IoT Quarter 2

PIAIC Team

Embedded Systems' Programming

Introduction

IoT Course Timeline

Rust Language

Embedded Systems Programming

Web Applications

Voice Applications

Cloud Computing

Introduction to Embedded Systems

Embedded (definition):

An object fixed firmly and deeply in a surrounding mass; implanted.[1]

Example : “a gold ring with nine embedded stones”

Introduction to Embedded Systems (cont.)

Embedded Systems:

1. “An embedded system is a combination of computer hardware and software, either fixed in capability or programmable, designed for a specific function or functions within a larger system.”

2. “An embedded system is a controller with a dedicated function within a larger mechanical or electrical system, often with real-time computing constraints.”

Examples of Embedded Systems

Examples:

1. HVAC(Heating, ventilation, and air conditioning) Systems
2. ABS(Anti-lock braking) System

Components of Embedded Systems' Programming

- Microprocessors/ Microcontrollers
- Peripherals
- Sensors & Input devices
- Actuators & output devices
- Registers
- Protocols

Microprocessors/ Microcontrollers

Microprocessor:

It is a computer processor that incorporates the functions of a central processing unit on a single IC. The microprocessor is a multipurpose, clock driven, register based, digital integrated circuit that accepts binary data as input, processes it according to instructions stored in its memory and provides results as output.

Microcontroller:

A microcontroller is a **system on a chip**. Unlike computers those have many discrete parts it has all CPUs(Processors cores), memory and input/output **peripherals**. This makes it possible to build systems with minimal part count.

Difference between Microcontroller & Microprocessor[1]

Microprocessor

- Its an IC which only has CPU (Processing power).
- They find application where task are unspecified like developing games, websites, photo editing etc.
- They runs operating systems.

Microcontroller

- It has a CPU along with RAM, ROM and other peripherals all on a single chip.
- They are designed to perform specific tasks. (i.e. cars, bikes, microwave)
- They run bare-metal

Microcontrollers

Peripherals

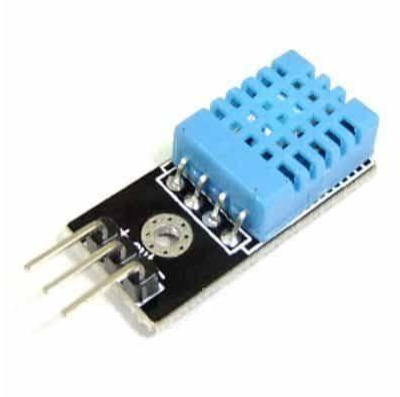
A peripheral or peripheral device is "an ancillary device used to put information into and get information out of the computer".[1]

Sensors & Input devices

Sensors are sophisticated devices that are frequently used to detect and respond to electrical or optical signals.

Examples:

- Temperature sensor
- IR(PIR) sensor
- Touch sensor
- Pressure sensor



Actuators & output devices

An actuator is a component of a machine that is responsible for moving and controlling a mechanism or system, for example by opening a valve.

Example:

- Electric motor
- Screw jack
- Hydraulic Cylinder

Registers

A processor register (CPU register) is one of a small set of data holding places that are part of the computer processor.[1]

A register may hold an instruction, a storage address, or any kind of data (such as a bit sequence or individual characters). Some instructions specify registers as part of the instruction. For example, an instruction may specify that the contents of two defined registers be added together and then placed in a specified register.

Protocols

A protocol is a standard set of rules that allow electronic devices to communicate with each other. These rules include what type of data may be transmitted, what commands are used to send and receive data, and how data transfers are confirmed.

Examples:

I2C : Inter-Integrated Circuit

SPI : Serial Peripheral Interface

USART/UART : Universal Synchronous/Asynchronous Receiver Transmitter

Books and References

Books and References

Book:

- Discovery[1]

References:

The Embedded Rust Book[2]


PRESIDENTIAL
INITIATIVE
ARTIFICIAL INTELLIGENCE AND COMPUTING

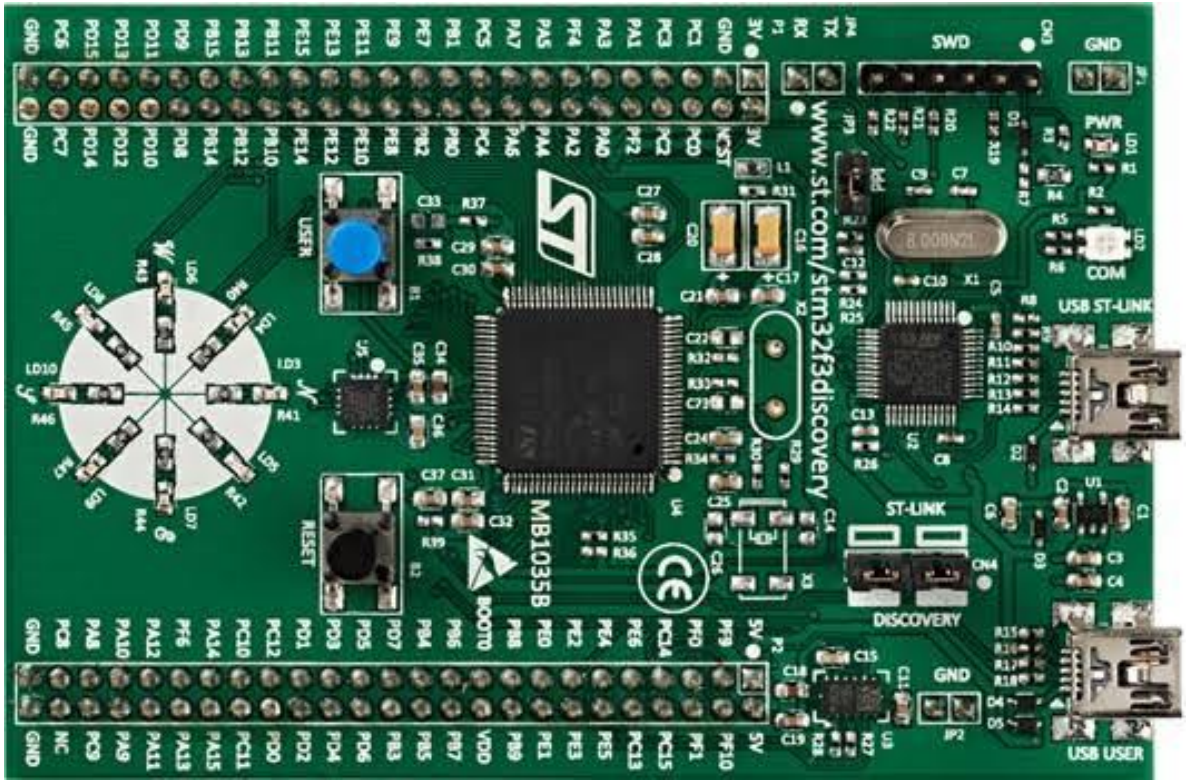
Hardware

Hardware Required

Primarily we'll be required the following hardware:

- STM32F303 Board
- 3.3V USB <-> Serial Module
- Mini-b USB Cable
- 10 jumper wires each (Female to female, Female to male)

STM32F303 Board



3.3V USB <-> Serial Module

Mini-B USB Cable

That’s all

Discovery

Introduction to the book

Introduction to the book

Using this book, we will be learning microcontroller-based embedded systems that uses Rust as programming language. In this lecture we will be discussing the following aspects of this book:

- Scope
- Approach
- Non-goals

Scope

- Writing, building, flashing and debugging
- Peripherals: Digital input/output, Pulse width modulation (PWM), Analog to digital convertors(ADC)
- Control systems: Sensors, actuators, calibration, open loop control (non-feedback), closed loop control (feedback).

Approach

- Beginner friendly - No prior experience required.
- Hands on - Less boring theory!
- Tools centered - Debugging tools will be used

Non-goals

- Included goals!
- We will cover all the related aspects in great details. So no worries!

That’s all

Few Prerequisites

Bitwise & Unsafe Rust

Few Prerequisites

Following are the prerequisites need to be covered before jumping into sea of embedded programming.

1. Binary Numbering System
2. Bitwise operators
3. Unsafe Rust
4. Raw Pointers
5. Mutable Static variables

Binary Number System

Binary Number System

Keep the points in mind in sequence.

1. Binary numbers are **made up of only 1's and 0's**. They don't contain any other number we usually see in decimals (i.e. 2,3,4,...,9). Here are few examples.

Decimal	0	1	2	4	13	21	27	45
Binary	0	1	10	100	1101	10101	11011	101101

Binary Number System

2. Binary numbers have **base of 2** in contrast our decimal numbers which have **base 10**. What base tells us? It tells how many combination can a number give us.

Decimal		Binary	
0000	Starts at 0	0000	Starts at 0
0009	Goes upto 9	0001	Goes upto 1
0010	Reset the digit and increment the next digit.	0010	Reset the current digit and increment the next
0019	Again goes upto 9	0011	What happens next?
0099	What happens with 99, now 2 digits are at optimum	0100	Initial 2 digits got reset and next digit will be incremented
0100	Both will got reset and next digit will be incremented.		

Binary Number System

Now, let's see how to read the binary numbers and also how to convert decimal numbers into binary numbers.

Significance of each bit.

Binary Base = 2

	Column 8	Column 7	Column 6	Column 5	Column 4	Column 3	Column 2	Column 1
$Base^{exp}$	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Weight	128	64	32	16	8	4	2	1

$2^0 = 1$

$2^1 = 2$

$2^2 = 2 * 2 = 4$

$2^3 = 2 * 2 * 2 = 8$

$2^4 = 2 * 2 * 2 * 2 = 16$

$2^5 = 2 * 2 * 2 * 2 * 2 = 32$

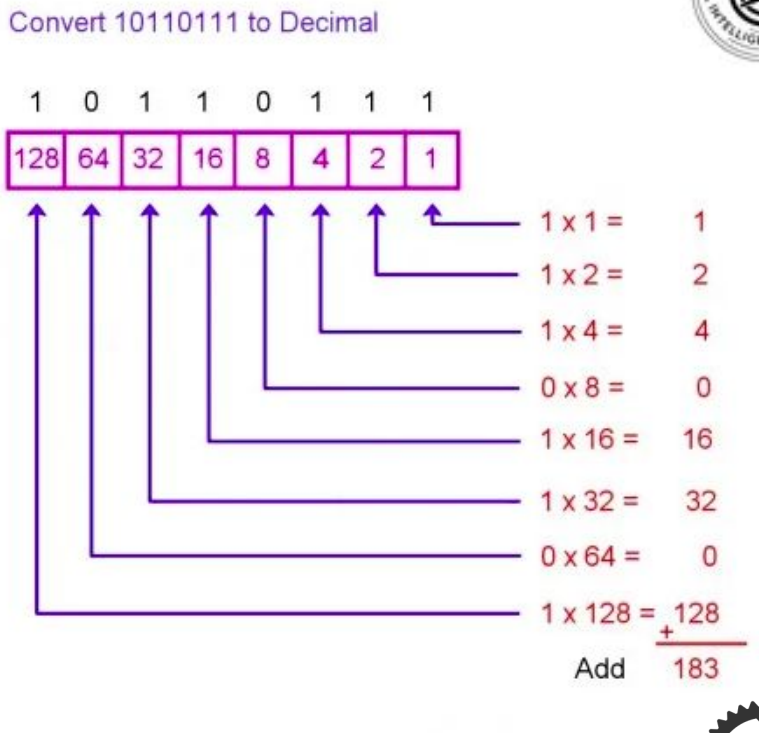
$2^6 = 2 * 2 * 2 * 2 * 2 * 2 = 64$

$2^7 = 2 * 2 * 2 * 2 * 2 * 2 * 2 = 128$

Binary Number System

Now we actually going to convert a decimal number into binary.

Keep the points we learnt so far while converting the number.



Bitwise Operators

Rust Bitwise Operator

Wondering! What these are?

Now you are familiar with binary numbers. Next we will see concept of Bitwise operators that based on the binary numbers.

There are variety of operators in programming languages (i.e. arithmetic, relational, logical and assignment operators). Bitwise operator is one of them.

Bitwise operators: & **,** | **,** ^ **,** ! **,** << **,** >>

Assume values: **A** = 2 (b10) , **B** = 3 (b11)

Bitwise AND (&)

It performs a Boolean AND operation on each bit of its integer arguments.

Example:

A & B => 2, How come this result evaluated ?

main.rs saving...

```
1  fn main() {
2      let a:i32 = 2;
3      let b:i32 = 3;
4
5      let mut result:i32;
6
7      result = a & b;
8      println!("(a & b) => {}",result);
9  }
```

1 0 & 1 1
1 0

Result :

```
(a & b) => 2
```

Bitwise OR (|)

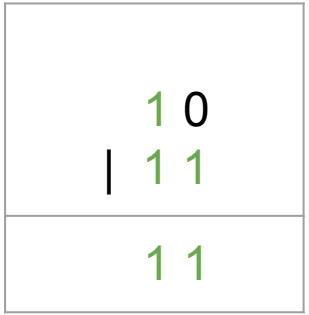
It performs a Boolean OR operation on each bit of its integer arguments.

Example:

A | B => 3, How come this result evaluated ?

main.rs saving...

```
1  fn main() {
2      let a:i32 = 2;
3      let b:i32 = 3;
4
5      let mut result:i32;
6
7      result = a | b;
8      println!("(a | b) => {}",result);
9  }
```



Result :

```
(a | b) => 3
```

Bitwise XOR (^)

It performs a Boolean exclusive OR operation on each bit of its integer arguments.

Example:

A ^ B => 1, How come this result evaluated ?

main.rs saving...

```
1  fn main() {
2    let a:i32 = 2;
3    let b:i32 = 3;
4
5    let mut result:i32;
6
7    result = a ^ b;
8    println!("(a ^ b) => {}",result);
9  }
```

1 0 ^ 1 1
0 1

Result :

```
(a ^ b) => 1
```

Bitwise NOT (!)

It is a unary operator and operates by reversing all the bits in the operand.

Example:

!B => -14, How come this result evaluated ?

main.rs saving...

```
1  fn main() {
2      let b : i8 = 13;
3
4      let mut result : i8 ;
5      result = !b;
6      println!("(!b) => {} , {:b}",result,result);
7
8  }
```

! 0 0 0 0 1 1 0 1
1 1 1 1 0 0 1 0 (- 1 4)

Result :

```
(!b) => -14 , 11110010
```

Bitwise left shift (<<)

It moves all the bits in its first operand to the left by the number of places specified in the second operand.

Example:

A << B = 16, How come this result evaluated ?

(2) 0 0 0 0 0 0 1 0 << 3
(16) 0 0 0 1 0 0 0 0

main.rs saving...

```
1  fn main() {
2      let a : i8 = 2;
3      let b : i8 = 3;
4
5      let mut result : i8 ;
6      result = a << b;
7      println!("(a << b) => {}",result);
8
9  }
```

Result :

```
(a << b) => 16
```

Bitwise right shift (>>)

Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand.

Example:

A >> B = 0, How come this result evaluated ?

main.rs saving...

```
1  fn main() {
2   let a : i8 = 2;
3   let b : i8 = 3;
4
5   let mut result : i8;
6   result = a >> b;
7   println!("(a >> b) => {}",result);
8  }
```

(2) 0 0 0 0 0 0 1 0 >> 3
(0) 0 0 0 0 0 0 0 0

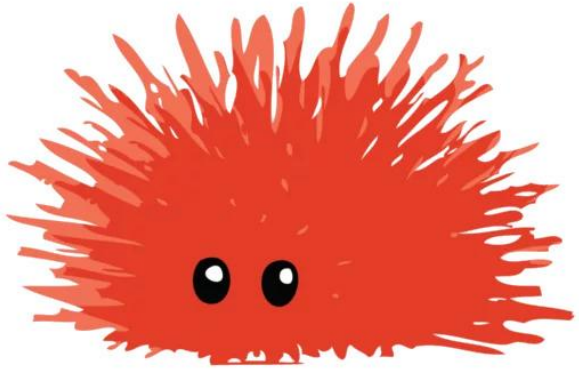
Result :

```
(a << b) => 16
```


PRESIDENTIAL
INITIATIVE
ARTIFICIAL INTELLIGENCE AND COMPUTING

Unsafe Rust

Unsafe Rust



What is that?

- Hidden language inside Rust
- Gives us extra superpowers

Why is it?

- It let us do the operations which compiler restricts us from
- Without unsafe Rust we can’t do operations on hardware level

Unsafe Superpowers:

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- Implement an unsafe trait

Unsafe Rust

While using unsafe Rust keep few things in mind:

- Unsafe rust doesn’t disable borrow checker or any other safety feature.
- It gives you only access to these four features and compiler don’t check for memory safety in them.
- Inside unsafe block the code doesn’t need to be necessarily dangerous.
- While writing code keep the unsafe block smaller.
- For the isolation of unsafe code, it’s better to enclose unsafe code with in a safe abstraction. Wrapper!
- When using unsafe rust you as programmer have to ensure that you are accessing memory in a valid way.

Unsafe Rust

A simple unsafe code block:

main.rs saved

```
1   fn main() {
2
3     let mut num = 5;
4
5     let r1 = &num as *const i32;
6     let r2 = &mut num as *mut i32;
7
8     unsafe{
9       println!("r1 is : {}", *r1);
10      println!("r2 is : {}", *r2);
11    }
12
13  }
```

Raw Pointers

Raw pointer

Unsafe Rust has two new types called **raw pointers** that are similar to references. They can be immutable or mutable and can be written as:

- *const T (Immutable)
- *mut T (Mutable)

Note: Asterisk " ***** " is not a dereferencing operator, it's part of type name.

Let's see how to create an immutable and mutable raw pointer from **reference**.

```
3   let mut num = 5;
4
5   let r1 = &num as *const i32;
6   let r2 = &mut num as *mut i32;
```

Is something wrong! We didn't include **unsafe** keyword here? Well, its okay to create raw pointers outside unsafe but it's must to include while dereferencing them.

Raw pointer

Now, let’s create a raw pointer whose validity can’t be guaranteed.

main.rs saved

```
1  fn main() {
2
3      let address = 0x012345usize;
4      let r = address *const i32;
5
6  }
```

Above approach to access memory is not recommended, however, you might see or write this kind of code.

Dereference a raw pointer

At the end we will access these raw pointers or dereference them.

Case 1:
Creating a raw pointer and dereference it from a valid reference is completely okay.

main.rs saving...

```
1   fn main() {
2
3     //case 1
4     let mut num = 5;
5
6     let r1 = &num as *const i32;
7     let r2 = &mut num as *mut i32;
8
9     unsafe{
10      println!("r1 is : {}", *r1);
11      println!("r2 is : {}", *r2);
12    }
13  }
```

Upon running this code we'll get the result:

```
r1 is : 5
r2 is : 5
```

What if we print values of r1 and r2.

```
r1 is: 0x7ffd359774a4
r2 is: 0x7ffd359774a4
```

Dereference a raw pointer

Case 2:

But when creating a reference from arbitrary location of memory and trying dereference it. It could be problematic. There might be data at that location or might be not.

We'll get the following result upon executing this code.

main.rs saving...

```rust
fn main() {

    //case 2
    let address = 0x012345usize;
    let r = address as *const i32;

    unsafe {
        println!("r is : {:?}", *r);
    }
}
```

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 0.49s
     Running `target/debug/playground`
timeout: the monitored command dumped core
/root/entrypoint.sh: line 8:     8 Segmentation fault      timeout --signal=KILL ${timeout} "$@"
```

Raw pointer

How **raw pointers** different from **references**!

- Are allowed to ignore the borrowing rules by having both immutable and mutable pointers or multiple mutable pointers to the same location
- Aren’t guaranteed to point to valid memory
- Are allowed to be null
- Don’t implement any automatic cleanup

Static Variables

Static variables

- Global variables in Rust are called static variables
- Static variables are similar to **Constants**
- Naming style of static variables is in SCREAMING_SNAKE_CASE
- Type annotation for static variables is must (like constants).
- Rust allows access read-only static variables (immutable static variables), however doesn’t allow access to immutable static variables.
- Static variables always have fixed address in memory.

Summary

Chapter 1

Background

Background

- Microcontroller (MCU) In Depth
- Rust vs C

Microcontroller In Depth

Microcontroller In Depth

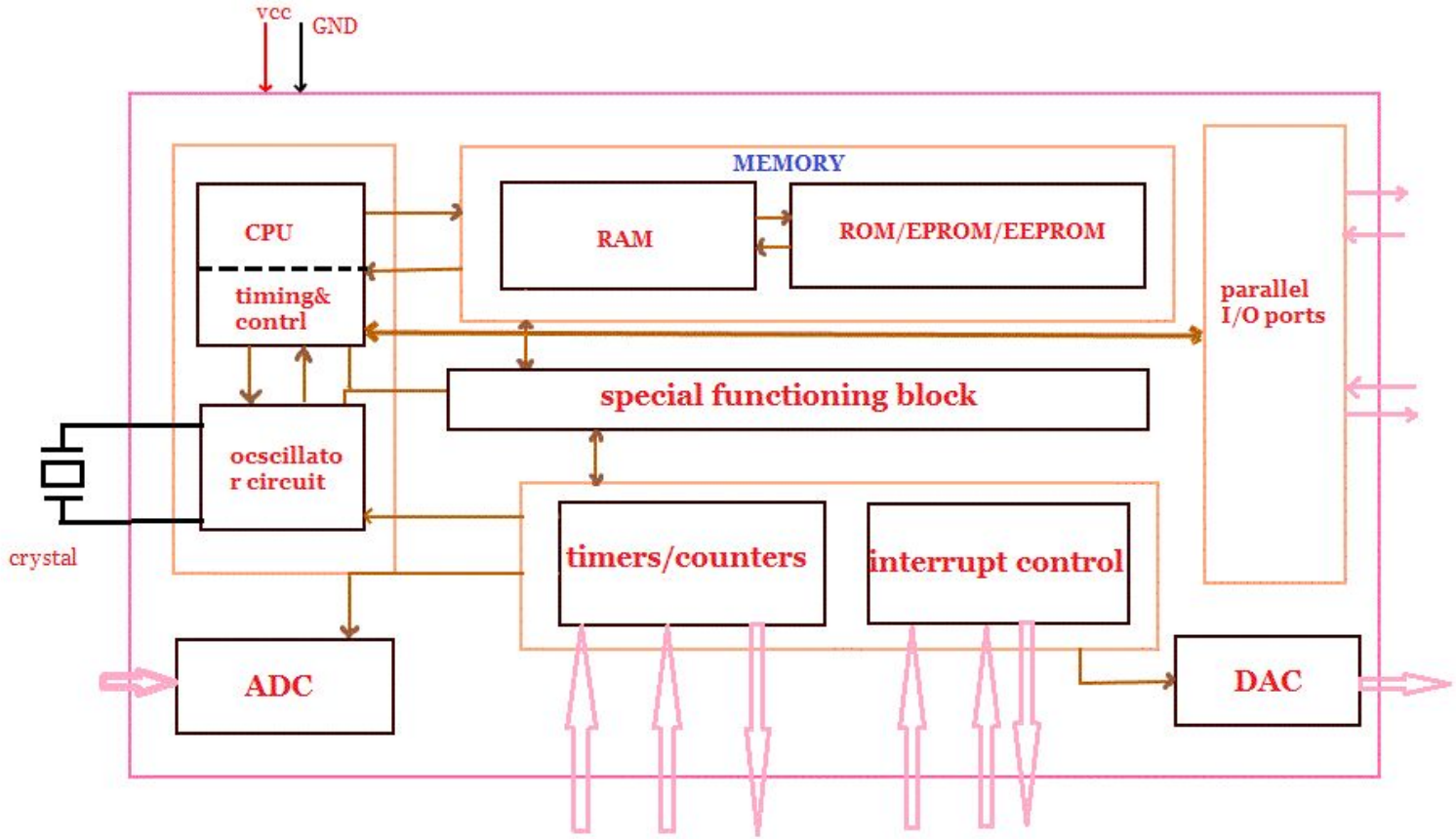
- What is Microcontroller?
- Components of MCU
- Applications
- Why Microcontrollers?
- What are the pros and cons?

What Is Microcontroller?

- **MCU** in simplest means are Integrated Circuit (**IC**)
- **MCU** is a system on chip (**SOC**); a chip consist of several components. We'll discuss these components
- **MCUs** are designed for **embedded** applications

Microcontroller Components[1]

- **Central processing unit(CPU)**
- **Random Access Memory)(RAM) - Volatile Memory**
- **Read Only Memory(ROM)/ Flash Memory - Non-Volatile Memory**
- **Input/output ports**
- Timers and Counters
- Interrupt Controls
- Analog to digital converters
- Digital analog converters
- Serial interfacing ports
- Oscillatory circuits



Microcontroller - Structure Block Diagram

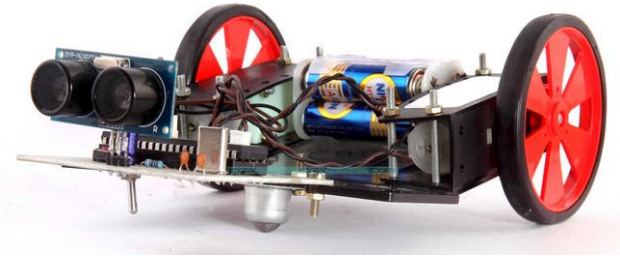
Applications

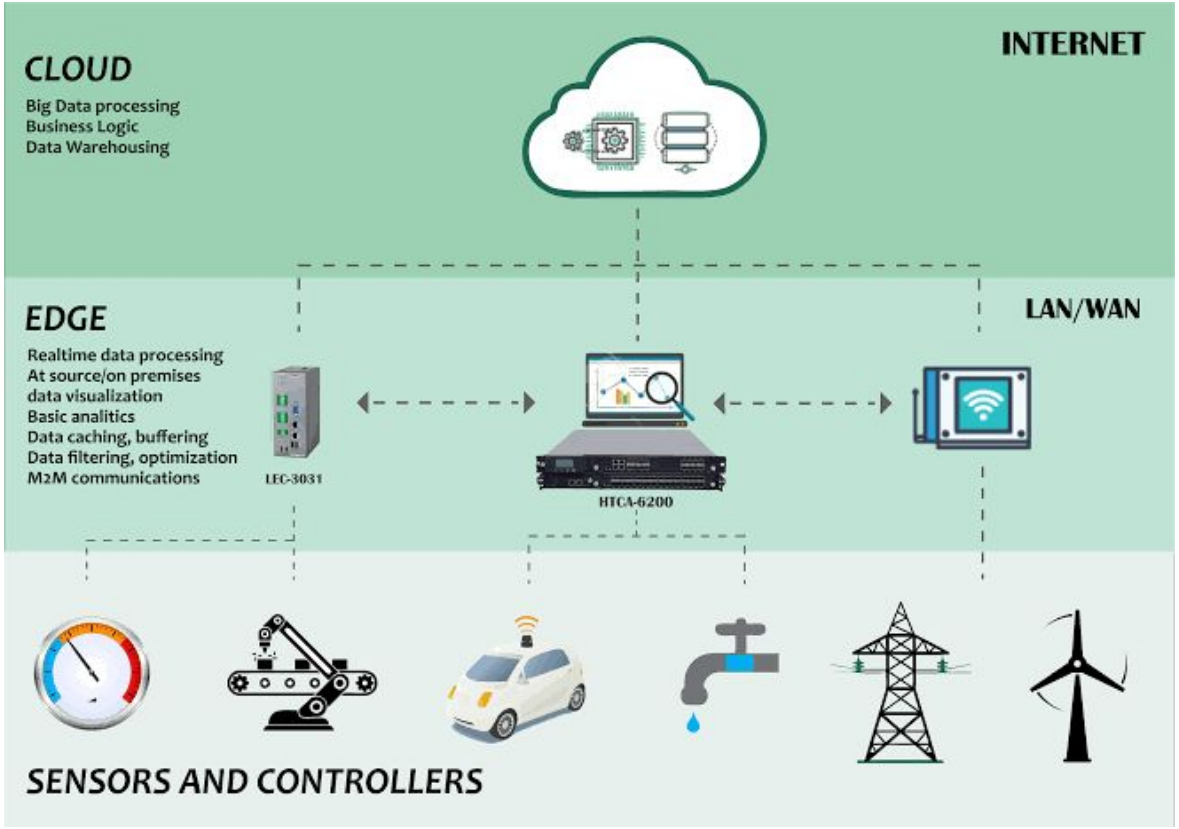
You can find microcontrollers all around, every device that calculates, stores, controls or displays information must have microcontrollers. From cell phones to smart watches to ACs to washing machines to vehicle all are the applications of microcontrollers.

Applications (cont.)

Further the following are areas where MCUs are extensively being used:

- Used in biomedical instruments
- Communication Systems
- Robotics
- Automobile Industry
- IoT Application





IoT Architecture - Edge Computing

Why Microcontroller?

All the applications mentioned in previous slides, can be implemented using just Raspberry Pi. Then why we should use microcontroller that operates een without OS?

There are 3 reasons that pushes us to use MCUs:

- Low cost
- Low power consumption
- Real time constraints

Pros

- Act as a microcomputer without any digital part.
- Usage is simple, easy to troubleshoot and system maintaining.
- Immediately respond to any event occur.
- Most of the pins are programmable by user.

Cons

- Got complex architecture than microprocessors.
- Perform limited number of executions simultaneously.
- Cannot perform heavy computations due to limited resources.


PRESIDENTIAL
INITIATIVE
ARTIFICIAL INTELLIGENCE AND COMPUTING

Rust vs C

C

- Difficult to learn for beginners at least.
- C lacks official package manager.
- C ecosystem is mature. Solution to many problems already exist.

Rust

- Easy to learn because of easy syntax (compare to C).
- Rust has Cargo.
- Less support and existing solutions available for Rust.

Chapter 3

Setting up a development environment

Documentation

Documentation is key to know nitty gritty of microcontrollers. Following are the resources we will be required much in this course:

- STM32F3DISCOVERY User Manual
- STM32F303VC Datasheet
- STM32F303VC Reference Manual
- LSM303DLHC
- L3GD20

Tools Required

Following are the tools will be required in this course.

- **Rust V - 1.31** (or newer).
- **itmdump V - 0.3.1**
- **OpenOCD** >=0.8. Tested versions: v0.9.0 and v0.10.0
- **arm-none-eabi-gdb V - 7.12** (or newer highly recommended). Tested versions: 7.10, 7.11, 7.12 and 8.1.
- **Cargo-binutils V - 0.1.4** (or newer).
- **minicom** on **Linux** and **macOS**. Tested version: 2.7.
- **PuTTY** on **Windows**.

Verification of Rust Installation

Since you are here, means you must have already installed rust. For the sake of confirmation of the version installed.

1. Open command prompt/terminal (in linux) and run this command.

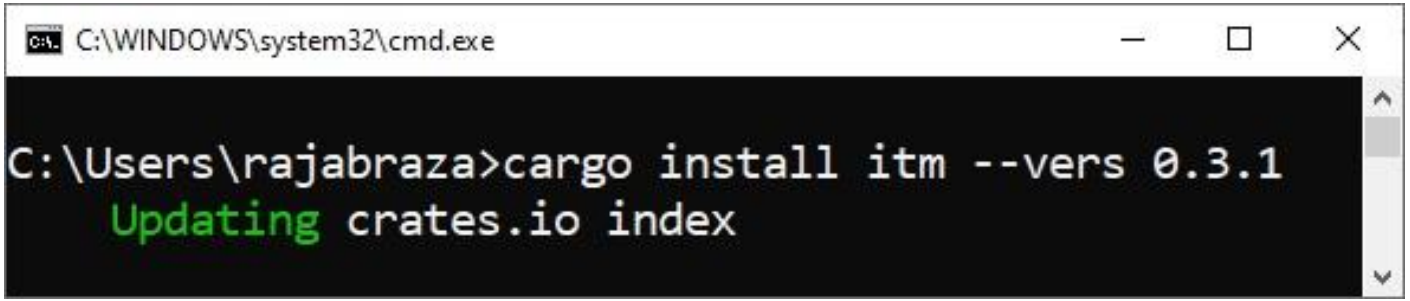
- In this machine version 1.38.0 of rust is installed. All okay here.

Itmdump

ITM -> Instrumentation Trace Macrocell (Communication Protocol)

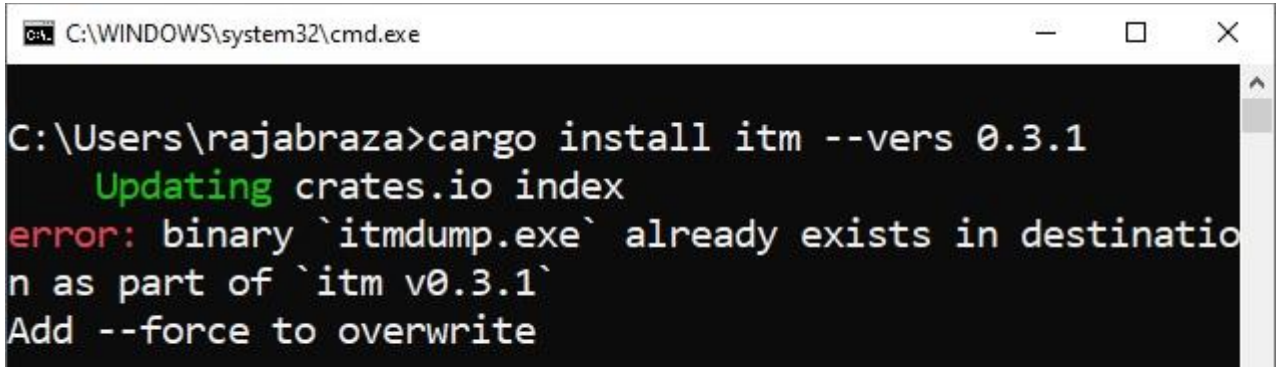
For installing itmdump you need to follow the steps below:

1. On command prompt/terminal *



itmdump

2. I have installed already, therefore i got this output. You probably got different (ending up installed successfully).

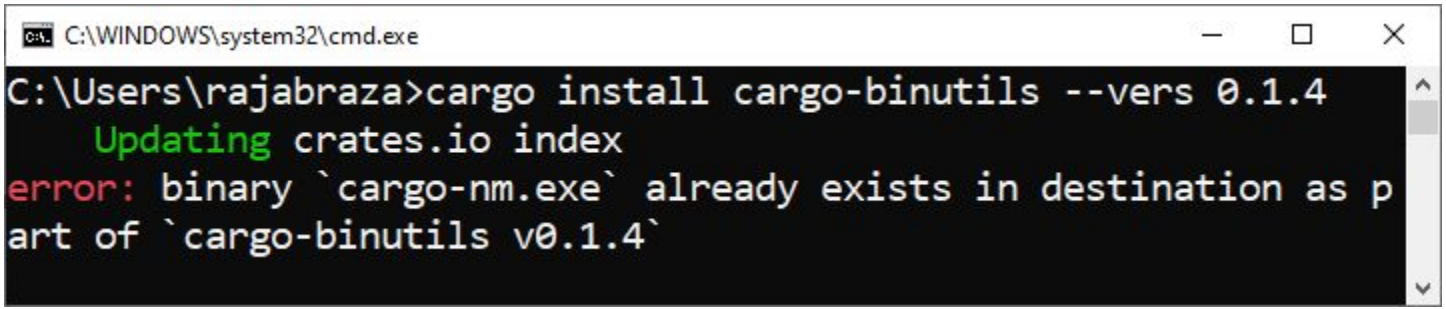


3. Verifying installation by

cargo-binutils

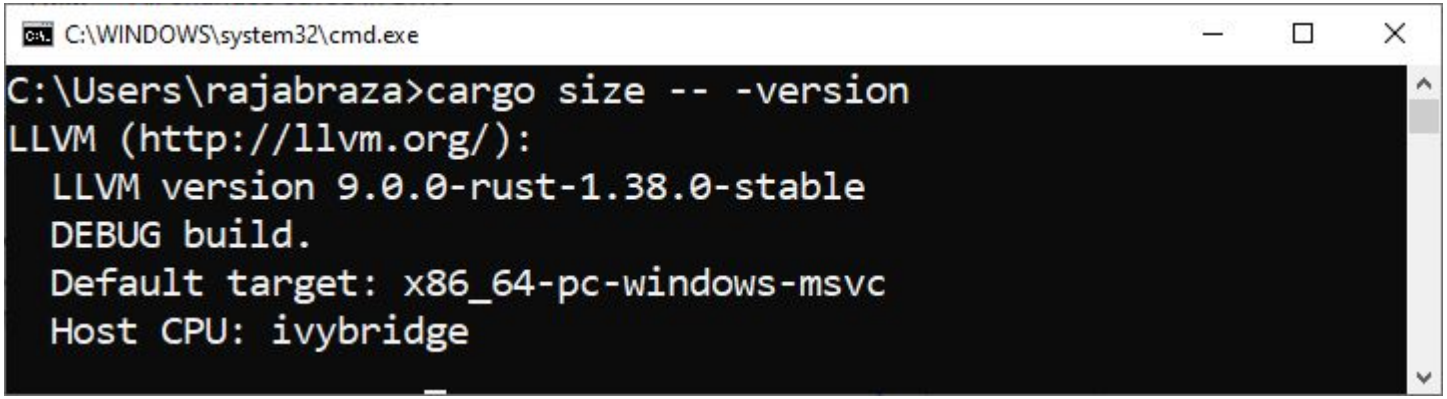
1. Before installing binary utilities, first we have to install llvm tools

2. After successful installation of llvm, we will install binary utilities.



cargo-binutils

3. After successful installation, verify it.



P.S. It's good habit to verify things once done.

Note : Also the output may slight differ depending upon your system architecture and OS.

OS Specific Instructions

OS specific instructions

So far, the instructions we performed was independent of Operating systems but now the remaining instructions will be performed are OS specific. Therefore we will cater the cases of:

1. Windows
2. Linux

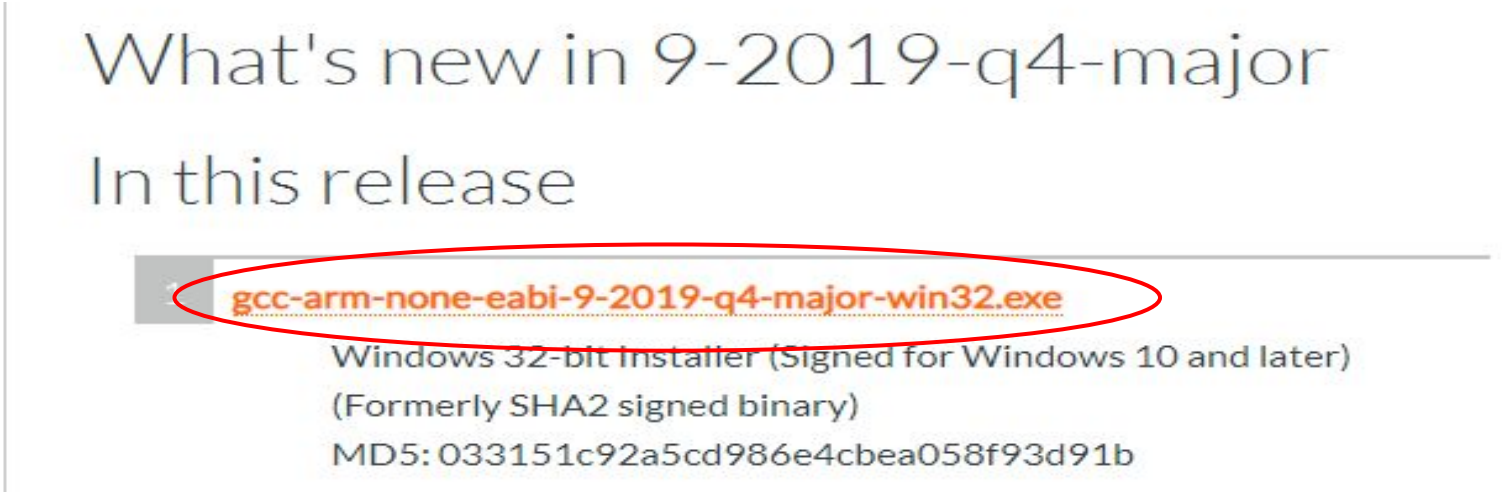
Windows

In Windows following installations will be required:

1. GDB
2. OpenOCD
3. PuTTy
4. ST-LINK Driver

arm-none-eabi-gdb

1. First thing first, visit arm website and download executable (.exe) file
2. Under release section click the highlighted link to download file.

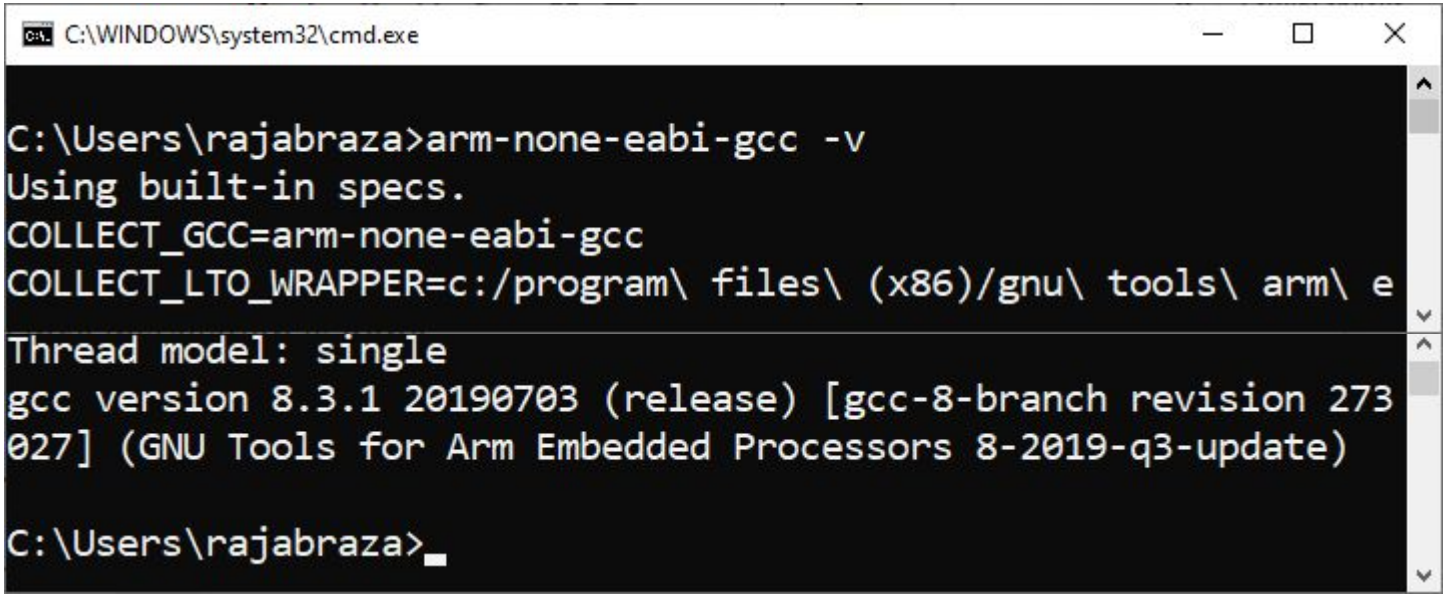


https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads

arm-none-eabi-gdb (cont)

3. Upon successful download locate the file and launch executable.

4. Once installation is done



Long output will filled-up your screen, ending up with similar lines as above.

PuTTy

Get puTTy right from here.

https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

(Not sure whether you want the 32-bit or the 64-bit version? Read the FAQ entry.)

MSI ('Windows Installer')

32-bit:	putty-0.73-installer.msi	(or by FTP)	(signature)
64-bit:	putty-64bit-0.73-installer.msi	(or by FTP)	(signature)

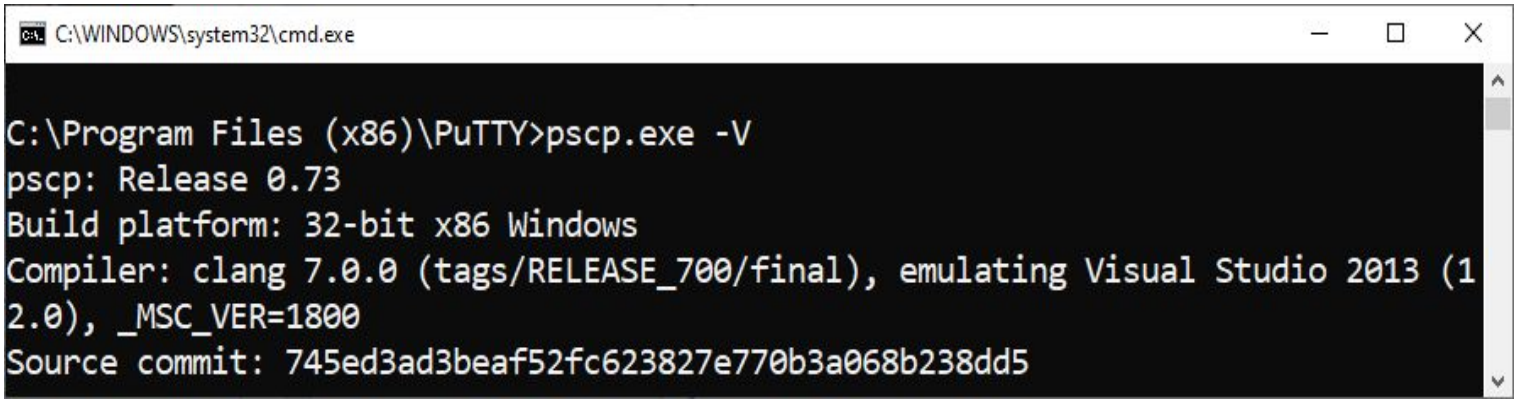
Unix source archive

.tar.gz:	putty-0.73.tar.gz	(or by FTP)	(signature)

2. Once done with download, run the setup.

PuTTy

3. Upon successful installation, verify the installation.



Note: Run the above command from the directory where PuTTy installed.

ST-LINK USB driver

Go get your driver from here.

https://www.st.com/en/development-tools/stsw-link009.html

1. Create account and download driver.
2. Extract compressed folder and launch the file “**stlink_winusb_install.bat**” as administrator.

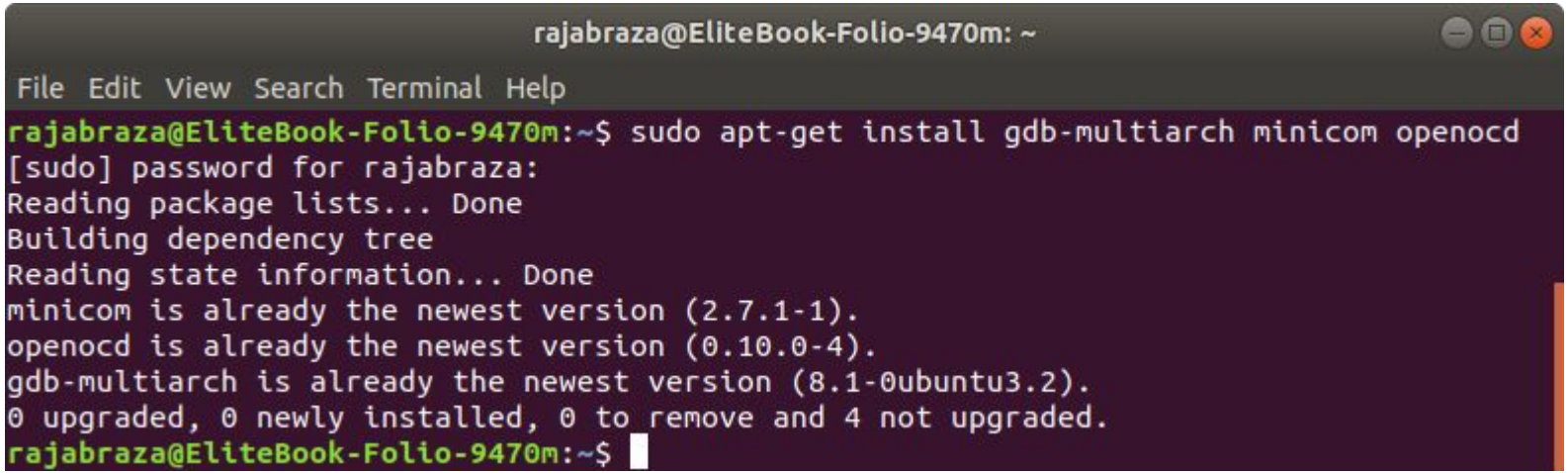
This is it, you are all set for embedded programming. It may be not that simple as it sounds. Do some extra efforts!

Linux : Required packages

In Linux following installations are required.*

1. GDB
2. OpenOCD
3. Minicom

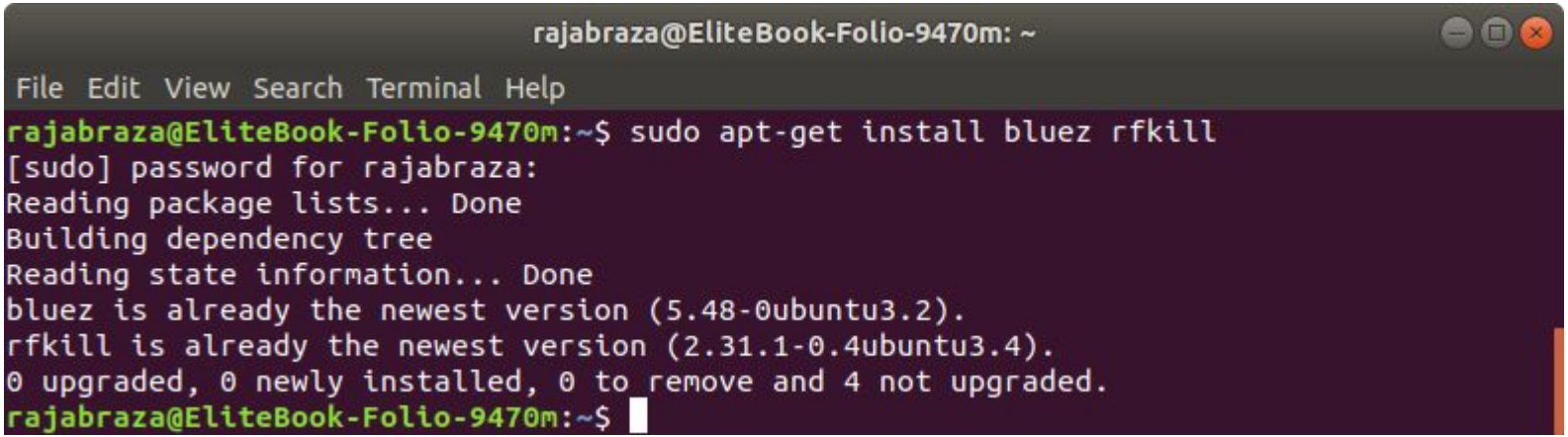
In Linux things are much simple. Like 1,2,3..



Linux : Optional packages

Following are the optional installations.

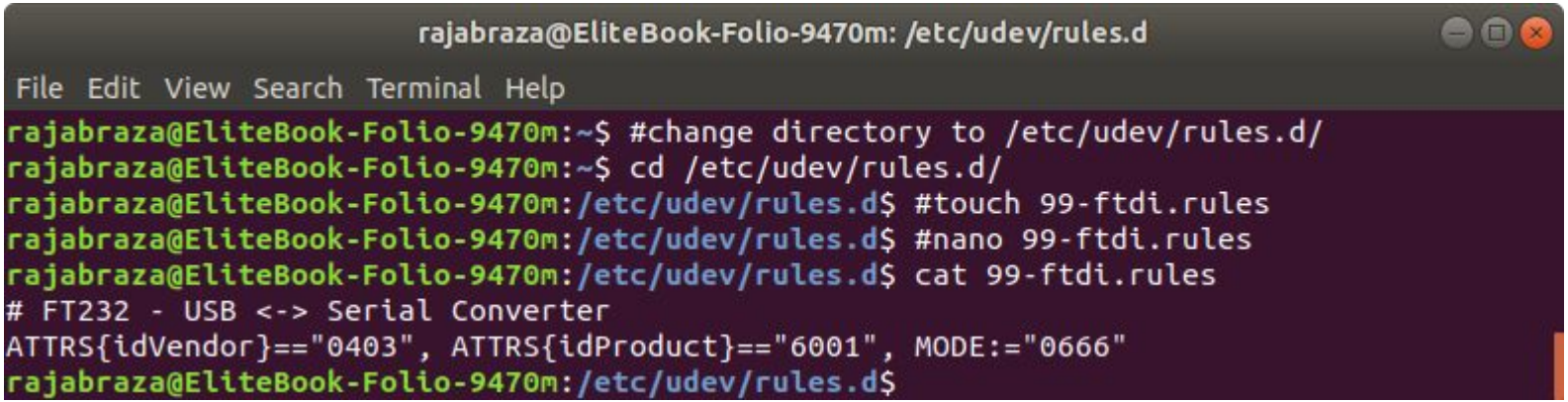
1. Bluez
2. Rfkill



Linux : udev rules

The purpose of setting up these rules is to let you use USB devices without **sudo** privilege.

1. First rule for ftdi (serial module)



Linux : udev rules

2. Second rule is for ST-LINK debugger (F3 USB Port)

```
rajabraza@EliteBook-Folio-9470m: /etc/udev/rules.d
File  Edit  View  Search  Terminal  Help
rajabraza@EliteBook-Folio-9470m:/etc/udev/rules.d$ #touch 99-openocd.rules
rajabraza@EliteBook-Folio-9470m:/etc/udev/rules.d$ #nano 99-openocd.rules
rajabraza@EliteBook-Folio-9470m:/etc/udev/rules.d$ cat 99-openocd.rules
# STM32F3DISCOVERY rev A/B - ST-LINK/V2
ATTRS{idVendor}=="0483", ATTRS{idProduct}=="3748", MODE:="0666"

# STM32F3DISCOVERY rev C+ - ST-LINK/V2-1
ATTRS{idVendor}=="0483", ATTRS{idProduct}=="374b", MODE:="0666"

rajabraza@EliteBook-Folio-9470m:/etc/udev/rules.d$
```

Finall, you just need to run one last command.

```
$ sudo udevadm control --reload-rules
```

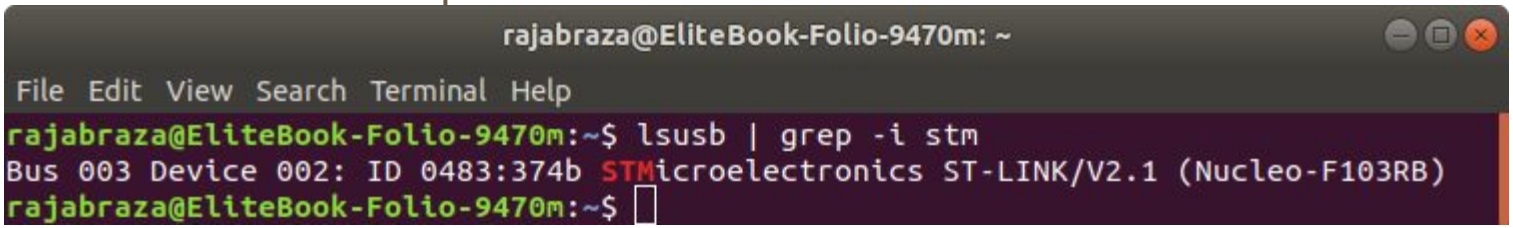
Now, you are equipped with all the tools required.

Verify the installation

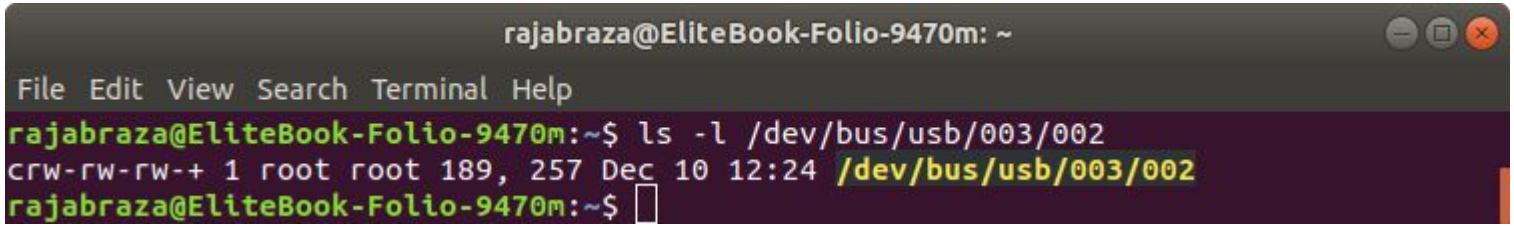
Checking F3 board permissions

Verify permission first, for that:

1. First check the port on which F3 board is connected.

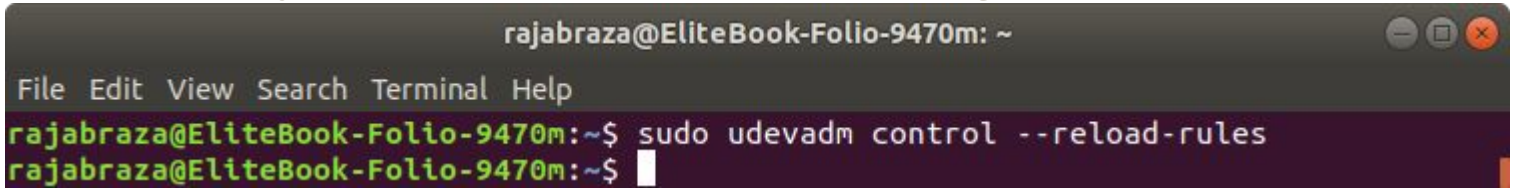


2. The above result shows F3 board connected on **bus 003** as **device 002**. Now for checking permissions for this device.



Checking F3 board permissions

3. If you get the same result then no need to do extra work, but if it differs then you need to check udev-rules again and then reload them.

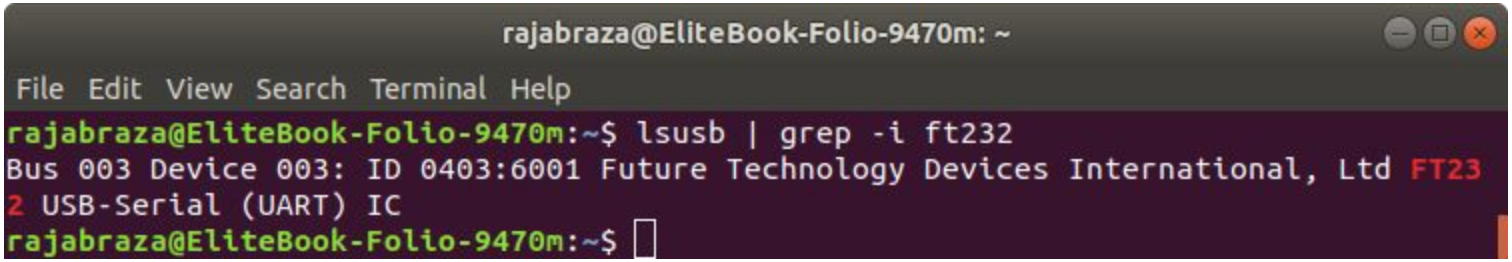


Note : Aboe command won't show any output, so after running it just repeat the steps.

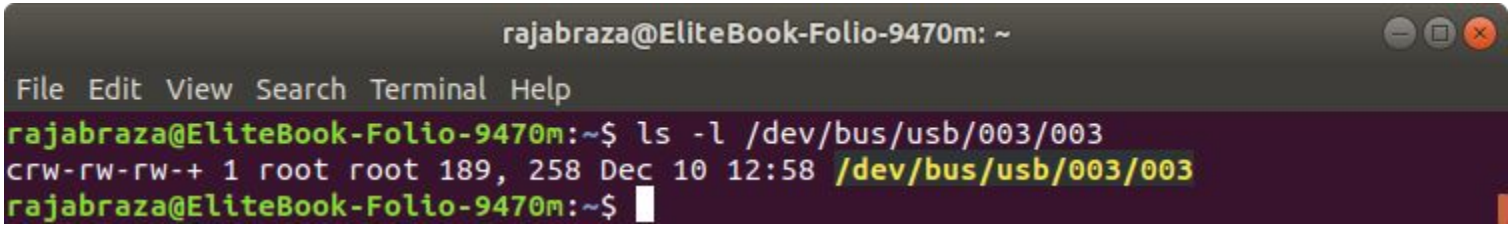
Checking ftdi permissions

Verify permission first, for that:

1. First check the port on which ftdi module is connected.



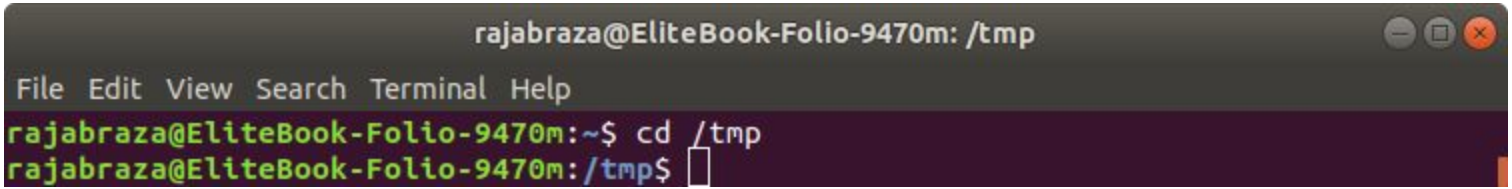
2. The above result shows ftdi module connected on **bus 003** as **device 003**. Now for checking permissions for this device.



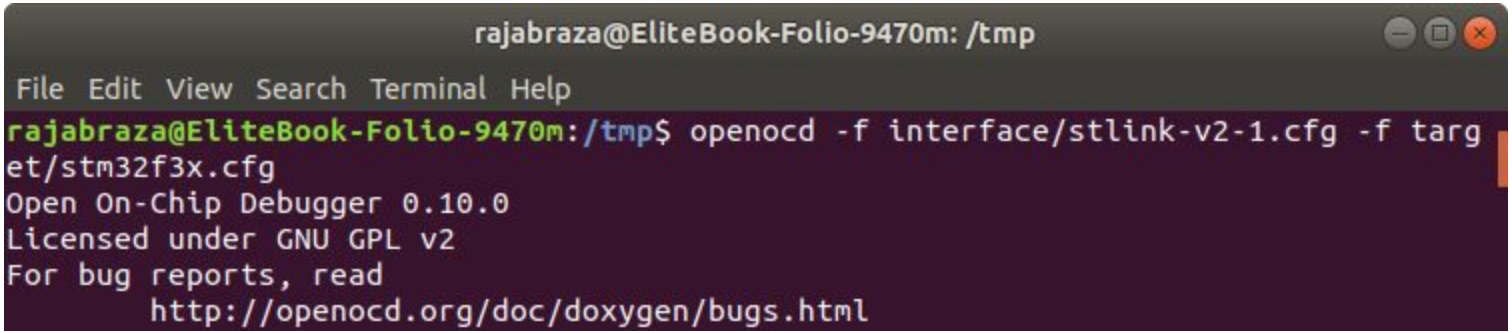
Establishing OpenOCD Connection

We have already discussed the purpose of using OpenOCD. For establishing a connection between host and debugging target.

1. Change you current directory to **/tmp**

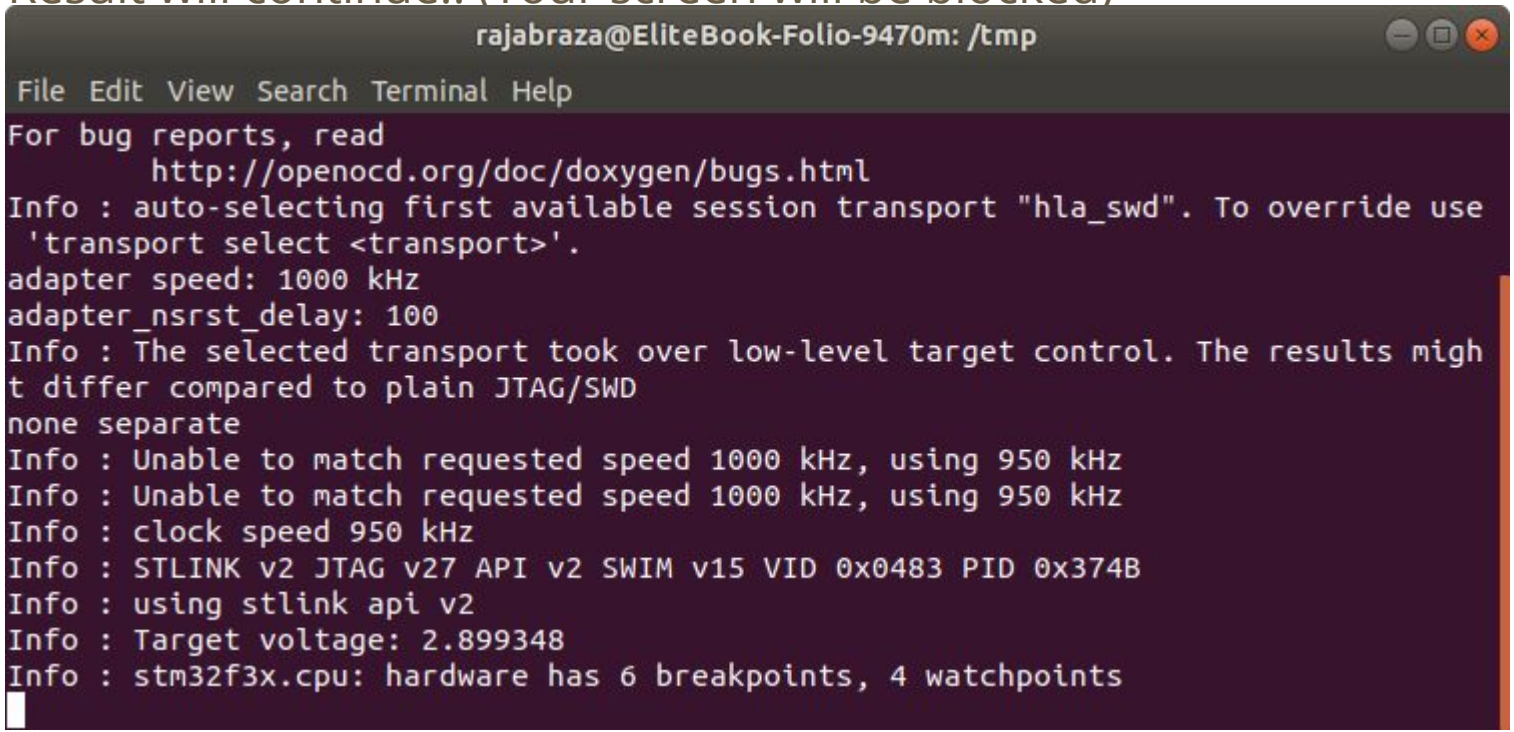


2. Next run this command to establish link between devices.



Establishing OpenOCD Connection

Result will continue.. (Your screen will be blocked)



Summary

Chapter 4

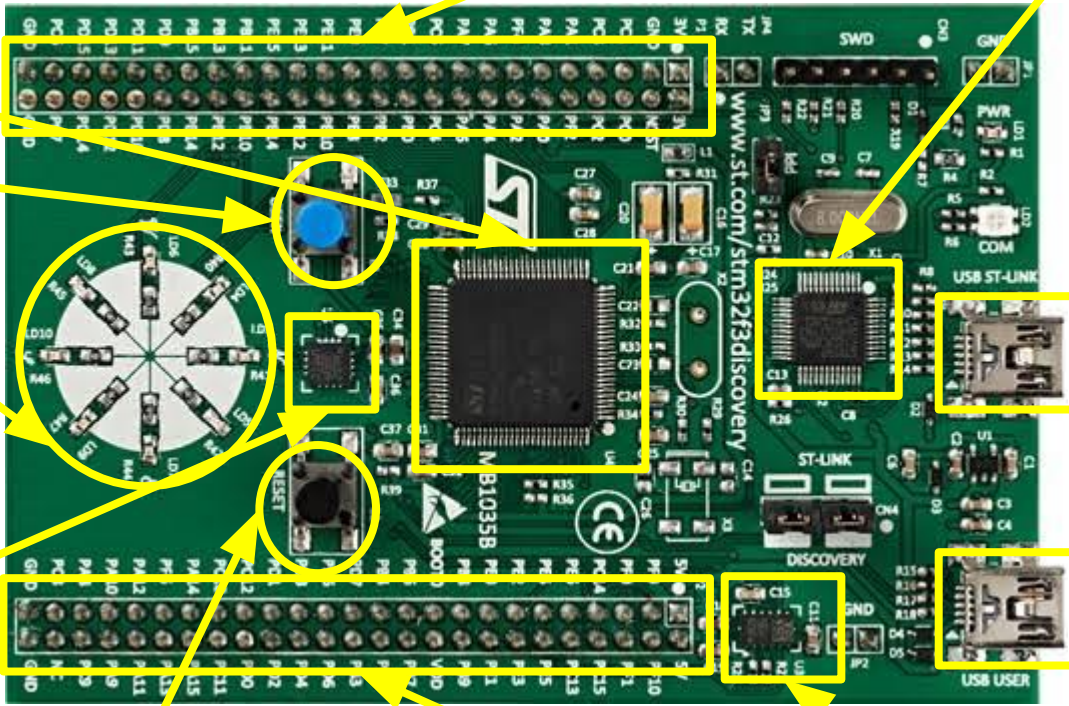
Meet your Hardware

Meet your hardware

We will be seeing the following hardware in details.

1. STM32F3Discovery (STM/ F3/ Discovery) board.
2. Serial Module
3. Bluetooth Module

STM32F3Discovery



GPIO Pins

STM32F103CBT

STM32F303VCT6

User Button

ST-LINK USB

Output LEDs

User USB

L3GD20

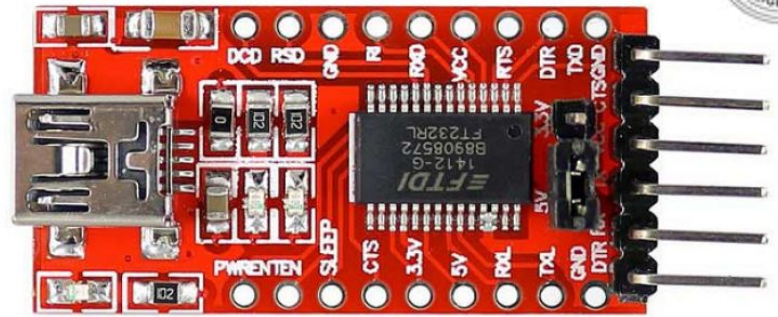
Reset Button

GPIO Pins

LSM303DLHC

Serial Module

- Primarily 2 variants are available.
- 4 Pins are common in both variants.
  - GND (Ground pin)
  - VCC (Power pin)
  - T/ TX/ TXI (Transmitter pin)
  - R/ RX/ RXI (Receiver pin)
- Uncommon pins are
  - DTR (Reset pin) -> FTDI
  - CTS (Test pin) -> FTDI
  - Another voltage pin



Bluetooth Module

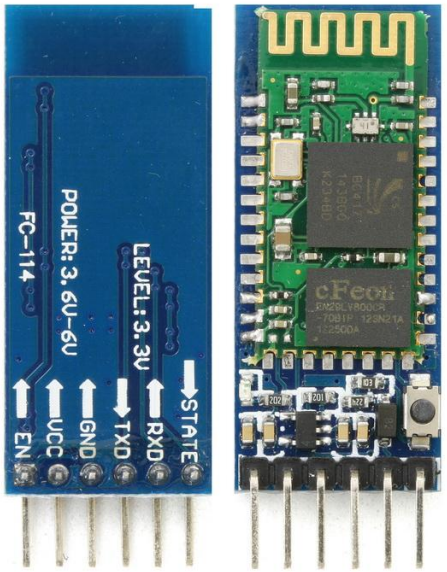
There are even 2 variants.

- HC-05
- HC-06

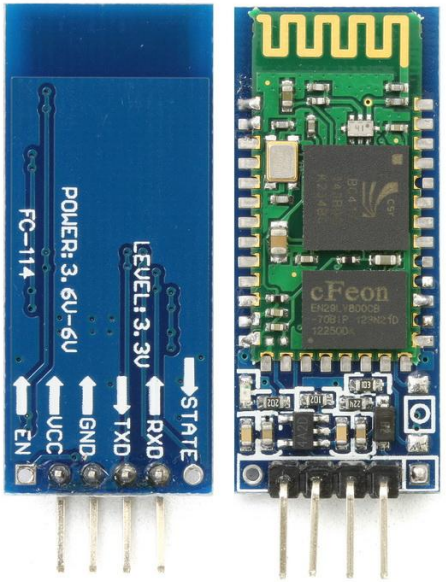
State and enable (**EN**) pins.

Defaults of these modules.

HC-05 FC-114



HC-06 FC-114



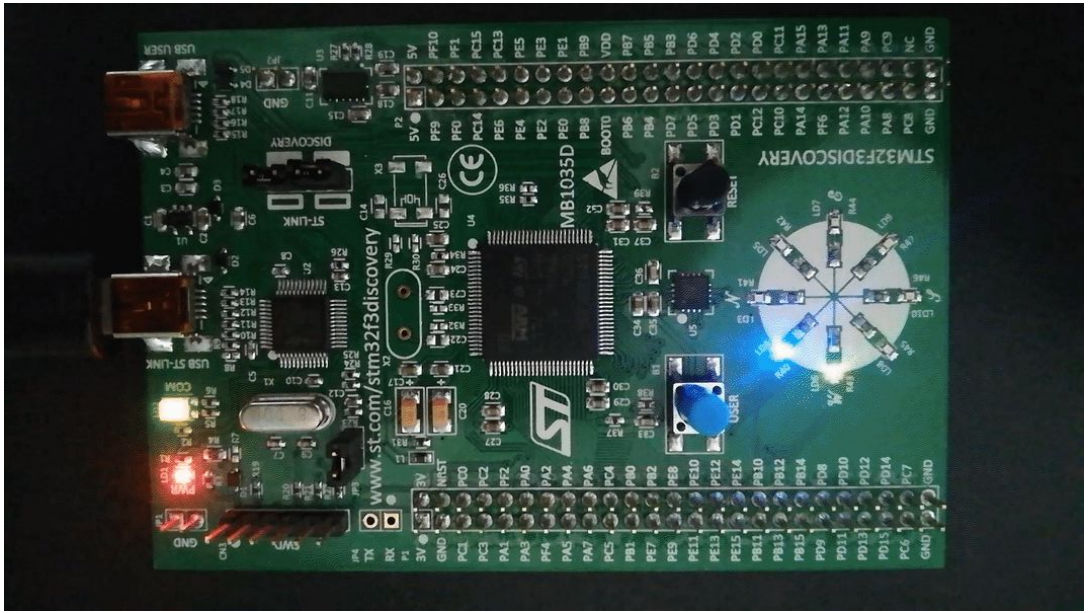
Summary

Chapter 5

LED Roulette

LED Roulette

We will implement the LED roulette program in this lecture.



Above is the result of the exercises we will be doing in this lecture.

LED Roulette

- To implement this roulette program we will be using a high level API.
- Main objective of this lecture is to get familiar with Building, Flashing and Debugging.
- We will be following code from this repository **https://github.com/PIAIC-IOT/Quarter2-Online.git**
- Programs we will be writing for microcontrollers are different from our standard programs in two ways.
  - #![no_std]
  - #![no_main]

LED Roulette

#![no_std] : no_std means this program won't use **std** crate.
#![no_main] : no_main means program won't use standard main interface, that's used for argument receiving command line apps.

Note : We named **entry point** "main" here but that could be anything. However signature of entry point function must be **fn() -> !**.

Building Program

Building Program

- ➔ After successfully writing code the very first thing is building your program.
- ➔ Since we're not building program for our computers but for a different architecture (i.e. MCU), therefore we have to cross compile.
- ➔ Cross compiling in Rust is as simple as passing an extra **--target** flag.
- ➔ The hectic part is to figuring out **the name of the target**.
- ➔ MCU in F3 has a Cortex-M4F processor and cargo knows how to cross compile to the Cortex-M architecture.

Building Program

➔ Cargo provides 4 different targets that cover the different processor families within that architecture.

- **thumbv6m-none-eabi**, for the Cortex-M0 and Cortex-M1 processors
- **thumbv7m-none-eabi**, for the Cortex-M3 processor
- **thumbv7em-none-eabi**, for the Cortex-M4 and Cortex-M7 processors
- **thumbv7em-none-eabihf**, for the Cortex-M4F and Cortex-M7F processors

➔ The one we are interested in is last one (i.e. thumbv7em-none-eabihf) because F3 has Cortex-M4F processor in it.

Building Program

- Before cross compiling we have to download pre-compiled version of the standard library. For that we have to run:

  $ rustup target add thumbv7em-none-eabihf

- We have to download it once after that it will be updated automatically whenever we update rust tool chain.
- All set, we can now compile our program for whatever target we are compiling, using following command:

  $ cargo build --target thumbv7em-none-eabihf

Flashing Program

Flashing Program

Flashing => Moving program to MCU's memory

Every time MCU will power on it will run the program flashed last time.

Steps:

1. Launch OpenOCD
2. Open GDB Server console
3. Connect GDB Server to OpenOCD
4. Load the program

Launching OpenOCD

We saw already what is OpenOCD, now let's see how it works and how to initiate it.

First we see how to initiate:

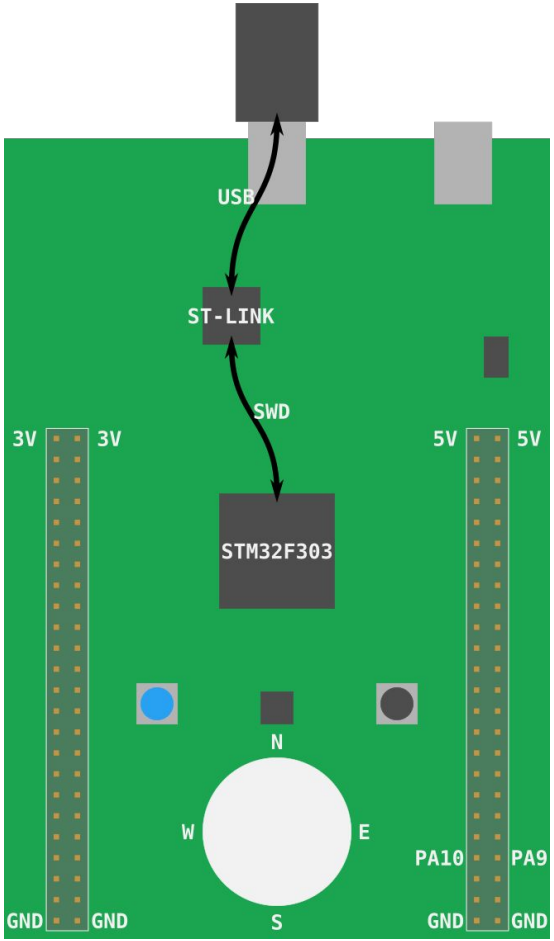
1. Open up a new terminal
2. Change directory to /temp (i.e. ***$ cd /tmp***)
3. Next run this command
   $ openocd -f interface/stlink-v2-1.cfg -f target/stm32f3x.cfg

Launching OpenOCD

Now look at the command carefully at previous slide and the image at right.

You might observed that first we are interacting with **st-link** to target **stm32f303**.

ST-LINK opens up a communication channel for us to target.



Initiating GDB Server

These commands only specific to gdb console. They have no meaning to our normal terminal/command prompt.

1. ***$*** *<gdb> -q target/thumbv7em-none-eabihf/debug/<project-name>*

   <gdb> => System dependent debugging program to read arm binaries. It has 3 ariants **gdb**, **gdb-multiarch*** and **arm-none-eabi-gdb**

   *In our case gdb-multiarch will work.

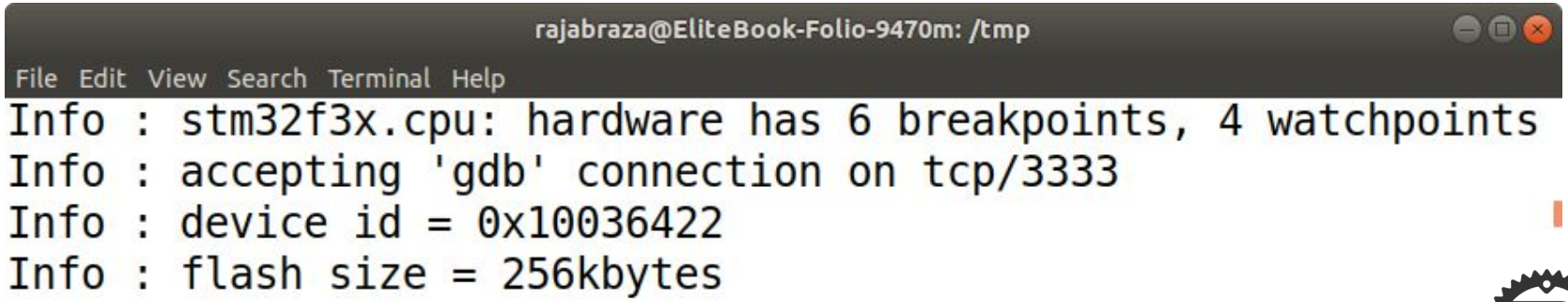
   target/thumbv7em-none-eabihf/debug/<project-name> => Path to the binaries of projects we want to flash in MCU.

Connecting OpenOCD

Previous command only opens gdb shell. To connect to Openocd GDB Server run the following command.

2. ***(gdb) target remote :3333***

   This **3333** highlights the port number on which GDB server listens requests. And this command connects us to this port.

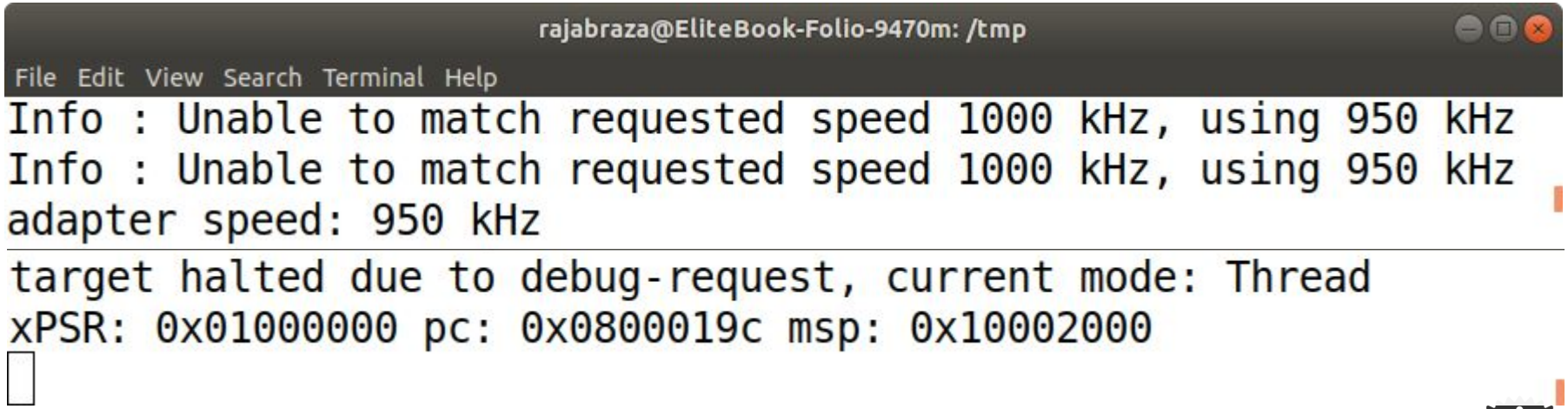


   Above is the result of this command.

Loading Program

Once we connected successfully, now final step to flash our code to MCU's persistent memory.

3. ***(gdb) load***



Debugging Program

Debugging Program

After **load** command our program stopped at **entry point**.

There are some helpful commands used for debugging purpose.

- break
- continue

break => break command is used to break program at certain point.
For example: if we want to stop program at the beginning of main function then we will run: ***(gdb) break main***

continue => this command will take us from one breakpoint to another breakpoint.

Debugging Program

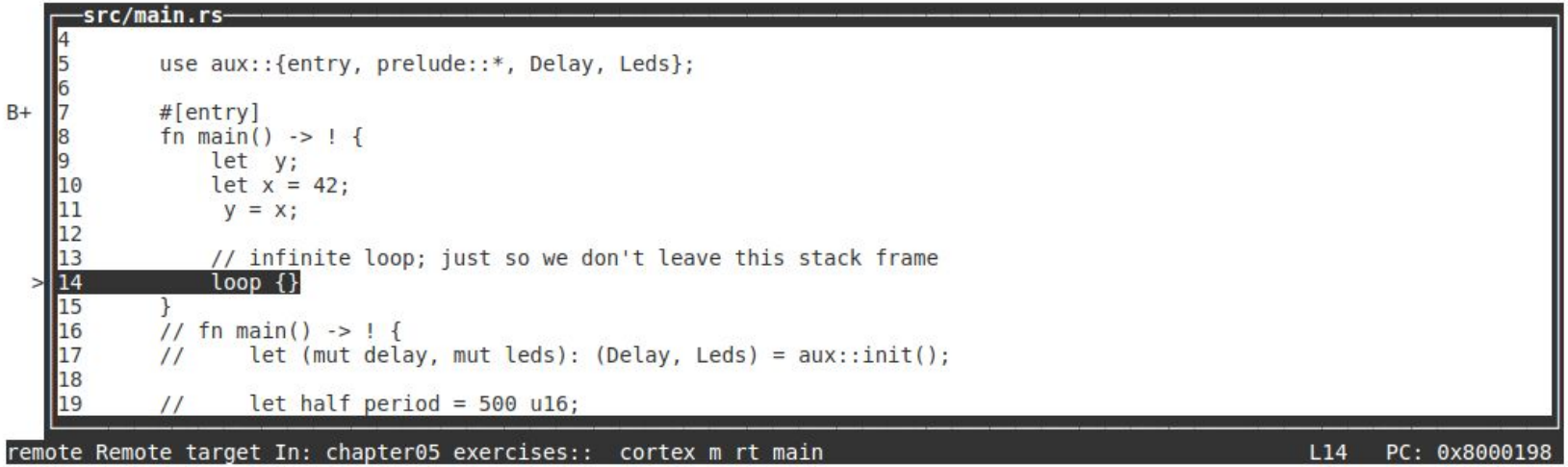
For viewing line by line execution of program we can switch to GDB Text User Interface (TUI), for that run:
(gdb) layout src *(output on next slide)*

For disabling TUI mode we can run:
(gdb) tui disable

For resetting the program to initial state while debugging run:
(gdb) monitor reset halt

For terminating GDB session
(gdb) quit

Debugging Program



Debugging Program

Further commands:

- **step** : for moving to next line
- **print** : for printing values of variables
- **Info locals** : for printing all values of variables at once
- **Clear shell** : for clearing TUI screen

LED Program

LED Program

- So far we have learnt the basics of the embedded application life-cycle.
- Now we will head to the application we started with desire of, LED Roulette.
- For this program we will use some abstrations, which are already written in the **library**.
- The library function will provide us **LEDs** and **Delays.**
- Library gives us two functions to use with LEDs; **on()** and **off()**
- Also it gives us a function for Delay; **delay_ms()**

Alias

Summary

Chapter 6

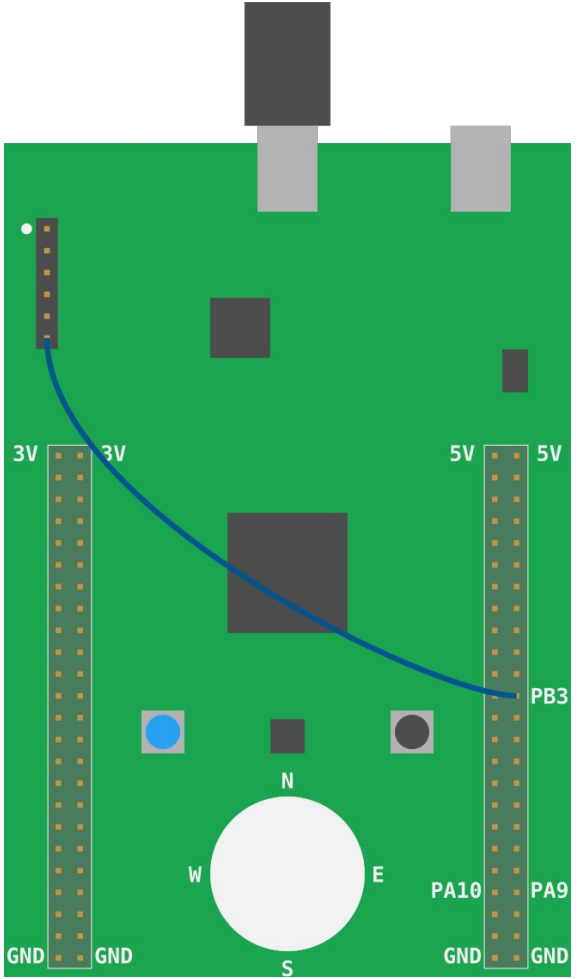
Hello World from F3

Hello World

- In this short lesson we will see how to send message to debugging host from F3
- ITM (Instrumentation Trace Macrocell) is responsible for this message sending/communication
- ITM is a communication protocol
- It's a unidirectional protocol that can be used to send message to debugging host but can't send message to MCU from host
- Being the debugging session manager, OpenOCD facilitates this message passing, received message and forward it to a file.
- **Use-case** : Application status update, logging, etc

Hello World

- ➔ To enable ITM we have fix an issue
- ➔ Connect SWO and PB3 pins (as shown)
- ➔ Once it’s done you are ready to code for this exercise



Hello World

Sample Code:

```
#![deny(unsafe_code)]
#![no_main]
#![no_std]

#[allow(unused_imports)]
use aux::{entry, iprint, iprintln};

#[entry]
fn main() -> ! {

    let mut itm = aux::init();
    iprintln!(&mut itm.stim[0], "Hello, world!");

    loop {}
}
```

Extra Work

After writing code and making an explicit connection few additional things will also needed to be done.

1. First thing first, we have to install **itmdump** if didn't installed in the initial lesson of Installation and configuration
2. After building and before executing program we have to open an additional(beside OpenOCD's terminal) terminal in **/tmp** directory
3. Creating a file (i.e. itm.txt) in /tmp directory
4. After creation of file creating a watch on the file (created in step 3)

Extra Work

5. Next we will build the program, flashing it and perform steps ... the point where our program stops at first breakpoint.
6. After that we will execute 2 additional commands specific to ITM
   - **(gdb) monitor tpiu config internal itm.txt uart off 8000000**
   - **(gdb) monitor itm port 0 on**
7. Next we will execute our program and once we passed this line:

```
iprintln!(&mut itm.stim[0], "Hello, world!");
```

8. We will see that on terminal we opened for itmdump, a new line appended saying "**Hello, world!**"

Panic!

- Not only **iprintln!**, but we can also use panic! macro
- Panic! macro is used for forcefully terminating the program in case of any undesired behavior of program.
- Sample code:

Summary

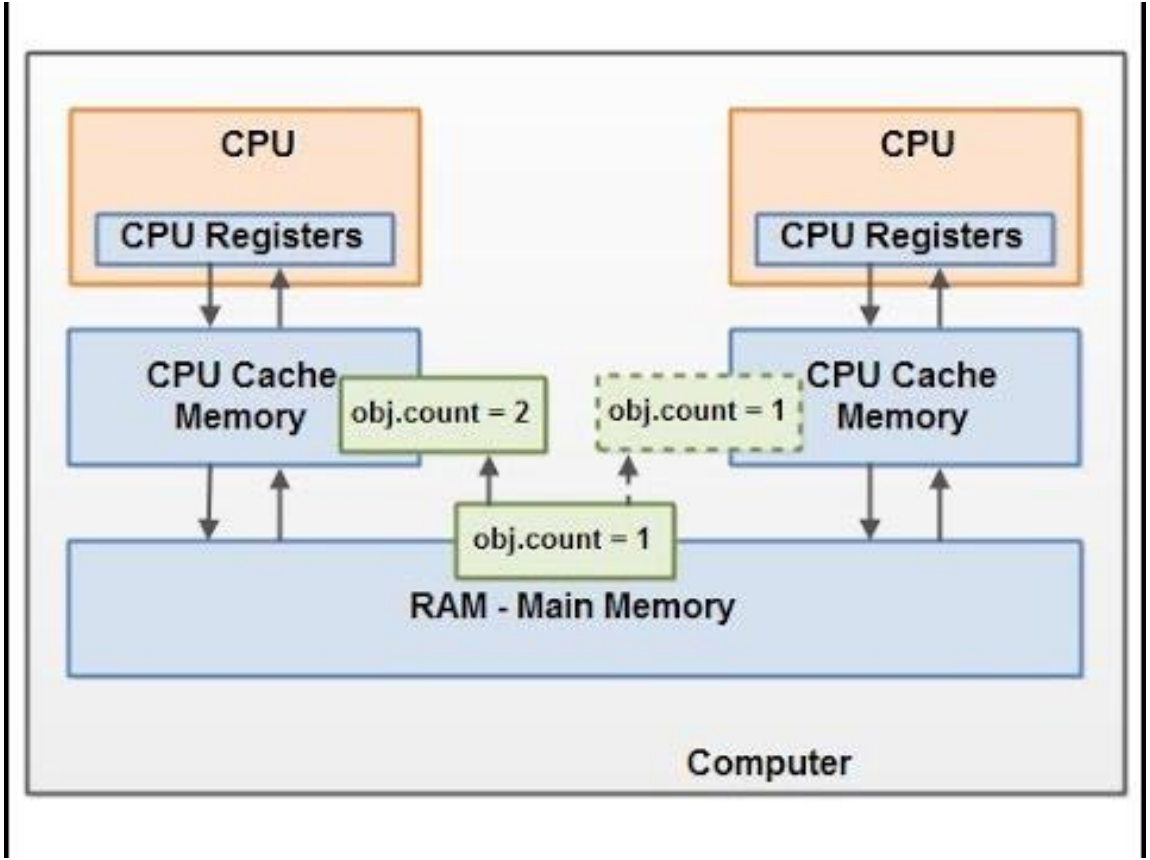
Chapter 7

Registers

Registers

- Registers are type of computer memory
- They are reside near to CPU and used immediately by CPU
- Can hold data (bits of sequence), storage address (pointers) or instructions.
- Registers holding the memory location are used to calculate the address of the next instruction.
- In this chapter we will revisit the LED's program, but this time we will see things in a bit depth
- In terms of MCU's registers are special memory regions

Registers


CPU
CPU Registers
CPU
CPU Registers
CPU Cache Memory
obj.count = 2
obj.count = 1
CPU Cache Memory
obj.count = 1
RAM - Main Memory
Computer

Registers

- These special regions map to or control some peripherals
- And we already know we have many of of them in our board
- These peripherals are electronic components built in the MCU to provide it's processor the ability to interact with IO devices.
- We will be following code from this repository

https://github.com/PIAIC-IOT/Quarter2-Online.git

Registers

- Here is the starter code:

```
#[allow(unused_imports)]
use aux::{entry, iprint, iprintln};

#[entry]
fn main() {
    aux::init();

    unsafe {

        const GPIOE_BSRR: u32 = 0x48001018;

        (GPIOE_BSRR as *mut u32) = 1 << 9;
        (GPIOE_BSRR as *mut u32) = 1 << 11;
        (GPIOE_BSRR as *mut u32) = 1 << (9 + 16);
        (GPIOE_BSRR as *mut u32) = 1 << (11 + 16);
    }
    loop {}
}
```

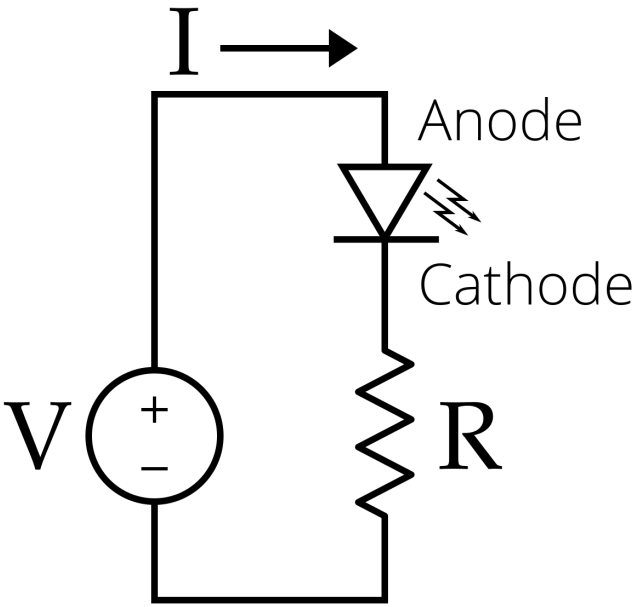
Registers

- We are using raw pointer here for demonstration purpose.
  It's not recommended to use in normal programs btw except in genuine case
- Address (i.e. **0x48001018**) points to a register
- This register controls **GPIO** pins. **GPIO** is a peripheral
- Also using GPIO we can drive the mapped pins to low or high

Light Emitting Diode

Light Emitting Diode (LED)

- Pins are electrical contacts and our MCU has many of them
- Few of them are connected to LEDs, few are connected to other devices on board
- Once a pin connected to an LED in right polarity and a high voltage applied to that pin that's the only case LED will glow



Reading The Reference Manual

RTRM

Whenever we start learning a new programming language, using a new hardware or buy a new cell phone. It's documentation/guide is **must read/follow** thing to take full advantage of that.

- Manual says that MCU has several pins.
- These pins are **grouped in ports** of 16 pins
- Each port named with a letter (e.g. A, B, C, .. , H)
- Pins within each port named with a number (e.g. 0, 1, 2, .. , 15)

Since we are interested in finding pins associated with LEDs, so lets check the manual to find out those pins.

RTRM

On page 18, under section 6.4 (LEDs) it is mention which LED connected to which pin.

- User LD3: red LED is a user LED connected to the I/O PE9 of the STM32F303VCT6.
- User LD4: blue LED is a user LED connected to the I/O PE8 of the STM32F303VCT6.
- User LD5: orange LED is a user LED connected to the I/O PE10 of the STM32F303VCT6.
- User LD6: green LED is a user LED connected to the I/O PE15 of the STM32F303VCT6.
- User LD7: green LED is a user LED connected to the I/O PE11 of the STM32F303VCT6.

Since we are interested in LD3 and LD7 as we are using in our program.

RTRM

- Manual says LD3 connected to PE9 and LD7 connected to PE11
- Here “PE” in PEXX represents the port name (i.e. Port E of GPIO peripheral)
- The number after the port name represents pin number in the port.

Conclusion:

After this exercise we find out what pins we need to **low**/**high** to **on**/**off** LEDs.

RTRM

- Each peripheral has a register block associated to it.
- A register block is a **collection of registers** allocated in **memory contiguously**
- The starting address of this block called **base address**
- Now we know to driving LEDs we need to manipulate with Port E registers. For that we need to find out the base address of Port E.

For finding base address of Port E we have to follow the reference manual

RTRM

- On page 51, under section 3.2.2 (Memory map and register boundary addresses) boundary addresses of Port E mentioned.

3.2.2 Memory map and register boundary addresses

See the datasheet corresponding to your device for a comprehensive diagram of the memory map.

The following table gives the boundary addresses of the peripherals available in the devices.

Table 2. STM32F303xB/C and STM32F358xC peripheral register boundary addresses(1)

Bus	Boundary address	Size (bytes)	Peripheral	Peripheral register map
AHB3	0x5000 0400 - 0x5000 07FF	1 K	ADC3 - ADC4	*Section 15.6.4 on page 410*
	0x5000 0000 - 0x5000 03FF	1 K	ADC1 - ADC2	
	0x4800 1800 - 0x4FFF FFFF	~132 M	Reserved	
	0x4800 1400 - 0x4800 17FF	1 K	GPIOF	
	0x4800 1000 - 0x4800 13FF	1 K	GPIOE	
	0x4800 0C00 - 0x4800 0FFF	1 K	GPIOD	

RTRM

At this point we found the base address of Port E (0x48001000)

- Now we need to target a register that will let us set/reset the bits of Port E.
- That register is BSRR => Bit Set Reset Register.
- For even that we need to consult manual

Section 11.4.7 GPIO port bit set/reset register (GPIOx_BSRR) - Page 240

RTRM

11.4.7 GPIO port bit set/reset register (GPIOx_BSRR) (x = A..H)

Address offset: 0x18

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Bits 31:16 **BRy:** Port x reset bit y (y = 0..15)

These bits are write-only. A read to these bits returns the value 0x0000.

0: No action on the corresponding ODRx bit
1: Resets the corresponding ODRx bit

Note: If both BSx and BRx are set, BSx has priority.

Bits 15:0 **BSy:** Port x set bit y (y= 0..15)

These bits are write-only. A read to these bits returns the value 0x0000.

0: No action on the corresponding ODRx bit
1: Sets the corresponding ODRx bit

RTRM

- Documentation says it is **write-only** register, which means we can write to this register only but can't read from it.
- To verify this we will code and try to read from this register

```
// We'll use GDB's examine command: x.
(gdb) next
16                  *(GPIOE_BSRR as *mut u32) = 1 << 9;

(gdb) x 0x48001018
0x48001018:     0x00000000
```

- Reading the register return us **0**

RTRM

Few other informations we extracted from the documentation:

- Bits **0-15** can be used to set the corresponding pins (i.e. bit 0 sets pin 0 and bit 15 sets bit 15)
- Also, bits **16-31** can be used to reset the corresponding pins

Correlating that information with our program, all seems to be in agreement:

- Writing **1 << 9** (BS9 = 1) to BSRR sets PE9 high
- Writing **1 << 11** (BS11 = 1) to BSRR sets PE11 high
- Writing **1 << 25** (BR9 = 1) to BSRR sets PE9 low
- Finally, writing **1 << 27** (BR11 = 1) to BSRR sets PE11 low

mis(Optimization)

mis(Optimization)

- Read/write to registers are special operations
- Have some side-effects
- We wrote 4 different values to the same address
- If not knowing upfront we end up writing only **1 << (11 + 16)**
- LLVM doesn't know that we're writing to register
- It optimizes the code and will merge all the writes
- For verifying this we can run **cargo run --release**
- After running the only line in the code we run this command **disassemble /m**

mis(Optimization)

```
10
11          unsafe {
12              // A magic address!
13              const GPIOE_BSRR: u32 = 0x48001018;
14
15              // Turn on the "North" LED (red)
16              *(GPIOE_BSRR as *mut u32) = 1 << 9;
17
18              // Turn on the "East" LED (green)
19              *(GPIOE_BSRR as *mut u32) = 1 << 11;
20
21              // Turn off the "North" LED
22              *(GPIOE_BSRR as *mut u32) = 1 << (9 + 16);
23
24              // Turn off the "East" LED
25              *(GPIOE_BSRR as *mut u32) = 1 << (11 + 16);
=> 0x08000198 <+16>:    str     r1, [r0, #0]

26          }
27
```

mis(Optimization)

- We will observe this time LED won't change its state
- The noted **str** instruction is the one which writes the values to the register
- Our debug (unOptimized) version has four of them
- To verify this we will run this command **cargo objdump --bin registers -- -d -no-show-raw-insn -print-imm-hex -source**

Solution

- There is a solution for this mis(Optimized) problem
- That is **volatile operation** rather than plain read/write operation
- This will prevent LLVM from optimizing our program
- Below is the code we need to substitute in our program

```
ptr::write_volatile(GPIOE_BSRR as *mut u32, 1 << 9);

ptr::write_volatile(GPIOE_BSRR as *mut u32, 1 << 11);

ptr::write_volatile(GPIOE_BSRR as *mut u32, 1 << (9 + 16));

ptr::write_volatile(GPIOE_BSRR as *mut u32, 1 << (11 + 16));
```

Solution

- Now if we run the following command, we will observe that our program has 4 **str** instructions in optimized version.
- **cargo objdump --bin registers --release -- -d -no-show-raw-insn -print-imm-hex -source**

Solution

```
Disassembly of section .text:
main:
; #[entry]
 8000188:       bl      #0x22
; aux7::init();
 800018c:       movw    r0, #0x1018
 8000190:       mov.w   r1, #0x200
 8000194:       movt    r0, #0x4800
 8000198:       str     r1, [r0]
 800019a:       mov.w   r1, #0x800
 800019e:       str     r1, [r0]
 80001a0:       mov.w   r1, #0x2000000
 80001a4:       str     r1, [r0]
 80001a6:       mov.w   r1, #0x8000000
 80001aa:       str     r1, [r0]
; loop {}
 80001ac:       b       #-0x4 <main+0x24>
```

0xBAAAAAAD Address

0xBAAAAAAD Address

- Not all the memory addresses can be accessed
- Look at the code below, in this if we try to access this address

```
unsafe {
        ptr::read_volatile(0x4800_1800 as *const u32);
    }
```

- This address is close to GPIOE_BSRR register's address but this one is invalid
- There is no register at this address
- If we run this code, we will get the following error

0xBAAAAAAD Address

```
$ cargo run
Breakpoint 3, main () at src/07-registers/src/main.rs:9
9           aux7::init();

(gdb) continue
Continuing.

Breakpoint 2, UserHardFault_ (ef=0x10001fc0)
    at $REGISTRY/cortex-m-rt-0.6.3/src/lib.rs:535
535         loop {
```

- This exception raised because we tried to access an address that doesn’t exist.

Exceptions

- Exceptions are raised when processor tries to perform an invalid operation
- Exception breaks the normal flow of our program
- It forces the processor to execute an exception handler
- Exception handler is just a function/subroutine
- There different kind of exceptions
- Each one raised by different condition
- Raised exception even handled by different exception handler

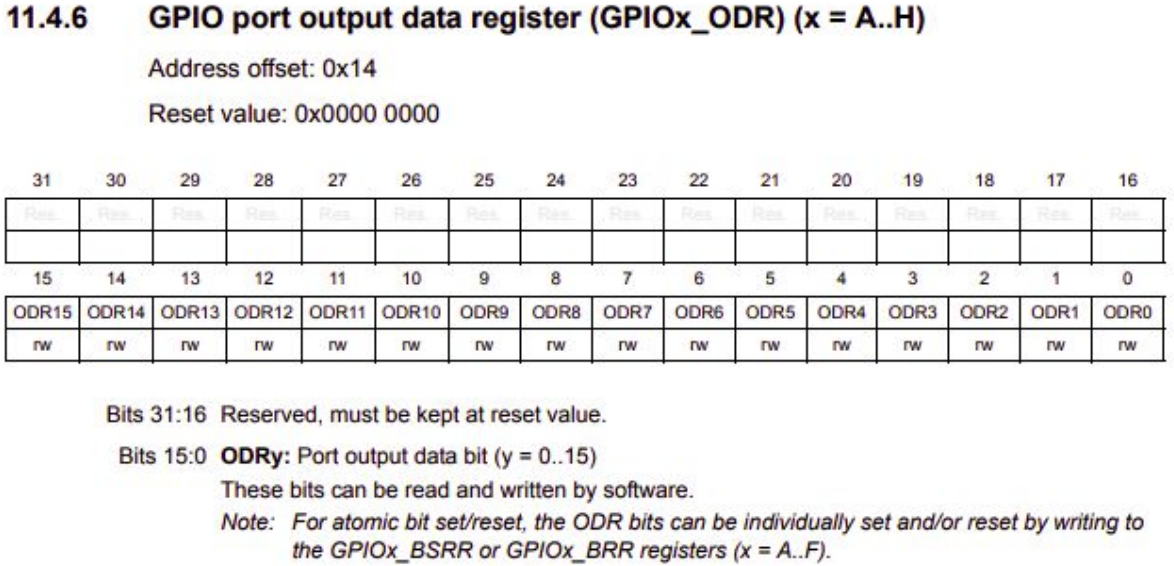
Spooky action at a distance

Spooky action at a distance

- **BSRR** is not the only register that can control the pins
- **ODR** is the register that give you read/write access to the port's pins
- For **ODR** register details, we will consult to the manual again

Section 11.4.6 GPIO port output data register - Page 239

Spooky action at a distance



11.4.6 GPIO port output data register (GPIOx_ODR) (x = A..H)

Address offset: 0x14

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ODRy:** Port output data bit (y = 0..15)

These bits can be read and written by software.

Note: For atomic bit set/reset, the ODR bits can be individually set and/or reset by writing to the GPIOx_BSRR or GPIOx_BRR registers (x = A..F).

- Lets try running the code below in the description, which writes to BSRR and reads from ODR register.

Spooky action at a distance

We will be getting the following result on the **itmdump console**

```
$ # itmdump's console
(..)
ODR = 0x0000
ODR = 0x0200
ODR = 0x0a00
ODR = 0x0800
```

Type safe manipulation

Type safe manipulation

- Last register we were working with was ODR register, and documentation says:

> Bits 16:31 Reserved, must be kept at reset value

- So, we can not write to these bits otherwise something unexpected may happen
- Peripheral ports in MCU have many registers and each one has different read/write permissions
- Directly working with hexadecimal addresses is error prone

API Usage

- Unintentionally, if tried to access an invalid memory location it will break our program
- So wouldn't it be nice if we write code in safe manner using API
- The API will then take care of the concerns we have while directly interacting with registers
- We won't be dealing with addresses directly
- We won't have to worry about read/write permissions
- We also won't be hitting the reserved area of registers
- Now lets see how we do it

API Usage

```
#[entry]
fn main() -> ! {
    let gpioe = aux7::init().1;
    // Turn on the North LED
    gpioe.bsrr.write(|w| w.bs9().set_bit());
    // Turn on the East LED
    gpioe.bsrr.write(|w| w.bs11().set_bit());
    // Turn off the North LED
    gpioe.bsrr.write(|w| w.br9().set_bit());
    // Turn off the East LED
    gpioe.bsrr.write(|w| w.br11().set_bit());
    loop {}
}
```

API Usage

- If you observed there is no magic address now
- Also it is more human readable way to access BSRR using **gpioe.bsrr**
- Then we are having **write** closure that responsible for setting/resetting the bits of register
- Another thing to notice that we have 2 methods **bsx()** and **brx()** to set the values

Summary

LEDs Again

The Embedded Discovery Book

Chapter 8

LEDs again

In the last section, I gave you initialized (configured) peripherals (I initialized them in aux7::init). That's why just writing to BSRR was enough to control the LEDs. But, peripherals are not initialized right after the microcontroller boots.

LEDs again

In this section, you'll have more fun with registers. I won't do any initialization and you'll have to initialize configure GPIOE pins as digital outputs pins so that you'll be able to drive LEDs again.

GPIOE

gpioe.odr.write

LEDs again / Power

```
p/x *gpioe
odr: stm32f30x::gpioc::ODR {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0 }
      }
    }
```

Power

- **Turns out that, to save power, most peripherals start in a powered off state -- that's their state right after the microcontroller boots.**
- **The Reset and Clock Control (RCC) peripheral can be used to power on or off every other peripheral.**

Power

You can find the list of registers in the RCC register block in:

Section 9.4.14 - RCC register map - Page 166 - Reference Manual

The registers that control the power status of other peripherals are:

- **AHBENR**
- **APB1ENR**
- **APB2ENR**

Power

The registers that control the power status of other peripherals are:

- **AHBENR**
- **APB1ENR**
- **APB2ENR**

Each bit in these registers controls the power status of a single peripheral, including GPIOE.

Configuration

After turning on the GPIOE peripheral. The peripheral still needs to be configured. In this case, we want the pins to be configured as digital outputs so they can drive the LEDs; by default, most pins are configured as digital inputs.

Configuration

You can find the list of registers in the GPIOE register block in:

Section 11.4.12 - GPIO registers - Page 243 - Reference Manual

The register we'll have to deal with is: MODER.

Configuration

```
    gpioe.moder.modify(|_, w| {
        w.moder8().output();
        w.moder9().output();
        w.moder10().output();
        w.moder11().output();
        w.moder12().output();
        w.moder13().output();
        w.moder14().output();
        w.moder15().output()
    });
```

Clocks and Timers

The Embedded Discovey Book

What is Clock ?

- **Computers use an internal clock to synchronize all of their calculations.**
- **The clock ensures that the various circuits inside a computer work together at the same time.**

What is Clock ?

- **Clock speed is measured by how many ticks per second the clock makes.**
- **The unit of measurement called a hertz (Hz).**

For loop isn't good for creating delays

```
#[inline(never)]
fn delay(tim6: &tim6::RegisterBlock, ms: u16) {
    for _ in 0..1_000 {}
}
```

No Operation

There is a way to prevent LLVM from optimizing the for loop delay: add a *volatile* assembly instruction. Any instruction will do but NOP (No OPeration) is a particular good choice in this case because it has no side effect.

Your for loop delay would become:

```
#[inline(never)]
fn delay(_tim6: &tim6::RegisterBlock, ms: u16) {
    const K: u16 = 3;
    for _ in 0..(K * ms) {
        aux9::nop()
    }
}
```

Hardware Timer

The basic function of a (hardware) timer is ... to keep precise track of time. A timer is yet another peripheral that's available to the microcontroller; thus it can be controlled using registers.

TIM6

We'll be using one of the *basic* timers: TIM6. This is one of the simplest timers available in our microcontroller

Registers of TIM6 Peripheral

The registers we'll be using in this section are:

- **SR,** the status register.

Registers of TIM6 Peripheral

The registers we'll be using in this section are:

- **SR,** the status register.
- **EGR,** the event generation register.

Registers of TIM6 Peripheral

The registers we'll be using in this section are:

- **SR,** the status register.
- **EGR,** the event generation register.
- **CNT,** the counter register.

Registers of TIM6 Peripheral

The registers we'll be using in this section are:

- **SR,** the status register.
- **EGR,** the event generation register.
- **CNT,** the counter register.
- **PSC,** the prescaler register.

One Pulse mode

Step 1:

Set the timer trough auto reload register **ARR**

One Pulse mode

Step 2:

We have to enable the counter register

(CR1.CEN = 1)

One Pulse mode

Step 3:

We have to reset the value of count register **CNT** to zero.

On each tick it’s value get incremented.

One Pulse mode

Step 4:

Once the CNT register has reached the value of the ARR register, the counter will be disabled by hardware

(CR1.CEN = 0)

and an *update event* will be raised

(SR.UIF = 1).

CNT register increase at a Frequency

CNT register increase at a frequency of **apb1 / (PSC +1)** times per second.

Also note : **1 Khz = 1 ms (period)**

Configure the prescaler to have the counter operate at 1 KHz

Where:

APB1_CLOCK = 8 MHz

PSC = ?

8_000_000/ (PSC + 1) = ?

Formula: **apb1 / (PSC +1)**

Configure the prescaler to have the counter operate at 1 KHz

Where:

Formula: **apb1 / (PSC +1)**

APB1_CLOCK = 8 MHz

PSC = ?

8_000_000/ (7_999 + 1) = 1_000 Hz ***OR*** 1kHz

The counter (CNT) will increase on every millisecond

Chapter 10

Serial Communication

Serial Communication

What's the serial communication?

- It's a communication mechanism takes place using the **asynchronous protocol**
- In this communication 2 devices exchange data serially, 1 bit at a time
- These 2 devices exchange data using **2 shared data lines** and **1 common ground**

Serial Communication

- The **protocol is asynchronous** because **none of the shared lines carries clock**
- But both (transmitter and receiver) **have to agree prior to the communication** that how fast data will be sent/received
- This protocol allows **duplex communication** which means both can send data simultaneously.

Serial Communication

Why we need this protocol? What's the use-case?

- We are using this protocol to exchange data between MCU and host computer (i.e. our laptops)
- Unlike ITM protocol, this protocol let us send data from laptop to MCU

Serial Communication

Now a question raise. How fast we can send data using this protocol?

- This protocol works with **frames**
- Each frame has **1 start** bit, **1-2 stop** bits and **5-9 data** bits
- Speed of protocol is known as **baud rate** and it's quoted in baud per seconds (**bps**)
- Few baud rate **examples** are : 9600, 19200, 38400, 57600 and 115200 bps

Serial Communication

- To actually answering the question
- With a common configuration of 1-start bit, 8-data bits, 1-stop bit and a **baud rate of 115200 bps**
- Theoretically, we can send 11,520 frames per second
- But practically, data rates would probably be lower because of processing time on the slower side of communication (the MCU)

Serial Communication

Now the problem is our laptops don’t support serial communication protocol!

- It means we can’t connect our laptop directly to the MCU
- Here is the s**erial module come in picture**
- Our module will sit between the 2 devices and expose a serial interface to the MCU and a USB interface to the laptop
- This is how both MCU and laptop will see each other as a serial device

Linux Tooling

Linux Tooling

It’s time to configure our module to work with our board and laptop

- First step is to connect the module to laptop using a mini-B USB cable
- Run this command : `$ dmesg | grep -i tty`

```
[ 3063.967949] usb 3-4: FTDI USB Serial Device converter now attached to ttyUSB0
```

- **ttyUSB0** is a file created and associated with our module

```
crw-rw-rw-+ 1 root plugdev 188, 0 Feb  3 17:52 /dev/ttyUSB0
```

Linux Tooling

- If you are following along then you can test your configuration by sending a message to the module by running the command below
- $ echo 'Hello, world!' > /dev/ttyUSB0
- Now if all okay then you should see an led blink on the module as you enter this command (LED blink once)

Minicom

Minicom

Interacting with serial module using **echo** command is not a good approach. For this purpose we will use **minicom**

- We have to configure minicom before we can use it
- For configuring minicom we will create a file named “.minirc.dfl” in home directory
- After successful creation we will copy the content below in the file

```
pu baudrate 115200
pu bits 8
pu parity N
pu stopbits 1
pu rtscts No
pu xonxoff No
```

Minicom

NOTE Make sure this file ends in a newline! Otherwise, `minicom` will fail to read it.

Let's read the minirc.dfl

- `pu baudrate 115200`. Sets baud rate to 115200 bps.
- `pu bits 8`. 8 bits per frame.
- `pu parity N`. No parity check.
- `pu stopbits 1`. 1 stop bit.
- `pu rtscts No`. No hardware control flow.
- `pu xonxoff No`. No software control flow.

Now we configured the file, let's launch the minicom

Minicom

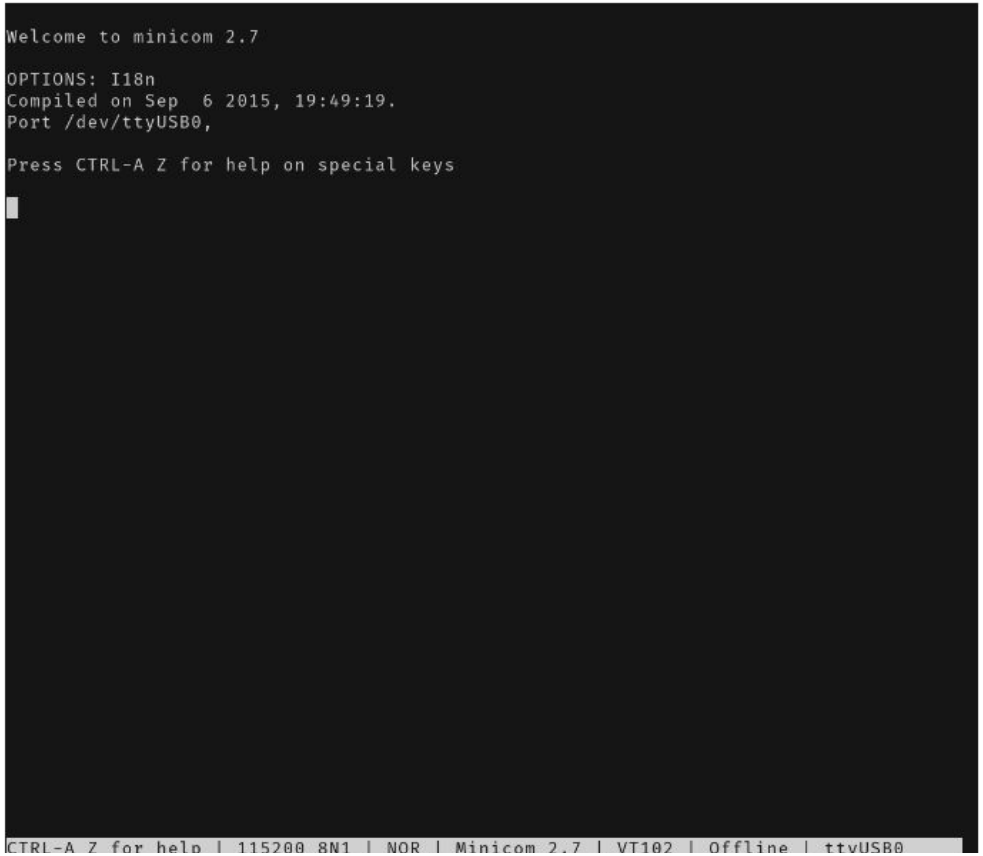
For launching minicom we will run this command :

```
$ minicom -D /dev/ttyUSB0 -b 115200
```

This command will tell minicom to open serial device at `/dev/ttyUSB0`

Also it tells to set baud rate to 115200 and pop up a text-based user interface (TUI)

Minicom



- Now we can send data using keyboard, go ahead and type something
- Now you'll observe, as you type in an LED on module will blink on each keystroke from keyboard

Minicom Commands

Minicom exposes commands via keyboard shortcuts. Few of them are:

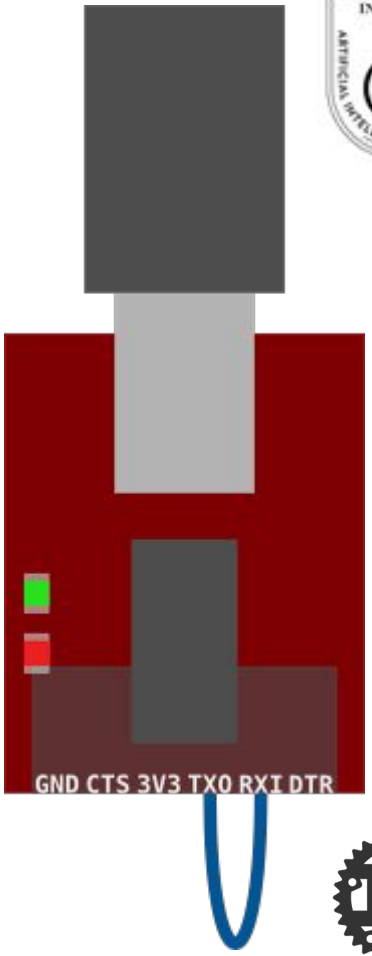
- `Ctrl+A` + `Z`. Minicom Command Summary
- `Ctrl+A` + `C`. Clear the screen
- `Ctrl+A` + `X`. Exit and reset
- `Ctrl+A` + `Q`. Quit with no reset

Loopbacks

Loopbacks

We already tested sending data to the module.
Now we try receiving data from serial module.

- For that first we have to connect **Tx** and **Rx** pins of the module using a jumper wire
- Once connected launch the minicom using the previous command
- Now if you hit any key you'll see that instead 1, now 2 leds are blinking
- Also now you can see keystrokes on your screen



Summary

Chapter 11

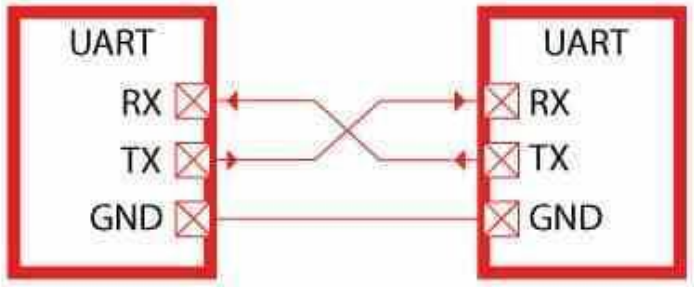
USART

USART

- Another peripheral in our microcontroller
- USART => Universal Synchronous/Asynchronous Receive Transmit
- This peripheral can work with many protocols (e.g. serial communication protocol, etc)
- In last chapter we demonstrated the interaction between debugging host/laptop and serial module
- In this chapter we will use serial communication between MCU and laptop (With the help of USART peripheral and our module)

USART Connections

- Working for this communication we have to make some connections
- We discussed in the last chapter that this protocol (i.e. serial communication protocol) involves 2 data lines (i.e. **Tx** and **Rx**)
- **Tx** => Transmitter , **Rx** => Receiver
- Transmitter and receiver are relative terms

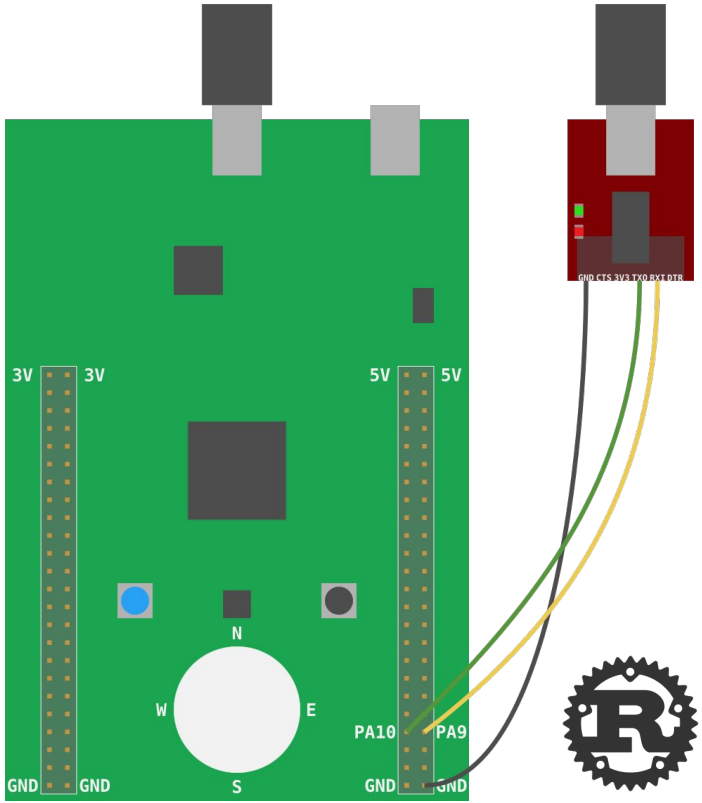


USART Connections

- We'll be using **PA9** as MCU's **Tx** line (or data output pin)
- And **PA10** as MCU's **Rx** line (or data input pin)
- However we can use a different pair of Tx and Rx pin from MCU for communication, for the possibilities we'll refer to the manual
- Manual tells us we can also use **PA2**, **PA3** and **PA14**,**PA15** as **Tx** line and **Rx** line respectively

USART Connections

- Our serial module also has **Tx** and **Rx** pin
- For making this communication possible we have to **cross these two** devices
- We'll **connect MCU's Tx pin to module's Rx pin**
- And **MCU's Rx pin to module's Tx pin**



USART Connections Steps

Recommended sequence of steps for connections:

- Close OpenOCD and itmdump
- Disconnect the USB cables from the F3 and the serial module.
- Connect one of F3 GND pins to the GND pin of the serial module using a female to male (F/M) wire. Preferably, a black one.
- Connect the PA9 pin on the back of the F3 to the RXI pin of the serial module using a F/M wire.

USART Connections Steps

- Connect the PA10 pin on the back of the F3 to the TXO pin of [illegible] serial module using a F/M wire.
- Now connect the USB cable to the F3.
- Finally connect the USB cable to the Serial module.
- Re-launch OpenOCD and itmdump

Everything is done now let's try sending data

Sending A Single Byte

Sending A Single Byte

After successful connections our first task is to send a single test byte from MCU to debugging host/laptop using **serial channel**

- As we know, for this process we are using **USART peripheral** and for convenience it is **initialized already** in the library
- Now we need to focus the core logic and the **registers** that are responsible for this communication

Sending A Single Byte

Here is starter code

```
#![deny(unsafe_code)]
#![no_main]
#![no_std]

#[allow(unused_imports)]
use aux11::{entry, iprint, iprintln};

#[entry]
fn main() -> ! {
    let (usart1, mono_timer, itm) = aux11::init();

    // Send a single character
    usart1.tdr.write(|w| w.tdr().bits(u16::from(b'X')));

    loop {}
}
```

Sending A Single Byte

- This program writes to **tdr** register
- Which will cause USART peripheral to send a byte to serial interface
- Upon successfully running this program you'll see on the laptop's minicom console a **letter X** will appear

Sending A String

Sending A String

This time we'll be sending a complete string rather than a byte

```
#![deny(unsafe_code)]
#![no_main]
#![no_std]

#[allow(unused_imports)]
use aux11::{entry, iprint, iprintln};

#[entry]
fn main() -> ! {
    let (usart1, mono_timer, itm) = aux11::init();

    // Send a string
    for byte in b"The quick brown fox jumps over the lazy dog.".iter() {
        usart1.tdr.write(|w| w.tdr().bits(u16::from(*byte)));
    }

    loop {}
}
```

Sending A String

Upon running this program in **debug mode** uninterrupted

```
Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on Aug 13 2017, 15:25:34.
Port /dev/ttyUSB0, 11:12:40

Press CTRL-A Z for help on special keys

Thequc bon o jms ve helzydg.
```

Sending A String

Upon running this program in **release mode** uninterrupted

```
Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on Aug 13 2017, 15:25:34.
Port /dev/ttyUSB0, 11:12:40

Press CTRL-A Z for help on special keys

T
```

Buffer Overrun

Buffer Overrun

What’s the problem?

- Actually time taken by bytes transmission is more than the time processor takes for execution of the program
- Let's do some calculations
- With a common configuration of **1-start** bit, **8-data** bits and **1-stop** bit and a **baud rate of 115200** bps
- For calculating baud rate in frames/S ;
  115200/10-bits => **11520 frames/S** (11.52K frames/S)

Buffer Overrun

If we have 23,040 bytes to send then time taken to send them is:

- 23,040 bytes / (11,520 frames/S) => **2S**
- We have 45 bytes so ; 45 bytes/(11,520 frames/S) => 0.00390625S or (3,906 uS)
- So ideally 3,906uS is the time required to send our string
- Now let's see how much time our program take to execute

Buffer Overrun

We will change our starter code to

```rust
#![deny(unsafe_code)]
#![no_main]
#![no_std]

#[allow(unused_imports)]
use aux11::{entry, iprint, iprintln};

#[entry]
fn main() -> ! {
    let (usart1, mono_timer, mut itm) = aux11::init();

    let instant = mono_timer.now();
    // Send a string
    for byte in b"The quick brown fox jumps over the lazy dog.".iter() {
        usart1.tdr.write(|w| w.tdr().bits(u16::from(*byte)));
    }
    let elapsed = instant.elapsed(); // in ticks

    iprintln!(
        &mut itm.stim[0],
        "`for` loop took {} ticks ({} us)",
        elapsed,
        elapsed as f32 / mono_timer.frequency().0 as f32 * 1e6
    );

    loop {}
}
```

Buffer Overrun

Above program will calculate the time taken by processor to execute it

In debug mode the output on itm console is:

```
rajabraza@EliteBook-Folio-9470m:/tmp$ touch itm.txt && itmdump -F -f itm.txt
`for` loop took 18909 ticks (2363.625 us)
```

In release mode the output on itm console is:

```
rajabraza@EliteBook-Folio-9470m:/tmp$ touch itm.txt && itmdump -F -f itm.txt
`for` loop took 91 ticks (11.375 us)
```

Buffer Overrun Solution

Buffer Overrun Solution

Solution to this problem is

- We will be using register ISR
- ISR is a status register in **USART peripheral** like TDR register
- ISR has a **flag TXE** that indicates whether it is safe to write data on TDR register or not
- We will use this flag to actually **halt our program** unless it is safe to write
- We will modify the existing code like

Buffer Overrun Solution

```
#![deny(unsafe_code)]
#![no_main]
#![no_std]

#[allow(unused_imports)]
use aux11::{entry, iprint, iprintln};

#[entry]
fn main() -> ! {
    let (usart1, mono_timer, mut itm) = aux11::init();

    let instant = mono_timer.now();
    // Send a string
    for byte in b"The quick brown fox jumps over the lazy dog.".iter() {
        // wait until it's safe to write to TDR
        while usart1.isr.read().txe().bit_is_clear() {} // <- NEW!

        usart1.tdr.write(|w| w.tdr().bits(u16::from(*byte)));
    }
    let elapsed = instant.elapsed(); // in ticks

    iprintln!(
        &mut itm.stim[0],
        "`for` loop took {} ticks ({} us)",
        elapsed,
        elapsed as f32 / mono_timer.frequency().0 as f32 * 1e6
    );

    loop {}
}
```

Buffer Overrun Solution

Now after this modification, when we ran program in debug mode

```
Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on Aug 13 2017, 15:25:34.
Port /dev/ttyUSB0, 15:50:42

Press CTRL-A Z for help on special keys

The quick brown fox jumps over the lazy dog.
```

And on itm console

```
rajabraza@EliteBook-Folio-9470m:/tmp$ touch itm.txt && itmdump -F -f itm.txt
`for` loop took 30384 ticks (3798 us)
```

Buffer Overrun Solution

Now after this modification, when we ran program in release mode

```
Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on Aug 13 2017, 15:25:34.
Port /dev/ttyUSB0, 15:50:42

Press CTRL-A Z for help on special keys

The quick brown fox jumps over the lazy dog.
```

And on itm console

```
rajabraza@EliteBook-Folio-9470m:/tmp$ touch itm.txt && itmdump -F -f itm.txt
`for` loop took 29690 ticks (3711.25 us)
```

Receive A Single Byte

Receive A Single Byte

So far we have sent data from the MCU to laptop. Now let's try receiving data from laptop.

- This time we will be using RDR
- RDR register receives data from the senders and filled up the Rx line
- Once the Rx line is filled by the other side of the channel we will retrieve data from the board
- Here is again ISR register comes and tell us whenever some new data sent by any connected device

Receive A Single Byte

- ISR register has another flag RXNE, which keeps a check on and whenever some new data comes in it intimates

```
#![deny(unsafe_code)]
#![no_main]
#![no_std]

#[allow(unused_imports)]
use aux11::{entry, iprint, iprintln};

#[entry]
fn main() -> ! {
    let (usart1, mono_timer, itm) = aux11::init();

    loop {
        // Wait until there's data available
        while usart1.isr.read().rxne().bit_is_clear() {}

        // Retrieve the data
        let _byte = usart1.rdr.read().rdr().bits() as u8;

        aux11::bkpt();
    }
}
```

Summary

Chapter 14

I2C

I2C

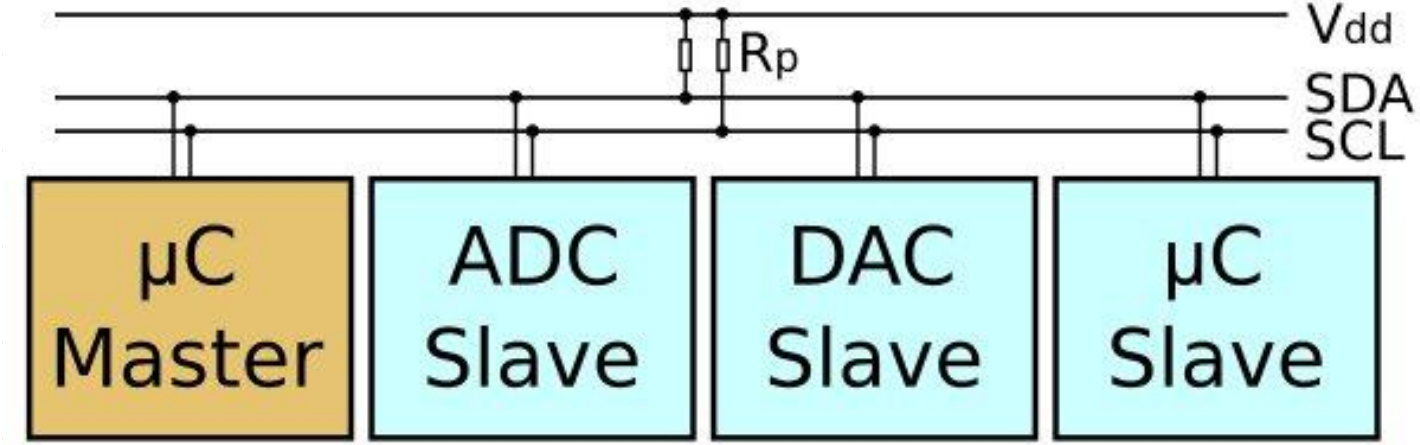
- Previously we were working with serial communication protocol which is a widely used protocol because of its very simple
- This simplicity makes it easy to implement on top of other protocols like bluetooth and USB
- However this simplicity brings some downsides
- data exchanges, like reading a digital sensor, would require the sensor vendor to come up with another protocol on top of it
- Luckily for us, there are plenty of other communication protocols in the embedded space.

I2C

- Our F3 board has **3 motion sensors** (i.e. accelerometer, magnetometer and gyroscope)
- The accelerometer and magnetometer are **packaged in a single component** and can be accessed using an I2C bus.
- I2C stands for Inter-Integrated Circuit and it is a **synchronous serial communication protocol**
- This protocol uses two lines to exchange data
    - SDA : A data line
    - SCL : A clock line

I2C

- Protocol is synchronous because a clock line is used to synchronize the communication



- As you can see this protocol uses **master slave** model

I2C

- Master is the device that starts and derive the communication with the slave
- Several devices, both masters and slaves can be connected to the same bus at the same time
- Master can communicate with specific slave by broadcasting it's address (address can be 7 or 10 bit long)
- Once master started communication, no other device can make use of bus until master stops communication

I2C

- SCL or clock line determines how fast data can be exchanged
- Usually operates at 100 KHz (standard mode) or 400 KHz (fast mode)

General Protocol

General Protocol

- Using I2C protocol we can communicate with multiple devices unlike serial communication protocol where we cannot.
- Now let see how this communication takes place
- Two way communication is possible in this protocol
- Master -> Slave (Where master writes to slave)
- Master <- Slave (Where master reads from slave)

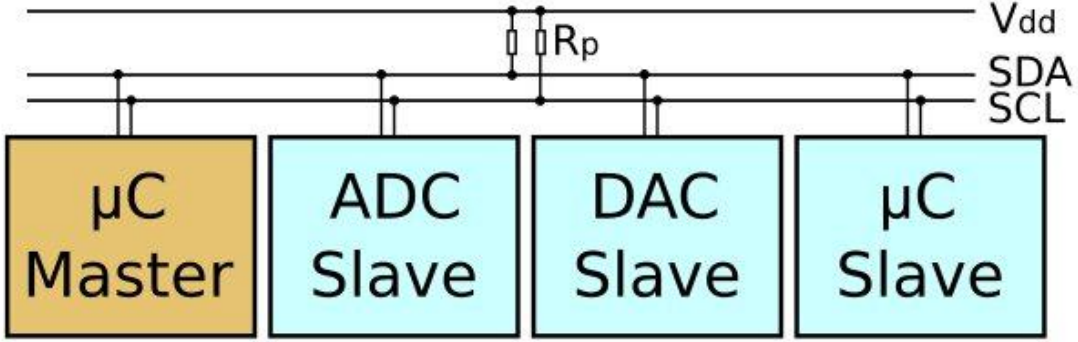
Master -> Slave

When master wants to send data to slave

1. Master: Broadcast START
2. M: Broadcast slave address(7 bits)

   + the R/W (8th) bit set to **WRITE**
3. Slave: Responds **ACK**

   (ACKnowledgement)
4. M: Send one byte
5. S: Responds ACK
6. Repeat steps 4 and 5 zero or more times
7. M: Broadcast STOP OR (broadcast RESTART and go back to (2))



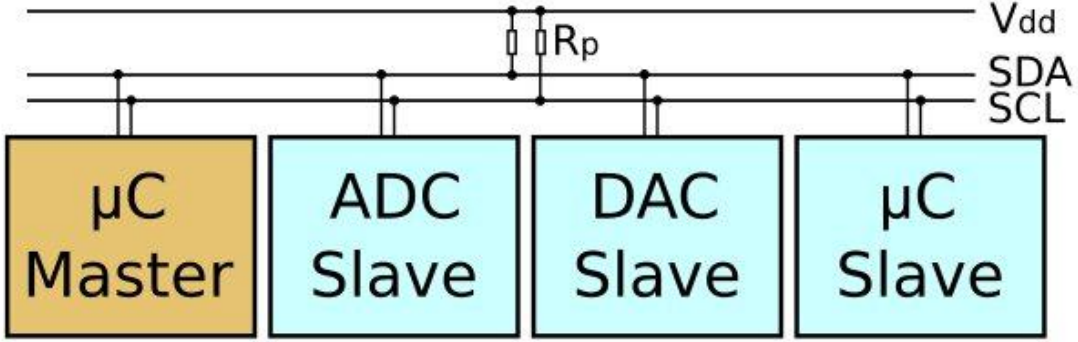
Master <- Slave

When master wants to read data from slave

1. M: Broadcast START
2. M: Broadcast slave address(7 bits)

   + the R/W (8th) bit set to **READ**
3. Slave: Responds ACK

   (ACKnowledgement)
4. S: Send one byte
5. M: Responds ACK
6. Repeat steps 4 and 5 zero or more times
7. M: Broadcast STOP OR (broadcast RESTART and go back to (2))



LSM303DLHC

LSM303DLHC

- Two sensors, the magnetometer and the accelerometer, are packaged in this chip
- These sensors can be accessed via an I2C bus
- Each sensor behaves like an I2C slave and has a different address
- Each sensor has its own memory where it stores the results of sensing its environment
- Our interaction with these sensors will mainly involve reading their memory

LSM303DLHC

- Memory is modeled as byte addressable registers of these sensors
- Sensors can be configured by writing to their registers
- This way, these sensors are similar to the peripherals inside the uC
- Difference is that their registers are not mapped into the uCs' memory. Instead, their registers have to be accessed via the I2C bus

LSM303DLHC

- After all this theory let's try reading some data from these sensors
- Following are the registers and fields will be used in reading/writing from/to sensors:
  - CR2: SADD1, RD_WRN, NBYTES, START, AUTOEND
  - ISR: TXIS, RXNE, TC
  - TXDR: TXDATA
  - RXDR: RXDATA

Starter code can be viewed in the source repository

Summary

Chapter # 15

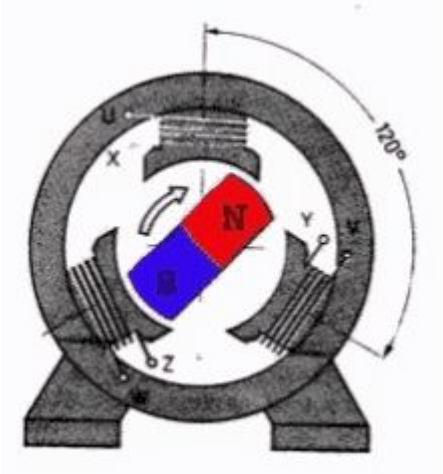
LED COMPASS

Compass

A compass is an instrument used for navigation and orientation that shows direction relative to the geographic cardinal directions. Usually, a diagram called a compass rose shows the directions north, south, east, and west on the compass face as abbreviated initials.

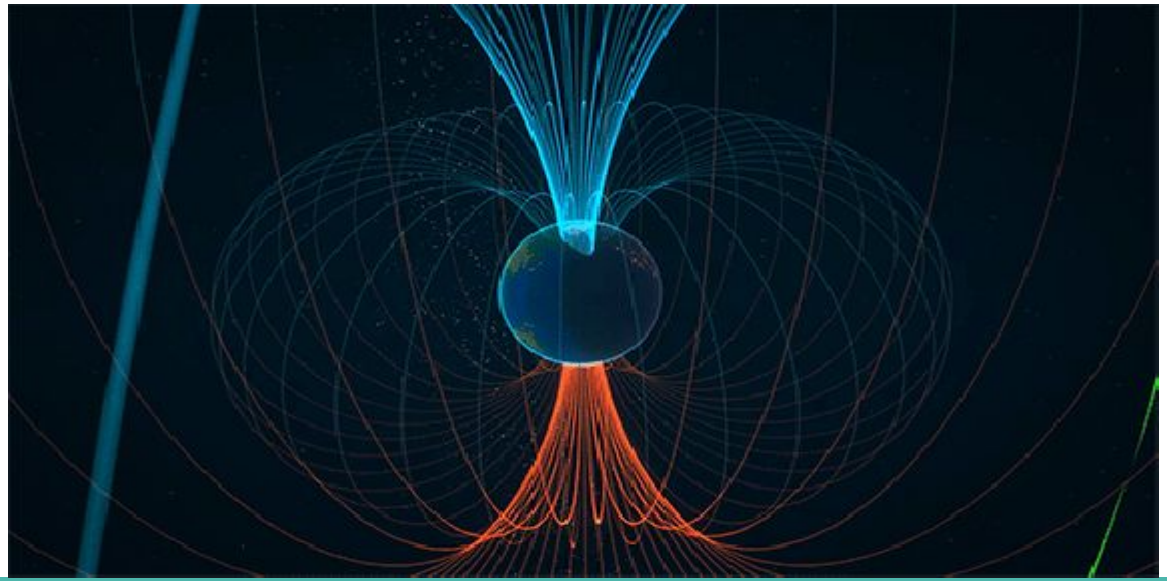
Working of a Compass

Compasses work so effortlessly because their design allows the magnet to respond freely to Earth's magnetic field. Earth itself is like a giant magnet that creates its own magnetic field. The north end of a **compass** is drawn to align with Earth's magnetic North Pole.



Earth’s Magnetic Field

Earth's magnetic field, also known as the **geomagnetic field**, is the magnetic field that extends from the Earth's interior out into space, where it interacts with the solar wind, a stream of charged particles emanating from the Sun.

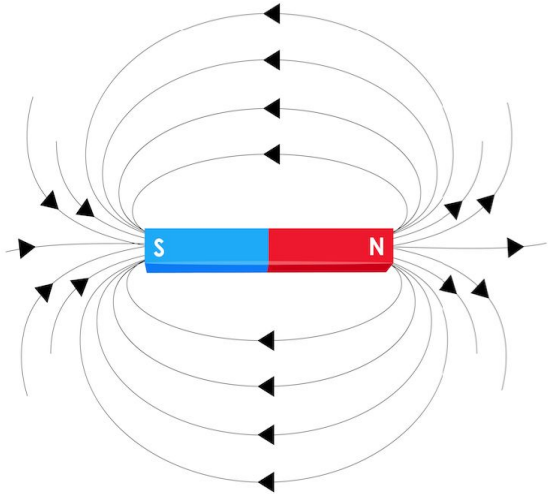


LED Compass

We'll implement a compass using the LEDs on the F3. Like proper compasses, our LED compass must point north somehow. It will do that by turning on one of its eight LEDs; the on LED should point towards north.

Magnetic Field

A **magnetic field** is a vector **field** that describes the **magnetic** influence of electric charges in relative motion and magnetized materials. The effects of **magnetic fields** are commonly seen in permanent **magnets**, which pull on **magnetic** materials (such as iron) and attract or repel other **magnets.**



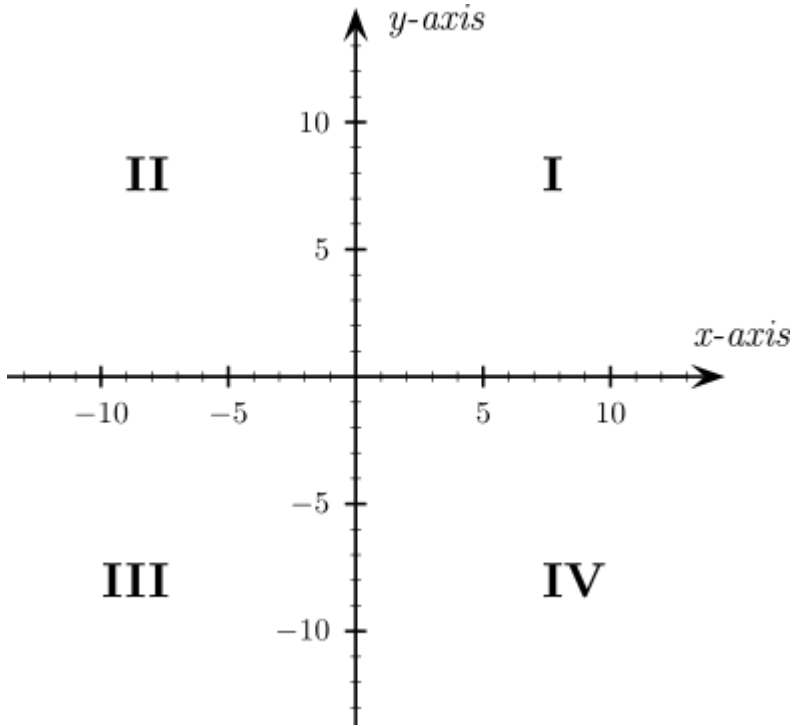
LSM303DLHC Package

Magnetic fields have both a magnitude, measured in Gauss or Teslas, and a *direction*. The magnetometer on the F3 measures both the magnitude and the direction of an external magnetic field but it reports back the *decomposition* of said field along *its axes*. As shown in console.

```
$ # itmdump terminal
(..)
I16x3 { x: 45, y: 194, z: -3 }
I16x3 { x: 46, y: 195, z: -8 }
I16x3 { x: 47, y: 197, z: -2 }
```

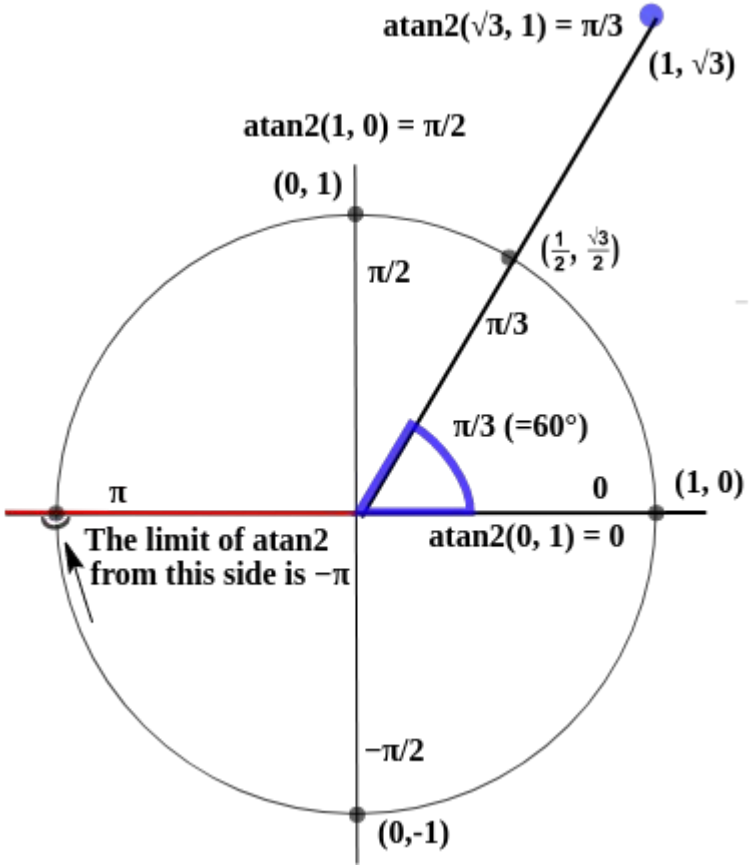
Take 1

If we only looked at the signs of the X and Y components we could determine to which quadrant the magnetic field belongs to.



Take 2

We'll use math to get the precise angle that the magnetic field forms with the X and Y axes of the magnetometer. We'll use the atan2 function. This function returns an angle in the -PI to PI range.



Magnitude

- The number that the magnetic_field function reports are unit-less.
- We convert those values to Gauss.
- *Magnetic gain setting* that has different values according to the values of the GN bits. By default, those GN bits are set to 001.
- We need to do is divide the X, Y and Z values that the sensor outputs by its corresponding *gain*.
- The magnitude of the Earth's magnetic field is in the range of 250 mG to 650 mG (the magnitude varies depending on your geographical location)

Calibration

- The direction of the Earth's magnetic field with respect to the magnetometer should change but its magnitude should not
- The magnetometer indicates that the magnitude of the magnetic field changes as the board rotates.
- The calibration involves quite a bit of math (matrices).
- Let's record the readings of the magnetometer while we slowly rotate the board in different directions.

Conclusion

- Reading values from magnetometer.
- Design LED Compass.
- Calculate its Magnitude.
- Calibrate the sensor to get accurate Readings.

Chapter # 16

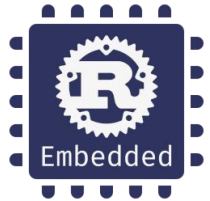
Punch-O-Meter

Punch-o-Meter

This meter calculates one's **punch** strength by throwing an air **punch** and shows the reading in newton/gravity.

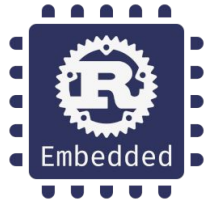
Working of a punch-o-meter

It measures the power of your jabs. Actually the maximum acceleration that you can reach because acceleration is what accelerometers measure. Strength and acceleration are proportional though so it's a good approximation.



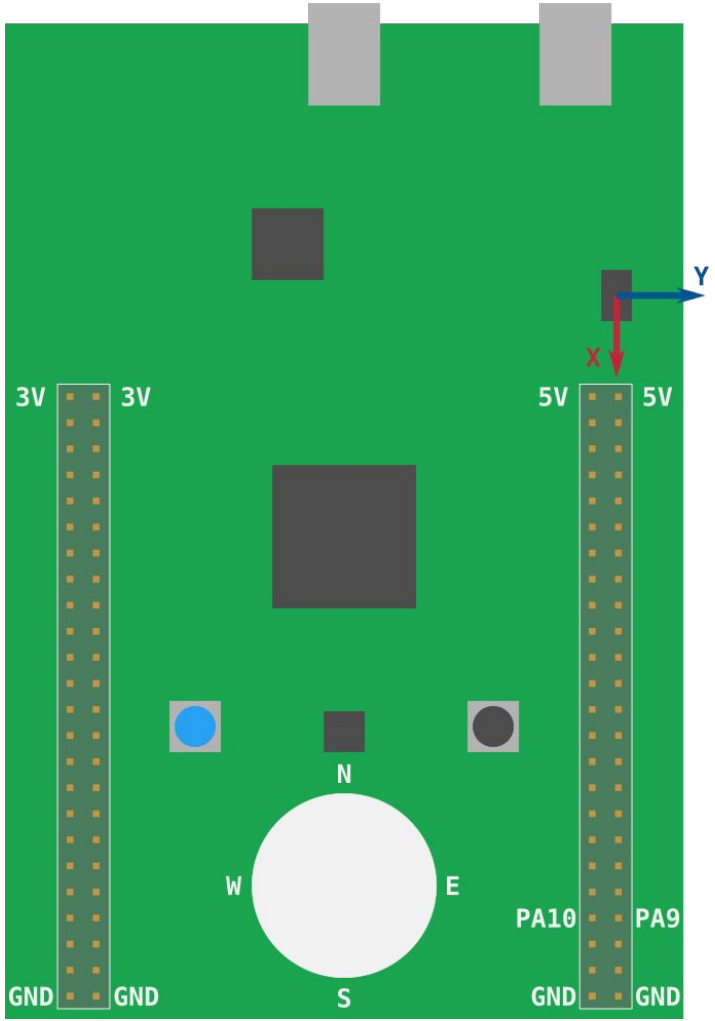
What's inside punch-o-meter?

Inside the ball, there is a force detecting sensor named accelerometer. When a person hits the ball acceleration produces in it. The accelerometer senses this change and the meter will show reading accordingly.



LSM303DLHC Package

The accelerometer is also built inside the LSM303DLHC package. And just like the magnetometer, it can also be accessed using the I2C bus. It also has the same coordinate system as the magnetometer



Gravity/Gravitational Force

Gravity is a force of attraction that exists between any two masses, any two bodies, any two particles. **Gravity** is like the Earth pulling on you and keeping you on the ground. Every object in the universe that has mass exerts a gravitational pull, or force, on every other mass. The size of the pull depends on the masses of the objects.

Gravity is up?

The starter code prints the X, Y and Z components of the acceleration measured by the accelerometer. The values have already been "scaled" and have units of gs. Where 1 g is equal to the acceleration of the gravity, about 9.8 meters per second squared.

```
#[entry]
fn main() -> ! {

 let (mut lsm303dlhc, mut delay, ,mut itm)=aux16::init();

    lsm303dlhc.set_accel_sensitivity(Sensitivity::G12).unwrap();

loop {
        const SENSITIVITY: f32 = 12. / (1 << 14) as f32;

        let I16x3 { x, y, z } = lsm303dlhc.accel().unwrap();

        let x = f32::from(x) * SENSITIVITY;
        let y = f32::from(y) * SENSITIVITY;
        let z = f32::from(z) * SENSITIVITY;

        iprintln!(&mut itm.stim[0], "{:?}", (x, y, z));

        delay.delay_ms(1_000_u16);
    }
}
```

Why this reading?

Because the board is not moving yet its acceleration is non-zero so the gravity only exerts in z-axis . The acceleration of gravity is 1 g.

```
$ # itmdump console
(..)
(0.0, 0.0, 1.078125)
(0.0, 0.0, 1.078125)
(0.0, 0.0, 1.171875)
(0.0, 0.0, 1.03125)
(0.0, 0.0, 1.078125)
```

What we are going to do?

We'll measure the acceleration only in the X axis to keep things simple while the board remains horizontal.

Here's what the punch-o-meter must do:

- When a significant acceleration in X-axis is detected, the app will start a new measurement.
- During that measurement interval, the app will keep track of the maximum acceleration observed.
- The app will report the maximum acceleration observed at the end of interval.

Thanks

PIAIC Team