
Beyond Arduino
Workshop

Beyond Arduino #1

01 Why go beyond Arduino?

02 What exactly is Arduino?

03 What is a HAL?

04 Bare-metal Blink

Why beyond Arduino? Defining the Issue

- You learn Arduino, you build projects with it, and use every available “arduino-compatible” sensor and actuator.
- What to do when you want to use a module that nobody wrote a library for?
- What to do when you find conflicts between libraries?
- How to start using other richer and more professional MCUs like ESP32, STM32 from here?

Why beyond Arduino? Defining the Issue

- Arduino is the highly recommended choice for beginners because it abstracts away the scary stuff.
- But, have you ever wanted to know how digitalWrite() and Servo.write() functions work? What exactly happens under the hood?
- What does the Arduino IDE do to compile and upload your code to your Arduino board?
- If Arduino is not the MCU but a shield around it, can you use the MCU without the board?

Why beyond Arduino? Motivation!

We will answer all these questions, and more!

How?

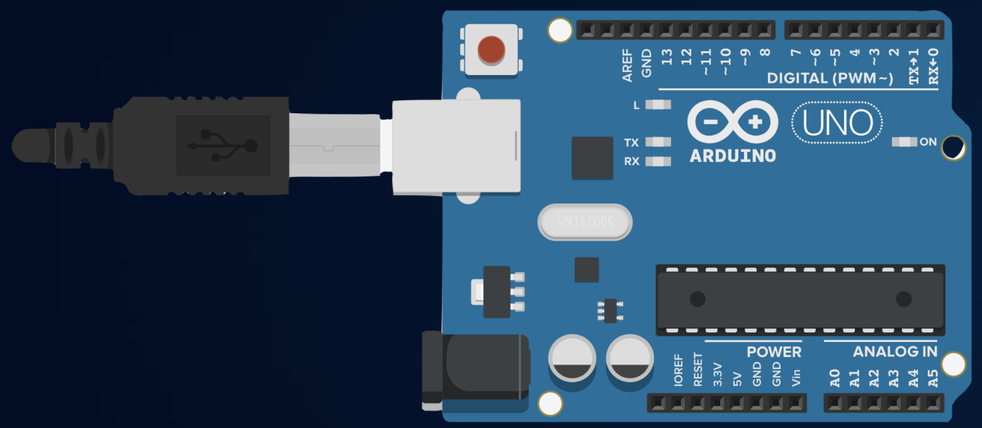
By becoming less restricted on the Arduino platform itself!

We will level up our understanding of Microcontrollers by removing the layers of abstraction of the Arduino Ecosystem one at a time!

We will learn how to read the atmega328p reference manual to get the maximum out of this MCU!

What exactly is Arduino?

Is it a development board?



Arduino is all of them!
It’s an Ecosystem!

Is it the IDE?

ARDUINO IDE

Is it a Framework/Codebase(HAL)?

```
#include <Arduino.h>
```

What is a HAL?

- Generally speaking, a Hardware Abstraction Layer is a software layer (Classes* and functions) that sits between the application code and hardware drivers such as: GPIO drivers, Serial COM drivers, Motor drivers… etc

- Hardware drivers are specifically written for a particular hardware, i.e. for a specific Microcontroller architecture.

- For Arduino, the HAL is a set of C++ functions and libraries that abstract away the low level code that is specific to the underlying MCU architecture.

*If the HAL is written in OOP language.

What is a HAL?

- The HAL hides the hardware details: registers and configurations for interfaces like ADC, UART, I2C, I2S and SPI, from the application programmer. Which means we don’t need to know “how” to setup certain functionalities to get them to run.
- A HAL ensures portability of code across different MCU architectures, because it’s not hardware-specific code.
- This means your LED blinking example can run on an UNO board and on any arduino-compatible board, regardless of the board’s microcontroller.

Why does hardware-dependent code exist?

Why can’t we write one implementation of `digitalWrite()` and have it run on every board?

Because every Microcontroller have its own:

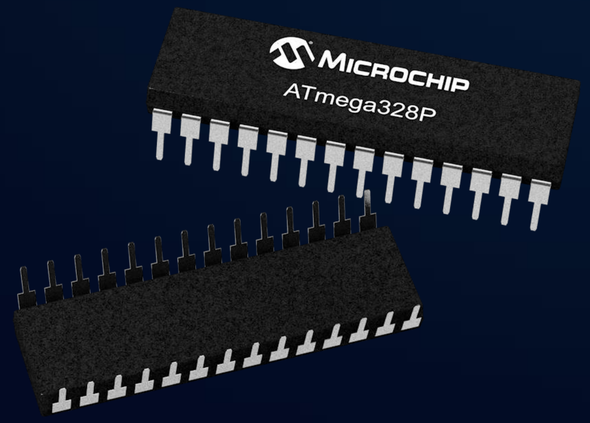
- Hardware Architecture
- Instruction Set (assembly)
- Registers Names/Addresses

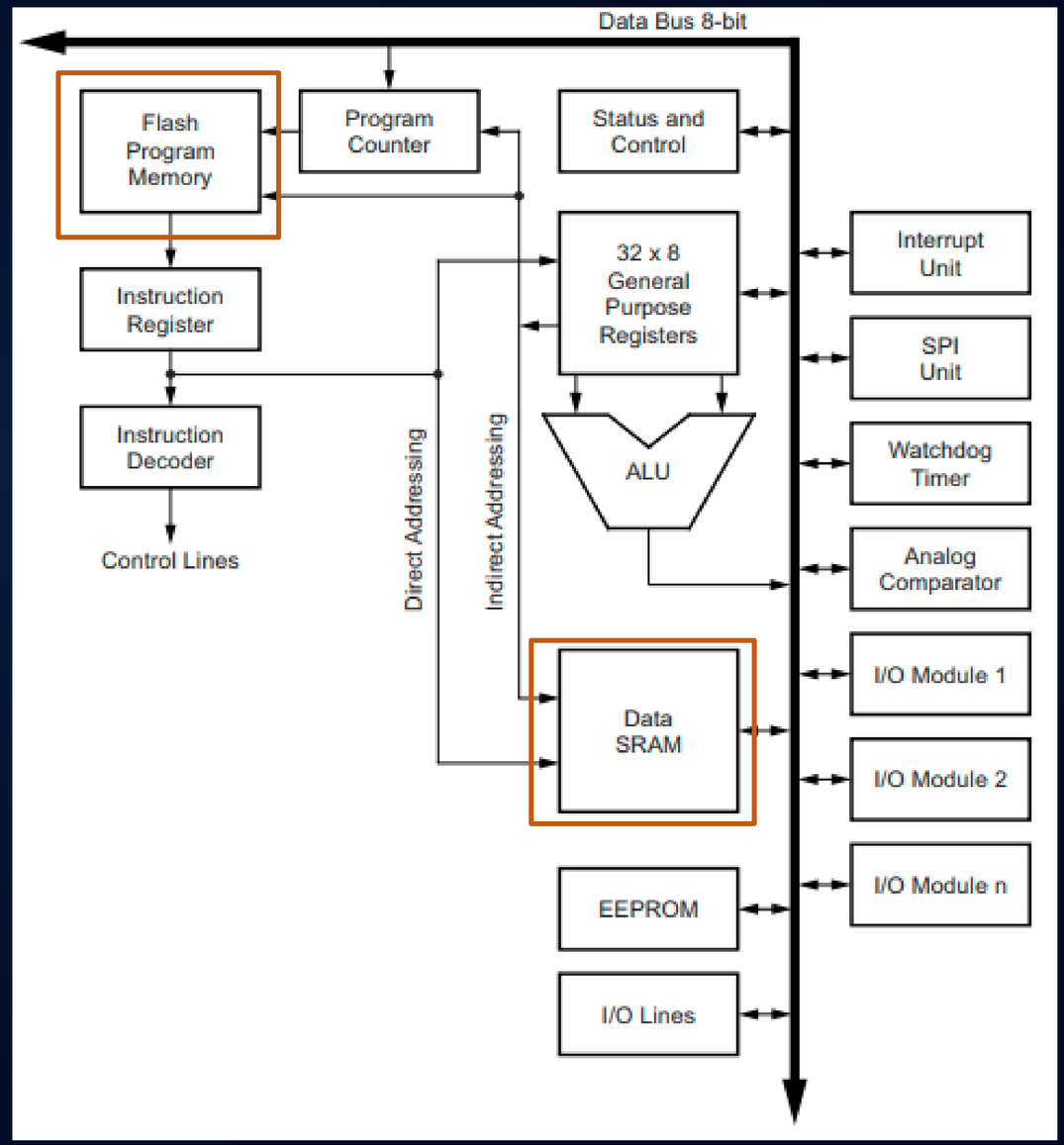
Let’s take the atmega328p as an example to verify this!

Atmega328p stuff

It’s a single-chip microcontroller created by Atmel (now part of Microchip Tech) in the mega-AVR family of MCUs. It has modified Harvard architecture 8-bit RISC processor core.

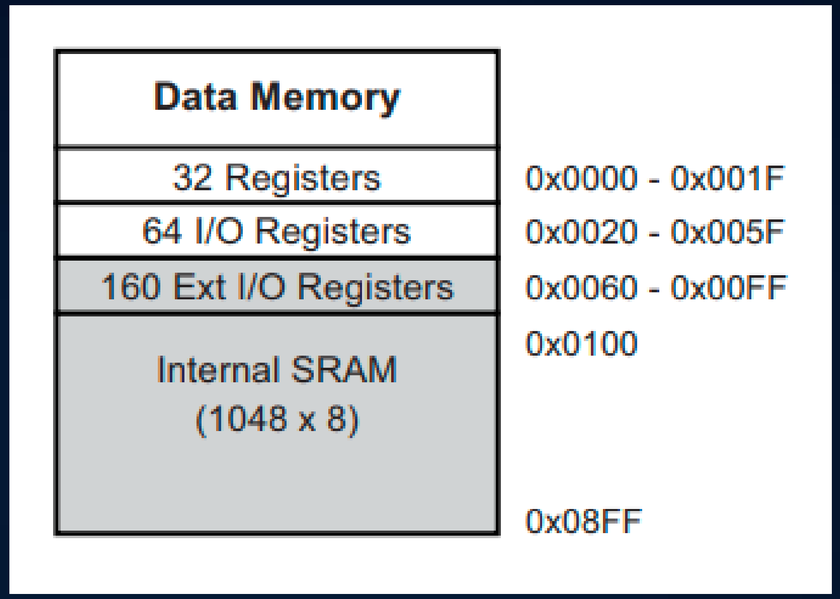
AVR uses a Harvard architecture – with separate memories and buses for program and data.



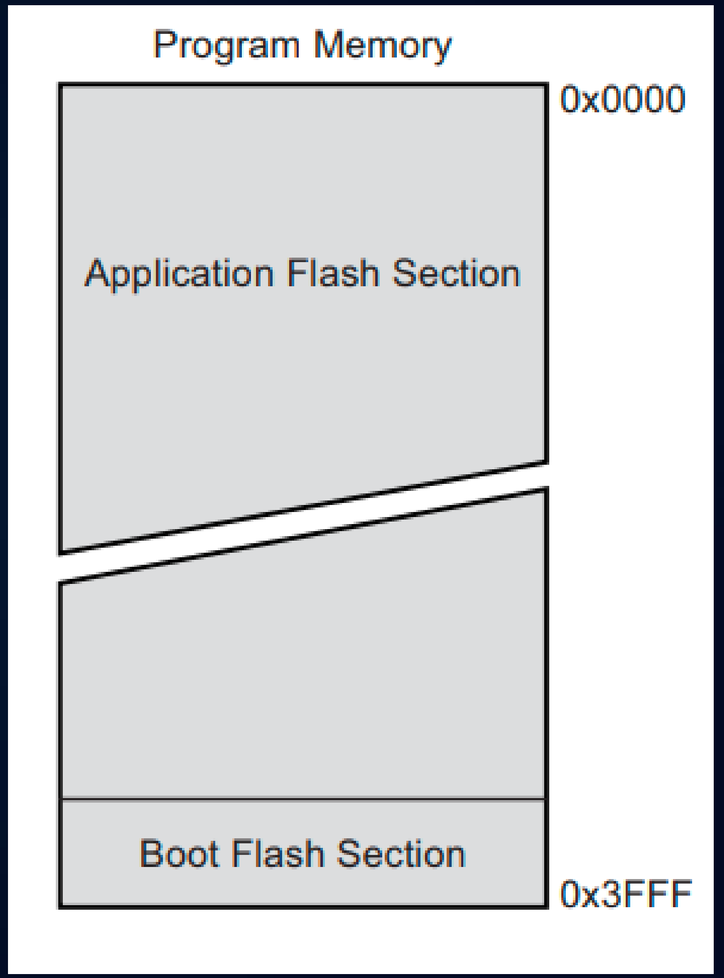


Atmega328p stuff (Memory)

Flash(32KB):
 -> Non-volatile
 -> Program memory
 -> read-only
SRAM(2KB):
 -> volatile
 -> Data Memory - Temporary storage
 -> 32 8-bit core registers reside here.



RAM has 2K addressable locations, each is 8bit wide => 2KB RAM.

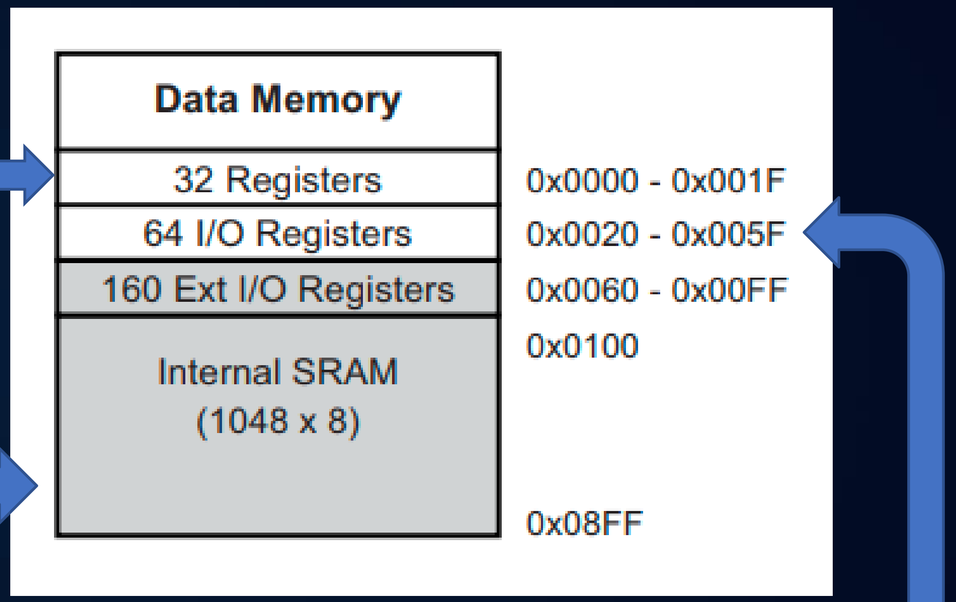


Flash has 16K addressable locations, each is 16bit wide => 32KB Flash.

Atmega328p stuff (Memory)

I/O registers are memory-mapped

Here live 8-bit general purpose registers, that are directly accessed by the ALU. They exist physically inside the CPU core, but are addressed and included in the data memory map.



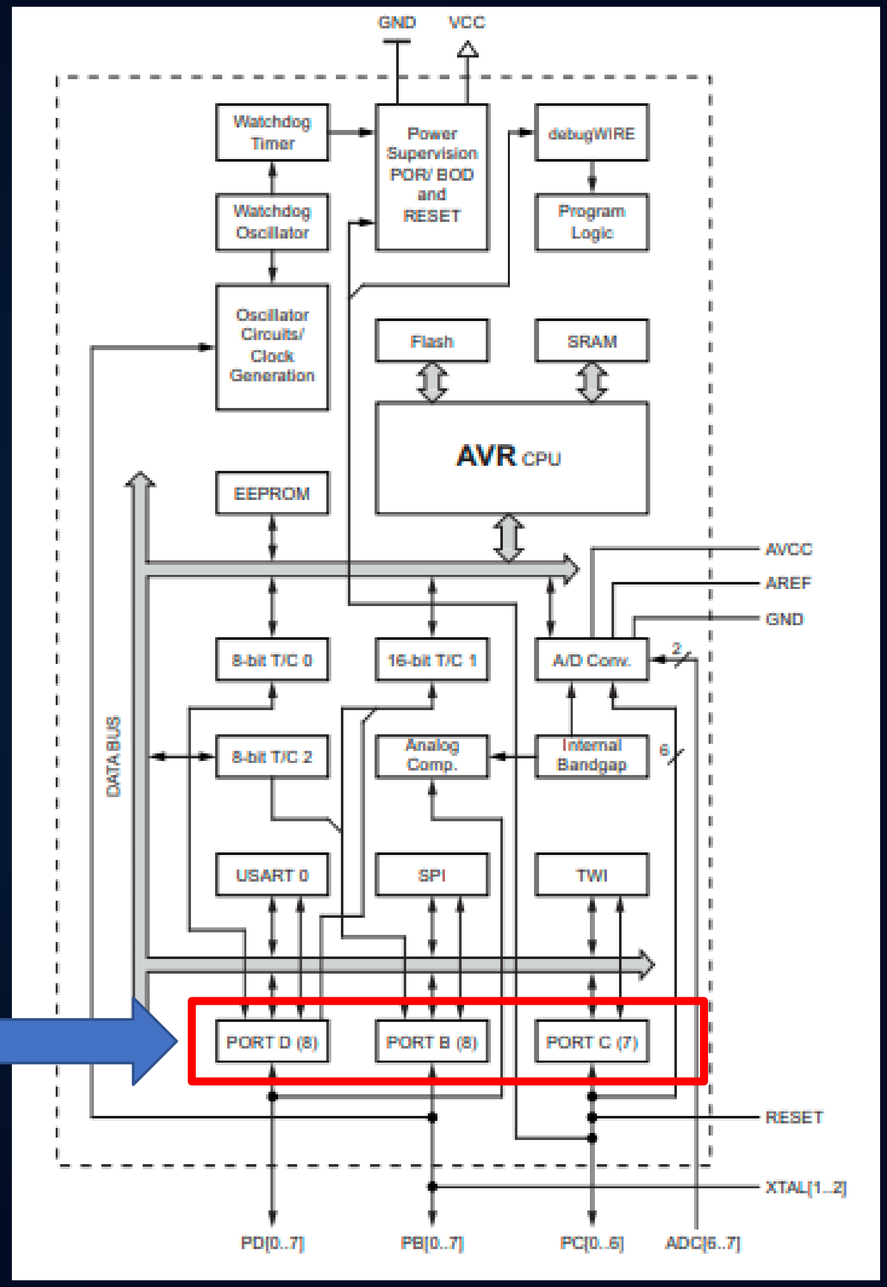
Here live the variables we create (`int i = 9;`), the Stack (for function calls, return addresses (PC) from interrupts & subroutines, and local vars), and other temporary data.

Here live I/O REGISTERS! special-purpose memory locations that the CPU uses to control peripherals and I/O.

Atmega328p stuff (Memory)

Registers are used by the CPU to control many things: pins (GPIO and alternate functions), SPI, internal clock, and other configurations.

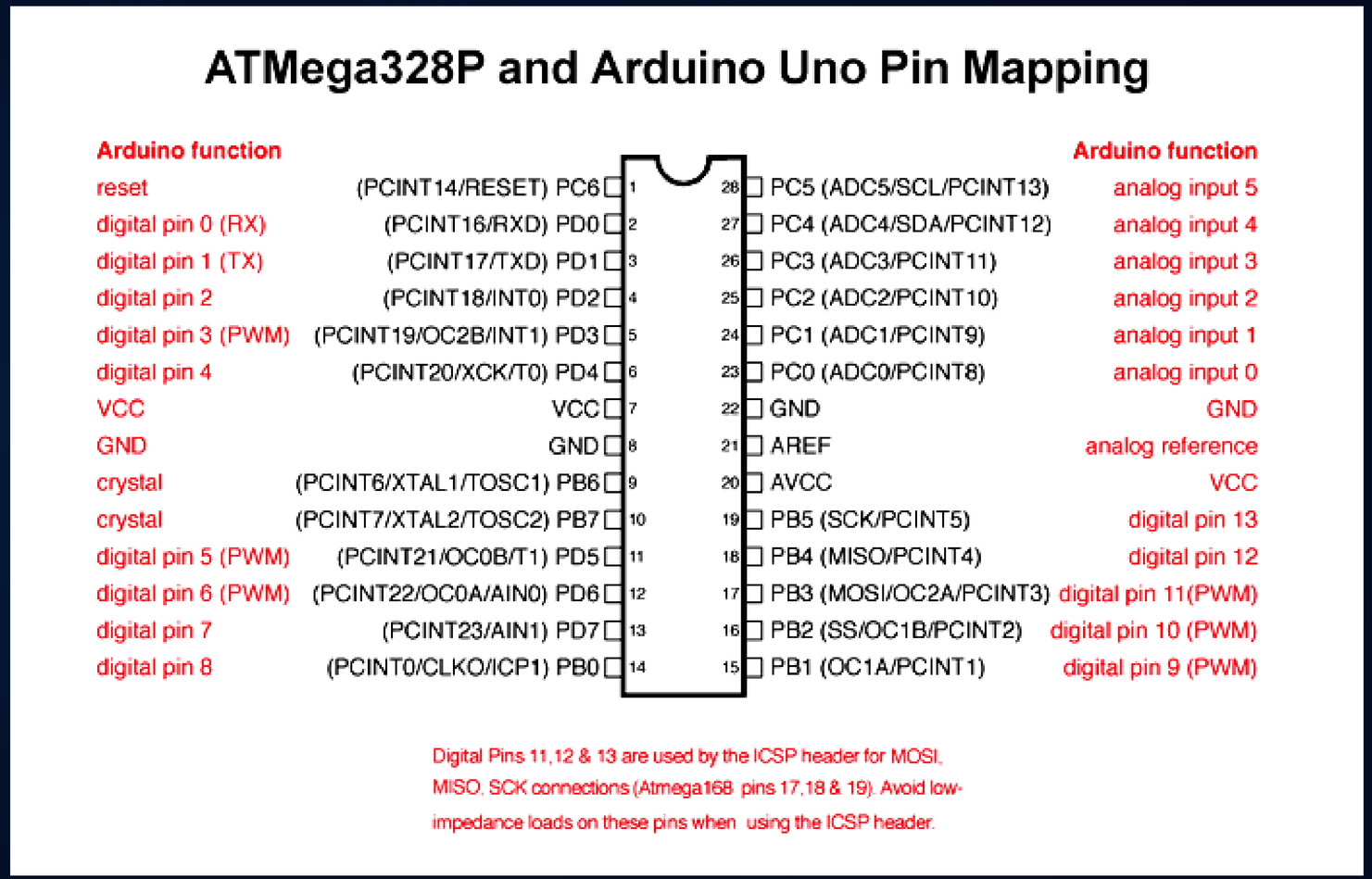
I/O ports: Interface to Pins



Bare-Metal blink (Pins map)

In order to control arduino pins, we need to map conventional Arduino pins (0–13) to the “real” atmega328p pins. In our MCU, there are 3 ports:

PORTD, PORTB, PORTC each with Pins (0–7)



Bare-Metal blink (registers)

Now that we know what registers are, let's use them to blink our LED!

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	

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	

PINB – The Port B Input Pins Address[(1)]

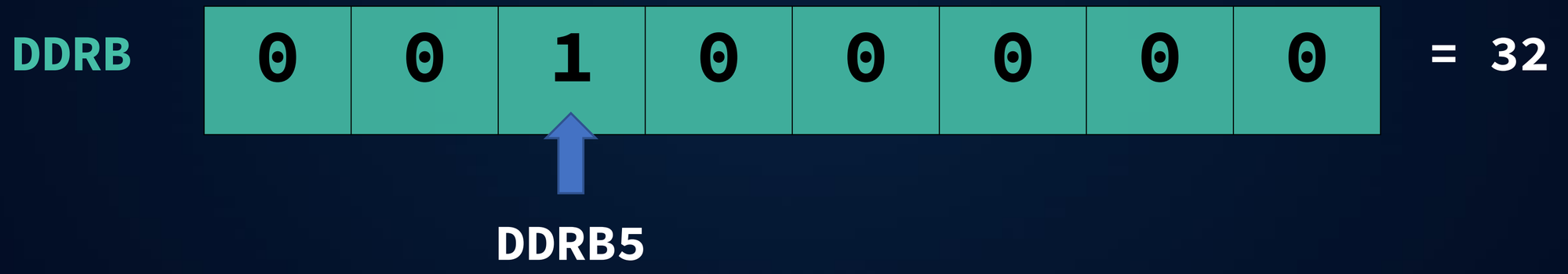
Bit	7	6	5	4	3	2	1	0	
0x03 (0x23)	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

Each I/O port's pin is configured and controlled using 3 register bits from 3 I/O registers: DDRx, PORTx, and PINx.

Bare-Metal blink (registers)

Now that we know what registers are, let’s use them to blink our LED!

DDRx: Data Direction Register (R/W), writing 0 sets pin as input, writing 1 sets pin as output.

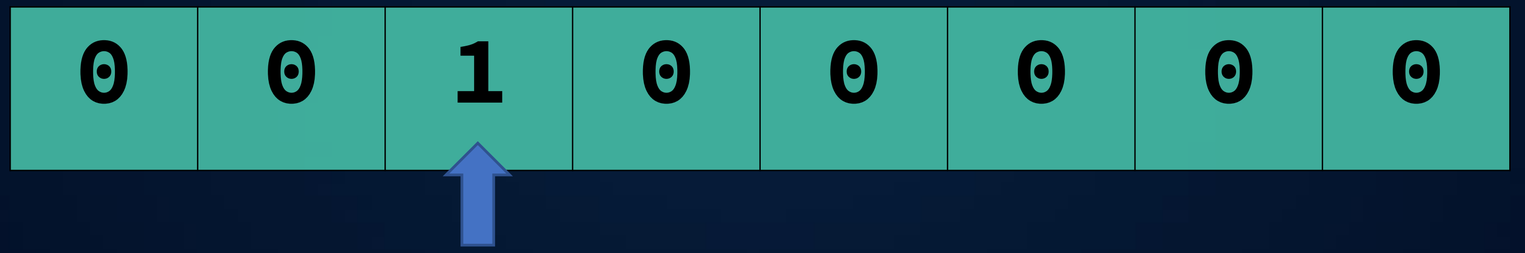


Setting DDRB5 to 1 will set pin 13 on arduino to output

Bare-Metal blink (registers)

Now that we know what registers are, let’s use them to blink our LED!

PORTx: Data Register (R/W), writing 0 sets pin value to 0 (LOW), writing 1 sets pin value to 1 (HIGH).



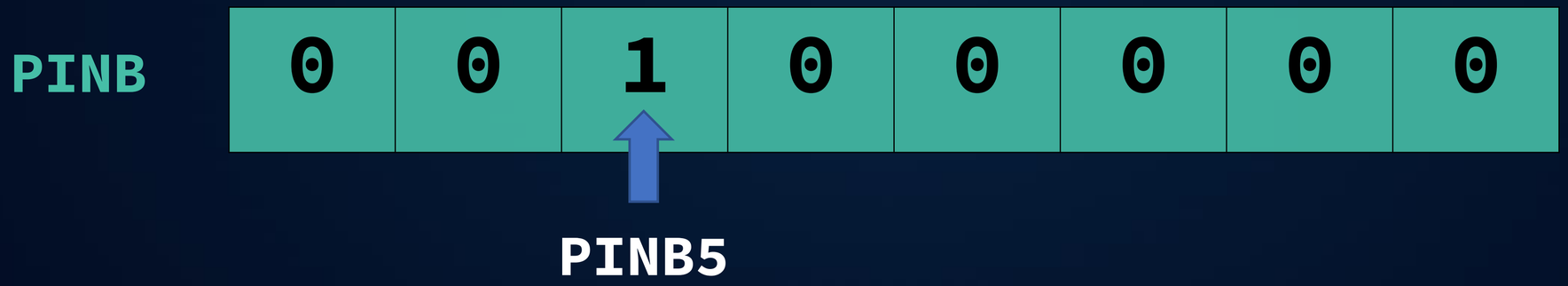
Setting PORTB5 to 1 will set pin 13 on arduino high
Setting PORTB5 to 0 will set pin 13 on arduino low

(Note: PORTx enables/disables pull-up resistors for input pins.)

Bare-Metal blink (registers)

Now that we know what registers are, let’s use them to blink our LED!

PINx: Data Register (read-only), reading this register returns value of pins. However, writing a logic 1 to any pin in this register will toggle it.



Setting PINB5 to 1 will toggle pin 13 on arduino.
Reading PINB5 will return value of pin 13 on arudino.

Bare-Metal blink (bitwise C)

In order to blink our LED, we need to configure registers DDRB and PORTB. How do we write into these registers?

For now, we will accept that there are DDRB and PORTB variables which we will use to access those registers. Later, we will know what these variables actually are!

1- Set a bit: we use OR for that!

There are many ways to code setting a bit to 1, but some are better than others!

```
1) DDRB = 32; // does the job but affects all other bits!
2) DDRB = DDRB | B00100000; // preserves bits, nice! but too many digits
3) DDRB |= 0x20; // nice :)
```

Bare-Metal blink (bitwise C)

2- Reset a bit: we use AND for that!

```
1) PORTB = 0; // does the job but affects all other bits!
2) PORTB = PORTB & ~(B00100000); // preserves bits, nice! but too many digits
3) PORTB &= ~0x20; // nice :)
```

3- Toggling a bit: we use XOR for that!

Note: remember a^0 = a

```
1) PORTB = PORTB ^ B00100000; // toggles 5th bit of PORTB
2) PORTB ^= 0x20; // nice :)
```

Bare-Metal blink (bitwise C)

One final thing before we make an LED blink, promise!

There is even a **BETTER** way to write sth like: **B00100000** and it's by using bitwise shift operations:

We can use: **(1U << 5)** to represent **B00100000**!

```
PORTB &= ~(1U << 5);
PORTB |=  (1U << 5);
```

We can go above and beyond and make it a **MACRO*** and use it as a **bit mask**:

```
#define PINMASK(x) (1U << x)
```

Like this, when we want a bit mask for the 5^{th} bit, we can just:

```
PORTB &= ~PINMASK(5);
```

* It's also called a pseudo-function in this case

Bare-Metal blink (Code)

Bare-metal pinMode() and digitalWrite()!

```
/*------------------Code with Arduino delay()--------------------*/
#define PINMASK(x) (1U << x)
void setup() {
  DDRB |= PINMASK(5); // pinMode(LED_BUILTIN, OUTPUT);

}

void loop() {
  PORTB |= PINMASK(5); // digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  PORTB &= ~PINMASK(5); // digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
}
```

Bare-Metal blink (replacing delay())

Let’s make our own simple_delay()

```
/*-----------------Code with our simple_delay()--------------------*/
#define PINMASK(x) (1U << x)
void simple_delay();

void setup() {
  DDRB |= PINMASK(5); // pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
  PORTB |= PINMASK(5); // digitalWrite(LED_BUILTIN, HIGH);
  simple_delay();
  PORTB &= ~PINMASK(5); // digitalWrite(LED_BUILTIN, LOW);
  simple_delay();
}

void simple_delay(){
  for(volatile uint32_t i=0; i<0x0000FFFF; i++);
}
```

Bare-Metal blink (program size)

Original Blink

```
Sketch uses 924 bytes (2%) of program storage space.
Global variables use 9 bytes (0%) of dynamic memory,
```

Bare metal blink with `delay()`

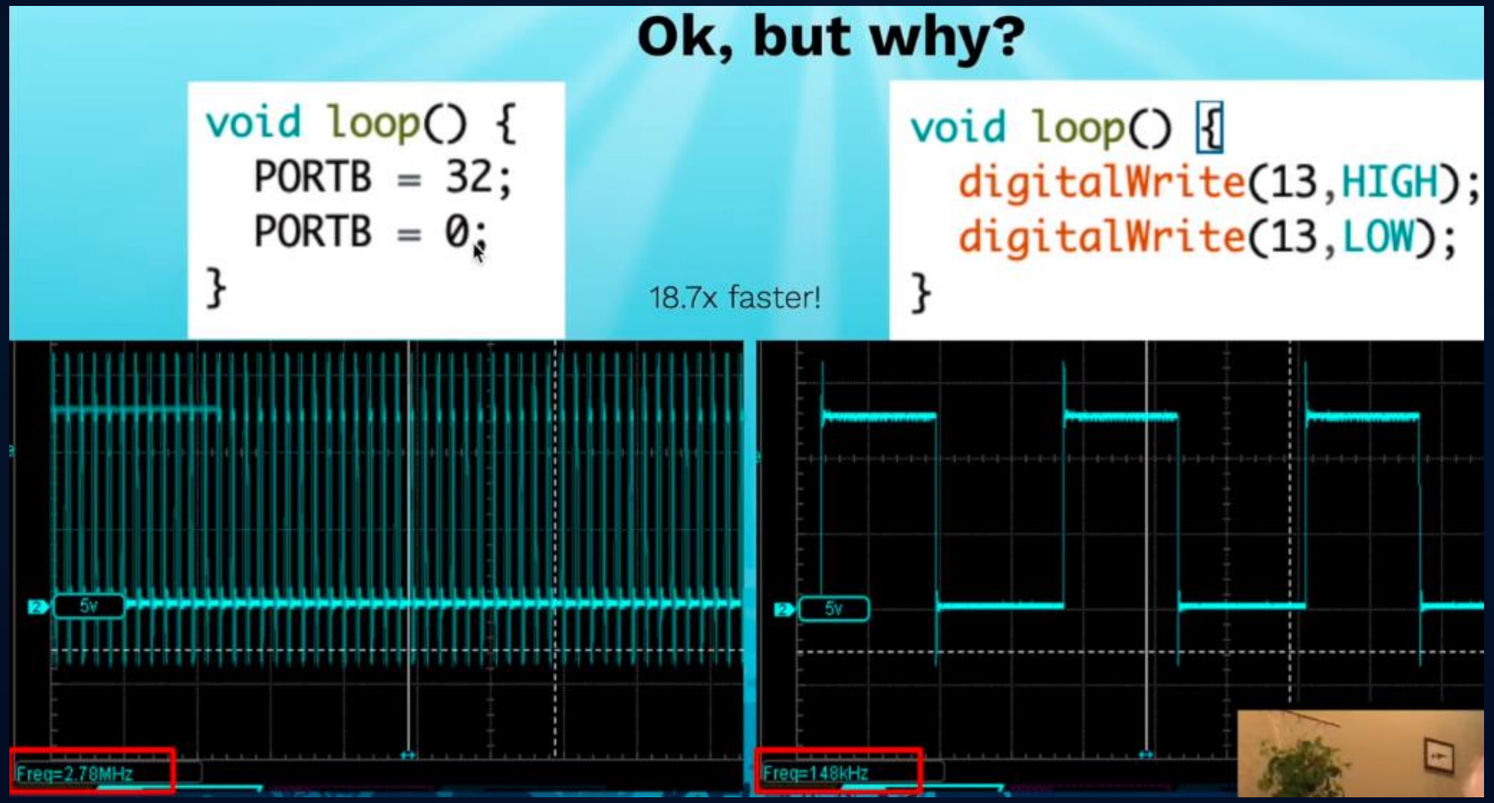
```
Sketch uses 640 bytes (1%) of program storage space
Global variables use 9 bytes (0%) of dynamic memory
```

Bare metal blink with `simple_delay()`

```
Sketch uses 534 bytes (1%) of program storage space. M
Global variables use 9 bytes (0%) of dynamic memory, 
```

Why HAL is sometimes.. bad

In addition to the introduced bloat in our program size, a function like `digitalWrite()` has significant overhead!



Source: Mitch Davis, [Fundamentals of Microcontrollers Playlist]

(Note: digitalWrite() and pinMode() can only work on 1 pin at a time, while directly accessing registers can set multiple at once!)

Now you know why HAL saves the day!

- Beginner friendly
- Hides the scary stuff away
- Makes you focus on your project not how to get things running with the Microcontroller
- Your code and everyone else’s is portable
- You can use libraries and not worry about compatibility issues

Conclusion

- Understanding how atmega328p works will help you transition to other MCUs like STM32 and ESP32. Most concepts in embedded systems are the same on every platform!

- The atmega328p reference manual is very friendly and you will learn a lot from it!

- Are you going to program like this forever? Yes and No. There are many platforms and other HALs which abstract hardware interfacing, but the knowledge of how it works will always be necessary to understand the configurations and even tutorials.

- Learn and be curious. The sky is your limit!