AVR Course

Erik Buer 2016

In collaboration with Atmel®

Contents

1 Preface

This document was written by Erik Buer in preparation for the 2016 Elektra AVR-course.
Thanks to Magnus Mørk and Erik Hole for proofreading.

2 What is a microcontroller?

A microcontroller is a computer on a chip! It contains all necessary components to operate: Flash memory, RAM, and CPU among other things. Microcontrollers comes in all sizes, 4-,8-,16-,32-bit and upwards. Our ATmega324PB has an 8-bit architecture. This means that the microcontrollers registers and internal databus and ALU (Arithmetic and Logic Unit) have a size of 8 bits. The ATmega architecture have some 16 bit registers, but they are loaded and operated one byte at a time. Most 8-bit operations can be performed in one clock cycle, 16-bit operations can be orders of magnitude longer and is therefore more time and power-consuming. The architectures instruction set lists all possible operations the architecture can perform with the amount of clock cycles necessary. The AVR instruction set can be found at: http://www.atmel.com/images/atmel-0856-avr-instruction-set-manual.pdf

Most microcontroller families are what is known as a RISC (Reduced instruction set computer). RISC's have a simple versatile instruction set. These simple instructions are however fast (most instructions are executed in a single clock cycle!). Additional RISC architectures include ARM, PowerPC, Hitachi SuperH among others.

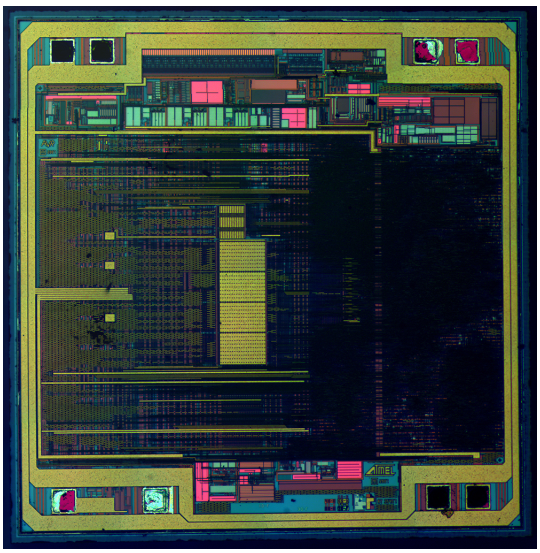

Figure 1: ATtiny13 silicon die

3 Where to start

To start a new project simply go to File > new > project, or press Ctrl+Shift+N.
In the "New Project" window select GCC C Executable Project, give the project a name then "OK". Choose your device in the device selector (in our case ATmega324PB), then click "OK". You should now have a file looking like this:

```
1  /*
2  * empty_app.c
3  *
4  * Created: 18.02.2016 19:36:26
5  * Author : Erik
6  */
7
8  #include <avr/io.h>
9
10
11 int main(void)
12 {
13     /* Replace with your application code */
14     while(1)
15     {
16     }
17 }
```

The application code is placed in the "main" function. Code that only needs to run once should be placed in "main" above the while(1) loop (line 13). This is equivalent to "`void setup()`" in the Arduino environment. The main program should be inside the while(1) loop, equivalent to "`void loop()`".

4 I/O architecture

Microcontrollers come with anything from a couple of IO pins to hundreds. Our ATmega324PB has 38 GPIO pins (general purpose in/out) spread across 5 ports. Each port has an index letter (A-E), and up to 8 physical pins. Each PORT has 3 control registers; DDRX, PORTX and PINX. Some physical pins may contain other functions like ADC, PWM or communication busses like SPI, I^2C (TWI), UART etc. The different functions are listed and explained thoroughly in the datasheet. The I/O structure is as follows:

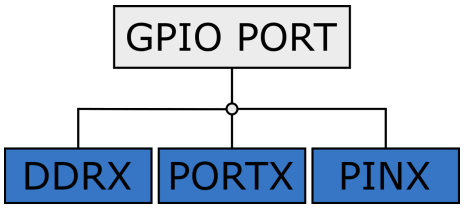

Figure 2: PORT architecture

4.1 Data Direction Register (DDRX)

DDRX is an 8-bit register which controls the data direction of the physical pins connectet to the PORT. Writing a 1 to DDRX configures the corresponding pin as an output; Writing a zero to DDRX configures the physical pin as an input. All pins are configured as inputs by default. You can read and write from this register as any other. Reading this register will tell you the data direction of a physical pin.

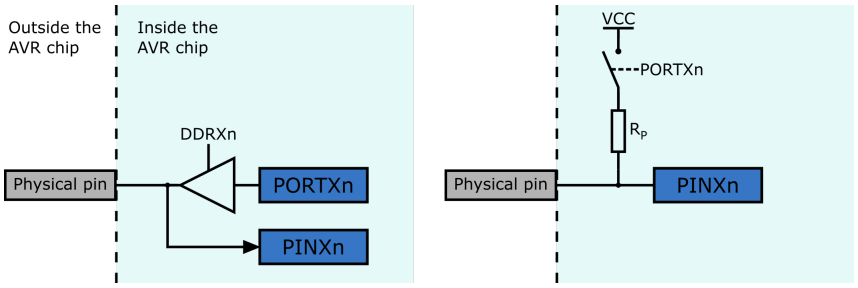

Figure 3: AVR output architecture

4.2 PORT data register (PORTX)

PORTX is an 8-bit register which controls the state of the physical pins connected to the PORT. If a physical pin is configured as an output, the PORT register controls whether the pin is in a high or low state. Writing a 1 to the PORTX register set's the corresponding physical pin in a high state. Writing a zero to the PORTX register configures the corresponding pin to a low state.
If the physical pin is configured as an input, the PORTX register controls whether the physical pin is in a state of high impendance, or has an internal pull-up ressistor. The internal pull-up resisor has a resistance anywhere from 20 to 50kΩ (on the ATmega324PB). The resistance varies with different VCC's, and from chip to chip. The pull-up resistance range is listed in each processor's datasheet. The pull-up resistor is used to avoid floating pins. A pin is floating if it isn't connectet to a voltage level. The logical state of a floating pin is somewhat random and can vary, causing power consumption.
You can read and write from this register. Reading the register will tell you the configuration (high/low, high Z/pull-up) of a physical pin.

4.3 PORT input register (PINX)

PINX is the PORT's input register. This 8-bit register holds the state of the physical pins. Reading this register will tell you whether a pin is in a low or high state. This register also has an alternate function. Writing to PINX will toggle the state of the corresponding PORTX register.

5 Bit manipulation

To write to a register or variable we simply write `int a = /*value */`. This updates the whole byte. The μC uses bits in special register to configure peripherals and I/O. To update a single bit we need to know the state of the whole byte! To change bits independently we use bitwise operations and **bit shifting**. AVR GCC supports multiple numbersystems. See figure 4.

Number systems	Designator	Example
Decimal	10,	10, 250
Hexadecimal	0x	0xfa
Octal	010,	010, 372
Binary	0b	0b11111010

Figure 4: Different number types in AVR GCC

5.1 Setting a bit

To set a bit we OR the target register with a byte which holds the bit(s) we want to set. By writing "`(1<<BIT)`" we create an empty byte with one bit in the place of the target bit. For example (1 << 2) is equivalent to 000 0100. The generic expression for setting a bit is:

```
BYTE |= (1<<BIT);
```

"`BYTE`" is a macro for a memory adress. The `BIT` is simply a number between 0 and 7, which represent a place in an arbitrary byte, see figure 8 " PORT macro values". The compiler doesn't care which bit-macro is used, it simply extracts the value of the macro. Therefore it doesn't matter if you use `PORTXn`,`PINXn`,`DDRXn`,`DDXn`, a number, or any other macro you may choose. It is possible to create custom macros, see section 8 " Creating macros". All register and bit-macros are located in the avr/io.h libary.

Let's say we want to set the third bit in the `PORTB` register. We turn to:

```
PORTB |= (1<<PORTB2);
```

Target byte	XXXX XXXX
Bitmask	OR 0000 0100
Output	= XXXX X1XX

Figure 5: Setting a bit

5.2 Clearing a bit

Clearing a bit is similar to setting a bit. First we create a bitmask "`~(1<<BIT)`", then we AND the bitmask with the target byte. A practical example:

```
PORTB &= ~(1<<PORTB2);
```

Target byte	XXXX XXXX
Bitmask	& 1111 1011
Output	= XXXX X0XX

Figure 6: Clearing a bit

5.3 Toggling a bit

To toggle a bit we XOR a bitmask with the target byte like so:

```
PORTX ^=  (1<<PORTXn);
```

We can alternatively write to the `PIN` register to toggle the `PORTX` register.

```
PINX |=  (1<<PORTXn);
```

Target byte	XXXX XXXX
Bitmask	XOR 0000 0100
Output	= XXXX X$\bar{X}$XX

Figure 7: Toggling a bit (notice bit 2 is inverted)

6 Reading a register

To read from a register or variable, simply read the macro. An example would be to give a variable the value of PINB as written below.

```
uint8_t a = PINB;
```

6.1 Reading a bit

Reading a particular bit is somewhat trickier. Lets say you want to run a function if the third pin on PINB is pulled high. We now need the AVR to recognize a single bit in the PINB register. We turn to bitmasks and write as follows:

```
1  void coolfunction();    // function prototype
2
3  int main(void)
4  {
5      // PINB2 is an input by default
6      while(1)
7      {
8          if(PINB & (1<<PINB2))
9          {
10             coolfunction();
11         }
12     }
13 }
```

Note that the if-statement interprets boolean values. If we were to write `if(PINB)` the statement would be true if ONE of the 8 bits had the logical value of 1.

7 Delay

The library delay.h has a simple delay function. We need to define F_CPU before including util/delay.h. The delay function is handy during development but I discourage you from using it in any timing applications. For more accurate timing we use the timer/counter module. There are two basic delay functions, _delay_ms(), and _delay_us().

7.1 Delay example

```
1  #define F_CPU   1000000UL
2
3  #include<avr/io.h>
4  #include<util/delay.h>
5
6  int main(void)
7  {
8      DDRB |= (1<<PORTB2);          // PB2 set as output
9
10     while(1)
11     {
12         PORTB ^= (1<<PORTB2);
13         _delay_ms(100)            // a ca 100 milisecond delay
14     }
15 }
```

8 Creating macros

A macro is a name for a piece of code. Whenever the macro is called it is replaced by the code it represents. Macros can represent a constant, a variable or a function. This simplifies writing and reading the code. avr/io.h contain macros for all registers and bits. It might be preferable to give pins, registers or functions application-specific names. The syntax for creating a macro is as follows:

```
#define name        /* piece of code */
```

Some practical examples:

```
1  #define toggle_pin  PORTB ^=  (1<<PORTB2)
2  #define io_reg      DDRB
3  #define led         PORTB2
```

To write multiple lines of code in a single macro we use a "do{}while(0);"

```
#define name        do{/*code*/;/*code*/;}while(0);
```

7	6	5	4	3	2	1	0
PORTX7	PORTX6	PORTX5	PORTX4	PORTX3	PORTX2	PORTX1	PORTX0
DDRX7	DDRX6	DDRX5	DDRX4	DDRX3	DDRX2	DDRX1	DDRX0
PINX7	PINX6	PINX5	PINX4	PINX3	PINX2	PINX1	PINX0

Figure 8: PORT macro values

9 Switch bounce

When dealing with buttons and switches it is important to compensate for button bouncing. Bouncing happens when a button is pressed, and released. Ideally the input goes low instantly until the button is released. The reality is a little different. Due to mechanical strain and elasticity in the mechanism, the button will bounce on contact and separation. See figure 9. Due to bouncing the AVR will read a random sequence of highs and lows which might affect your program.
There are two ways to deal with button bouncing, one is using a filter (parallel coupled capacitor see figure 10), the other is using a software debounce algorithm. This sounds worse than it is, all we have to do is check if the believed logic level repeats itself when tested after some time.

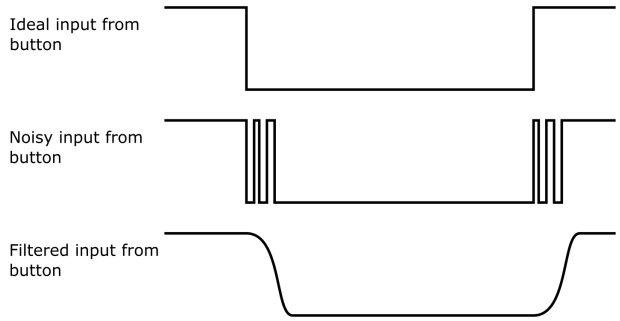

Figure 9: Switch bounce

Note that "Filtered input from button" is a simplified approximation, there will be more ripple in reality.

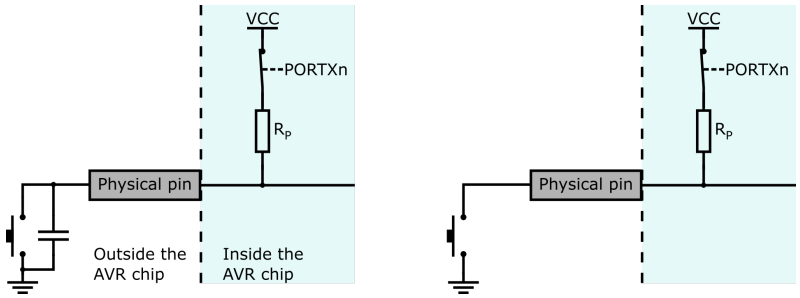

Figure 10: Hardware debounce example

9.1 Software debounce

In a software debounce algorithm we check the pin state multiple times to verify that the button was in fact pressed.

if *button is pressed* **then**
  wait;
  if *button is still pressed* **then**
    // do something
  end
end

```
1  int main(void)                              // Software debounce
2  {
3      while (1)
4      {
5          if (!(PINC & (1<<btn)))             // Is button SW0 pressed?
6          {
7              _delay_ms(100);
8
9              if (!(PINC & (1<<btn)))         // Is button SW0 still pressed?
10             {
11                 toggle_led;
12
13                 // Wait until button is released
14                 while(!(PINC & (1<<btn)));
15             }
16         }
17     }
18 }
```

This is an example of a debounce routine. We only had to check the pin state two times to get a reliable result. We can also build the debounce routine as a function:

```
1  #define button_check     !(PINC & (1<<btn))
2  #define toggle_led;      PORTC ^= (1<<led)
3
4  uint8_t button_pressed()
5  {
6      if (button_check)              // Is button SW0 pressed?
7      {
8          _delay_ms(100);
9          if (button_check)          // Is button SW0 still pressed?
10         {
11             _delay_ms(100);
12             return(1);
13         }
14         else
15         {
16             return(0);
17         }
18     }
19     return(0);
20 }
```

`button_pressed` will now check if the pin is pressed and verify with the debounce routine. If the button in fact is pressed the function returns a logical 1.

```
1 if(button_pressed)
2 {
3     toggle_led;
4 }
```

10 Interrupts

Most microcontrollers have one core, this means that it can only execute one task at a time. But no need to worry, thanks to interrupts we can "interrupt" the main program whenever an important task need to me executed. There are two types of interrupts: hardware and software. The AVR only has hardware interrupts, but it is possible to create software-accessible interrupts through PCINT. When an interrupt is launced the processor drops whatever it was doing, runs the task linked to the interrupt, then returns to its initial task. An interrupt can come from all kinds of functions in the IC. We are going to focus the EXTINT (external interrupts) and interrupts from the timer/counter module.

All interrupts are disabled by default. Each interrupt is enabled by setting a bit in each interrupts respective control register. We also need to enable the global interrupt mask, this bit enables or disables ALL interrupts in the μC. To set the global interrupt mask we use the macro `sei()`, and to clear the global interrupt mask we use `cli()`. When an interrupt occurs the mcu runs the code defined in an `ISR()` (interrupt service routine). The ISR is linked to interrupt vectors as so; `ISR(PCINT2_vect)`. This ISR runs every time the PCINT2 interrupt occurs. A list of all interrupt vectors can be found here: http://www.atmel.com/webdoc/AVRLibcReferenceManual/group__avr__interrupts.html

10.1 Linking vectors

You can link vectors to prevent multiple ISR's for a single task. We link vectors like so:

```
1 ISR(PCINT2_vect)
2 {
3     // Code to handle the event.
4 }
5 ISR(PCINT1_vect, ISR_ALIASOF(PCINT2_vect));
```

`ISR_ALIASOF` "redirects" `PCINT1_vect` to `PCINT2_vect` (line 5).

10.2 External interrupts

The AVR architecture has two different types of external interrupts; PCINTn, and INTn. INTn is the most advanced of the two. INTn can be configured to trigger on rising/falling edge, low level or any logical change on the corresponding physical pin. The PCINTn only register a logical change. Most GPIO's have PCINT capabilities.

10.2.1 Setting up PCINT for SW0

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

#define btn          PORTC6  // sw0 button
#define led          PORTC7  // indicator leds
void coolfunction();         // arbitrary function

int main(void)
{
    DDRC  &= ~(1<<btn);      // button pin as input
    PORTC |=  (1<<btn);      // internal pull-up on the sw0 pin

    PCICR |= (1<<PCIE2);     // Activates external interrupts on PCI0
    PCMSK2|= (1<<PCINT22);   // Activates PCINT for pcint22 (PC6)
    sei();                   // set global interrupt mask

    while(1)
    {
        /* application code */
    }
}
ISR(PCINT2_vect)
{
    coolfunction();
}
```

11 Timer/Counter

The `_delay_ms()` function works by performing the operation `asm("nop")` (does nothing) for as long as the delay lasts. This means that the μC is busy during a delay. Doing accurate time measurements and timing multiple actions at once would be quite tricky using only the delay function. Thankfully the AVR has multiple hardware timer/counter modules. Some are asynchronous of the MCU's clock, which means that the timer runs independent from the μC clock (Assuming you use an external clock source). The timer/counter work by counting clock cycles and comparing with a predetermined value (stored in `OCRnX`). Once the timer reaches the predetermined value an interrupt flag is set (`OCFX`), see figure 11. The interrupt can be used to update multiple counter variables making it possible to keep track of multiple time sequences at once.

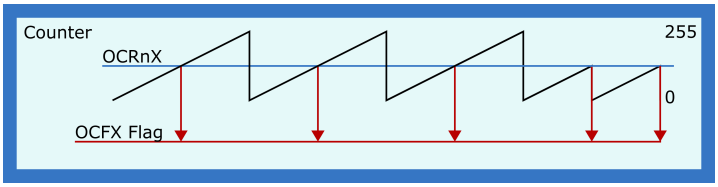

Figure 11: Timer/counter

The last two count sessions are reset after compare mach. This can be done manually by writing to `TCNTn`, or automatic in CTC mode (Clear Timer on Compare).

The AVR has both 8-, and 16-bit timer/counters. An 8-bit counter can only count to 255. With a $16MHz$ clock this corresponds to $16,3ms$ at the most. A 16-bit counter can count up to $2^{16} = 65\,536$ which corresponds to $4,2s$. The 16-bit timers give more accuracy to a PWM signal. An 8-bit PWM signal at 5 Volts give voltage intervals of $20mV$, at 10-bit the intervals are $4,8mV$.

To read the current counter value of a timer we read `TCNTn`. This is a read/write register. We can reset the counter (or give it any other value) by writing to `TCNTn`. On the 16-bit timers the TCNTn register is in fact two 8 bit registers, `TCNTnH` and `TCNTnL`.
The timer compares its value to the `OCRnX` register (Output Compare Register n X). Timers can have multiple OCR registers, each has an interrupt vector.

11.1 Prescaler

A prescaler is a device used to lower the frequency of a clock signal. The AVR has prescalers on the different timers. Wich factors the prescaler can divide by is listed in the μC's datasheet. Timer resolution is determind by the frequency of the input clock. The prescaler is configured in the `TCCRnB` register. A neat AVR prescaler calculator can be found here: http://eleccelerator.com/avr-timer-calculator/

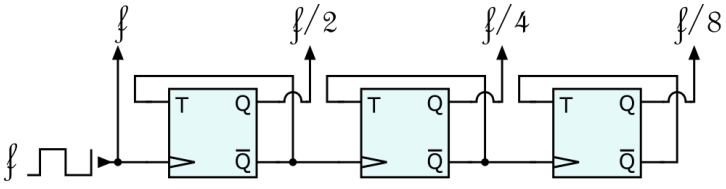

Figure 12: Clock divider principle

The AVR prescaler doesn't necessarily look like figure 12. It is added to show the basic principle of a clock divider.

11.2 1 second timer

Intention: Blink an LED with a frequency of $0,5Hz$.

11.2.1 Example: 16-bit timer

$$\texttt{F\_CPU} = 1\,000\,000Hz \qquad \frac{1MHz}{1024} = 976.5625Hz$$
$$2^{16} = 65\,536 \qquad 977 < 65\,535,\ OK.$$

Figure 13: 16-bit compare

We need an interrupt to occur every 977 ticks:

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

#define btn          PORTC6  // sw0 button
#define led          PORTC7  // indicator led
#define toggle_led   PORTC ^= (1<<led)

int main(void)
{
    DDRC   |=  (1<<led);     // Led pin as output
    PORTC  |=  (1<<led);     // Led off

    TCCR1B |=  (1<<CS02)| (1<<CS00);     // set prescaler to clk/1024
    TIMSK1 |= (1<<OCIE1A);               // interrupt enabled, OCR1A
    OCR1A   = 976;                       // 976 == 1s

    sei();              // set the global interrupt mask

    while(1)
    {
        /* application code */
    }
}

ISR(TIMER1_COMPA_vect)  // the vector is listed on Atmel's website
{
    toggle_led;
    TCNT1 = 0;          // reset counter
}
```

In this Example we manually rest the counter `TCNT1`. This can be done automaticly in hardware when using CTC mode.

11.2.2 Example: 8-bit timer

$$\texttt{F\_CPU} = 1\,000\,000Hz \qquad \frac{1MHz}{1024} = 976.5625Hz$$
$$2^8 = 255 \qquad \frac{977}{256} = 3.8 \approx 4$$

Figure 14: 8-bit compare

Since the 8-bit timer can't count to 977 we need a variable to keep track of the number of OCR's.

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

#define btn          PORTC6  // sw0 button
#define led          PORTC7  // indicator leds
#define toggle_led   PORTC ^= (1<<led)

uint8_t count;       // keeps track of output compare maches (OCR)

int main(void)
{
    count  = 0;
    DDRC  |=  (1<<led);                 // Led pin as output
    PORTC |=  (1<<led);                 // Led off

    TCCR0B |=  (1<<CS02)| (1<<CS00);    // set prescaler to clk/1024
    TIMSK0 |=  (1<<OCIE1A);             // enable interrupt, OCR0A
    OCR0A   = 255;          // count to 255 (max)
    sei();                  // set the global interrupt mask

    while(1)
    {
        /* application code */
    }
}

ISR(TIMER0_COMPA_vect)
{
    count++;         // update compare mach counter value
    TCNT0=00;        // reset counter

    if(count==4)
    {
        count=0;     // reset compare mach counter
        toggle_led;
    }
}
```

11.3 PWM

Pulse Width Modulation (PWM) is a technique used to convert digital signals into analog voltage. The output voltage (RMS) is determined by the duration of the ON period, compared to the OFF period (duty cycle). Longer on-time gives higher voltage. It is important to switch at a high rate. The switching frequency is determined by what the load can handle. It can be anywhere between 100mHz to multiple kilohertz (kHz). The PWM principle is used in class-D audio amplifiers, light-dimmers, motors etc.

The AVR has multiple PWM configurations, 8-10 bit resolution and multiple speeds. The speed is configured by the prescaler. Each Timer module has pins connected directly to the output compare unit of a timer. To change the duty cycle write to "Output Compare Register" (OCRnX).

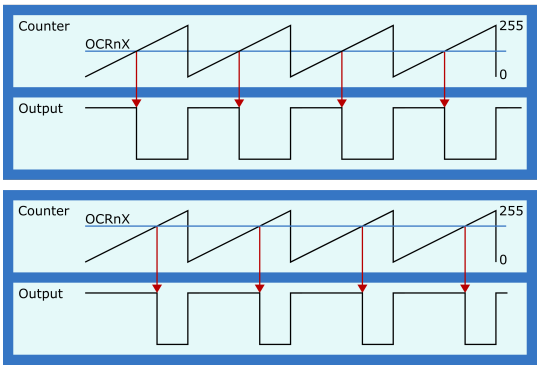

Figure 15: Fast PWM Generation

There are many different PWM possibilities, I advice you to read the specification for the different configurations in the datasheet. Here's a setup example for an 8-bit fast PWM:

```
1 TCCR1A  = (1<<COM1A1)   |   (1<<COM1A0) |   (1<<WGM10);
2 TCCR1B  = (1<<WGM12)    |   (1<<CS10);
3 // CS10 activates the clock
4 // WGM12 and WGM10 defines the PWM mode (fast pwm, 8-bit)
5 // COM1A1 and COM1A0 configures LED1 (PD5) to PWM (inverted mode)
```

To change the mean voltage we write to `OCR1A`. The output can have a value between 0 and 255.

11.3.1 PWM modes

There are three different PWM modes:

1. Fast PWM
2. Phase correct PWM
3. Phase and Frequency Correct PWM

"Phase and Frequency Correct PWM" is similar to "Phase correct PWM" but allow changes in frequency (without using the prescaler). By rising the frequency we sacrifice the PWM resolution.

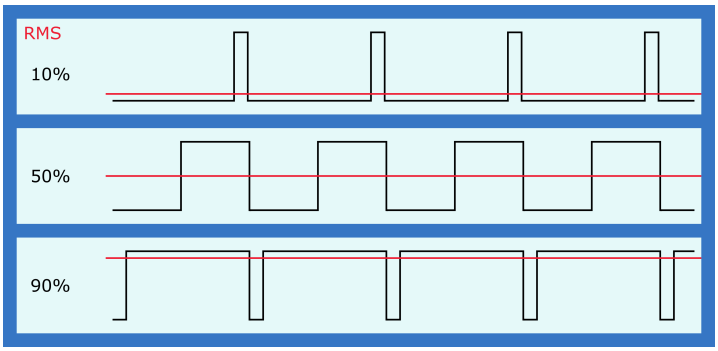

Figure 16: Fast PWM

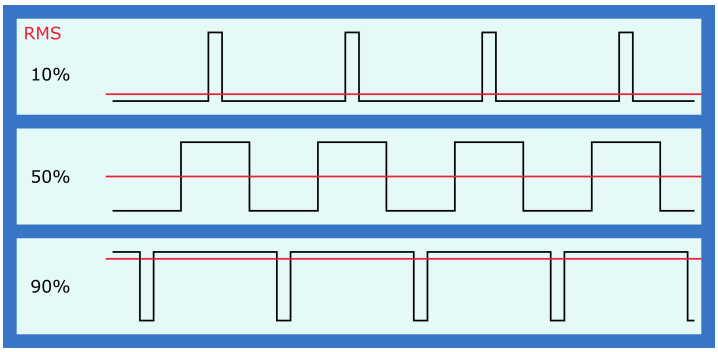

Figure 17: Phase correct PWM

12 USART module

When designing a system with more than one IC we need a way to communicate between chips. There are two fundamental ways: serial and parallel, see figure 18 "Serial VS Parallel communication".

Parallel communication use a bus to send a certan amount of bits at once, often 8, 16, or 32. Each bit has its own line on the bus. This is simple and fast, but the bus require a lot of circuit board area and many outputs from both transmitter and receiver.

Serial communication only need one line to carry the data from transmitter to receiver (two in reality; Data and reference). Serial protocols only send one bit at a time, This is time consuming but allow for simple parallel coupling of devices, see figure 19 "Parallel coupled devices".

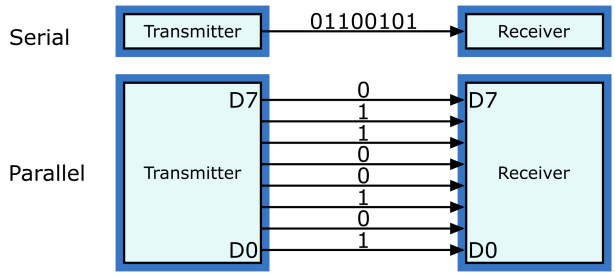

Figure 18: Serial VS Parallel communication

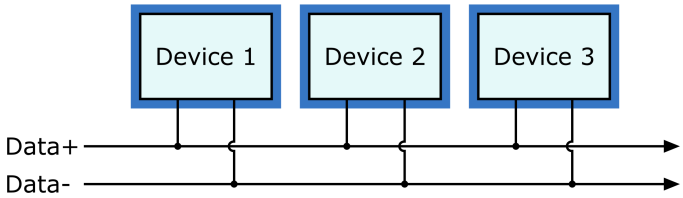

Figure 19: Parallel coupled devices

There are 3 fundamental configurations of serial communication. Simplex, half-duplex and full-duplex. With simplex the datatransfer only goes one direction (transmitter to receiver). Half-duplex offers bidirectional communication, and uses timebased multiplexing. This require accurate timing. Full duplex which need two data lines, offers a continuous bidirectional data transfer.

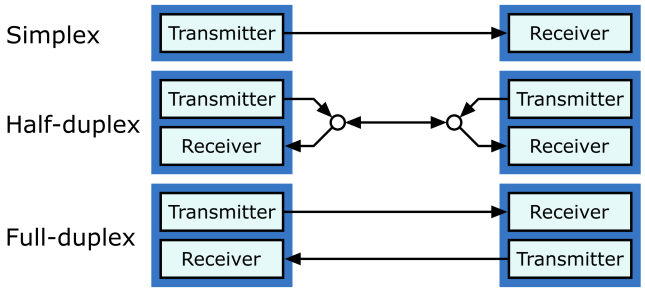

Figure 20: Serial terminologies

USART (Universal Synchronous Asynchronous Receiver Transmitter) is a form of serial communication. It uses two lines to send and receive data (RX and TX). USART is full-duplex (Independent serial receive and transmit registers). The AVR can therefore send and receive data at the same time. USART supports both synchronous and asyncronous communication. (Synchronous USART uses a clock signal as well).

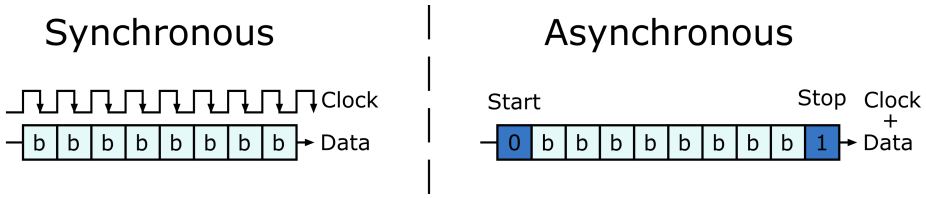

Figure 21: Synchronous VS Asynchronous serial

UART is a similar protocol (Universal Asynchronous Receiver Transmitter). UART only supports asynchronous communication. UART sends a 5-9-bit packet marked with start and stop bit(s) and optional parity bits . When no packet is sent the transmition line is held high (VCC). When a transmission is started the line is pulled low (start bit). After a packet is sent, the line is pulled high (stop bit), see figure 22 "UART packet".

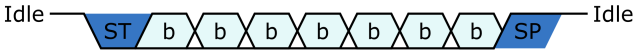

Figure 22: UART packet

The transmitter and receiver must be configured to the same *baud rate* (bit rate).
Baud is the symbol for the amount of symbols transmitted per second. Popular baud rates are 9600, 19200, 38400, 57600, 115200 and 1.8432 Mbaud. Other common standards are 31.25 kbps (MIDI) and 250 kbps (DMX).

The ATmega324PB Explained has a serial connection to the USB. The data can be viewed from Putty. In Putty the packets will be visible in ASCII format.

12.1 Configuration

12.1.1 Baud rate

```
1 #define F_CPU   1000000UL
2 // desired baud rate:
3 #define USART_BAUDRATE 9600
4 // UBRRn value:
5 #define BAUD_PRESCALE (((F_CPU / (USART_BAUDRATE * 16UL))) - 1)
```

12.1.2 UART configuration

```
1 // Set UBBR according to system clock and desired baud rate
2 UBRR1    = BAUD_PRESCALE;
3
4 // Turn on the transmission and reception circuitry
5 UCSR1B   = (1 << RXEN1) | (1 << TXEN1);
6 // packet size is 8-bit by default.
```

13 Tasks 1

1. Create a macro for the LED pin.
2. Create a program that turns the LED on, waits one second, then turns it off.
3. Create a macro for switching the LED on, and one for switching the LED off.
4. Modify the program so it turns the LED on while the button SW0 is pressed.

SW0	PC6
LED	PC7

Figure 23: User interface pinout

14 Tasks 2

1. Create a program which toggles the user LED when button SW0 is pressed.
2. Modify the previous program to register button presses through pin change interrupts.
3. Connect the board "OLED1 Xplained pro" to EXT1 and use PCINT on the 3 buttons to toggle the corresponding LEDs.

SW0	PC6 (PCINT22)
LED	PC7

Figure 24: User interface pinout

15 Tasks 3

1. Create a program that blinks the user LED at 2Hz using a 16 bit timer.
2. Set up PWM for LED1 on the "OLED1 Xplained pro" board.
3. Use a 16-bit timer to toggle LED1 on output compare match.

SW0	PC6 (PCINT22)
LED	PC7

Figure 25: User interface pinout

16 Cheat sheet

```
1 PORTB |=  (1<<PORTB2);  // Sets a bit
2 PORTB &= ~(1<<PORTB2);  // Clears a bit
3 PORTB ^=  (1<<PORTB2);  // Toggle bit
4 PINB  |=  (1<<PORTB2);  // Alternate toggling
5 (PINB & (1<<PINB2))     // Reading a bit, (PINB2 HIGH?)
6 _delay_ms(50);          // 50ms delay
```

SW0	PC6 (PCINT22)
LED	PC7

Figure 26: User interface pinout (ATmega324PB)

DDRX	Output/Input
PORTX	High/Low (High Z/Pull-up)
PINX	Read state (toggle PORTX)

Figure 27: GPIO registers

PCICR	Enable PCINT bus
PCMSKn	Enable PCINT for a pin on bus n
sei();	Enable interrupt
cli();	Disable interrupt
PCINTn_vect	PCIE interrupt vector

Figure 28: Pin change interrupt registers/macro

TCCRnX	Prescaler
TIMSKO	Enable interrupt on compare mach
OCRnX	Output compare register
TCNTn	Counter value
TIMERn_COMPA_vect	Timer/counter interrupt vector

Figure 29: Timer/Counter registers

UDRn	Send/Receive
UBRRn	Baud prescale
UCSRnB	Activate RX/TX circuitry

Figure 30: UART registers