Introduction to Microcontrollers

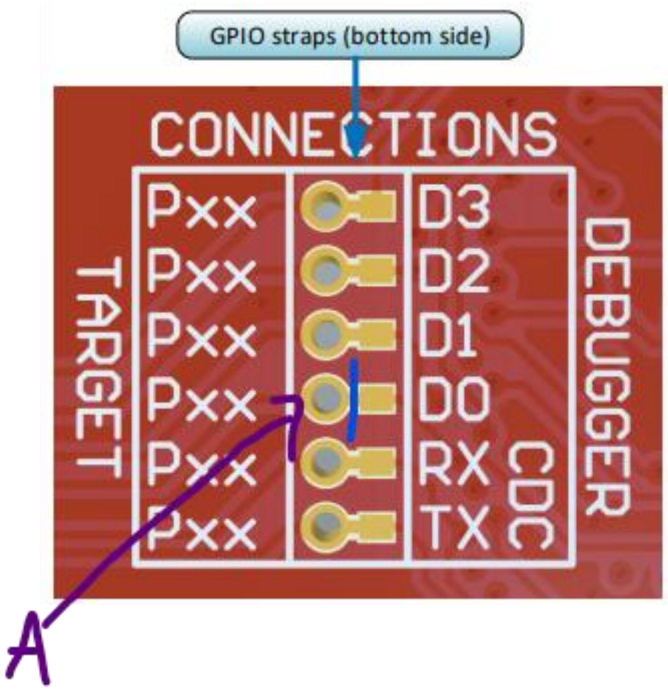
Practical Implementations

Goals

- Learn how microcontrollers can be used to control real life hardware
- Learn the basics of transistors
- Use what we have learnt to control a DC motor (Teams of two)
  - Speed control via PWM
  - Direction via GPIO
  - Speed control input via ADC (potentiometer)
  - Direction control input via button (GPIO)

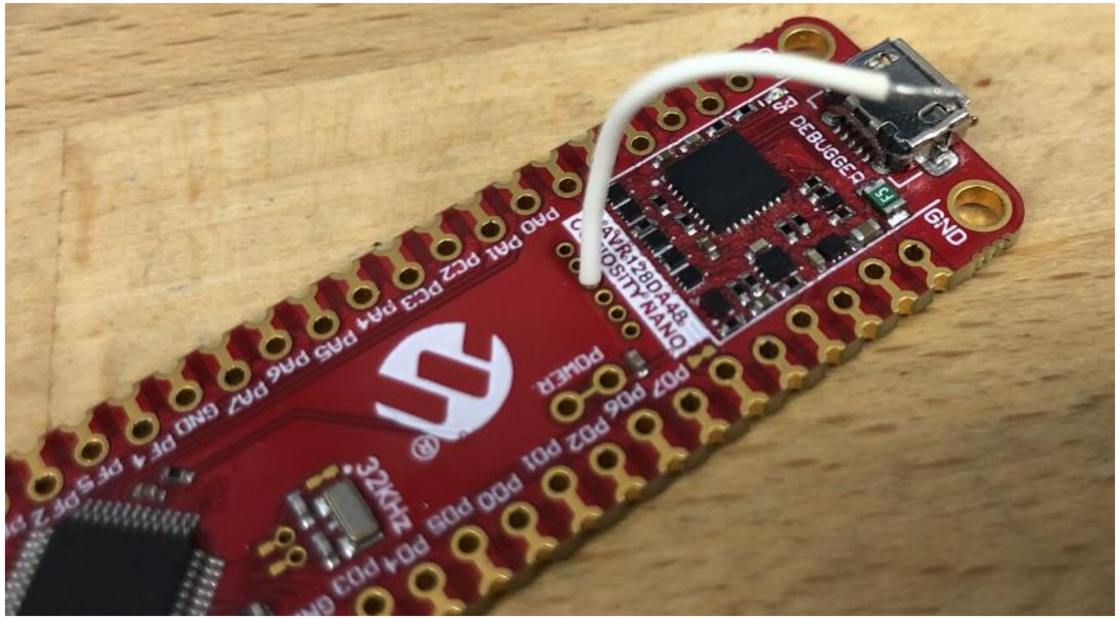
Prep: «Hack» Devboard

- A: Cut wire (blue line) on the back of devboard
- Configure IDE to not hide all devices
  - Microchip
    - Tools/Options
    - Tools/Tool Settings
    - Set «Hide unsupported devices» to «False»
  - MPLAB
    - Not easily available
    - Atmel ICE



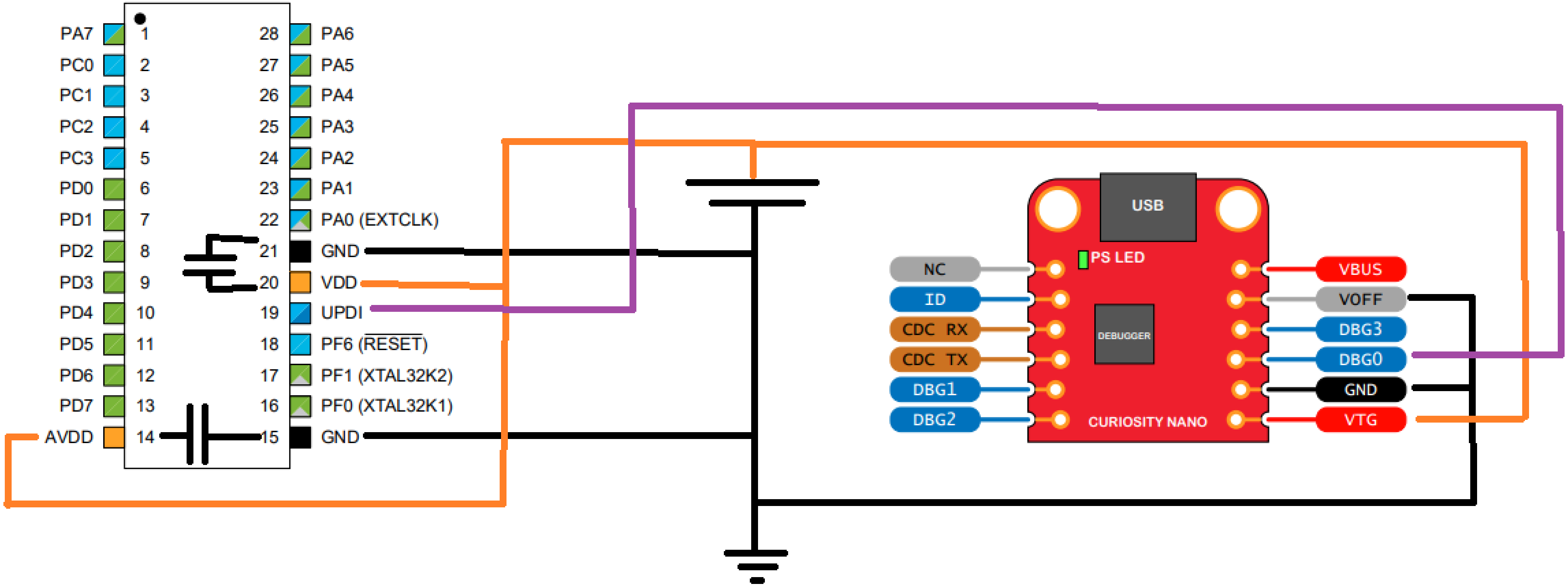
Prep: Hack Devboard

- Optional (later)
  - Solder a wire to UPDI of AVR on devboard
    - You can now program external microcontrollers and the built-in AVR effortlessly
- To get it back to normal
  - Make a solder bridge between cut wires
    - Or wire



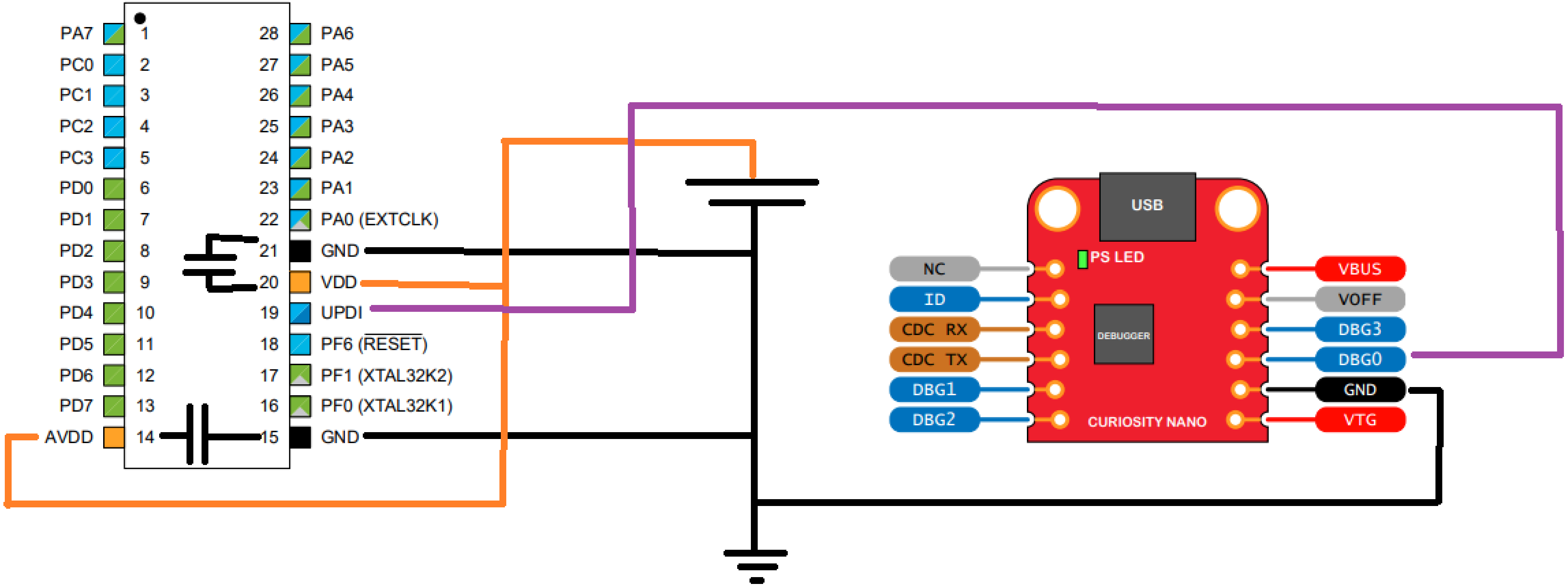
Task 1: Minimum setup

- Debugger can measure VTG
  - Make sure VOFF is connected to GND so the internal regulator is OFF



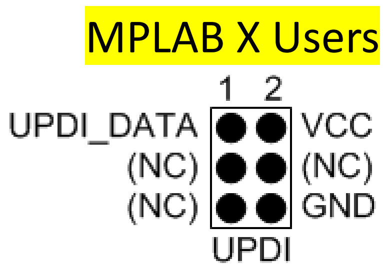
Task 1: Cheating setup

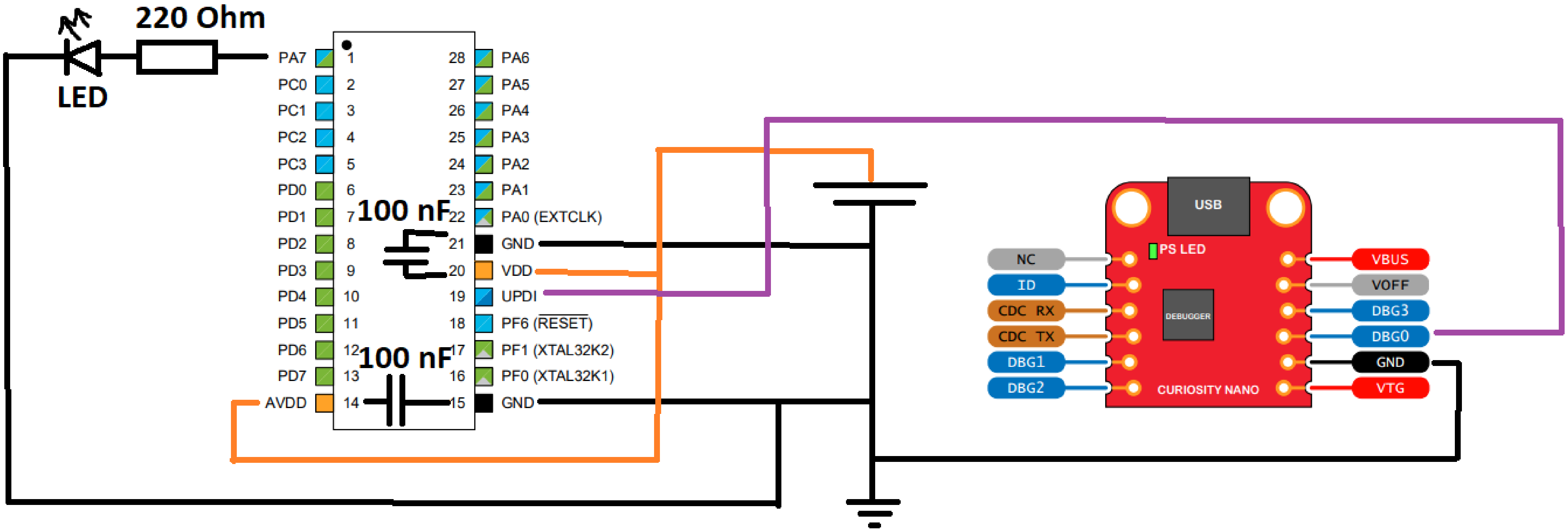
- Should be OK because UPDI is common drain with pull-up
  - Though it is not a good practice, it is ok for testing



Task 1: Building!

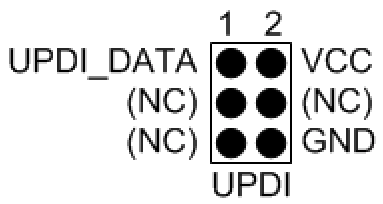
- Keep capacitors as close as possible to GND and VDD
- Create a blink LED program





Standard Programming header

- AVRs
  - Top view
- Atmel ICE compatible
  - Atmel ICE does not provide power
  - Circuit should therefore be externally powered
- Your custom printed circuit boards should use this layout
  - 2.54 mm pitch
  - 2x3 pin header



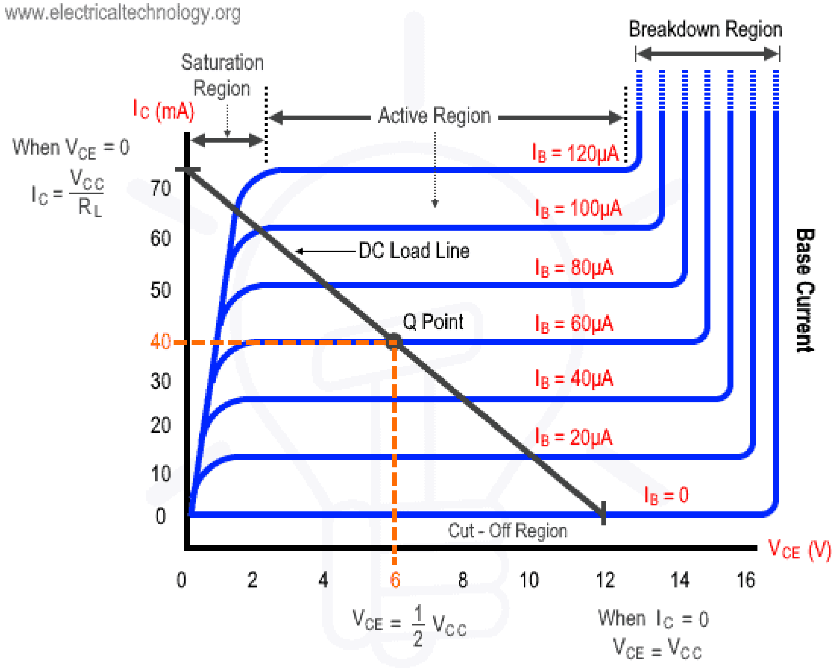
Quick Introduction to Transistors

- MOSFET
  - Metal Oxide Semiconductor Field Effect Transistor
  - Common in Ics
  - Voltage controlled
  - N and P type
- BJT
  - Bipolar Junction Transistor
  - Cheap
  - Current controlled
  - NPN and PMP type
- And many other
  - JFET, Thyristor, Darlington pair, IGBT etc
- Pros/Cons can easily be found online
  - We will stick with BJT due to their simplicity, availability, and price

BJT

- BJT
  - Collector current - Maximum current through device
  - Collector-Emitter breakdown voltage - Maximum voltage over device
  - Power dissipation - «Maximum» power of device
  - Gain / hFE / β - Current gain – Varies greatly
- NPN
  - Low-Side control
- PNP
  - High-Side control
- BJT vs FET
  - Drain ≈ Collector
  - Source ≈ Emitter
  - Gate ≈ Base

www.electricaltechnology.org

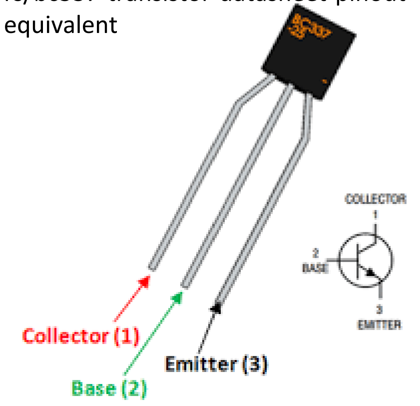


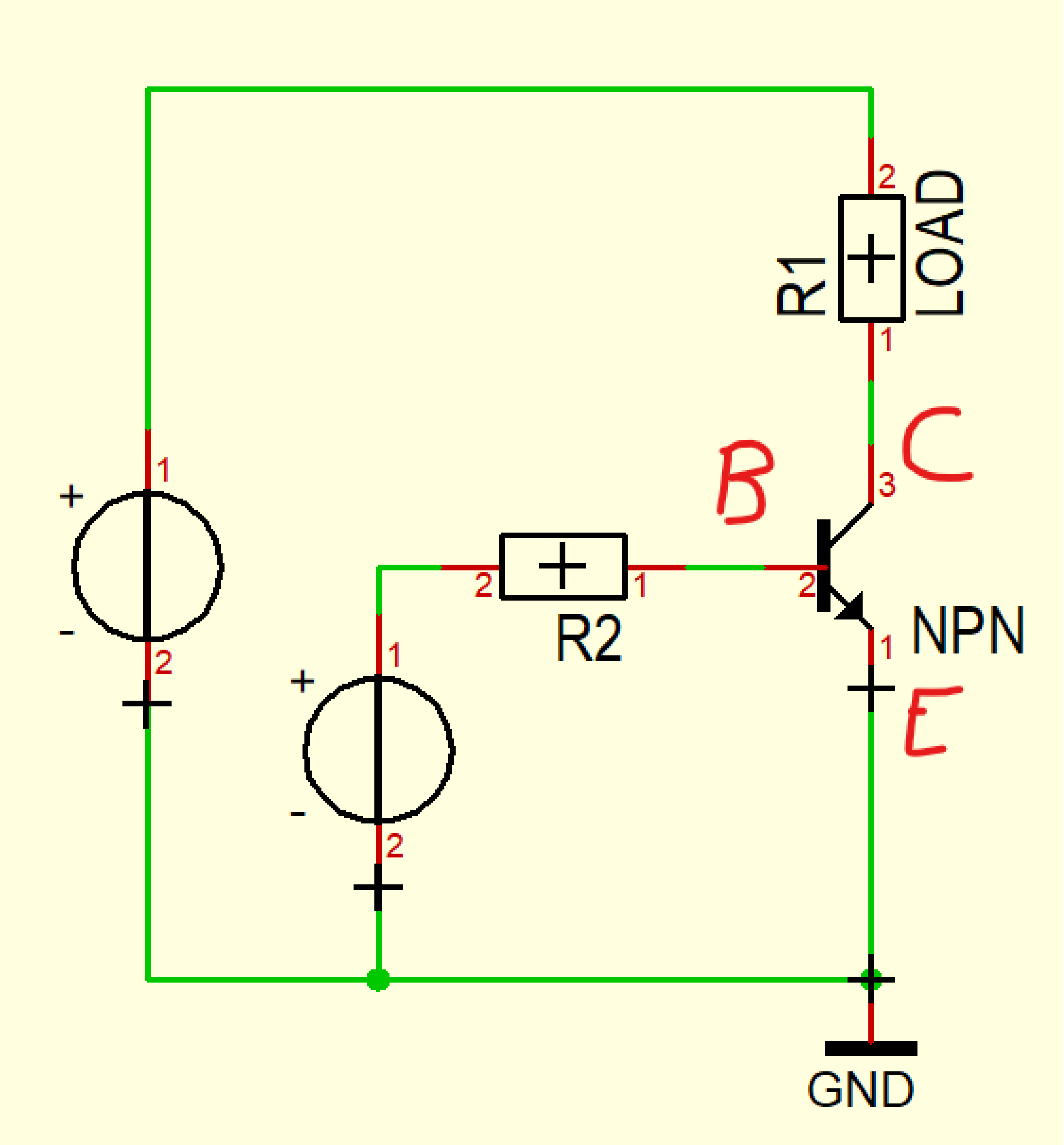
Collector Characteristics Curves & Operation Regions of Transistor
Common Emitter (CE) - NPN Transistor

NPN

- As a switch
  - We will not explore any further
- Gate Voltage
  - Must be at least 0.7 V higher than Emitter voltage
- Emitter
  - Connected to GND
- Collector
  - Connected to LOAD
- Current through B to E
  - Controlled by R2
  - I_BE
- Current through C to E
  - I_CE = hFE * I_BE

https://components101.com/transistors/bc337-transistor-datasheet-pinout-equivalent

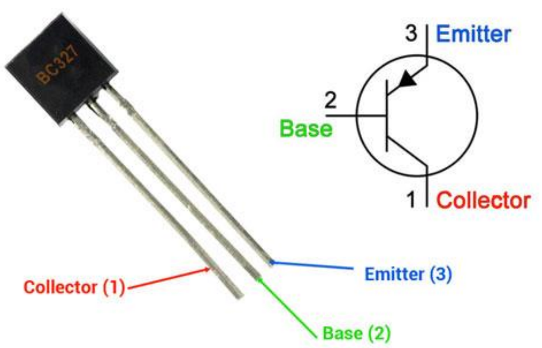


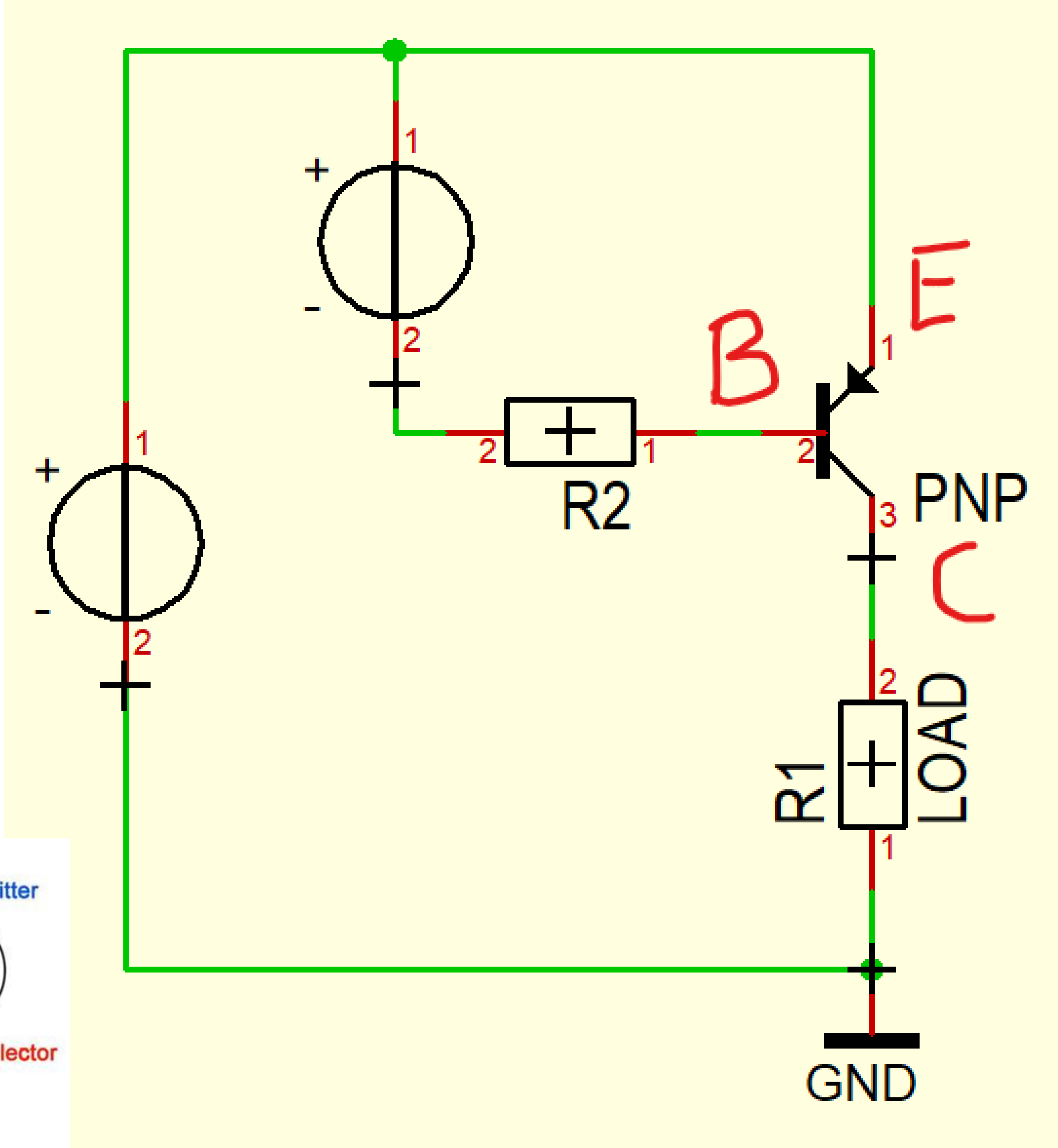


PNP

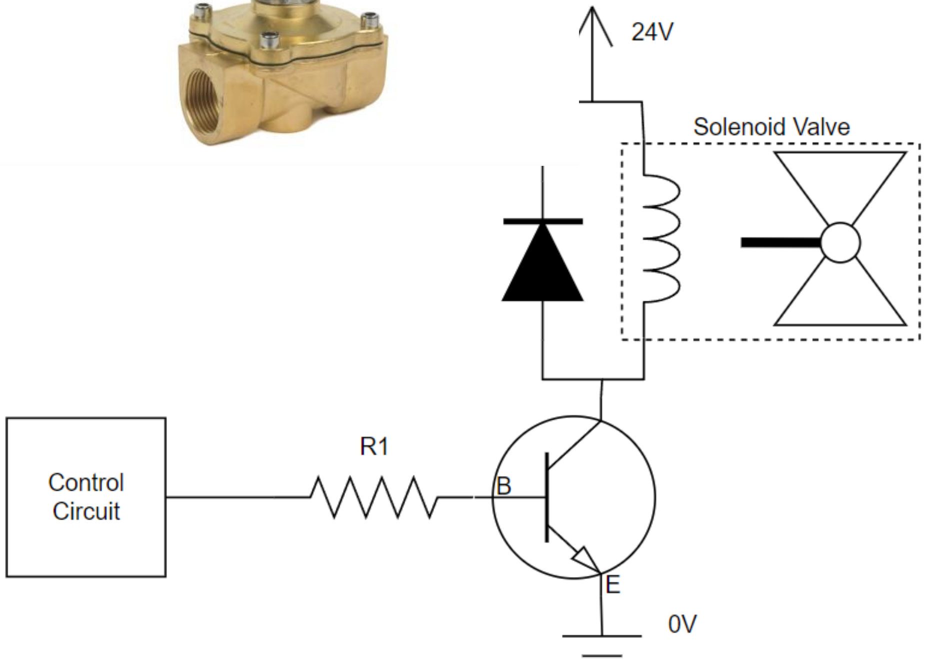
- As a switch
  - We will not explore any further
- Gate Voltage
  - Must be at least 0.7 V lower than Emitter voltage
- Emitter
  - Connected to positive power supply
- Collector
  - Connected to LOAD
- Current through B to E
  - Controlled by R2
  - I_BE
- Current through C to E
  - I_CE = hFE * I_BE

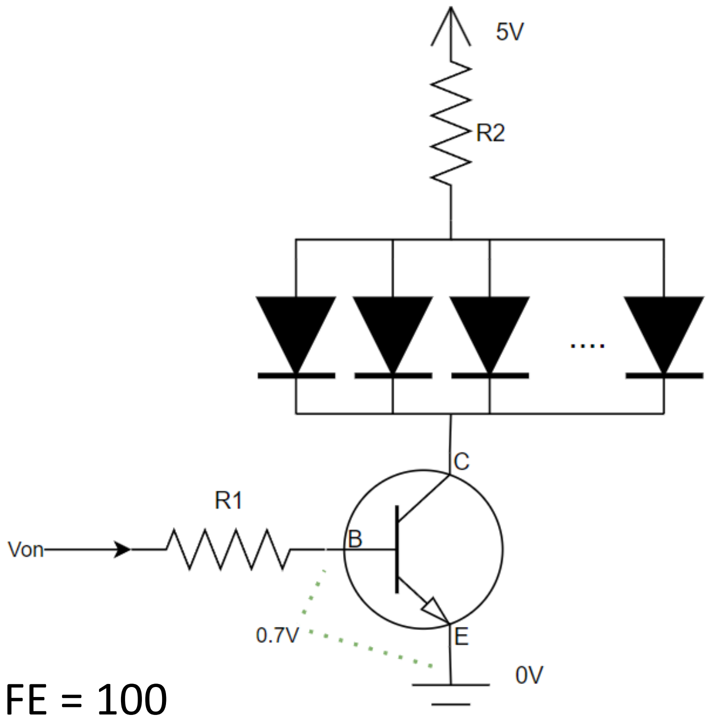
https://www.bitfoic.com/components/bc327-pnp-transistor-pinout-cad-model-features-working-principle-and-applications3-examples?id=46





Typical Use





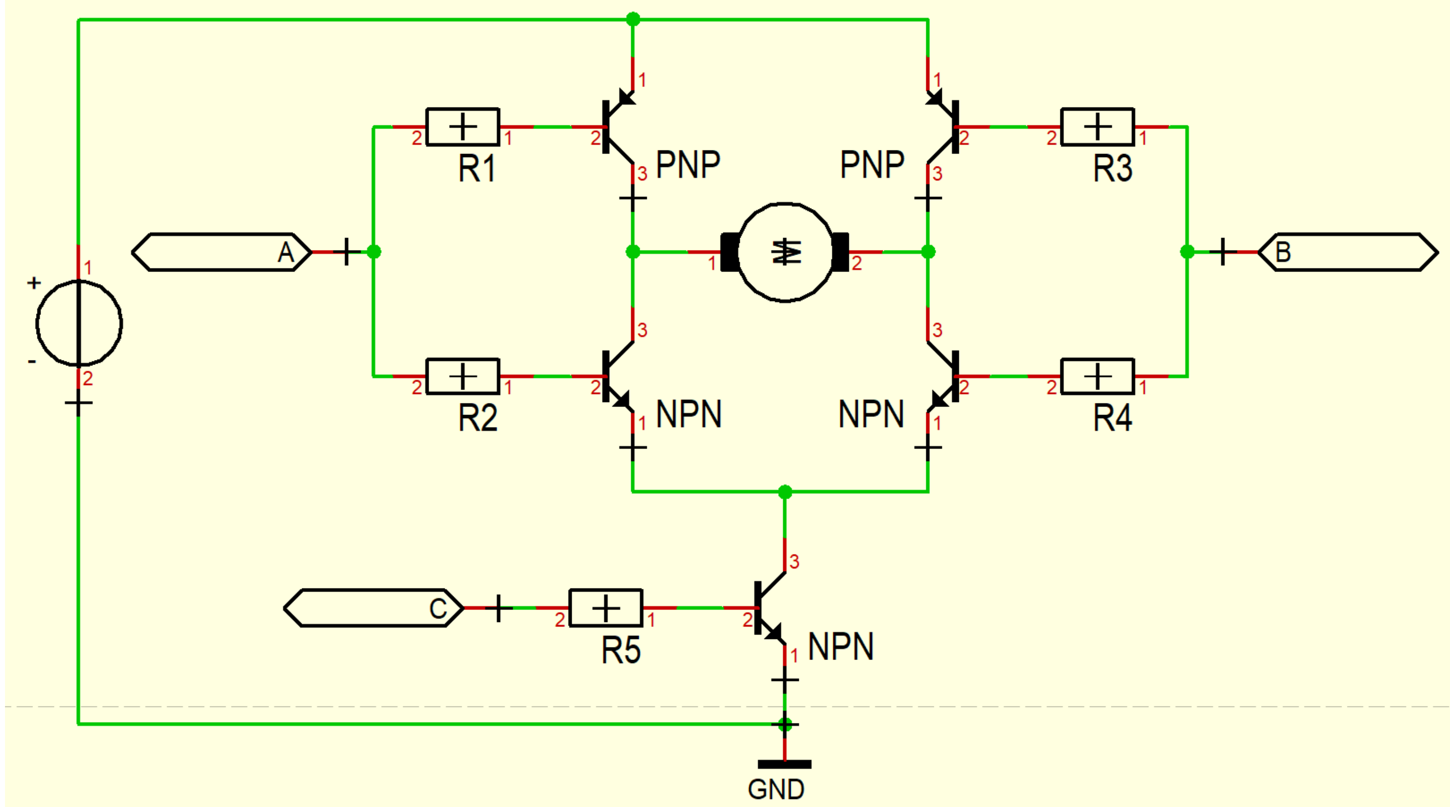
hFE = 100
Von = 1.7 V
R2 = 100 Ohm
Target current = 10 mA

I_BE = (1.7 – 0.7) / R1
I_CE = hFE * I_BE = 10 mA
I_BE = 0.01 / 100 = 0.0001
R1 = (1.7 – 0.7) / 0.0001 = 10 kOhm
V_R2 = 0.01 * 100 = 1 V
V_CE = 5 – V_R2 – V_Diode

Setting R1 lower than 10 kOhm ensures transistor operates “like a switch”

Use Today

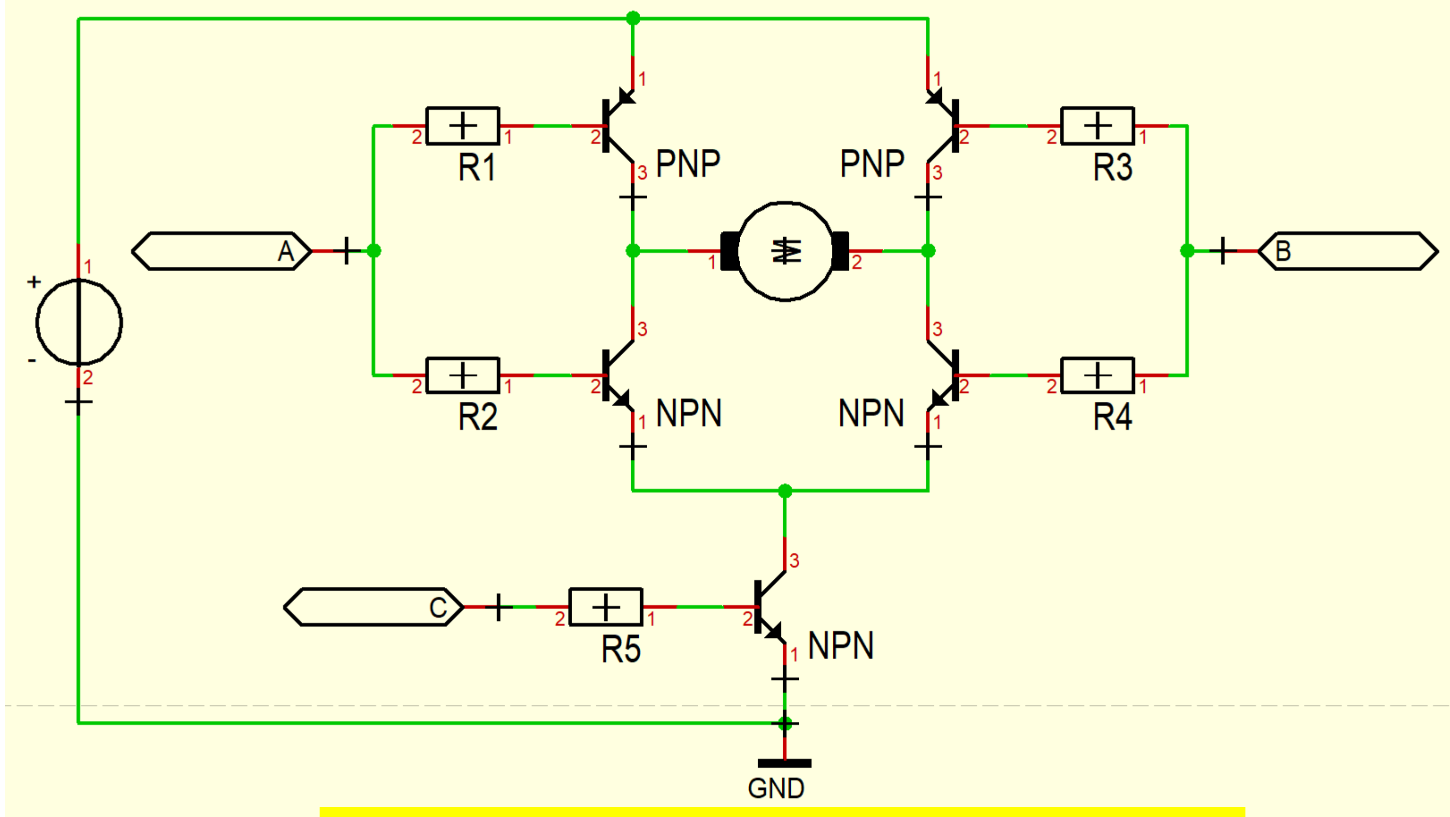
- H-Bridge
  - Though not a good one
  - Fine for demo purposes
  - Do not use this design for your experiments
- A
  - Direction control
  - GPIO Logic
- B
  - Direction control
  - Complementary to A
- C
  - Speed control
  - PWM



This is not a good implementation of a H bridge

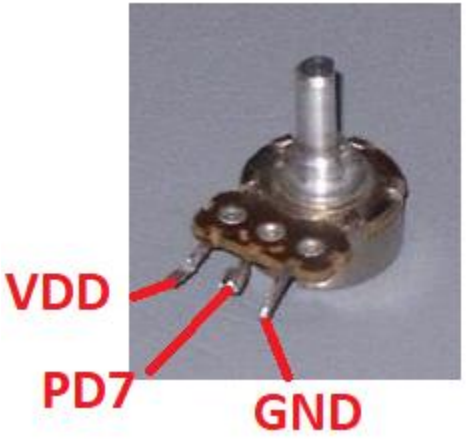
- MOSFET or IGBT are better suited transistors
- There should be diodes protecting the transistors
- There should be a bulk capacitance near the bridge
- A and B should be inverted in hardware

Task 2: Hardware



This is not a good implementation of a H bridge

- Modify devboard
  - PC0 to A
  - PC1 to B
  - PA0 to C
- Add a potentiometer
  - Middle pin to PD7
- Add a button
  - Connect to PA1
- Copy ADC and PWM code from previous labs
  - Github
- Power supply should be from VBUS on your devboard
  - DC motor needs 5 V
  - DC motor uses about 50 mA
  - hFE approx. 100
  - Assuming no voltage drop over transistors (Saturation – "switch")
    - I_BE = 50 / 100 = 0.5 mA
    - R < V/I = (5 – 0.7) / 0.0005 = 8600 Ohm
    - Closes value I found was 4.7 kOhm



Task 3: Software

- New project
  - AVR128DA28
- Create two functions for control
  - Motor speed control
    - Updates TCA0.SINGLE.CMP0BUF
    - We will set TOP top be 4095 as the ADC result is a value between 0 and 4095
      - No need to rescale
  - Motor direction control
    - Updates OUT value of PC0 and PC1
      - PC0 and PC1 should never be the same value

Write these down!

```
void motor_direction_set(uint8_t dir)
{
    if (dir)
    {
        // Clear first, then set
        PORTC.OUTCLR = PIN0_bm;
        PORTC.OUTSET = PIN1_bm;
    }
    else
    {
        // Clear first, then set
        PORTC.OUTCLR = PIN1_bm;
        PORTC.OUTSET = PIN0_bm;
    }
}

void motor_speed_set(uint16_t speed)
{
    // Guard for invalid speed setting
    if (speed > 4095)
    {
        speed = 4095;
    }

    // Update duty cycle
    TCA0.SINGLE.CMP0BUF = speed;
}
```

Task 3: Software

- PORT Output
  - Use motor_direction_set() before setting PORT direction!
- PORT Direction
  - PA0 – Output
  - PA1 – Input
    - With pull-up!
    - PORTA.PIN1CTRL = PORT_PULLUPEN_bm;
  - PC0 – Output
  - PC1 – Output
- ADC
  - Same as potentiometer example
  - Ratiometric measurement
  - New prescaler value of DIV12
- TCA
  - Same as PWM example
  - With a new prescaler of DIV8
    - This is something that should be tuned depending on motor

Don’t bother writing this yet

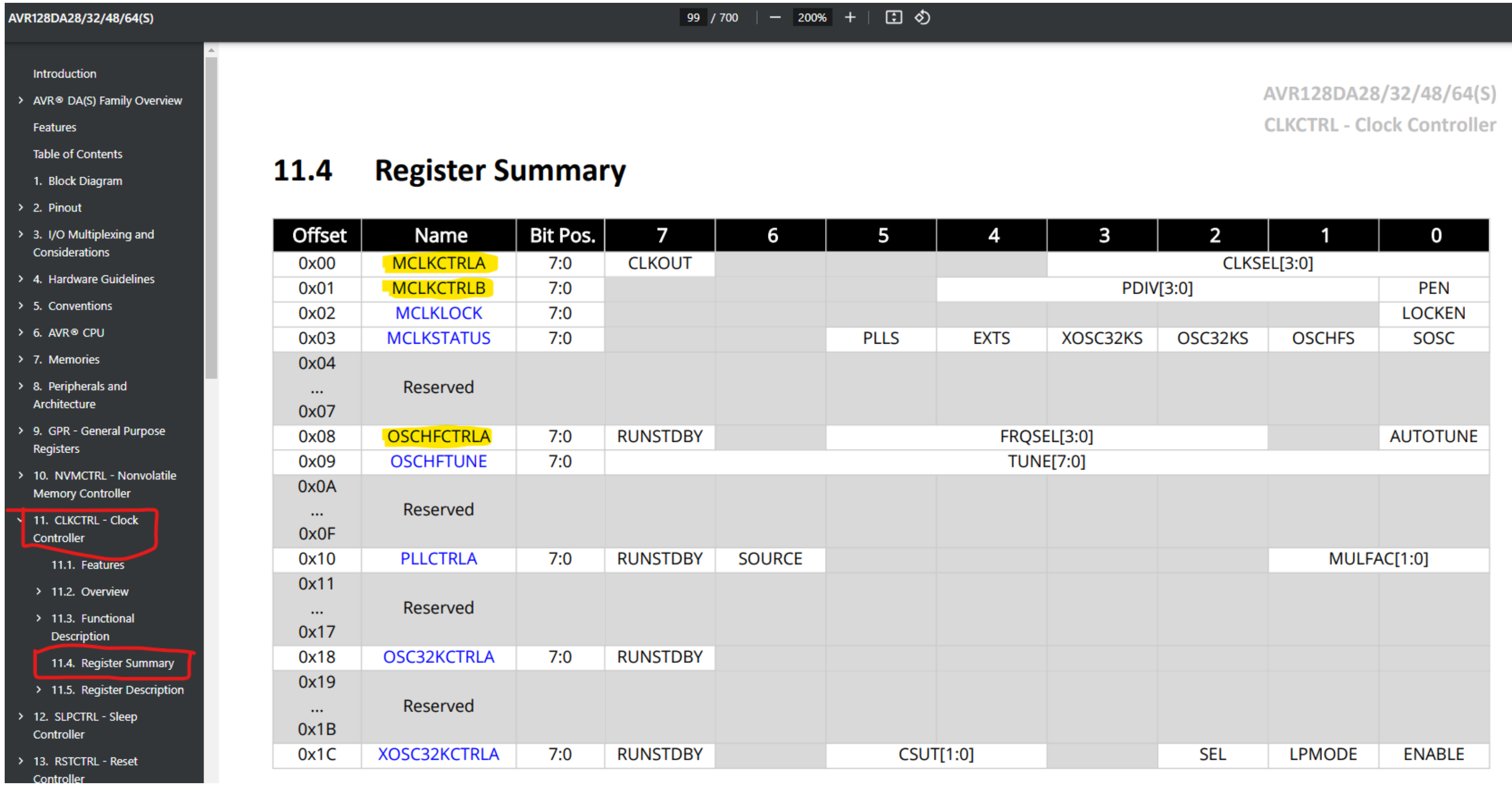
```
/*
    Setup ADC
*/
VREF.ADC0REF         = VREF_REFSEL_VDD_gc;   // VDD as ADC reference

ADC0.MUXPOS          = ADC_MUXPOS_AIN7_gc;   // PD7
ADC0.CTRLC           = ADC_PRESC_DIV12_gc;   // 24 MHz / 12 = 2 MHz (max freq)
ADC0.CTRLA           = ADC_ENABLE_bm;

/*
    Setup TCA
*/
TCA0.SINGLE.PER      = 4095;                 // Period
TCA0.SINGLE.CTRLB    = TCA_SINGLE_CMP0EN_bm  // Waveform output enable
                     | TCA_SINGLE_WGMODE_SINGLESLOPE_gc;
TCA0.SINGLE.CTRLA    = TCA_SINGLE_ENABLE_bm
                     | TCA_SINGLE_CLKSEL_DIV8_gc; // DIV 8, motor dependent
```

Task 3: F_CPU

- CLKCTRL peripheral
- MCLKCTRLA
  - Which clock is used by F_CPU
  - Default is OSCHF (Correct oscillator)
- MCLKCTRLB
  - Prescaler for F_CPU
  - Divide CLK by some value (Default = DIV1)
  - Leave as is
- OSCHFCTRLA
  - Control register for high frequency clock
  - Default value = 4 MHz
  - Change to 24 MHz


AVR128DA28/32/48/64(S)   99 / 700   200%

11.4 Register Summary

Offset	Name	Bit Pos.	7	6	5	4	3	2	1	0
0x00	MCLKCTRLA	7:0	CLKOUT				CLKSEL[3:0]			
0x01	MCLKCTRLB	7:0				PDIV[3:0]				PEN
0x02	MCLKLOCK	7:0								LOCKEN
0x03	MCLKSTATUS	7:0			PLLS	EXTS	XOSC32KS	OSC32KS	OSCHFS	SOSC
0x04 ... 0x07	Reserved									
0x08	OSCHFCTRLA	7:0	RUNSTDBY		FRQSEL[3:0]					AUTOTUNE
0x09	OSCHFTUNE	7:0	TUNE[7:0]							
0x0A ... 0x0F	Reserved									
0x10	PLLCTRLA	7:0	RUNSTDBY	SOURCE					MULFAC[1:0]	
0x11 ... 0x17	Reserved									
0x18	OSC32KCTRLA	7:0	RUNSTDBY							
0x19 ... 0x1B	Reserved									
0x1C	XOSC32KCTRLA	7:0	RUNSTDBY		CSUT[1:0]			SEL	LPMODE	ENABLE

Task 3: F_CPU

- A
  - Reset value
  - Simply that default output is 4 MHz
- B
  - Locked register!
  - We need to unlock it before writing
- C
  - Group we want to change
- D
  - Default value
- E
  - Desired value

11.5.5 Internal High-Frequency Oscillator Control A

A B

Name: OSCHFCTRLA
Offset: 0x08
Reset: 0x0C
Property: Configuration Change Protection

C

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

Bit 7 – RUNSTDBY Run Standby

This bit controls whether the internal high-frequency oscillator (OSCHF) is always running or not.

Value	Description
0	The OSCHF oscillator will only run when requested by a peripheral or by the main clock [(1)]
1	The OSCHF oscillator will always run in Active, Idle and Standby sleep modes [(2)]

Notes:

1. The requesting peripheral, or the main clock, must take the oscillator start-up time into account.
2. The oscillator signal is only available if requested and will be available after two OSCHF cycles.

Bits 5:2 – FRQSEL[3:0] Frequency Select

This bit field controls the output frequency of the internal high-frequency oscillator (OSCHF).

Value	Name	Description
0x0	1 MHz	1 MHz output
0x1	2 MHz	2 MHz output
0x2	3 MHz	3 MHz output
0x3	4 MHz	4 MHz output (default)
0x4	-	Reserved
0x5	8 MHz	8 MHz output
0x6	12 MHz	12 MHz output
0x7	16 MHz	16 MHz output
0x8	20 MHz	20 MHz output
0x9	24 MHz	24 MHz output
Other	-	Reserved

D E

Task 3: F_CPU

```
#define F_CPU 24000000
```

Remember to change your definition of F_CPU if you use util/delay.h

- Configuration Change Protection
  - CCP
- Some registers are more important than others
  - A bit flip could brick the device
  - Solution: Lock registers
  - Unlock with a specific key which only lasts a few clock cycles
- CCP is a register
  - To unlock all IO registers use the key CCP_IOREG_gc

```
// Disable IOREG protected registers for a few clock cycles
CCP = CCP_IOREG_gc;

// Set the clock frequency of the AVR to 24 MHz
CLKCTRL.OSCHFCTRLA  = CLKCTRL_FRQSEL_24M_gc; // Protected register
```

Write this to the start of main

```c
int main(void)
{
    // Disable IOREG protected registers for a few clock cycles
    CCP = CCP_IOREG_gc;

    // Set the clock frequency of the AVR to 24 MHz
    CLKCTRL.OSCHFCTRLA  = CLKCTRL_FRQSEL_24M_gc; // Protected register

    uint8_t motor_dir = 0;
    uint16_t motor_speed = 2048;

    PORTA.DIRSET        = PIN0_bm;          // PWM for speed control
    PORTA.PIN1CTRL      = PORT_PULLUPEN_bm; // Button for direction control

    // Set output first such that H-bridge is not on in invalid state
    motor_direction_set(motor_dir);
    motor_speed_set(motor_speed);
    PORTC.DIRSET        = PIN0_bm
                        | PIN1_bm;

    /*
        Setup ADC
    */
    VREF.ADC0REF        = VREF_REFSEL_VDD_gc;   // VDD as ADC reference

    ADC0.MUXPOS         = ADC_MUXPOS_AIN7_gc;   // PD7
    ADC0.CTRLC          = ADC_PRESC_DIV12_gc;   // 24 MHz / 12 = 2 MHz (max freq)
    ADC0.CTRLA          = ADC_ENABLE_bm;

    /*
        Setup TCA
    */
    TCA0.SINGLE.PER     = 4095;                 // Period
    TCA0.SINGLE.CTRLB   = TCA_SINGLE_CMP0EN_bm  // Waveform output enable
                        | TCA_SINGLE_WGMODE_SINGLESLOPE_gc;
    TCA0.SINGLE.CTRLA   = TCA_SINGLE_ENABLE_bm
                        | TCA_SINGLE_CLKSEL_DIV8_gc; // DIV 8, motor dependent
```

```c
while (1)
{
    // Check button to see if we need to change direction
    if (!(PORTA.IN & PIN1_bm))
    {
        // Invert motor direction
        motor_dir = !motor_dir;
        motor_direction_set(motor_dir);

        // Simple button debounce
        _delay_ms(50);
    }

    // Measure potentiometer value and update motor speed
    ADC0.COMMAND = ADC_STCONV_bm;
    while(!(ADC0.INTFLAGS & ADC_RESRDY_bm));
    motor_speed = ADC0.RES;

    motor_speed_set(motor_speed);
}
```

Going Forward

- Three paths
  - Continue learning to code on register level
    - Pro
      - You will learn how your hardware is actually working
      - You can stick with AVRs for longer as your code is more efficient, and you can create custom solutions for your particular situation
    - Con
      - Takes time to become good
      - You must read the datasheet
        - Divide and conquer!
  - Go back to Arduino
    - Pro
      - Simplified environment with use of libraries makes it good for rapid prototyping
      - Open source - Lots of code available
    - Con
      - Not efficient
        - Code size, code optimization, libraries do not support every hardware possibility
      - Open source - Prone to failure
  - Use of code configurators
    - Midway between Arduino and coding with registers. Better control, but requires more knowledge
    - For AVR
      - MPLAB Code Configurator (MCC) Melody
      - Atmel Start (online tool)
        - Horrible

That’s all

- Hope you’ve had a good time
- Feedback (QR on next page)
- This is just an introduction!
  - Learn by doing
  - I am available for any of your projects, just send me a message!
    - Any project
    - I am also available if you want to discuss project ideas
    - Send me a message on slack!
- Resources
  - I will post a few additional resources on github by Sunday
    - Common problems
      - Such as receiving data via UART, writing to SD cards, non-blocking ADC, power saving, safety features, etc
    - Exotic examples
      - Where coding on this level truly shines – Examples no library or configurator could do for you
    - Let me know if there is anything in particular you want to know more about

Feedback

- Course completion gifts!
  - In addition to keeping the devboard you also get to keep
  - AVR128DA28 IC and capacitors
  - MCP9700 temperature sensor
    - From last time if you didn’t keep it
  - Transistors if you’re so inclined
  - ATtiny404 and adapter board
    - ATtiny series are the cheapest AVRs on the market
    - ATtiny404 is one of them
    - Limited set peripherals
    - Perfect for small projects
    - Woops! Default F_CPU is 3333333 (20 MHz/6)
      - 3.3 V compatible only up to 10 MHz
      - 5.0 V compatible up to 20 MHz
    - Good solder training!
    - The AVR used as the SPI “sensor”
      - Code on github for both ATtiny404 and AVR128DA48