Interface 1

EMBEDDED DIPLOMA

Agent

- GPIO
- Timers and PWM
- ADC
- EEPROM
- Serial communication
  - I2C
  - SPI
  - UART

Input and Output (GPIO or DIO)

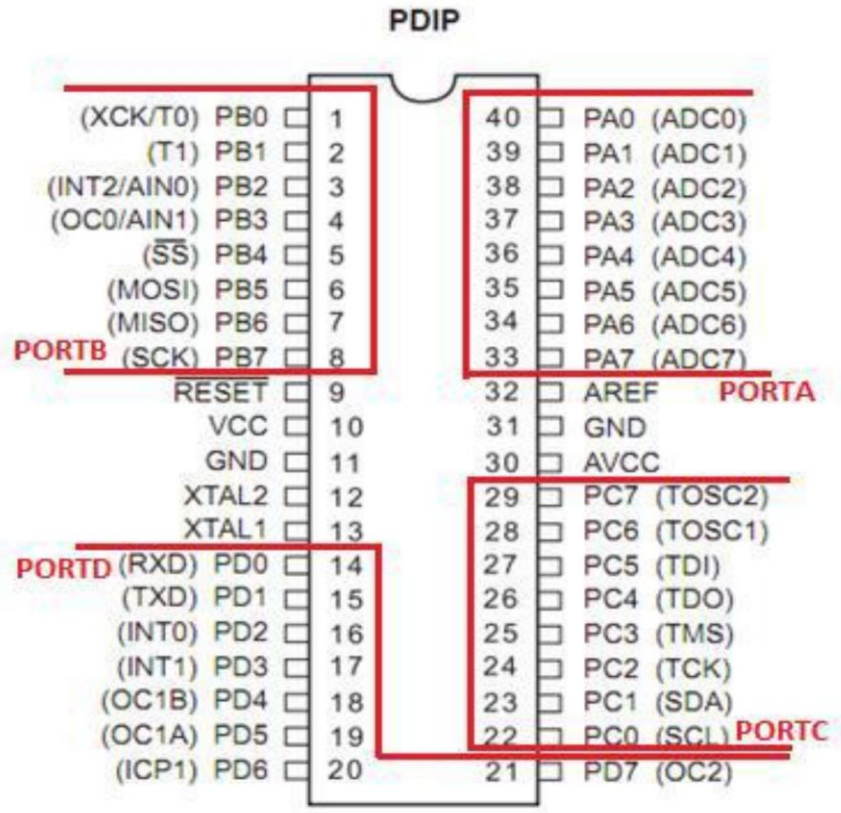
Atmega 32 has 32 programmable I/O lines divided into 4PORTS(groups):

1. PORTA(PA7……..PAO)
2. PORTB(PB7……..PBO)
3. PORTC(PC7……..PCO)
4. PORTD(PD7……..PDO)

Input and Output (GPIO or DIO)

- Each PORT is controlled by 3 registers:
  1. DDRx(Data Direction Register)
  2. PORTx(Output Register)
  3. PINx(Input Register)

- Note: Most pins in µC make more than one function(multiplexed functions)



GPIO datasheet

PINx

Port D Input Pins Address – PIND

Bit	7	6	5	4	3	2	1	0	
	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	PIND
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

PORTD

Port D Data Register – PORTD

Bit	7	6	5	4	3	2	1	0	
	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	PORTD
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

DDRD

Port D Data Direction Register – DDRD

Bit	7	6	5	4	3	2	1	0	
	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	DDRD
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Input and Output (GPIO or DIO)

PORTx register: examples:

- to output 0xFF data on PORTB
  - **DDRB = 0b11111111;** //set all pins of PORTB as outputs
  - **PORTB = 0xFF;** //write data on PORTB
- to output 0x55 in variable x on PORTA
  - **DDRA = 0xFF;** //make PORTA as output
  - **PORTA = x;** //output variable on PORTA
- to output HIGH/LOW on only pin2 of PORTC
  - **DDRC |= 1 << 2;** //set only pin2 of PORTC as output
  - **PORTC |= 1 << 2;** //make it high.
  - **PORTC &= ~(1 << 2);** //make it LOW.

Input and Output (GPIO or DIO)

PINx register

- PINx (Port IN) used to read data from port pins. In order to read the data from port pin, first you have to change port's data direction to input. This is done by setting bits in DDRx to zero. If port is made output, then reading PINx register will give you data that has been output on port pins.

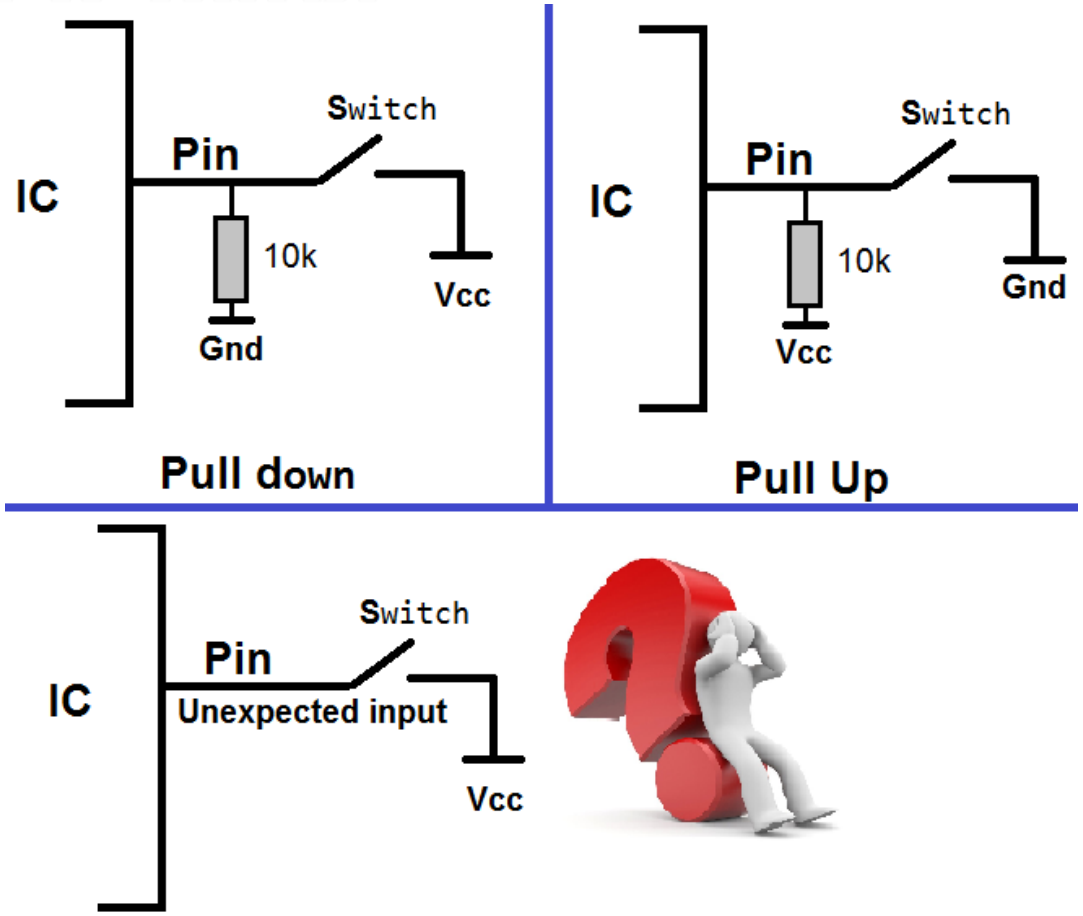
example :

- to read data from PORTA.
  - **DDRA = 0x00;** //Set PORTA as input
  - **x = PINA;** //Read contents of PORTA
- to read data from PIN5 PORTA.
  - **DDRA &= ~(1<<5);** //Set PORTA as input
  - **x = (PINA>>5) & 1;** //Read contents of PIN5 PORTA

PULL UP vs PULL DOWN

We use switches to give an order to the MCU to do something by changing the voltage level applied on an I/O pin (input).

If we connect switch direct to microcontroller we have a bad problem !



Input and Output (GPIO or DIO)

Notes:

- To use the interrupt features inside AtmelStudio, **#include <avr/interrupt.h>** header file should be included.
- To use the GPIO features inside AtmelStudio, **avr/io.h** header file should be included.
- To use the delay feature, **util/delay .h** header file should be included, then we can write:
  - **_delay_ms(1000);** //Delay 1 second
  - **_delay_us(100);** //Delay 100 micro second

Task1

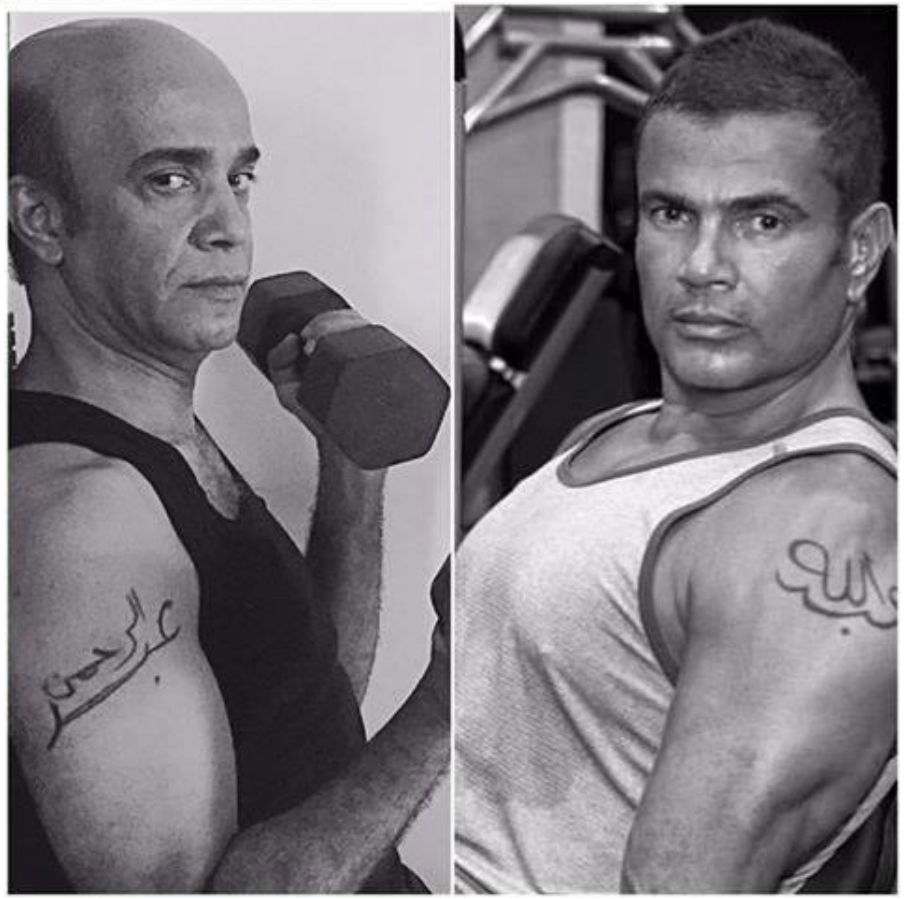
Write a code which turns led on PortD pin7 on.

Task2

Write a code which toggle led on PortD pin7 every 1 sec.

Requirement vs delivery

Please give me what I expected not yours ☺



Task3

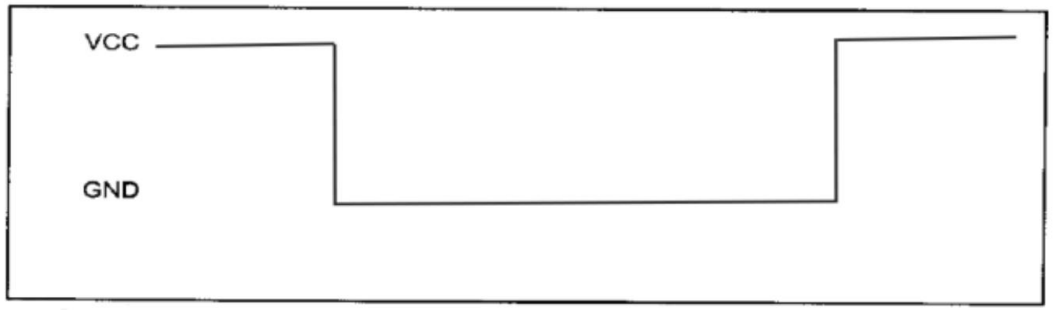
Write a code to make Push button on PortD pin0 control the led on PortD pin7.

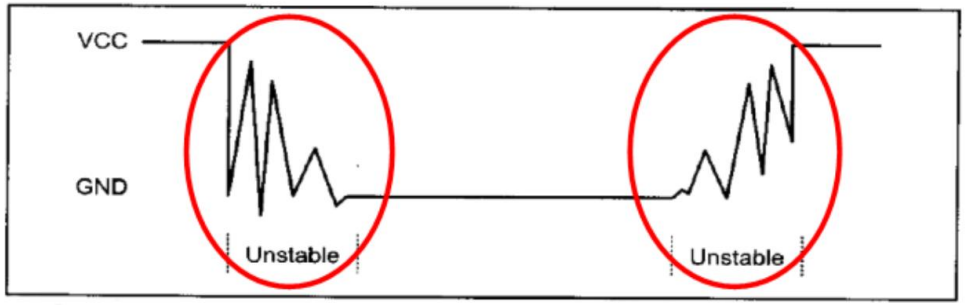
Switch Bounce Problem

Switch De-bounce:

- Hardware solution.
- Software Solution

It relies on the fact that bouncing takes a maximum period of 20-30 ms





Task 4

Use interrupts with push button to control the led to be on or off

Steps to work with GPIO interrupts

1. Disable global interrupts:

Inside SREG register (status register), there is a bit to enable/disable all interrupts in the AVR, at initialization it is better to disable all interrupts to avoid unexpected interrupts behavior, then enable it again after initialization steps: Page(10)

The AVR Status Register – SREG – is defined as:

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

Note that SREG register contains other useful information about the result of the most recently executed arithmetic instruction (carry flag, zero flag, etc…). This information can be used for altering program flow in order to perform conditional operations.

Note :we can use the built in macro: **cli()** to disable it.

Steps to work with GPIO interrupts

2. Make the corresponding pin to be input:
   - To work with INT0 (PD2): DDRD &= ~(1<<2);

3. Choose pin interrupt mode of operation:
   - For INT0 and INT1, use the MCUCR register: Page66

Bit	7	6	5	4	3	2	1	0	
	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Table 35. Interrupt 0 Sense Control

ISC01	ISC00	Description
0	0	The low level of INT0 generates an interrupt request.
0	1	Any logical change on INT0 generates an interrupt request.
1	0	The falling edge of INT0 generates an interrupt request.
1	1	The rising edge of INT0 generates an interrupt request.

Steps to work with GPIO interrupts

3. Choose pin interrupt mode of operation:

- **Falling edge** : this mean the input by default HIGH (pull up) and if we pressed the PB it will be LOW, which mean input transfer from HIGH to LOW level.
- **Rising edge** : this mean the input by default LOW (pull down) and if we pressed the PB it will be HIGH , which mean input transfer from LOW to HIGH level.
- **Any logical change:** this mean any variation in the input will cause an interrupt
- **Low level:** this mean that if the input is LOW: the ISR will be valid.

Steps to work with GPIO interrupts

4. Enable the required interrupt:

The required interrupt (ex. INT0) should be enabled, this is done through GICR register, writing 1 to the bit enable the interrupt: Page 67

Bit	7	6	5	4	3	2	1	0	
	INT1	**INT0**	**INT2**	**–**	**–**	**–**	**IVSEL**	**IVCE**	**GICR**
Read/Write	R/W	R/W	R/W	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Example: to enable INT0 only:

GICR |= 1<<6;

Steps to work with GPIO interrupts

5. Enable global interrupts:

To enable the global interrupt, the I-bit should be set to 1:

- SREG |= 1<<7; OR:

Or we can use the built in macro:

- sei();

Steps to work with GPIO interrupts

6. Write your ISR body for the required INTx:

For example, to write the INT0 ISR body:

```
ISR(INT0_vect)
{
        Flag = 1; /* Any short code */
}
```

Timer and PWM

- In Normal Mode, the timer triggers interrupt handlers. These can do practically any function you want, but the ISR runs on the CPU, which prevents anything else from running at the same time.
- In CTC mode, you can trigger interrupts if timer value matched the compare value.
- Frequency generator, It is also possible not to use interrupts and still toggle an output pin (OC0 for example). Using it this way, the functionality occurs parallel to the CPU and doesn't interrupt anything.
- PWM, runs in the background like CTC, but the timing of the output on the pin is different. It is more suited to devices like servos that take pulse width modulation as input as we can choose any duty cycle (Not only 50% like OC0 toggle in CTC mode).

Timers

What is a timer ?

- Timers generally have a resolution of 8 or 16 Bits.
- So a 8 bit timer is 8Bits wide so capable of holding value within 0-255. But this register has a special property, Its value increases/decreases automatically at a predefined rate (supplied by user) - This is the timer clock - and this operation does not need CPU's intervention.
- Since Timer works independently of CPU it can be used to measure time accurately .
- One of the basic condition is the situation when timer overflows i.e. it counts up to its maximum value (255 for 8-bit timers) and rolled back to 0. In this situation timer can issue an interrupt and you must write an Interrupt Service Routine (ISR) to handle the event.

Timers

- In ATMega32 we have three different kinds of timers:
  - TIMER0: 8-bit timer .
  - TIMER1: 16-bit timer .
  - TIMER2: 8-bit timer .
- The best part is that the timer is totally independent of the CPU. Thus, it runs parallel to the CPU and there is no CPU's intervention, which makes the timer quite accurate.
- Mode of operation :
  - normal mode (Timer is cleared after it overflows),
  - CTC mode (Timer is cleared after matching a specific value)
  - PWM mode.
  - Frequency generator

Timers

Timer frequency

$TTime\ period = 1/\ Frequency$

- Now suppose, we need to flash an LED every 5 ms. Now let's assume that we have an external crystal of 8 MHz. Hence, the CPU clock frequency is 8 MHz.
- For an 8-bit timer, it counts from 0 to 255 whereas for a 16-bit timer it counts from 0 to 65535. After that, they overflow.
- Let's say the timer's value is zero now. To go from 0 to 1, it takes one clock pulse. To go from 1 to 2, it takes another clock pulse. And so on.
- For CPU frequency F_CPU = 8 MHz, time period T = 1/8M = 0.000125 ms. Thus for every transition (0 to 1, 1 to 2, etc), it takes only 0.000125 ms!

Timers

- Now, as stated above, we need a delay of 10 ms. This maybe a very short delay, but for the microcontroller which has a resolution of 0.000125 ms, its quite a long delay!
- To get an idea of how long it takes, let's calculate the timer count from the following formula:

  $Timer\ count = (Required\ delay\ /\ Clock\ time\ period\ ) - 1$

  The -1 is for the fact that timer starts from zero not 1.
- Substitute Required Delay = 5 ms and Clock Time Period = 0.000125 ms, and you get Timer Count = 39999. The clock has already ticked 39999 times to give a delay of only 10 ms
- So we will use 16 bit timer not 8 bit timer (which is capable of counting up to 65535) to achieve this delay .

Timers

Using prescaler concept:

- Assuming F_CPU = 8 MHz and a 16-bit timer (MAX = 65535), and substituting in the above formula, we can get a maximum delay of 8.192 ms. Now what if we need a greater delay, say 10 ms? We are stuck!

- The technique of prescaler depend on make 1 count of timer take more than 1 clock cycle

  For example if the prescaler is 8 so to count fro 0 to 1 it will take 8 clock cycle not 1 as so every count will take 0.0001msec (0.000125 before) which mean resolution decreased. BUT the max delay for 16bit counter is 65.535ms which increase duration.

- So prescaler comes at a cost. There is a trade-off between resolution and duration.

- The problem is that the accuracy has decreased. Earlier, you were able to measure duration like 0.1125 ms accurately (0.1125/0.000125 = 900), but now we cannot (0.1125/0.001 = 112.5). The new timer can measure 0.112 ms and then 0.113 ms. No other value in between.

Timers

Working with TIMER0 steps:

We will take TIMER0 as an example, we will work in CTC mode (Clear Timer on Compare match) with interrupt enabled to toggle led every 1 sec

1. Disable global interrupts.

2. Initialize Timer/Counter0 value to be 0: This is the initial actual timer value, we want to start from 0, and this register is incremented each timer clock: TCNT0 = 0

Bit	7	6	5	4	3	2	1	0	
	TCNT0[7:0]								TCNT0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Timers

3. Set the required compare value: Set the compare value that we need to always compare to it after each timer count: OCR0 = 250; /* We want to stop counting at 250 */

Bit	7	6	5	4	3	2	1	0	
	OCR0[7:0]								OCR0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

4. Enable the required interrupt: The required interrupt for CTC mode should be enabled, this is done through TIMSK register, writing 1 to the bit enable the interrupt:

TIMSK = (1<<OCIE0); and if we worked in normal mode **TIMSK = (1<<TOIE0);**

Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Timers

5. Enable global interrupts.
6. Set Timer0 operation mode and set the prescaler value, this will start timer counting:

```
TCCR0 = (1<<FOC0) | (1<<WGM01) | (1<<CS00);
```

If we worked at normal mode we will just set CS00 and FOC0

Bit	7	6	5	4	3	2	1	0	
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0
Read/Write	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

From the data sheet we can find that we must:

- Set FOC0 to operate in none PWM mode (PWM will be discussed later).
- Set WGM01 to operate in CTC mode.
- Set CS00 to set the no- prescaler after this step the timer will start counting.

Timers

7. Write your ISR body for TIMER0:

```
ISR(TIMER0_COMP_vect) {
    /* Any short code */
    Number_of_complete_count++;
    If(((float)(Number_of_complete_count * OCR0 * 1000) / (float)F_CPU) > 1000){
            Number_of_complete_count = 0;
            TOGBIT(PORTD,7);
    }
}
```

If we deal with normal overflow mode:

```
ISR(TIMER0_OVF_vect){
        /* Any short code */
}
```

Note: If we set the CS00 to be zeros, this will disconnect the clock from the timer which means stopping the timer from counting.

PWM

What is PWM?

- Pulse-width modulation (PWM), or pulse-duration modulation (PDM), is a technique for getting analog results with digital means.
- Digital control is used to create a square wave, a signal switched between ON and OFF .
- The duration of “ON time" is called the pulse width. To get varying analog values, you change, or modulate, that pulse width.
- If you repeat this ON-OFF pattern fast enough with an LED for example, the result is controlling the brightness of the LED

PWM

What is PWM?

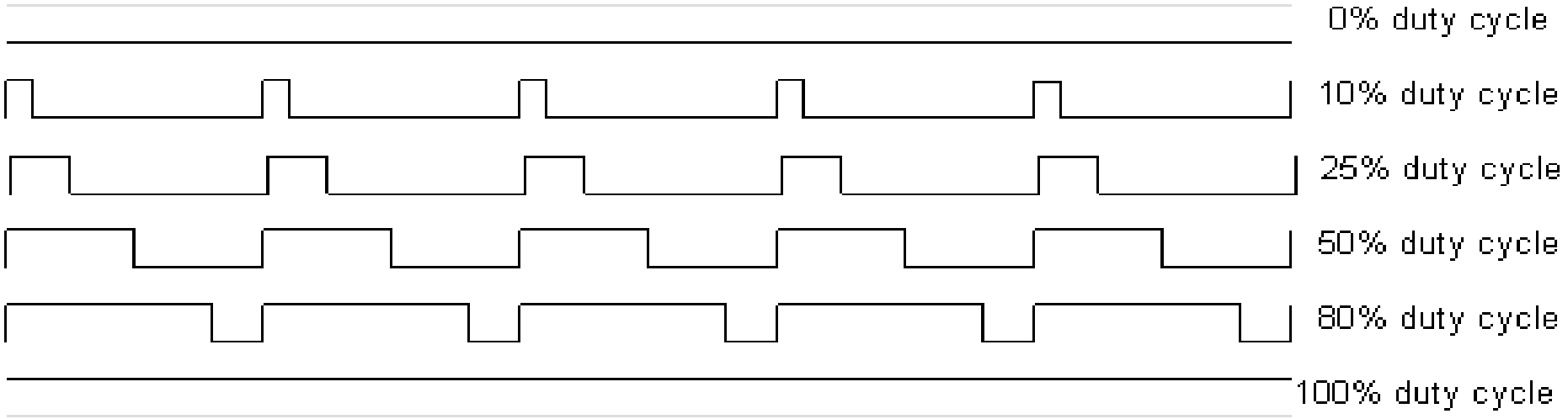
- The average value of voltage (and current) fed to the load is controlled by turning the switch between supply and load ON and OFF at a fast rate.
- The longer the switch is ON compared to the OFF periods, the higher the total power supplied to the load.
- For the 0-5V example, if ON duration is 75% of the period, then if we connected a LED it will feel like as if it is 3.75V (75% * 5V).
- The PWM switching frequency has to be much higher than what would affect the load (the device that uses the power), which is to say that the resultant waveform perceived by the load must be as smooth as possible.

PWM

PWM duty cycle

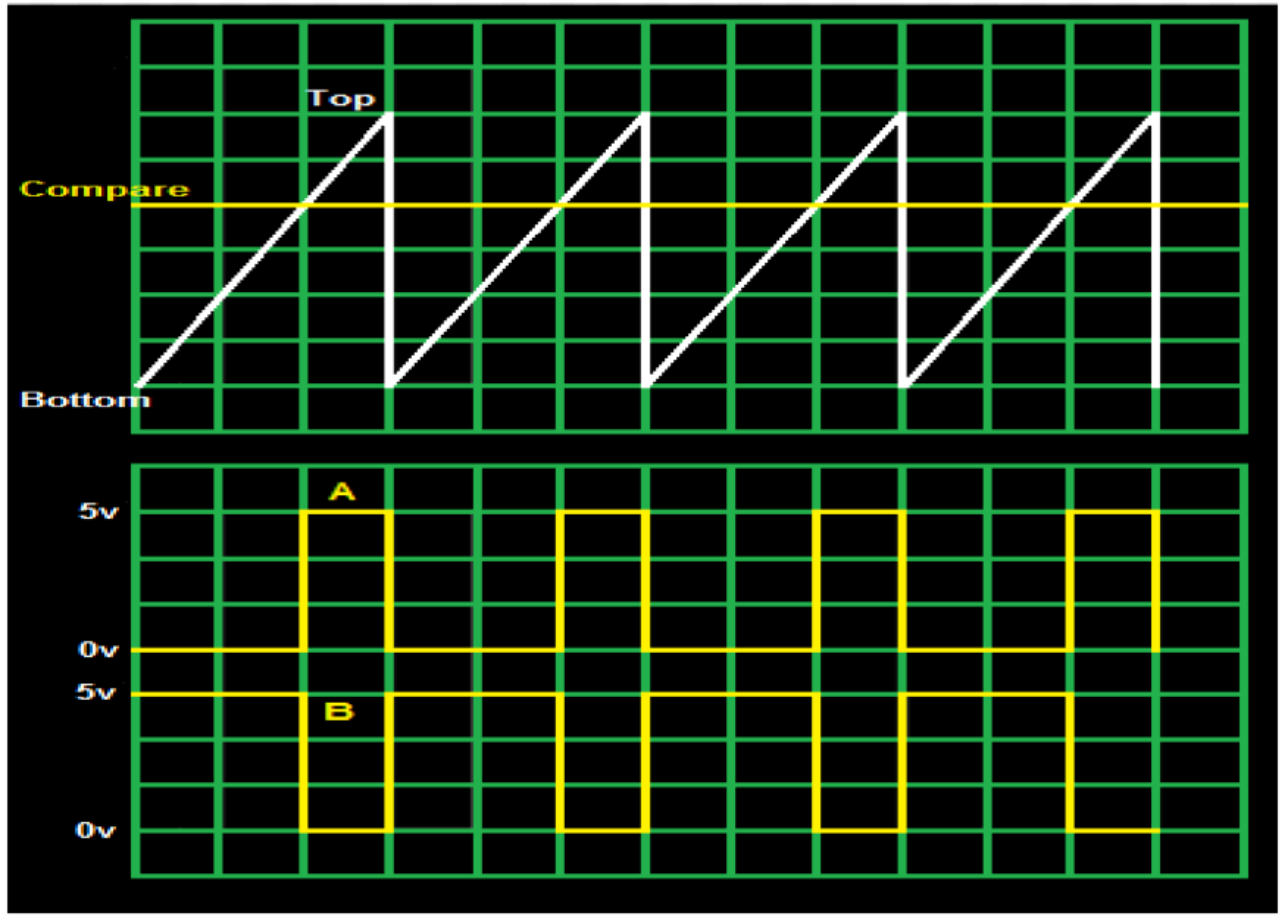
• Describes the proportion of 'on' time to the regular interval or 'period' of time, a low duty cycle corresponds to low power, because the power is off for most of the time. Duty cycle is expressed in percent, 100% being fully on.

$$DDuty\ cycle = (T\ on /(T\ on + \ Toff)\ ) \times 100\%$$

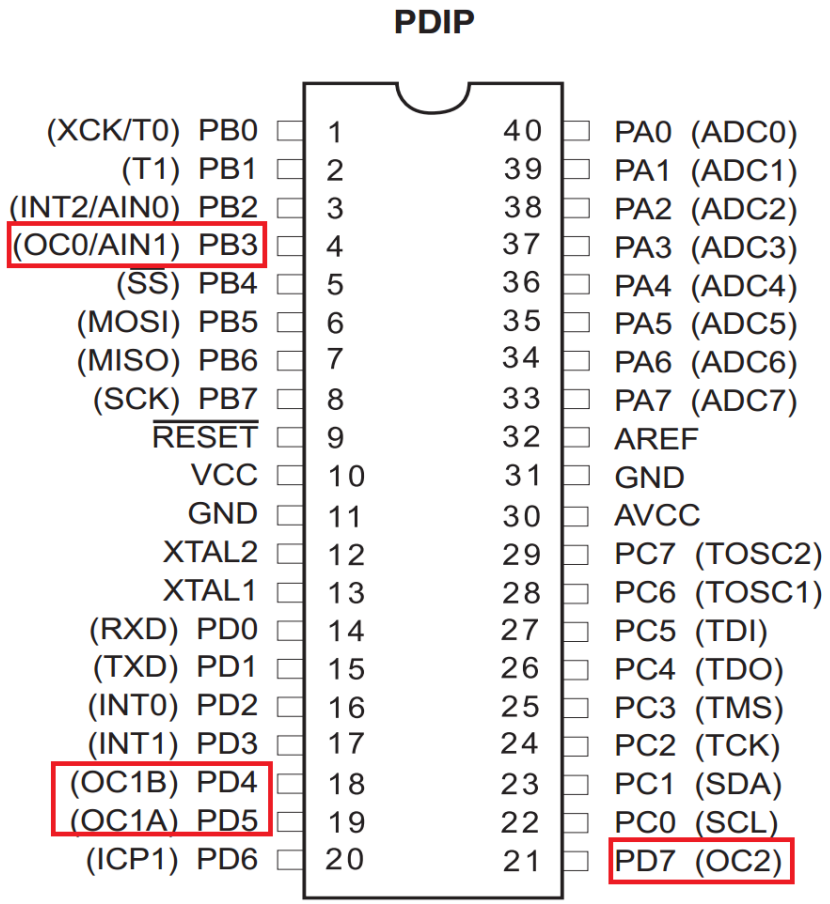


PWM

How PWM work



PWM PINS


PDIP
(XCK/T0) PB0 1
(T1) PB1 2
(INT2/AIN0) PB2 3
(OC0/AIN1) PB3 4
(SS) PB4 5
(MOSI) PB5 6
(MISO) PB6 7
(SCK) PB7 8
RESET 9
VCC 10
GND 11
XTAL2 12
XTAL1 13
(RXD) PD0 14
(TXD) PD1 15
(INT0) PD2 16
(INT1) PD3 17
(OC1B) PD4 18
(OC1A) PD5 19
(ICP1) PD6 20
40 PA0 (ADC0)
39 PA1 (ADC1)
38 PA2 (ADC2)
37 PA3 (ADC3)
36 PA4 (ADC4)
35 PA5 (ADC5)
34 PA6 (ADC6)
33 PA7 (ADC7)
32 AREF
31 GND
30 AVCC
29 PC7 (TOSC2)
28 PC6 (TOSC1)
27 PC5 (TDI)
26 PC4 (TDO)
25 PC3 (TMS)
24 PC2 (TCK)
23 PC1 (SDA)
22 PC0 (SCL)
21 PD7 (OC2)

PWM Modes

AVR timers offers more than one operation mode for PWM. Number of modes depend on timer. Timer1 is the most function rich timer, it offers 13 PWM modes, *timer0* and *timer1*offer only 3 modes.

Phase correct PWM

Use it to control motor speed controllers and servos. If you do not know which mode to use, choose this one. As name says its phase is always correct (see drawings to guess what *phase correct* means).

Fast PWM

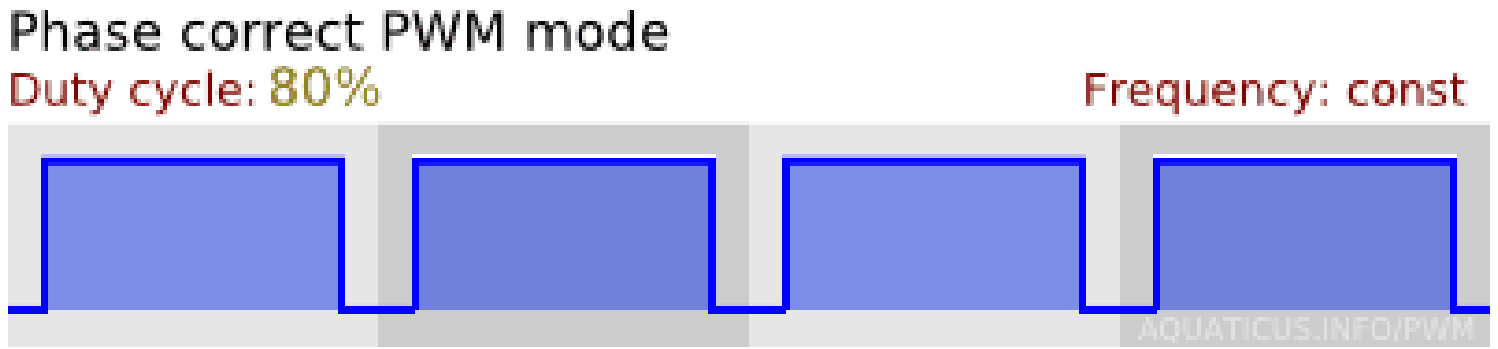
You can guess it is *fast*. To be more precise its frequency is twice that of the *Phase Correct* mode but it is not phase correct. Used for digital to analog converters and where phase does not matter but higher frequency is needed.

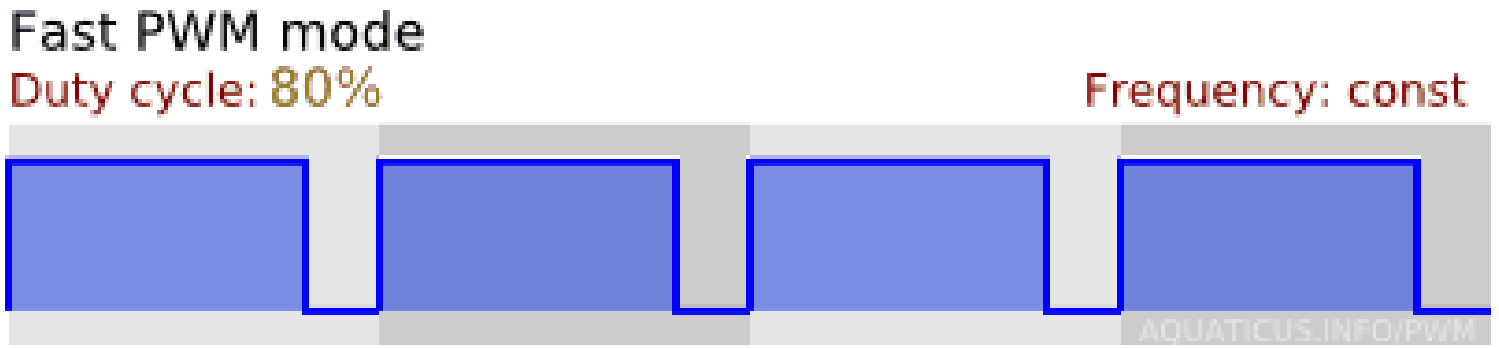
Phase and Frequency Correct

Unlike the above modes, frequency of the signal can be easily changed. In the *fast* and *phase correct* PWM modes frequency is determined by the prescaler value which limits number of possible frequencies to 5.

This mode is used to generate sound or to interface any device that is controlled by the variable frequency. Best suited to generate sound. Use when the phase of the signals should not change.

PWM Modes


Phase correct PWM mode
Duty cycle: 80%
Frequency: const
AQUATICUS.INFO/PWM


Fast PWM mode
Duty cycle: 80%
Frequency: const
AQUATICUS.INFO/PWM

PWM Modes

Summary

- Simplifying things we can say that in *Phase Correct* and *Fast* PWM modes ability to change duty cycle is useful. In *Phase and Frequency Correct* and *CTC* modes the possibility to change the frequency is the most useful thing.
- In addition 16 bit *timer1* allows to select 8, 9 or 10 bits accuracy. 8 bits timers (*timer0* and *timer2*) provide always 8 bit PWM.
- For full list of modes see *Waveform Generation Mode Bit Description* table in Atmel documentation (e.g. for ATmega32A timer1 it's on page 114).
- In Atmel documentation one of the timer modes is called *Normal*. It has nothing to do with PWM (it is not *normal PWM* mode). In this mode timer operates as an ordinary timer/counter without generating PWM signal. Although normal mode of timer can be used to generate software PW

PWM

Working with Fast PWM (Non-inverted mode) steps

1. Set PD5 as output: We will be working with OC1A output pin which is on PD5, so we need to set this pin as output first: DDRD |= (1<<PIND5);
2. Initialize Timer/Counter1 value to be 0: This is the initial actual timer value, we want to start from 0: TCNT1 = 0;
3. Set the required compare value: Set the compare value that we need to always compare to it after each timer count, note that after a compare match, ***the timer will continue counting up till reaching TOP value***, not like CTC mode that clears the timer after compare match, so we can output PWM signal. OCR1A = $(Duty\ cycle\ /\ 100) \times 1024$

Note That:

- We multiplied by 1024 "10BIT" as we will use TIMER1 which has a PWM max bits of 10 bits, and we want to know how many ticks we need to go before clearing the PWM signal to LOW .Duty cycle can have any value from 0 to 100.
- Doing this equation in code on integer operations may lead to zero value so do it on float or do the multiply first.

PWM

4. Set Timer1 operation mode in TCCR1A Register:

```
TCCR1A= (1<<WGM11)|(1<<WGM10)|(1<<COM1A1);
```

Bit	7	6	5	4	3	2	1	0	
	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	W	W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Set WGM10 and WGM11 to operate in PWM mode “10 bit”.
- Set COM1A1 to clear OC1A on compare match (Non-inverted PWM).

PWM

5. Set the prescaler value in TCCR1B, this will start timer counting:

Bit	7	6	5	4	3	2	1	0	
	ICNC1	**ICES1**	**–**	**WGM13**	**WGM12**	**CS12**	**CS11**	**CS10**	**TCCR1B**
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

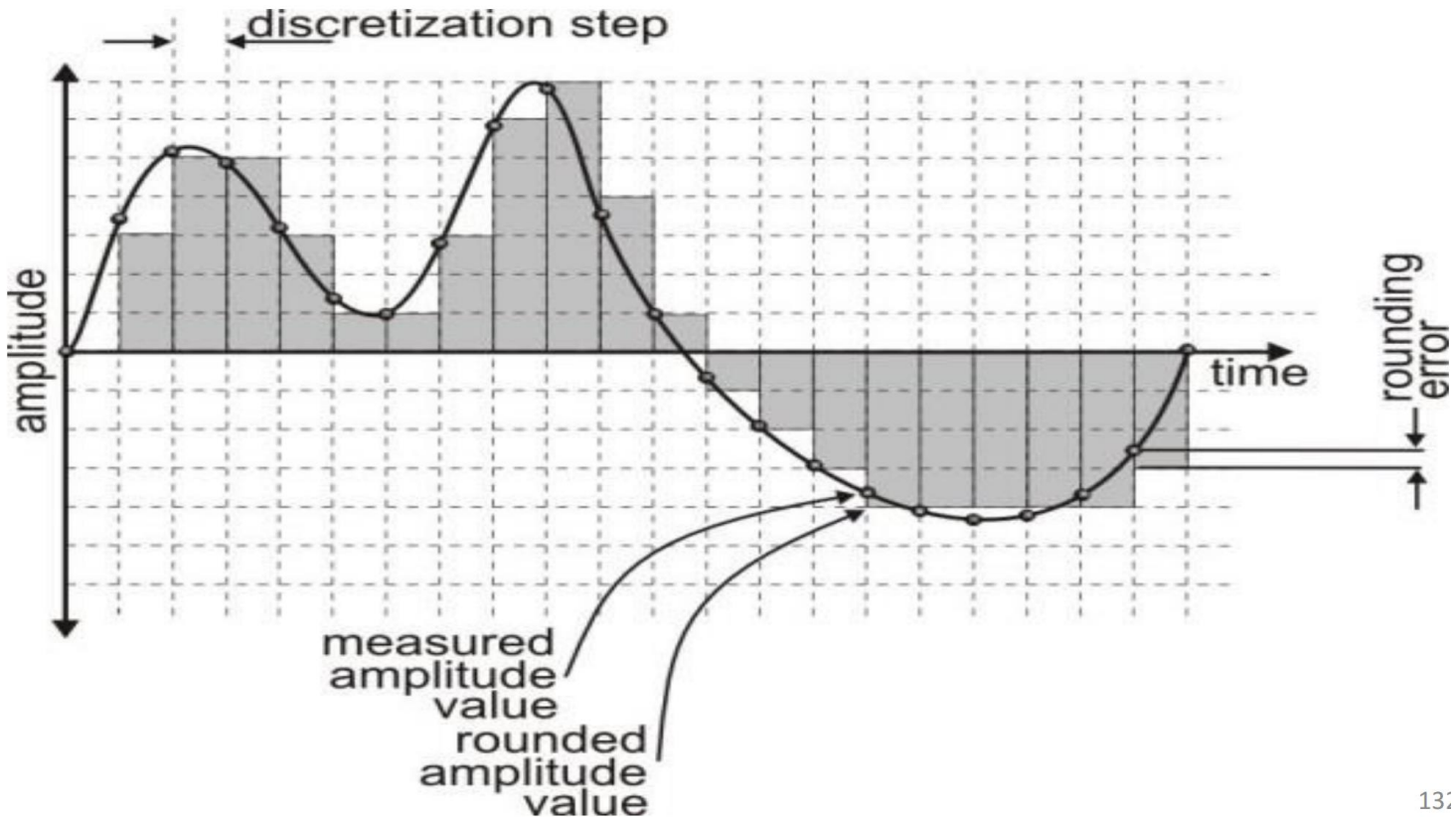
```
TCCR1B = (1<<CS10);
```

ADC

What is ADC?

- An analog-to-digital converter (ADC, A/D, or A to D) is a device that converts a continuous physical quantity (usually voltage) to a digital number that represents the quantity's amplitude.
- The conversion involves quantization of the input, so it necessarily introduces a small amount of error .
- Instead of doing a single conversion, an ADC often performs the conversions ("samples" the input) periodically . The result is a sequence of digital values that have been converted from a continuous-time and continuous-amplitude analog signal to a discrete-time and discrete amplitude digital signal.

ADC


discretization step
amplitude
time
rounding error
measured amplitude value
rounded amplitude value

ADC

ADC concepts

- **Sampling:** is the process of *taking captures of the signal* each specific amount of time.
- **Quantization**: is the process of **rounding** the taken value to a specific predefined constant values.
- **Sampling rate**: the rate at which new digital values are *sampled from the analog signal*.

ADC

ADC concepts

- **Resolution:** indicates the number of discrete values it can produce over the range of analog values (ex. 8-bits ADC resolution).
  - Resolution can also be defined electrically, and expressed in volts as the smallest change in analog signal (ΔV) that will result in a change in the digital output.
  - $\Delta V = (VH - VL)/ 2^n$ Where: $\boldsymbol{VH}$ and $\boldsymbol{VL}$ are maximum and minimum analog voltages and n is total number of bits in digital output.

$$\text{Resolution(volts)} = \frac{\text{Range(volts)}}{\text{Precision(alternatives)}} = \frac{5}{1024} = 4.88\text{mv}$$

- **Quantization error:** it is the noise introduced by quantization in the ADC. It is a rounding error between the analog input voltage to the ADC and the output digitized value. The noise is non-linear and signal-dependent.

ADC

ADC on ATMega32

- The ATmega32 features a 10-bit successive approximation ADC.
- The ADC is connected to an 8-channel Analog Multiplexer which allows 8 single ended voltage inputs constructed from the pins of Port A.
- The ADC contains a Sample and Hold circuit which is an analog device that samples (captures, grabs) the voltage of the continuously varying analog signal and holds (locks, freezes) its value at a constant level for a specified minimum period of time (until Analog to Digital conversion is done), which ensures that the input voltage to the ADC is held at a constant level during conversion

ADC

ADC on ATMega32

- The ADC has a separate analog supply voltage pin, AVCC.
- AVCC must not differ more than ±0.3 V from VCC.
- Internal reference voltages of nominally 2.56V or AVCC are provided On chip.
- The voltage reference may be externally decoupled at the AREF pin by a capacitor for better noise performance.
- A normal conversion takes 13 ADC clock cycles.

ADC

ADC on ATMega32

- There is 2 modes of operation:

1. Single ended mode (ADC signal is obtained from one ADC pin).
2. Differential mode (ADC signal is obtained from the voltage difference between 2 ADC pins).

- An n-bit single-ended ADC converts a voltage linearly between GND and VREF in 2^n steps (LSBs).
- The lowest code is read as 0, and the highest code is read as 2^n-1.
- VREF can be AREF, AVCC or internal 2.56V .

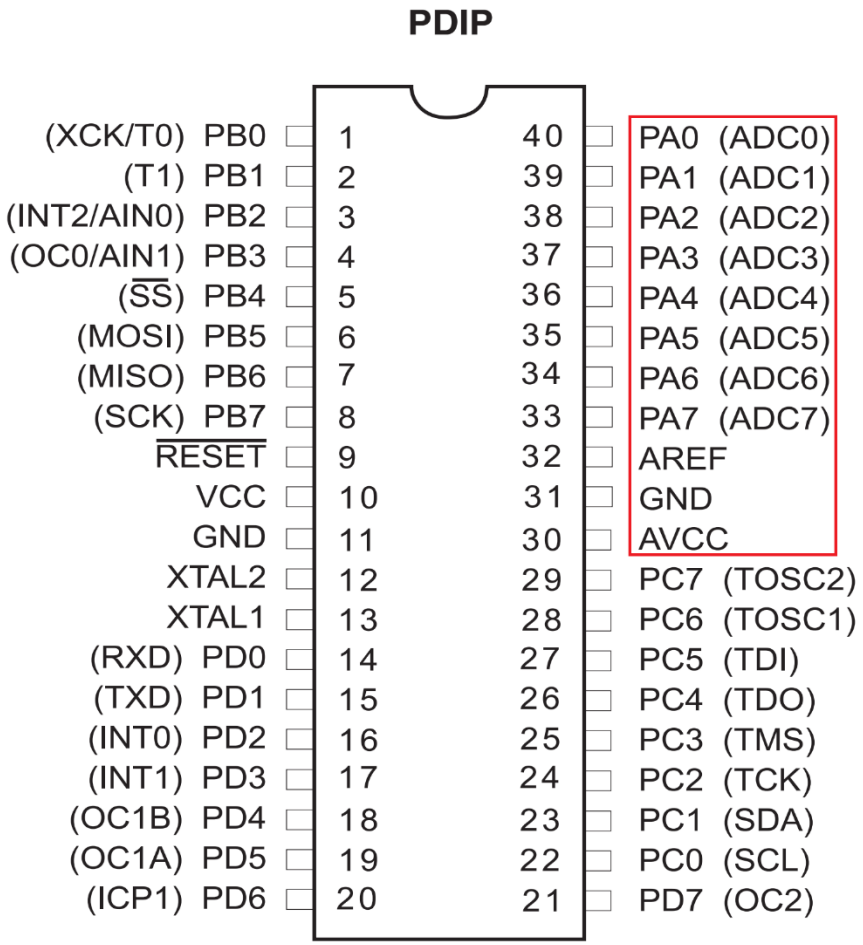
ADC

Differential mode:

$$ADC = \frac{(V_{POS} - V_{NEG}) \cdot GAIN \cdot 512}{V_{REF}}$$

The result is presented in two's complement form, from 0x200 (-512d) through 0x1FF (+511d) so the MSB will represent sign

ADC Pins


PDIP
(XCK/T0) PB0 1
(T1) PB1 2
(INT2/AIN0) PB2 3
(OC0/AIN1) PB3 4
(SS) PB4 5
(MOSI) PB5 6
(MISO) PB6 7
(SCK) PB7 8
RESET 9
VCC 10
GND 11
XTAL2 12
XTAL1 13
(RXD) PD0 14
(TXD) PD1 15
(INT0) PD2 16
(INT1) PD3 17
(OC1B) PD4 18
(OC1A) PD5 19
(ICP1) PD6 20
40 PA0 (ADC0)
39 PA1 (ADC1)
38 PA2 (ADC2)
37 PA3 (ADC3)
36 PA4 (ADC4)
35 PA5 (ADC5)
34 PA6 (ADC6)
33 PA7 (ADC7)
32 AREF
31 GND
30 AVCC
29 PC7 (TOSC2)
28 PC6 (TOSC1)
27 PC5 (TDI)
26 PC4 (TDO)
25 PC3 (TMS)
24 PC2 (TCK)
23 PC1 (SDA)
22 PC0 (SCL)
21 PD7 (OC2)

Working with ADC steps

1. Initialize the ADC voltage reference:
   - To choose between AREF, AVCC or internal 2.56V, REFS1:0 bits in ADMUX register are used
   - ADMUX = 0x40; we will use AVCC

Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Table 82. Voltage Reference Selections for ADC

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal Vref turned off
0	1	AVCC with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

Working with ADC steps

2. Choose the ADC prescaler:
   - To choose the ADC prescaler which decides the ADC sampling frequency, ADPS2:0 bits in ADCSRA register are used: ADCSRA |= 0b00000111

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Table 84. ADC Prescaler Selections

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

141

Working with ADC steps

3. Enable the ADC:
   - To enable the ADC, set ADEN bit in ADCSRA register:
     - ADCSRA |= 0x80;

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Working with ADC steps

4. Choose the required ADC channel :
   - To choose the ADC channel, the MUX4:0 bits in ADMUX register are used:

Bit	7	6	5	4	3	2	1	0	
	REFS1	**REFS0**	**ADLAR**	**MUX4**	**MUX3**	**MUX2**	**MUX1**	**MUX0**	**ADMUX**
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

   - For example to work on single ended channel ADC1: ADMUX |= (1<<MUX0)

Working with ADC steps

5. Read current ADC value :

To read the ADC channel value, we should start ADC conversion through writing 1 to ADSC bit in ADCSRA register, then wait for the conversion to finish.

When the conversion finishes, ADIF bit in ADCSRA will be set automatically to 1, the software should clear it by writing 1 to it:

```
ADCSRA |= (1<<ADSC);

while((ADCSRA & (1<<ADSC)));

Uint16_t data = ADCL + ADCH << 8;
```

Working with ADC steps

- Note that after reading the ADC we should take only the least 10 significant bits (10-bits ADC) as the result is right adjusted.
- There is another option which is to make the ADC result left adjusted in the ADC register, then the value will be like that:

Bit	15	14	13	12	11	10	9	8	
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
	ADC1	ADC0	–	–	–	–	–	–	ADCL

- This is achieved by setting the ADLAR bit in ADMUX register to 1.

Task

Use ADC0 to read analog data from variable resistance and use the analog input data to control the duty cycle of OCR1A which control the brightness of the LED on PORTD5

EEPROM

What is the EEPROM memory and why would I use it?

- EEPROM, short for Electronically Erasable Read-Only Memory, is a form of non-volatile memory with a reasonably long lifespan. Because it is non-volatile, it will retain its information during periods of no power and thus is a great place for storing data

How is is accessed?

- The AVR's internal EEPROM is accessed via special registers inside the AVR, which control the address to be written to (EEPROM uses byte addressing), the data to be written (or the data which has been read) as well as the flags to instruct the EEPROM controller to perform the requested read or write operation.

To deal with EEPROM please follow this link EEPROM_Access

Communication protocols

- What is communication in embedded systems ?

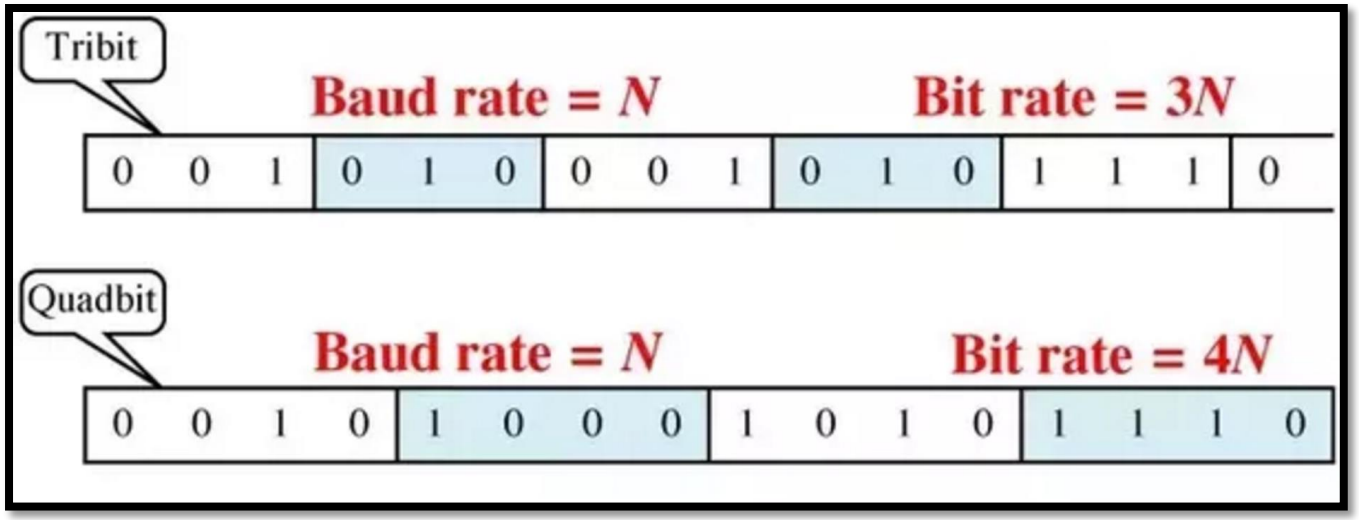
  It is a way of exchanging data or commands between 2 or more different embedded systems devices.

  Communication protocols are standards that contains data exchange rules and format between embedded systems

- Communication protocols examples in embedded systems
  - UART .
  - I2C.
  - SPI.
  - CAN.
  - LIN

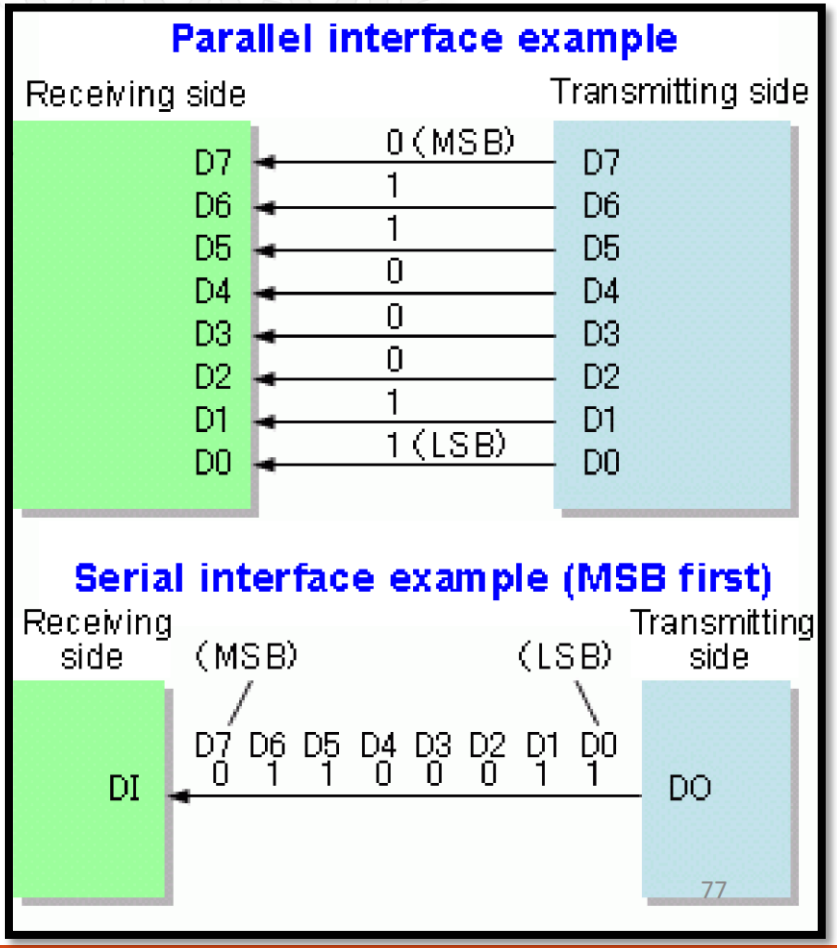
Communication protocols

- **Bit rate**
  - It is the number of bits that are sent per unit of time usually bits/sec.
- **Baud rate**
  - It is the number of symbols that are sent per unit of time usually, symbol can be combination of any number of bits according to design choice.
  - In case that symbol is only 1 bit, then baud rate will equal the bit rate.

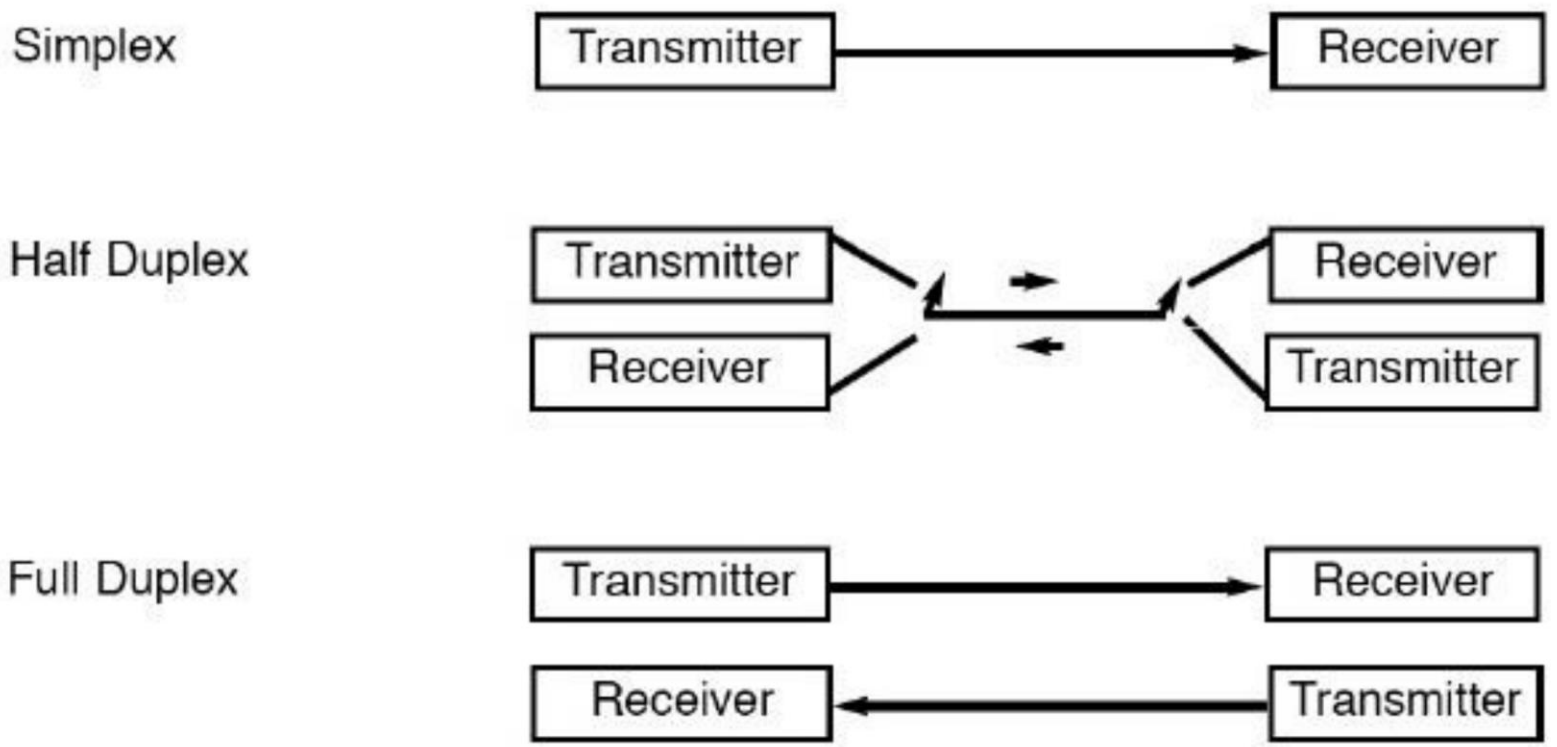


Communication protocols

- Parallel Communication
  - Sending multiple bits simultaneously.
- Serial communication
  - Sending bit by bit.



Communication protocols



Communication protocols

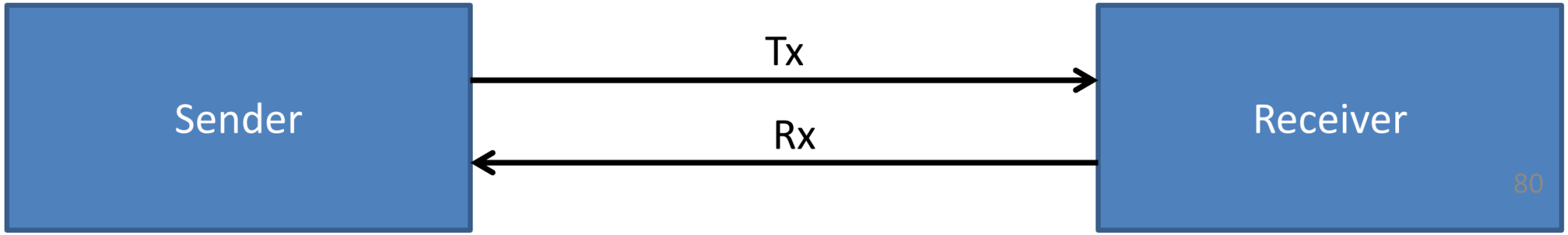
Synchronous serial communication

- Synchronous serial communication describes a serial communication protocol in which data is sent in a continuous stream at a constant rate.
- Synchronous communication requires that the clocks in the transmitting and receiving devices are synchronized – *running at the same rate* – so the receiver can sample the signal at the same time intervals used by the transmitter so that there is an extra wire for clock carrying.

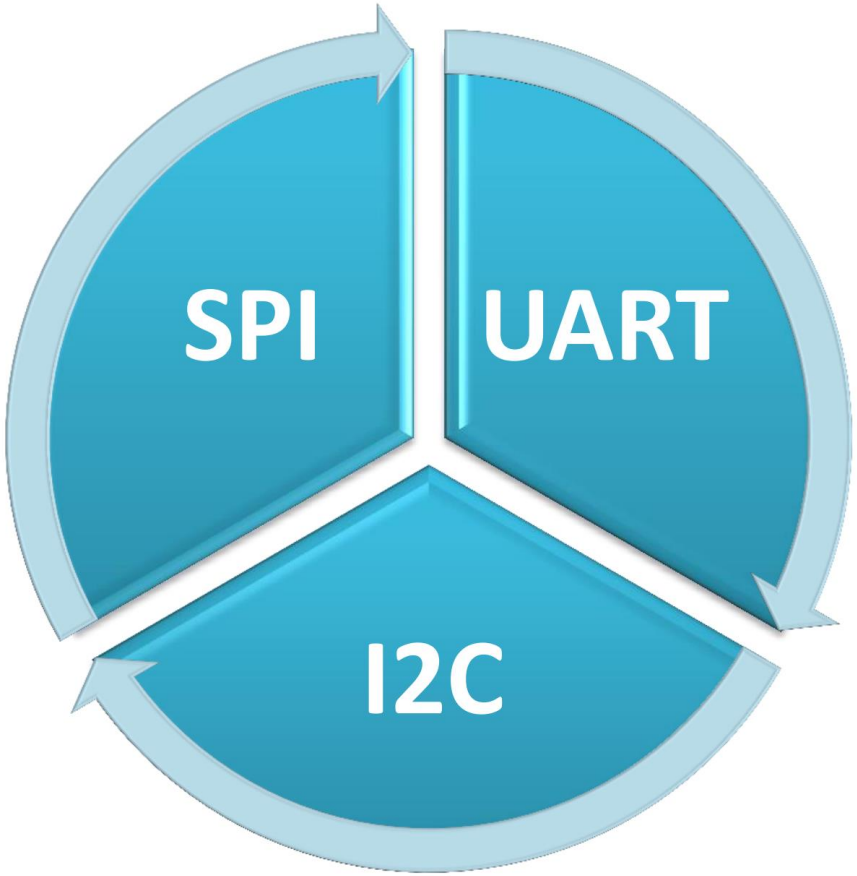
Communication protocols

Asynchronous serial communication

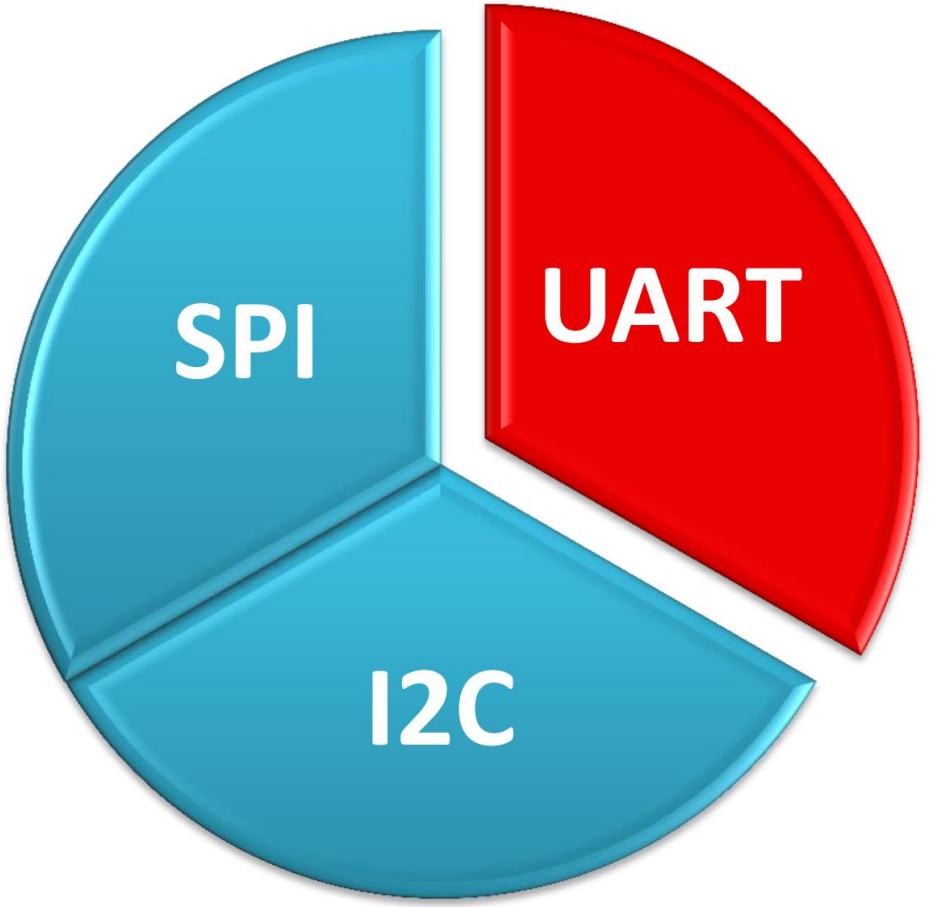
- Asynchronous serial communication is a form serial communication in which the communicating endpoints' interfaces are not continuously synchronized by a common clock signal.
- Instead of a common synchronization signal, the data stream contains synchronization information in form of start and stop signals, before and after each unit of transmission, respectively . The start signal prepares the receiver for arrival of data and the stop signal resets its state to enable triggering of a new sequence.



Communication protocols



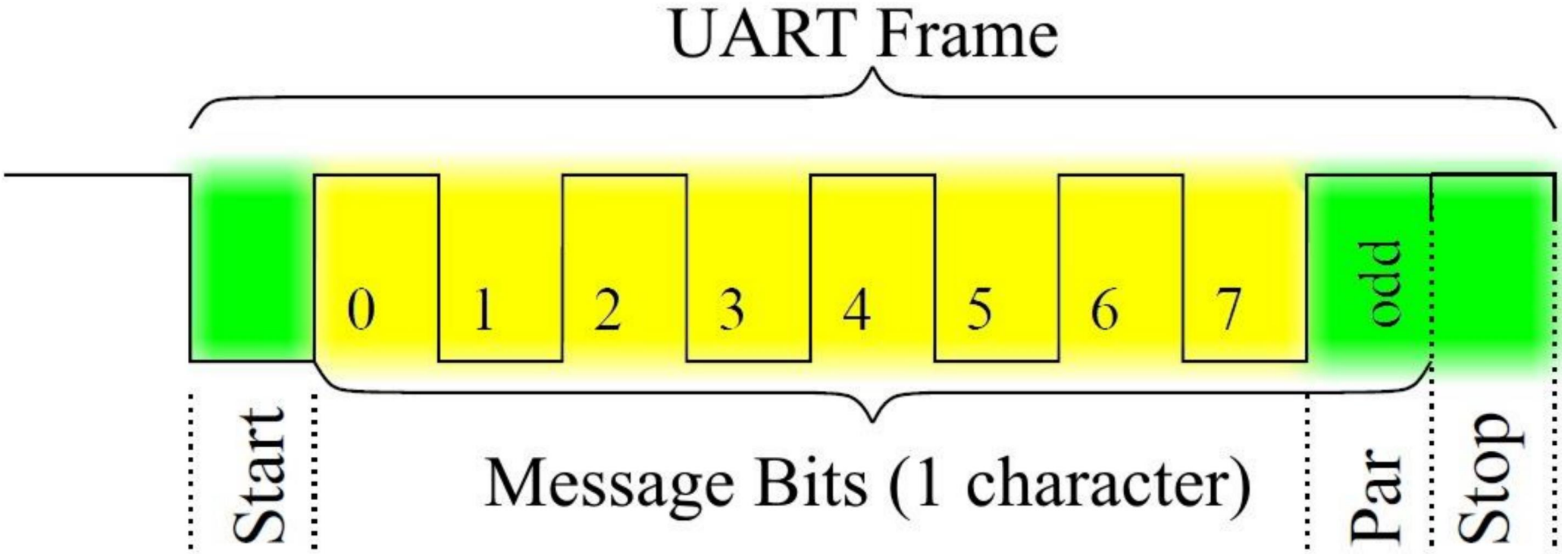
Communication protocols



UART

- UART stands for Universal Asynchronous Receiver Transmitter .
- There is one wire for transmitting data, and one wire to receive data.
- UART is a full duplex communication protocol.
- UART is point to point communication protocol, which means communication can be between 2 devices only at the same time.
- A common parameter is the baud rate known as "bps" which stands for bits per second. If a transmitter is configured with 9600bps, then the receiver must be listening on the other end at the same speed.
- UART is a serial communication, so bits must travel on a single wire. If you wish to send a char (8-bits) over UART, the char is enclosed within a start and a stop bit, so to send 8-bits of char data, it would require 2-bit overhead, this 10-bit of information is called a UART frame.

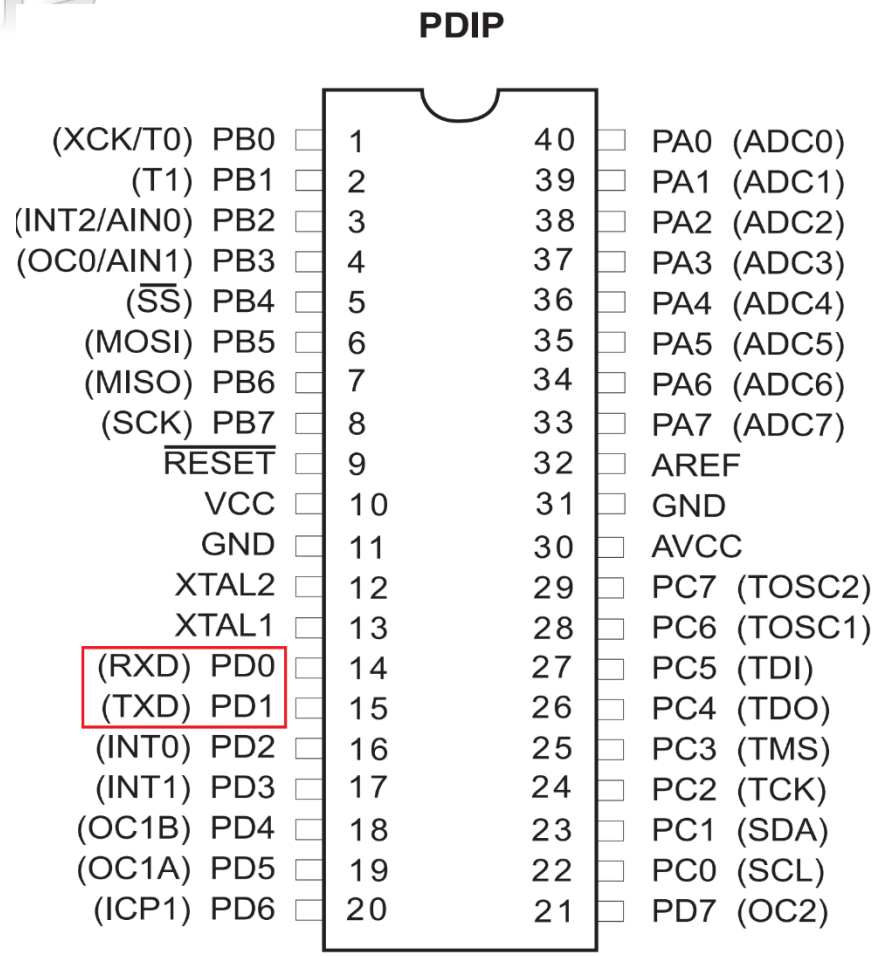
UART frame format



UART frame format

- Start bit: 1 bit indicates the start of a new frame, always logic low .
  - Data: 5 to 9 bits of sent data.
  - Parity bit: 1 bit for error checking
- Even parity: clear parity bit if number of 1s sent is even.
- Odd parity: clear parity bit if number of 1s sent is odd.
  - Stop bit: 1 or 2 bits indicate end of frame, always logic high

UART pins


PDIP
(XCK/T0) PB0 1
(T1) PB1 2
(INT2/AIN0) PB2 3
(OC0/AIN1) PB3 4
(SS) PB4 5
(MOSI) PB5 6
(MISO) PB6 7
(SCK) PB7 8
RESET 9
VCC 10
GND 11
XTAL2 12
XTAL1 13
(RXD) PD0 14
(TXD) PD1 15
(INT0) PD2 16
(INT1) PD3 17
(OC1B) PD4 18
(OC1A) PD5 19
(ICP1) PD6 20
40 PA0 (ADC0)
39 PA1 (ADC1)
38 PA2 (ADC2)
37 PA3 (ADC3)
36 PA4 (ADC4)
35 PA5 (ADC5)
34 PA6 (ADC6)
33 PA7 (ADC7)
32 AREF
31 GND
30 AVCC
29 PC7 (TOSC2)
28 PC6 (TOSC1)
27 PC5 (TDI)
26 PC4 (TDO)
25 PC3 (TMS)
24 PC2 (TCK)
23 PC1 (SDA)
22 PC0 (SCL)
21 PD7 (OC2)

Working with UART steps

1. Set transfer rate mode:
   - AVR supports 2 different transfer modes: normal and double transfer modes, they differ only in the calculations of the baud rate as follows:

Table 60. Equations for Calculating Baud Rate Register Setting

Operating Mode	Equation for Calculating Baud Rate[1]	Equation for Calculating UBRR Value
Asynchronous Normal Mode (U2X = 0)	$BAUD = \frac{f_{OSC}}{16(UBRR+1)}$	$UBRR = \frac{f_{OSC}}{16BAUD} - 1$
Asynchronous Double Speed Mode (U2X = 1)	$BAUD = \frac{f_{OSC}}{8(UBRR+1)}$	$UBRR = \frac{f_{OSC}}{8BAUD} - 1$
Synchronous Master Mode	$BAUD = \frac{f_{OSC}}{2(UBRR+1)}$	$UBRR = \frac{f_{OSC}}{2BAUD} - 1$

Note: 1. The baud rate is defined to be the transfer rate in bit per second (bps).

BAUD Baud rate (in bits per second, bps)

f_{OSC} System Oscillator clock frequency

UBRR Contents of the UBRRH and UBRRL Registers, (0 - 4095)

   - Setting the transfer mode is through the U2X bit inside the UCSRA register

Working with UART steps

2. Enable Tx and Rx:

Enable the transmitter and receiver through UCSRB register:

UCSRB |= (1<<TXEN) | (1<<RXEN);

Bit	7	6	5	4	3	2	1	0	
	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8	UCSRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Working with UART steps

3. Setup the UART frame: From the data sheet we can find that we must:
   - Set URSEL when writing to UCSRC register .
   - Set UCSZ0 and UCSZ1 for 8-bits data frame.
   - Clear UMSEL to operate in Asynchronous mode.
   - Clear UPM0 and UPM1 to for zero parity bits frame.
   - Clear USBS for 1 stop bit frame.

The step in one line of code: **UCSRC |= (1<<URSEL) | (1<<UCSZ1) | (1<<UCSZ0);**

Working with UART steps

4. Start sending data:

To start sending data, at each byte we want to send:

- we should wait first until making sure that previous byte was sent (polling mode),
- then start sending another byte.
- This is done through polling on UDRE bit inside UCSRA register, when transmitter is ready to send new data the **UDRE** bit is set to 1.
- After making sure that UDRE bit is high, we can start sending through
- writing the required byte to send inside the UDR register:

```
while (!((UCSRA) & (1<<UDRE)));
UDR = 'A';  /* Send A through UART */
```

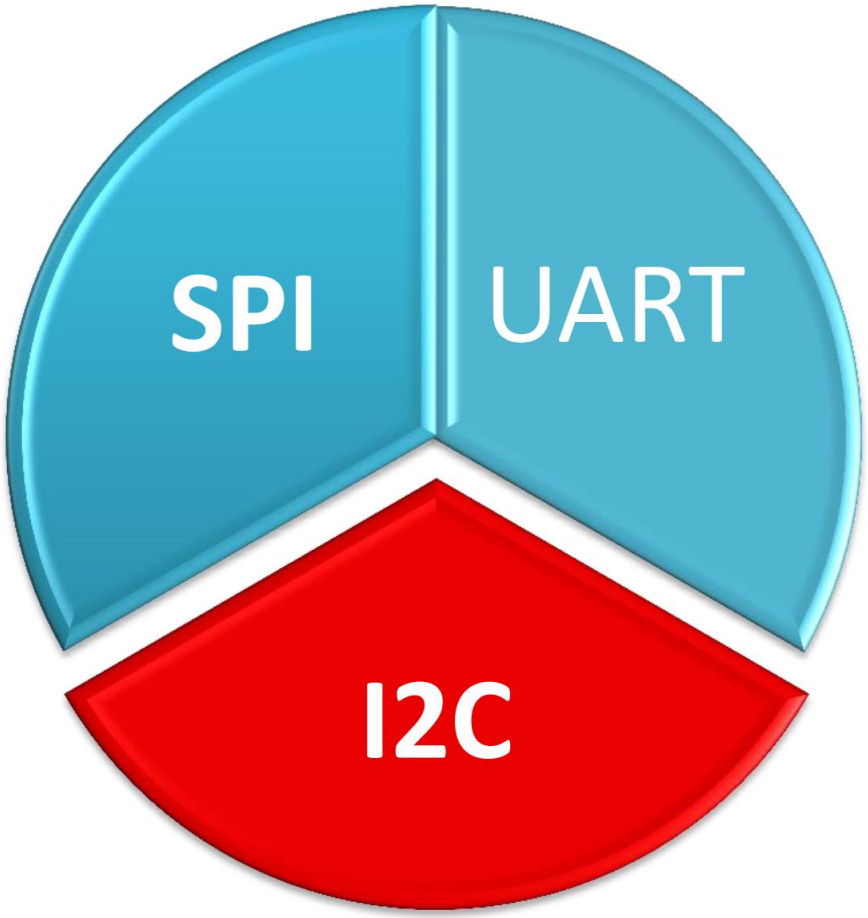
Working with UART steps

5. Start receiving data:

To start receiving data, At each byte we want to receive

- we should wait first until there is new unread bytes in the Rx buffer (polling mode),
- then start reading the new byte.
- This is done through polling on RXC bit inside UCSRA register, when receiver has new data the RXC bit is set to 1 and cleared after reading the new data.
- After making sure that RXC bit is high, we can read the new byte inside the UDR register:

```
while (!((UCSRA) & (1<<RXC)));
uint8 data = UDR;  /* receive byte through UART */
```


SPI
UART
I2C

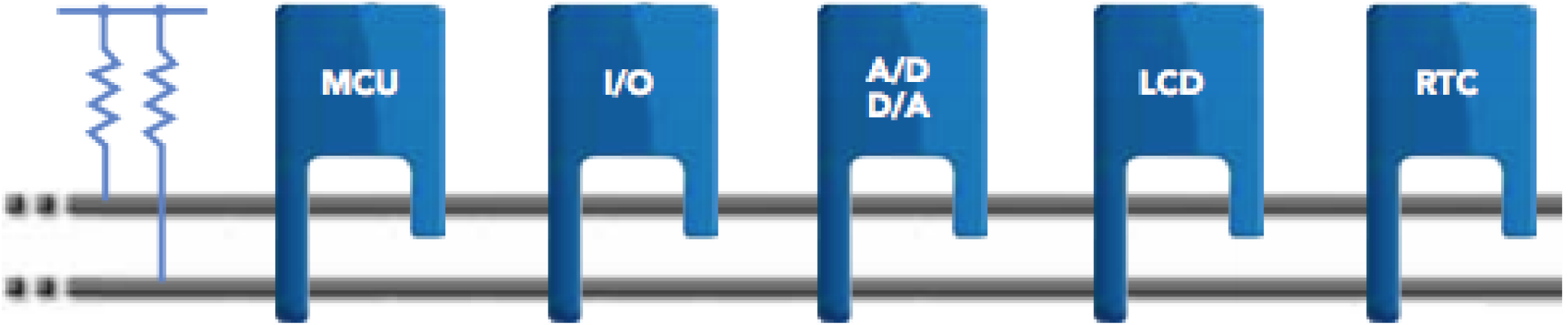
I2C

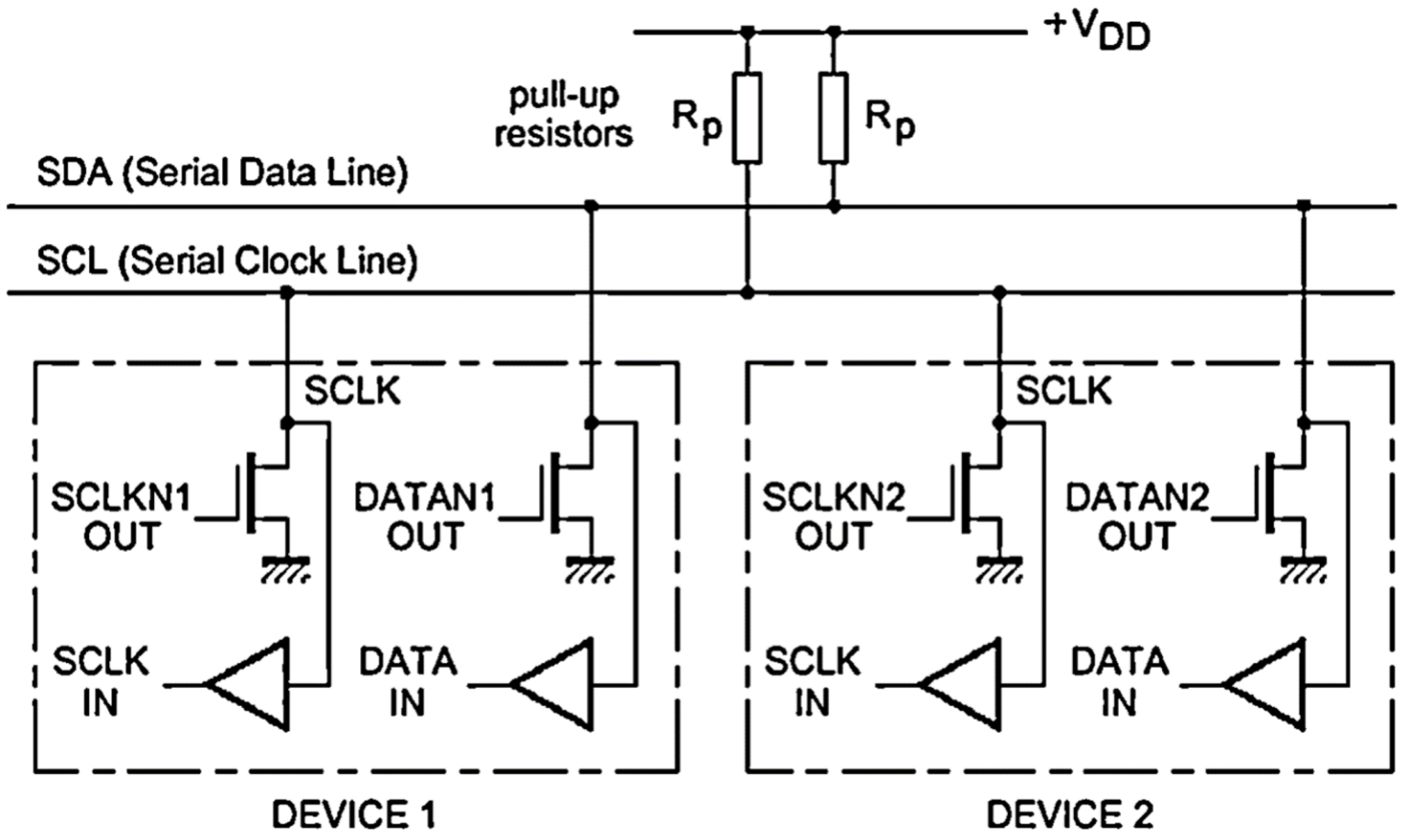
What is I2C?

• I^2C (Inter-Integrated Circuit), is a multi-master, multi-slave, serial computer bus invented by Philips Semiconductor (now NXP Semiconductors). It is typically used for attaching lower-speed peripheral ICs to processors and microcontrollers.

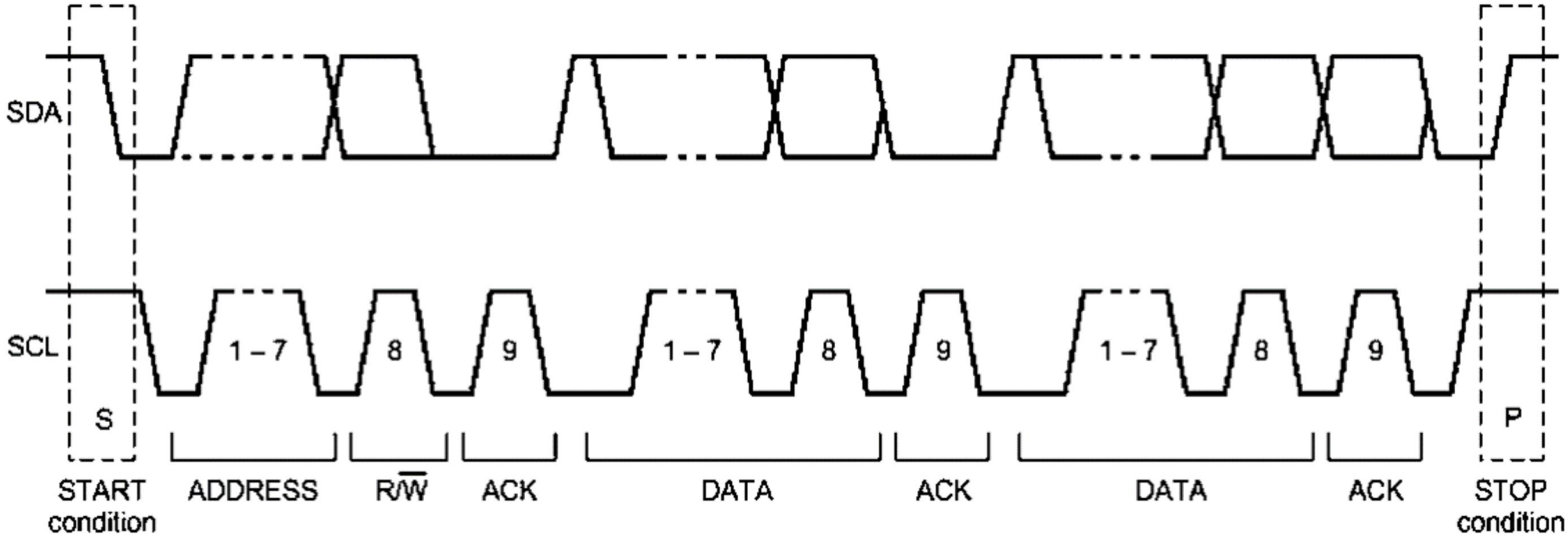
• I2C is a synchronous serial communication using two wires, one wire for data (SDA) and the other wire for clock (SCL).

I2C bus




+VDD
pull-up resistors
Rp
Rp
SDA (Serial Data Line)
SCL (Serial Clock Line)
SCLK
SCLKN1 OUT
DATAN1 OUT
SCLK IN
DATA IN
DEVICE 1
SCLK
SCLKN2 OUT
DATAN2 OUT
SCLK IN
DATA IN
DEVICE 2

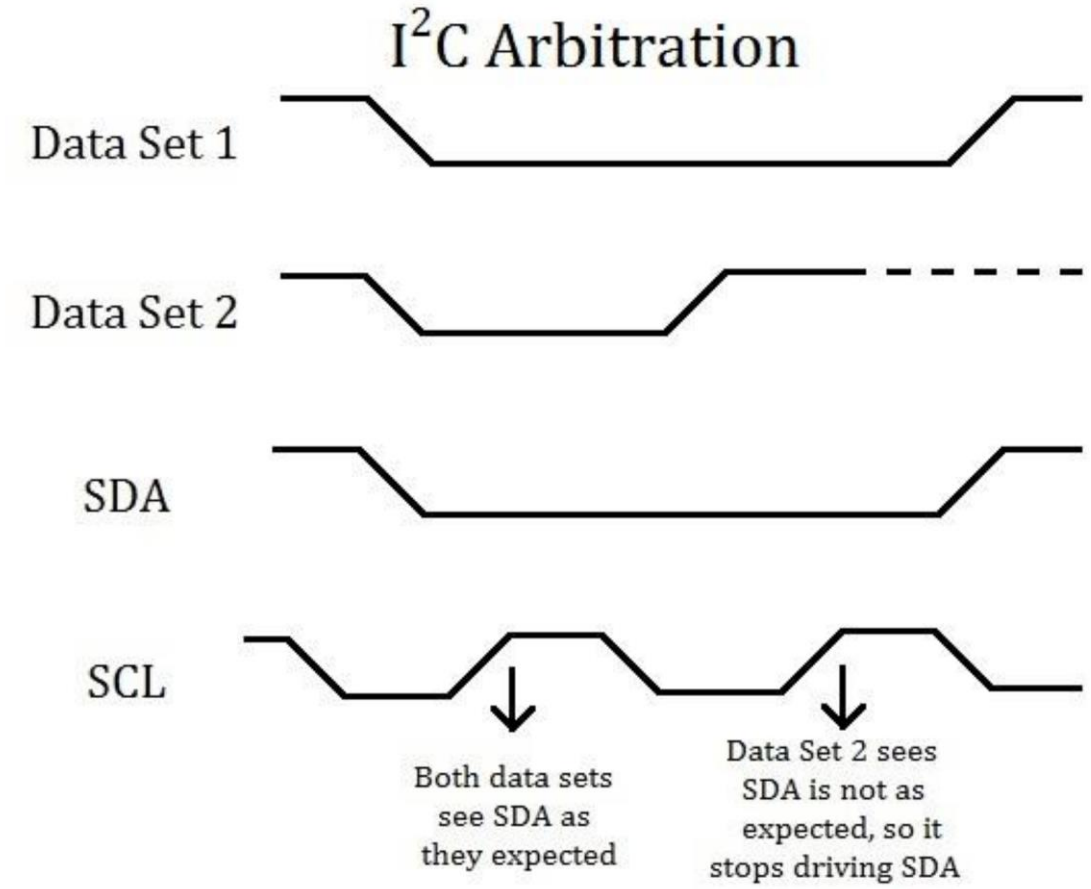
I2C frame format



I2C frame format

- Start bit: 1 bit indicates the start of a new frame, always logic low .
- Address: 7 bits that decides slave address which is the device that will process the upcoming sent data by the master .
- R/W : 1 bit to decide type of operation, logic high for read, logic low for write.
- ACK: 1 bit sent by the slave as acknowledge by polling line low .
- Data: each 8-bits of data sent is followed by ACK bit by the receiver .
- Stop bit: 1 bit indicates the end of the frame.

I2C bus arbitration mechanism



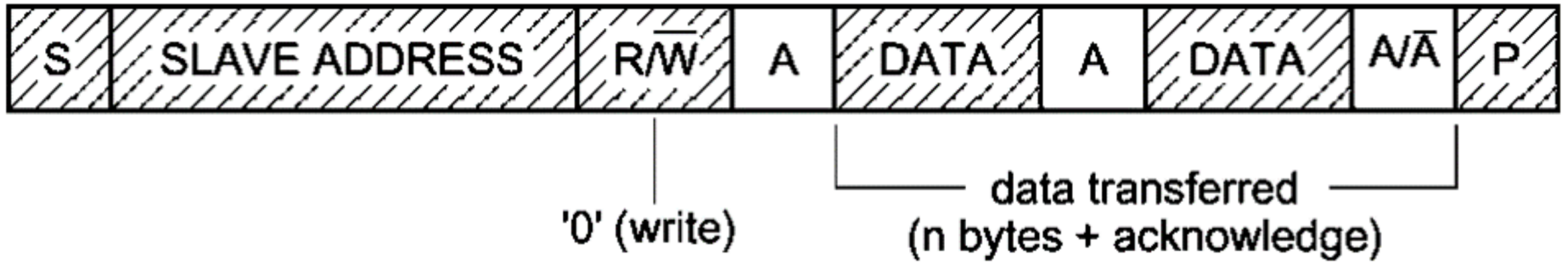
I2C bus arbitration mechanism

Case 1: Master (Transmitter) to Slave (Receiver) Data Transfer:

- The Master sends the START sequence to begin the transaction.
- It is followed by Master sending 7-bit Slave address and the R/W̄ bit set to zero. We set it to zero because the Master is writing to the Slave.
- The Slave acknowledges by pulling the ACK bit low .
- Once the Slave acknowledges the address, Master can now send data to the Slave byte-by-byte. The Slave has to send the ACK bit after every byte it receives.
- Ā (Non Ack) : This goes on till Slave can no longer receive data and does NOT send the ACK bit.
- P (Stop bit) : This is when the Master realizes that the Slave is not accepting anymore and then STOPs the transaction (or RE-START).
- An example of this case would be like performing page write operations on a EEPROM chip.

I2C bus arbitration mechanism

Case 1: Master (Transmitter) to Slave (Receiver) Data Transfer



from master to slave

from slave to master

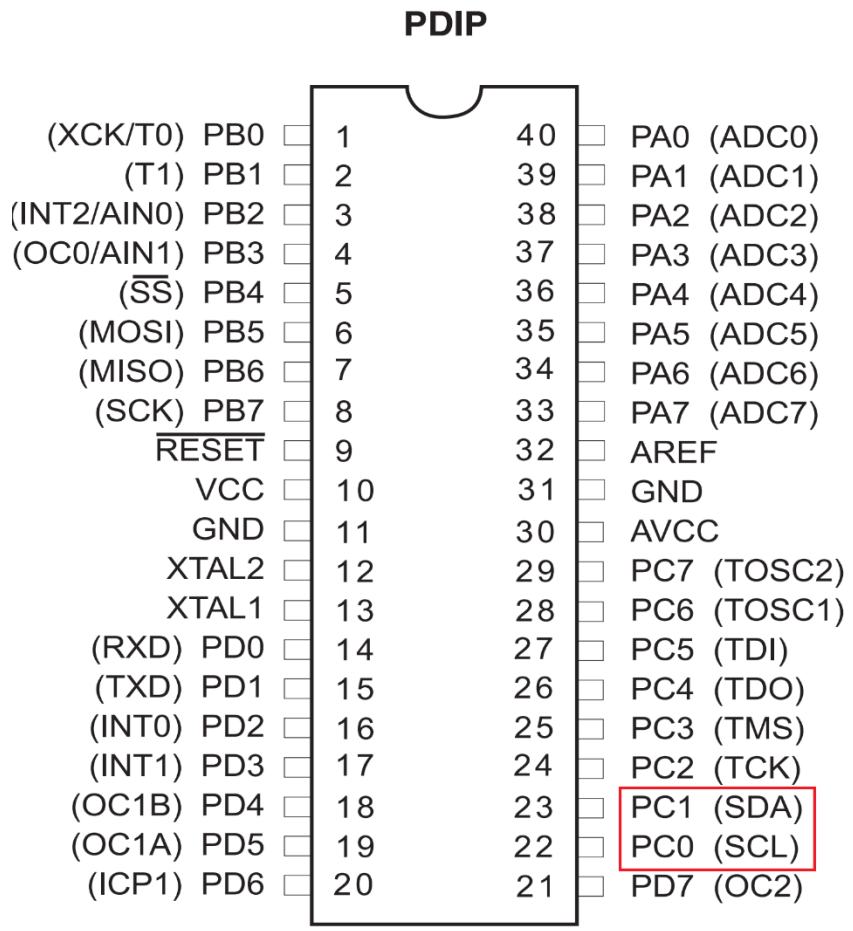
A = acknowledge (SDA LOW)

Ā = not acknowledge (SDA HIGH)

S = START condition

P = STOP condition

TWI pins


PDIP
(XCK/T0) PB0 1
(T1) PB1 2
(INT2/AIN0) PB2 3
(OC0/AIN1) PB3 4
(SS) PB4 5
(MOSI) PB5 6
(MISO) PB6 7
(SCK) PB7 8
RESET 9
VCC 10
GND 11
XTAL2 12
XTAL1 13
(RXD) PD0 14
(TXD) PD1 15
(INT0) PD2 16
(INT1) PD3 17
(OC1B) PD4 18
(OC1A) PD5 19
(ICP1) PD6 20
40 PA0 (ADC0)
39 PA1 (ADC1)
38 PA2 (ADC2)
37 PA3 (ADC3)
36 PA4 (ADC4)
35 PA5 (ADC5)
34 PA6 (ADC6)
33 PA7 (ADC7)
32 AREF
31 GND
30 AVCC
29 PC7 (TOSC2)
28 PC6 (TOSC1)
27 PC5 (TDI)
26 PC4 (TDO)
25 PC3 (TMS)
24 PC2 (TCK)
23 PC1 (SDA)
22 PC0 (SCL)
21 PD7 (OC2)

Working with TWI steps

I2C on ATMega32

- ATMega32 supports I2C (TWI: Two Wire Interface) on PC1 (SDA) and PC0 (SCL).
- It supports up to 400 KHz data transfer speed.
- Fully Programmable Slave Address with General Call Support.
- We will work in polling mode

Working with TWI steps

To deal with TWI in master mode we will follow the below steps:

1. set the working clock frequency for the TWI in this register TWBR
2. enable TWI to start in TWCR.
3. Send start bit.
4. Send address of the target slave in TWDR
5. The LSB of the TWDR decide read(1) or write(0) operation

Working with TWI steps

We will discuss how to configure TWI to be a master and send data to slave:

1. Set the CLOCK of th I2C in the **TWBR_val** register:

$$\text{SCL frequency} = \frac{\text{CPU Clock frequency}}{16 + 2(\text{TWBR}) \cdot 4^{TWPS}}$$

- TWBR = Value of the TWI Bit Rate Register
- TWPS = Value of the prescaler bits in the TWI Status Register

We decide here the speed of the clock generated from I2C in SCL pin this clock depend on FCPU clock

Working with TWI steps

In our example here we need to set the clock to be 800khz so follow the below steps:

1. **SCL freq.** = 200000, we will put **TWPS** = 0, our **F_CPU** = 8000000.

2. Calculate TWBR then update TWBR Register:

Bit	7	6	5	4	3	2	1	0	
	TWBR7	**TWBR6**	**TWBR5**	**TWBR4**	**TWBR3**	**TWBR2**	**TWBR1**	**TWBR0**	**TWBR**
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Working with TWI steps

2. TWCR is used to control the operation of the TWI. It is used to enable the TWI, to initiate a master access by applying a START condition to the bus.

Bit	7	6	5	4	3	2	1	0	
	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE	TWCR
Read/Write	R/W	R/W	R/W	R/W	R	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

TWCR = (1<<TWSTA) | (1<<TWEN);

- Enable TWI.
- Send start bit
- Put TWINT = 1 to clear it, and after the TWI finish his job it will be put automatically 1.
- So we will poll on this bit**: while(!(TWCR>>TWINT & 1));**

Working with TWI steps

3. Put the target address - of the slave you need to connect with - in TWDR :

TWDR = address<<1;

4. Then again use TWCR to start transmission of the address:

TWCR = (1<<TWINT) | (1<<TWEN);

5. Wait the transimission process till the end:

while(!(TWCR & (1<<TWINT)));

Working with TWI steps

6. Now put the data you need to transmit in TWDR, then again repeat steps 5 and 6
   - **TWCR = (1<<TWINT) | (1<<TWEN);**
   - **while(!(TWCR & (1<<TWINT)));**

7. finally send stop bit to end the connection with the slave:
   - **TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO);**

Working with TWI steps

We will discuss how to configure TWI to be a slave and receive data from master:

1. Set slave address for our slave, and enable TWI.

- **TWAR = My_Add<<1;**
- **TWCR = (1<<TWEA) | (1<<TWINT) | (1<<TWEN);**
- **while(!(TWCR & (1<<TWINT)));**

Working with TWI steps

2. wait for master to call slave:

- **TWCR |= (1<<TWINT) | (1<<TWEN);**
- **while (!(TWCR & (1<<TWINT)));**
- **if((TWSR & 0xF8) == TW_SR_SLA_ACK) return 1;**

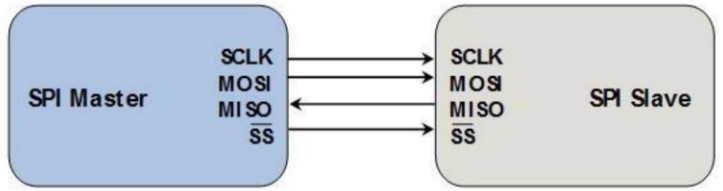
3. finally read data:

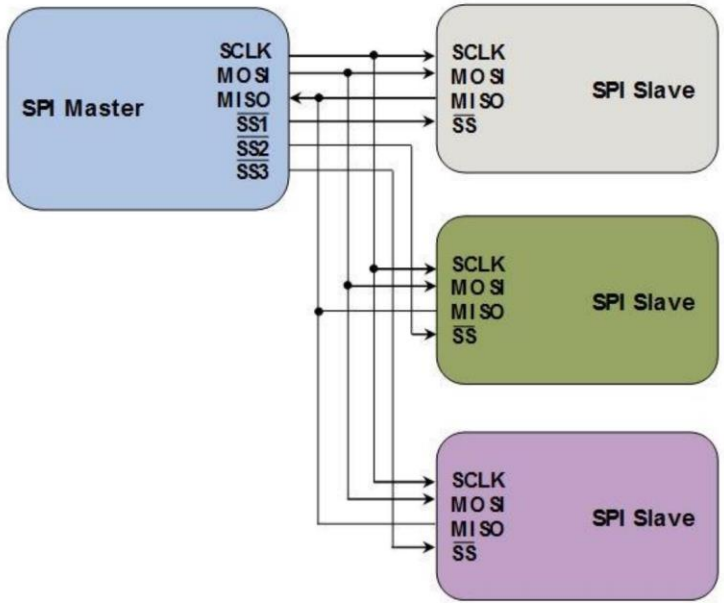
- **TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWEA);**
- **while(!(TWCR & (1<<TWINT)));**
- **return TWDR;**

Serial Peripheral Interface (SPI)

- Serial Peripheral Interface (SPI) bus is a synchronous serial communication interface specification used for short distance communication.
- The SPI bus can operate with a single master device and with one or more slave devices.
- SPI is a full duplex communication protocol.
- SPI is single master multiple slaves communication protocol.
- SPI protocol has complete flexibility for the bits transferred as there is no limit for 8-bit word or message size or purpose

Serial Peripheral Interface (SPI)



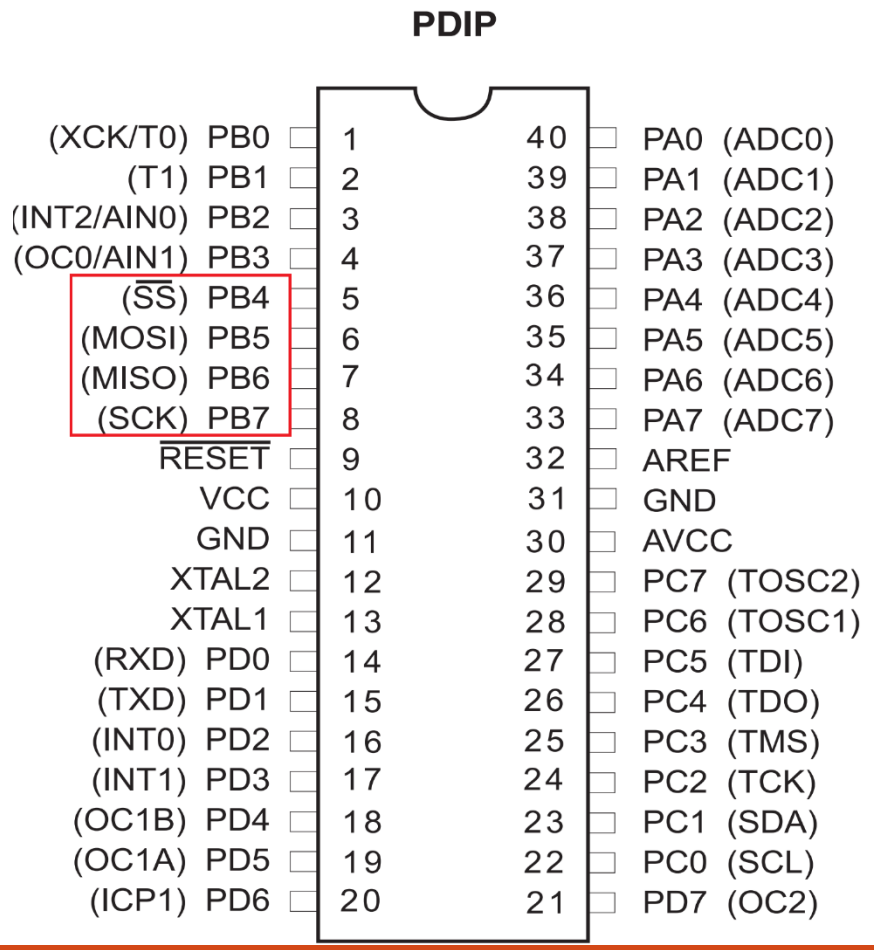


Serial Peripheral Interface (SPI)

SPI connection

- SCLK : Serial Clock (output from master).
- MOSI : Master Output, Slave Input (output from master).
- MISO : Master Input, Slave Output (output from slave).
- SS : Slave Select (active low, output from master).

SPI pins


PDIP
(XCK/T0) PB0 1
(T1) PB1 2
(INT2/AIN0) PB2 3
(OC0/AIN1) PB3 4
(SS) PB4 5
(MOSI) PB5 6
(MISO) PB6 7
(SCK) PB7 8
RESET 9
VCC 10
GND 11
XTAL2 12
XTAL1 13
(RXD) PD0 14
(TXD) PD1 15
(INT0) PD2 16
(INT1) PD3 17
(OC1B) PD4 18
(OC1A) PD5 19
(ICP1) PD6 20
40 PA0 (ADC0)
39 PA1 (ADC1)
38 PA2 (ADC2)
37 PA3 (ADC3)
36 PA4 (ADC4)
35 PA5 (ADC5)
34 PA6 (ADC6)
33 PA7 (ADC7)
32 AREF
31 GND
30 AVCC
29 PC7 (TOSC2)
28 PC6 (TOSC1)
27 PC5 (TDI)
26 PC4 (TDO)
25 PC3 (TMS)
24 PC2 (TCK)
23 PC1 (SDA)
22 PC0 (SCL)
21 PD7 (OC2)

Working with SPI steps

SPI on ATMega32

- ATMega16L supports SPI on PB5 (MOSI), PB6 (MISO), PB7 (SCK) and PB4(SS).
- It supports seven programmable bit rates.
- We will work in polling mode.

Working with SPI steps

1. Initialize the master controller:

To initialize master controller ,

- set PB4, PB5 and PB7 as output pins (SS, MOSI, SCK),
- enable SPI module through setting SPE bit in SPCR register and set the SPI module in master mode by setting MSTR bit in the same register:

```
DDRB |= (1<<PB4)|(1<<PB5)|(1<<PB7);

SPCR = (1<<SPE) | (1<<MSTR);
```

Working with SPI steps

2. Initialize the slave controller:

- On the other controller that will operate in slave mode, set PB6 as output pin (MISO) and enable SPI module:

```
DDRB |= (1<<PB6);

SPCR = (1<<SPE);
```

When a serial transfer is complete, the SPIF flag in SPSR register is set and it is cleared by hardware when executing the corresponding interrupt handling vector . Alternatively, the SPIF bit is cleared by first reading the SPI Status Register with SPIF set, then accessing the SPI Data Register (SPDR).

Working with SPI steps

3. Send 1 byte:

- To send a new byte, we just write the byte in SPDR register and wait until operation is complete (SPIF bit becomes set):

```
SPDR = data;  /* Replace it with your byte value */

while(!(SPSR & (1<<SPIF)));
```

4. Receive 1 byte:

- To receive a new byte, wait until receive operation is complete (SPIF bit becomes set) and read the byte from the SPDR register :

```
while(!(SPSR & (1<<SPIF)));

uint8 data = SPDR;
```

Contact Me

Email: sameh.m.afifi@gmail.com

Facebook: https://www.facebook.com/Same7Afifi

linkedIn: https://www.linkedin.com/in/sameh-afifi-8389173a/

Phone: 01127346781

Work phone: 224135537