Day 2

Starting remarks

Agenda

Timers/counters

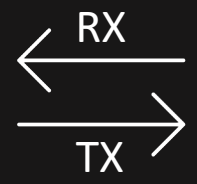
Interrupts

Task 1

Pulse width modulation

Task 2

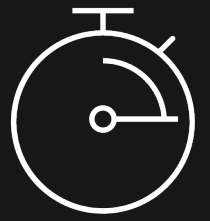
Analog to digital converter



USART

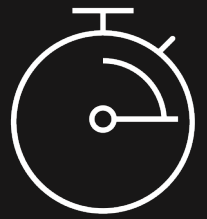
Task 3

Timers/counters – In general



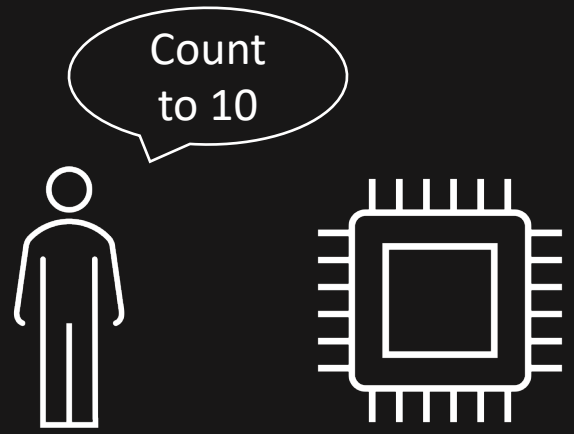
Basically, like a plain stopwatch on your phone

Timers/counters – In general

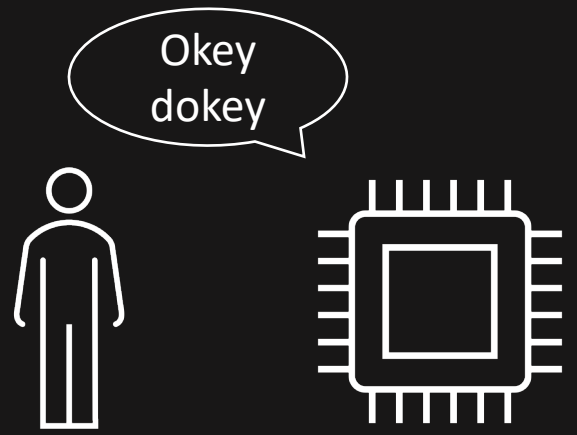


Why do we care?

Timers/counters – In general

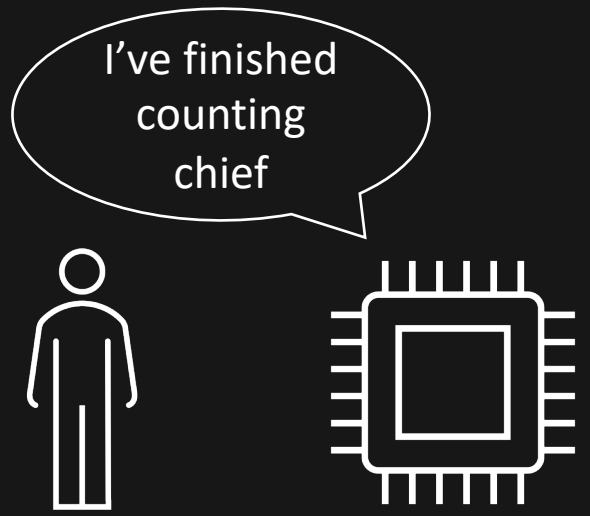


Timers/counters – In general

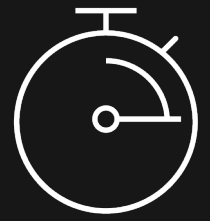


Timers/counters – In general

Timers/counters – In general



Timers/counters – In practice



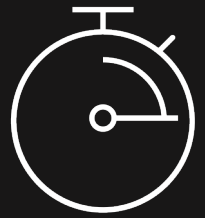
We have multiple timers/counter on the Atmega4809:

- Timer A ← We will use this one
- Timer B

We need to tell the timer/counter what *value* to count to (in practice how many clock cycles)

We need to tell how *quickly* the timer should count, with a *prescaler*

Timers/counters – Prescalers



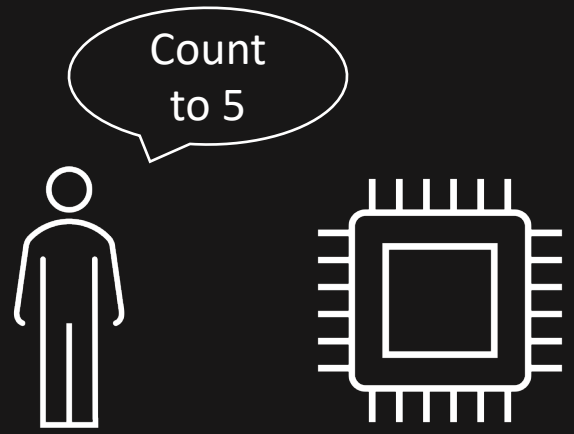
Prescalers determine the clock frequency of the timer/counter

$$frequency_{timer} = \frac{frequency_{micocontroller}}{prescaler}$$

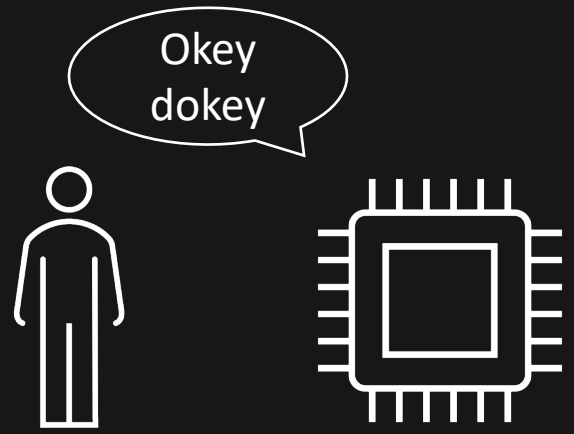
Example

We set the prescaler to 2 so that the timer counts twice as slow as the microcontroller frequency

Timers/counters – Prescalers

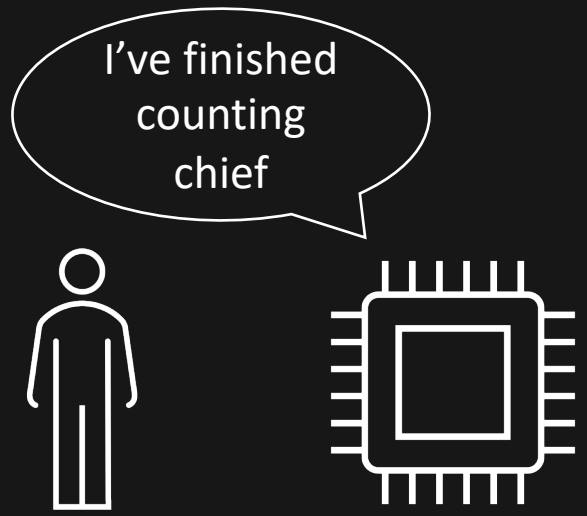


Timers/counters – Prescalers

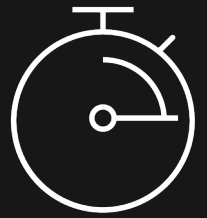


Timers/counters – Prescalers

Timers/counters – Prescalers



Timers/counters – Registers



We need to tell the timer/counter what *value* to count to (in practice how many clock cycles)

TCA0.SINGLE.PER = *some_value*;

- The timer we want to use
- We're interested in the timer running in single mode
- The register holding the value the timer will count to

20.5.15 Period Register - Normal Mode

Name: PER
Offset: 0x26
Reset: 0xFFFF
Property: -

TCAn.PER contains the 16-bit TOP value in the timer/counter in all modes of operation, except Frequency Waveform Generation (FRQ).

The TCAn.PERL and TCAn.PERH register pair represents the 16-bit value, TCAn.PER. The low byte [7:0] (suffix L) is accessible at the original offset. The high byte [15:8] (suffix H) can be accessed at offset + 0x01.

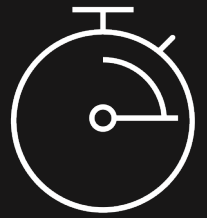
Bit	15	14	13	12	11	10	9	8
	PER[15:8]							
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	1	1	1	1	1	1	1	1

Bit	7	6	5	4	3	2	1	0
	PER[7:0]							
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	1	1	1	1	1	1	1	1

Bits 15:8 – PER[15:8] Periodic High Byte
These bits hold the MSB of the 16-bit Period register.

Bits 7:0 – PER[7:0] Periodic Low Byte
These bits hold the LSB of the 16-bit Period register.

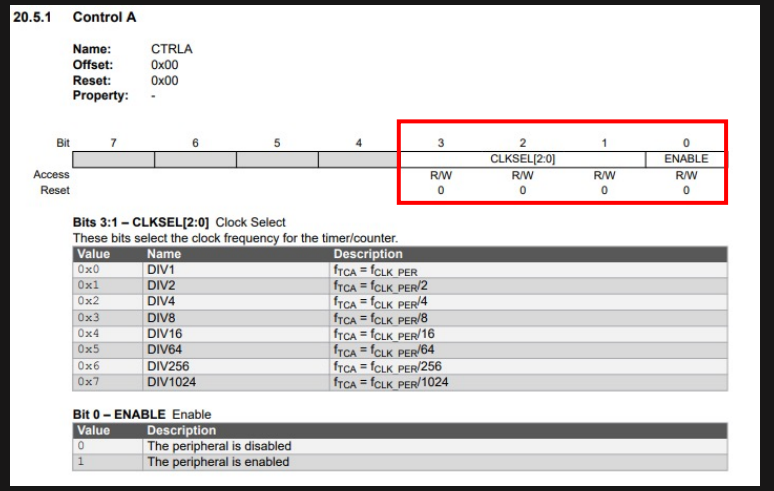
Timers/counters – Registers



We need to tell how *quickly* the timer should count, with a *prescaler*

```
TCA0.SINGLE.CTRLA |= TCA_SINGLE_CLKSEL_DIV2_gc;
```

- The timer we want to use
- We're interested in the timer running in single mode
- Register holding the prescaler
- Use bitwise or and pass in the *group mask* for setting the prescaler to 2



20.5.1 Control A

Name: CTRLA
Offset: 0x00
Reset: 0x00
Property: -

Bit	7	6	5	4	3	2	1	0
					CLKSEL[2:0]			ENABLE
Access					R/W	R/W	R/W	R/W
Reset					0	0	0	0

Bits 3:1 – CLKSEL[2:0] Clock Select
These bits select the clock frequency for the timer/counter.

Value	Name	Description
0x0	DIV1	$f_{TCA} = f_{CLK\_PER}$
0x1	DIV2	$f_{TCA} = f_{CLK\_PER}/2$
0x2	DIV4	$f_{TCA} = f_{CLK\_PER}/4$
0x3	DIV8	$f_{TCA} = f_{CLK\_PER}/8$
0x4	DIV16	$f_{TCA} = f_{CLK\_PER}/16$
0x5	DIV64	$f_{TCA} = f_{CLK\_PER}/64$
0x6	DIV256	$f_{TCA} = f_{CLK\_PER}/256$
0x7	DIV1024	$f_{TCA} = f_{CLK\_PER}/1024$

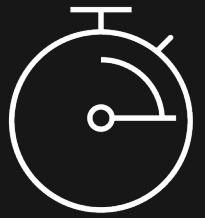
Bit 0 – ENABLE Enable

Value	Description
0	The peripheral is disabled
1	The peripheral is enabled

We also need to *enable* the timer

```
TCA0.SINGLE.CTRLA |= TCA_SINGLE_ENABLE_bm;
```

Timers/counters – Single?

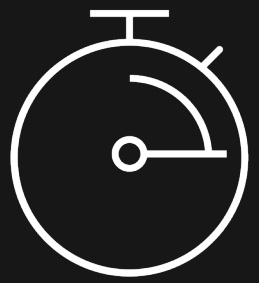


```
TCA0.SINGLE.CTRLA |= TCA_SINGLE_CLKSEL_DIV2_gc;
```

What is this?

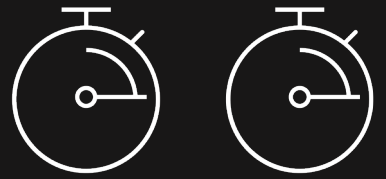
The timer can run in *single* (16-bit) or *split* (two 8-bit)

Single



Can count to 65 535 (the maximum value of a 16-bit number)

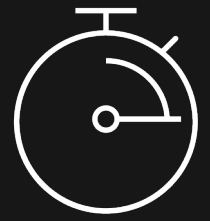
Split



Can count to 255 (the maximum value of an 8-bit number)

Can count to 255 (the maximum value of an 8-bit number)

Timers/counters



Questions?

Interrupts – In general

Basically, like a notification
on your phone

Interrupts – In general

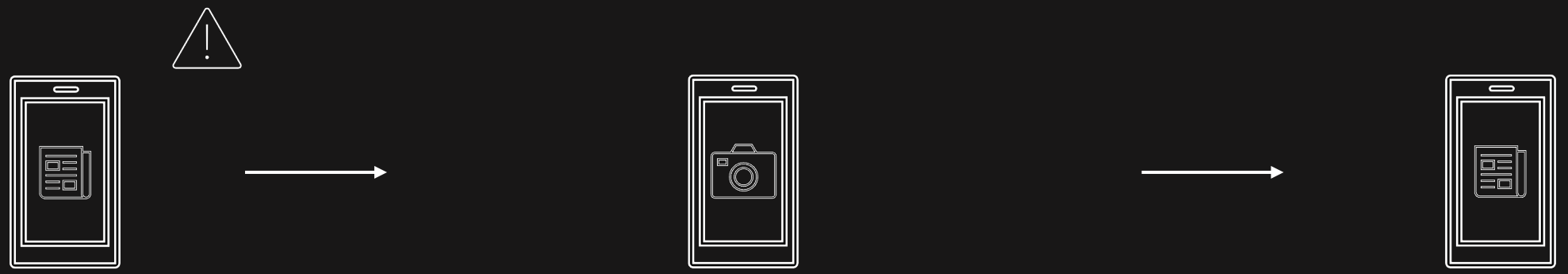
Why do we care?

Interrupts – What are they?

Code blocks executed at some condition

Stops the main program flow temporarily, resumes immediately after the interrupt

Interrupts – Example



ISR = Interrupt service routine

The interrupt *vector* name

```
int main() {

        while (true) {
                // Doing some
                // work here
                // More code…
        }
}
```

```
ISR (TCA0_OVF_vect) {
        // Do something when the timer has
        // finished counting to the value

        // Tell the timer that we've received the
        // notification (acknowledge the interrupt)
        TCA.SINGLE.INTFLAGS = TCA_SINGLE_OVF_bm;
}
```

```
int main() {

        while (true) {
                // Doing some
                // work here
                // More code…
        }
}
```

Interrupts – How do we configure these things?

We must enable interrupts *globally.* Done by calling *sei();*

We must enable the corresponding interrupt bit for the module we want interrupts from

20.5.10 Interrupt Control Register - Normal Mode

Name: INTCTRL
Offset: 0x0A
Reset: 0x00
Property: -

Bit	7	6	5	4	3	2	1	0
		CMP2	CMP1	CMP0				OVF
Access		R/W	R/W	R/W				R/W
Reset		0	0	0				0

Bit 6 – CMP2 Compare Channel 2 Interrupt Enable
See CMP0.

Bit 5 – CMP1 Compare Channel 1 Interrupt Enable
See CMP0.

Bit 4 – CMP0 Compare Channel 0 Interrupt Enable
Writing the CMPn bit to '1' enables the interrupt from Compare Channel n.

Bit 0 – OVF Timer Overflow/Underflow Interrupt Enable
Writing the OVF bit to '1' enables the overflow/underflow interrupt.

We must include the code for the *interrupt service routing* with the correct *interrupt vector name*

```
ISR (TCA0_OVF_vect) {
        // Some code
        TCA.SINGLE.INTFLAGS = TCA_SINGLE_OVF_bm;
}
```

Interrupts – Closing remarks

Keep them short

Interrupts are in most cases a better solution to your problem than using polling

Interrupts

Questions?

Timers/counters Interrupts

Task 1

Pulse width modulation

Task 2

Analog signals USART

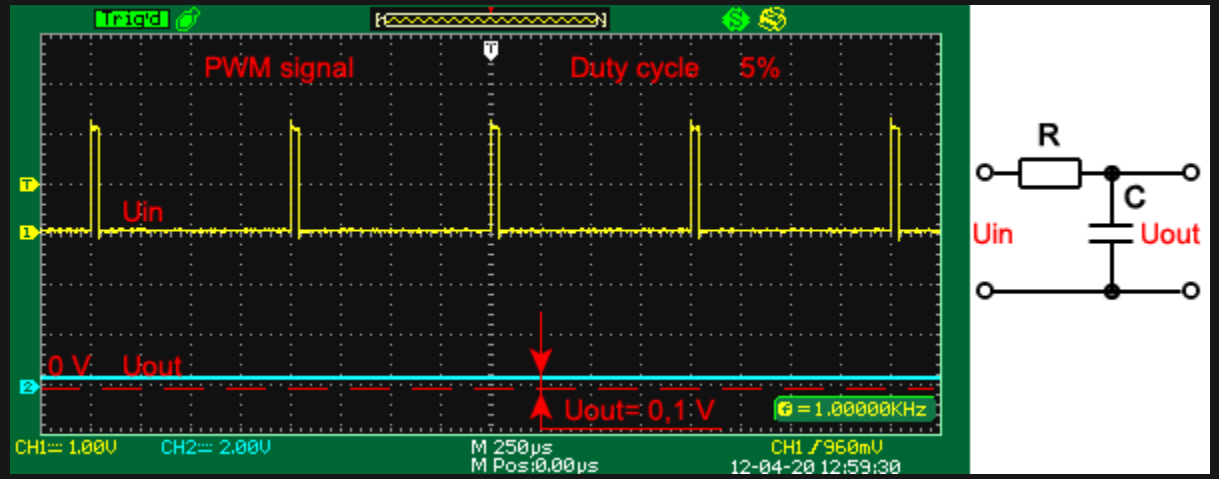
Task 3

PWM

Demo

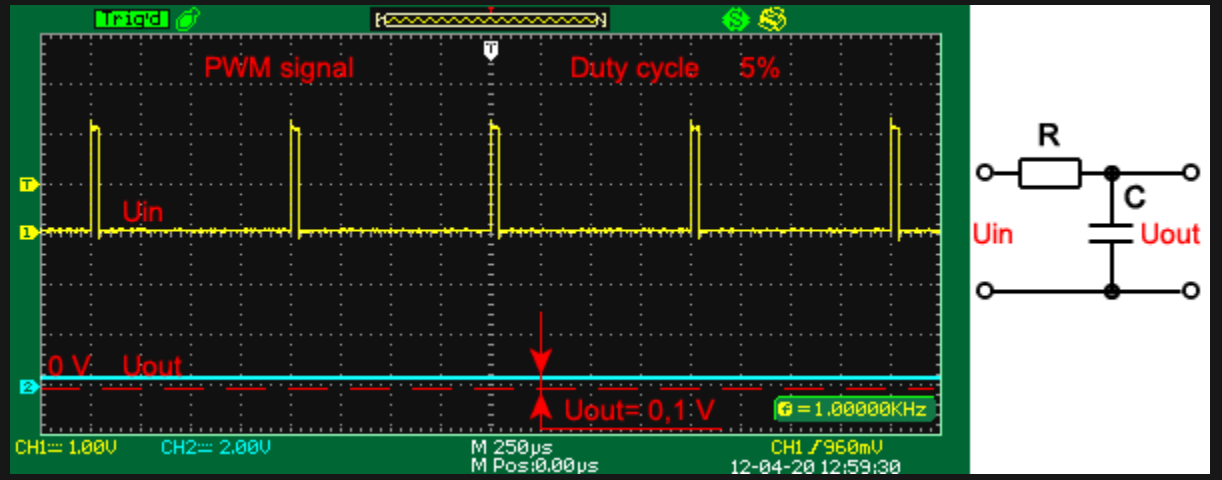
PWM – In general

A signal where we can vary how long the signal is low compared to high



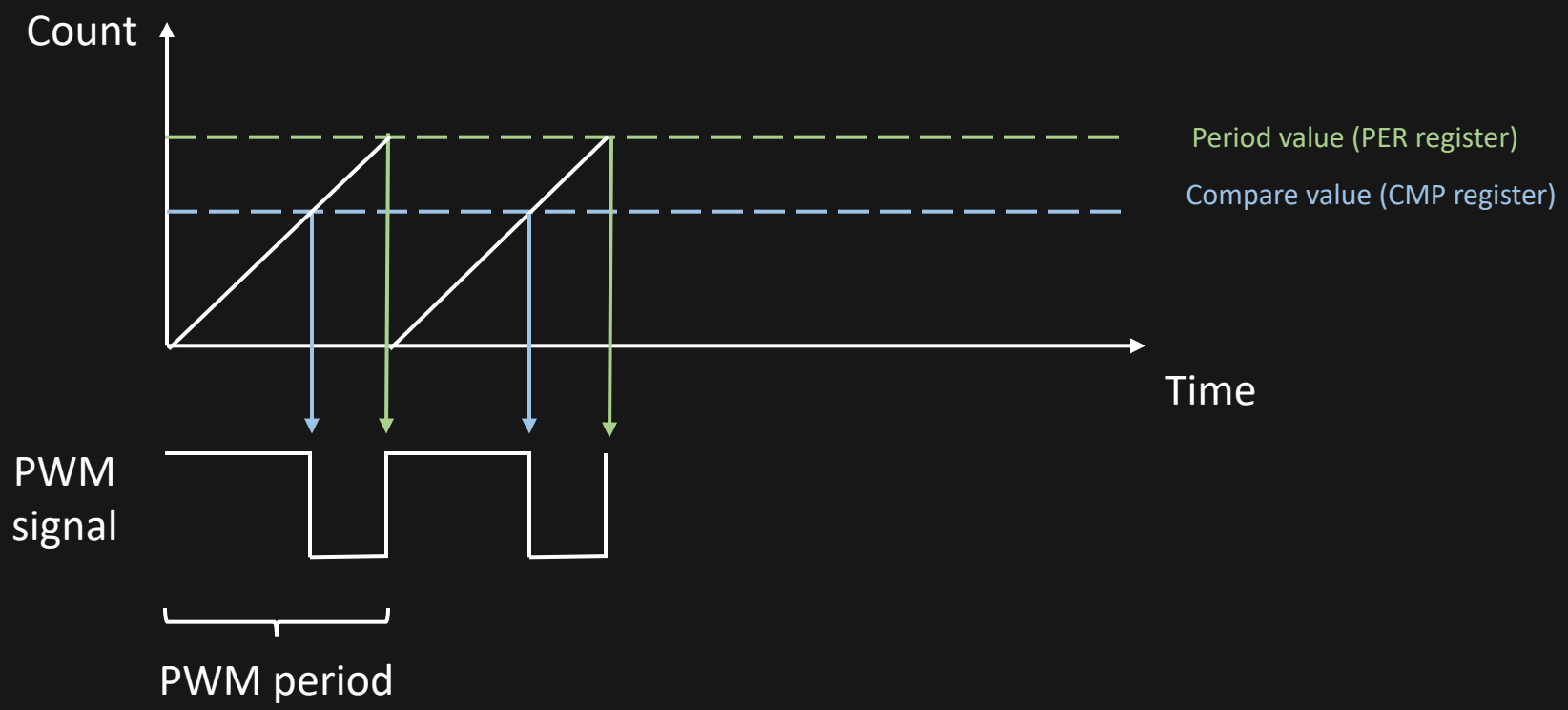
PWM – In general

Why do we care?



PWM – With the timer/counter modules

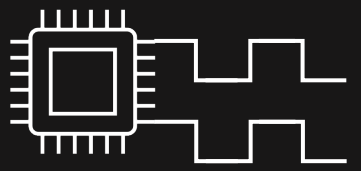
We can make the timers/counters output a PWM signal on a pin



PWM – With the timer/counter modules

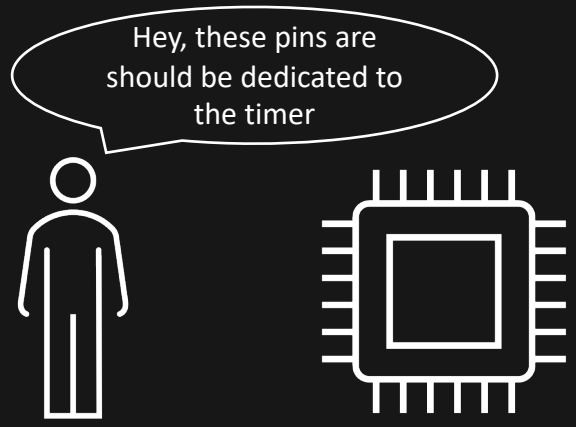
We want to output a PWM signal on two pins for this task

- We need to use split mode for the timer (every register should be on the form of TCA0.SPLIT.<register> = some_value;)
- We then have two compare values (CMP1 and CMP2)



We need to tell the microcontroller what pins it should output the PWM signal on

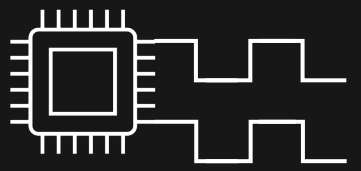
- We need to use *port multiplexing*



PWM – With the timer/counter modules

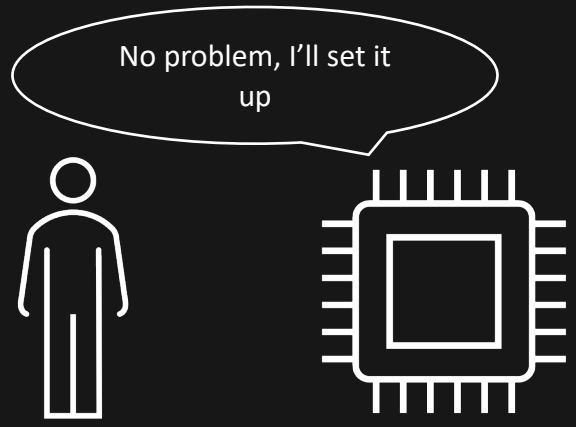
We want to output a PWM signal on two pins for this task

- We need to use split mode for the timer (every register should be on the form of TCA0.SPLIT.<register> = some_value;)
- We then have two compare values (CMP1 and CMP2)



We need to tell the microcontroller what pins it should output the PWM signal on

- We need to use *port multiplexing*



PWM – Closing remarks

This is a task people usually struggle with, so read the comments in your handed out code thoroughly. It will point you in the correct direction.

PWM

Questions?

Timers/counters

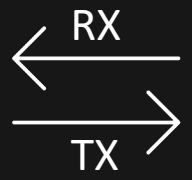
Interrupts

Task 1

Pulse width modulation

Task 2

Analog to digital converter



USART

Task 3

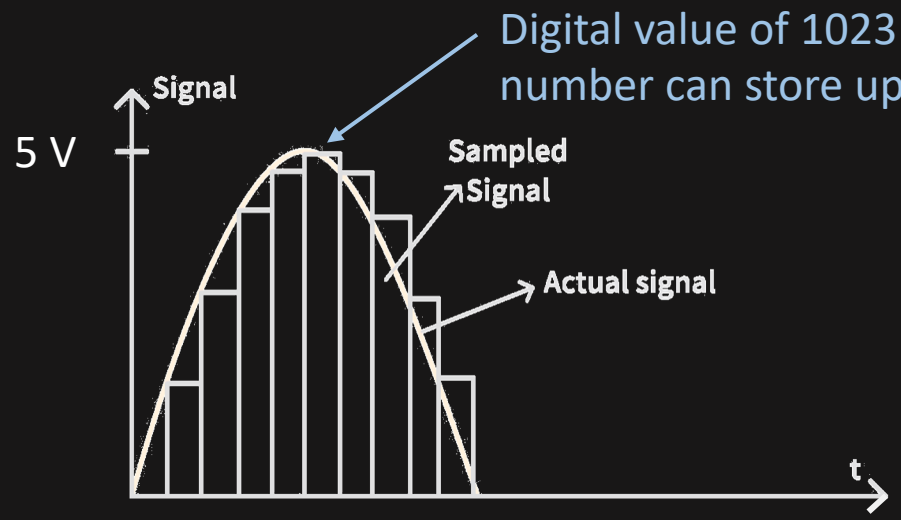
Analog to digital converter – In general

Demo

Analog to digital converter – In general

Converts analog signals to digital values

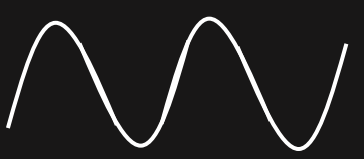
Analog to digital converter – In general

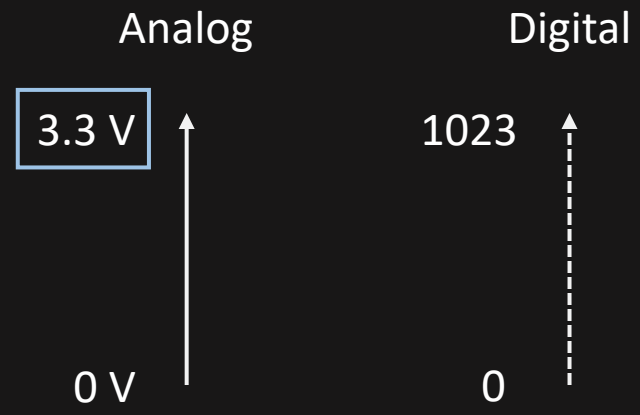


Analog to digital converter – In general

Why do we care?

Analog to digital converter – What we need to setup

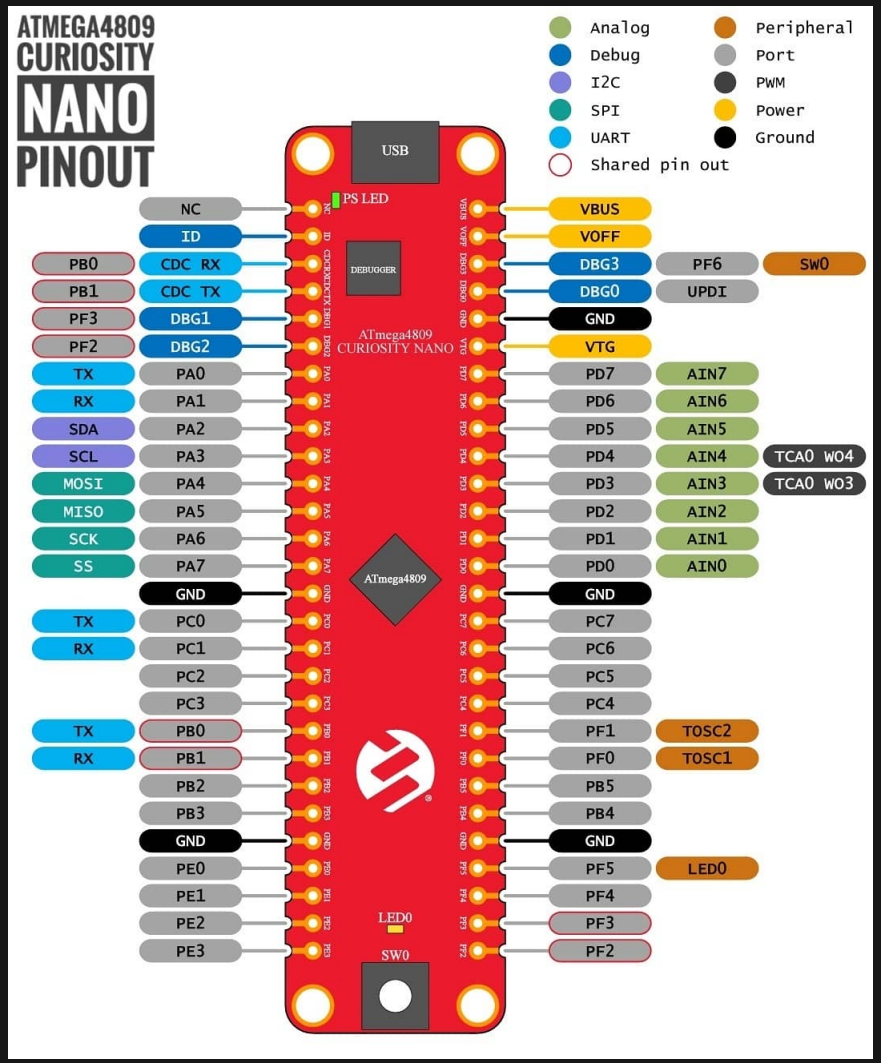




1. Set bit resolution (10 bits in this task)
2. Set the number of samples per conversion
3. Set the reference voltage
4. Set the prescaler (how fast the ADC will run and thus sample signal)
5. Enable the ADC

Analog to digital converter – How we get a sample

1. Choose which ADC channel to use
2. Start a conversion
3. Wait for it to complete
4. Read the value



The ADC has 8 channels

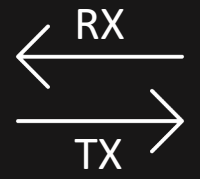
Analog to digital converter - Closing remarks

All the steps outlined are commented in the code with the datasheet sections. Ask us you have any questions

Analog to digital converter

Questions?

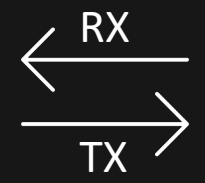
USART

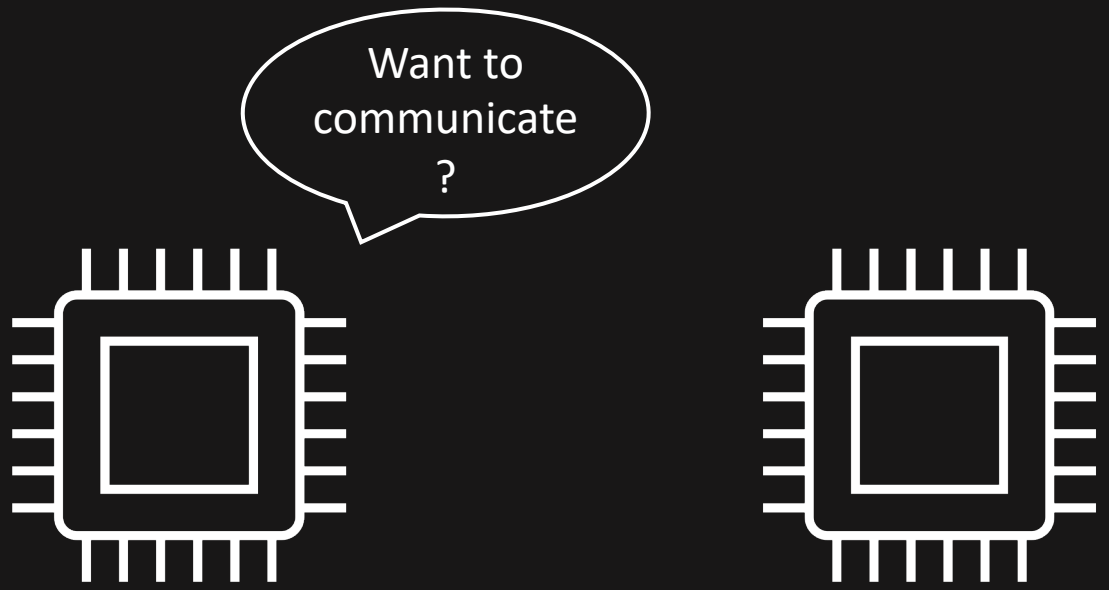


A way to send and receive data

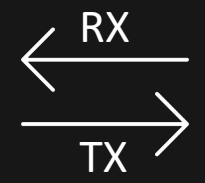
Universal Synchronous and Asynchronous Receiver and Transmitter

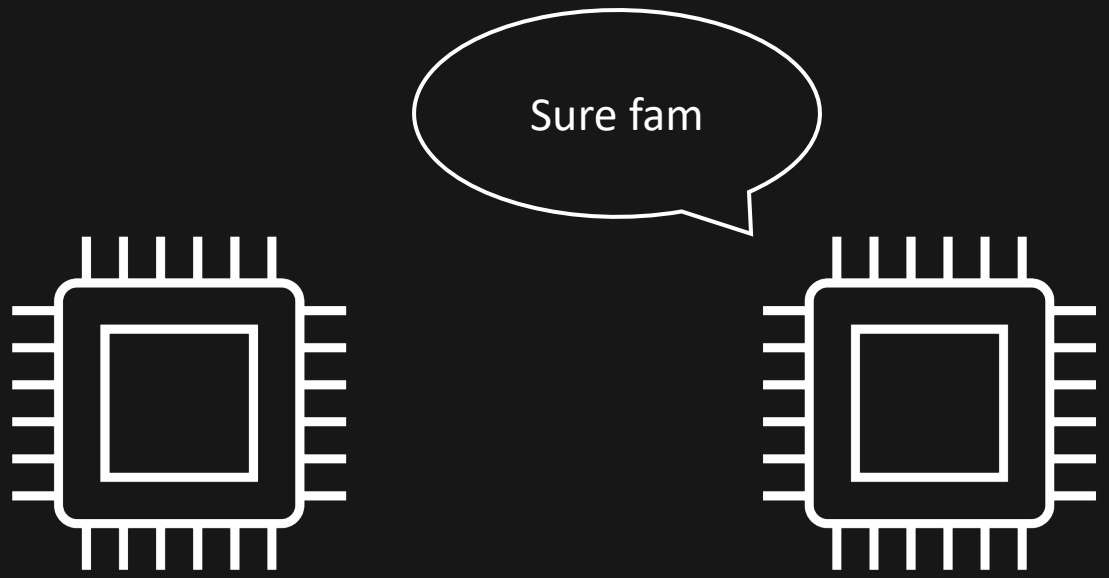
USART



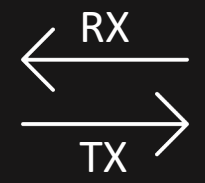


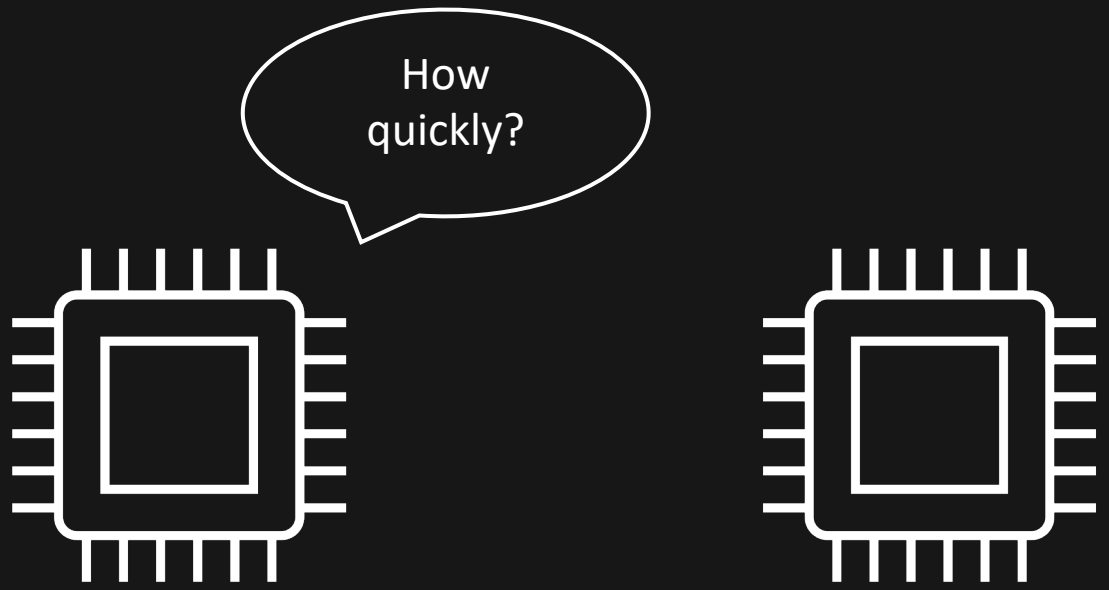
USART



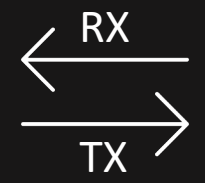


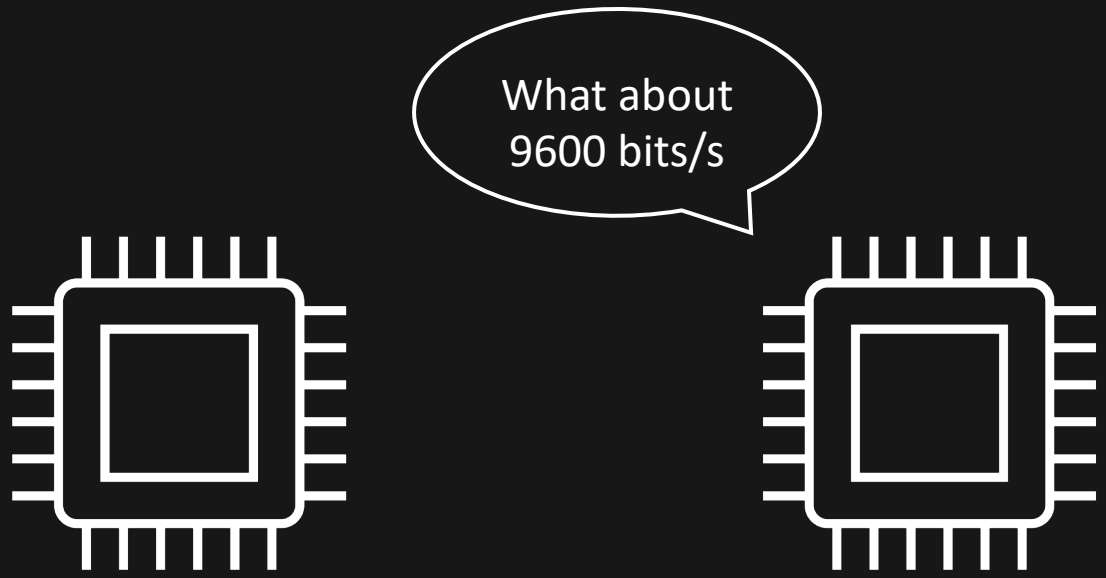
USART



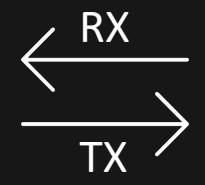


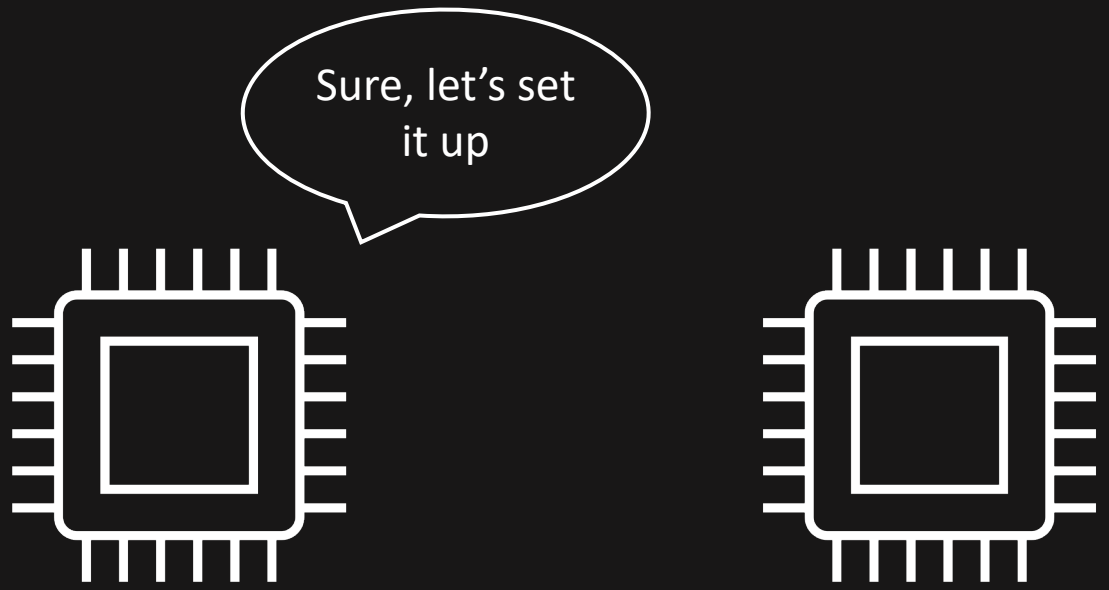
USART



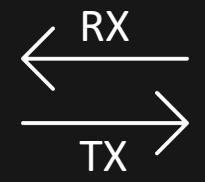


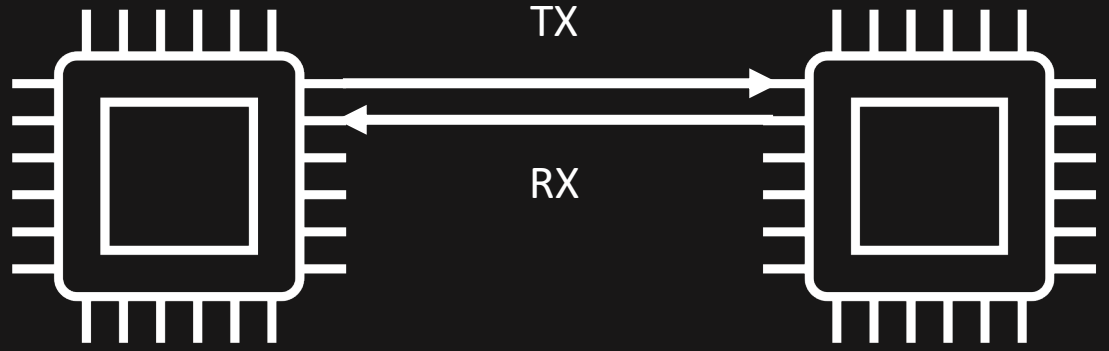
USART



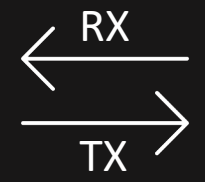


USART

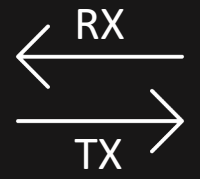




USART

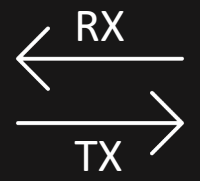


USART



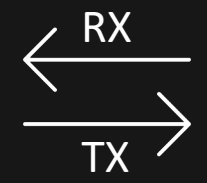
Why do we care?

USART – Configurations



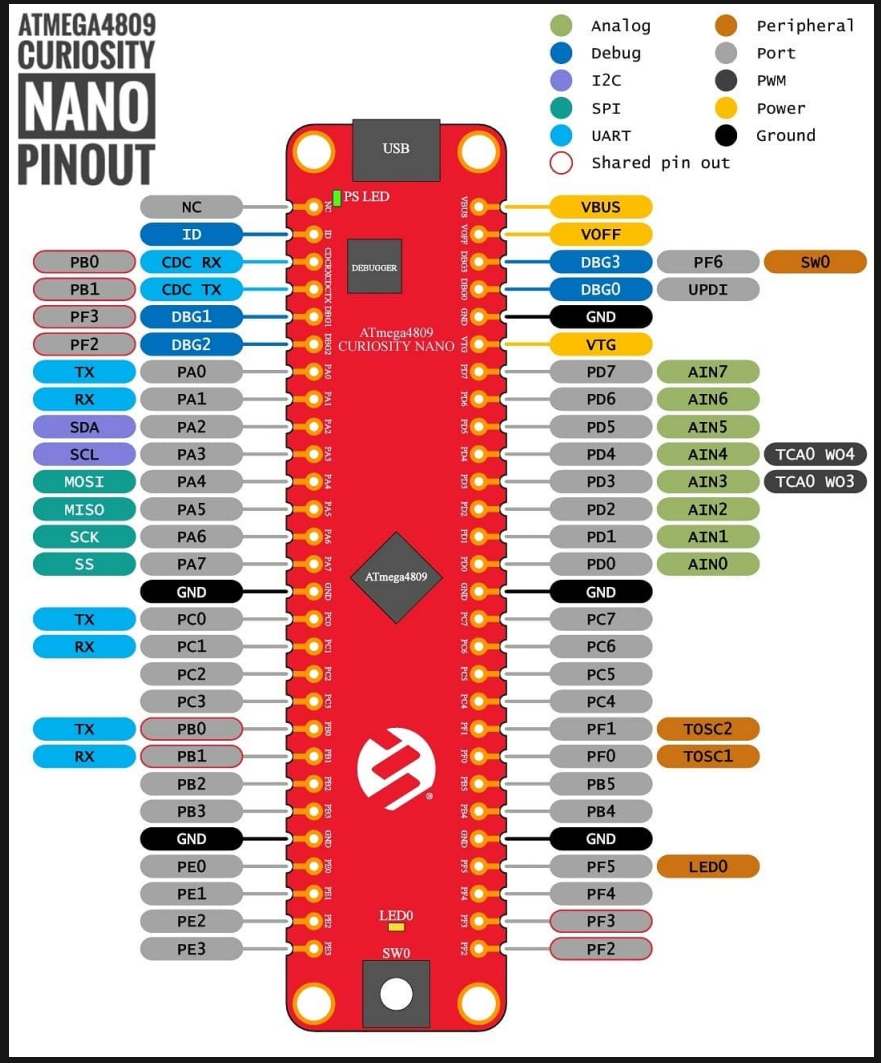
- Need to specify baud rate
- Need to specify bits per transfer (usually 8, so 1 byte per transfer).
- There are some more configurations as well, but we won’t go into them (we’ll use the default values)

USART – How do we set it up?

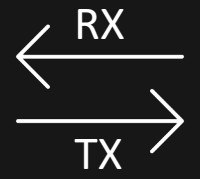


1. Need to enable the TX pin for output (and optionally the RX pin for input).
2. Need to set the baudrate (bits per second) in the USART's BAUD register. Both sides need to have the same baudrate.
3. Enable transmitter (and optionally receiver) for the USART module (in the CTRLB register of the USART module).

USART pins we're going to use



USART – Baud rate formula example



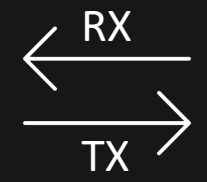
$$BAUD_{VALUE} = \frac{4 * frequency_{microcontroller}}{frequency_{baud}}$$

$$BAUD_{VALUE} = \frac{4 * \frac{20\,000\,000}{6}}{9600}$$

$$BAUD_{VALUE} \approx 1388$$

```
USART3.BAUDL  = (uint8_t) 1388          = 108;
USART3.BAUDH = (uint8_t) (1388 >> 8)  = 5;
```

USART – Sending characters



1. Check if data is not currently being sent (USART's STATUS register).
2. Set USART's TXDATAL register to some value to send one byte.

23.5.5 USART Status Register

Name: STATUS
Offset: 0x04
Reset: 0x00
Property: -

Bit	7	6	5	4	3	2	1	0
	RXCIF	TXCIF	DREIF	RXSIF	ISFIF		BDF	WFB
Access	R	R/W	R	R/W	R/W		R/W	R/W
Reset	0	0	1	0	0		0	0

Bit 7 – RXCIF USART Receive Complete Interrupt Flag
This flag is set to '1' when there are unread data in the receive buffer and cleared when the receive buffer is empty (that is, does not contain any unread data). When the receiver is disabled the receive buffer will be flushed and, consequently, the RXCIF bit will become '0'.
When interrupt-driven data reception is used, the receive complete interrupt routine must read the received data from RXDATA in order to clear the RXCIF. If not, a new interrupt will occur directly after the return from the current interrupt.

Bit 6 – TXCIF USART Transmit Complete Interrupt Flag
This flag is set when the entire frame in the Transmit Shift register has been shifted out, and there are no new data in the transmit buffer (TXDATA).
This flag is automatically cleared when the transmit complete interrupt vector is executed. The flag can also be cleared by writing a '1' to its bit location.

Bit 5 – DREIF USART Data Register Empty Flag
This flag indicates if the transmit buffer (TXDATA) is ready to receive new data. The flag is set to '1' when the transmit buffer is empty and is '0' when the transmit buffer contains data to be transmitted but has not yet been moved into the Shift register. The DREIF bit is set after a Reset to indicate that the transmitter is ready. Always write this bit to '0' when writing the STATUS register.
DREIF is cleared to '0' by writing TXDATAL. When interrupt-driven data transmission is used, the Data Register Empty interrupt routine must either write new data to TXDATA in order to clear DREIF or disable the Data Register Empty interrupt. If not, a new interrupt will occur directly after the return from the current interrupt.

23.5.3 Transmit Data Register Low Byte

Name: TXDATAL
Offset: 0x02
Reset: 0x00
Property: -

The Transmit Data Buffer (TXB) register will be the destination for data written to the USARTn.TXDATAL register location.

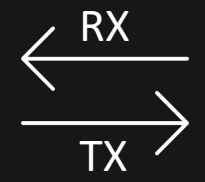
For 5-, 6-, or 7-bit characters the upper, unused bits will be ignored by the transmitter and set to zero by the receiver.

The transmit buffer can only be written when the DREIF flag in the USARTn.STATUS register is set. Data written to the DATA bits when the DREIF flag is not set will be ignored by the USART transmitter. When data are written to the transmit buffer, and the transmitter is enabled, the transmitter will load the data into the Transmit Shift register when the Shift register is empty. The data are then transmitted on the TXD pin.

Bit	7	6	5	4	3	2	1	0
	DATA[7:0]							
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

Bits 7:0 – DATA[7:0] Transmit Data Register

USART – Sending strings

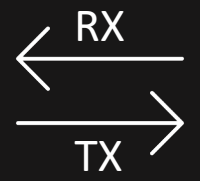


Strings in C are character arrays
terminated with a null character (’\0’)

const char* hello_string = “hello”;

h	e	l	l	o	\0

USART



Questions?

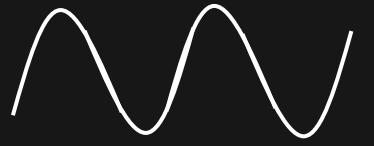
Timers/counters

Interrupts

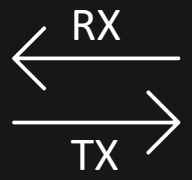
Task 1

Pulse width modulation

Task 2



Analog to digital converter



USART

Task 3