Eat pizza – drink brus
we will start ~17:45

pick up your kits before that
(and pay for the training)

(AVR) Microcontroller Basics

A Leading Provider of Smart, Connected and Secure Embedded Control Solutions

Egil Rotevatn with Snorre Vestli, Amund Aune and Runar Sivertsen

28.03.2022

µBedpress

Leading Total Systems Solutions Provider:

- High-performance standard and specialized Microcontrollers, Digital Signal Controllers and Microprocessors
- Mixed-Signal, Analog, Interface and Security solutions
- Clock and Timing solutions
- Wireless and Wired Connectivity solutions
- FPGA solutions
- Non-volatile EEPROM and Flash Memory solutions
- Flash IP solutions

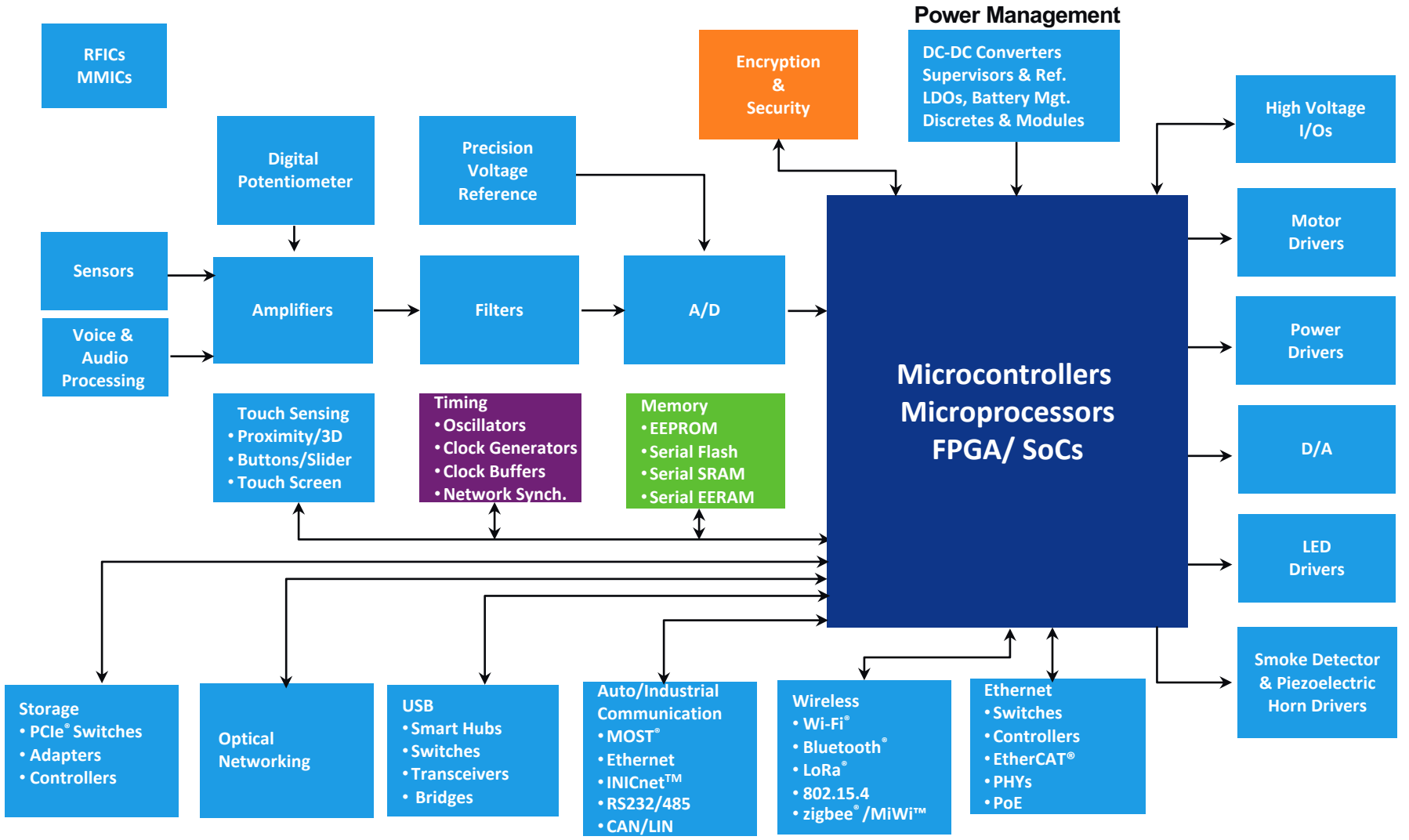
$5.3 Billion Revenue
FY2020

Headquartered near
Phoenix in Chandler, AZ

~18,000
Employees

MICROCHIP

µBedpress



MICROCHIP

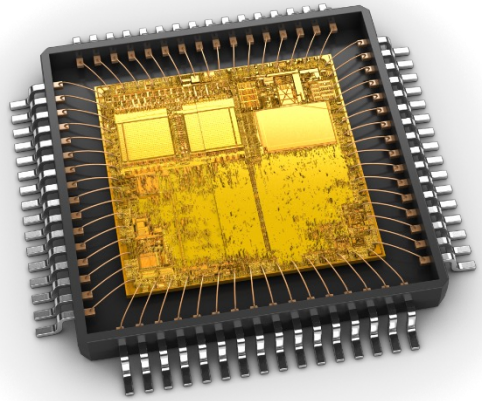
µBedpress

MICROCHIP

µBedpress

Microchip Technology Norway AS

- ~130 employees
- Located at Lyngården, Moholt
- Design, Test, Tools, Applications, Marketing
  - 8-bit Microcontrollers – AVR
  - Human-Machine Interface – maXTouch
  - Tools, Microchip Studio, MPLAB X IDE, Code composers

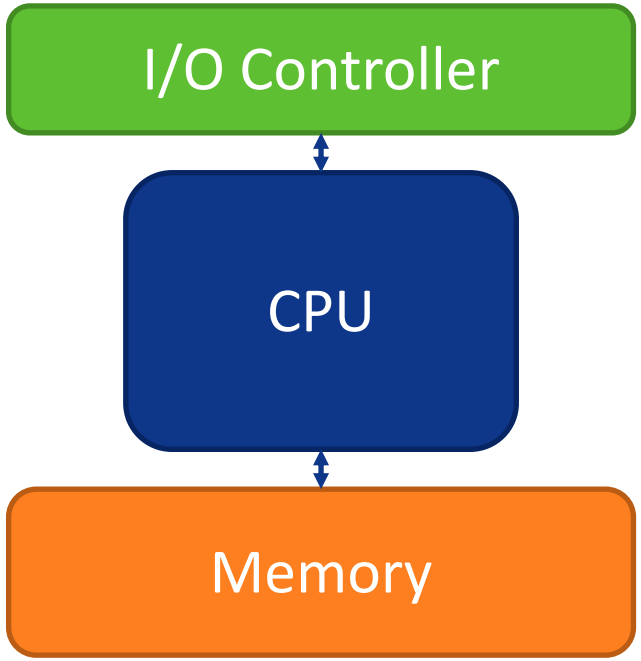


Microcontroller

The basics

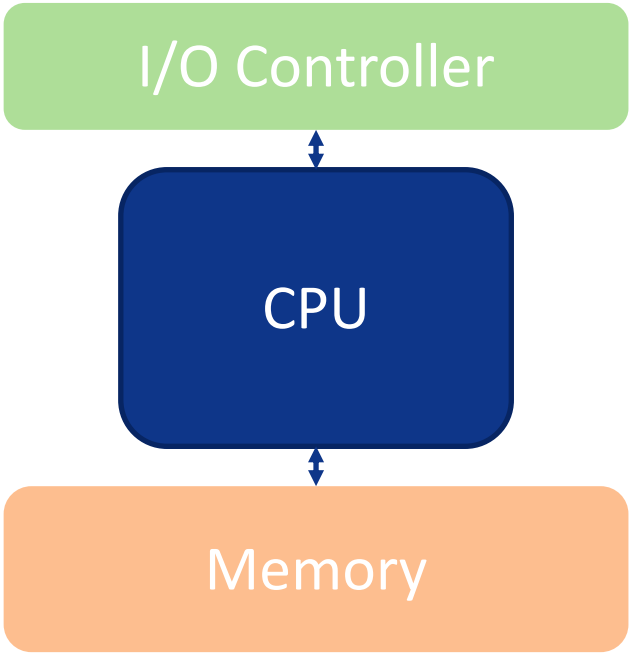
Basic Parts

- **A microcontroller must have**
  - A CPU
  - Memory
  - I/O controller



The CPU

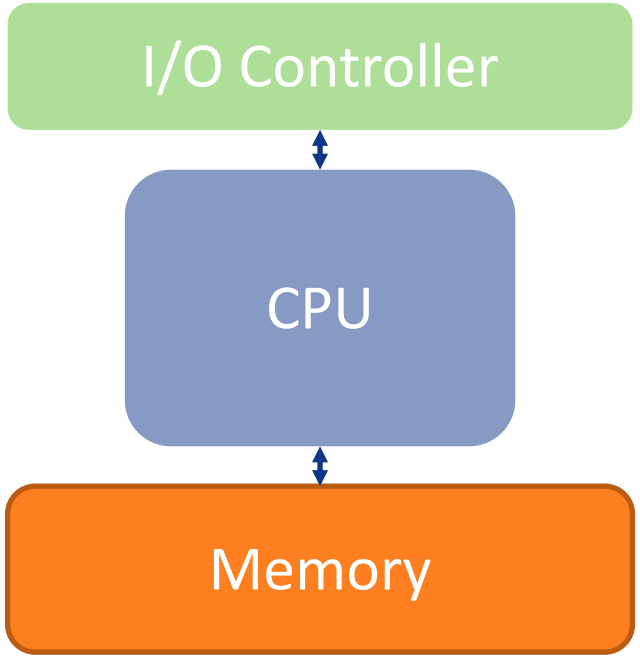
- This is what executes your program
- Can do math
- Can make decisions based on input
- Runs the code stored in memory




```
if (mentor == smart){
        printf("Of course!");
} else
{
        printf("Lies!");
}
```

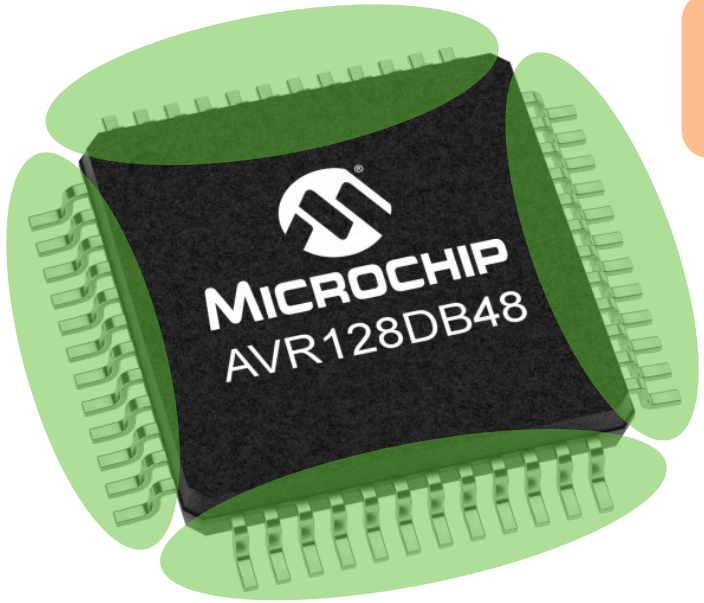
Memory

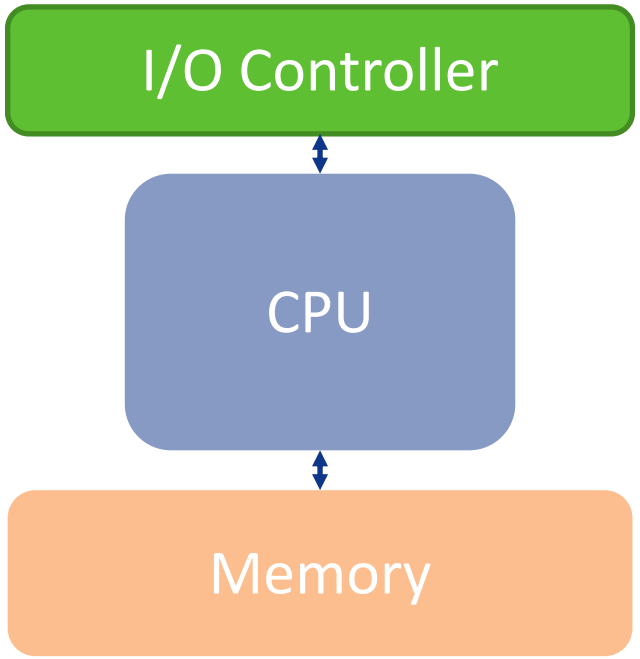
- **Stores the program**
  - Read and executed by the CPU
- **Stores temporary data**
  - Variables are saved here
- **Stores static data**
  - Not deleted when device is turned off



I/O Controller

- **Pins are used to communicate with the outside world**
- **Controls input/output pins**
  - Can set pin high or low
  - Can read if pin is pulled high or low
  - High = V_{DD} = input voltage
  - Low = GND = 0V
- **Some pins can output or read any voltage**



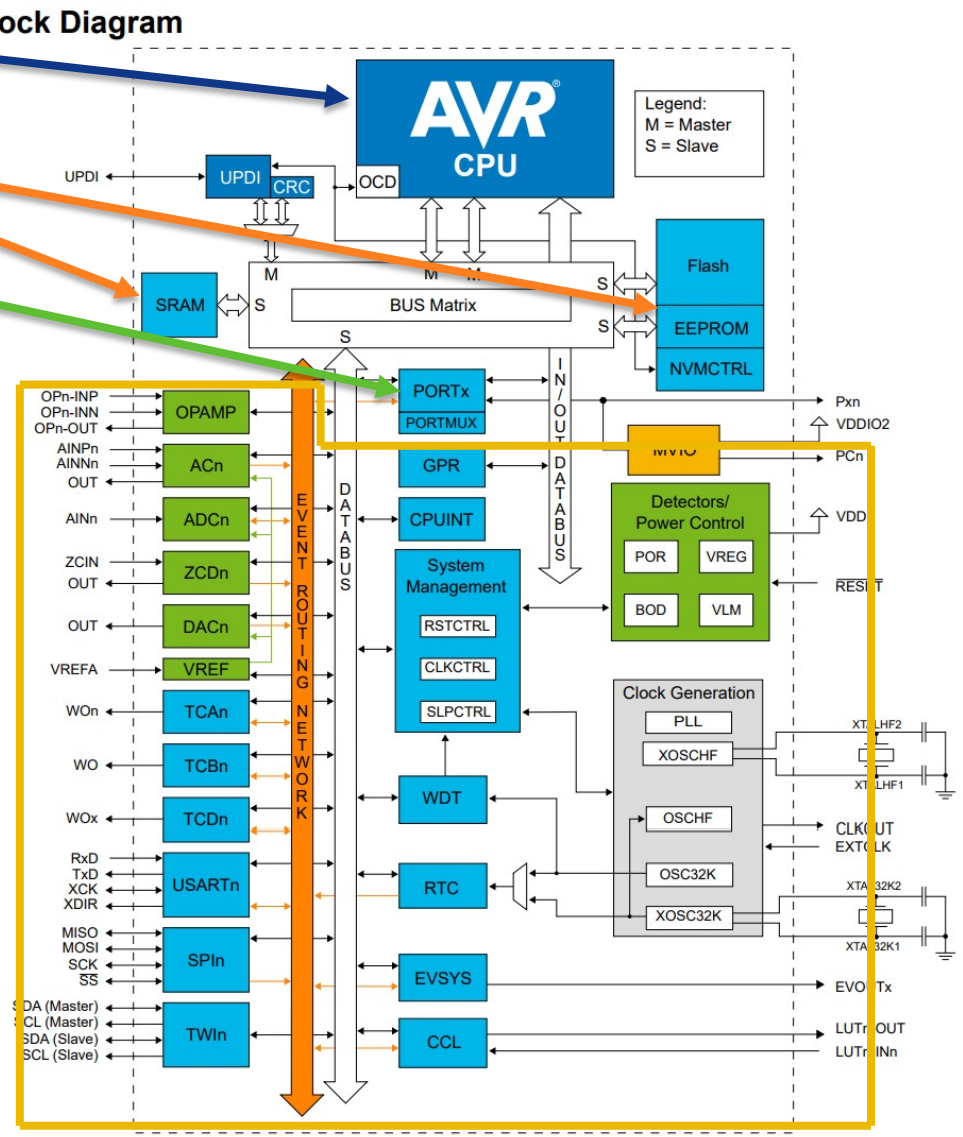


Microcontroller

- **It's that easy!**

…Okay, not quite.

- **Microcontrollers normally have more stuff**
  - Called Peripherals
  - Vary from type to type
  - Explained in the Data Sheet



MICROCHIP

C

- AVR microcontrollers are programmed in C
  - Or Assembly for those who like a challenge
  - Or other languages someone added support for
- C is a hardware-near programming language from the 1970s
- Each statement must end with a semicolon;
- The program always starts in the main() function
- Must be compiled for the AVR architecture

C

```c
#define F_CPU 3333333UL

#include <avr/io.h>
#include <util/delay.h>

#define LED0PORT PORTF
#define LED0_bp PIN5_bp

int main(void)
{
    LED0PORT.DIR |= (1 << LED0_bp);

    while (1)
    {
        LED0PORT.OUT ^= (1 << LED0_bp);

        _delay_ms(500);
    }
}
```

Bits & Bytes

- 1 bit = 0 or 1
- 8 bits == 1 byte

Representation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Sum
Binary	1	1	1	1	1	1	1	1	0b11111111
Decimal	128	64	32	16	8	4	2	1	255
Hexadecimal	F				F				0xFF

Bits & Bytes

Bitwise OR: (A | B)

Truth table:

Input		Output
0	0	0
0	1	1
1	0	1
1	1	1

Example:

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Input A	1	0	0	0	1	0	1	0
Input B	1	0	1	1	0	0	1	0
Output	1	0	**1**	**1**	**1**	0	1	0

Bitwise AND: (A & B)

Truth table:

Input		Output
0	0	0
0	1	0
1	0	0
1	1	1

Example:

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Input A	1	0	0	0	1	0	1	0
Input B	1	0	1	1	0	0	1	0
Output	1	0	**0**	**0**	**0**	0	1	0

Bits & Bytes

Bitwise NOT: (~A)

Truth table:

Input	Output
0	1
1	0

Example:

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Input A	1	0	1	1	1	0	1	0
Output	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**1**

Bitwise XOR: (A ^ B) - Exclusive OR

Truth table:

Input		Output
0	0	0
0	1	1
1	0	1
1	1	0

Example:

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Input A	1	0	1	1	1	0	1	0
Input B	1	0	0	0	1	0	1	0
Output	**0**	0	**1**	**1**	**0**	0	**0**	0

Bits & Bytes

Left shifting: Multiply with 2^n

Operation	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	bit -1	bit -2	Decimal
Start value			**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**			3
Left shift 1													
Left shift 2													
Left shift 3													
Left shift 4													
Left shift 5													
Left shift 6													
Left shift 7													
Left shift 8													

Right shifting: Divide by 2^n

Operation	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	bit -1	bit -2	Decimal
Start value			1	0	1	0	0	0	0	0			160
Right shift 1													
Right shift 2													
Right shift 3													
Right shift 4													
Right shift 5													
Right shift 6													
Right shift 7													
Right shift 8													

Bits & Bytes

Left shifting: Multiply with 2^n

Operation	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	bit -1	bit -2	Decimal
Start value			0	0	0	0	0	0	1	1			3
Left shift 1			**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**			6
Left shift 2													
Left shift 3													
Left shift 4													
Left shift 5													
Left shift 6													
Left shift 7													
Left shift 8													

Right shifting: Divide by 2^n

Operation	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	bit -1	bit -2	Decimal
Start value			1	0	1	0	0	0	0	0			160
Right shift 1													
Right shift 2													
Right shift 3													
Right shift 4													
Right shift 5													
Right shift 6													
Right shift 7													
Right shift 8													

Bits & Bytes

Left shifting: Multiply with 2^n

Operation	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	bit -1	bit -2	Decimal
Start value			0	0	0	0	0	0	1	1			3
Left shift 1			0	0	0	0	0	1	1	0			6
Left shift 2			**0**	**0**	**0**	**0**	**1**	**1**	0	**0**			12
Left shift 3													
Left shift 4													
Left shift 5													
Left shift 6													
Left shift 7													
Left shift 8													

Right shifting: Divide by 2^n

Operation	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	bit -1	bit -2	Decimal
Start value			1	0	1	0	0	0	0	0			160
Right shift 1													
Right shift 2													
Right shift 3													
Right shift 4													
Right shift 5													
Right shift 6													
Right shift 7													
Right shift 8													

Bits & Bytes

Left shifting: Multiply with 2^n

Operation	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	bit -1	bit -2	Decimal
Start value			0	0	0	0	0	0	1	1			3
Left shift 1			0	0	0	0	0	1	1	0			6
Left shift 2			0	0	0	0	1	1	0	0			12
Left shift 3			0	0	0	1	1	0	0	0			24
Left shift 4			0	0	1	1	0	0	0	0			48
Left shift 5			0	1	1	0	0	0	0	0			96
Left shift 6			**1**	**1**	0	0	0	0	0	**0**			192
Left shift 7													
Left shift 8													

Right shifting: Divide by 2^n

Operation	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	bit -1	bit -2	Decimal
Start value			1	0	1	0	0	0	0	0			160
Right shift 1													
Right shift 2													
Right shift 3													
Right shift 4													
Right shift 5													
Right shift 6													
Right shift 7													
Right shift 8													

Bits & Bytes

Left shifting: Multiply with 2^n

Operation	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	bit -1	bit -2	Decimal
Start value			0	0	0	0	0	0	1	1			3
Left shift 1			0	0	0	0	0	1	1	0			6
Left shift 2			0	0	0	0	1	1	0	0			12
Left shift 3			0	0	0	1	1	0	0	0			24
Left shift 4			0	0	1	1	0	0	0	0			48
Left shift 5			0	1	1	0	0	0	0	0			96
Left shift 6			1	1	0	0	0	0	0	0			192
Left shift 7		**1**	**1**	0	0	0	0	0	0	**0**			128
Left shift 8													

Right shifting: Divide by 2^n

Operation	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	bit -1	bit -2	Decimal
Start value			1	0	1	0	0	0	0	0			160
Right shift 1													
Right shift 2													
Right shift 3													
Right shift 4													
Right shift 5													
Right shift 6													
Right shift 7													
Right shift 8													

Bits & Bytes

Left shifting: Multiply with 2^n

Operation	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	bit -1	bit -2	Decimal
Start value			0	0	0	0	0	0	1	1			3
Left shift 1			0	0	0	0	0	1	1	0			6
Left shift 2			0	0	0	0	1	1	0	0			12
Left shift 3			0	0	0	1	1	0	0	0			24
Left shift 4			0	0	1	1	0	0	0	0			48
Left shift 5			0	1	1	0	0	0	0	0			96
Left shift 6			1	1	0	0	0	0	0	0			192
Left shift 7		1	1	0	0	0	0	0	0	0			128
Left shift 8	1	1	0	0	0	0	0	0	0	0			0

Right shifting: Divide by 2^n

Operation	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	bit -1	bit -2	Decimal
Start value			1	0	1	0	0	0	0	0			160
Right shift 1													
Right shift 2													
Right shift 3													
Right shift 4													
Right shift 5													
Right shift 6													
Right shift 7													
Right shift 8													

Bits & Bytes

Left shifting: Multiply with 2^n

Operation	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	bit -1	bit -2	Decimal
Start value			0	0	0	0	0	0	1	1			3
Left shift 1			0	0	0	0	0	1	1	0			6
Left shift 2			0	0	0	0	1	1	0	0			12
Left shift 3			0	0	0	1	1	0	0	0			24
Left shift 4			0	0	1	1	0	0	0	0			48
Left shift 5			0	1	1	0	0	0	0	0			96
Left shift 6			1	1	0	0	0	0	0	0			192
Left shift 7		1	1	0	0	0	0	0	0	0			128
Left shift 8	1	1	0	0	0	0	0	0	0	0			0

Right shifting: Divide by 2^n

Operation	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	bit -1	bit -2	Decimal
Start value			1	0	1	0	0	0	0	0			160
Right shift 1			0	1	0	1	0	0	0	0			80
Right shift 2			0	0	1	0	1	0	0	0			40
Right shift 3			0	0	0	1	0	1	0	0			20
Right shift 4			0	0	0	0	1	0	1	0			10
Right shift 5			0	0	0	0	0	1	0	1			5
Right shift 6			0	0	0	0	0	0	1	0	1		2
Right shift 7			0	0	0	0	0	0	0	1	0	1	1
Right shift 8			0	0	0	0	0	0	0	0	1	0	0

Bits & Bytes

Left shifting: Multiply with 2^n

Operation	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	bit -1	bit -2	Decimal
(A << 0)			0	0	0	0	0	0	1	1			3
(A << 1)			0	0	0	0	0	1	1	0			6
(A << 2)			0	0	0	0	1	1	0	0			12
(A << 3)			0	0	0	1	1	0	0	0			24
(A << 4)			0	0	1	1	0	0	0	0			48
(A << 5)			0	1	1	0	0	0	0	0			96
(A << 6)			1	1	0	0	0	0	0	0			192
(A << 7)			1	0	0	0	0	0	0	0			128
(A << 8)			0	0	0	0	0	0	0	0			0

Right shifting: Divide by 2^n

Operation	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	bit -1	bit -2	Decimal
(A >> 0)			1	0	1	0	0	0	0	0			160
(A >> 1)			0	1	0	1	0	0	0	0			80
(A >> 2)			0	0	1	0	1	0	0	0			40
(A >> 3)			0	0	0	1	0	1	0	0			20
(A >> 4)			0	0	0	0	1	0	1	0			10
(A >> 5)			0	0	0	0	0	1	0	1			5
(A >> 6)			0	0	0	0	0	0	1	0			2
(A >> 7)			0	0	0	0	0	0	0	1			1
(A >> 8)			0	0	0	0	0	0	0	0			0

Bitmask

- **A defined left-shifted bit or bits**
- **Think bit positions, not values**
  - `(1 << 3)` is read as «**Write ‘1’ to bit 3**»
  - You configure a bit, not writing a number
- **Bitmasks can be combined**
  - `(1 << 3) | (1 << 6)`
    is «**Write ‘1’ to bits 3 and 6**»
- **Zero-indexed**
  - `(1 << 0)` the first bit is bit `0`.

Bits & Bytes

Logical operators != bitwise operators

Logical NOT: (!A)

Truth table:

Input	Output
0	1
<0	0
>0	0

Logical OR: (A || B)

Truth table:

Input		Output
0	0	0
0	≠0	1
≠0	0	1
≠0	≠0	1

Logical AND: (A && B)

Truth table:

Input		Output
0	0	0
0	≠0	0
≠0	0	0
≠0	≠0	1

Bits & Bytes

Logical operators != bitwise operators

Relational operators

Operator	Meaning of Operator	Example
==	Equal to	5 == 3 evaluates to 0
>	Greater than	5 > 3 evaluates to 1
<	Less than	5 < 3 evaluates to 0
!=	Not equal to	5 != 3 evaluates to 1
>=	Greater than or equal to	5 >= 3 evaluates to 1
<=	Less than or equal to	5 <= 3 evaluates to 0

0 == «False» **1 == «True»**

Bit-fu 1st Dan

Syntax	Operation
A + B	Add A and B
A - B	Substract B from A
A / B	Divide A by B
A * B	Multiply A with B
~A	Bitwise NOT
A & B	Bitwise AND
A | B	Bitwise OR
A ^ B	Bitwise XOR
A >> n	Right shift A bitwise n bits
A << n	Left shift A bitwise n bits
A || B	Logical OR
A && B	Logical AND
!A	Logical NOT
A == B	Equal to
A > B	Greater than
A < B	Less than
A != B	Not equal to
A >= B	Greater than or equal to
A <= B	Less than or equal to

Registers

- Special location in RAM
- Full of bits linked to module functions
- Explained in data sheet and manual

15.5.12 Pin n Control

Name: PINCTRL
Offset: 0x10 + n*0x01 [n=0..7]
Reset: 0x00
Property: -

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

Registers

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

Bit 7 – INVEN Inverted I/O Enable

Value	Description
0	Input and output values are not inverted
1	Input and output values are inverted

Bit 3 – PULLUPEN Pullup Enable

Value	Description
0	Pullup disabled for pin n
1	Pullup enabled for pin n

Bits 2:0 – ISC[2:0] Input/Sense Configuration

These bits configure the input and sense configuration of pin n. The sense configuration determines how a port interrupt can be triggered. If the input buffer is disabled, the input cannot be read in the IN register.

Value	Name	Description
0x0	INTDISABLE	Interrupt disabled but input buffer enabled
0x1	BOTHEDGES	Interrupt enabled with sense on both edges
0x2	RISING	Interrupt enabled with sense on rising edge
0x3	FALLING	Interrupt enabled with sense on falling edge
0x4	INPUT_DISABLE	Interrupt and digital input buffer disabled
0x5	LEVEL	Interrupt enabled with sense on low level
other	-	Reserved

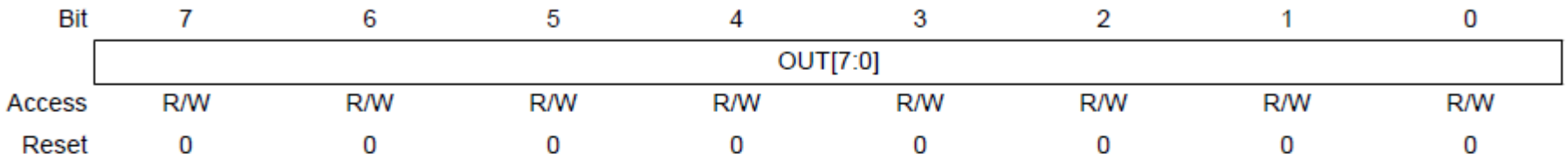
Registers

- **Header file contains definitions and names**
  - #include <avr/io.h>

```
/* I/O Ports */
typedef struct PORT_struct
{
    register8_t DIR; /* Data Direction */
    register8_t DIRSET; /* Data Direction Set */
    register8_t DIRCLR; /* Data Direction Clear */
    register8_t DIRTGL; /* Data Direction Toggle */
    register8_t OUT; /* Output Value */
    register8_t OUTSET; /* Output Value Set */
    register8_t OUTCLR; /* Output Value Clear */
    register8_t OUTTGL; /* Output Value Toggle */
    register8_t IN; /* Input Value */
    register8_t INTFLAGS; /* Interrupt Flags */
    register8_t PORTCTRL; /* Port Control */
    ...
} PORT_t;

#define PORTA (*(PORT_t *) 0x0400) /* I/O Ports */
```

Registers



Bit	7	6	5	4	3	2	1	0
	OUT[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 – OUT[7:0] Output Value

This bit field defines the data output value for the individual pins of the port.
If OUT[n] is written to '1', pin n is driven high.

If OUT[n] is written to '0', pin n is driven low.

In order to have any effect, the pin direction must be configured as output.

```
PORTA.DIR = 0xff; // Set all PORTA pins as output

PORTA.OUT = 0; // Set all PORTA pins low

PORTA.OUT = 12; // You can do this, but only if you want to annoy people
                // Better to do it like this:
PORTA.OUT = (1 << PIN3_bp) | (1 << PIN2_bp); // Set pins PA2 and PA3 high

PORTA.PIN1CTRL = (1 << PORT_INVEN_bp); // Invert pin functionality of PA1
```

Registers

- **Think bit positions, not values**
  - You configure bits, not writing a number

- **Bad examples:**

```
PORTA.OUT = 8;
PORTA.OUT = 0b00001000;
```

- **Good examples:**

```
PORTA.OUT = (1 << 3);
PORTA.OUT = (1 << PIN3_bp);
```

Bits & Registers

Normal write to register: (A = B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A	1	1	1	0	0	0	0	0	0xE0
New value B	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	0x22
Updated register A									

Set bit(s) in register: (A |= B), long form: (A = A | B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A									
New value B									
Updated register A									

Bits & Registers

Normal write to register: (A = B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A	x	x	x	x	x	x	x	x	
New value B	0	0	**1**	0	0	0	**1**	0	0x22
Updated register A	0	0	**1**	0	0	0	**1**	0	0x22

Set bit(s) in register: (A |= B), long form: (A = A | B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A									
New value B									
Updated register A									

Bits & Registers

Normal write to register: (A = B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A	x	x	x	x	x	x	x	x	
New value B	0	0	**1**	0	0	0	**1**	0	0x22
Updated register A	0	0	**1**	0	0	0	**1**	0	0x22

Set bit(s) in register: (A |= B), long form: (A = A | B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A	1	0	1	0	0	1	0	1	0xA5
New value B	0	0	**1**	0	0	0	**1**	0	0x22
Updated register A									

Bits & Registers

Normal write to register: (A = B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A	x	x	x	x	x	x	x	x	
New value B	0	0	**1**	0	0	0	**1**	0	0x22
Updated register A	0	0	**1**	0	0	0	**1**	0	0x22

Set bit(s) in register: (A |= B), long form: (A = A | B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A	1	0	1	0	0	1	0	1	0xA5
New value B	0	0	**1**	0	0	0	**1**	0	0x22
Updated register A	x	x	**1**	x	x	x	**1**	x	

Bits & Registers

Normal write to register: (A = B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A	x	x	x	x	x	x	x	x	
New value B	0	0	**1**	0	0	0	**1**	0	0x22
Updated register A	0	0	**1**	0	0	0	**1**	0	0x22

Set bit(s) in register: (A |= B), long form: (A = A | B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A	1	0	1	0	0	1	0	1	0xA5
New value B	0	0	1	0	0	0	1	0	0x22
Updated register A	1	0	1	0	0	1	**1**	1	0xA7

Bits & Registers

Clear bit(s) in register: (A &= ~B), long form: (A = A & ~B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A	1	0	1	0	0	1	0	1	0xA5
New value B	0	0	1	0	0	0	1	0	0x22
Inverted value B									
Updated register A									

Toggle bit(s) in register: (A ^= B), long form: (A = A ^ B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A									
New value B									
Updated register A									

Bits & Registers

Clear bit(s) in register: (A &= ~B), long form: (A = A & ~B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A	1	0	1	0	0	1	0	1	0xA5
New value B	0	0	1	0	0	0	1	0	0x22
Inverted value B	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**1**	0xDD
Updated register A									

Toggle bit(s) in register: (A ^= B), long form: (A = A ^ B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A									
New value B									
Updated register A									

Bits & Registers

Clear bit(s) in register: (A &= ~B), long form: (A = A & ~B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A	1	0	1	0	0	1	0	1	0xA5
New value B	0	0	1	0	0	0	1	0	0x22
Inverted value B	1	1	**0**	1	1	1	**0**	1	0xDD
Updated register A	x	x	**0**	x	x	x	**0**	x	

Toggle bit(s) in register: (A ^= B), long form: (A = A ^ B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A									
New value B									
Updated register A									

Bits & Registers

Clear bit(s) in register: (A &= ~B), long form: (A = A & ~B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A	1	0	1	0	0	1	0	1	0xA5
New value B	0	0	1	0	0	0	1	0	0x22
Inverted value B	1	1	**0**	1	1	1	0	1	0xDD
Updated register A	1	0	**0**	0	0	1	0	1	0x85

Toggle bit(s) in register: (A ^= B), long form: (A = A ^ B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A									
New value B									
Updated register A									

Bits & Registers

Clear bit(s) in register: (A &= ~B), long form: (A = A & ~B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A	1	0	1	0	0	1	0	1	0xA5
New value B	0	0	1	0	0	0	1	0	0x22
Inverted value B	1	1	**0**	1	1	1	0	1	0xDD
Updated register A	1	0	**0**	0	0	1	0	1	0x85

Toggle bit(s) in register: (A ^= B), long form: (A = A ^ B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A	x	x	x	x	x	x	x	x	
New value B	0	0	1	0	0	0	1	0	0x22
Updated register A									

Bits & Registers

Clear bit(s) in register: (A &= ~B), long form: (A = A & ~B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A	1	0	1	0	0	1	0	1	0xA5
New value B	0	0	1	0	0	0	1	0	0x22
Inverted value B	1	1	**0**	1	1	1	0	1	0xDD
Updated register A	1	0	**0**	0	0	1	0	1	0x85

Toggle bit(s) in register: (A ^= B), long form: (A = A ^ B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A	x	x	x	x	x	x	x	x	
New value B	0	0	1	0	0	0	1	0	0x22
Updated register A	x	x	**~x**	x	x	x	**~x**	x	

If a bit in B is ‘1’, the corresponding bit in A will be inverted (toggled).

Bits & Registers

Clear bit(s) in register: (A &= ~B), long form: (A = A & ~B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A	1	0	1	0	0	1	0	1	0xA5
New value B	0	0	1	0	0	0	1	0	0x22
Inverted value B	1	1	**0**	1	1	1	0	1	0xDD
Updated register A	1	0	**0**	0	0	1	0	1	0x85

Toggle bit(s) in register: (A ^= B), long form: (A = A ^ B)

Operation	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
Register value A	1	0	**1**	0	0	1	**0**	1	0xA5
New value B	0	0	**1**	0	0	0	**1**	0	0x22
Updated register A	1	0	**0**	0	0	1	**1**	1	0x87

If a bit in B is ‘1’, the corresponding bit in A will be inverted (toggled).

Bit-fu 2nd Dan

Syntax	Long Form	Operation
A |= B	A = A | B	A = Bitwise OR of A and B
A &= B	A = A & B	A = Bitwise AND of A and B
A ^= B	A = A ^ B	A = Bitwise XOR of A and B
A &= ~B	A = A & (~B)	A = Bitwise AND of A and NOT B
A += B	A = A + B	A = Add A and B
A -= B	A = A - B	A = Subtract B from A
A *= B	A = A * B	A = Multiply A with B
A /= B	A = A / B	A = Divide A with B
A >>= n	A = A >> n	A = Right shift A n bits
A <<= n	A = A << n	A = Left shift A n bits
A++	A = A + 1	A = Add 1 to A (Increment)
A--	A = A - 1	A = Subtract 1 from A (Decrement)

Registers

- C example

```
// Start out with pins PA2 and PA3 high
PORTA.OUT = (1 << PIN3_bp) | (1 << PIN2_bp);

// Set PA5 high without changing PA2 and PA3
PORTA.OUT |= (1 << PIN5_bp);

// Set PA2 low without changing PA5 and PA3
PORTA.OUT &= ~(1 << PIN2_bp);
```

Registers

- C example

```
// Start out 0b00001100 PA3 high
PORTA.OUT = (                << PIN2_bp);

// Set PA5 high without changing PA2 and PA3
PORTA.OUT |= (1 << PIN5_bp);

// Set PA2 low without changing PA5 and PA3
PORTA.OUT &= ~(1 << PIN2_bp);
```

Registers

- C example

```
// Start out with pins PA2 and PA3 high
PORTA.OUT = (1 << PIN3_bp) | (1 << PIN2_bp);

// Set PA5 hi                g PA2 and PA3
PORTA.OUT |=
```

0b00101100

```
// Set PA2 low without changing PA5 and PA3
PORTA.OUT &= ~(1 << PIN2_bp);
```

Registers

- C example

```
// Start out with pins PA2 and PA3 high
PORTA.OUT = (1 << PIN3_bp) | (1 << PIN2_bp);

// Set PA5 high without changing PA2 and PA3
PORTA.OUT |= (1 << PIN5_bp);

// Set PA2 lo                                PA5 and PA3
PORTA.OUT &= 0b00101000
```

Registers

- **Check if bit(s) are set**

```
if (PORTA.IN & (1 << PIN3_bp) ) {
  // PA3 is high, do something
}
```

- **In C, everything except 0 is «True»**

```
if (PORTA.IN & (1 << PIN3_bp) != 0)
{
  // PA3 is high, do something
}
```

Registers

- **Check if bit(s) are cleared**

```
if (!(PORTA.IN & (1 << PIN3_bp))) {
  // PA3 is low, do something
}
```

- **Can also be written as compare with 0**

```
if (PORTA.IN & (1 << PIN3_bp) == 0)
{
  // PA3 is low, do something
}
```

Registers

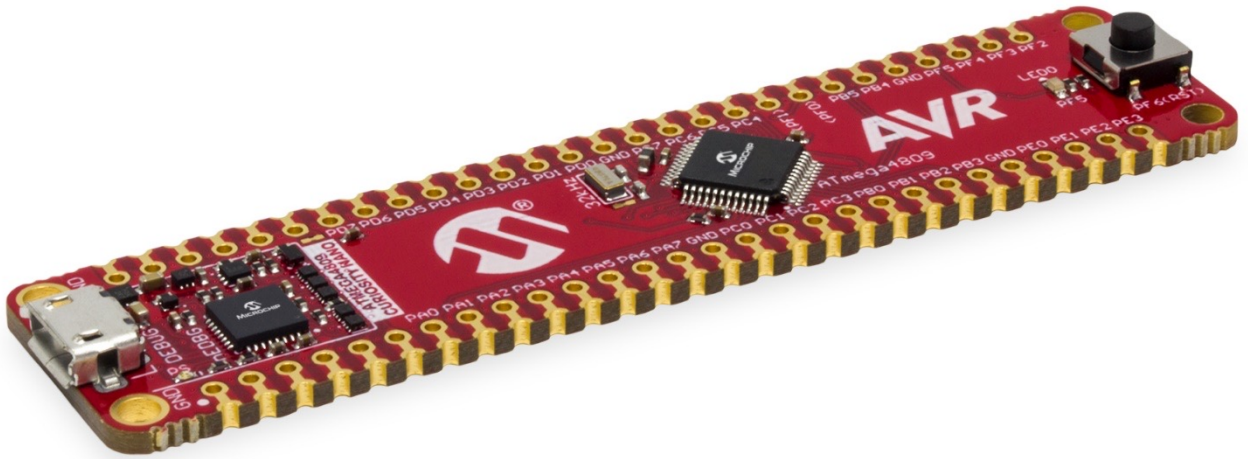
- **Store the current register state**

```
uint8_t my_variable = PORTA.IN;
```

- **Check if bit was set, may have changed**

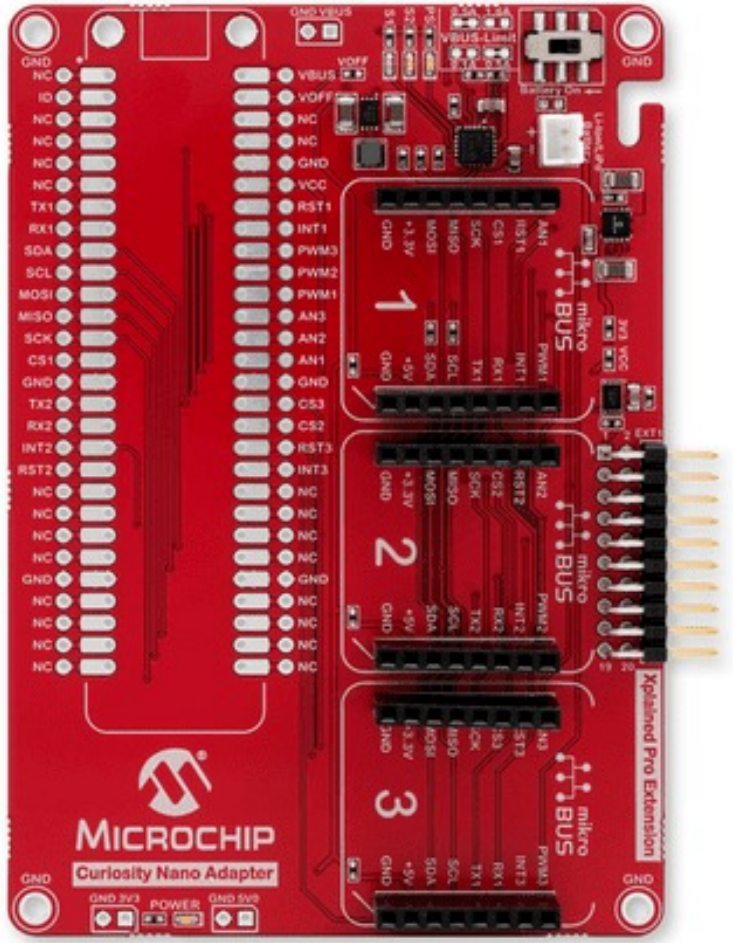
```
if (my_variable & (1 << PIN3_bp)) {
// PA3 was high, do something
}
```

ATmega4809 Curiosity Nano



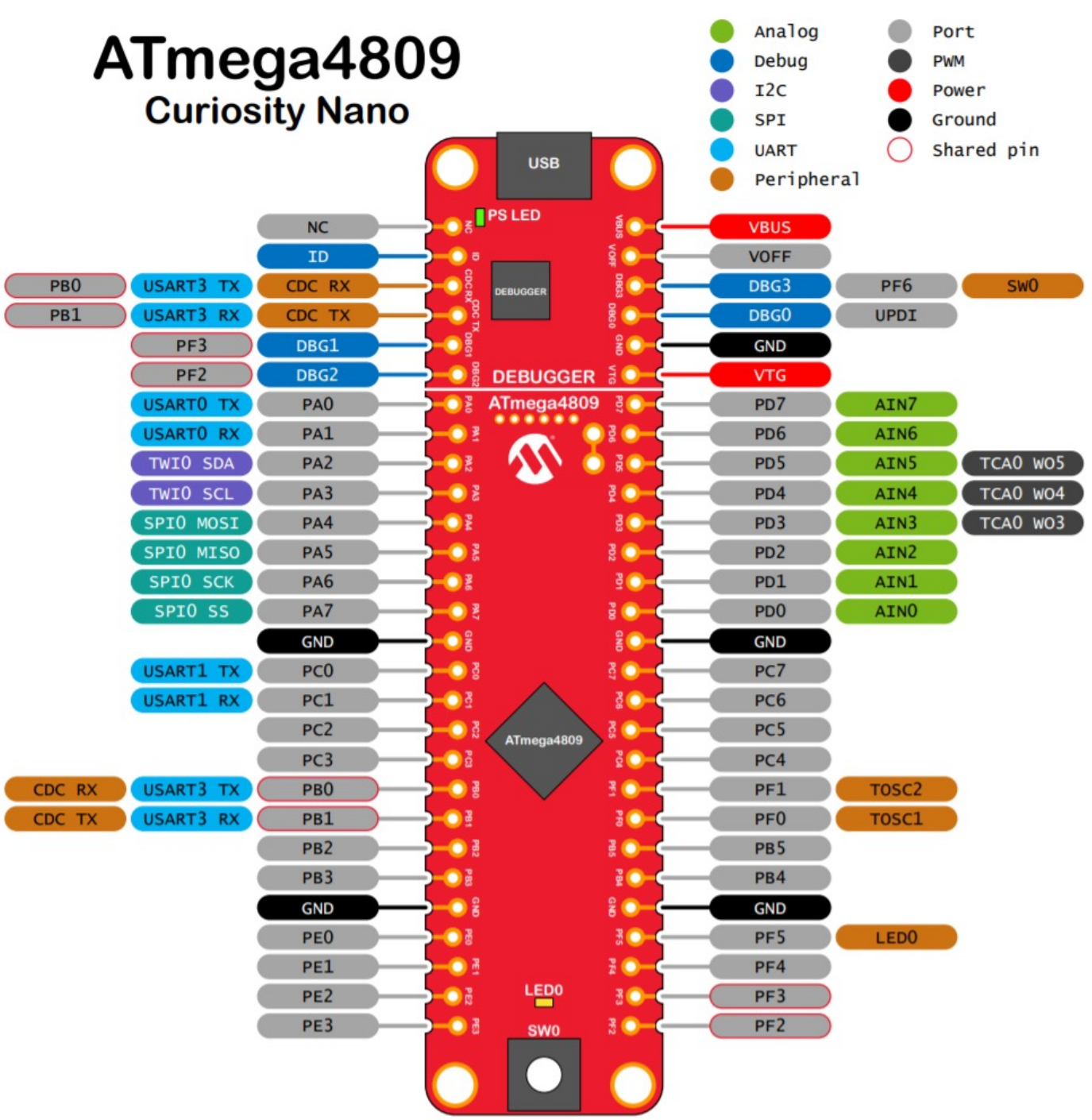
MICROCHIP

Curiosity Nano Adapter



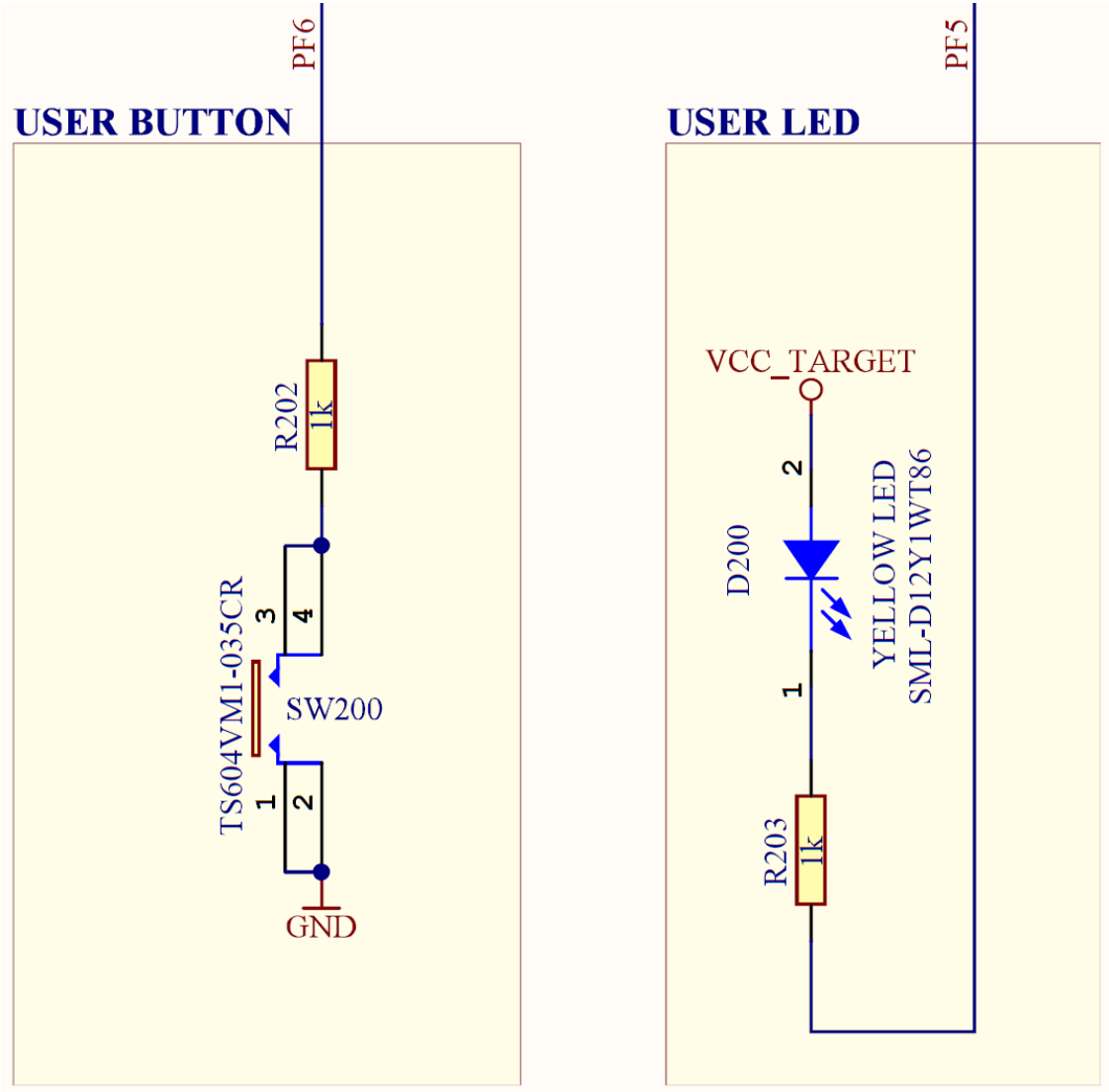
MICROCHIP

Hardware User Guide



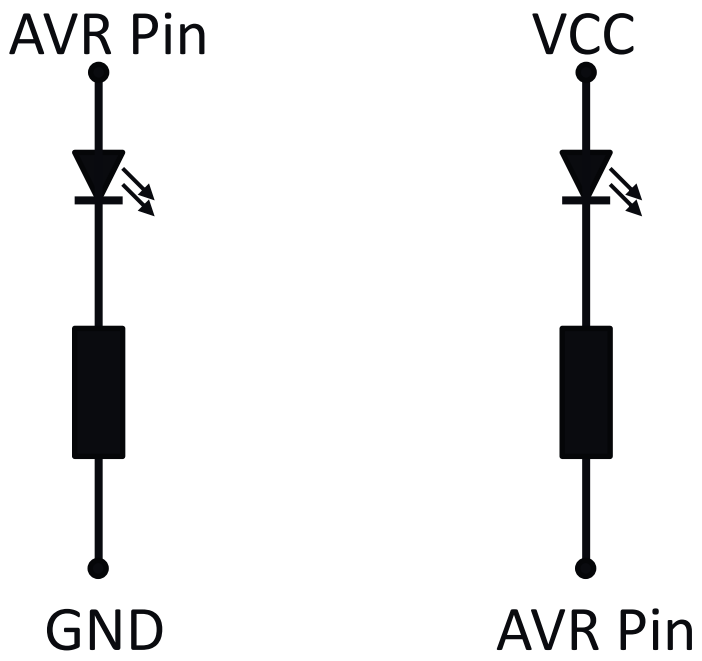
MICROCHIP

Hardware User Guide



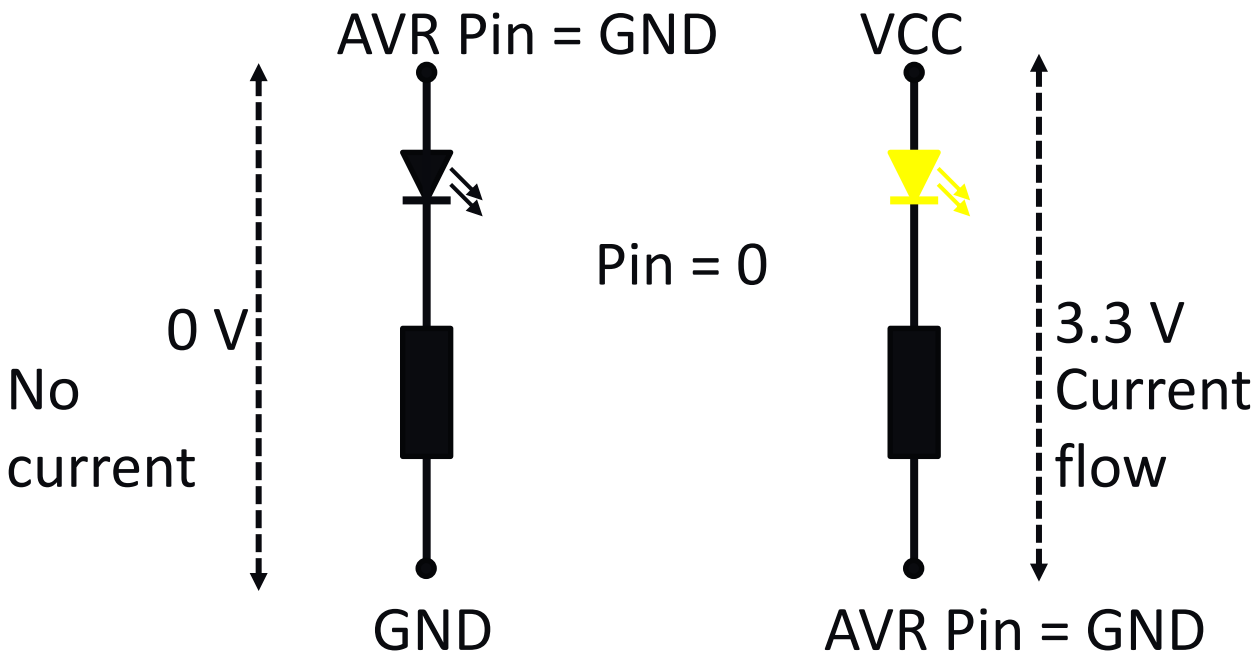
AVR I/O - Output

- **Controlled by the MCU**
  - 0 == GND, 1 == VCC
  - Example: LEDs
    - «Active High» or «Active Low»



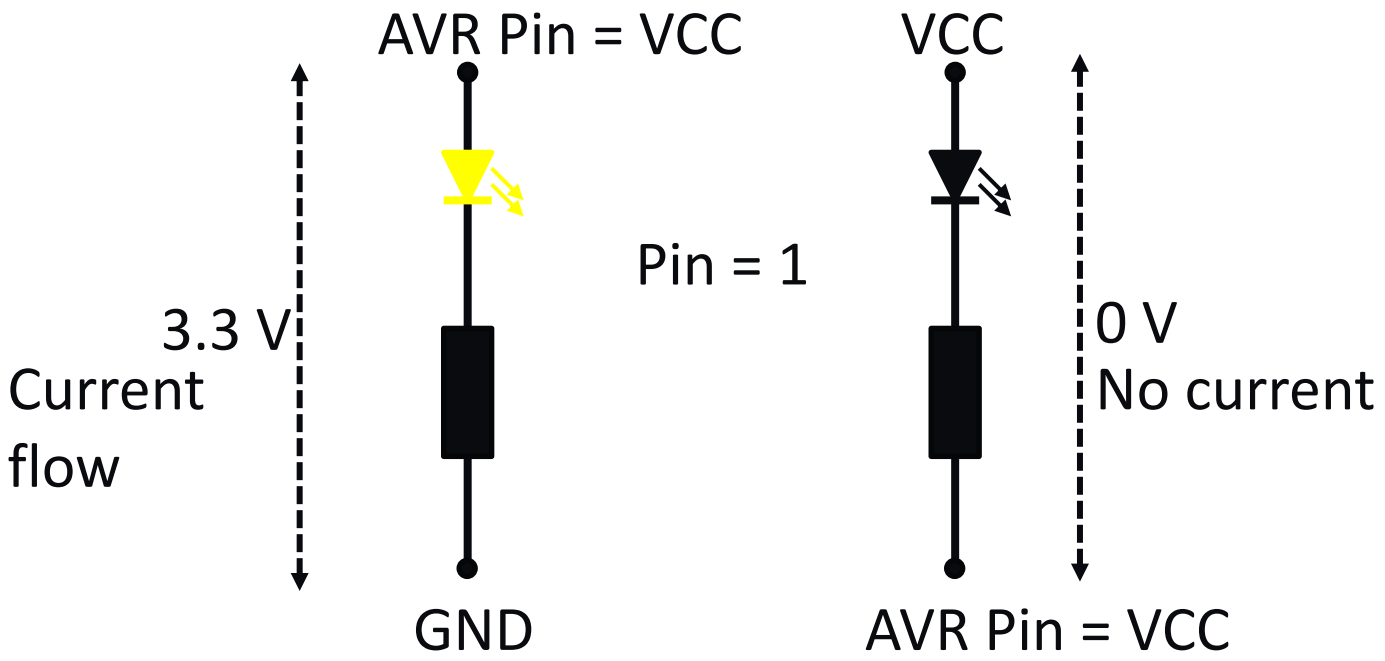
AVR I/O - Output

- **Controlled by the MCU**
  - Pin = 0 == GND
  - Example: LEDs
    - «Active High» or «Active Low»



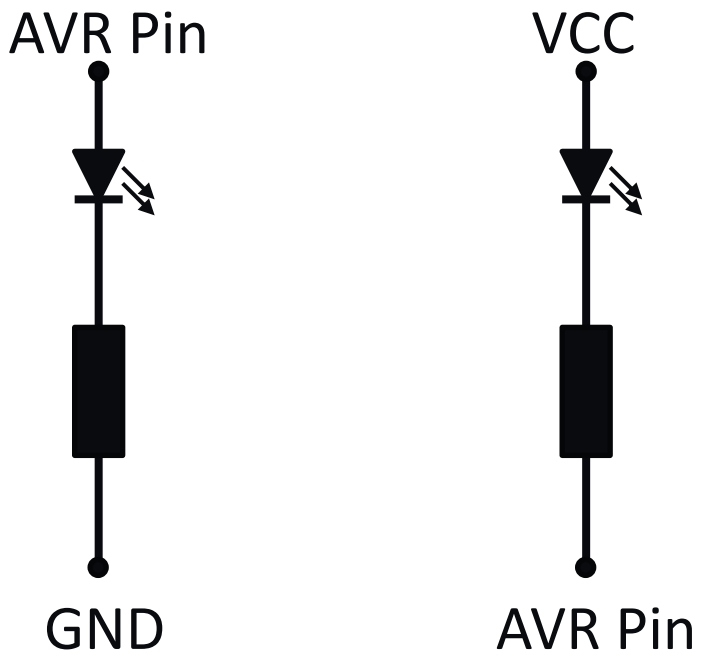
AVR I/O - Output

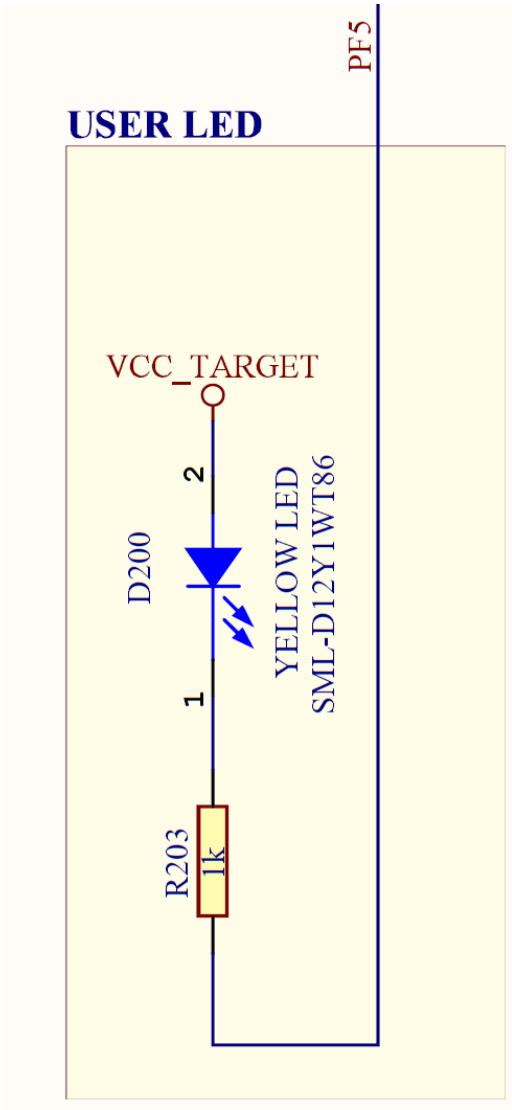
- **Controlled by the MCU**
  - Pin = 1 == VCC
  - Example: LEDs
    - «Active High» or «Active Low»



AVR I/O - Output

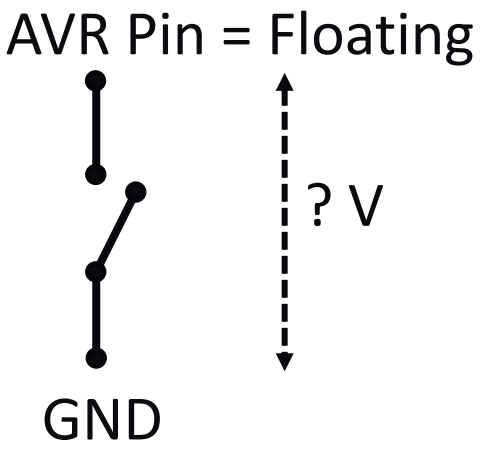
- **Controlled by the MCU**
  - 0 == GND, 1 == VCC
  - Example: LEDs
    - «Active High» or «Active Low»





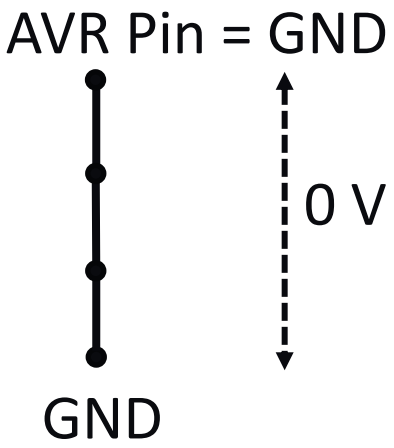
AVR I/O - Input

- **Controlled externally**
  - Floating - Undefined state when not connected
  - 0 ~= GND, 1 ~= VCC
  - The pin voltage needs to be «pulled» to 0 or 1



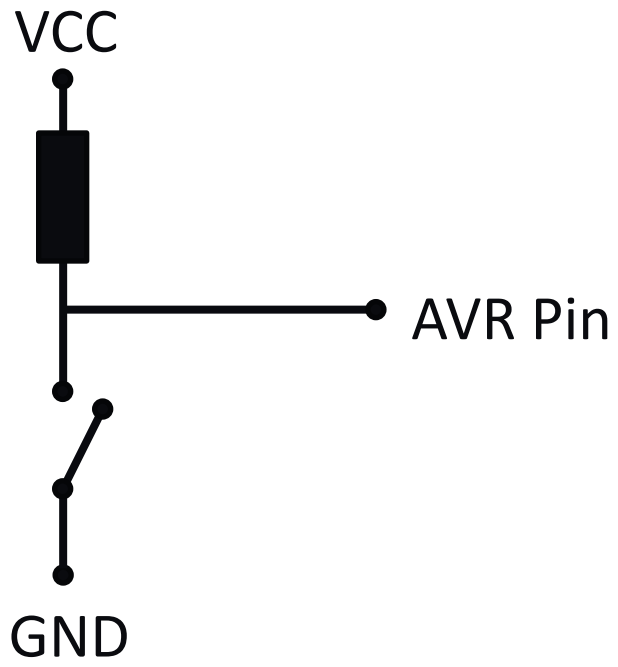
AVR I/O - Input

- **Controlled externally**
  - Floating - Undefined state when not connected
  - 0 ~= GND, 1 ~= VCC
  - The pin voltage needs to be «pulled» to 0 or 1



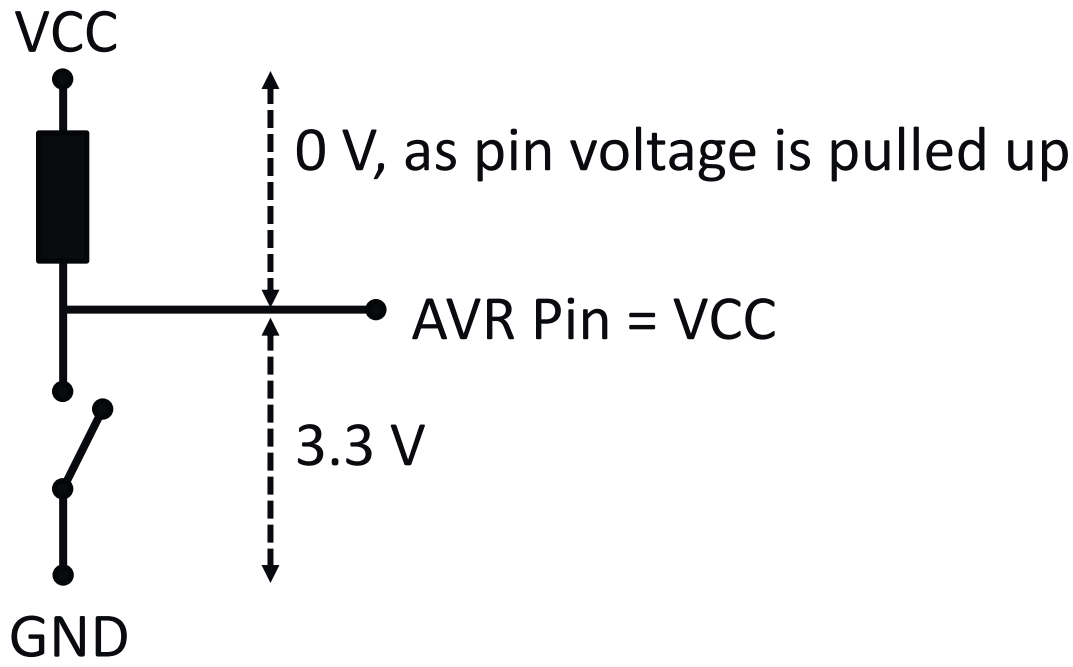
AVR I/O - Input

- **Example: Switches**
  - Pull-up



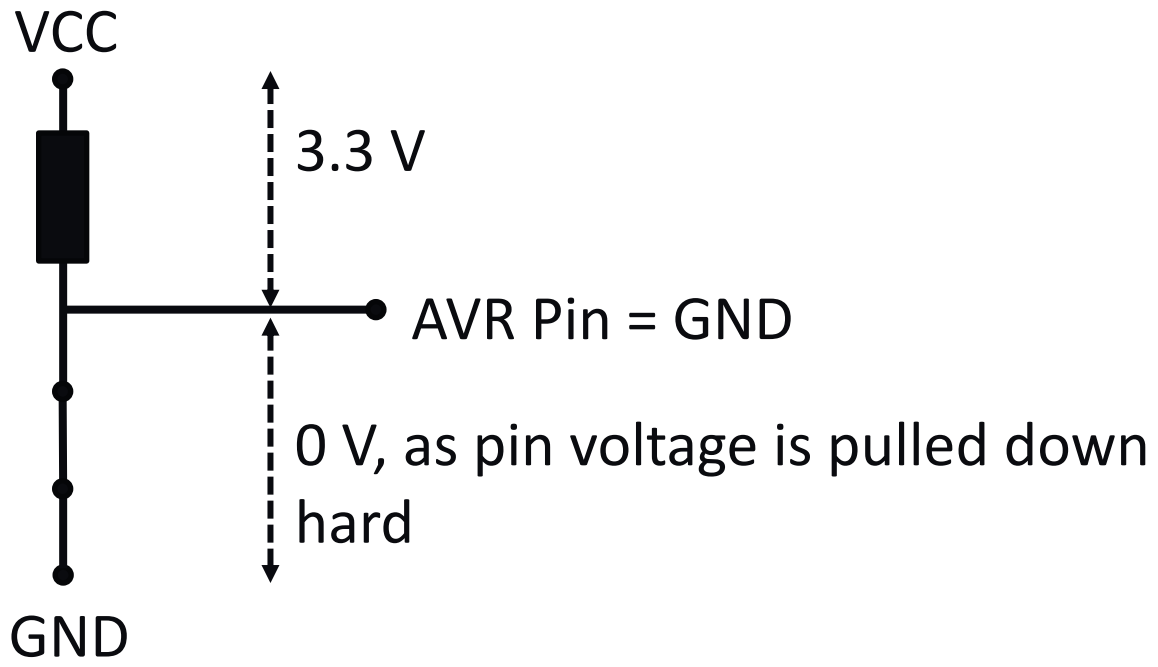
AVR I/O - Input

- **Example: Switches**
  - Pull-up



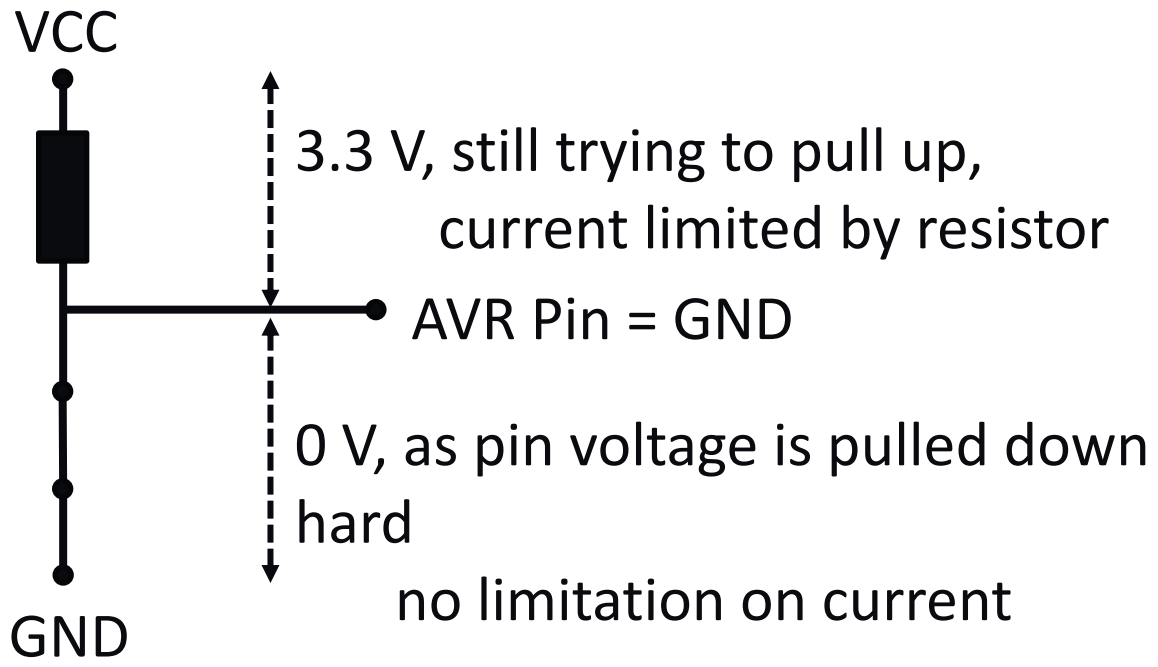
AVR I/O - Input

- **Example: Switches**
  - Pull-up - Active Low



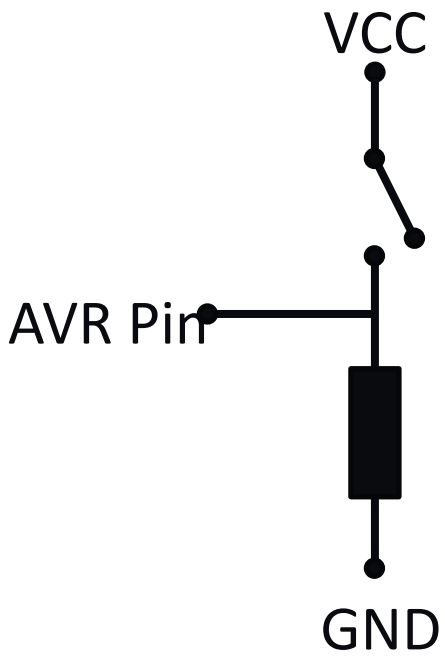
AVR I/O - Input

- **Example: Switches**
  - Pull-up - Active Low



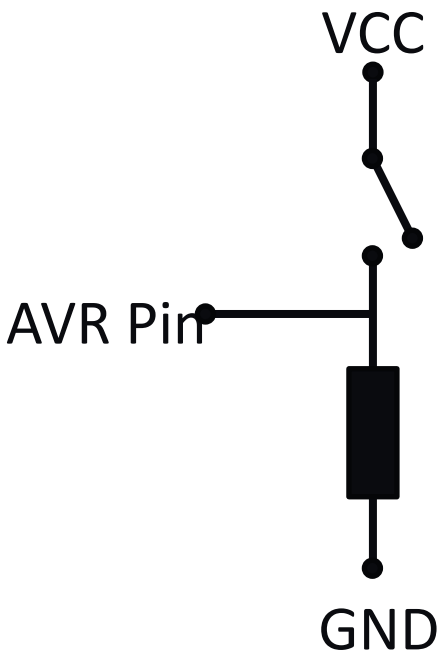
AVR I/O - Input

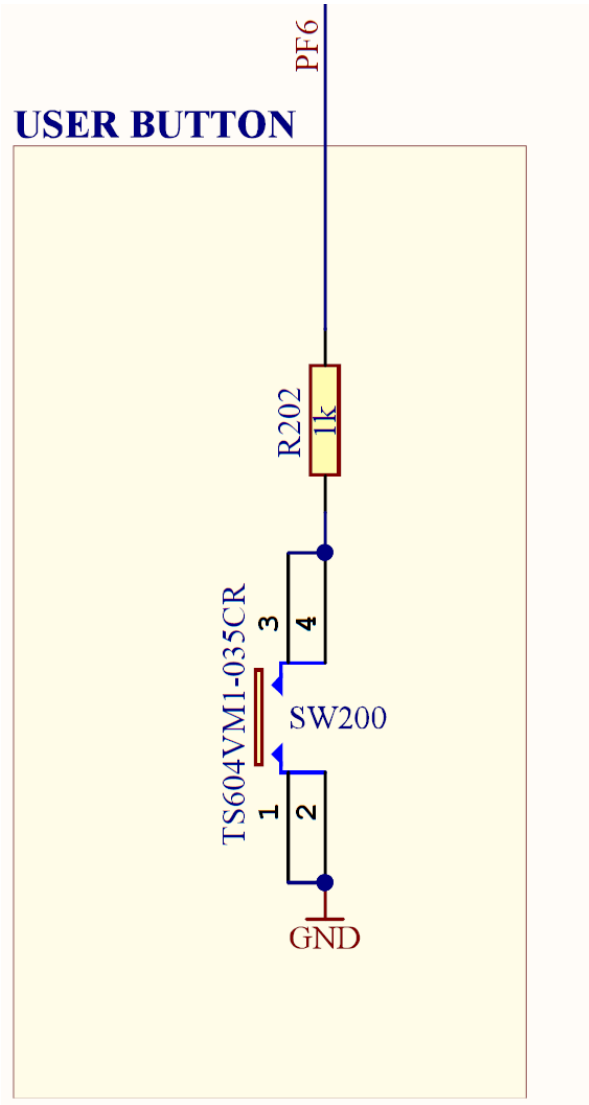
- **Example: Switches**
  - Pull-down - Active High
  - Same same, only inverted



AVR I/O - Input

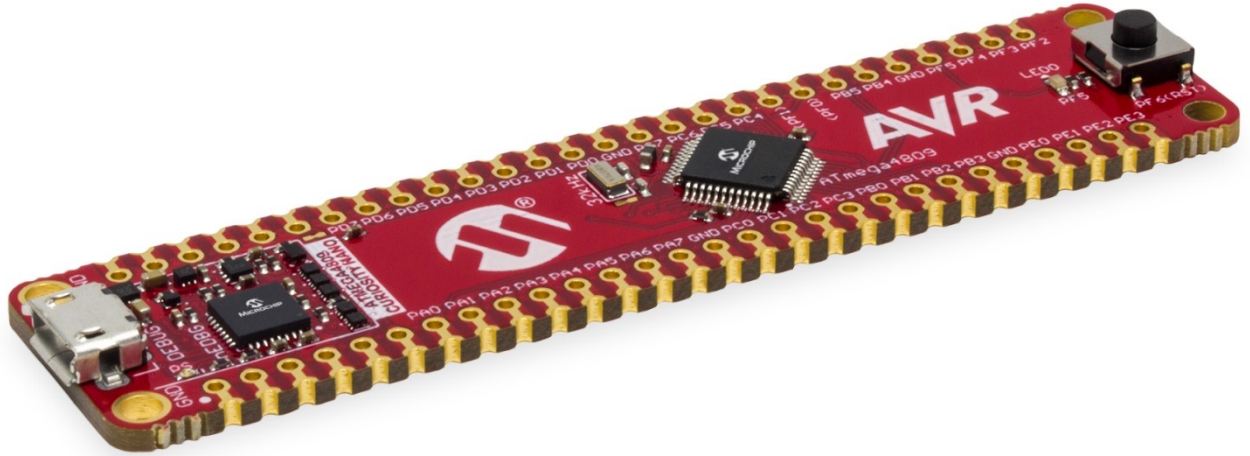
- **Example: Switches**
  - Pull-down - Active High
  - Same same, only inverted





Time to play!

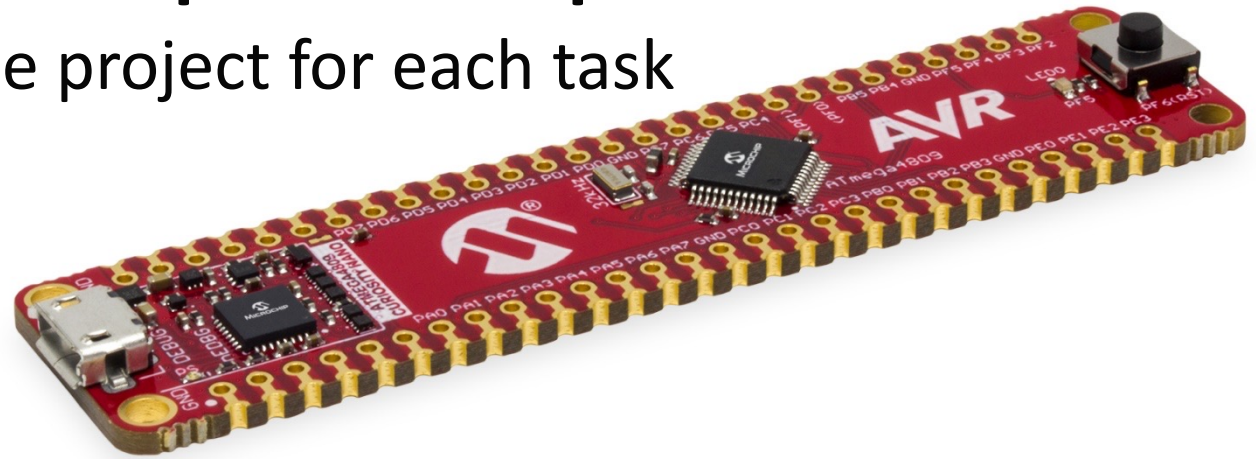
- https://omegav.no/avrkurs
- OV people provide support
- Microchip people support the support



MICROCHIP

Time to play!

- **https://omegav.no/avrkurs**
- **MPLAB X IDE: Open the .X folder**
  - Each task is a configuration in the project
- **Microchip Studio: Open the *.atsln file**
  - One project for each task



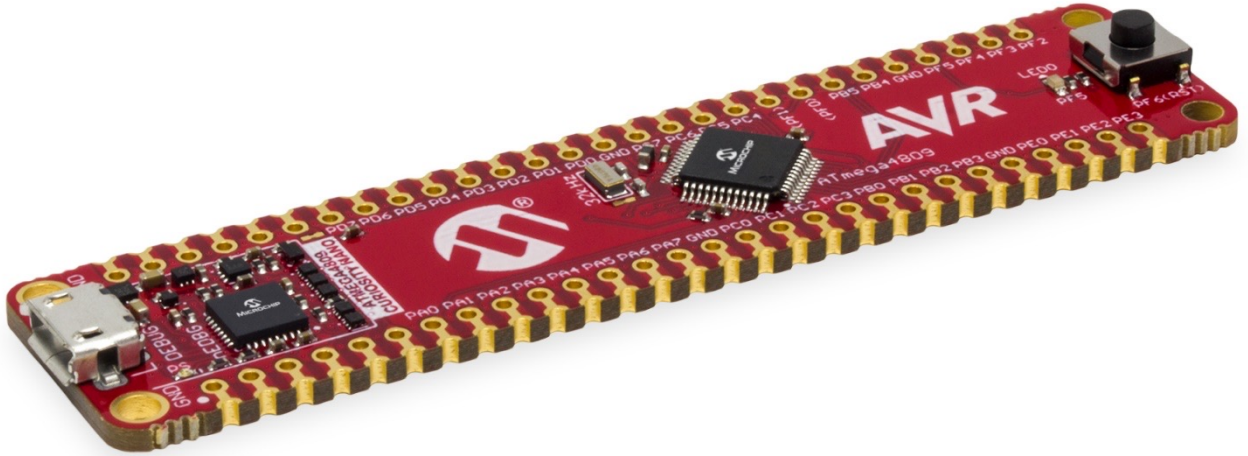
MICROCHIP

The three next slides are to be presented 1-2 hours into the hands-on tasks,

if we are confident the bitshifting has been understood..

Some tips to simplify stuff

- Egil will talk a bit more at 20:00



MICROCHIP

Bitmasks again

- **In the header file:**

```
#define PIN3_bp 3
#define PIN3_bm (1 << PIN3_bp)
```

- **How I would write it:**

```
PORTA.OUT = PIN3_bm;


USART.CTRLB = USART_TXEN_bm;
PORTA.PIN2CTRL = PORT_ISC_LEVEL_gc;
```

Bitmasks again

- **In the header file:**

```
#define PIN3_bp 3
#define PIN3_bm 0x08
```

- **How I would write it:**

```
PORTA.OUT = PIN3_bm;
```

- **Named settings in header file:**

```
USART.CTRLB = USART_TXEN_bm;
PORTA.PIN2CTRL = PORT_ISC_LEVEL_gc;
```

- **(Not valid for ATmega328 and other old devices)**

Making GPIO life easier

- We have added PORT registers that does set, clear and toggling for you!
- DO NOT USE |= OR &= OR ^= HERE

```
PORTA.OUTSET = PIN3_bm;

PORTA.OUTTGL = PIN7_bm;

PORTA.OUTCLR = PIN3_bm;
```

Correct Read-Modify-Write

- The Control C register was configured like this:

```
ADC0.CTRLC = ADC_REFSEL_VDDREF_gc | ADC_PRESC_DIV128_gc;
```

- We want to change reference from VDD to VREFA

```
ADC0.CTRLC &= ~ADC_REFSEL_gm;
ADC0.CTRLC |= ADC_REFSEL_VREFA_gc;
```

- Can be combined into one line

```
ADC0.CTRLC = (ADC0.CTRLC & ~ADC_REFSEL_gm) | ADC_REFSEL_VREFA_gc;
```

- Or just reconfigure all the bitfields

```
ADC0.CTRLC = ADC_REFSEL_VREFA_gc | ADC_PRESC_DIV128_gc;
```

Correct Read-Modify-Write – Compiled

```
 18e:e0 e0       ldi r30, 0x00    ; Address to ADC0 struct (Z-pointer low byte)
 190:f6 e0       ldi r31, 0x06    ; Address to ADC0 struct (Z-pointer high byte)

ADC0.CTRLC &= ~ADC_REFSEL_gm;
 196:82 81       ldd r24, Z+2     ; Load ADC0.CTRLC into r24 (byte 2 in ADC0 struct)
 198:8f 7c       andi r24, 0xCF   ; r24 &= ~ADC_REFSEL_gm
 19a:82 83       std Z+2, r24     ; Load r24 into ADC0.CTRLC
ADC0.CTRLC |= ADC_REFSEL_VREFA_gc;
 19c:82 81       ldd r24, Z+2     ; Load ADC0.CTRLC into r24
 19e:80 62       ori r24, 0x20    ; r24 |= ADC_REFSEL_VREFA_gc
 1a0:82 83       std Z+2, r24     ; Load r24 into ADC0.CTRLC

ADC0.CTRLC = (ADC0.CTRLC & ~ADC_REFSEL_gm) | ADC_REFSEL_VREFA_gc;
 1a2:82 81       ldd r24, Z+2     ; Load ADC0.CTRLC into r24
 1a4:8f 7c       andi r24, 0xCF   ; r24 &= ~ADC_REFSEL_gm
 1a6:80 62       ori r24, 0x20    ; r24 |= ADC_REFSEL_VREFA_gc
 1a8:82 83       std Z+2, r24     ; Load r24 into ADC0.CTRLC

ADC0.CTRLC = ADC_REFSEL_VREFA_gc | ADC_PRESC_DIV128_gc;
 1aa:86 e2       ldi r24, 0x26    ; Hex value of ADC_REFSEL_VREFA_gc | ADC_PRESC_DIV128_gc
 1ac:82 83       std Z+2, r24     ; Load r24 into ADC0.CTRLC
```

Thank You, Microchip leaves at 21:00 ☺

Two hands in the air if you want merch in the mean time