

The Arduino Mega 2560 uses a microcontroller known as the ATmega 2560. The ATmega is built on the AVR architecture.

Arduino provides a single header file <Arduino.h> which gives you access to the GPIO, timers, serial communication and more. There are also many libraries available, which makes Arduino a good choice for rapid prototyping.

However, relying only on the Arduino header file can put limitations on your program. In particular, it can be difficult to manage precise timing of functions inside your loop.

There is an open source library called AVR libc that can make programming the ATmega much easier. Importantly for us, it allows you access the hardware registers of the microcontroller. If you were going to design an embedded system based on an AVR microcontroller, then AVR libc is probably what you would use.

<https://github.com/avrdudes/avr-libc>

Luckily, <Arduino.h> gives us access to some of the AVR libc headers!

This guide will show you how to use AVR libc on the Arduino Mega 2560 to set timer interrupts for coordinating the different tasks in your program.

In order for me to explain what an interrupt is, I have to start from the beginning...

# Microarchitecture

---

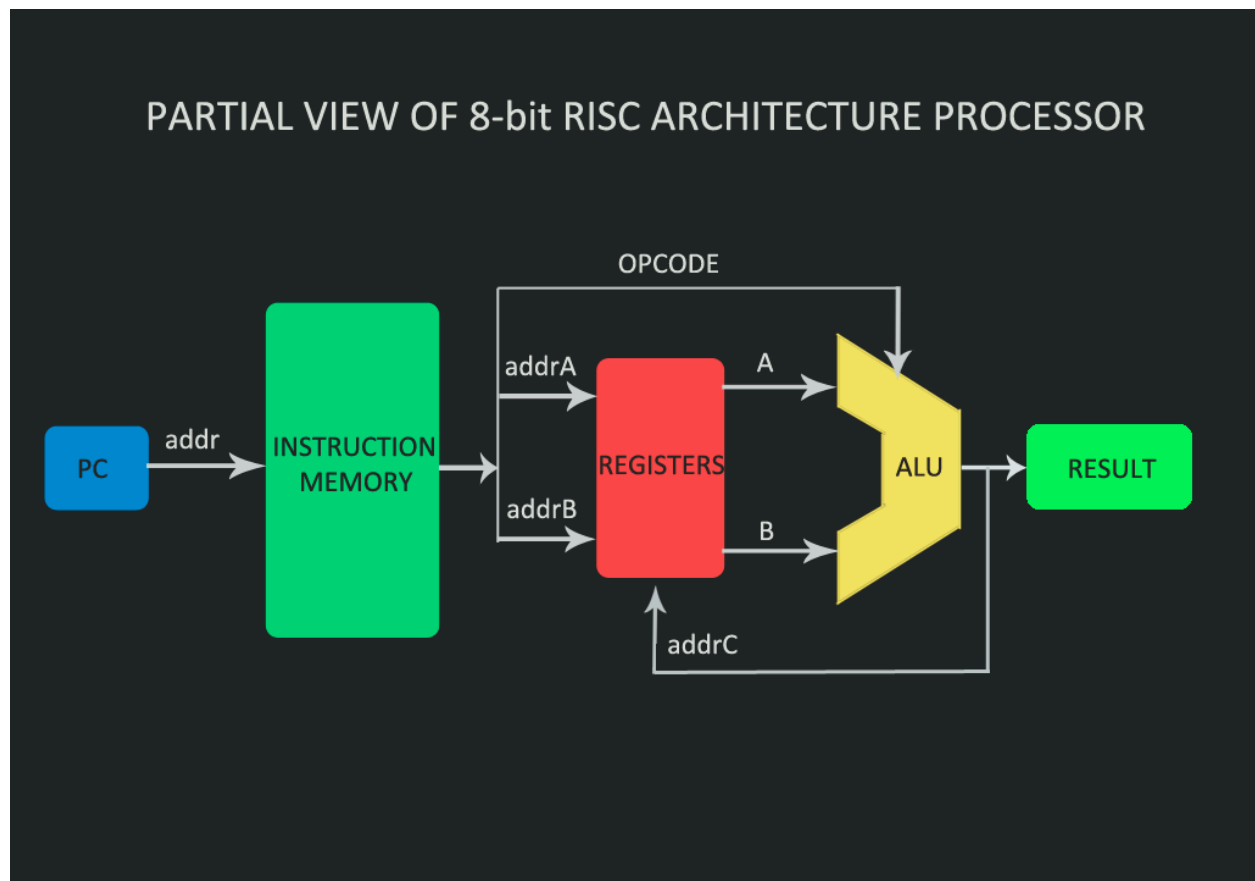
What is a CPU?

The CPU uses an arrangement of digital circuits to execute instructions. Many of these circuits are synchronized to a clock signal, and it typically takes one clock cycle to complete an instruction.

Perhaps the most fundamental component in a CPU is a register. The register is like a chain of memory elements, whose state change is synchronized with the clock. Registers perform many crucial functions for CPU operation. They are used to store data, manipulate operands, and control the flow of instructions.

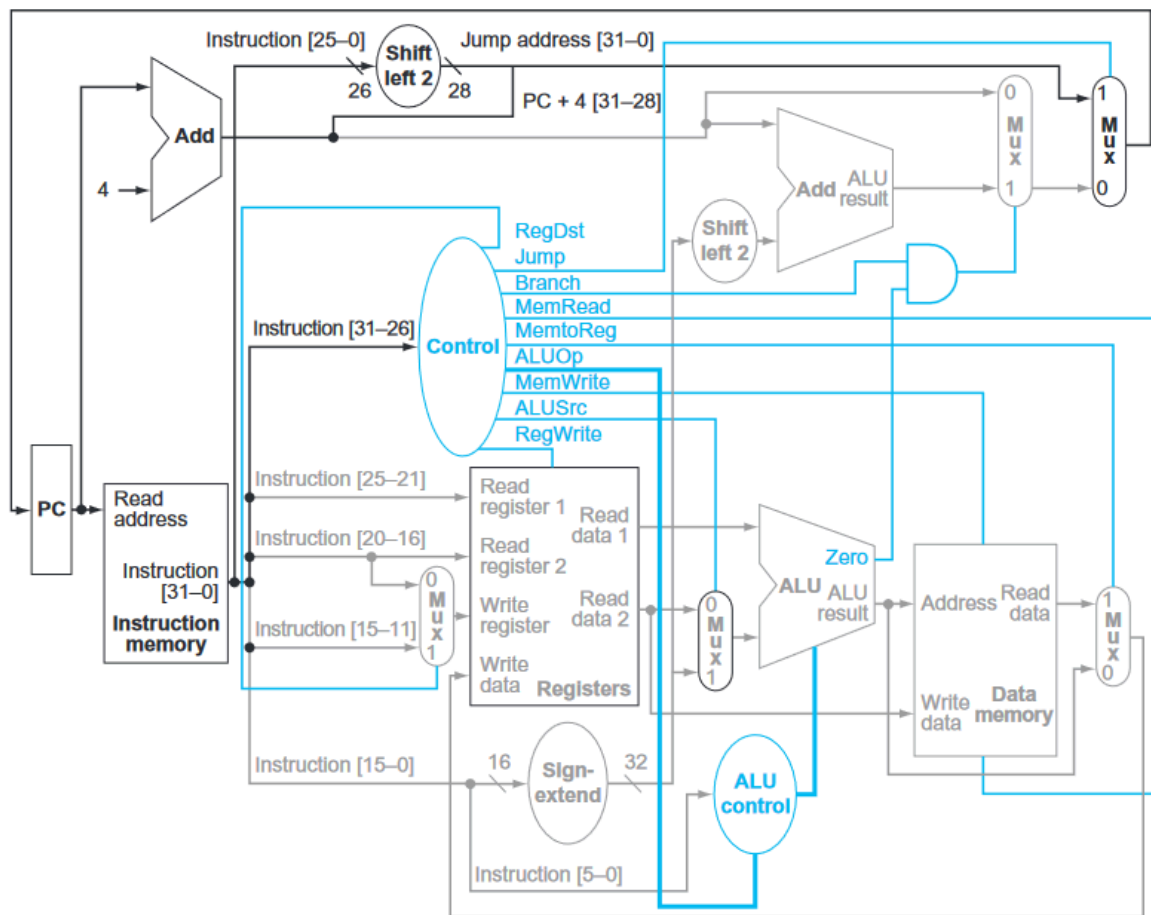
A CPU instruction is a binary number known as machine code. The values of this number's bits are mapped to voltages on wires connected to digital circuits, which can cause those circuits to change their outputs in some way. The human readable form of machine code is known as assembly language.

Here is an overview of some of the main components that make up a CPU.



<https://4.bp.blogspot.com/-kKR5cBljWhY/VSXMrZtgJ9I/AAAAAAAAAcE/dYyeb6uZPk/s1600/procc.png>

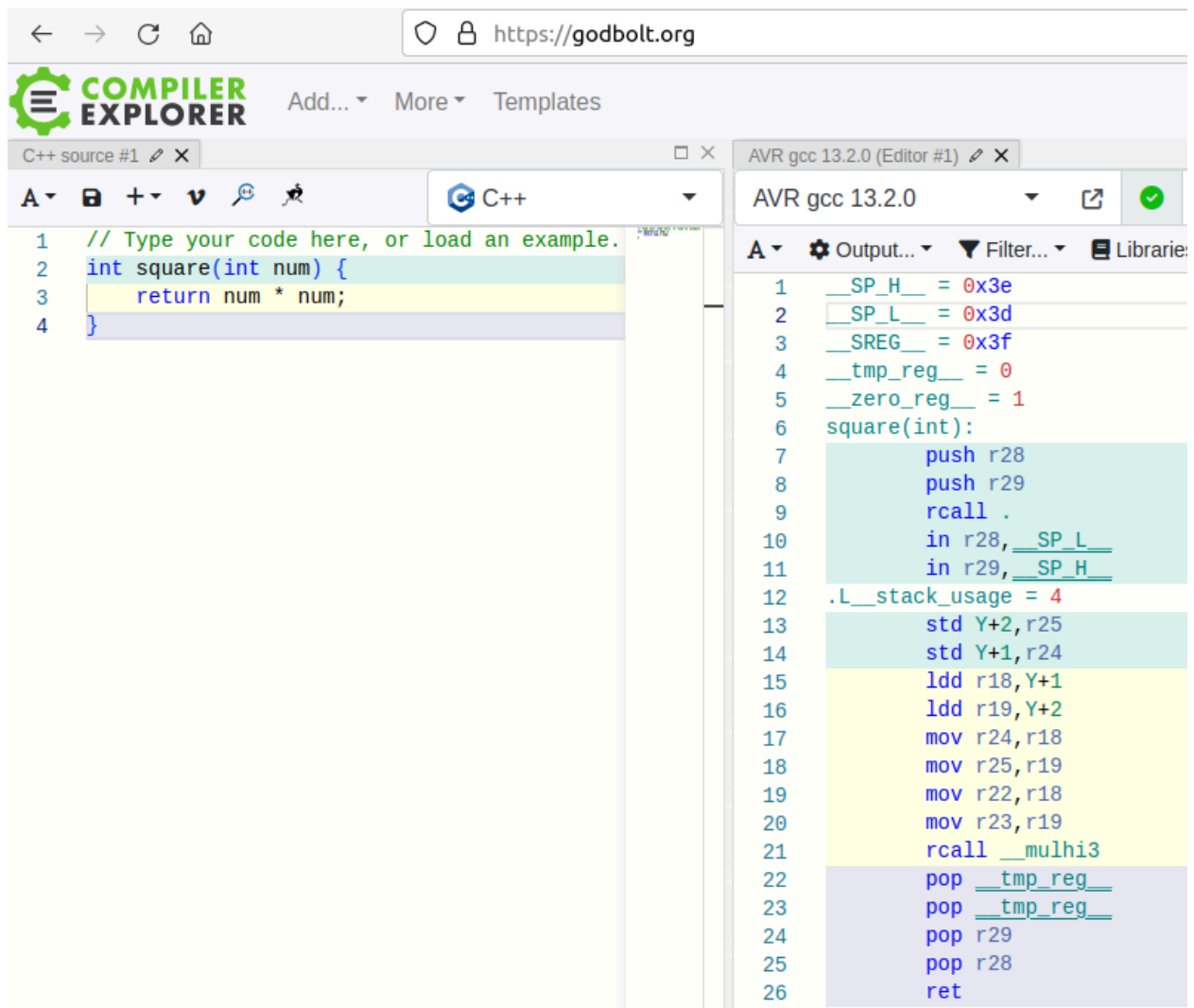
Here is a similar diagram as above, but with some control signals shown. This CPU is arranged in 5 “pipeline” stages: Instruction fetch, Instruction Decode, Execute, Memory, and Write-back.



D. A. Patterson and J. L. Hennessy, *Computer Organization and design: The Hardware/Software Interface*. Elsevier, 2011.

All of the instructions in a running program are stored in instruction memory. The address of the next instruction to be executed is stored in the program counter. After every instruction, the program counter is either incremented to the address of the next instruction, or loaded with some other address.

The program counter might need to change for a function call. When a program is converted into assembly language using a compiler, a function could be converted into a “subroutine”, which will have its own address.



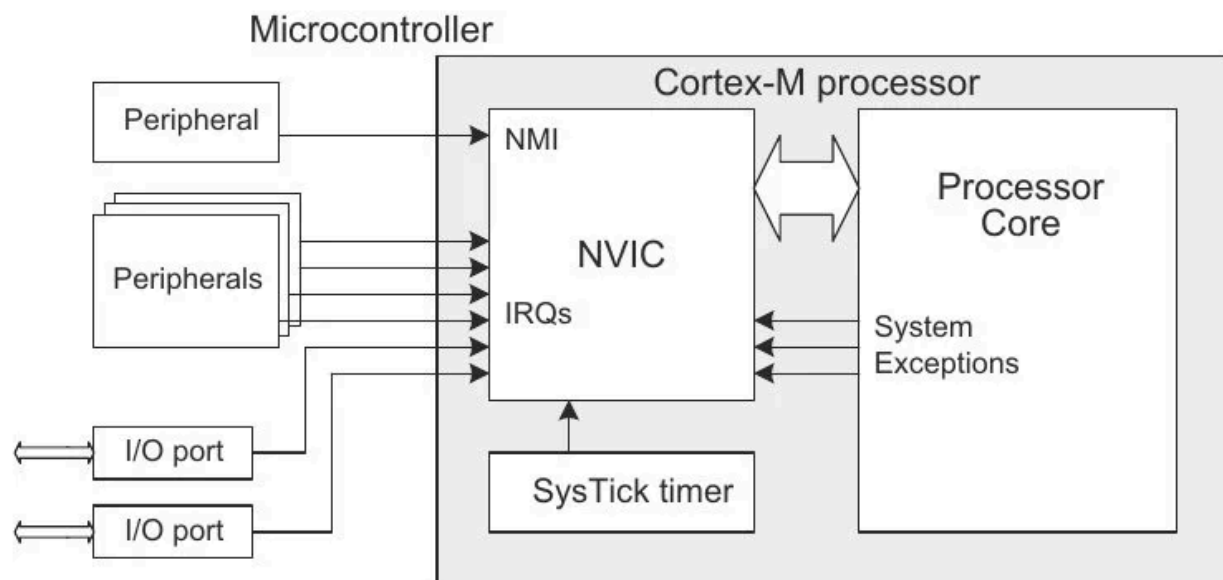
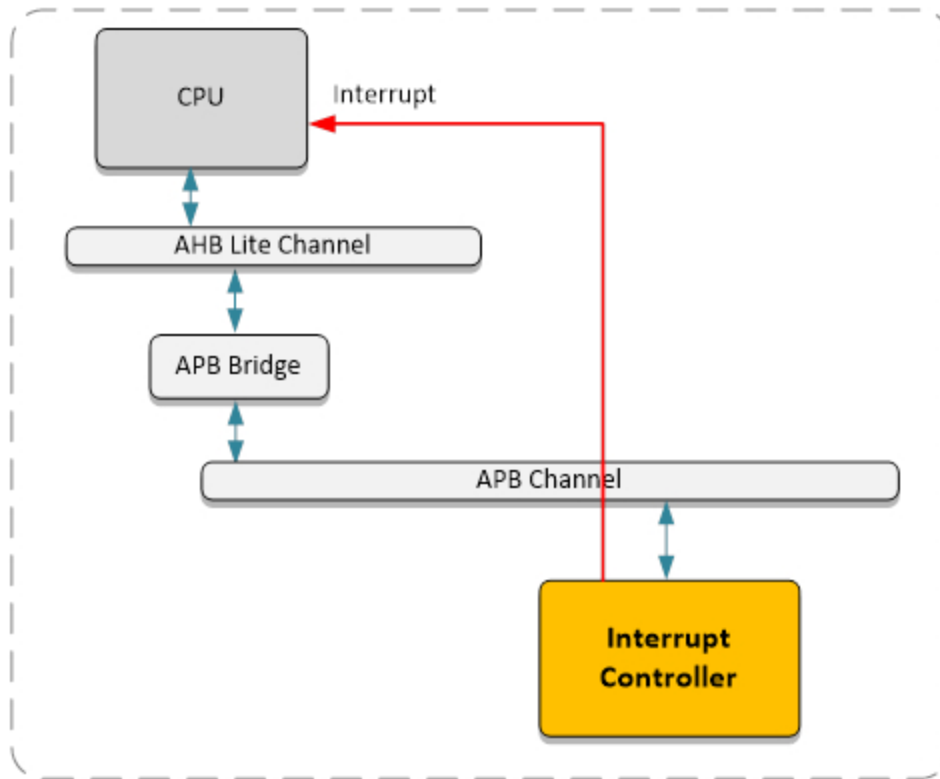
The screenshot displays the Godbolt Compiler Explorer interface. The left pane, titled 'C++ source #1', contains a C function named 'square' that takes an integer 'num' and returns its square. The right pane, titled 'AVR gcc 13.2.0 (Editor #1)', shows the corresponding AVR assembly code. The assembly begins with stack frame setup, including saving the return address and setting up the stack pointer. The function 'square(int)' then pushes the arguments onto the stack, calls the 'mulhi3' instruction to perform the multiplication, and finally pops the arguments and returns the result.

```
1 // Type your code here, or load an example.
2 int square(int num) {
3     return num * num;
4 }

1 __SP_H__ = 0x3e
2 __SP_L__ = 0x3d
3 __SREG__ = 0x3f
4 __tmp_reg__ = 0
5 __zero_reg__ = 1
6 square(int):
7     push r28
8     push r29
9     rcall .
10    in r28, __SP_L__
11    in r29, __SP_H__
12    .L__stack_usage = 4
13    std Y+2, r25
14    std Y+1, r24
15    ldd r18, Y+1
16    ldd r19, Y+2
17    mov r24, r18
18    mov r25, r19
19    mov r22, r18
20    mov r23, r19
21    rcall __mulhi3
22    pop __tmp_reg__
23    pop __tmp_reg__
24    pop r29
25    pop r28
26    ret
```

Left: C function for multiplying two numbers. Right: Same function compiled into a subroutine in AVR assembly.

Another reason the program counter might change is when a CPU interrupt occurs. The interrupt controller causes the program counter to jump to the address of a special subroutine for loading interrupts. This subroutine will save the current CPU context, and load another subroutine. The correct subroutine is loaded by indexing a “vector table” using the interrupt “vector number” based on the interrupt source.

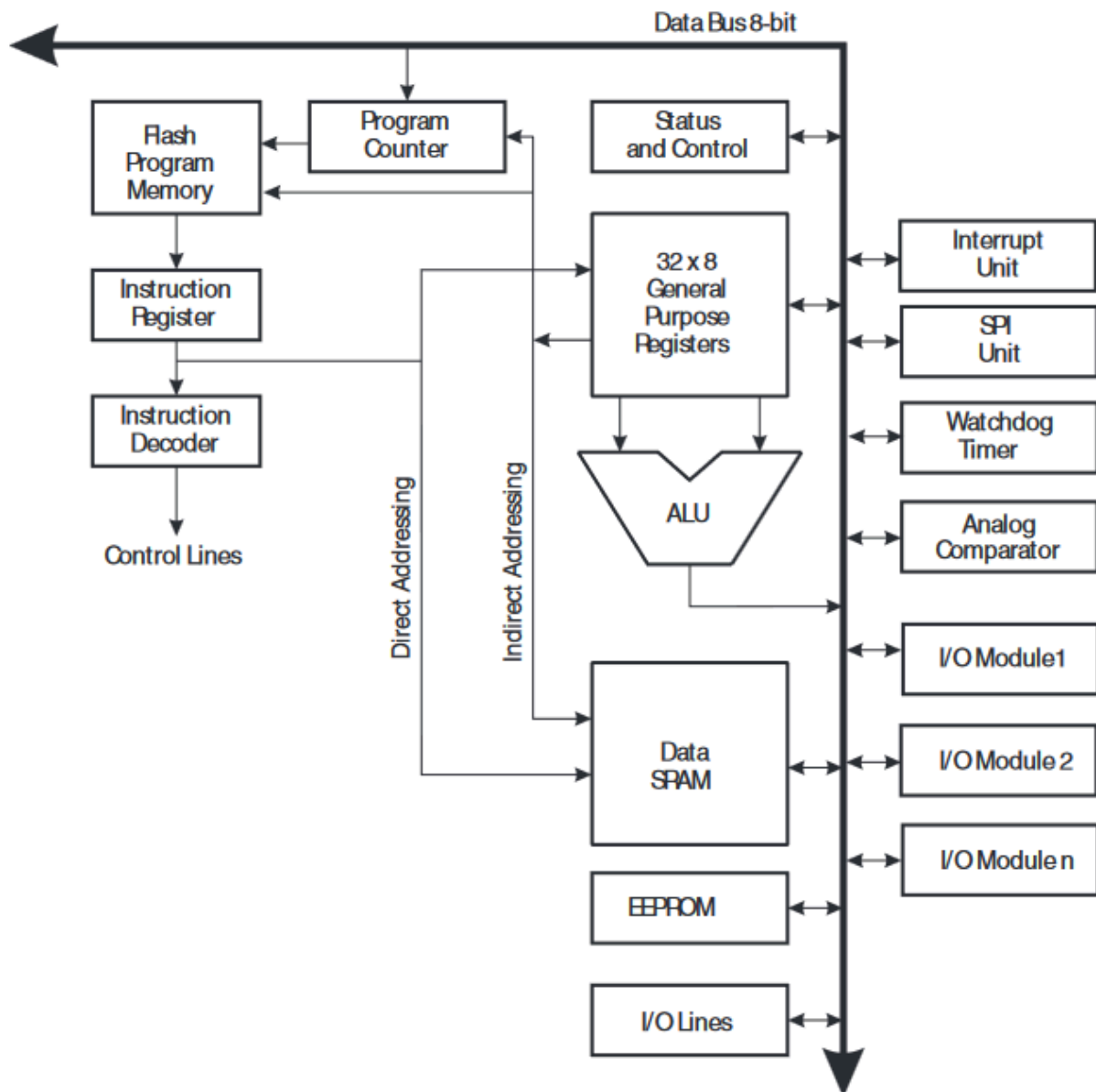


Interrupt controller diagrams for ARM CPUs. Upper: Controller connected to the Advanced Peripheral Bus. Lower: Nested Vector Interrupt Controller (NVIC) for an ARM Cortex-M.

So where do the interrupts come from? Either from a system exception generated by the CPU, or from one of the CPU's "peripherals". The CPU peripherals have their own registers, and on a microcontroller we can read and modify these registers within our programs.

From here on I'll start referencing pages on the ATmega 2560 datasheet. You can download it here and follow along.

[https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561\\_datasheet.pdf](https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf)



**Table 14-1.** Reset and Interrupt Vectors

Vector No.	Program Address <sup>(2)</sup>	Source	Interrupt Definition
1	\$0000 <sup>(1)</sup>	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	\$0002	INT0	External Interrupt Request 0
3	\$0004	INT1	External Interrupt Request 1
4	\$0006	INT2	External Interrupt Request 2
5	\$0008	INT3	External Interrupt Request 3
6	\$000A	INT4	External Interrupt Request 4
7	\$000C	INT5	External Interrupt Request 5
8	\$000E	INT6	External Interrupt Request 6
9	\$0010	INT7	External Interrupt Request 7
10	\$0012	PCINT0	Pin Change Interrupt Request 0
11	\$0014	PCINT1	Pin Change Interrupt Request 1
12	\$0016 <sup>(3)</sup>	PCINT2	Pin Change Interrupt Request 2
13	\$0018	WDT	Watchdog Time-out Interrupt
14	\$001A	TIMER2 COMPA	Timer/Counter2 Compare Match A
15	\$001C	TIMER2 COMPB	Timer/Counter2 Compare Match B
16	\$001E	TIMER2 OVF	Timer/Counter2 Overflow
17	\$0020	TIMER1 CAPT	Timer/Counter1 Capture Event
18	\$0022	TIMER1 COMPA	Timer/Counter1 Compare Match A
19	\$0024	TIMER1 COMPB	Timer/Counter1 Compare Match B
20	\$0026	TIMER1 COMPC	Timer/Counter1 Compare Match C
21	\$0028	TIMER1 OVF	Timer/Counter1 Overflow

Interrupt vectable table for the ATmega 2560

## Timer Interrupts

---

We want to use the timers to set a repeating interrupt in order to generate events with a set frequency. The Timer/Counter1 Compare Match A looks like it might work for setting a timer.

First, let's define an interrupt handle. This function will run whenever the Timer/Counter1 Compare Match A interrupt is raised.

```
ISR(TIMER1_COMPA_vect) {  
    // process the interrupt here  
}
```

The `TIMER1_COMPA_vect` macro is defined with the other interrupt vectors in `<avr/io.h>`

Now let's take a look at the code needed for setting the timer:

```
init_timer() {
    TCCR1A = 0; // set entire TCCR1A register to 0
    TCCR1B = 0; // same for TCCR1B
    TCNT1 = 0; //initialize counter value to 0
    // set compare match register for 1hz increments
    OCR1A = 15624; // = (16*10^6) / (1*1024) - 1 (must be <65536)
    // turn on CTC mode
    TCCR1B |= (1 << WGM12);
    // Set CS10 and CS12 bits for 1024 prescaler
    TCCR1B |= (1 << CS12) | (1 << CS10);
    // enable timer compare interrupt
    TIMSK1 |= (1 << OCIE1A);
}
```

What's going on here?

First, let's read the Output Compare section of the datasheet for Timer 1.

Section 17.7 - Output Compare Units on page 141 tells us that

"The 16-bit comparator continuously compares `TCNTn` with the Output Compare Register (`OCRnx`). If `TCNT` equals `OCRnx` the comparator signals a match. A match will set the Output Compare Flag (`OCFnx`) at the next timer clockcycle. If enabled (`OCIEnx = 1`), the Output Compare Flag generates an Output Compare interrupt"

Output Compare interrupt isn't quite the same as Timer/Counter1 Compare Match A. Let's check what the `OCIE1A` flag is.

In the `TIMSK` register description on page 162 we find the `OCIE1A` bit. This section tells us

"When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter Output Compare A Match interrupt is enabled."

We now know we need to set the Output Compare register to some value so that when the Timer/Counter matches, an interrupt will be triggered at a certain frequency.



How fast is the timer counting? Let's keep reading.

In the Register description of TCCR1B at the top of page 157, this table shows us how the Clock Select bits are used for selecting the clock source of the timer.

**Table 17-6.** Clock Select Bit Description

CSn2	CSn1	CSn0	Description
0	0	0	No clock source. (Timer/Counter stopped)
0	0	1	clk <sub>IO</sub> /1 (No prescaling)
0	1	0	clk <sub>IO</sub> /8 (From prescaler)
0	1	1	clk <sub>IO</sub> /64 (From prescaler)
1	0	0	clk <sub>IO</sub> /256 (From prescaler)
1	0	1	clk <sub>IO</sub> /1024 (From prescaler)
1	1	0	External clock source on Tn pin. Clock on falling edge
1	1	1	External clock source on Tn pin. Clock on rising edge

Page 157

The clock on the Arduino Mega 2560 is 16MHz, according to the Arduino Mega datasheet.

We can use these lines to select the 1024 prescaler.

```
TCCR1B |= (1 << CS12) | (1 << CS10);
```

Suppose we want a frequency of 1Hz. Since we selected the 1024 prescaler, our clock source is 15.625kHz. To achieve a frequency of 1Hz, we would want to trigger the interrupt 15,625 cycles or 1 second. We can then set the value of OCR1A to 15625 - 1 because we are starting the count at 0.

```
OCR1A = 15624; // (16*10^6) / (1*1024) - 1
```

Another way to think about this is to consider that when we wait for the counter to reach a certain value, we are dividing the clock by that value.

If we set the counter to 1, the interrupt would go off with a frequency of our clock source 15.625 kHz. If we set the counter to 2, the frequency would be halved to 7.8kHz. So when we set the Output Compare register to 15625, we are dividing our 15625Hz clock source by itself to get a frequency of 1Hz.

We can identify the count needed for a target frequency with the following formula.

$$\frac{\text{clk source}}{\text{count}} = f$$
$$\text{count} = \frac{\text{clk source}}{f}$$

Remember that our clock source is 16MHz divided by our prescaler.

How do we know the TCNTn register resets every time?

In the register description for TCCR1A section on the top of page 155 is an explanation that the combined WGMn bits from register TCCRxA and TCCRxB sets the mode of operation for the Timer/Counter. Table 17-2 shows the timer mode for each bit combination..

**Table 17-2.** Waveform Generation Mode Bit Description<sup>(1)</sup>

Mode	WGMn3	WGMn2 (CTCn)	WGMn1 (PWMn1)	WGMn0 (PWMn0)	Timer/Counter Mode of Operation	TOP	Update of OCRnx at	TOVn Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCRnA	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP

Page 145

Further down page 145 in section 17.9.2 Clear Timer on Compare Match (CTC), it explains that CTC mode clears the timer counter when TCNT1 matches the value in OCR1A.

That is why we use the line

```
TCCR2A |= (1 << WGM21);
```

## Power Management

---

We might want to reduce the power consumption of our microcontroller while we are waiting for the timer interrupts. We can read about that in section 11 - Power Management and Sleep Modes on page 50.

The table below shows us the different low power modes available on the ATmega 2560. The Idle mode has the least power savings, but will allow us to wake up from our timer interrupts. The Power-save mode will have much lower power consumption, but we can only wake up from Timer2 interrupts.

We also have to be careful about how quickly we wake up after going to sleep, as the overhead from entering sleep mode might outweigh the power reductions during sleep. If we only go to sleep for a very short time, we won't have a chance to save much power.

For our projects today, it might not be worth it at all to enter power-save or even idle mode. We would have to measure the current draw to be sure.

**Table 11-1.** Active Clock Domains and Wake-up Sources in the Different Sleep Modes.

Sleep Mode	Active Clock Domains					Oscillators		Wake-up Sources						
	clk <sub>CPU</sub>	clk <sub>FLASH</sub>	clk <sub>IO</sub>	clk <sub>ADC</sub>	clk <sub>ASY</sub>	Main Clock Source Enabled	Timer Osc Enabled	INT7:0 and Pin Change	TWI Address Match	Timer2	SPM/EEPROM Ready	ADC	WDT Interrupt	Other I/O
Idle			X	X	X	X	X <sup>(2)</sup>	X	X	X	X	X	X	X
ADCNRM				X	X	X	X <sup>(2)</sup>	X <sup>(3)</sup>	X	X <sup>(2)</sup>	X	X	X	
Power-down								X <sup>(3)</sup>	X				X	
Power-save					X		X <sup>(2)</sup>	X <sup>(3)</sup>	X	X			X	
Standby <sup>(1)</sup>						X		X <sup>(3)</sup>	X				X	
Extended Standby					X <sup>(2)</sup>	X	X <sup>(2)</sup>	X <sup>(3)</sup>	X	X			X	

Page 50

The Sleep Mode Control Register SMCR description contains a table showing how to select the mode using the Sleep Mode Bits. The SMCR Sleep Enable bit is used to make the CPU enter sleep mode when the SLEEP instruction is executed.

**Table 11-2.** Sleep Mode Select

SM2	SM1	SM0	Sleep Mode
0	0	0	Idle
0	0	1	ADC Noise Reduction
0	1	0	Power-down
0	1	1	Power-save
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Standby <sup>(1)</sup>
1	1	1	Extended Standby <sup>(1)</sup>

Page 54

To set the Idle sleep mode, we can use

```
void init_idle()
{
    SMCR &= ~(1 << SM2) & ~(1 << SM1) & ~(1 << SM0);
    SMCR |= (1 << SE);
}
```

For power save mode

```
void init_power_save()
{
    SMCR |= (1 << SM1) | (1 << SM0);
    SMCR |= (1 << SE);
}
```

When we are ready to enter the sleep mode, we can use

```
void atmega_sleep()
{
    sleep_enable();
    sleep_cpu();
    sleep_disable();
}
```

Let's go into some examples of how to structure our main loop. Suppose we have three tasks or functions we want to run, possibly at different rates.

The first option is to simply run each function one after the other inside the loop. This is known as **superloop**. We might want to delay at the end of the loop, or put the microcontroller to sleep. This will run each task at the same frequency.

<https://github.com/albertalooop/elegoo-demo/blob/main/superloop/src/main.cpp>

```
loop() {
    task1();
    task2();
    task3();
    atmega_sleep();
}
```

Another approach we could take is to put a switch statement inside the loop to create a small state machine. The state variable is an integer that we increment at the end of each loop, and reset to zero once we move through all the states.

This allows us to run some tasks to run in integer multiple frequencies of each other. Consider the example below. Task 1 runs twice as often as tasks 2 and 3. Let's refer to this approach as **harmonic loop periods**.

<https://github.com/albertaloop/elegoo-demo/blob/main/harmonic%20loop%20periods/src/main.cpp>

```
#define TASK_STATE1 0
#define TASK_STATE2 1
#define TASK_STATE3 2
#define TASK4_STATE 3
#define NUM_STATES 4

void loop() {
    if(timer_ready)
    {
        switch(state)
        {
            case TASK_STATE1:
                task1();
                break;
            case TASK_STATE2:
                task2();
                break;
            case TASK_STATE3:
                task1();
                break;
            case TASK_STATE4:
                task3();
                break;
            default: break;
        }
        state++;

        if(state%(NUM_STATES) == 0) state = TASK_STATE1;
        timer_ready = false;
    }
}
```

In this final example each task will get its own timer interrupt. Instead of setting a boolean in the ISR, we will push a value to a circular buffer. The loop will check each buffer to see if a new value came in. We will call this approach **multiple interrupts**.

<https://github.com/albertaloop/elegoo-demo/blob/main/multiple%20interrupts/src/main.cpp>

The circular buffer

```
#define NUM_BITS    4
#define BUFFER_LEN  (1 << NUM_BITS)

struct circular_buffer_t
{
    int values[BUFFER_LEN];
    unsigned int write_ptr = 0;
    unsigned int read_ptr = 0;
} ;
typedef circular_buffer_t circular_buffer;

void push(int val, volatile circular_buffer *buf)
{
    if (buf->write_ptr >= buf->read_ptr && buf->write_ptr < BUFFER_LEN)
    {
        buf->values[buf->write_ptr] = val;
        buf->write_ptr &= BUFFER_LEN-1;
    }
}

int pop(volatile circular_buffer *buf)
{
    if (buf->read_ptr != buf->write_ptr)
    {
        int ret = buf->values[buf->read_ptr];
        buf->read_ptr &= BUFFER_LEN-1;
    }
    else
    {
        return -1;
    }
}
```

The loop

```
void loop()
{
    static int val;
    if ((val = pop(&timer1_buffer)) >= 0)
    {
        task1();
    }
    if ((val = pop(&timer3_buffer)) >= 0)
    {
        task2();
    }
    if ((val = pop(&timer5_buffer)) >= 0)
    {
        task3();
    }
    atmega_sleep();
}
```

```
ISR(TIMER1_COMPA_vect)
{
    push(1, &timer1_buffer);
}
```

```
ISR(TIMER3_COMPA_vect)
{
    push(1, &timer3_buffer);
}
```

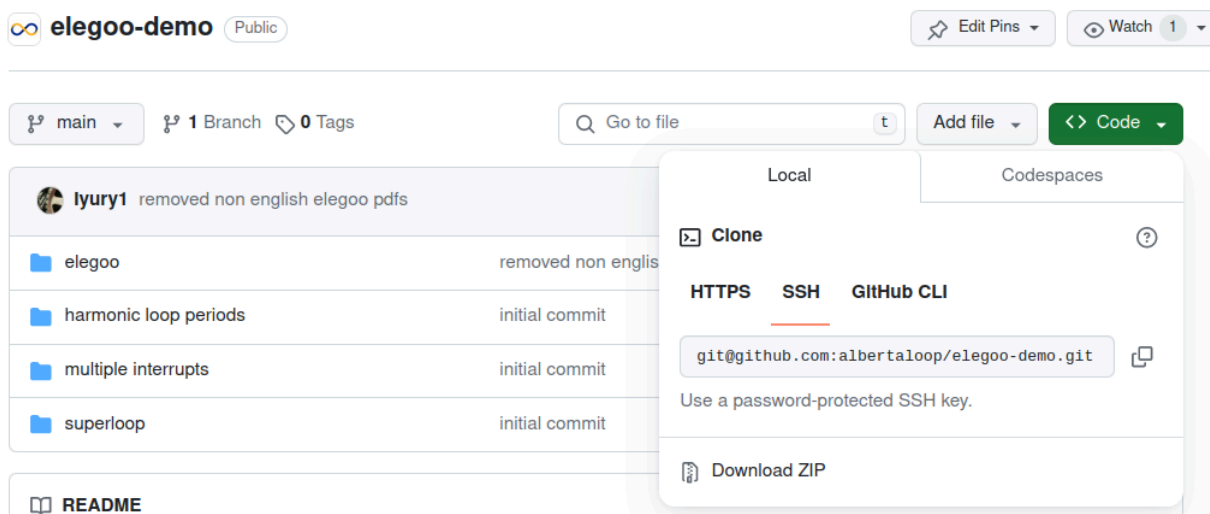
```
ISR(TIMER5_COMPA_vect)
{
    push(1, &timer5_buffer);
}
```

# Environment setup

---

To download the example projects, download the git repository

<https://github.com/albertaloop/elegoo-demo>



The Arduino Mega can be programmed using the Arduino IDE, the PlatformIO extension for VSCode, or from the command line using avr-dude. PlatformIO is recommended for this workshop.

## 1. Arduino IDE

<https://www.arduino.cc/en/software>

Under “Download Options”, select the download for your operating system.

Linux:

The Zip download contains a directory with the arduino ide already installed. You can extract this directory wherever you wish.



I copied mine to /opt/, which is a place applications can be installed that are not tracked by the package manager by your distribution.

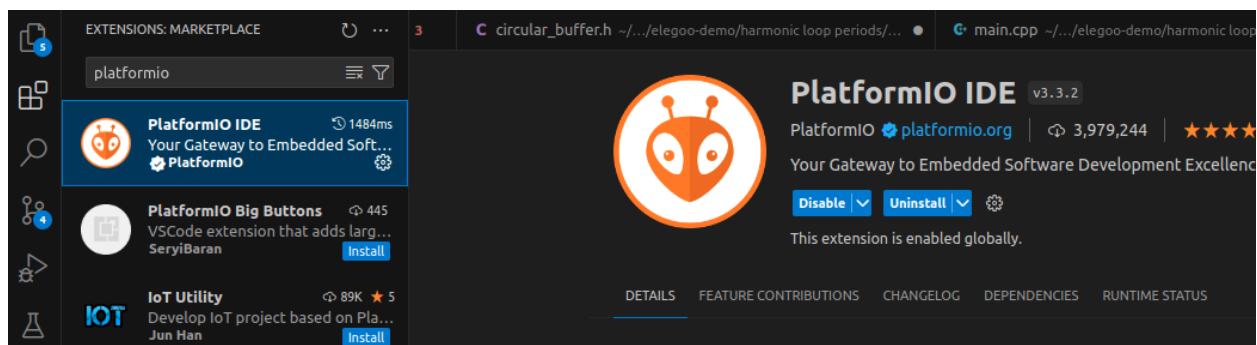
```
ian@ian-GL553VD:~/Downloads$ sudo mv arduino-ide_2.2.1_Linux_64bit /opt/
ian@ian-GL553VD:~/Downloads$ cd /opt/arduino-ide_2.2.1_Linux_64bit/
ian@ian-GL553VD:/opt/arduino-ide_2.2.1_Linux_64bit$ ls
arduino-ide          libffmpeg.so          resources
chrome_100_percent.pak  libGLSv2.so          resources.pak
chrome_200_percent.pak  libvk_swiftshader.so  snapshot_blob.bin
chrome_crashpad_handler  libvulkan.so.1        v8_context_snapshot.bin
chrome-sandbox          LICENSE.electron.txt  vk_swiftshader_icd.json
icudtl.dat             LICENSES.chromium.html
libEGL.so              locales
ian@ian-GL553VD:/opt/arduino-ide_2.2.1_Linux_64bit$ pwd
/opt/arduino-ide_2.2.1_Linux_64bit
ian@ian-GL553VD:/opt/arduino-ide_2.2.1_Linux_64bit$ gedit ~/.bashrc
```

## 2. Platform IO on VSCode

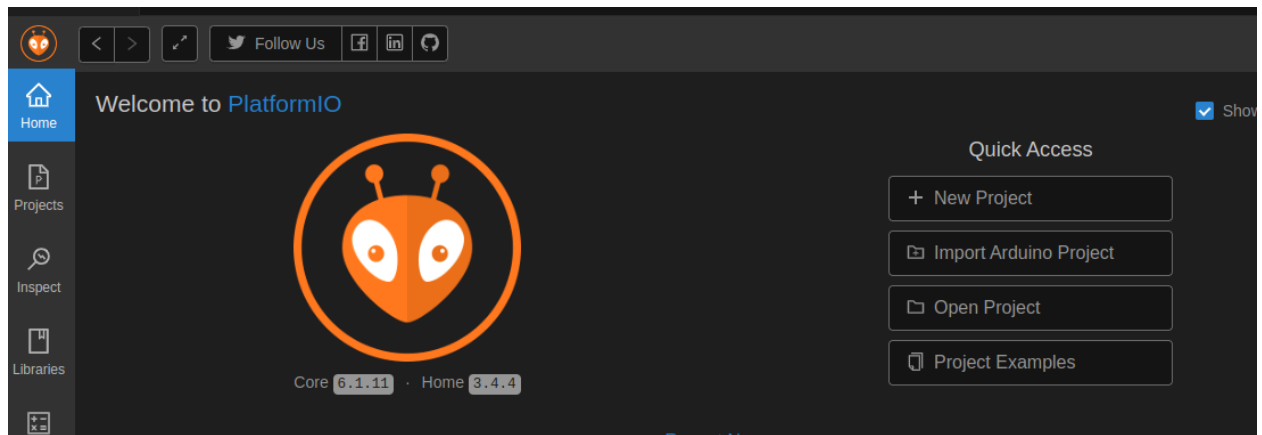
Download VSCode from their downloads page:

<https://code.visualstudio.com/download>

Open VSCode once you have installed it, and search for platformio in the extensions tab.



Once you have installed the extension, you can create a New project under PlatformIO Home. You can also open the project examples provided above.



Enter a name for your project, and set the board to “Arduino Mega or Mega 2560 ATmega2560 (Mega 2560)”. The Arduino framework will be selected automatically. Press Finish when ready.

### Project Wizard

This wizard allows you to **create new** PlatformIO project or **update existing**. In the last case, you need to uncheck "Use default location" and specify path to existing project.

Name:

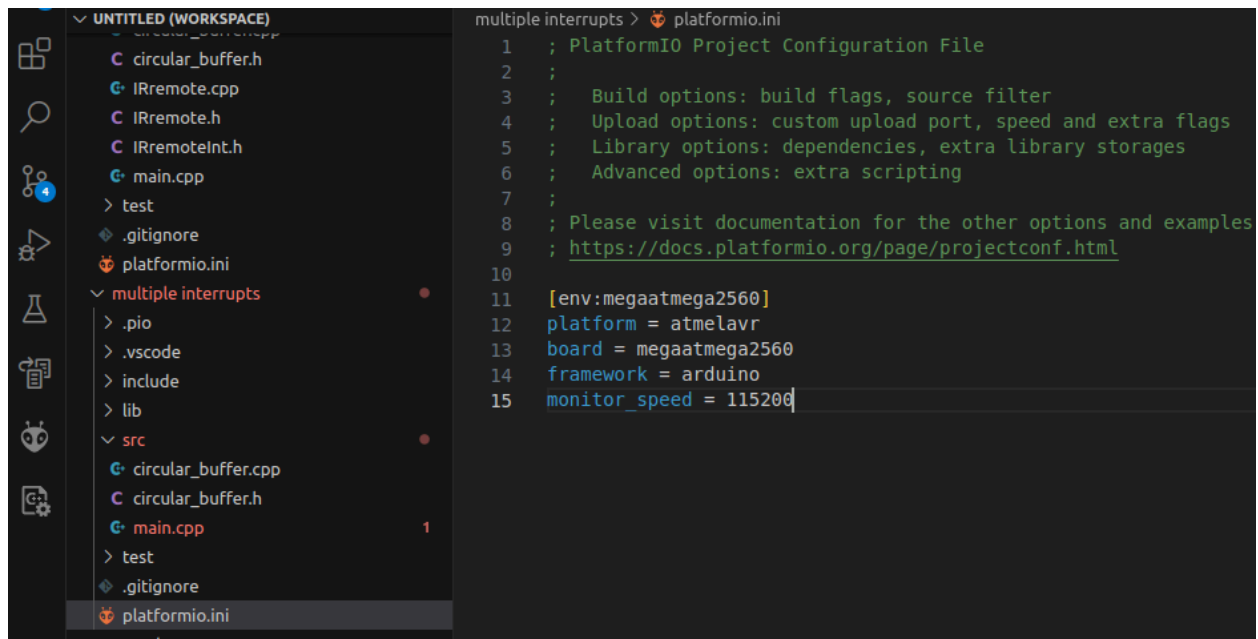
Board:

Framework:

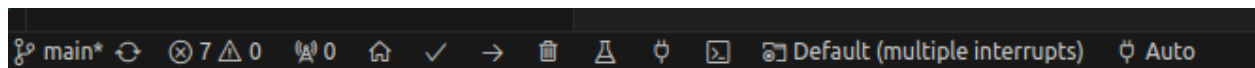
Location: ☒ Use default location

CancelFinish

You can set the baud rate for the serial monitor by adding "monitor\_speed" to the platformio.ini file.

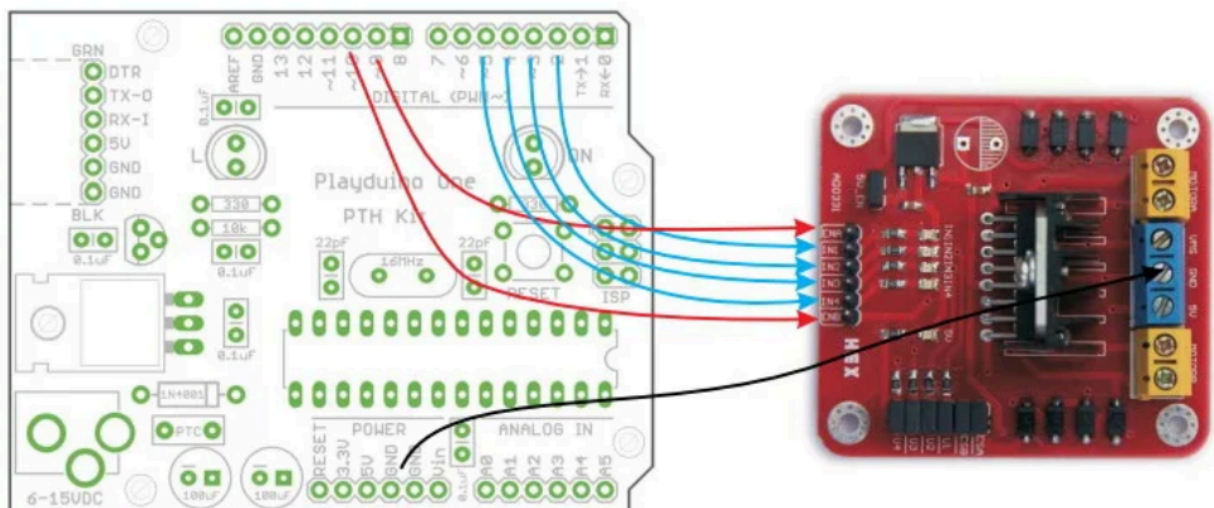


You can upload your project to the board by pressing the -> arrow button on the bottom toolbar. The plug-in button will open a serial monitor.

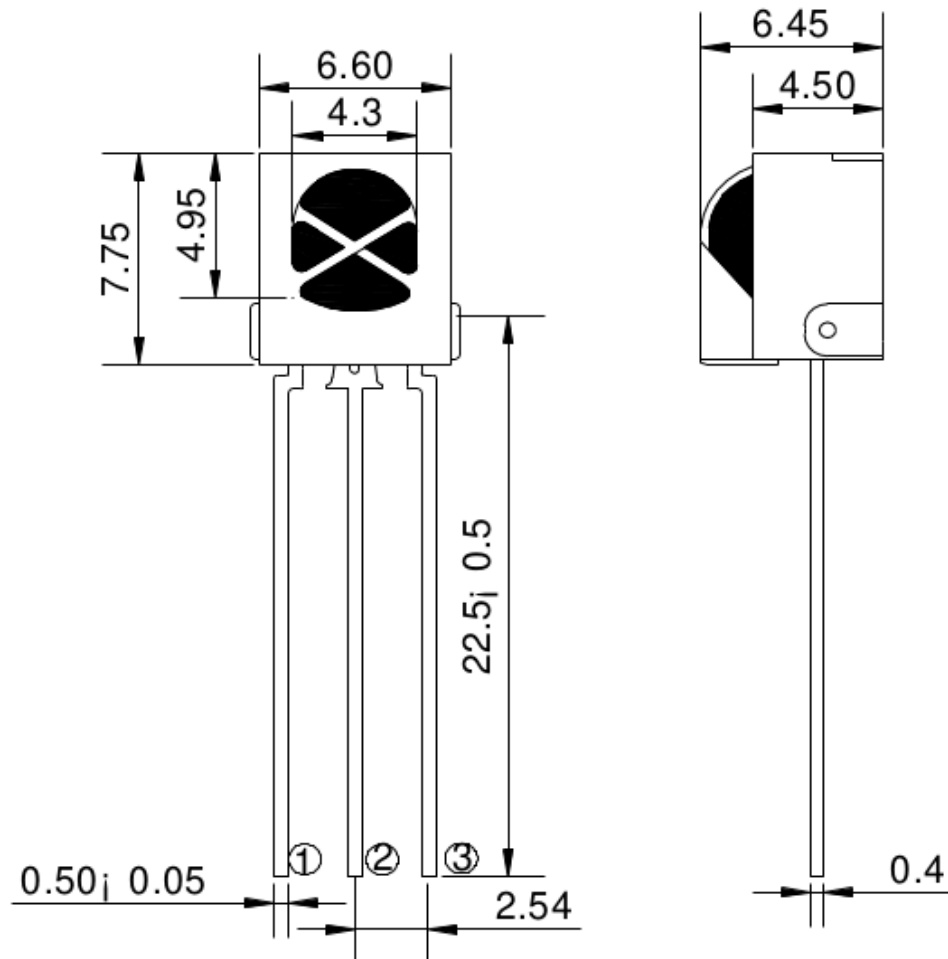


Motor Controller:

<https://www.instructables.com/Arduino-Modules-L298N-Dual-H-Bridge-Motor-Controll/>



IR Receiver:



- ① OUT
- ② GND
- ③ VCC