Getting started with baremetal programming

Before starting

So you want to start baremetal programming but you are sure where to start. You look online and see chip schematics, complex diagrams, unfamiliar dictionary and get discouraged. Does that sound familiar? Fear not! In this this short introduction to baremetal programming I plan on helping you get your feet wet. Take your first step and see where it leads you.

Prerequisites:

So what do you have to have before reading this? All of the skills below are not a must but they would greatly help you follow along:

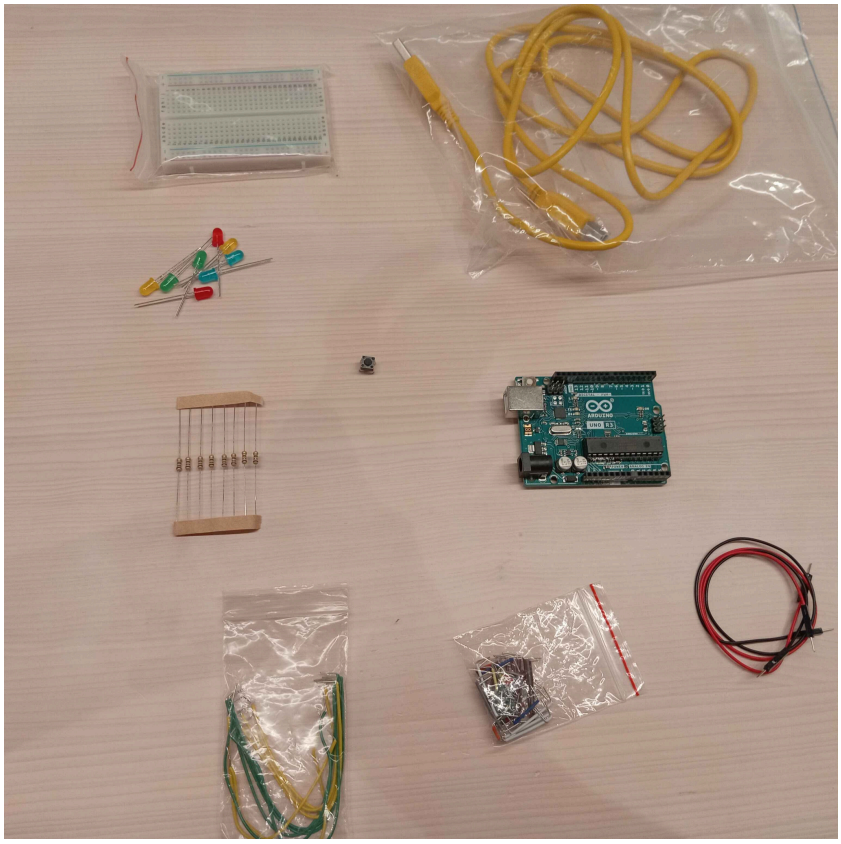
- C programming language basics
- Assembly programming basics
- Boolean algebra (bit manipulation)
- Interest in learning (this is a must)

Hardware prerequisites:

I recommend that you do not buy anything before reading through this introduction atleast once, so you get a better picture of whether this is something truly fit for you or not. However if you do want to follow along then this is what I will be using:

- A machine with Linux
    - Any machine and OS would work but I am using Linux so the burden of making it work on other OS is on the reader. Naturally the code we write is not OS specific but the compile pipeline is. Apple users usually can simply follow along by brew installing everything I will be using.
- An Arduino Uno R3 Board and its type a/b usb cable
- A breadboard
- Eight breadboard LED lights
- Breadboard wires
- Four breadboard jumper wires
- Eight 150-350 OHM resistors (as many as LED lights) for help finding the strength consult: https://www.calculator.net/resistor-calculator.html
- A breadboard button switch
- A breadboard buttonswitch

Below is a picture of all the needed components:



All the code will be available over at Github at:

https://github.com/Doxakis1/IntroductionToBareMetal

Installing the compiler

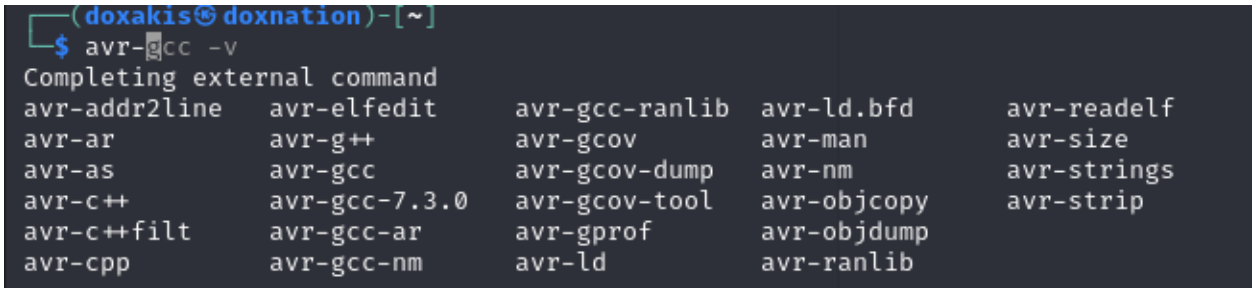
We need to install the following three packages:

- gcc-avr
- avr-libc
- avrdude

In the github repository you will find an installation.sh file which you can run that installs all the dependencies.

NOTE: if you are using a different package manager make sure to update the scripts package manager variables

After installing the avr compiler and closing/re-opening your terminal, if you type ‘avr-’ and press SHIFT you should see the following options:



This means you have all the tools needed to start the baremetal programming!

Why no Arduino IDE

If you have programmed with arduino before or you have seen any other beginner arduino tutorials you might have come across the Arduino IDE (integrated developer environment). The IDE is extremely useful for getting started and doing small Arduino projects as it provides out-of-the-box functionality. However this is an introduction to baremetal programming so we are not going to be using the IDE, instead we will be coding things ourselves!

What is an Arduino Uno R3

The Arduino Uno R3 is a microcontroller. It is widely used by beginners, hobbyists, and professionals for prototyping and learning electronics.

The core of the Arduino Uno is the ATmega328P microcontroller, which contains a CPU, memory (flash, SRAM, EEPROM), and input/output pins, all in one chip. This is the ‘brain’ of the Arduino and handles the execution of the code uploaded to it. ATmega328P is self-sufficient, meaning that it contains all the necessary components (CPU, memory, I/O) on a single chip to perform tasks without needing a full operating system or external peripherals (unlike a general-purpose computer).

The programmer of the ATmega328P has full control over everything in relation to the chip, all the memory, all peripherals and all the processing power it holds. This inherently means that unlike a traditional computer with a running operating system and kernel, you are not restricted

or safeguarded from doing anything you please. This means that you have full control and like uncle Ben would say, with great power comes great responsibility.

Memory management, resource management and interrupt logic is all up to you. If you make a mistake, you do not have a safeguard, you're doomed. This is in my opinion what makes baremetal programming so fun, you learn how everything works from the ground up and you get to carefully craft code that serves the purpose of your project.

The scary documentation

Through-out this introduction I will be using two important resources:

1) ATmega328P documentation (https://ww1.microchip.com/downloads/aemDocuments/documents/MCU08/ProductDocuments/DataSheets/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf)
2) AVR Instruction Set Manual (http://ww1.microchip.com/downloads/en/DeviceDoc/AVR-InstructionSet-Manual-DS40002198.pdf)
3) AVR-GCC Manual (https://gcc.gnu.org/wiki/avr-gcc)

I understand that those two files look extremely intimidating and of course we will not be discussing everything written in them, however my goal is to teach you and show you how I fetched the information about what I am teaching you in the hopes that in the future you will be able to find things on your own.

LET US START

Lesson00: Light the world up

Like in many programming languages the first program you write is a 'hello world' program, baremetal has a similar 'first light on' project. We will be doing this project, and all the rest projects, both in AVR assembly and C

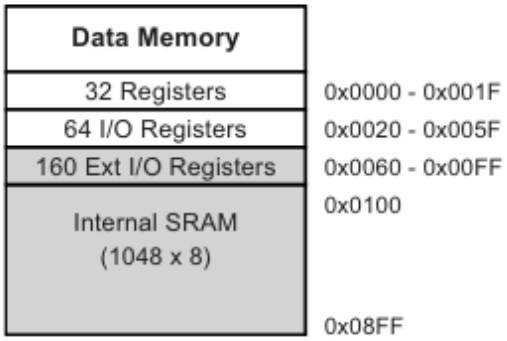
THEORY:

The registers:

If you have programmed assembly before, perhaps in an OS like linux you have been introduced to registers. Registers are the fastest memory that can be accessed by the cpu. In modern CPUs these registers are not mapped in physical memory, however that has not always been the case and is certainly not the case with the ATmega328p. All the registers in the ATmega328p live in addressable physical memory.

If we inspect the ATmega328P documentation on page 18, we will find figure 7.2 that looks like this:

Figure 7-2. Data Memory Map



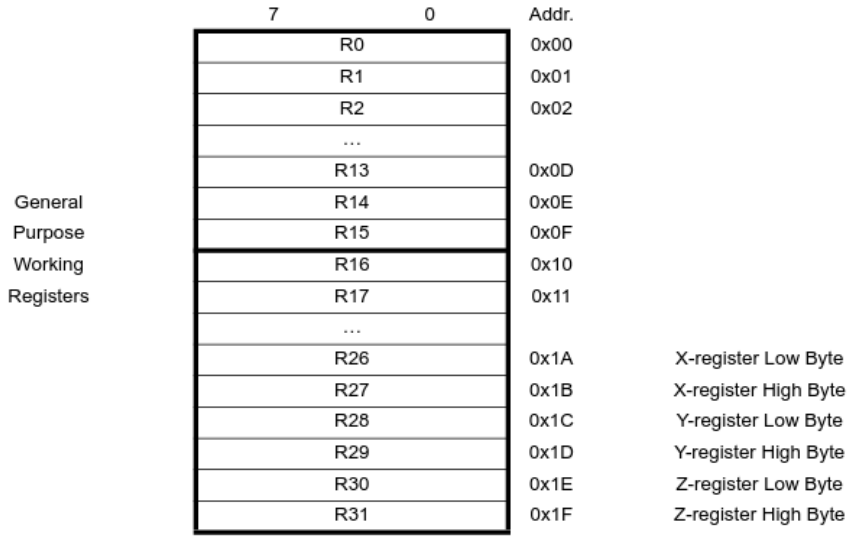
This is the our memory map, the bread and butter of baremetal programming. Everything we need can be done by manipulating these addresses from 0x0000 to 0x08FF.

As the figure shows, the first addresses 0x0000 - 0x001F are the 32 8-bit registers of our processor. That means that unlike other assembly languages we are free to name our registers however we want. However as we can see in the AVR Instruction Set Manual , they usually denote registers by R(num) so register one would be R1 and is just the address of 0x00.

As a C programmer you might have gotten a heart attack… DID WE JUST RAWDOG ADDRESS 0x00??? Yes… yes we did!

All the memory is ours, there are no restrictions, and even better the memory is not virtualized, it is all pure physical real memory addresses. Below is a figure from the ATmega328p documentation page 12 that shows the naming of the registers:

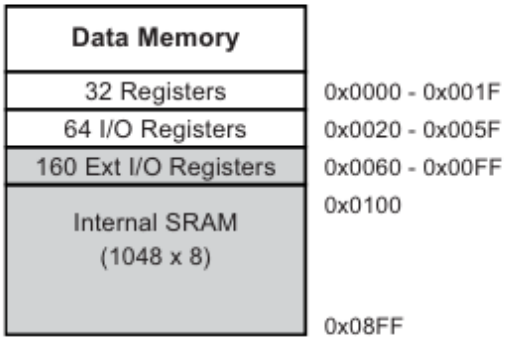
Figure 6-2. AVR CPU General Purpose Working Registers



The peripherals:

If we look at the Figure 7.2 again we will see that after the general purpose registers are some memory addresses called ‘I/O Registers’:

Figure 7-2. Data Memory Map



What exactly are those?

Well those are the addresses where our peripherals are mapped, every external device is mapped on those 64 bytes between 0x0020 and 0x005F. On the ATmega328P documentation on page 72, we find more information about the addresses and their purpose but don’t worry we will come back to this.

How does communication happen:

Majority of the devices are I/O (Input and Output) that means that you can send them data, and they can also send you data back. Obviously there are some devices that are unidirectional meaning you can only get data to flow in one direction, but for our introduction purposes let us

discuss bidirectional devices. How does the device know if it is supposed to be sending data or receiving data? Well it happens with electricity of course!

There are specific addresses in our physical memory called Data Direction Registers. These locations dictate whether the device is in send mode, or receive mode.

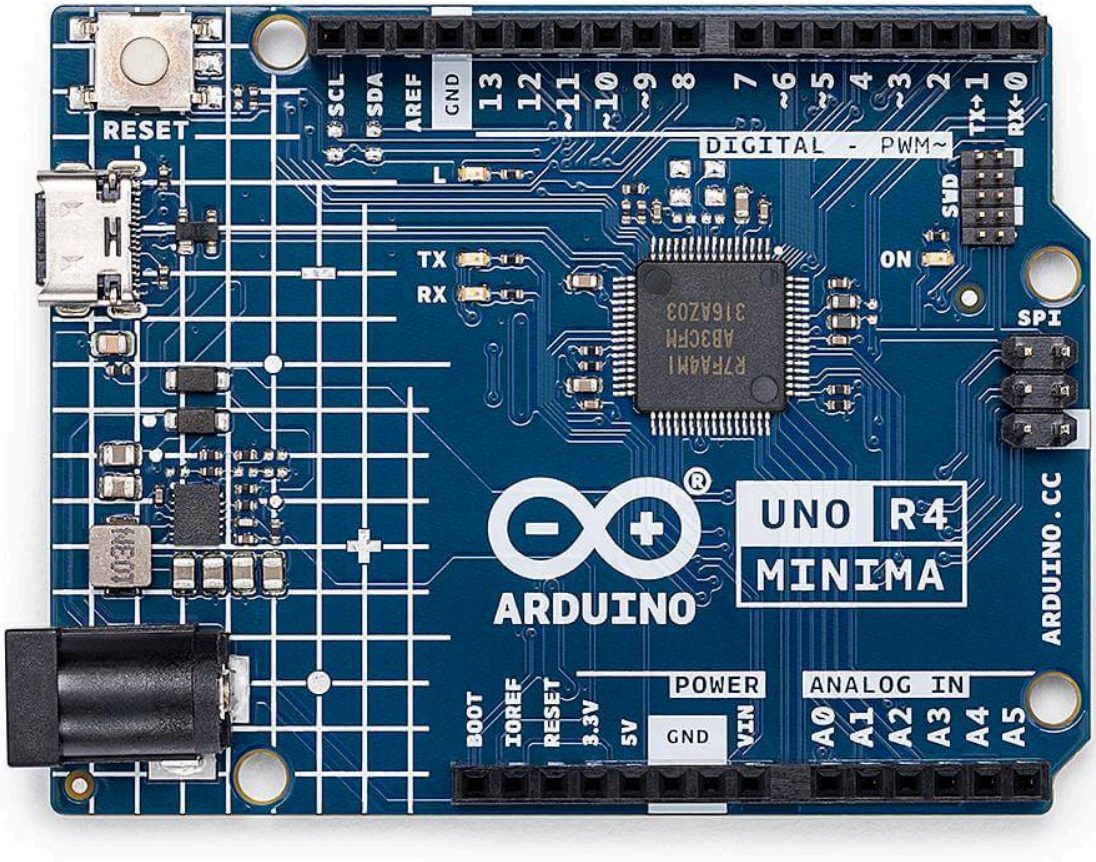
Still a bit confused? Don’t worry, it will become clear once we get into the first example of coding.

Clock cycles:

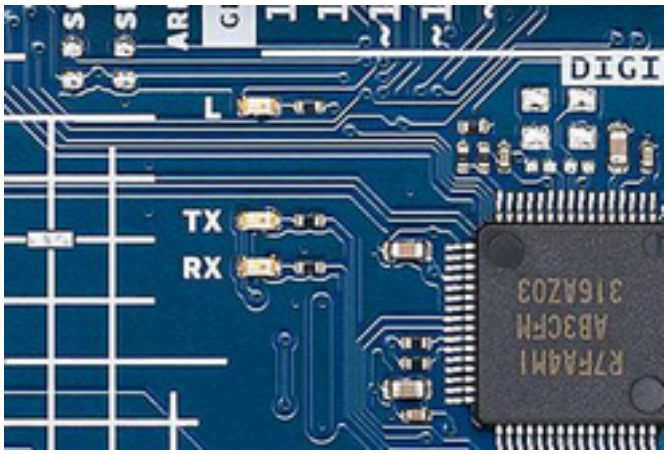
Every electronic device that has a processor has a clock cycle. If processors were humen, clock cycles would be their heart beats… and they beat fast… the ATmega328p runs at 16Mhz which means its heartbeat is 16 000 000 times per second. Yes you saw the zeros correctly. And as crazy as this sounds, modern cpus consider that speed laughable as they beat at billions of times per second.

Each clock cycle, electrical mechanics advance the program's execution forward. If we look at the AVR Instruction manual on page 18 (table 5.2) we will see that each ATmega328p instruction takes a specific amount of cycles to be executed. Some take one cycle like the ADD instruction, some take three cycles like the JMP instruction. That means that per second we can perform 16 000 000 ADD instructions or 5 333 333 JMP instructions. These are not always 100% accurate but quite accurate nevertheless.

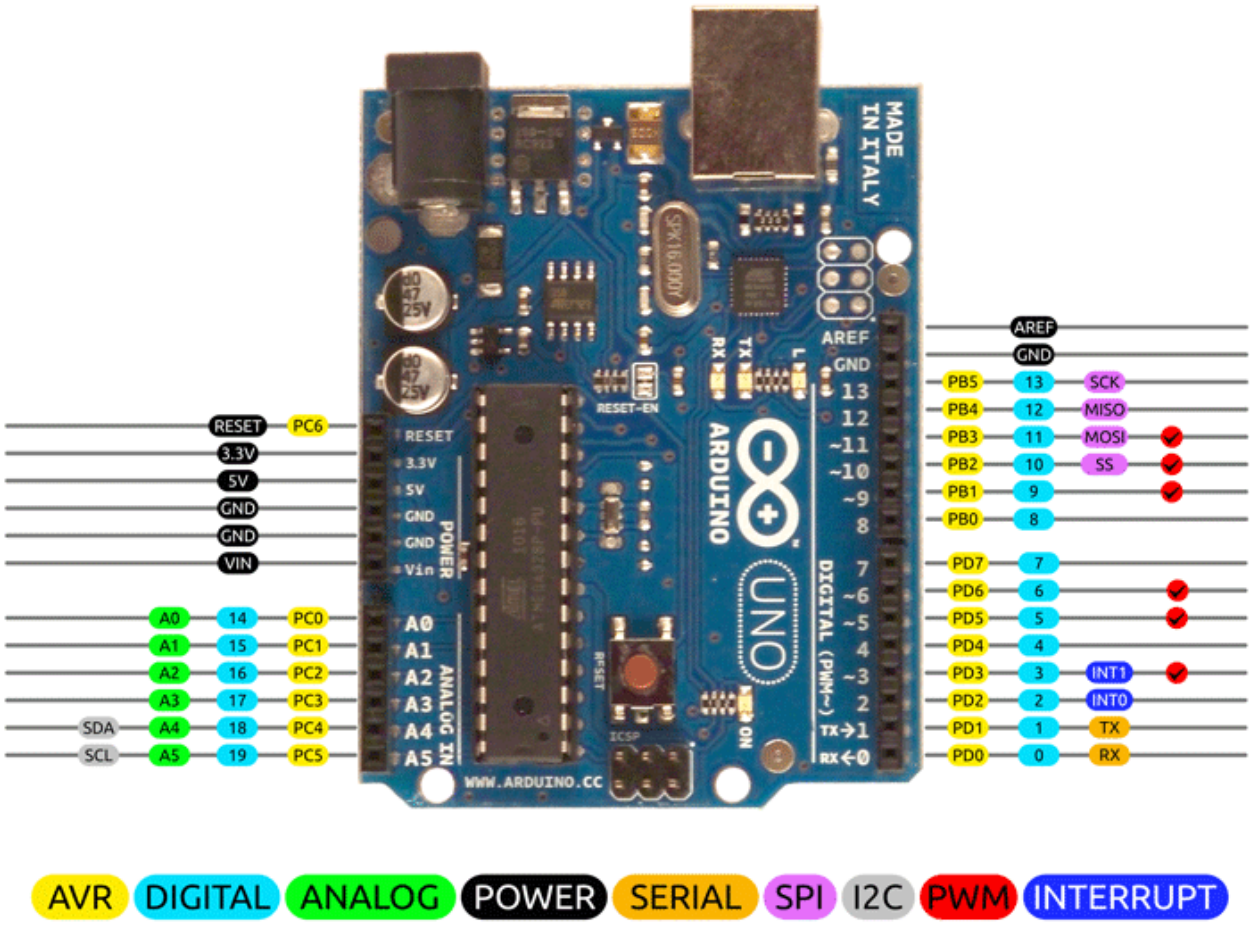
Let us take a closer look at the microcontroller:



Actually let us take an even closer look right above the ‘A’ letter of the ‘Arduino’ text:



You see that ‘L’? Well that L is a tiny LED, a tiny light! But how can we access it? Well below is a nice clear image that shows us what each pin/location is:



From that image although, admittedly maybe not as clearly as we would hope, we can see that L lies on PB5 also 13 also SCK. Although these names refer to the same location, how we user the location dictates which naming we want to use. We will be calling it PB5 or Port B pin 5. To make this light glow we need to do 2 things, first we have to make the pin writable, so make the data direction flow be towards the pin and then we have to write into the pin.

Let us start first by setting the pin to write mode. On the ATmega328P documentation page 72 we will find the PORTB and DDRB registers:

13.4.2 PORTB – The Port B Data Register

Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
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	

13.4.3 DDRB – The Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x04 (0x24)	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
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	

As we can see they exist at memory location 0x25 and 0x24 respectively if we look at the number inside the parenthesis. The number before the parenthesis is the registers offset from the 32 general purpose registers. We use the number outside the parentheses when we use instructions that require I/O registers.

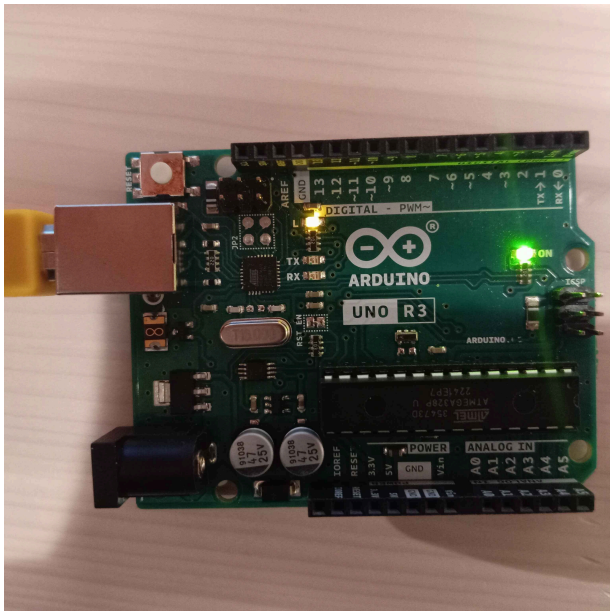
As the name Data Direction Register implies, this register controls the direction of the flow of data of the PORT B pins. In order for us to turn the light on, we have to first set DDB5 high (put a 1 in that bits location) and after that we can set PORTB5 high (put a 1 in that bits location) .The following assembly code (Lesson00/assembly/main.asm) does exactly that:

NOTE: even though we know that registers and everything else on the ATmega328P is memory mapped, we still respect the AVR instruction set and use naming R0 - R31 for all instructions that take registers to help the readers. We also name our I/O registers as per ATmega328p documentation.

```
1  .equ DDRB , 0x04          ; define the DDRB address as per ATmega328P docs
2  .equ PORTB , 0x05         ; define the PORTB address as per ATmega328P docs
3  .equ r16 , 0x10           ; define the register we want to use
4  .equ r15 , 0x0F           ; define the register we want to use
5  .org 0x0000      ; Set the origin to the zero
6
7  SET_PIN_HIGH:
8          ldi     r16, 0x20        ; Load (1 << 5)  into r16 to set DDB5
9          in      r15, DDRB        ; Store current DDRB state
10         ori     r16, r15         ; Do binary or to set DDRB5 bit high
11         out     DDRB, r16        ; Set DDB5 high to make it write pin
12         ldi     r16, 0x20        ; Load (1 << 5)  into r16 to set DDB5
13         in      r15, PORTB       ; Store current PORTB state
14         ori     r16, r15         ; Do binary or to set PORTB5 bit high
15         out     PORTB, r16   ; Set PORTB5 high to turn the LED on
16 MAIN_LOOP:
17         rjmp MAIN_LOOP   ; Make an infinite loop like while(1)
18
```

if we run ‘make upload’ using the makefile provided in the Github repo, our board goes from this:

To this:



Amazingly, we just turned on an LED!

In C we achieve the same result by running the following code:

```c
#include <avr/io.h> // includes definitions of I/O registers
#include <stdint.h> // includes definition of uint8_t

void main(void){
        volatile register uint8_t pin5High = 0x20; // (1 << 5)
        // we use binary or to reserve registers other bits

        DDRB |= pin5High; // we set DDRB5 high to make it writable
        PORTB |= pin5High; // we set PORTB5 high
        while (1)
                ;
        return ;
}
```

For the record, in C I am choosing to use <avr/io,h> only to use the useful macros for the IO registers instead of having to create them myself. I can however make them myself if I wanted to as seen below:

```c
//#include <avr/io.h> // includes definitions of I/O registers
#include <stdint.h> // includes definition of uint8_t

uint8_t *DDRB_ptr = (uint8_t *)0x24;
#define DDRB (*DDRB_ptr)
uint8_t *PORTB_ptr = (uint8_t *)0x25;
#define PORTB (*PORTB_ptr)

void main(void){
        volatile register uint8_t pin5High = 0x20; // (1 << 5)
        // we use binary or to reserve registers other bits

        DDRB |= pin5High; // we set DDRB5 high to make it writable
        PORTB |= pin5High; // we set PORTB5 high
        while (1)
                ;
        return ;
}
```

Congratulations you just learned how to make a light go on in baremetal! Let's see what else we can learn to do in the next 3 lessons!

Lesson01: Making the light blink

THEORY:

Timers and counters:

Majority of microcontrollers have integrated chips for time-keeping. The ATmega328P is no exception, it has three ways to count time built into it! If we take a look at the ATmega328P documentation on page 74, we will see the 3 timer modes provided. Timers are super useful as they are extremely accurate. However, we are not going to be using timer… at least for our assembly program… The reason why we are not going to use them is that they easy once you understand how they work (understand interruprts and so on..) but they might be hard enough to demotivate you from continuing forward with this introduction. Instead of timers we will be making our own scuffed timers!

Clock cycles to the rescue:

Remember when in the earlier theory lesson we learned that instructions take a specific amount of cycles? We will be using this knowledge to make a simple timer in assembly! Given 16 000 000 cycles happen per second, if we do some instruction that takes one cycle for 16 000 000 times, by the time we are done, 1 second should have gone by.

We will be looking at the following 3 instructions from the AVR Instruction Manual:

LDI - 1 cycle
DEC - 1 cycle
CP - 1 cycle
BRNE - 1 / 2 cycles
RJMP - 2
RET - 4 / 5 cycles

Using these instructions we can create a loop that should take 500ms (0.5 seconds) with the following code:

```
17 SLEEP_FOR_500ms:
18         ldi     r20, 0x00; 1 cycle
19         ldi     r21, 0x20; 1 cycle
20 _SLEEP_FOR_500ms_main:
21         dec     r21     ; 1 cycle
22         cp      r21, r20; 1 cycle
23         breq    _SLEEP_FOR_500ms_end; 2 cycles
24         ldi     r18, 0xFF; 1 cycle
25 _SLEEP_FOR_500ms_loop1:
26         dec     r18     ; 1 cycle
27         cp      r18, r20; 1 cycle
28         breq    _SLEEP_FOR_500ms_main; 2 cycles
29         ldi     r19, 0xFF; 1 cycle
30 _SLEEP_FOR_500ms_loop2:
31         dec     r19     ; 1 cycle
32         cp      r19, r20; 1 cycle
33         brne    _SLEEP_FOR_500ms_loop2; 2 cycles
34         rjmp    _SLEEP_FOR_500ms_loop1; 2 cycles
35 _SLEEP_FOR_500ms_end:
36         ret
```

Let us examine how we got the 500ms time. You can skip this part and just trust me if you want :P if you do not want to see the maths.

FORMULA START --->
The formula is:

time_in_sec = total_Cycles / m_CPU

total_cycles = loop1_total_cycles

main_total_cycles = 1 + 1 + 31 * (1 + 1 + 2+ loop1_total_cycles)

loop1_total_cycles = 254 * (1 + 1 + 2 + 1 + loop2_total_cycles)

loop2_total_cycles = 254 * (1 + 1 + 2)

No we can start solving backwards:

loop2_total_cycles = 1016

loop1_total_cycles = 254 * (1 + 1 + 2 + 1 + 1016) = 259334

total_cycles = main_total_cycles = 1 + 1 + 31 * (1 + 1 + 2+ 259334) = 8039480

time_in_sec = 8039480 / 16000000 = 0.50s

FORMULA END---->

Now that we have a 500ms timer, we can add it to our code and make the LED blink! First we need to make sure that we set our stack pointer properly. Now that we are going to be making a function call we need to make sure our stack pointer is pointing to the correct place so that when we push the return address to the stack, it is at the correct place. Inspecting page 14 of the ATmega328P documentation we see that:

6.5.1 SPH and SPL – Stack Pointer High and Stack Pointer Low Register

Bit	15	14	13	12	11	10	9	8	
0x3E (0x5E)	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SPH
0x3D (0x5D)	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SPL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	
	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	

And we know from previously (ATmega382p documentation page 74) that RAM is from 0x0100 to 0x08FF so we have to set:

SPH => 0x08
SPL => 0xFF

Our new code (Lesson01/assembly/main.asm) looks like this:

```
.equ DDRB , 0x04        ; define the DDRB address as per ATmega328P docs
.equ PORTB , 0x05       ; define the PORTB address as per ATmega328P docs
.equ SPH , 0x0E         ; define the SPH address as per ATmega328P docs
.equ SPL , 0x3D         ; define the SPH address as per ATmega328P docs
.equ r16 , 0x10         ; define the register we want to use
.equ r15 , 0x0F         ; define the register we want to use
; in AVR-GCC documentation we see that registers R2-R17,R28 and R29 are
; preserved while R18-R27, R30 and R31 are clobbered, that is why I will next
; introduce R18,R19 and R20 so that we can use them when we are calling
; functions without having to preserve them
.equ r18 , 0x12         ; define the register we want to use
.equ r19 , 0x13         ; define the register we want to use
.equ r20 , 0x14         ; define the register we want to use
.equ r21 , 0x14         ; define the register we want to use
.org 0x0000     ; Set the origin to the zero
        rjmp    SETUP
SLEEP_FOR_500ms:
        ldi     r20, 0x00; 1 cycle
        ldi     r21, 0x20; 1 cycle
_SLEEP_FOR_500ms_main:
        dec     r21     ; 1 cycle
        cp      r21, r20; 1 cycle
        breq    _SLEEP_FOR_500ms_end; 2 cycles
        ldi     r18, 0xFF; 1 cycle
_SLEEP_FOR_500ms_loop1:
        dec     r18     ; 1 cycle
        cp      r18, r20; 1 cycle
        breq    _SLEEP_FOR_500ms_main; 2 cycles
        ldi     r19, 0xFF; 1 cycle
_SLEEP_FOR_500ms_loop2:
        dec     r19     ; 1 cycle
        cp      r19, r20; 1 cycle
        brne    _SLEEP_FOR_500ms_loop2; 2 cycles
        rjmp    _SLEEP_FOR_500ms_loop1; 2 cycles
_SLEEP_FOR_500ms_end:
        ret
SETUP:
        ldi     r16, 0x08
        out     SPH, r16        ; set stackpointer high byte to 0x08
        ldi     r16, 0xFF
        out     SPH, r16        ; set stackpointer low byte to 0xFF
        ldi     r16, 0x20       ; Load (1 << 5)  into r16 to set DDB5
        in      r15, DDRB       ; Store current DDRB state
        ori     r16, r15        ; Do binary or to set DDRB5 bit high
        out     DDRB, r16       ; Set DDB5 high to make it write pin
        rjmp    SET_PIN_HIGH    ; we could have left this drop technically
SET_PIN_HIGH:
        ldi     r16, 0x20       ; Load (1 << 5)  into r16 to set PORTB5
        in      r15, PORTB      ; Store current PORTB state
        ori     r16, r15        ; Do binary or to set PORTB5 bit high
        out     PORTB, r16      ; Set PORTB5 high to turn the LED on
        rcall   SLEEP_FOR_500ms ; Sleep for 500 ms
        rjmp    SET_PIN_LOW     ; Set pin low
SET_PIN_LOW:
        ldi     r16, 0x20       ; Load (1 << 5)  into r16 to set PORTB5
        com     r16             ; Get compliment of 0x20 (flip all bits)
        in      r15, PORTB      ; Store current PORTB state
        and     r16, r15        ; Do binary and to set PORTB5 bit low
        out     PORTB, r16      ; Set PORTB5 high to turn the LED off
        rcall   SLEEP_FOR_500ms ; Sleep for 500 ms
        rjmp    SET_PIN_HIGH    ; Set pin high again
```

That is a lot of code but now your arduino ‘L’ light should be flickering every 500ms ! LETS GO!

For the C enjoyers we do the same logic in the following way:

NOTE: the compiler can optimise things away from us, that is why we are heavily using the keyword volatile!

```c
#include <avr/io.h> // includes definitions of I/O registers
#include <stdint.h> // includes definition of uint8_t

void    setDDRB5_high(void){
        volatile register uint8_t bit5High = 0x20; // (1 << 5)
        // we use binary or to reserve registers other bits
        DDRB |= bit5High; // we set DDRB5 high to make it writable
}

void sleep_for_500ms(void){
        volatile register uint8_t r_21 = 0x20;
        while(r_21 > 0){
                volatile register uint8_t r_18 = 0xFF;
                while(r_18 > 0){
                        volatile register uint8_t r_19 = 0xFF;
                        while(r_19 > 0){
                                --r_19;
                        }
                        --r_18;
                }
                --r_21;
        }
        return ;
}

int main(void){
        volatile register uint8_t pin5High = 0x20; // (1 << 5)
        volatile register uint8_t pin5Low = 0xDF; // all bits except 5th high
        setDDRB5_high();
        while (1)
        {
                PORTB |= pin5High; // we set PORTB5 high
                sleep_for_500ms();
                PORTB &= pin5Low; // we set PORTB5 low
                sleep_for_500ms();
        }
        return 0;
}
```

This effectively is the exact same logic as in our assembly program but programmed in C. We use the volatile and register keyword to achieve our desired estimated cycles. Once again, in C the compiler does a lot behind the scenes so we do not have entrile control over the assembly generated!

Lesson02: Time to outside

The last 2 lessons, we learned how to work on our embedded LED , now it is time to spread our wings and go outside, time to work with external peripherals.

THEORY:

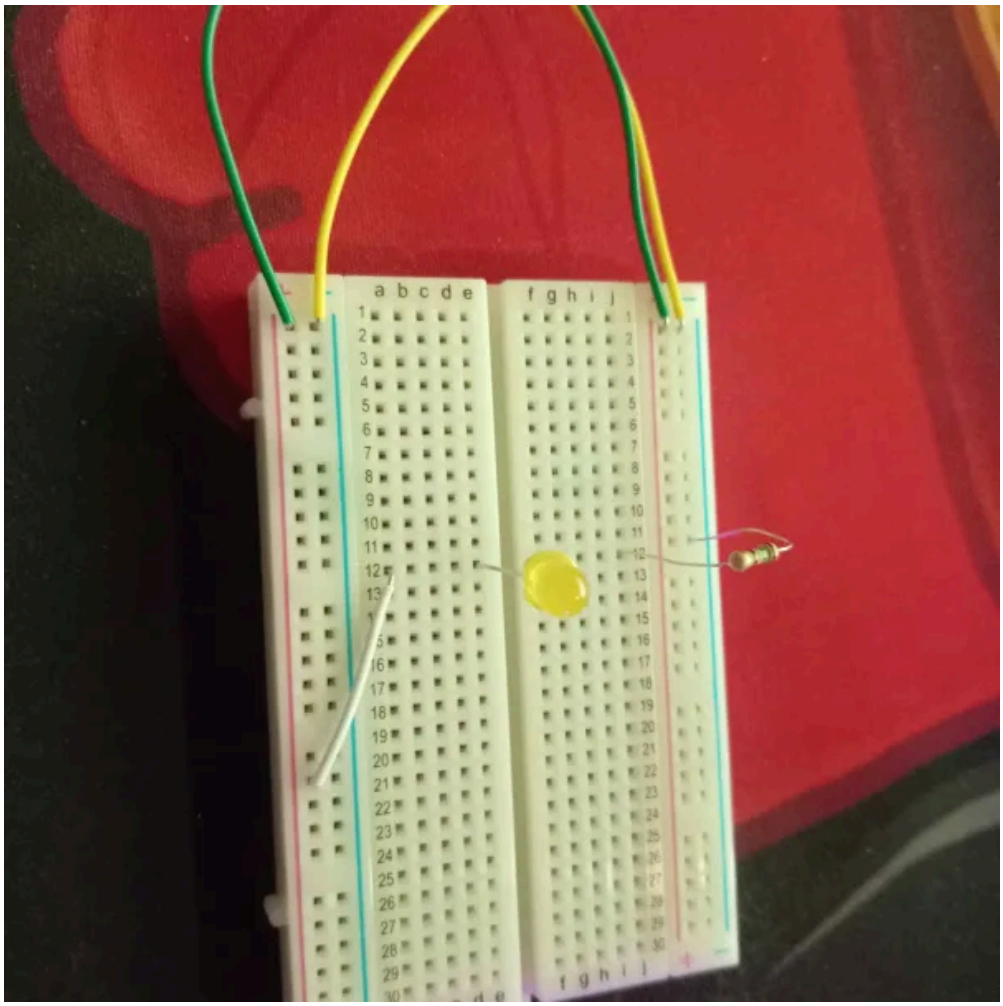
It is all electricity:

We have discussed that the ATmega328P is clocking at 16Mhz. But what does that mean for the devices connected to it. Every cycle that that the ATmega328P has an I/O pin set to high, it receives electricity. 5 volts of it.

LED cannot handle the pressure:

Because LED lights are small and fragile they cannot support 5v of current, that is why we are in need for resistors. Resistors are pretty cool, as they protect our fragile electronics from harmful amounts of electricity. We will not be going into how electronics work in more details, since we want to keep this fun and interesting. But I recommend you do research if you are interested. BenEater (www.youtube.com/BenEater) ia a great source to look at!

In order to follow what I am going to be doing in this lesson you have to set you breadboard to look like this:

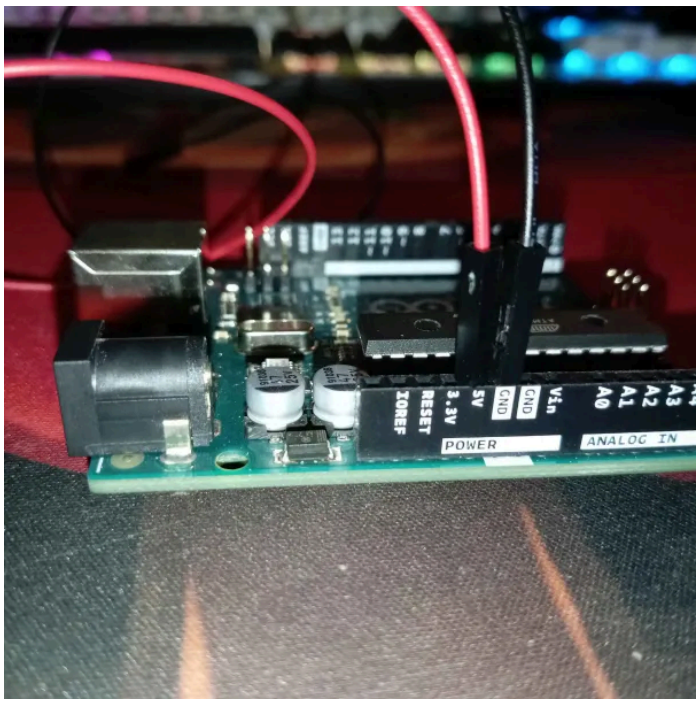


Breakdown of the image:

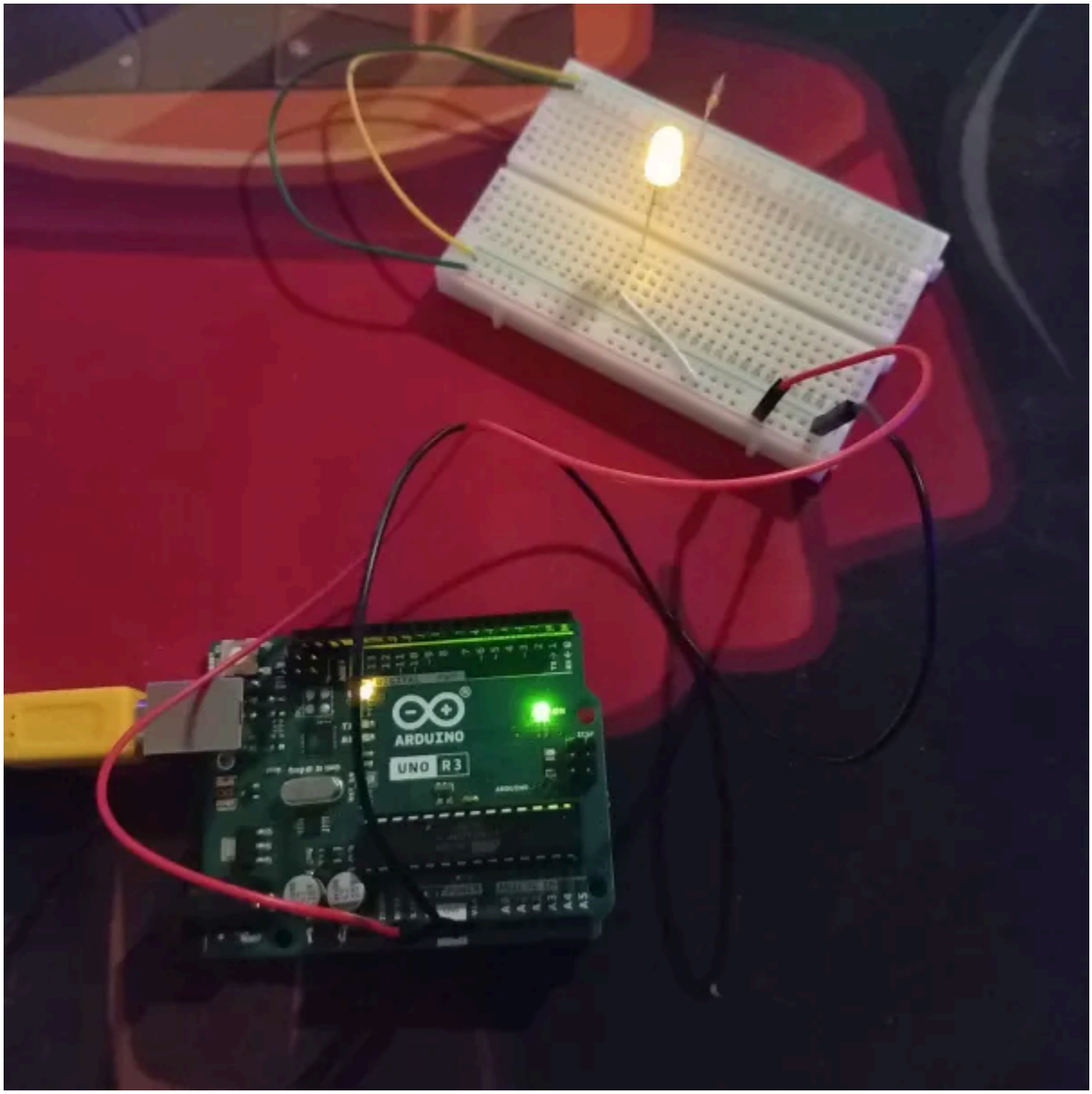
We have a cable connecting the two (+) lines from each side. (Green cable)
We have a cable connecting the two (-) lines from each side. (Yellow cable)
We have a resistor connected from (-) to the 12th rows jth column [12][j]
We have an LED connected LED SHORT LEG at [12][f] and LED LONG LEG [12][e].
We have a small cable connecting [12][a] and (+)

from now on, I will be referring to (-) as GROUND and (+) POWER

Then we are going to hook our jumper cables on our arduino like this:

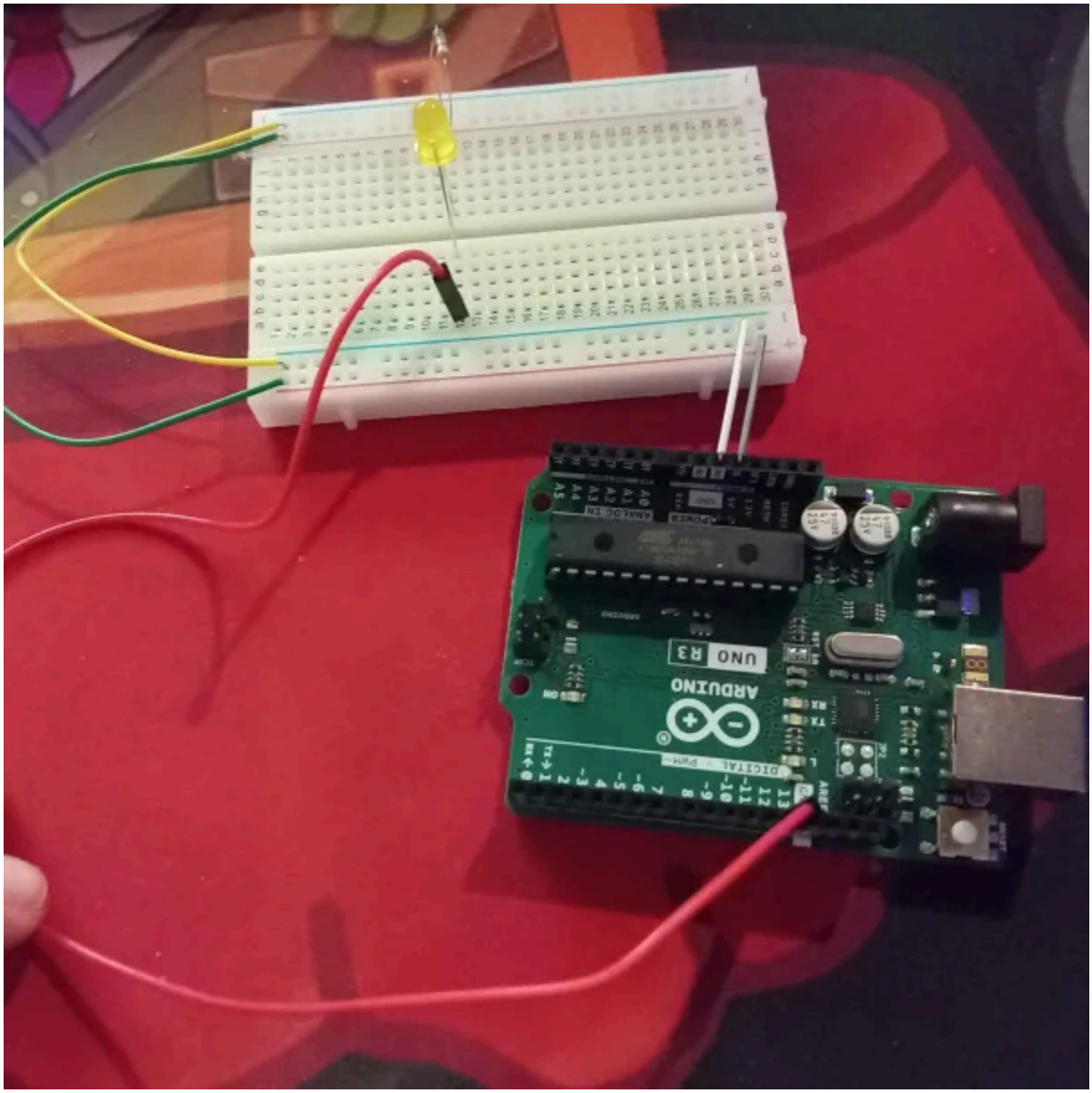


Arduino provides with both 3.3 volt and 5 volt current. We will use the 5 volts. it also provides us with 2 grounds so technically we can use both 3.3 volt line and 5 volt line at the same time. To make sure everything works properly we can connect our 5volts line to the breadboard POWER and the ground to the GROUND. When we boot the connect the arduino with the usb to our computer, we should see the LED power on:



if the LED does not turn on make sure all your connections are properly inserted and also test the with another LED, the amount of times I have tested with broken LEDs is just sad…

Now that we have everything tested lets swap the arrangement a bit:



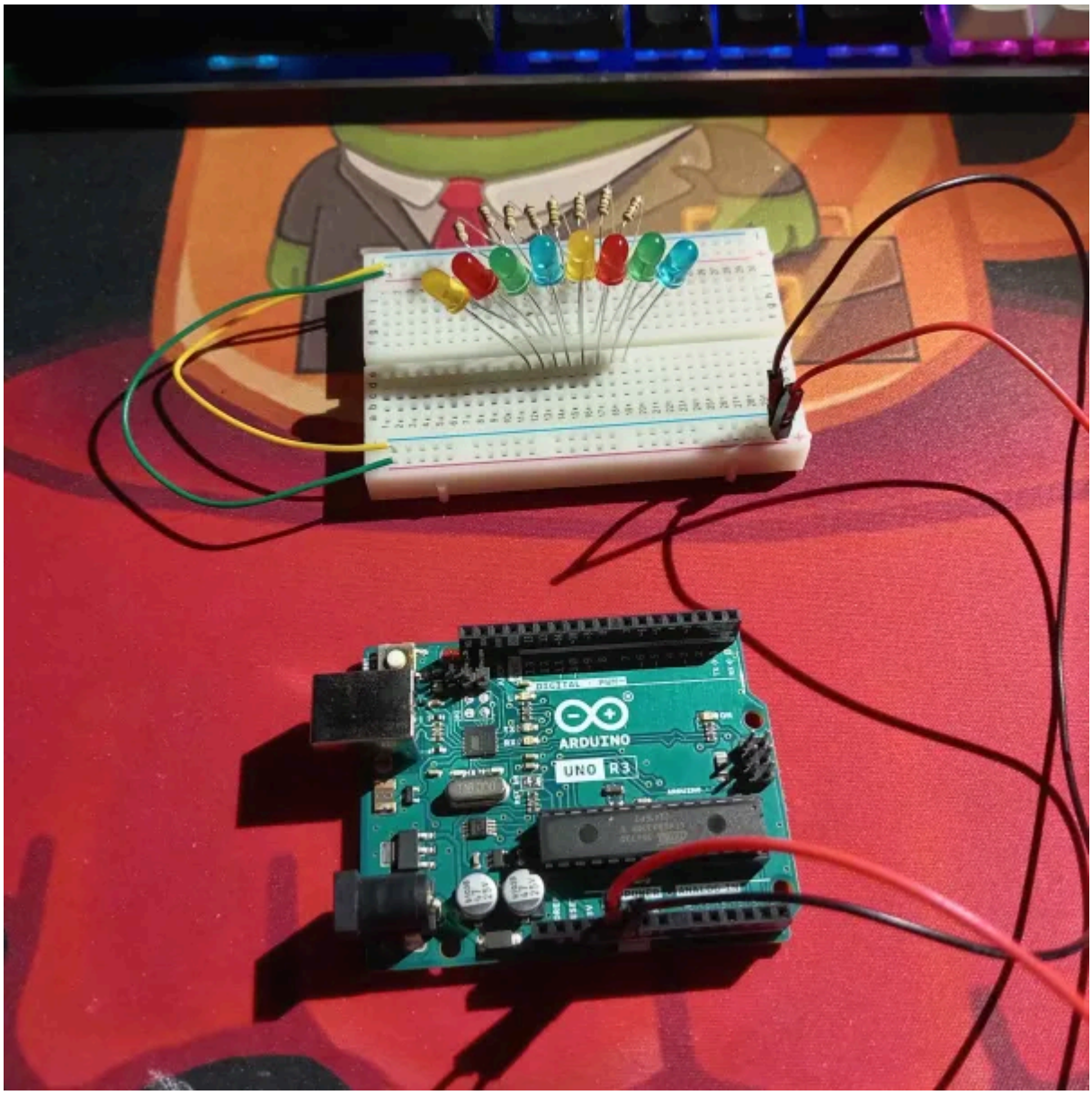
we used smaller wires to connect the ground to GROUND and 5v to POWER.
We removed the small wire connecting the LED to POWER.
We connected a jumper wire from pin13 of the Arduino to the [12][a] pin on the board.

Now we can run the same code as Lesson01, and our blinking light should be blinking alongside the LED on the board!!!

Lesson03: Time for a ‘Hello world’

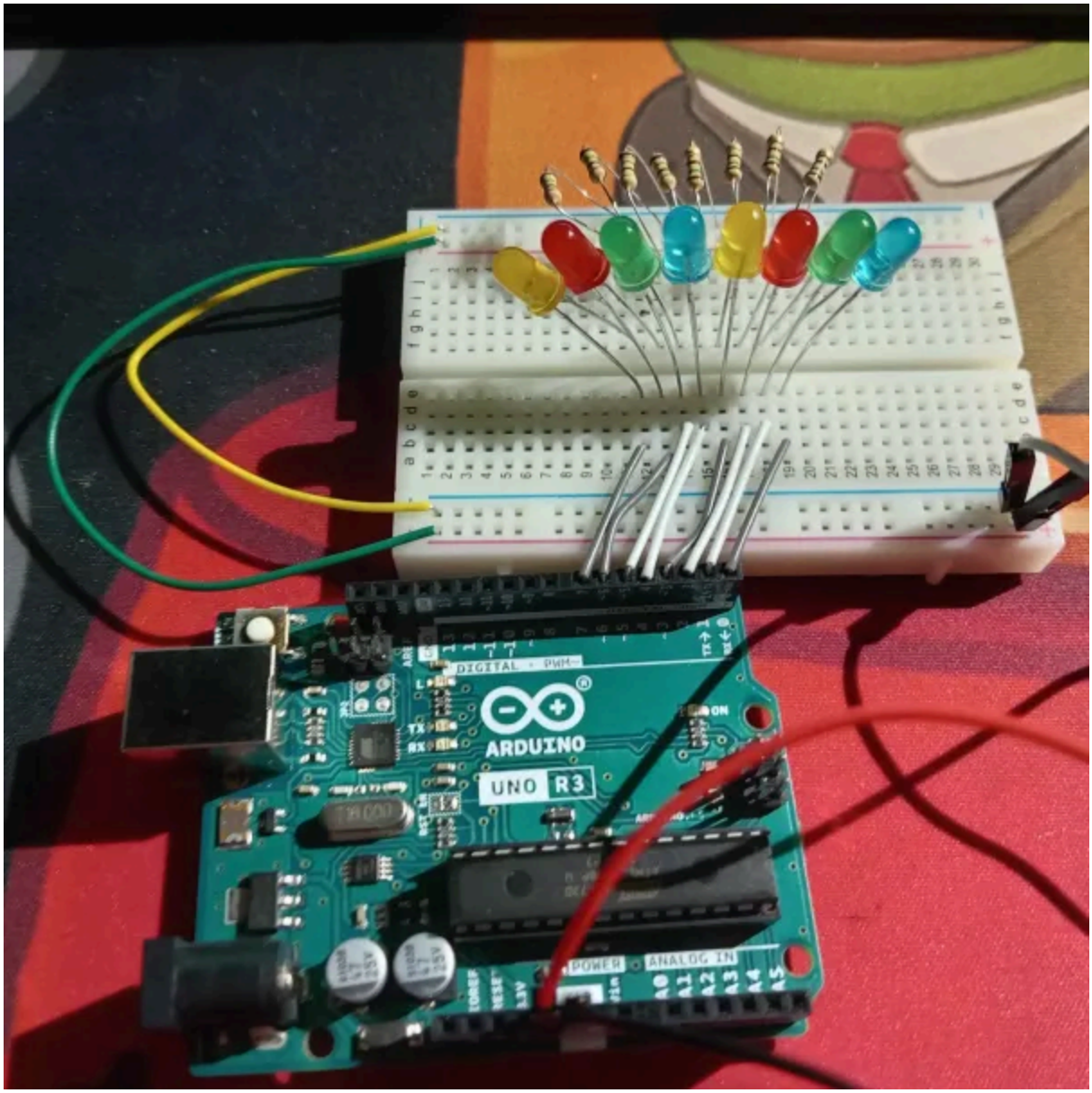
Now we are finally ready to write a proper ‘hello world’ with baremetal!

we first connect 8 LEDs to our board like this:



We basically added 7 more LED next to our previous LED so now we have 8 LEDs and their resistors from rows 12 to 19.

Next what we are going to do is we are going to connect from [12] to [19] all the LED rows to our Arduino to our PORTD like this:



Now, originally I wanted to make a hello world by having the 8 LEDs blink hello world in binary since ascii characters are 8 bits each then it should be possible and I encourage you do that if you have single colour LEDs. However because I have 4 uniques colors, I am going to make them blink one color at a time.

So like before, we have to follow the 3 rules:

Set PORTD Direction Register to OUT, since we are going to be writing the pins.
Figure out what pins each rotation should have lit on, and close the rest.
Set the PORTD to that configuration.

So let's get coding! Our code files are getting a bit bigger so I am going to break them down in pictures for the this and next lesson. First we are going to look at our assembly (Lesson03/assembly/main.asm)

We are going to look at the ATmega328P page 73:

13.4.8 PORTD – The Port D Data Register

Bit	7	6	5	4	3	2	1	0	
0x0B (0x2B)	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	

13.4.9 DDRD – The Port D Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x0A (0x2A)	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	

Hence we are going to add the following to our main.asm:

```
1 .equ DDRD , 0x0A        ; define the DDRB address as per ATmega328P docs
2 .equ PORTD , 0x0B       ; define the PORTB address as per ATmega328P docs
```

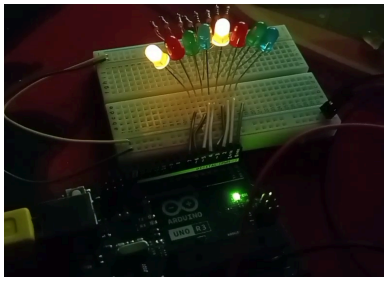
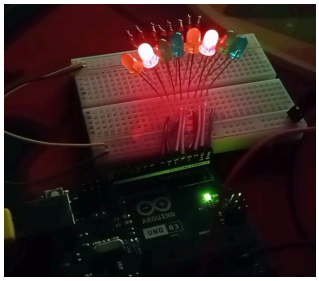
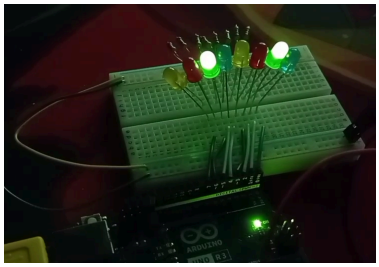
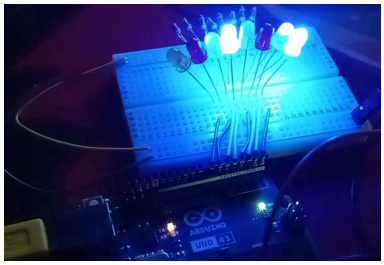
Then in our SETUP we are going to set all bits of DDRD high since we are going to use all of them:

```
39 SETUP:
40          ldi     r16, 0x08
41          out     SPH, r16        ; set stackpointer high byte to 0x08
42          ldi     r16, 0xFF
43          out     SPH, r16        ; set stackpointer low byte to 0xFF
44          ldi     r16, 0xFF       ; Load 255 into r16 to set all bits of DDRD
45          out     DDRD, r16       ; Set DDB5 high to make it write pin
46          rjmp    SET_YELLOW_PINS_ON; we could have left this drop technically
```

Then we make a small loop to loop through the colours:

```
47 SET_YELLOW_PINS_ON:
48          ldi     r16, 0x88       ; Load 0b10001000 to r16 (yellow pins)
49          out     PORTD, r16      ; Set PORTD yellow LEDS on
50          rcall   SLEEP_FOR_500ms ; Sleep for 500 ms
51 SET_RED_PINS:
52          ldi     r16, 0x44       ; Load 0b01000100 to r16 (red pins)
53          out     PORTD, r16      ; Set PORTD red LEDS on
54          rcall   SLEEP_FOR_500ms ; Sleep for 500 ms
55 SET_GREEN_PINS_ON:
56          ldi     r16, 0x22       ; Load 0b00100010 to r16 (green pins)
57          out     PORTD, r16      ; Set PORTD green LEDS on
58          rcall   SLEEP_FOR_500ms ; Sleep for 500 ms
59 SET_BLUE_PINS_ON:
60          ldi     r16, 0x11       ; Load 0b00010001 to r16 (blue pins)
61          out     PORTD, r16      ; Set PORTD blue LEDS on
62          rcall   SLEEP_FOR_500ms ; Sleep for 500 ms
63          rjmp    SET_YELLOW_PINS_ON; Set pin low
64
```

Compile and upload to your board an be amazed as the lights switch on in a sequence:

Simply beautiful!

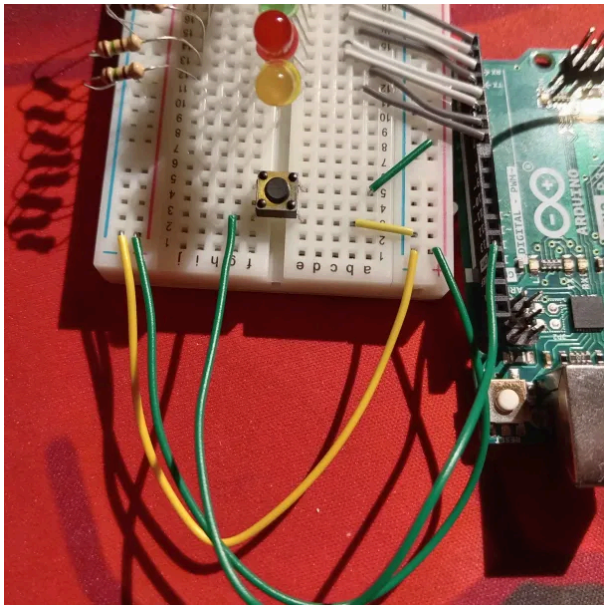
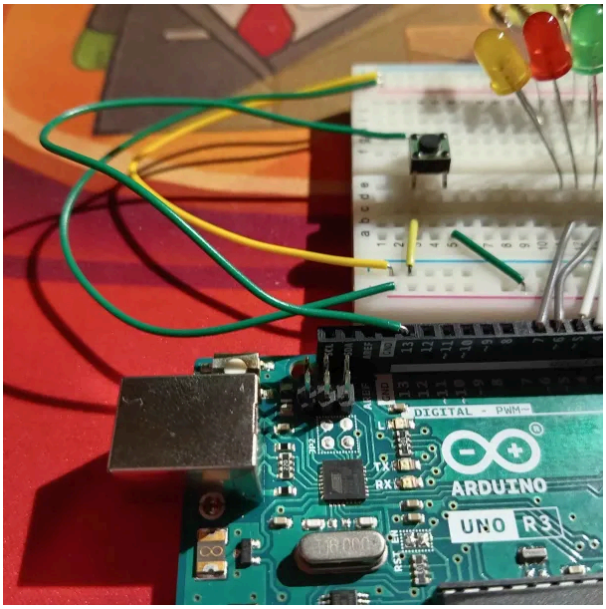
The same code in C looks like this:

```c
#include <avr/io.h> // includes definitions of I/O registers
#include <stdint.h> // includes definition of uint8_t

void    setDDRD_high(void){
        volatile register uint8_t bit5High = 0xFF; // all bits high
        DDRD |= bit5High; // we set DDRD high to make it writable
}

void sleep_for_500ms(void){
        volatile register uint8_t r_21 = 0x20;
        while(r_21 > 0){
                volatile register uint8_t r_18 = 0xFF;
                while(r_18 > 0){
                        volatile register uint8_t r_19 = 0xFF;
                        while(r_19 > 0){
                                --r_19;
                        }
                        --r_18;
                }
                --r_21;
        }
        return ;
}

int main(void){
        volatile register uint8_t colorYellow = 0x88; // 10001000
        volatile register uint8_t colorRed = 0x44; // 01000100
        volatile register uint8_t colorGreen = 0x22; // 00100010
        volatile register uint8_t colorBlue = 0x11; // 00010001
        setDDRD_high();
        while (1)
        {
                PORTD = colorYellow; // we set PORTB5 high
                sleep_for_500ms();
                PORTD = colorRed; // we set PORTB5 low
                sleep_for_500ms();
                PORTD = colorGreen; // we set PORTB5 low
                sleep_for_500ms();
                PORTD = colorBlue; // we set PORTB5 low
                sleep_for_500ms();
        }
        return 0;
}

```

Yeah so much shorter.. I know… that's why people program in C and not in assembly…

Lesson04: Getting input

For the last lesson, we worked with sending signal, now I want to show you as the last lesson, how to get data in. For the final lesson we are going to plug the button to your breadboard like this:

So we have now attache button on [3][e] and [3][f], [5][e] and [5][f]. Then we connected [5][e] to POWER and [3][e] to GROUND.

The purpose of the lesson is to learn how read from the button. For this we have attached [e][f] to our trusted PORTB5 <3

We are gonna write a program that when the arduino starts, it reads the pin, then sleeps 500 ms, then checks the pin again, and if the pin state changed, it starts our LED hellow world we wrote in Lesson03!

In assembly it will look like this:

In order to read we get from the ATmega328P documentation the address of PINB

13.4.2 PORTB – The Port B Data Register

Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
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	

13.4.3 DDRB – The Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x04 (0x24)	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
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	

13.4.4 PINB – The Port B Input Pins Address

Bit	7	6	5	4	3	2	1	0	
0x03 (0x23)	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
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	

and put it in our assembly:

```
5 .equ PINB , 0x03
```

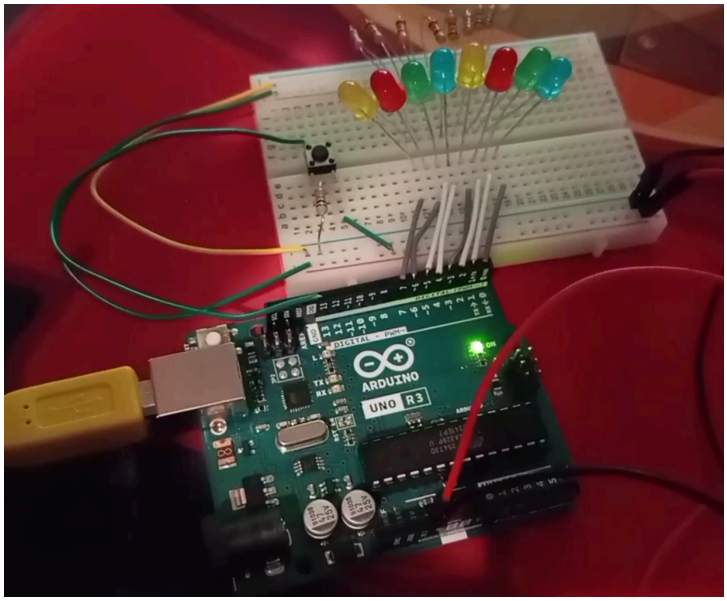
then we update our setup to look like this:

```
40 SETUP:
41          ldi     r16, 0x08
42          out     SPH, r16        ; set stackpointer high byte to 0x08
43          ldi     r16, 0xFF
44          out     SPH, r16        ; set stackpointer low byte to 0xFF
45          ldi     r16, 0xFF       ; Load 255 into r16 to set all bits of DDRD
46          out     DDRD, r16       ; Set DDB5 high to make it write pin
47          ldi     r16, 0x0        ; set DDRB low
48          out     DDRB, r16       ; now we can read from PINB
49          ; we are going to sleep at the begining to wait for the whole system
50          ; to be ready
51          rcall   SLEEP_FOR_500ms
52          ; then we are gonna get the initial PORTB5 state
53          in      r16, PINB
54          bst     r16, 0x05       ; store PORTB5
55 _wait_for_input:
56          rcall   SLEEP_FOR_500ms ; sleep half a second
57          in      r15, PINB       ; get PORTB5
58          bst     r15, 0x05       ; get PORTB5 state
59          cp      r15, r16        ; compare previous state with new state
60          brne    SET_YELLOW_PINS_ON; if change occured jump to the switching
61          rjmp    _wait_for_input ; else sleep and wait again
```

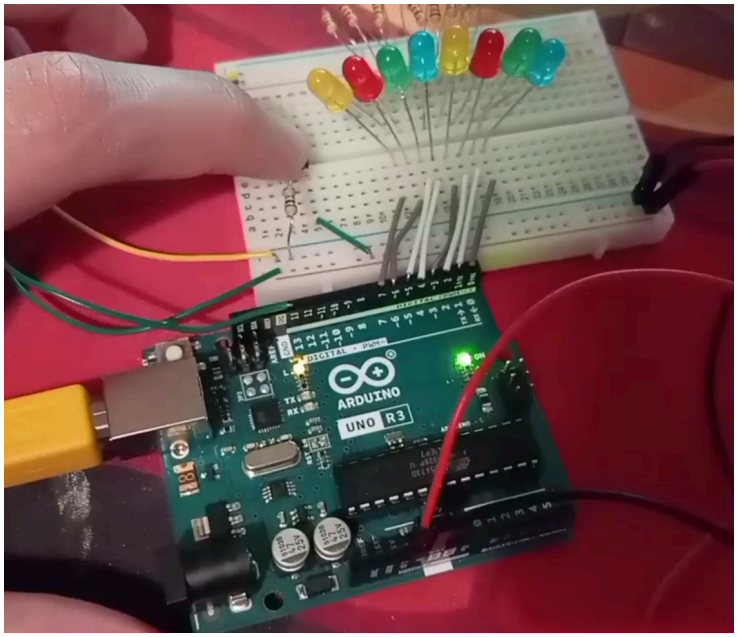
and now we are ready!

when I upload the code to the board the LEDs are turned off. Only when I click down on the switch for a bit does the switching start:

before I press the switch:

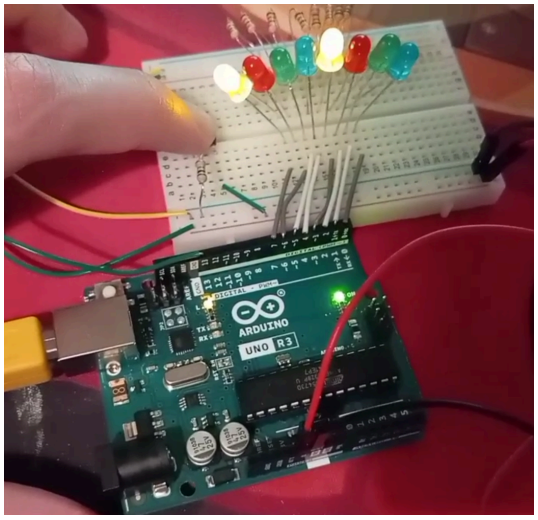


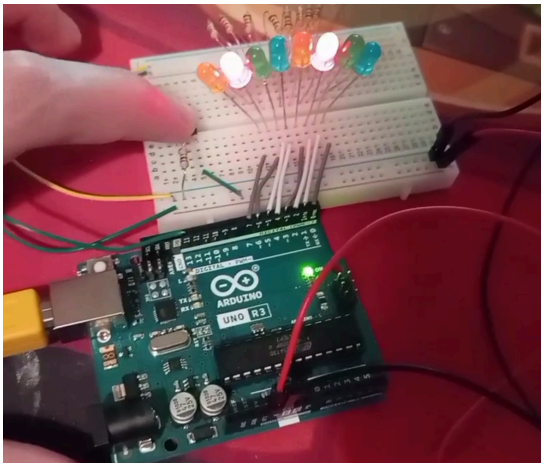
I press the switch down:

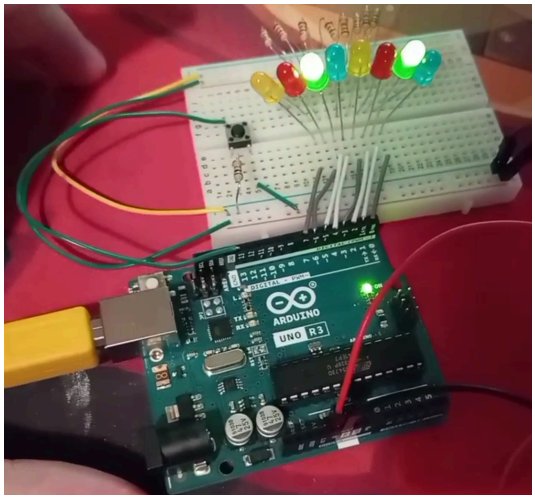


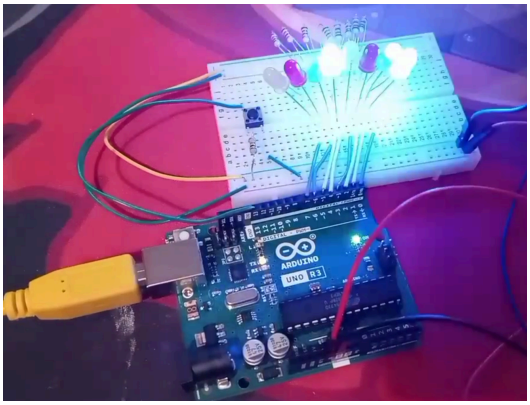
We can observe the switch is pressed because ‘L’ led turns on!

then the lightswtiching starts again:









In C the same code looks like this:

```c
#include <avr/io.h> // includes definitions of I/O registers
#include <stdint.h> // includes definition of uint8_t

void    setDDRD_high(void){
        volatile register uint8_t bit5High = 0x FF; // all bits high
        DDRD |= bit5High; // we set DDRD high to make it writable
}

void    setDDRB_low(void){
        volatile register uint8_t bitsLow = 0x0; // all bits low
        DDRB = bitsLow; // we set DDRD low to make it readable
}

void sleep_for_500ms(void){
        volatile register uint8_t r_21 = 0x20;
        while(r_21 > 0){
                volatile register uint8_t r_18 = 0x FF;
                while(r_18 > 0){
                        volatile register uint8_t r_19 = 0x FF;
                        while(r_19 > 0){
                                --r_19;
                        }
                        --r_18;
                }
                --r_21;
        }
        return ;
}

int main(void){
        volatile register uint8_t colorYellow = 0x88; // 10001000
        volatile register uint8_t colorRed = 0x44; // 01000100
        volatile register uint8_t colorGreen = 0x22; // 00100010
        volatile register uint8_t colorBlue = 0x11; // 00010001
        setDDRD_high();
        setDDRB_low();
        sleep_for_500ms();
        volatile register uint8_t previousState = PINB & 0x20; //PINB5 previousState
        while(1){
                sleep_for_500ms();
                volatile register uint8_t newState = PINB & 0x20; //PINB5 previousState
                if (previousState != newState)
                        break ;
        }
        while(1)
        {
                PORTD = colorYellow; // we set PORTB5 high
                sleep_for_500ms();
                PORTD = colorRed; // we set PORTB5 low
                sleep_for_500ms();
                PORTD = colorGreen; // we set PORTB5 low
                sleep_for_500ms();
                PORTD = colorBlue; // we set PORTB5 low
                sleep_for_500ms();
        }
        return 0;
}
```

AND THAT IS IT. This was a very simple introduction to baremetal programming. Hopefully this inspired you and motivated you to want to learn more!

Let me know if you wish me to make more guides by contacting me on any of my socials:

Linkedin: https://www.linkedin.com/in/michail-nektarios-karatzidis-5282862a8/

Youtube: https://www.youtube.com/@CodingWithDox
old-school-email: mike.karat@yahoo.com