type **`uint32_t`** has 32 bits, so it can be used to remember up to 32 independent YES/NO values.
- The trick is to use bitwise operators:
  - **`SET_BIT`** stores a YES value in a packed array – **`SET_BIT(collection,i)`** is analogous to **`collection[i] = true`**;

  - `CLEAR_BIT` stores a NO value in a packed array – `CLEAR_BIT(collection,i)` is analogous to `collection[i] = false`;
  - `BIT_IS_SET` asks if the value is YES – `if (BIT_IS_SET(collection,i)) { /* do something */ }` is analogous to `if (collection[i]) { /* do something */ }`;
- Practical application:
  - Remembering the state of a collection of switches.

Appendix 3: Digital I/O Cheat Sheet

Code snippets for frequently used registers:

Digital I/O	Data direction register	Detect	Turn on	Turn Off
Switch connected to PF6	`CLEAR_BIT(DDRF, 6)`	`BIT_IS_SET(PINF, 6)`	n/a	n/a
LED connected to PB2	`SET_BIT(DDRB, 2)`	n/a	`SET_BIT(PORTB, 2)`	`CLEAR_BIT(PORTB, 2)`

The End
