Discovery

> Discover the world of microcontrollers through Rust!

This book is an introductory course on microcontroller-based embedded systems that uses Rust as the teaching language rather than the usual C/C++.

Scope

The following topics will be covered (eventually, I hope):

- How to write, build, flash and debug an "embedded" (Rust) program.
- Functionality ("peripherals") commonly found in microcontrollers: Digital input and output, Pulse Width Modulation (PWM), Analog to Digital Converters (ADC), common communication protocols like Serial, I2C and SPI, etc.
- Multitasking concepts: cooperative vs preemptive multitasking, interrupts, schedulers, etc.
- Control systems concepts: sensors, calibration, digital filters, actuators, open loop control, closed loop control, etc.

Approach

- Beginner friendly. No previous experience with microcontrollers or embedded systems is required.
- Hands on. Plenty of exercises to put the theory into practice. *You* will be doing most of the work here.
- Tool centered. We'll make plenty use of tooling to ease development. "Real" debugging, with GDB, and logging will be introduced early on. Using LEDs as a debugging mechanism has no place here.

Non-goals

What's out of scope for this book:

- Teaching Rust. There's plenty of material on that topic already. We'll focus on microcontrollers and embedded systems.
- Being a comprehensive text about electric circuit theory or electronics. We'll just cover the minimum required to understand how some devices work.
- Covering Rustic, low level details. We won't be talking about linker scripts, the boot process or how to glue those two into a minimally working Rust program.

Also I don't intend to port this material to other development boards; this book will make exclusive use of the STM32F3DISCOVERY development board.

Reporting problems

The source of this book is in this repository. If you encounter any typo or problem with the code report it on the issue tracker.

Other embedded Rust resources

This Discovery book is just one of several embedded Rust resources provided by the Embedded Working Group. The full selection can be found at The Embedded Rust Bookshelf. This includes the list of Frequently Asked Questions.

Sponsored by

integer 32

Many thanks to integer 32 for sponsoring me to work on this book! Please give them lots of work (they do Rust consulting!) so they'll have no choice but to hire more Rustaceans <3.

Background

What's a microcontroller?

A microcontroller is a *system* on a chip. Whereas your laptop is made up of several discrete components: a processor, RAM sticks, a hard drive, an ethernet port, etc.; a microcontroller has all those components built into a single "chip" or package. This makes it possible to build systems with minimal part count.

What can you do with a microcontroller?

Lots of things! Microcontrollers are the central part of systems known as *embedded* systems. These systems are everywhere but you don't usually notice them. These systems control the brakes of your car, wash your clothes, print your documents, keep you warm, keep you cool, optimize the fuel consumption of your car, etc.

The main trait of these systems is that they operate without user intervention even if they expose a user interface like a washing machine does; most of their operation is done on their own.

The other common trait of these systems is that they *control* a process. And for that these systems usually have one or more sensors and one or more actuators. For example, an HVAC system has several sensors, thermometers and humidity sensors spread across some area, and several actuators as well, heating elements and fans connected to ducts.

When should I use a microcontroller?

All these application I've mentioned, you can probably implement with a Raspberry Pi, a computer that runs Linux. Why should I bother with a microcontroller that operates without an OS? Sounds like it would be harder to develop a program.

The main reason is cost. A microcontroller is much cheaper than a general purpose computer. Not only the microcontroller is cheaper; it also requires many fewer external electrical components to operate. This makes Printed Circuit Boards (PCB) smaller and cheaper to design and manufacture.

The other big reason is power consumption. A microcontroller consumes orders of magnitude less power than a full blown processor. If your application will run on batteries that makes a huge difference.

And last but not least: (hard) *real time* constraints. Some processes require their controllers to respond to some events within some time interval (e.g. a quadcopter/drone hit by a wind gust). If this *deadline* is not met, the process could end in catastrophic failure (e.g. the drone crashes to the ground). A general purpose computer running a general purpose OS has many services running in the background. This makes it hard to guarantee execution of a program within tight time constraints.

When should I *not* use a microcontroller?

Where heavy computations are involved. To keep their power consumption low, microcontrollers have very limited computational resources available to them. For example, some microcontrollers don't even have hardware support for floating point operations. On those devices, performing a simple addition of single precision numbers can take hundreds of CPU cycles.

Why use Rust and not C?

Hopefully, I don't need to convince you here as you are probably familiar with the language differences between Rust and C. One point I do want to bring up is package management. C lacks an official, widely accepted package management solution whereas Rust has Cargo. This makes development *much* easier. And, IMO, easy package management encourages code reuse because libraries can be easily integrated into an application which is also a good thing as libraries get more "battle testing".

Why should I not use Rust?

Or why should I prefer C over Rust?

The C ecosystem is way more mature. Off the shelf solution for several problems already exist. If you need to control a time sensitive process, you can grab one of the existing commercial Real Time Operating Systems (RTOS) out there and solve your problem. There are no commercial, production-grade RTOSes in Rust yet so you would have to either create one yourself or try one of the ones that are in development.

Hardware/knowledge requirements

The only knowledge requirement to read this book is to know *some* Rust. It's hard for me to quantify *some* but at least I can tell you that you don't need to fully grok generics but you do need to know how to *use* closures. You also need to be familiar with the idioms of the 2018 edition, in particular with the fact that `extern crate` is not necessary in the 2018 edition.

Also, to follow this material you'll need the following hardware:

(Some components are optional but recommended)

- A STM32F3DISCOVERY board.

(You can purchase this board from "big" electronics suppliers or from e-commerce sites)

STM32F3DISCOVERY

- OPTIONAL. A **3.3V** USB <-> Serial module. This particular model will be used throughout this material but you can use any other model as long as it operates at

3.3V.

(The (Chinese) CH340G module, which you can buy e-commerce sites, works too and it's probably cheaper for you to get)

A 3.3v USB <-> Serial module

- OPTIONAL. A HC-05 Bluetooth module (with headers!). A HC-06 would work too.

(As with other Chinese parts, you pretty much can only find these on e-commerce sites. (US) Electronics suppliers don't usually stock these for some reason)

The HC-05 Bluetooth module

- Two mini-B USB cables. One is required to make the STM32F3DISCOVERY board work. The other is only required if you have the Serial <-> USB module. Make sure that the cables both support data transfer as some cables only support charging devices.

mini-B USB cable

NOTE These are **not** the USB cables that ship with pretty much every Android phone; those are *micro* USB cables. Make sure you have the right thing!

- MOSTLY OPTIONAL. 5 female to female, 4 male to female and 1 Male to Male *jumper* (AKA Dupont) wires. You'll *very likely* need one female to female to get ITM working. The other wires are only needed if you'll be using the USB <-> Serial and Bluetooth modules.

(You can get these from electronics suppliers or from e-commerce sites)

Jumper wires

FAQ: Wait, why do I need this specific hardware?

It makes my life and yours much easier.

The material is much, much more approachable if we don't have to worry about hardware differences. Trust me on this one.

FAQ: Can I follow this material with a different development board?

Maybe? It depends mainly on two things: your previous experience with microcontrollers and/or whether there already exists a high level crate, like the `f3`, for your development board somewhere.

With a different development board, this text would lose most if not all its beginner friendliness and "easy to follow"-ness, IMO.

If you have a different development board and you don't consider yourself a total beginner, you are better off starting with the quickstart project template.

Setting up a development environment

Dealing with microcontrollers involves several tools as we'll be dealing with an architecture different than your laptop's and we'll have to run and debug programs on a "remote" device.

Documentation

Tooling is not everything though. Without documentation it is pretty much impossible to work with microcontrollers.

We'll be referring to all these documents throughout this book:

HEADS UP All these links point to PDF files and some of them are hundreds of pages long and several MBs in size.

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

Tools

We'll use all the tools listed below. Where a minimum version is not specified, any recent version should work but we have listed the version we have tested.

- Rust 1.31 or a newer toolchain.
- `itmdump` v0.3.1
- OpenOCD >=0.8. Tested versions: v0.9.0 and v0.10.0
- `arm-none-eabi-gdb`. Version 7.12 or newer highly recommended. Tested versions: 7.10, 7.11, 7.12 and 8.1
- `cargo-binutils`. Version 0.1.4 or newer.

- `minicom` on Linux and macOS. Tested version: 2.7. Readers report that `picocom` also works but we'll use `minicom` in this text.
- `PuTTY` on Windows.

If your laptop has Bluetooth functionality and you have the Bluetooth module, you can additionally install these tools to play with the Bluetooth module. All these are optional:

- Linux, only if you don't have a Bluetooth manager application like Blueman.
  - `bluez`
  - `hcitool`
  - `rfcomm`
  - `rfkill`

macOS / OSX / Windows users only need the default bluetooth manager that ships with their OS.

Next, follow OS-agnostic installation instructions for a few of the tools:

`rustc` & Cargo

Install rustup by following the instructions at https://rustup.rs.

If you already have rustup installed double check that you are on the stable channel and your stable toolchain is up to date. `rustc -V` should return a date newer than the one shown below:

```
$ rustc -V
rustc 1.31.0 (abe02cefd 2018-12-04)
```

`itmdump`

```
$ cargo install itm --vers 0.3.1

$ itmdump -V
itmdump 0.3.1
```

`cargo-binutils`

```
$ rustup component add llvm-tools-preview

$ cargo install cargo-binutils --vers 0.1.4

$ cargo size -- -version
LLVM (http://llvm.org/):
  LLVM version 8.0.0svn
  Optimized build.
  Default target: x86_64-unknown-linux-gnu
  Host CPU: skylake
```

OS specific instructions

Now follow the instructions specific to the OS you are using:

- Linux
- Windows
- macOS

Linux

Here are the installation commands for a few Linux distributions.

REQUIRED packages

Ubuntu 18.04 or newer / Debian stretch or newer

> **NOTE** `gdb-multiarch` is the GDB command you'll use to debug your ARM Cortex-M programs

```
$ sudo apt-get install \
  gdb-multiarch \
  minicom \
  openocd
```

Ubuntu 14.04 and 16.04

> **NOTE** `arm-none-eabi-gdb` is the GDB command you'll use to debug your ARM Cortex-M programs

```
$ sudo apt-get install \
  gdb-arm-none-eabi \
  minicom \
  openocd
```

Fedora 23 or newer

> **NOTE** `arm-none-eabi-gdb` is the GDB command you'll use to debug your ARM Cortex-M programs

```
$ sudo dnf install \
  arm-none-eabi-gdb \
  minicom \
  openocd
```

Arch Linux

> **NOTE** `arm-none-eabi-gdb` is the GDB command you'll use to debug your ARM Cortex-M programs

```
$ sudo pacman -S \
  arm-none-eabi-gdb \
  minicom \
  openocd
```

Other distros

> **NOTE** `arm-none-eabi-gdb` is the GDB command you'll use to debug your ARM Cortex-M programs

For distros that don't have packages for ARM's pre-built toolchain, download the "Linux 64-bit" file and put its `bin` directory on your path. Here's one way to do it:

```
$ mkdir -p ~/local && cd ~/local
$ tar xjf /path/to/downloaded/file/gcc-arm-none-eabi-7-2017-q4-major-
linux.tar.bz2.tbz
```

Then, use your editor of choice to append to your `PATH` in the appropriate shell init file (e.g. `~/.zshrc` or `~/.bashrc`):

```
PATH=$PATH:$HOME/local/gcc-arm-none-eabi-7-2017-q4-major/bin
```

Optional packages

Ubuntu / Debian

```
$ sudo apt-get install \
  bluez \
  rfkill
```

Fedora

```
$ sudo dnf install \
  bluez \
  rfkill
```

Arch Linux

```
$ sudo pacman -S \
  bluez \
  bluez-utils \
  rfkill
```

udev rules

These rules let you use USB devices like the F3 and the Serial module without root privilege, i.e. `sudo`.

Create these two files in `/etc/udev/rules.d` with the contents shown below.

```
$ cat /etc/udev/rules.d/99-ftdi.rules
```

```
# FT232 - USB <-> Serial Converter
ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6001", MODE:="0666"
```

```
$ cat /etc/udev/rules.d/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"
```

Then reload the udev rules with:

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

If you had any board plugged to your laptop, unplug them and then plug them in again.

Now, go to the next section.

Windows

`arm-none-eabi-gdb`

ARM provides `.exe` installers for Windows. Grab one from here, and follow the instructions. Just before the installation process finishes tick/select the "Add path to environment variable" option. Then verify that the tools are in your `%PATH%`:

```
$ arm-none-eabi-gcc -v
(..)
gcc version 5.4.1 20160919 (release) (..)
```

OpenOCD

There's no official binary release of OpenOCD for Windows but there are unofficial releases available here. Grab the 0.10.x zipfile and extract it somewhere in your drive (I recommend `C:\OpenOCD` but with the drive letter that makes sense to you) then update your `%PATH%` environment variable to include the following path: `C:\OpenOCD\bin` (or the path that you used before).

Verify that OpenOCD is in yout `%PATH%` with:

```
$ openocd -v
Open On-Chip Debugger 0.10.0
(..)
```

PuTTY

Download the latest `putty.exe` from this site and place it somewhere in your `%PATH%`.

ST-LINK USB driver

You'll also need to install this USB driver or OpenOCD won't work. Follow the installer instructions and make sure you install the right (32-bit or 64-bit) version of the driver.

That's all! Go to the next section.

macOS

All the tools can be install using Homebrew:

```
$ brew cask install gcc-arm-embedded

$ brew install minicom openocd
```

If the `brew cask` command doesn't work (`Error: Unknown command: cask`), then run `brew tap Caskroom/tap` first and try again.

That's all! Go to the next section.

Verify the installation

Let's verify that all the tools were installed correctly.

Linux only

Verify permissions

Connect the F3 to your laptop using an USB cable. Be sure to connect the cable to the "USB ST-LINK" port, the USB port in the center of the edge of the board.

The F3 should now appear as a USB device (file) in `/dev/bus/usb`. Let's find out how it got enumerated:

```
$ lsusb | grep -i stm
Bus 003 Device 004: ID 0483:374b STMicroelectronics ST-LINK/V2.1
$ # ^^^        ^^^
```

In my case, the F3 got connected to the bus #3 and got enumerated as the device #4. This means the file `/dev/bus/usb/003/004` *is* the F3. Let's check its permissions:

```
$ ls -l /dev/bus/usb/003/004
crw-rw-rw- 1 root root 189, 20 Sep 13 00:00 /dev/bus/usb/003/004
```

The permissions should be `crw-rw-rw-`. If it's not ... then check your udev rules and try re-loading them with:

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

Now let's repeat the procedure for the Serial module.

Unplug the F3 and plug the Serial module. Now, figure out what's its associated file:

```
$ lsusb | grep -i ft232
Bus 003 Device 005: ID 0403:6001 Future Technology Devices International, Ltd
FT232 Serial (UART) IC
```

In my case, it's the `/dev/bus/usb/003/005`. Now, check its permissions:

```
$ ls -l /dev/bus/usb/003/005
crw-rw-r-- 1 root root 189, 21 Sep 13 00:00 /dev/bus/usb/003/005
```

As before, the permissions should be `crw-rw-rw-`.

All

First OpenOCD connection

First, connect the F3 to your laptop using an USB cable. Connect the cable to the USB port in the center of edge of the board, the one that's labeled "USB ST-LINK".

Two *red* LEDs should turn on right after connecting the USB cable to the board.

Next, run this command:

```
$ # *nix
$ openocd -f interface/stlink-v2-1.cfg -f target/stm32f3x.cfg

$ # Windows
$ # NOTE cygwin users have reported problems with the -s flag. If you run into
$ # that you can call openocd from the `C:\OpenOCD\share\scripts` directory
$ openocd -s C:\OpenOCD\share\scripts -f interface/stlink-v2-1.cfg -f
target/stm32f3x.cfg
```

NOTE Windows users: `C:\OpenOCD` is the directory where you installed OpenOCD to.

> **IMPORTANT** There is more than one hardware revision of the STM32F3DISCOVERY board. For older revisions, you'll need to change the "interface" argument to `-f interface/stlink-v2.cfg` (note: no `-1` at the end). Alternatively, older revisions can use `-f board/stm32f3discovery.cfg` instead of `-f interface/stlink-v2-1.cfg -f target/stm32f3x.cfg`.

You should see output like this:

```
Open On-Chip Debugger 0.10.0
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "hla_swd". To override
use 'transport select <transport>'.
adapter speed: 1000 kHz
adapter_nsrst_delay: 100
Info : The selected transport took over low-level target control. The results
might differ compared to plain JTAG/SWD
none separate
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : clock speed 950 kHz
Info : STLINK v2 JTAG v27 API v2 SWIM v15 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 2.915608
Info : stm32f3x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

(If you don't ... then check the general troubleshooting instructions.)

`openocd` will block the terminal. That's fine.

Also, one of the red LEDs, the one closest to the USB port, should start oscillating between red light and green light.

That's it! It works. You can now close/kill `openocd`.

Meet your hardware

Let's get familiar with the hardware we'll be working with.

STM32F3DISCOVERY (the "F3")

F3

We'll refer to this board as "F3" throughout this book.

What does this board contain?

- A STM32F303VCT6 microcontroller. This microcontroller has
  - A single core ARM Cortex-M4F processor with hardware support for single precision floating point operations and a maximum clock frequency of 72 MHz.
  - 256 KiB of "Flash" memory. (1 KiB = 10**24** bytes)
  - 48 KiB of RAM.
  - many "peripherals": timers, GPIO, I2C, SPI, USART, etc.
  - lots of "pins" that are exposed in the two lateral "headers".
  - **IMPORTANT** This microcontroller operates at (around) 3.3V.
- An accelerometer and a magnetometer (in a single package).
- A gyroscope.
- 8 user LEDs arranged in the shape of a compass
- A second microcontroller: a STM32F103CBT. This microcontroller is actually part of an on-board programmer and debugger named ST-LINK and is connected to the USB port named "USB ST-LINK".
- There's a second USB port, labeled "USB USER" that is connected to the main microcontroller, the STM32F303VCT6, and can be used in applications.

The Serial module

Serial module

We'll use this module to exchange data between the microcontroller in the F3 and your laptop. This module will be connected to your laptop using an USB cable. I won't say more at this point.

The Bluetooth module

The HC-05 Bluetooth module

This module has the exact same purpose as the serial module but it sends the data over Bluetooth instead of over USB.

LED roulette

Alright, let's start by building the following application:

I'm going to give you a high level API to implement this app but don't worry we'll do low level stuff later on. The main goal of this chapter is to get familiar with the *flashing* and debugging process.

Throughout this text we'll be using the starter code that's in the discovery repository. Make sure you always have the latest version of the master branch because this website tracks that branch.

The starter code is in the `src` directory of that repository. Inside that directory there are more directories named after each chapter of this book. Most of those directories are starter Cargo projects.

Now, jump into the `src/05-led-roulette` directory. Check the `src/main.rs` file:

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

use aux5::entry;

#[entry]
fn main() -> ! {
    let _y;
    let x = 42;
    _y = x;

    // infinite loop; just so we don't leave this stack frame
    loop {}
}
```

Microcontroller programs are different from standard programs in two aspects: `#![no_std]` and `#![no_main]`.

The `no_std` attribute says that this program won't use the `std` crate, which assumes an underlying OS; the program will instead use the `core` crate, a subset of `std` that can run on bare metal systems (i.e., systems without OS abstractions like files and sockets).

The `no_main` attribute says that this program won't use the standard `main` interface, which is tailored for command line applications that receive arguments. Instead of the standard `main` we'll use the `entry` attribute from the `cortex-m-rt` crate to define a custom entry point. In this program we have named the entry point "main", but any other name could have been used. The entry point function must have signature `fn() -> !`; this type indicates that the function can't return -- this means that the program never terminates.

If you are a careful observer, you'll also notice there is a `.cargo` directory in the Cargo project as well. This directory contains a Cargo configuration file (`.cargo/config`) that tweaks the linking process to tailor the memory layout of the program to the requirements of the target device. This modified linking process is a requirement of the `cortex-m-rt` crate.

Alright, let's start by building this program.

Build it

The first step is to build our "binary" crate. Because the microcontroller has a different architecture than your laptop we'll have to cross compile. Cross compiling in Rust land is as simple as passing an extra `--target` flag to `rustc` or Cargo. The complicated part is figuring out the argument of that flag: the *name* of the target.

The microcontroller in the F3 has a Cortex-M4F processor in it. `rustc` knows how to cross compile to the Cortex-M architecture and 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-M4**F** and Cortex-M7**F** processors

For the F3, we'll use the `thumbv7em-none-eabihf` target. Before cross compiling you have to download pre-compiled version of the standard library (a reduced version of it actually) for your target. That's done using `rustup`:

```
$ rustup target add thumbv7em-none-eabihf
```

You only need to do the above step once; `rustup` will re-install a new standard library (`rust-std` component) whenever you update your toolchain.

With the `rust-std` component in place you can now cross compile the program using Cargo:

```
$ # make sure you are in the `src/05-led-roulette` directory

$ cargo build --target thumbv7em-none-eabihf
   Compiling semver-parser v0.7.0
   Compiling aligned v0.1.1
   Compiling libc v0.2.35
   Compiling bare-metal v0.1.1
   Compiling cast v0.2.2
   Compiling cortex-m v0.4.3
   (..)
   Compiling stm32f30x v0.6.0
   Compiling stm32f30x-hal v0.1.2
   Compiling aux5 v0.1.0 (file://$PWD/aux)
   Compiling led-roulette v0.1.0 (file://$PWD)
    Finished dev [unoptimized + debuginfo] target(s) in 35.84 secs
```

NOTE Be sure to compile this crate *without* optimizations. The provided Cargo.toml file and build command above will ensure optimizations are off.

OK, now we have produced an executable. This executable won't blink any leds, it's just a simplified version that we will build upon later in the chapter. As a sanity check, let's verify that the produced executable is actually an ARM binary:

```
$ # equivalent to `readelf -h target/thumbv7em-none-eabihf/debug/led-roulette`
$ cargo readobj --target thumbv7em-none-eabihf --bin led-roulette -- -file-
headers
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0x0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x8000197
  Start of program headers:          52 (bytes into file)
  Start of section headers:          740788 (bytes into file)
  Flags:                             0x5000400
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         20
  Section header string table index: 18
```

Next, we'll flash the program into our microcontroller.

Flash it

Flashing is the process of moving our program into the microcontroller's (persistent) memory. Once flashed, the microcontroller will execute the flashed program every time it is powered on.

In this case, our `led-roulette` program will be the *only* program in the microcontroller memory. By this I mean that there's nothing else running on the microcontroller: no OS, no "daemon", nothing. `led-roulette` has full control over the device.

Onto the actual flashing. First thing we need is to do is launch OpenOCD. We did that in the previous section but this time we'll run the command inside a temporary directory (`/tmp` on *nix; `%TEMP%` on Windows).

Make sure the F3 is connected to your laptop and run the following commands on a new terminal.

```
$ # *nix
$ cd /tmp

$ # Windows
$ cd %TEMP%

$ # Windows: remember that you need an extra `-s
%PATH_TO_OPENOCD%\share\scripts`
$ openocd \
  -f interface/stlink-v2-1.cfg \
  -f target/stm32f3x.cfg
```

NOTE Older revisions of the board need to pass slightly different arguments to `openocd`. Review this section for the details.

The program will block; leave that terminal open.

Now it's a good time to explain what this command is actually doing.

I mentioned that the F3 actually has two microcontrollers. One of them is used as a programmer/debugger. The part of the board that's used as a programmer is called ST-LINK (that's what STMicroelectronics decided to call it). This ST-LINK is connected to the target microcontroller using a Serial Wire Debug (SWD) interface (this interface is an ARM standard so you'll run into it when dealing with other Cortex-M based microcontrollers). This SWD interface can be used to flash and debug a microcontroller. The ST-LINK is connected to the "USB ST-LINK" port and will appear as a USB device when you connect the F3 to your laptop.

On-board ST-LINK

As for OpenOCD, it's software that provides some services like a *GDB server* on top of USB devices that expose a debugging protocol like SWD or JTAG.

Onto the actual command: those `.cfg` files we are using instruct OpenOCD to look for a ST-LINK USB device (`interface/stlink-v2-1.cfg`) and to expect a STM32F3XX microcontroller (`target/stm32f3x.cfg`) to be connected to the ST-LINK.

The OpenOCD output looks like this:

```
Open On-Chip Debugger 0.9.0 (2016-04-27-23:18)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "hla_swd". To override
use 'transport select <transport>'.
adapter speed: 1000 kHz
adapter_nsrst_delay: 100
Info : The selected transport took over low-level target control. The results
might differ compared to plain JTAG/SWD
none separate
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : clock speed 950 kHz
Info : STLINK v2 JTAG v27 API v2 SWIM v15 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 2.919073
Info : stm32f3x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

The "6 breakpoints, 4 watchpoints" part indicates the debugging features the processor has available.

Leave that `openocd` process running, and open a new terminal. Make sure that you are inside the project's `src/05-led-roulette/` directory.

I mentioned that OpenOCD provides a GDB server so let's connect to that right now:

```
$ <gdb> -q target/thumbv7em-none-eabihf/debug/led-roulette
Reading symbols from target/thumbv7em-none-eabihf/debug/led-roulette...done.
(gdb)
```

NOTE: `<gdb>` represents a GDB program capable of debugging ARM binaries. This could be `arm-none-eabi-gdb`, `gdb-multiarch` or `gdb` depending on your system -- you may have to try all three.

This only opens a GDB shell. To actually connect to the OpenOCD GDB server, use the following command within the GDB shell:

```
(gdb) target remote :3333
Remote debugging using :3333
0x00000000 in ?? ()
```

NOTE: If you are getting errors like `undefined debug reason 7 - target needs reset`, you can try running `monitor reset halt` as described here.

NOTE: If the debugger is still not connecting to the OpenOCD server, then you may need to try using `arm-none-eabi-gdb` instead of the `gdb` command, as described above.

By default OpenOCD's GDB server listens on TCP port 3333 (localhost). This command is connecting to that port.

After entering this command, you'll see new output in the OpenOCD terminal:

```
 Info : stm32f3x.cpu: hardware has 6 breakpoints, 4 watchpoints
+Info : accepting 'gdb' connection on tcp/3333
+Info : device id = 0x10036422
+Info : flash size = 256kbytes
```

Almost there. To flash the device, we'll use the `load` command inside the GDB shell:

```
(gdb) load
Loading section .vector_table, size 0x188 lma 0x8000000
Loading section .text, size 0x38a lma 0x8000188
Loading section .rodata, size 0x8 lma 0x8000514
Start address 0x8000188, load size 1306
Transfer rate: 6 KB/sec, 435 bytes/write.
```

And that's it. You'll also see new output in the OpenOCD terminal.

```
 Info : flash size = 256kbytes
+Info : Unable to match requested speed 1000 kHz, using 950 kHz
+Info : Unable to match requested speed 1000 kHz, using 950 kHz
+adapter speed: 950 kHz
+target state: halted
+target halted due to debug-request, current mode: Thread
+xPSR: 0x01000000 pc: 0x08000194 msp: 0x2000a000
+Info : Unable to match requested speed 8000 kHz, using 4000 kHz
+Info : Unable to match requested speed 8000 kHz, using 4000 kHz
+adapter speed: 4000 kHz
+target state: halted
+target halted due to breakpoint, current mode: Thread
+xPSR: 0x61000000 pc: 0x2000003a msp: 0x2000a000
+Info : Unable to match requested speed 1000 kHz, using 950 kHz
+Info : Unable to match requested speed 1000 kHz, using 950 kHz
+adapter speed: 950 kHz
+target state: halted
+target halted due to debug-request, current mode: Thread
+xPSR: 0x01000000 pc: 0x08000194 msp: 0x2000a000
```

Our program is loaded, let's debug it!

Debug it

We are already inside a debugging session so let's debug our program.

After the `load` command, our program is stopped at its *entry point*. This is indicated by the "Start address 0x8000XXX" part of GDB's output. The entry point is the part of a program that a processor / CPU will execute first.

The starter project I've provided to you has some extra code that runs *before* the `main` function. At this time, we are not interested in that "pre-main" part so let's skip right to the beginning of the `main` function. We'll do that using a breakpoint:

```
(gdb) break main
Breakpoint 1 at 0x800018c: file src/05-led-roulette/src/main.rs, line 10.

(gdb) continue
Continuing.
Note: automatically using hardware breakpoints for read-only addresses.

Breakpoint 1, main () at src/05-led-roulette/src/main.rs:10
10          let x = 42;
```

Breakpoints can be used to stop the normal flow of a program. The `continue` command will let the program run freely *until* it reaches a breakpoint. In this case, until it reaches the `main` function because there's a breakpoint there.

Note that GDB output says "Breakpoint 1". Remember that our processor can only use six of these breakpoints so it's a good idea to pay attention to these messages.

For a nicer debugging experience, we'll be using GDB's Text User Interface (TUI). To enter into that mode, on the GDB shell enter the following command:

```
(gdb) layout src
```

> **NOTE** Apologies Windows users. The GDB shipped with the GNU ARM Embedded Toolchain doesn't support this TUI mode `:-(`.

GDB session

At any point you can leave the TUI mode using the following command:

```
(gdb) tui disable
```

OK. We are now at the beginning of `main`. We can advance the program statement by statement using the `step` command. So let's use that twice to reach the `_y = x` statement. Once you've typed `step` once you can just hit enter to run it again.

```
(gdb) step
14          _y = x;
```

If you are not using the TUI mode, on each `step` call GDB will print back the current statement along with its line number.

We are now "on" the `_y = x` statement; that statement hasn't been executed yet. This means that `x` is initialized but `_y` is not. Let's inspect those stack/local variables using the `print` command:

```
(gdb) print x
$1 = 42

(gdb) print &x
$2 = (i32 *) 0x10001ff4

(gdb) print _y
$3 = -536810104

(gdb) print &_y
$4 = (i32 *) 0x10001ff0
```

As expected, `x` contains the value `42`. `_y`, however, contains the value `-536810104` (?). Because `_y` has not been initialized yet, it contains some garbage value.

The command `print &x` prints the address of the variable `x`. The interesting bit here is that GDB output shows the type of the reference: `i32*`, a pointer to an `i32` value. Another interesting thing is that the addresses of `x` and `_y` are very close to each other: their addresses are just `4` bytes apart.

Instead of printing the local variables one by one, you can also use the `info locals` command:

```
(gdb) info locals
x = 42
_y = -536810104
```

OK. With another `step`, we'll be on top of the `loop {}` statement:

```
(gdb) step
17            loop {}
```

And `_y` should now be initialized.

```
(gdb) print _y
$5 = 42
```

If we use `step` again on top of the `loop {}` statement, we'll get stuck because the program will never pass that statement. Instead, we'll switch to the disassemble view with the `layout asm` command and advance one instruction at a time using `stepi`. You can always switch back into Rust source code view later by issuing the `layout src` command again.

> **NOTE** If you used the `step` command by mistake and GDB got stuck, you can get unstuck by hitting `Ctrl+C`.

```
(gdb) layout asm
```

GDB session

If you are not using the TUI mode, you can use the `disassemble /m` command to disassemble the program around the line you are currently at.

```
(gdb) disassemble /m
Dump of assembler code for function main:
7       #[entry]
   0x08000188 <+0>:     sub     sp, #8
   0x0800018a <+2>:     movs    r0, #42 ; 0x2a

8       fn main() -> ! {
9           let _y;
10          let x = 42;
   0x0800018c <+4>:     str     r0, [sp, #4]

11          _y = x;
   0x0800018e <+6>:     ldr     r0, [sp, #4]
   0x08000190 <+8>:     str     r0, [sp, #0]

12
13          // infinite loop; just so we don't leave this stack frame
14          loop {}
=> 0x08000192 <+10>:    b.n     0x8000194 <main+12>
   0x08000194 <+12>:    b.n     0x8000194 <main+12>

End of assembler dump.
```

See the fat arrow `=>` on the left side? It shows the instruction the processor will execute next.

If not inside the TUI mode on each `stepi` command GDB will print the statement, the line number *and* the address of the instruction the processor will execute next.

```
(gdb) stepi
0x08000194      14          loop {}

(gdb) stepi
0x08000194      14          loop {}
```

One last trick before we move to something more interesting. Enter the following commands into GDB:

```
(gdb) monitor reset halt
Unable to match requested speed 1000 kHz, using 950 kHz
Unable to match requested speed 1000 kHz, using 950 kHz
adapter speed: 950 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08000196 msp: 0x10002000

(gdb) continue
Continuing.

Breakpoint 1, main () at src/05-led-roulette/src/main.rs:10
10          let x = 42;
```

We are now back at the beginning of `main`!

`monitor reset halt` will reset the microcontroller and stop it right at the program entry point. The following `continue` command will let the program run freely until it reaches the `main` function that has a breakpoint on it.

This combo is handy when you, by mistake, skipped over a part of the program that you were interested in inspecting. You can easily roll back the state of your program back to its very beginning.

> **The fine print**: This `reset` command doesn't clear or touch RAM. That memory will retain its values from the previous run. That shouldn't be a problem though, unless your program behavior depends of the value of *uninitialized* variables but that's the definition of Undefined Behavior (UB).

We are done with this debug session. You can end it with the `quit` command.

```
(gdb) quit
A debugging session is active.

        Inferior 1 [Remote target] will be detached.

Quit anyway? (y or n) y
Detaching from program: $PWD/target/thumbv7em-none-eabihf/debug/led-roulette,
Remote target
Ending remote debugging.
```

> **NOTE** If the default GDB CLI is not to your liking check out gdb-dashboard. It uses Python to turn the default GDB CLI into a dashboard that shows registers, the source view, the assembly view and other things.

Don't close OpenOCD though! We'll use it again and again later on. It's better just to leave it running.

What's next? The high level API I promised.

The `Led` and `Delay` abstractions

Now, I'm going to introduce two high level abstractions that we'll use to implement the LED roulette application.

The auxiliary crate, `aux5`, exposes an initialization function called `init`. When called this function returns two values packed in a tuple: a `Delay` value and a `Leds` value.

`Delay` can be used to block your program for a specified amount of milliseconds.

`Leds` is actually an array of eight `Led`s. Each `Led` represents one of the LEDs on the F3 board, and exposes two methods: `on` and `off` which can be used to turn the LED on or off, respectively.

Let's try out these two abstractions by modifying the starter code to look like this:

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

use aux5::{entry, prelude::*, Delay, Leds};

#[entry]
fn main() -> ! {
    let (mut delay, mut leds): (Delay, Leds) = aux5::init();

    let half_period = 500_u16;

    loop {
        leds[0].on();
        delay.delay_ms(half_period);

        leds[0].off();
        delay.delay_ms(half_period);
    }
}
```

Now build it:

```
$ cargo build --target thumbv7em-none-eabihf
```

NOTE It's possible to forget to rebuild the program *before* starting a GDB session; this omission can lead to very confusing debug sessions. To avoid this problem you can call `cargo run` instead of `cargo build`; `cargo run` will build *and* start a debug session ensuring you never forget to recompile your program.

Now, we'll repeat the flashing procedure that we did in the previous section:

```
$ # this starts a GDB session of the program; no need to specify the path to the
binary
$ arm-none-eabi-gdb -q target/thumbv7em-none-eabihf/debug/led-roulette
Reading symbols from target/thumbv7em-none-eabihf/debug/led-roulette...done.
(gdb) target remote :3333
Remote debugging using :3333
(..)

(gdb) load
Loading section .vector_table, size 0x188 lma 0x8000000
Loading section .text, size 0x3fc6 lma 0x8000188
Loading section .rodata, size 0xa0c lma 0x8004150
Start address 0x8000188, load size 19290
Transfer rate: 19 KB/sec, 4822 bytes/write.

(gdb) break main
Breakpoint 1 at 0x800018c: file src/05-led-roulette/src/main.rs, line 9.

(gdb) continue
Continuing.
Note: automatically using hardware breakpoints for read-only addresses.

Breakpoint 1, main () at src/05-led-roulette/src/main.rs:9
9           let (mut delay, mut leds): (Delay, Leds) = aux5::init();
```

OK. Let's step through the code. This time, we'll use the `next` command instead of `step`. The difference is that the `next` command will step *over* function calls instead of going inside them.

```
(gdb) next
11          let half_period = 500_u16;

(gdb) next
13          loop {

(gdb) next
14              leds[0].on();

(gdb) next
15              delay.delay_ms(half_period);
```

After executing the `leds[0].on()` statement, you should see a red LED, the one pointing North, turn on.

Let's continue stepping over the program:

```
(gdb) next
17              leds[0].off();

(gdb) next
18              delay.delay_ms(half_period);
```

The `delay_ms` call will block the program for half a second but you may not notice because the `next` command also takes some time to execute. However, after stepping over the `leds[0].off()` statement you should see the red LED turn off.

You can already guess what this program does. Let it run uninterrupted using the `continue` command.

```
(gdb) continue
Continuing.
```

Now, let's do something more interesting. We are going to modify the behavior of our program using GDB.

First, let's stop the infinite loop by hitting `Ctrl+C`. You'll probably end up somewhere inside `Led::on`, `Led::off` or `delay_ms`:

```
Program received signal SIGINT, Interrupt.
0x080033f6 in core::ptr::read_volatile (src=0xe000e010) at
/checkout/src/libcore/ptr.rs:472
472     /checkout/src/libcore/ptr.rs: No such file or directory.
```

In my case, the program stopped its execution inside a `read_volatile` function. GDB output shows some interesting information about that: `core::ptr::read_volatile (src=0xe000e010)`. This means that the function comes from the `core` crate and that it was called with argument `src = 0xe000e010`.

Just so you know, a more explicit way to show the arguments of a function is to use the `info args` command:

```
(gdb) info args
src = 0xe000e010
```

Regardless of where your program may have stopped you can always look at the output of the `backtrace` command (`bt` for short) to learn how it got there:

```
(gdb) backtrace
#0  0x080033f6 in core::ptr::read_volatile (src=0xe000e010)
    at /checkout/src/libcore/ptr.rs:472
#1  0x08003248 in <vcell::VolatileCell<T>>::get (self=0xe000e010)
    at $REGISTRY/vcell-0.1.0/src/lib.rs:43
#2  <volatile_register::RW<T>>::read (self=0xe000e010)
    at $REGISTRY/volatile-register-0.2.0/src/lib.rs:75
#3  cortex_m::peripheral::syst::<impl cortex_m::peripheral::SYST>::has_wrapped
(self=0x10001fbc)
    at $REGISTRY/cortex-m-0.5.7/src/peripheral/syst.rs:124
#4  0x08002d9c in <stm32f30x_hal::delay::Delay as
embedded_hal::blocking::delay::DelayUs<u32>>::delay_us (self=0x10001fbc,
us=500000)
    at $REGISTRY/stm32f30x-hal-0.2.0/src/delay.rs:58
#5  0x08002cce in <stm32f30x_hal::delay::Delay as
embedded_hal::blocking::delay::DelayMs<u32>>::delay_ms (self=0x10001fbc, ms=500)
    at $REGISTRY/stm32f30x-hal-0.2.0/src/delay.rs:32
#6  0x08002d0e in <stm32f30x_hal::delay::Delay as
embedded_hal::blocking::delay::DelayMs<u16>>::delay_ms (self=0x10001fbc, ms=500)
    at $REGISTRY/stm32f30x-hal-0.2.0/src/delay.rs:38
#7  0x080001ee in main () at src/05-led-roulette/src/main.rs:18
```

`backtrace` will print a trace of function calls from the current function down to main.

Back to our topic. To do what we are after, first, we have to return to the `main` function. We can do that using the `finish` command. This command resumes the program execution and stops it again right after the program returns from the current function. We'll have to call it several times.

```
(gdb) finish
cortex_m::peripheral::syst::<impl cortex_m::peripheral::SYST>::has_wrapped
(self=0x10001fbc)
    at $REGISTRY/cortex-m-0.5.7/src/peripheral/syst.rs:124
124             self.csr.read() & SYST_CSR_COUNTFLAG != 0
Value returned is $1 = 5

(gdb) finish
Run till exit from #0  cortex_m::peripheral::syst::<impl
cortex_m::peripheral::SYST>::has_wrapped (
    self=0x10001fbc)
    at $REGISTRY/cortex-m-0.5.7/src/peripheral/syst.rs:124
0x08002d9c in <stm32f30x_hal::delay::Delay as
embedded_hal::blocking::delay::DelayUs<u32>>::delay_us (
    self=0x10001fbc, us=500000)
    at $REGISTRY/stm32f30x-hal-0.2.0/src/delay.rs:58
58              while !self.syst.has_wrapped() {}
Value returned is $2 = false

(..)

(gdb) finish
Run till exit from #0  0x08002d0e in <stm32f30x_hal::delay::Delay as
embedded_hal::blocking::delay::DelayMs<u16>>::delay_ms (self=0x10001fbc, ms=500)
    at $REGISTRY/stm32f30x-hal-0.2.0/src/delay.rs:38
0x080001ee in main () at src/05-led-roulette/src/main.rs:18
18              delay.delay_ms(half_period);
```

We are back in `main`. We have a local variable in here: `half_period`

```
(gdb) info locals
half_period = 500
delay = (..)
leds = (..)
```

Now, we are going to modify this variable using the `set` command:

```
(gdb) set half_period = 100

(gdb) print half_period
$1 = 100
```

If you let program run free again using the `continue` command, you should see that the LED will blink at a much faster rate now!

Question! What happens if you keep lowering the value of `half_period`? At what value of `half_period` you can no longer see the LED blink?

Now, it's your turn to write a program.

The challenge

You are now well armed to face a challenge! Your task will be to implement the application I showed you at the beginning of this chapter.

Here's the GIF again:

Also, this may help:

This is a timing diagram. It indicates which LED is on at any given instant of time and for how long each LED should be on. On the X axis we have the time in milliseconds. The timing diagram shows a single period. This pattern will repeat itself every 800 ms. The Y axis labels each LED with a cardinal point: North, East, etc. As part of the challenge you'll have to figure out how each element in the `Leds` array maps to these cardinal points (hint: `cargo doc --open` `;-)`).

Before you attempt this challenge, let me give you one last tip. Our GDB sessions always involve entering the same commands at the beginning. We can use a `.gdb` file to execute some commands right after GDB is started. This way you can save yourself the effort of having to enter them manually on each GDB session.

Place this `openocd.gdb` file in the root of the Cargo project, right next to the `Cargo.toml`:

```
$ cat openocd.gdb
```

```
target remote :3333
load
break main
continue
```

Then modify the second line of the `.cargo/config` file:

```
$ cat .cargo/config
```

```
[target.thumbv7em-none-eabihf]
runner = "arm-none-eabi-gdb -q -x openocd.gdb" # <-
rustflags = [
  "-C", "link-arg=-Tlink.x",
]
```

With that in place, you should now be able to start a `gdb` session that will automatically flash the program and jump to the beginning of `main`:

```
$ cargo run --target thumbv7em-none-eabihf
     Running `arm-none-eabi-gdb -q -x openocd.gdb target/thumbv7em-none-
eabihf/debug/led-roulette`
Reading symbols from target/thumbv7em-none-eabihf/debug/led-roulette...done.
(..)
Loading section .vector_table, size 0x188 lma 0x8000000
Loading section .text, size 0x3b20 lma 0x8000188
Loading section .rodata, size 0xb0c lma 0x8003cc0
Start address 0x8003b1c, load size 18356
Transfer rate: 20 KB/sec, 6118 bytes/write.
Breakpoint 1 at 0x800018c: file src/05-led-roulette/src/main.rs, line 9.
Note: automatically using hardware breakpoints for read-only addresses.

Breakpoint 1, main () at src/05-led-roulette/src/main.rs:9
9           let (mut delay, mut leds): (Delay, Leds) = aux5::init();
(gdb)
```

My solution

What solution did you come up with?

Here's mine:

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

use aux5::{entry, prelude::*, Delay, Leds};

#[entry]
fn main() -> ! {
    let (mut delay, mut leds): (Delay, Leds) = aux5::init();

    let ms = 50_u8;
    loop {
        for curr in 0..8 {
            let next = (curr + 1) % 8;

            leds[next].on();
            delay.delay_ms(ms);
            leds[curr].off();
            delay.delay_ms(ms);
        }
    }
}
```

One more thing! Check that your solution also works when compiled in "release" mode:

```
$ cargo build --target thumbv7em-none-eabihf --release
```

You can test it with this `gdb` command:

```
$ # or, you could simply call `cargo run --target thumbv7em-none-eabihf --
release`
$ arm-none-eabi-gdb target/thumbv7em-none-eabihf/release/led-roulette
$ #                                                     ~~~~~~~
```

Binary size is something we should always keep an eye on! How big is your solution? You can check that using the `size` command on the release binary:

```
$ # equivalent to size target/thumbv7em-none-eabihf/debug/led-roulette
$ cargo size --target thumbv7em-none-eabihf --bin led-roulette -- -A
led-roulette  :
section                 size        addr
.vector_table            392   0x8000000
.text                  16404   0x8000188
.rodata                 2924   0x80041a0
.data                      0  0x20000000
.bss                       4  0x20000000
.debug_str            602185         0x0
.debug_abbrev          24134         0x0
.debug_info           553143         0x0
.debug_ranges         112744         0x0
.debug_macinfo            86         0x0
.debug_pubnames        56467         0x0
.debug_pubtypes        94866         0x0
.ARM.attributes           58         0x0
.debug_frame          174812         0x0
.debug_line           354866         0x0
.debug_loc               534         0x0
.comment                  75         0x0
Total                1993694

$ cargo size --target thumbv7em-none-eabihf --bin led-roulette --release -- -A
led-roulette  :
section                size        addr
.vector_table           392   0x8000000
.text                  1826   0x8000188
.rodata                  84   0x80008ac
.data                     0  0x20000000
.bss                      4  0x20000000
.debug_str            23334         0x0
.debug_loc             6964         0x0
.debug_abbrev          1337         0x0
.debug_info           40582         0x0
.debug_ranges          2936         0x0
.debug_macinfo            1         0x0
.debug_pubnames        5470         0x0
.debug_pubtypes       10016         0x0
.ARM.attributes          58         0x0
.debug_frame            164         0x0
.debug_line            9081         0x0
.comment                 18         0x0
Total                102267
```

NOTE The Cargo project is already configured to build the release binary using LTO.

Know how to read this output? The `text` section contains the program instructions. It's around 2KB in my case. On the other hand, the `data` and `bss` sections contain variables statically allocated in RAM (`static` variables). A `static` variable is being used in `aux5::init`; that's why it shows 4 bytes of `bss`.

One final thing! We have been running our programs from within GDB but our programs don't depend on GDB at all. You can confirm this be closing both GDB and OpenOCD and then resetting the board by pressing the black button on the board. The LED roulette application will run without intervention of GDB.

Hello, world!

> **HEADS UP** Several readers have reported that the "solder bridge" SB10 (see back of the board) on the STM32F3DISCOVERY, which is required to use the ITM and the `iprint!` macros shown below, is **not** soldered even though the User Manual (page 21) says that it **should be**.

> **TL;DR** You have two options to fix this: Either **solder** the solder bridge SB10 or connect a female to female jumper wire between SWO and PB3 as shown in the picture below.

Manual SWD connection

Just a little more of helpful magic before we start doing low level stuff.

Blinking an LED is like the "Hello, world" of the embedded world.

But in this section, we'll run a proper "Hello, world" program that prints stuff to your laptop console.

Go to the `06-hello-world` directory. There's some starter code in it:

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

#[allow(unused_imports)]
use aux6::{entry, iprint, iprintln};

#[entry]
fn main() -> ! {
    let mut itm = aux6::init();

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

    loop {}
}
```

The `iprintln` macro will format messages and output them to the microcontroller's *ITM*. ITM stands for Instrumentation Trace Macrocell and it's a communication protocol on top of SWD (Serial Wire Debug) which can be used to send messages from the microcontroller to the debugging host. This communication is only *one way*: the debugging host can't send data to the microcontroller.

OpenOCD, which is managing the debug session, can receive data sent through this ITM *channel* and redirect it to a file.

The ITM protocol works with *frames* (you can think of them as Ethernet frames). Each frame has a header and a variable length payload. OpenOCD will receive these frames and write them directly to a file without parsing them. So, if the microntroller sends the string "Hello, world!" using the `iprintln` macro, OpenOCD's output file won't exactly contain that string.

To retrieve the original string, OpenOCD's output file will have to be parsed. We'll use the `itmdump` program to perform the parsing as new data arrives.

You should have already installed the `itmdump` program during the installation chapter.

In a new terminal, run this command inside the `/tmp` directory, if you are using a *nix OS, or from within the `%TEMP%` directory, if you are running Windows. This should be the same directory from where you are running OpenOCD.

> **NOTE** It's very important that both `itmdump` and `openocd` are running from the same directory!

```
$ # itmdump terminal

$ # *nix
$ cd /tmp && touch itm.txt

$ # Windows
$ cd %TEMP% && type nul >> itm.txt

$ # both
$ itmdump -F -f itm.txt
```

This command will block as `itmdump` is now watching the `itm.txt` file. Leave this terminal open.

Alright. Now, let's build the starter code and flash it into the microcontroller.

To avoid passing the `--target thumbv7em-none-eabihf` flag to every Cargo invocation we can set a default target in .cargo/config:

```
 [target.thumbv7em-none-eabihf]
 runner = "arm-none-eabi-gdb -q -x openocd.gdb"
 rustflags = [
   "-C", "link-arg=-Tlink.x",
 ]

+[build]
+target = "thumbv7em-none-eabihf"
```

Now if `--target` is not specified Cargo will assume that the target is `thumbv7em-none-eabihf`.

```
$ cargo run
Reading symbols from target/thumbv7em-none-eabihf/debug/hello-world...done.
(..)
Loading section .vector_table, size 0x400 lma 0x8000000
Loading section .text, size 0x27c4 lma 0x8000400
Loading section .rodata, size 0x744 lma 0x8002be0
Start address 0x8002980, load size 13064
Transfer rate: 18 KB/sec, 4354 bytes/write.
Breakpoint 1 at 0x8000402: file src/06-hello-world/src/main.rs, line 10.
Note: automatically using hardware breakpoints for read-only addresses.

Breakpoint 1, main () at src/06-hello-world/src/main.rs:10
10          let mut itm = aux6::init();
```

Note that there's a `openocd.gdb` at the root of the Cargo project. It's pretty similar to the one we used in the previous section.

Before we execute the `iprintln!` statement. We have to instruct OpenOCD to redirect the ITM output into the same file that `itmdump` is watching.

```
(gdb) # globally enable the ITM and redirect all output to itm.txt
(gdb) monitor tpiu config internal itm.txt uart off 8000000

(gdb) # enable the ITM port 0
(gdb) monitor itm port 0 on
```

All should be ready! Now execute the `iprintln!` statement.

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

(gdb) next
14          loop {}
```

You should see some output in the `itmdump` terminal:

```
$ itmdump -F -f itm.txt
(..)
Hello, world!
```

Awesome, right? Feel free to use `iprintln` as a logging tool in the coming sections.

Next: That's not all! The `iprint!` macros are not the only thing that uses the ITM. `:-)`

`panic!`

The `panic!` macro also sends its output to the ITM!

Change the `main` function to look like this:

```
#[entry]
fn main() -> ! {
    panic!("Hello, world!");
}
```

Let's try this program. But before that let's update `openocd.gdb` to run that `monitor` stuff for us during GDB startup:

```
 target remote :3333
 set print asm-demangle on
 set print pretty on
 load
+monitor tpiu config internal itm.txt uart off 8000000
+monitor itm port 0 on
 break main
 continue
```

OK, now run it.

```
$ cargo run
(..)
Breakpoint 1, main () at src/06-hello-world/src/main.rs:10
10          panic!("Hello, world!");

(gdb) next
```

You'll see some new output in the `itmdump` terminal.

```
$ # itmdump terminal
(..)
panicked at 'Hello, world!', src/06-hello-world/src/main.rs:10:5
```

Another thing you can do is catch the panic *before* it does the logging by putting a breakpoint on the `rust_begin_unwind` symbol.

```
(gdb) monitor reset halt
(..)
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x080026ba msp: 0x10002000

(gdb) break rust_begin_unwind
Breakpoint 2 at 0x80011d2: file $REGISTRY/panic-itm-0.4.0/src/lib.rs, line 46.

(gdb) continue
Continuing.

Breakpoint 2, rust_begin_unwind (info=0x10001fac) at $REGISTRY/panic-itm-
0.4.0/src/lib.rs:46
46          interrupt::disable();
```

You'll notice that nothing got printed on the `itmdump` console this time. If you resume the program using `continue` then a new line will be printed.

In a later section we'll look into other simpler communication protocols.

Registers

It's time to explore what the `Led` API does under the hood.

In a nutshell, it just writes to some special memory regions. Go into the `07-registers` directory and let's run the starter code statement by statement.

```rust
#![no_main]
#![no_std]

#[allow(unused_imports)]
use aux7::{entry, iprint, iprintln};

#[entry]
fn main() -> ! {
    aux7::init();

    unsafe {
        // A magic address!
        const GPIOE_BSRR: u32 = 0x48001018;

        // Turn on the "North" LED (red)
        *(GPIOE_BSRR as *mut u32) = 1 << 9;

        // Turn on the "East" LED (green)
        *(GPIOE_BSRR as *mut u32) = 1 << 11;

        // Turn off the "North" LED
        *(GPIOE_BSRR as *mut u32) = 1 << (9 + 16);

        // Turn off the "East" LED
        *(GPIOE_BSRR as *mut u32) = 1 << (11 + 16);
    }

    loop {}
}
```

What's this magic?

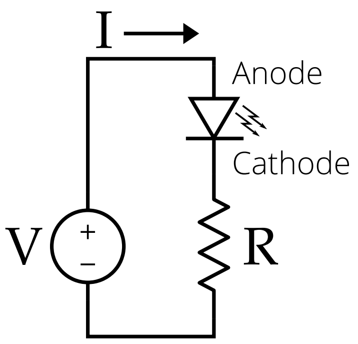
The address `0x48001018` points to a *register*. A register is a special region of memory that controls a *peripheral*. A peripheral is a piece of electronics that sits right next to the processor within the microcontroller package and provides the processor with extra functionality. After all, the processor, on its own, can only do math and logic.

This particular register controls General Purpose Input/Output (GPIO) *pins* (GPIO *is* a peripheral) and can be used to *drive* each of those pins *low* or *high*.

An aside: LEDs, digital outputs and voltage levels

Drive? Pin? Low? High?

A pin is a electrical contact. Our microcontroller has several of them and some of them are connected to LEDs. An LED, a Light Emitting Diode, will only emit light when voltage is applied to it with a certain polarity.



Luckily for us, the microcontroller's pins are connected to the LEDs with the right polarity. All that we have to do is *output* some non-zero voltage through the pin to turn the LED on. The pins attached to the LEDs are configured as *digital outputs* and can only output two different voltage levels: "low", 0 Volts, or "high", 3 Volts. A "high" (voltage) level will turn the LED on whereas a "low" (voltage) level will turn it off.

These "low" and "high" states map directly to the concept of digital logic. "low" is `0` or `false` and "high" is `1` or `true`. This is why this pin configuration is known as digital output.

OK. But how can one find out what this register does? Time to RTRM (Read the Reference Manual)!

RTRM: Reading The Reference Manual

I mentioned that the microcontroller has several pins. For convenience, these pins are grouped in *ports* of 16 pins. Each port is named with a letter: Port A, Port B, etc. and the pins within each port are named with numbers from 0 to 15.

The first thing we have to find out is which pin is connected to which LED. This information is in the STM32F3DISCOVERY User Manual (You downloaded a copy, right?). In this particular section:

> Section 6.4 LEDs - Page 18

The manual says:

- `LD3`, the North LED, is connected to the pin `PE9`. `PE9` is the short form of: Pin 9 on Port E.
- `LD7`, the East LED, is connected to the pin `PE11`.

Up to this point, we know that we want to change the state of the pins PE9 and PE11 to turn the North/East LEDs on/off. These pins are part of Port E so we'll have to deal with the `GPIOE` peripheral.

Each peripheral has a register *block* associated to it. A register block is a collection of registers allocated in contiguous memory. The address at which the register block starts is known as its base address. We need to figure out what's the base address of the `GPIOE` peripheral. That information is in the following section of the microcontroller Reference Manual:

> Section 3.2.2 Memory map and register boundary addresses - Page 51

The table says that base address of the `GPIOE` register block is `0x4800_1000`.

Each peripheral also has its own section in the documentation. Each of these sections ends with a table of the registers that the peripheral's register block contains. For the `GPIO` family of peripheral, that table is in:

> Section 11.4.12 GPIO register map - Page 243

We are interested in the register that's at an offset of `0x18` from the base address of the `GPIOE` peripheral. According to the table, that would be the register `BSRR`.

Now we need to jump to the documentation of that particular register. It's a few pages above in:

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

Finally!

This is the register we were writing to. The documentation says some interesting things. First, this register is write only ... so let's try reading its value `:-)`.

We'll use GDB's `examine` command: `x`.

```
(gdb) next
16              *(GPIOE_BSRR as *mut u32) = 1 << 9;

(gdb) x 0x48001018
0x48001018:     0x00000000

(gdb) # the next command will turn the North LED on
(gdb) next
19              *(GPIOE_BSRR as *mut u32) = 1 << 11;

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

Reading the register returns `0`. That matches what the documentation says.

The other thing that the documentation says is that the bits 0 to 15 can be used to *set* the corresponding pin. That is bit 0 sets the pin 0. Here, *set* means outputting a *high* value on the pin.

The documentation also says that bits 16 to 31 can be used to *reset* the corresponding pin. In this case, the bit 16 resets the pin number 0. As you may guess, *reset* means outputting a *low* value on the pin.

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

- Writing `1 << 9` (`BS9 = 1`) to `BSRR` sets `PE9` *high*. That turns the North LED *on*.
- Writing `1 << 11` (`BS11 = 1`) to `BSRR` sets `PE11` *high*. That turns the East LED *on*.
- Writing `1 << 25` (`BR9 = 1`) to `BSRR` sets `PE9` *low*. That turns the North LED *off*.
- Finally, writing `1 << 27` (`BR11 = 1`) to `BSRR` sets `PE11` *low*. That turns the East LED *off*.

(mis)Optimization

Reads/writes to registers are quite special. I may even dare to say that they are embodiment of side effects. In the previous example we wrote four different values to the same register. If you didn't know that address was a register, you may have simplified the logic to just write the final value `1 << (11 + 16)` into the register.

Actually, LLVM, the compiler's backend / optimizer, does not know we are dealing with a register and will merge the writes thus changing the behavior of our program. Let's check that really quick.

```
$ cargo run --release
(..)
Breakpoint 1, main () at src/07-registers/src/main.rs:9
9           aux7::init();

(gdb) next
25              *(GPIOE_BSRR as *mut u32) = 1 << (11 + 16);

(gdb) disassemble /m
Dump of assembler code for function main:
7       #[entry]

8       fn main() -> ! {
9           aux7::init();
   0x08000188 <+0>:     bl      0x800019c <aux7::init>
   0x0800018c <+4>:     movw    r0, #4120       ; 0x1018
   0x08000190 <+8>:     mov.w   r1, #134217728  ; 0x8000000
   0x08000194 <+12>:    movt    r0, #18432      ; 0x4800

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
28          loop {}
   0x0800019a <+18>:    b.n     0x800019a <main+18>

End of assembler dump.
```

The state of the LEDs didn't change this time! The `str` instruction is the one that writes a value to the register. Our *debug* (unoptimized) program had four of them, one for each write to the register, but the *release* (optimized) program only has one.

We can check that using `objdump`:

```
$ # same as cargo objdump -- -d -no-show-raw-insn -print-imm-hex -source
target/thumbv7em-none-eabihf/debug/registers
$ cargo objdump --bin registers -- -d -no-show-raw-insn -print-imm-hex -source
registers:      file format ELF32-arm-little

Disassembly of section .text:
main:
; #[entry]
 8000188:       sub     sp, #0x18
; aux7::init();
 800018a:       bl      #0xbc
 800018e:       str     r0, [sp, #0x14]
 8000190:       b       #-0x2 <main+0xa>
; *(GPIOE_BSRR as *mut u32) = 1 << 9;
 8000192:       b       #-0x2 <main+0xc>
 8000194:       movw    r0, #0x1018
 8000198:       movt    r0, #0x4800
 800019c:       mov.w   r1, #0x200
 80001a0:       str     r1, [r0]
; *(GPIOE_BSRR as *mut u32) = 1 << 11;
 80001a2:       b       #-0x2 <main+0x1c>
 80001a4:       movw    r0, #0x1018
 80001a8:       movt    r0, #0x4800
 80001ac:       mov.w   r1, #0x800
 80001b0:       str     r1, [r0]
 80001b2:       movs    r0, #0x19
; *(GPIOE_BSRR as *mut u32) = 1 << (9 + 16);
 80001b4:       mov     r1, r0
 80001b6:       cmp     r0, #0x9
 80001b8:       str     r1, [sp, #0x10]
 80001ba:       bvs     #0x54 <main+0x8a>
 80001bc:       b       #-0x2 <main+0x36>
 80001be:       ldr     r0, [sp, #0x10]
 80001c0:       and     r1, r0, #0x1f
 80001c4:       movs    r2, #0x1
 80001c6:       lsl.w   r1, r2, r1
 80001ca:       lsrs    r2, r0, #0x5
 80001cc:       cmp     r2, #0x0
 80001ce:       str     r1, [sp, #0xc]
 80001d0:       bne     #0x4c <main+0x98>
 80001d2:       b       #-0x2 <main+0x4c>
 80001d4:       movw    r0, #0x1018
 80001d8:       movt    r0, #0x4800
 80001dc:       ldr     r1, [sp, #0xc]
 80001de:       str     r1, [r0]
 80001e0:       movs    r0, #0x1b
; *(GPIOE_BSRR as *mut u32) = 1 << (11 + 16);
 80001e2:       mov     r2, r0
 80001e4:       cmp     r0, #0xb
 80001e6:       str     r2, [sp, #0x8]
 80001e8:       bvs     #0x42 <main+0xa6>
 80001ea:       b       #-0x2 <main+0x64>
 80001ec:       ldr     r0, [sp, #0x8]
 80001ee:       and     r1, r0, #0x1f
 80001f2:       movs    r2, #0x1
 80001f4:       lsl.w   r1, r2, r1
 80001f8:       lsrs    r2, r0, #0x5
 80001fa:       cmp     r2, #0x0
 80001fc:       str     r1, [sp, #0x4]
 80001fe:       bne     #0x3a <main+0xb4>
 8000200:       b       #-0x2 <main+0x7a>
 8000202:       movw    r0, #0x1018
 8000206:       movt    r0, #0x4800
 800020a:       ldr     r1, [sp, #0x4]
 800020c:       str     r1, [r0]
; loop {}
 800020e:       b       #-0x2 <main+0x88>
 8000210:       b       #-0x4 <main+0x88>
; *(GPIOE_BSRR as *mut u32) = 1 << (9 + 16);
 8000212:       movw    r0, #0x41bc
 8000216:       movt    r0, #0x800
 800021a:       bl      #0x3b28
 800021e:       trap
 8000220:       movw    r0, #0x4204
 8000224:       movt    r0, #0x800
 8000228:       bl      #0x3b1a
 800022c:       trap
; *(GPIOE_BSRR as *mut u32) = 1 << (11 + 16);
 800022e:       movw    r0, #0x421c
 8000232:       movt    r0, #0x800
 8000236:       bl      #0x3b0c
 800023a:       trap
 800023c:       movw    r0, #0x4234
 8000240:       movt    r0, #0x800
 8000244:       bl      #0x3afe
 8000248:       trap
```

How do we prevent LLVM from misoptimizing our program? We use *volatile* operations instead of plain reads/writes:

```
#![no_main]
#![no_std]

use core::ptr;

#[allow(unused_imports)]
use aux7::{entry, iprint, iprintln};

#[entry]
fn main() -> ! {
    aux7::init();

    unsafe {
        // A magic address!
        const GPIOE_BSRR: u32 = 0x48001018;

        // Turn on the "North" LED (red)
        ptr::write_volatile(GPIOE_BSRR as *mut u32, 1 << 9);

        // Turn on the "East" LED (green)
        ptr::write_volatile(GPIOE_BSRR as *mut u32, 1 << 11);

        // Turn off the "North" LED
        ptr::write_volatile(GPIOE_BSRR as *mut u32, 1 << (9 + 16));

        // Turn off the "East" LED
        ptr::write_volatile(GPIOE_BSRR as *mut u32, 1 << (11 + 16));
    }

    loop {}
}
```

If we look at the disassembly of this new program compiled in release mode:

```
$ cargo objdump --bin registers --release -- -d -no-show-raw-insn -print-imm-hex
-source
registers:      file format ELF32-arm-little

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>
```

We see that the four writes (`str` instructions) are preserved. If you run it (use `stepi`), you'll also see that behavior of the program is preserved.

`0xBAAAAAAD` address

Not all the peripheral memory can be accessed. Look at this program.

```
#![no_main]
#![no_std]

use core::ptr;

#[allow(unused_imports)]
use aux7::{entry, iprint, iprintln};

#[entry]
fn main() -> ! {
    aux7::init();

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

    loop {}
}
```

This address is close to the `GPIOE_BSRR` address we used before but this address is *invalid*. Invalid in the sense that there's no register at this address.

Now, let's try it.

```
$ 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 {
```

We tried to do an invalid operation, reading memory that doesn't exist, so the processor raised an *exception*, a *hardware* exception.

In most cases, exceptions are raised when the processor attempts to perform an invalid operation. Exceptions break the normal flow of a program and force the processor to execute an *exception handler*, which is just a function/subroutine.

There are different kind of exceptions. Each kind of exception is raised by different conditions and each one is handled by a different exception handler.

The `aux7` crate depends on the `cortex-m-rt` crate which defines a default *hard fault* handler, named `UserHardFault`, that handles the "invalid memory address" exception. `openocd.gdb` placed a breakpoint on `HardFault`; that's why the debugger halted your program while it was executing the exception handler. We can get more information about the exception from the debugger. Let's see:

```
(gdb) list
530
531     #[allow(unused_variables)]
532     #[doc(hidden)]
533     #[no_mangle]
534     pub unsafe extern "C" fn UserHardFault_(ef: &ExceptionFrame) -> ! {
535         loop {
536             // add some side effect to prevent this from turning into a UDF
instruction
537             // see rust-lang/rust#28728 for details
538             atomic::compiler_fence(Ordering::SeqCst);
539         }
```

`ef` is a snapshot of the program state right before the exception occurred. Let's inspect it:

```
(gdb) print/x *ef
$1 = cortex_m_rt::ExceptionFrame {
  r0: 0x48001800,
  r1: 0x48001800,
  r2: 0xb,
  r3: 0xc,
  r12: 0xd,
  lr: 0x800019f,
  pc: 0x80028d6,
  xpsr: 0x1000000
}
```

There are several fields here but the most important one is `pc`, the Program Counter register. The address in this register points to the instruction that generated the exception. Let's disassemble the program around the bad instruction.

```
(gdb) disassemble /m ef.pc
Dump of assembler code for function core::ptr::read_volatile:
471     /checkout/src/libcore/ptr.rs: No such file or directory.
   0x080028ce <+0>:     sub     sp, #16
   0x080028d0 <+2>:     mov     r1, r0
   0x080028d2 <+4>:     str     r0, [sp, #8]

472     in /checkout/src/libcore/ptr.rs
   0x080028d4 <+6>:     ldr     r0, [sp, #8]
   0x080028d6 <+8>:     ldr     r0, [r0, #0]
   0x080028d8 <+10>:    str     r0, [sp, #12]
   0x080028da <+12>:    ldr     r0, [sp, #12]
   0x080028dc <+14>:    str     r1, [sp, #4]
   0x080028de <+16>:    str     r0, [sp, #0]
   0x080028e0 <+18>:    b.n     0x80028e2 <core::ptr::read_volatile+20>

473     in /checkout/src/libcore/ptr.rs
   0x080028e2 <+20>:    ldr     r0, [sp, #0]
   0x080028e4 <+22>:    add     sp, #16
   0x080028e6 <+24>:    bx      lr

End of assembler dump.
```

The exception was caused by the `ldr r0, [r0, #0]` instruction, a read instruction. The instruction tried to read the memory at the address indicated by the `r0` register. By the way, `r0` is a CPU (processor) register not a memory mapped register; it doesn't have an associated address like, say, `GPIO_BSRR`.

Wouldn't it be nice if we could check what the value of the `r0` register was right at the instant when the exception was raised? Well, we already did! The `r0` field in the `ef` value we printed before is the value of `r0` register had when the exception was raised. Here it is again:

```
(gdb) p/x *ef
$1 = cortex_m_rt::ExceptionFrame {
  r0: 0x48001800,
  r1: 0x48001800,
  r2: 0xb,
  r3: 0xc,
  r12: 0xd,
  lr: 0x800019f,
  pc: 0x80028d6,
  xpsr: 0x1000000
}
```

`r0` contains the value `0x4800_1800` which is the invalid address we called the `read_volatile` function with.

Spooky action at a distance

`BSRR` is not the only register that can control the pins of Port E. The `ODR` register also lets you change the value of the pins. Furthermore, `ODR` also lets you retrieve the current output status of Port E.

`ODR` is documented in:

> Section 11.4.6 GPIO port output data register - Page 239

Let's try this program:

```rust
#![no_main]
#![no_std]

use core::ptr;

#[allow(unused_imports)]
use aux7::{entry, iprint, iprintln};

#[entry]
fn main() -> ! {
    let mut itm = aux7::init().0;

    unsafe {
        const GPIOE_BSRR: u32 = 0x4800_1018;
        const GPIOE_ODR: u32 = 0x4800_1014;

        iprintln!(
            &mut itm.stim[0],
            "ODR = 0x{:04x}",
            ptr::read_volatile(GPIOE_ODR as *const u16)
        );

        // Turn on the NORTH LED (red)
        ptr::write_volatile(GPIOE_BSRR as *mut u32, 1 << 9);

        iprintln!(
            &mut itm.stim[0],
            "ODR = 0x{:04x}",
            ptr::read_volatile(GPIOE_ODR as *const u16)
        );

        // Turn on the EAST LED (green)
        ptr::write_volatile(GPIOE_BSRR as *mut u32, 1 << 11);

        iprintln!(
            &mut itm.stim[0],
            "ODR = 0x{:04x}",
            ptr::read_volatile(GPIOE_ODR as *const u16)
        );

        // Turn off the NORTH LED
        ptr::write_volatile(GPIOE_BSRR as *mut u32, 1 << (9 + 16));

        iprintln!(
            &mut itm.stim[0],
            "ODR = 0x{:04x}",
            ptr::read_volatile(GPIOE_ODR as *const u16)
        );

        // Turn off the EAST LED
        ptr::write_volatile(GPIOE_BSRR as *mut u32, 1 << (11 + 16));
    }

    loop {}
}
```

If you run this program, you'll see:

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

Side effects! Although we are reading the same address multiple times without actually modifying it, we still see its value change every time `BSRR` is written to.

Type safe manipulation

The last register we were working with, `ODR`, had this in its documentation:

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

We are not supposed to write to those bits of the register or Bad Stuff May Happen.

There's also the fact the registers have different read/write permissions. Some of them are write only, others can be read and wrote to and there must be others that are read only.

Finally, directly working with hexadecimal addresses is error prone. You already saw that trying to access an invalid memory address causes an exception which disrupts the execution of our program.

Wouldn't it be nice if we had an API to manipulate registers in a "safe" manner? Ideally, the API should encode these three points I've mentioned: No messing around with the actual addresses, should respect read/write permissions and should prevent modification of the reserved parts of a register.

Well, we do! `aux7::init()` actually returns a value that provides a type safe API to manipulate the registers of the `GPIOE` peripheral.

As you may remember: a group of registers associated to a peripheral is called register block, and it's located in a contiguous region of memory. In this type safe API each register block is modeled as a `struct` where each of its fields represents a register. Each register field is a different newtype over e.g. `u32` that exposes a combination of the following methods: `read`, `write` or `modify` according to its read/write permissions. Finally, these methods don't take primitive values like `u32`, instead they take yet another newtype that can be constructed using the builder pattern and that prevent the modification of the reserved parts of the register.

The best way to get familiar with this API is to port our running example to it.

```rust
#![no_main]
#![no_std]

#[allow(unused_imports)]
use aux7::{entry, iprint, iprintln};

#[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 {}
}
```

First thing you notice: There are no magic addresses involved. Instead we use a more human friendly way, for example `gpioe.bsrr`, to refer to the `BSRR` register in the `GPIOE` register block.

Then we have this `write` method that takes a closure. If the identity closure (`|w| w`) is used, this method will set the register to its *default* (reset) value, the value it had right after the microcontroller was powered on / reset. That value is `0x0` for the `BSRR` register. Since we want to write a non-zero value to the register, we use builder methods like `bs9` and `br9` to set some of the bits of the default value.

Let's run this program! There's some interesting stuff we can do *while* debugging the program.

`gpioe` is a reference to the `GPIOE` register block. `print gpioe` will return the base address of the register block.

```
$ cargo run
Breakpoint 3, main () at src/07-registers/src/main.rs:9
9           let gpioe = aux7::init().1;

(gdb) next
12          gpioe.bsrr.write(|w| w.bs9().set_bit());

(gdb) print gpioe
$1 = (stm32f30x::gpioc::RegisterBlock *) 0x48001000
```

But if we instead `print *gpioe`, we'll get a *full view* of the register block: the value of each of its registers will be printed.

```
(gdb) print *gpioe
$2 = stm32f30x::gpioc::RegisterBlock {
  moder: stm32f30x::gpioc::MODER {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x55550000
      }
    }
  },
  otyper: stm32f30x::gpioc::OTYPER {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0
      }
    }
  },
  ospeedr: stm32f30x::gpioc::OSPEEDR {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0
      }
    }
  },
  pupdr: stm32f30x::gpioc::PUPDR {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0
      }
    }
  },
  idr: stm32f30x::gpioc::IDR {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0xcc
      }
    }
  },
  odr: stm32f30x::gpioc::ODR {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0
      }
    }
  },
  bsrr: stm32f30x::gpioc::BSRR {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0
      }
    }
  },
  lckr: stm32f30x::gpioc::LCKR {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0
      }
    }
  },
  afrl: stm32f30x::gpioc::AFRL {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0
      }
    }
  },
  afrh: stm32f30x::gpioc::AFRH {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0
      }
    }
  },
  brr: stm32f30x::gpioc::BRR {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0
      }
    }
  }
}
```

All these newtypes and closures sound like they'd generate large, bloated programs but, if you actually compile the program in release mode with LTO enabled, you'll see that it produces exactly the same instructions that the "unsafe" version that used `write_volatile` and hexadecimal addresses did!

```
$ cargo objdump --bin registers --release -- -d -no-show-raw-insn -print-imm-hex
registers:      file format ELF32-arm-little

Disassembly of section .text:
main:
 8000188:       bl      #0x22
 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]
 80001ac:       b       #-0x4 <main+0x24>
```

The best part of all this is that I didn't have to write a single line of code to implement the GPIOE API. All was automatically generated from a System View Description (SVD) file using the svd2rust tool. This SVD file is actually an XML file that microcontroller vendors provide and that contains the register maps of their microcontrollers. The file contains the layout of register blocks, the base addresses, the read/write permissions of each register, the layout of the registers, whether a register has reserved bits and lots of other useful information.

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.

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.

This is the starter code.

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

use aux8::entry;

#[entry]
fn main() -> ! {
    let (gpioe, rcc) = aux8::init();

    // TODO initialize GPIOE

    // Turn on all the LEDs in the compass
    gpioe.odr.write(|w| {
        w.odr8().set_bit();
        w.odr9().set_bit();
        w.odr10().set_bit();
        w.odr11().set_bit();
        w.odr12().set_bit();
        w.odr13().set_bit();
        w.odr14().set_bit();
        w.odr15().set_bit()
    });

    aux8::bkpt();

    loop {}
}
```

If you run the starter code, you'll see that nothing happens this time. Furthermore, if you print the `GPIOE` register block, you'll see that every register reads as zero even after the `gpioe.odr.write` statement was executed!

```
$ cargo run
Breakpoint 1, main () at src/08-leds-again/src/main.rs:9
9           let (gpioe, rcc) = aux8::init();

(gdb) continue
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x08000f3c in __bkpt ()

(gdb) finish
Run till exit from #0  0x08000f3c in __bkpt ()
main () at src/08-leds-again/src/main.rs:25
25          aux8::bkpt();

(gdb) p/x *gpioe
$1 = stm32f30x::gpioc::RegisterBlock {
  moder: stm32f30x::gpioc::MODER {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0
      }
    }
  },
  otyper: stm32f30x::gpioc::OTYPER {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0
      }
    }
  },
  ospeedr: stm32f30x::gpioc::OSPEEDR {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0
      }
    }
  },
  pupdr: stm32f30x::gpioc::PUPDR {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0
      }
    }
  },
  idr: stm32f30x::gpioc::IDR {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0
      }
    }
  },
  odr: stm32f30x::gpioc::ODR {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0
      }
    }
  },
  bsrr: stm32f30x::gpioc::BSRR {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0
      }
    }
  },
  lckr: stm32f30x::gpioc::LCKR {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0
      }
    }
  },
  afrl: stm32f30x::gpioc::AFRL {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0
      }
    }
  },
  afrh: stm32f30x::gpioc::AFRH {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0x0
      }
    }
  },
  brr: stm32f30x::gpioc::BRR {
    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.

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`

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

Your task in this section is to power on the `GPIOE` peripheral. You'll have to:

- Figure out which of the three registers I mentioned before has the bit that controls the power status.
- Figure out what value that bit must be set to, `0` or `1`, to power on the `GPIOE` peripheral.
- Finally, you'll have to change the starter code to *modify* the right register to turn on the `GPIOE` peripheral.

If you are successful, you'll see that the `gpioe.odr.write` statement will now be able to modify the value of the `ODR` register.

Note that this won't be enough to actually turn on the LEDs.

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*.

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`.

Your task for this section is to further update the starter code to configure the *right* `GPIOE` pins as digital outputs. You'll have to:

- Figure out *which* pins you need to configure as digital outputs. (hint: check Section 6.4 LEDs of the *User Manual* (page 18)).
- Read the documentation to understand what the bits in the `MODER` register do.
- Modify the `MODER` register to configure the pins as digital outputs.

If successful, you'll see the 8 LEDs turn on when you run the program.

The solution

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

use aux8::entry;

#[entry]
fn main() -> ! {
    let (gpioe, rcc) = aux8::init();

    // enable the GPIOE peripheral
    rcc.ahbenr.modify(|_, w| w.iopeen().set_bit());

    // configure the pins as outputs
    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()
    });

    // Turn on all the LEDs in the compass
    gpioe.odr.write(|w| {
        w.odr8().set_bit();
        w.odr9().set_bit();
        w.odr10().set_bit();
        w.odr11().set_bit();
        w.odr12().set_bit();
        w.odr13().set_bit();
        w.odr14().set_bit();
        w.odr15().set_bit()
    });

    aux8::bkpt();

    loop {}
}
```

Clocks and timers

In this section, we'll re-implement the LED roulette application. I'm going to give you back the `Led` abstraction but this time I'm going to take away the `Delay` abstraction `:-)`.

Here's the starter code. The `delay` function is unimplemented so if you run this program the LEDs will blink so fast that they'll appear to always be on.

```rust
#![no_main]
#![no_std]

use aux9::{entry, tim6};

#[inline(never)]
fn delay(tim6: &tim6::RegisterBlock, ms: u16) {
    // TODO implement this
}

#[entry]
fn main() -> ! {
    let (mut leds, rcc, tim6) = aux9::init();

    // TODO initialize TIM6

    let ms = 50;
    loop {
        for curr in 0..8 {
            let next = (curr + 1) % 8;

            leds[next].on();
            delay(tim6, ms);
            leds[curr].off();
            delay(tim6, ms);
        }
    }
}
```

`for` loop delays

The first challenge is to implement the `delay` function without using any peripheral and the obvious solution is to implement it as a `for` loop delay:

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

Of course, the above implementation is wrong because it always generates the same delay for any value of `ms`.

In this section, you'll have to:

- Fix the `delay` function to generate delays proportional to its input `ms`.
- Tweak the `delay` function to make the LED roulette spin at a rate of approximately 5 cycles in 4 seconds (800 milliseconds period).
- The processor inside the microcontroller is clocked at 8 MHz and executes most instructions in one "tick", a cycle of its clock. How many (`for`) loops do you *think* the `delay` function must do to generate a delay of 1 second?

- How many `for` loops does `delay(1000)` actually do?
- What happens if compile your program in release mode and run it?

NOP

If in the previous section you compiled the program in release mode and actually looked at the disassembly, you probably noticed that the `delay` function is optimized away and never gets called from within `main`.

LLVM decided that the function wasn't doing anything worthwhile and just removed it.

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; // this value needs to be tweaked
    for _ in 0..(K * ms) {
        aux9::nop()
    }
}
```

And this time `delay` won't be compiled away by LLVM when you compile your program in release mode:

```
$ cargo objdump --bin clocks-and-timers --release -- -d -no-show-raw-insn
clocks-and-timers:      file format ELF32-arm-little

Disassembly of section .text:
clocks_and_timers::delay::h711ce9bd68a6328f:
 8000188:       push    {r4, r5, r7, lr}
 800018a:       movs    r4, #0
 800018c:       adds    r4, #1
 800018e:       uxth    r5, r4
 8000190:       bl      #4666
 8000194:       cmp     r5, #150
 8000196:       blo     #-14 <clocks_and_timers::delay::h711ce9bd68a6328f+0x4>
 8000198:       pop     {r4, r5, r7, pc}
```

Now, test this: Compile the program in debug mode and run it, then compile the program in release mode and run it. What's the difference between them? What do you think is the main cause of the difference? Can you think of a way to make them equivalent or at least more similar again?

One-shot timer

I hope that, by now, I have convinced you that `for` loop delays are a poor way to implement delays.

Now, we'll implement delays using a *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.

The microcontroller we are using has several (in fact, more than 10) timers of different kinds (basic, general purpose, and advanced timers) available to it. Some timers have more *resolution* (number of bits) than others and some can be used for more than just keeping track of time.

We'll be using one of the *basic* timers: `TIM6`. This is one of the simplest timers available in our microcontroller. The documentation for basic timers is in the following section:

> Section 22 Timers - Page 670 - Reference Manual

Its registers are documented in:

> Section 22.4.9 TIM6/TIM7 register map - Page 682 - Reference Manual

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.
- `ARR`, the autoreload register.

We'll be using the timer as a *one-shot* timer. It will sort of work like an alarm clock. We'll set the timer to go off after some amount of time and then we'll wait until the timer goes off. The documentation refers to this mode of operation as *one pulse mode*.

Here's a description of how a basic timer works when configured in one pulse mode:

- The counter is enabled by the user (`CR1.CEN = 1`).
- The `CNT` register resets its value to zero and, on each tick, its value gets incremented by one.
- 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`).

`TIM6` is driven by the APB1 clock, whose frequency doesn't have to necessarily match the processor frequency. That is, the APB1 clock could be running faster or slower. The default, however, is that both APB1 and the processor are clocked at 8 MHz.

The tick mentioned in the functional description of the one pulse mode is *not* the same as one tick of the APB1 clock. The `CNT` register increases at a frequency of `apb1 / (psc + 1)` times per second, where `apb1` is the frequency of the APB1 clock and `psc` is the value of the prescaler register, `PSC`.

Initialization

As with every other peripheral, we'll have to initialize this timer before we can use it. And just as in the previous section, initialization is going to involve two steps: powering up the timer and then configuring it.

Powering up the timer is easy: We just have to set `TIM6EN` bit to 1. This bit is in the `APB1ENR` register of the `RCC` register block.

```
    // Power on the TIM6 timer
    rcc.apb1enr.modify(|_, w| w.tim6en().set_bit());
```

The configuration part is slightly more elaborate.

First, we'll have to configure the timer to operate in one pulse mode.

```
    // OPM Select one pulse mode
    // CEN Keep the counter disabled for now
    tim6.cr1.write(|w| w.opm().set_bit().cen().clear_bit());
```

Then, we'll like to have the `CNT` counter operate at a frequency of 1 KHz because our `delay` function takes a number of milliseconds as arguments and 1 KHz produces a 1 millisecond period. For that we'll have to configure the prescaler.

```
    // Configure the prescaler to have the counter operate at 1 KHz
    tim6.psc.write(|w| w.psc().bits(psc));
```

I'm going to let you figure out the value of the prescaler, `psc`. Remember that the frequency of the counter is `apb1 / (psc + 1)` and that `apb1` is 8 MHz.

Busy waiting

The timer should now be properly initialized. All that's left is to implement the `delay` function using the timer.

First thing we have to do is set the autoreload register (`ARR`) to make the timer go off in `ms` milliseconds. Because the counter operates at 1 KHz, the autoreload value will be the same as `ms`.

```
    // Set the timer to go off in `ms` ticks
    // 1 tick = 1 ms
    tim6.arr.write(|w| w.arr().bits(ms));
```

Next, we need to enable the counter. It will immediately start counting.

```
    // CEN: Enable the counter
    tim6.cr1.modify(|_, w| w.cen().set_bit());
```

Now we need to wait until the counter reaches the value of the autoreload register, `ms`, then we'll know that `ms` milliseconds have passed. That condition is known as an *update event* and its indicated by the `UIF` bit of the status register (`SR`).

```
    // Wait until the alarm goes off (until the update event occurs)
    while !tim6.sr.read().uif().bit_is_set() {}
```

This pattern of just waiting until some condition is met, in this case that `UIF` becomes `1`, is known as *busy waiting* and you'll see it a few more times in this text `:-)`.

Finally, we must clear (set to `0`) this `UIF` bit. If we don't, next time we enter the `delay` function we'll think the update event has already happened and skip over the busy waiting part.

```
    // Clear the update event flag
    tim6.sr.modify(|_, w| w.uif().clear_bit());
```

Now, put this all together and check if it works as expected.

Putting it all together

```rust
#![no_main]
#![no_std]

use aux9::{entry, tim6};

#[inline(never)]
fn delay(tim6: &tim6::RegisterBlock, ms: u16) {
    // Set the timer to go off in `ms` ticks
    // 1 tick = 1 ms
    tim6.arr.write(|w| w.arr().bits(ms));

    // CEN: Enable the counter
    tim6.cr1.modify(|_, w| w.cen().set_bit());

    // Wait until the alarm goes off (until the update event occurs)
    while !tim6.sr.read().uif().bit_is_set() {}

    // Clear the update event flag
    tim6.sr.modify(|_, w| w.uif().clear_bit());
}

#[entry]
fn main() -> ! {
    let (mut leds, rcc, tim6) = aux9::init();

    // Power on the TIM6 timer
    rcc.apb1enr.modify(|_, w| w.tim6en().set_bit());

    // OPM Select one pulse mode
    // CEN Keep the counter disabled for now
    tim6.cr1.write(|w| w.opm().set_bit().cen().clear_bit());

    // Configure the prescaler to have the counter operate at 1 KHz
    // APB1_CLOCK = 8 MHz
    // PSC = 7999
    // 8 MHz / (7999 + 1) = 1 KHz
    // The counter (CNT) will increase on every millisecond
    tim6.psc.write(|w| w.psc().bits(7_999));

    let ms = 50;
    loop {
        for curr in 0..8 {
            let next = (curr + 1) % 8;

            leds[next].on();
            delay(tim6, ms);
            leds[curr].off();
            delay(tim6, ms);
        }
    }
}
```

Serial communication

This is what we'll be using. I hope your laptop has one!

Nah, don't worry. This connector, the DE-9, went out of fashion on PCs quite some time ago; it got replaced by the Universal Serial Bus (USB). We won't be dealing with the DE-9 connector itself but with the communication protocol that this cable is/was usually used for.

So what's this *serial communication*? It's an *asynchronous* communication protocol where two devices exchange data *serially*, as in one bit at a time, using two data lines (plus a common ground). The protocol is asynchronous in the sense that neither of the shared lines carries a clock signal. Instead both parties must agree on how fast data will be sent along the wire *before* the communication occurs. This protocol allows *duplex* communication as data can be sent from A to B and from B to A simultaneously.

We'll be using this protocol to exchange data between the microcontroller and your laptop. In contrast to the ITM protocol we have used before, with the serial communication protocol you can send data from your laptop to the microcontroller.

The next practical question you probably want to ask is: How fast can we send data through this protocol?

This protocol works with frames. Each frame has one *start* bit, 5 to 9 bits of payload (data) and 1 to 2 *stop bits*. The speed of the protocol is known as *baud rate* and it's quoted in bits per second (bps). Common baud rates are: 9600, 19200, 38400, 57600 and 115200 bps.

To actually answer the question: With a common configuration of 1 start bit, 8 bits of data, 1 stop bit and a baud rate of 115200 bps one can, in theory, send 11,520 frames per second. Since each one frame carries a byte of data that results in a data rate of 11.52 KB/s. In practice, the data rate will probably be lower because of processing times on the slower side of the communication (the microcontroller).

Today's laptops/PCs don't support the serial communication protocol. So you can't directly connect your laptop to the microcontroller. But that's where the serial module comes in. This module will sit between the two and expose a serial interface to the microcontroller and an USB interface to your laptop. The microcontroller will see your laptop as another serial device and your laptop will see the microcontroller as a virtual serial device.

Now, let's get familiar with the serial module and the serial communication tools that your OS offers. Pick a route:

- *nix
- Windows

*nix tooling

Connect the serial module to your laptop and let's find out what name the OS assigned to it.

> **NOTE** On macs, the USB device will named like this: `/dev/cu.usbserial-*`. You won't find it using `dmesg`, instead use `ls -l /dev | grep cu.usb` and adjust the following commands accordingly!

```
$ dmesg | grep -i tty
(..)
[  +0.000155] usb 3-2: FTDI USB Serial Device converter now attached to ttyUSB0
```

But what's this `ttyUSB0` thing? It's a file of course! Everything is a file in *nix:

```
$ ls -l /dev/ttyUSB0
crw-rw-rw- 1 root uucp 188, 0 Oct 27 00:00 /dev/ttyUSB0
```

> **NOTE** if the permissions above is `crw-rw----`, the udev rules have not been set correctly see udev rules

You can send out data by simply writing to this file:

```
$ echo 'Hello, world!' > /dev/ttyUSB0
```

You should see the TX (red) LED on the serial module blink, just once and very fast!

minicom

Dealing with serial devices using `echo` is far from ergonomic. So, we'll use the program `minicom` to interact with the serial device using the keyboard.

We must configure `minicom` before we use it. There are quite a few ways to do that but we'll use a `.minirc.dfl` file in the home directory. Create a file in `~/.minirc.dfl` with the following contents:

```
$ cat ~/.minirc.dfl
pu baudrate 115200
pu bits 8
pu parity N
pu stopbits 1
pu rtscts No
pu xonxoff No
```

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

That file should be straightforward to read (except for the last two lines), but nonetheless let's go over it line by line:

- `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.

Once that's in place. We can launch `minicom`

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

This tells `minicom` to open the serial device at `/dev/ttyUSB0` and set its baud rate to 115200. A text-based user interface (TUI) will pop out.

minicom

You can now send data using the keyboard! Go ahead and type something. Note that the TUI *won't* echo back what you type but you'll see TX (red) LED on the serial module blink with each keystroke.

`minicom` commands

`minicom` exposes commands via keyboard shortcuts. On Linux, the shortcuts start with `Ctrl+A`. On mac, the shortcuts start with the `Meta` key. Some useful commands below:

- `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

NOTE mac users: In the above commands, replace `Ctrl+A` with `Meta`.

Windows tooling

Before plugging the Serial module, run the following command on the terminal:

```
$ mode
```

It will print a list of devices that are connected to your laptop. The ones that start with `COM` in their names are serial devices. This is the kind of device we'll be working with. Take note of all the `COM` *ports* `mode` outputs *before* plugging the serial module.

Now, plug the Serial module and run the `mode` command again. You should see a new `COM` port appear on the list. That's the COM port assigned to the serial module.

Now launch `putty`. A GUI will pop out.

PuTTY settings

On the starter screen, which should have the "Session" category open, pick "Serial" as the "Connection type". On the "Serial line" field enter the `COM` device you got on the previous step, for example `COM3`.

Next, pick the "Connection/Serial" category from the menu on the left. On this new view, make sure that the serial port is configured as follows:

- "Speed (baud)": 115200
- "Data bits": 8
- "Stop bits": 1
- "Parity": None
- "Flow control": None

Finally, click the Open button. A console will show up now:

PuTTY console

If you type on this console, the TX (red) LED on the Serial module should blink. Each key stroke should make the LED blink once. Note that the console won't echo back what you type so the screen will remain blank.

Loopbacks

We've tested sending data. It's time to test receiving it. Except that there's no other device that can send us some data ... or is there?

Enter: loopbacks

Serial module loopback

You can send data to yourself! Not very useful in production but very useful for debugging.

Connect the `TXO` and the `RXI` pins of the serial module together using a male to male jumper wire as shown above.

Now enter some text into minicom/PuTTY and observe. What happens?

You should see three things:

- As before, the TX (red) LED blinks on each key press.
- But now the RX (green) LED blinks on each key press as well! This indicates that the serial module is receiving some data; the one it just sent.
- Finally, on the minicom/PuTTY console, you should see that what you type echoes back to the console.

Now that you are familiar with sending and receiving data over serial port using minicom/PuTTY, let's make your microcontroller and your laptop talk!

USART

The microcontroller has a peripheral called USART, which stands for Universal Synchronous/Asynchronous Receiver/Transmitter. This peripheral can be configured to work with several communication protocols like the serial communication protocol.

Throughout this chapter, we'll use serial communication to exchange information between the microcontroller and your laptop. But before we do that we have to wire up everything.

I mentioned before that this protocol involves two data lines: TX and RX. TX stands for transmitter and RX stands for receiver. Transmitter and receiver are relative terms though; which line is the transmitter and which line is the receiver depends from which side of the communication you are looking at the lines.

We'll be using the pin `PA9` as the microcontroller's TX line and `PA10` as its RX line. In other words, the pin `PA9` outputs data onto its wire whereas the pin `PA10` listens for data on its wire.

We could have used a different pair of pins as the TX and RX pins. There's a table in page 44 of the Data Sheet that list all the other possible pins we could have used.

The serial module also has TX and RX pins. We'll have to *cross* these pins: that is connect the microcontroller's TX pin to the serial module's RX pin and the micro's RX pin to the serial module's TX pin. The wiring diagram below shows all the necessary connections.

F3 <-> Serial connection

These are the recommended steps to connect the microcontroller and the serial module:

- 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.
- Connect the PA10 pin on the back of the F3 to the TXO pin of the 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's wired up! Let's proceed to send data back and forth.

Send a single byte

Our first task will be to send a single byte from the microcontroller to the laptop over the serial connection.

This time, I'm going to provide you with an already initialized USART peripheral. You'll only have to work with the registers that are in charge of sending and receiving data.

Go into the `11-usart` directory and let's run the starter code therein. Make sure that you have minicom/PuTTY open.

```rust
#![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 {}
}
```

This program writes to the `TDR` register. This causes the `USART` peripheral to send one byte of information through the serial interface.

On the receiving end, your laptop, you should see show the character `X` appear on minicom/PuTTY's terminal.

Send a string

The next task will be to send a whole string from the micro to your laptop.

I want you to send the string `"The quick brown fox jumps over the lazy dog."` from the micro to your laptop.

It's your turn to write the program.

Execute your program inside the debugger, statement by statement. What do you see?

Then execute the program again but in *one go* using the `continue` command. What happens this time?

Finally, build the program in *release* mode and, again, run it one go. What happens this time?

Overruns

If you wrote your program like this:

```
#![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 {}
}
```

You probably received something like this on your laptop when you executed the program compiled in debug mode.

```
$ # minicom's terminal
(..)
The uic brwn oxjums oer helaz do.
```

And if you compiled in release mode, you probably only got something like this:

```
$ # minicom's terminal
(..)
T
```

What went wrong?

You see, sending bytes over the wire takes a relatively large amount of time. I already did the math so let me quote myself:

> With a common configuration of 1 start bit, 8 bits of data, 1 stop bit and a baud rate of 115200 bps one can, in theory, send 11,520 frames per second. Since each one frame carries a byte of data that results in a data rate of 11.52 KB/s

Our pangram has a length of 45 bytes. That means it's going to take, at least, 3,900 microseconds (`45 bytes / (11,520 bytes/s) = 3,906 us`) to send the string. The processor is working at 8 MHz, where executing an instruction takes 125 nanoseconds, so it's likely going to be done with the `for` loop in less than 3,900 microseconds.

We can actually time how long it takes to execute the `for` loop. `aux11::init()` returns a `MonoTimer` (monotonic timer) value that exposes an `Instant` API that's similar to the one in `std::time`.

```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 {}
}
```

In debug mode, I get:

```
$ # itmdump terminal
(..)
`for` loop took 22415 ticks (2801.875 us)
```

This is less than 3,900 microseconds but it's not that far off and that's why only a few bytes of information are lost.

In conclusion, the processor is trying to send bytes at a faster rate than what the hardware can actually handle and this results in data loss. This condition is known as buffer *overrun*.

How do we avoid this? The status register (`ISR`) has a flag, `TXE`, that indicates if it's "safe" to write to the `TDR` register without incurring in data loss.

Let's use that to slowdown the processor.

```
#![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 {}
}
```

This time, running the program in debug or release mode should result in a complete string on the receiving side.

```
$ # minicom/PuTTY's console
(..)
The quick brown fox jumps over the lazy dog.
```

The timing of the `for` loop should be closer to the theoretical 3,900 microseconds as well. The timing below is for the debug version.

```
$ # itmdump terminal
(..)
`for` loop took 30499 ticks (3812.375 us)
```

`uprintln!`

For the next exercise, we'll implement the `uprint!` family of macros. Your goal is to make this line of code work:

```
    uprintln!(serial, "The answer is {}", 40 + 2);
```

Which must send the string `"The answer is 42"` through the serial interface.

How do we go about that? It's informative to look into the `std` implementation of `println!`.

```
// src/libstd/macros.rs
macro_rules! print {
    ($($arg:tt)*) => ($crate::io::_print(format_args!($($arg)*)));
}
```

Looks simple so far. We need the built-in `format_args!` macro (it's implemented in the compiler so we can't see what it actually does). We'll have to use that macro in the exact same way. What does this `_print` function do?

```
// src/libstd/io/stdio.rs
pub fn _print(args: fmt::Arguments) {
    let result = match LOCAL_STDOUT.state() {
        LocalKeyState::Uninitialized |
        LocalKeyState::Destroyed => stdout().write_fmt(args),
        LocalKeyState::Valid => {
            LOCAL_STDOUT.with(|s| {
                if s.borrow_state() == BorrowState::Unused {
                    if let Some(w) = s.borrow_mut().as_mut() {
                        return w.write_fmt(args);
                    }
                }
                stdout().write_fmt(args)
            })
        }
    };
    if let Err(e) = result {
        panic!("failed printing to stdout: {}", e);
    }
}
```

That *looks* complicated but the only part we are interested in is: `w.write_fmt(args)` and `stdout().write_fmt(args)`. What `print!` ultimately does is call the `fmt::Write::write_fmt` method with the output of `format_args!` as its argument.

Luckily we don't have to implement the `fmt::Write::write_fmt` method either because it's a default method. We only have to implement the `fmt::Write::write_str` method.

Let's do that.

This is what the macro side of the equation looks like. What's left to be done by you is provide the implementation of the `write_str` method.

Above we saw that `Write` is in `std::fmt`. We don't have access to `std` but `Write` is also available in `core::fmt`.

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

use core::fmt::{self, Write};

#[allow(unused_imports)]
use aux11::{entry, iprint, iprintln, usart1};

macro_rules! uprint {
    ($serial:expr, $($arg:tt)*) => {
        $serial.write_fmt(format_args!($($arg)*)).ok()
    };
}

macro_rules! uprintln {
    ($serial:expr, $fmt:expr) => {
        uprint!($serial, concat!($fmt, "\n"))
    };
    ($serial:expr, $fmt:expr, $($arg:tt)*) => {
        uprint!($serial, concat!($fmt, "\n"), $($arg)*)
    };
}

struct SerialPort {
    usart1: &'static mut usart1::RegisterBlock,
}

impl fmt::Write for SerialPort {
    fn write_str(&mut self, s: &str) -> fmt::Result {
        // TODO implement this
        // hint: this will look very similar to the previous program
        Ok(())
    }
}

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

    let mut serial = SerialPort { usart1 };

    uprintln!(serial, "The answer is {}", 40 + 2);

    loop {}
}
```

Receive a single byte

So far we have sending data from the micro to your laptop. It's time to try the opposite: receiving data from your laptop.

There's a `RDR` register that will be filled with the data that comes from the RX line. If we read that register, we'll retrieve the data that the other side of the channel sent. The question is: How do we know that we have received (new) data? The status register, `ISR`, has a bit for that purpose: `RXNE`. We can just busy wait on that flag.

```
#![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();
    }
}
```

Let's try this program! Let it run free using `continue` and then type a single character in minicom/PuTTY's console. What happens? What are the contents of the `_byte` variable?

```
(gdb) continue
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x8003d48 in __bkpt ()

(gdb) finish
Run till exit from #0  0x8003d48 in __bkpt ()
usart::main () at src/11-usart/src/main.rs:19
19              aux11::bkpt();

(gdb) p/c _byte
$1 = 97 'a'
```

Echo server

Let's merge transmission and reception into a single program and write an echo server. An echo server sends back to the client the same text it sent. For this application, the microcontroller will be the server and you and your laptop will be the client.

This should be straightforward to implement. (hint: do it byte by byte)

Reverse a string

Alright, next let's make the server more interesting by having it respond to the client with the reverse of the text that they sent. The server will respond to the client every time they press the ENTER key. Each server response will be in a new line.

This time you'll need a buffer; you can use `heapless::Vec`. Here's the starter code:

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

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

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

    // A buffer with 32 bytes of capacity
    let mut buffer: Vec<u8, consts::U32> = Vec::new();

    loop {
        buffer.clear();

        // TODO Receive a user request. Each user request ends with ENTER
        // NOTE `buffer.push` returns a `Result`. Handle the error by responding
        // with an error message.

        // TODO Send back the reversed string
    }
}
```

My solution

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

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

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

    // A buffer with 32 bytes of capacity
    let mut buffer: Vec<u8, consts::U32> = Vec::new();

    loop {
        buffer.clear();

        loop {
            while usart1.isr.read().rxne().bit_is_clear() {}
            let byte = usart1.rdr.read().rdr().bits() as u8;

            if buffer.push(byte).is_err() {
                // buffer full
                for byte in b"error: buffer full\n\r" {
                    while usart1.isr.read().txe().bit_is_clear() {}
                    usart1.tdr.write(|w| w.tdr().bits(u16::from(*byte)));
                }

                break;
            }

            // Carriage return
            if byte == 13 {
                // Respond
                for byte in buffer.iter().rev().chain(&[b'\n', b'\r']) {
                    while usart1.isr.read().txe().bit_is_clear() {}
                    usart1.tdr.write(|w| w.tdr().bits(u16::from(*byte)));
                }

                break;
            }
        }
    }
}
```

Bluetooth setup

It's time to get rid of some wires. Serial communication can not only be emulated on top of the USB protocol; it can also be emulated on top of the Bluetooth protocol. This serial over Bluetooth protocol is known as RFCOMM.

Before we use the Bluetooth module with the microcontroller, let's first interact with it using minicom/PuTTY.

The first thing we'll need to do is: turn on the Bluetooth module. We'll have to share some of the F3 power to it using the following connection:

F3 <-> Bluetooth connection (power only)

The recommend steps to wire this up are:

- Close OpenOCD and `itmdump`
- Disconnect the USB cables from the F3 and the serial module.
- Connect F3's GND pin to the Bluetooth's GND pin using a female to female (F/F) wire. Preferably, a black one.
- Connect F3's 5V pin to the Bluetooth's VCC pin using a F/F wire. Preferably, a red one.
- Then, connect the USB cable back to the F3.
- Re-launch OpenOCD and `itmdump`

Two LEDs, a blue one and a red one, on the Bluetooth module should start blinking right after you power on the F3 board.

Next thing to do is pair your laptop and the Bluetooth module. AFAIK, Windows and mac users can simply use their OS default Bluetooth manager to do the pairing. The Bluetooth module default pin is 1234.

Linux users will have to follow (some of) these instructions.

Linux

If you have a graphical Bluetooth manager, you can use that to pair your laptop to the Bluetooth module and skip most of these steps. You'll probably still have to this step though.

Power up

First, your laptop's Bluetooth transceiver may be OFF. Check its status with `hciconfig` and turn it ON if necessary:

```
$ hciconfig
hci0:   Type: Primary  Bus: USB
        BD Address: 68:17:29:XX:XX:XX  ACL MTU: 310:10  SCO MTU: 64:8
        DOWN  <--
        RX bytes:580 acl:0 sco:0 events:31 errors:0
        TX bytes:368 acl:0 sco:0 commands:30 errors:0

$ sudo hciconfig hci0 up

$ hciconfig
hci0:   Type: Primary  Bus: USB
        BD Address: 68:17:29:XX:XX:XX  ACL MTU: 310:10  SCO MTU: 64:8
        UP RUNNING  <--
        RX bytes:1190 acl:0 sco:0 events:67 errors:0
        TX bytes:1072 acl:0 sco:0 commands:66 errors:0
```

Then you need to launch the BlueZ (Bluetooth) daemon:

- On systemd based Linux distributions, use:

```
$ sudo systemctl start bluetooth
```

- On Ubuntu (or upstart based Linux distributions), use:

```
$ sudo /etc/init.d/bluetooth start
```

You may also need to unblock your Bluetooth, depending on what `rfkill list` says:

```
$ rfkill list
9: hci0: Bluetooth
        Soft blocked: yes # <--
        Hard blocked: no

$ sudo rfkill unblock bluetooth

$ rfkill list
9: hci0: Bluetooth
        Soft blocked: no  # <--
        Hard blocked: no

```

Scan

```
$ hcitool scan
Scanning ...
        20:16:05:XX:XX:XX       Ferris
$ #                             ^^^^^^
```

Pair

```
$ bluetoothctl
[bluetooth]# scan on
[bluetooth]# agent on
[bluetooth]# pair 20:16:05:XX:XX:XX
Attempting to pair with 20:16:05:XX:XX:XX
[CHG] Device 20:16:05:XX:XX:XX Connected: yes
Request PIN code
[agent] Enter PIN code: 1234
```

rfcomm device

We'll create a device file for our Bluetooth module in `/dev`. Then we'll be able to use it just like we used `/dev/ttyUSB0`.

```
$ sudo rfcomm bind 0 20:16:05:XX:XX:XX
```

Because we used `0` as an argument to `bind`, `/dev/rfcomm0` will be the device file assigned to our Bluetooth module.

You can release (destroy) the device file at any time with the following command:

```
$ # Don't actually run this command right now!
$ sudo rfcomm release 0
```

Loopback, again

After pairing your laptop to the Bluetooth module, your OS should have created a device file / COM port for you. On Linux, it should be `/dev/rfcomm*`; on mac, it should be `/dev/cu.*`; and on Windows, it should be a new COM port.

We can now test the Bluetooth module with minicom/PuTTY. Because this module doesn't have LED indicators for the transmission and reception events like the serial module did, we'll test the module using a loopback connection:

F3 <-> Bluetooth connection (loopback)

Just connect the module's TXD pin to its RXD pin using a F/F wire.

Now, connect to the device using `minicom`/`PuTTY`:

```
$ minicom -D /dev/rfcomm0
```

Upon connecting, the blinking pattern of the Bluetooth module should change to: long pause then blink twice quickly.

Typing inside minicom/PuTTY terminal should echo back what you type.

Serial over Bluetooth

Now that we verify that the Bluetooth module works with minicom/PuTTY, let's connect it to the microcontroller:

F3 <-> Bluetooth connection

Recommended steps to wire this up:

- Close OpenOCD and `itmdump`.
- Disconnect the F3 from your laptop.
- Connect F3's GND pin to the module's GND pin using a female to female (F/F) wire (preferably, a black one).
- Connect F3's 5V pin to the module's VCC pin using a F/F wire (preferably, a red one).
- Connect the PA9 (TX) pin on the back of the F3 to the Bluetooth's RXD pin using a F/F wire.
- Connect the PA10 (RX) pin on the back of the F3 to the Bluetooth's TXD pin using a F/F wire.
- Now connect the F3 and your laptop using an USB cable.
- Re-launch OpenOCD and `itmdump`.

And that's it! You should be able to run all the programs you wrote in section 11 without modification! Just make sure you open the right serial device / COM port.

NOTE If you are having trouble communicating with the bluetooth device, you may need to initialize USART1 with a lower baud rate. Lowering it from 115,200 bps to 9,600 bps might help, as described here

I2C

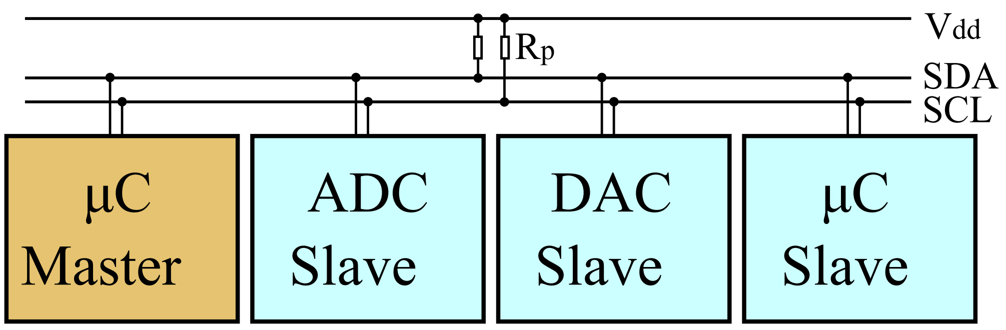
We just saw the serial communication protocol. It's a widely used protocol because it's very simple and this simplicity makes it easy to implement on top of other protocols like Bluetooth and USB.

However, it's simplicity is also a downside. More elaborated data exchanges, like reading a digital sensor, would require the sensor vendor to come up with another protocol on top of it.

(Un)Luckily for us, there are *plenty* of other communication protocols in the embedded space. Some of them are widely used in digital sensors.

The F3 board we are using has three motion sensors in it: an accelerometer, a magnetometer and gyroscope. The accelerometer and magnetometer are packaged in a single component and can be accessed via an I2C bus.

I2C stands for Inter-Integrated Circuit and is a *synchronous serial* communication protocol. It uses two lines to exchange data: a data line (SDA) and a clock line (SCL). Because a clock line is used to synchronize the communication, this is a *synchronous* protocol.



This protocol uses a *master slave* model where the master is the device that *starts* and drives the communication with a slave device. Several devices, both masters and slaves, can be connected to the same bus at the same time. A master device can communicate with a specific slave device by first broadcasting its *address* to the bus. This address can be 7 bits or 10 bits long. Once a master has *started* a communication with a slave, no other device can make use of the bus until the master *stops* the communication.

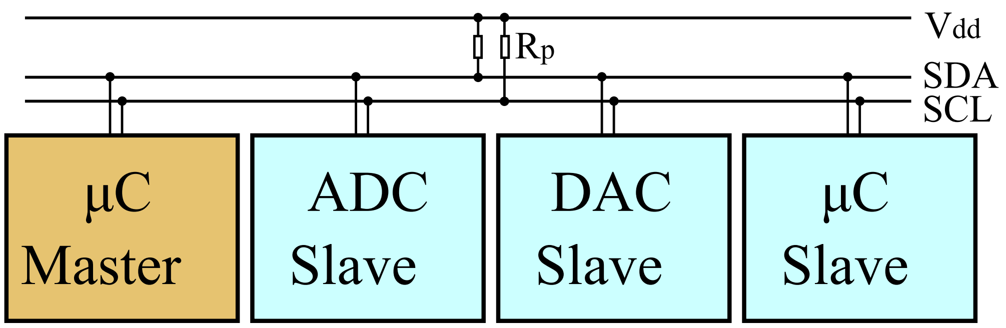
The clock line determines how fast data can be exchanged and it usually operates at a frequency of 100 KHz (standard mode) or 400 KHz (fast mode).

General protocol

The I2C protocol is more elaborated than the serial communication protocol because it has to support communication between several devices. Let's see how it works using examples:

Master -> Slave

If the master wants to send data to the slave:



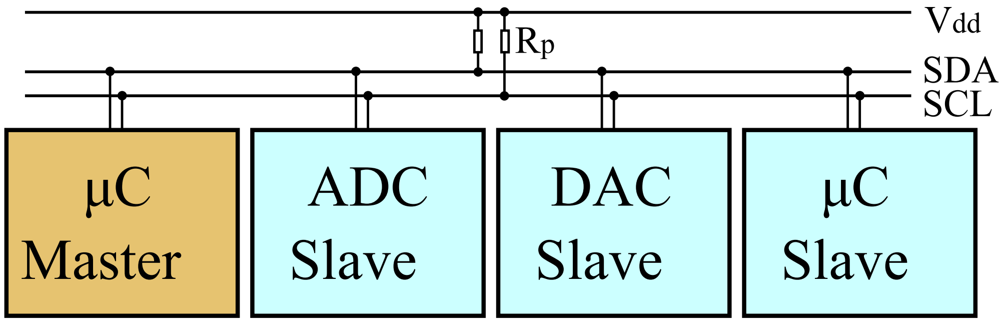
1. Master: Broadcast START
2. M: Broadcast slave address (7 bits) + the R/W (8th) bit set to WRITE
3. Slave: Responds ACK (ACKnowledgement)
4. M: Send one byte

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))

NOTE The slave address could have been 10 bits instead of 7 bits long. Nothing else would have changed.

Master <- Slave

If the master wants to read data from the slave:



1. M: Broadcast START
2. M: Broadcast slave address (7 bits) + the R/W (8th) bit set to READ
3. S: Responds with ACK
4. S: Send byte
5. M: Responds with ACK
6. Repeat steps 4 and 5 zero or more times
7. M: Broadcast STOP OR (broadcast RESTART and go back to (2))

NOTE The slave address could have been 10 bits instead of 7 bits long. Nothing else would have changed.

LSM303DLHC

Two of the sensors in the F3, the magnetometer and the accelerometer, are packaged in a single component: the LSM303DLHC integrated circuit. These two 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.

The memory of these sensors is modeled as byte addressable registers. These sensors can be configured too; that's done by writing to their registers. So, in a sense, these sensors are very similar to the peripherals *inside* the microcontroller. The difference is that their registers are not mapped into the microcontrollers' memory. Instead, their registers have to be accessed via the I2C bus.

The main source of information about the LSM303DLHC is its Data Sheet. Read through it to see how one can read the sensors' registers. That part is in:

> Section 5.1.1 I2C Operation - Page 20 - LSM303DLHC Data Sheet

The other part of the documentation relevant to this book is the description of the registers. That part is in:

> Section 7 Register description - Page 25 - LSM303DLHC Data Sheet

Read a single register

Let's put all that theory into practice!

Just like with the USART peripheral, I've taken care of initializing everything before you reach `main` so you'll only have to deal with the following registers:

- `CR2`. Control register 2.
- `ISR`. Interrupt and status register.
- `TXDR`. Transmit data register.
- `RXDR`. Receive data register.

These registers are documented in the following section of the Reference Manual:

> Section 28.7 I2C registers - Page 868 - Reference Manual

We'll be using the `I2C1` peripheral in conjunction with pins `PB6` (`SCL`) and `PB7` (`SDA`).

You won't have to wire anything this time because the sensor is on the board and it's already connected to the microcontroller. However, I would recommend that you disconnect the serial / Bluetooth module from the F3 to make it easier to manipulate. Later on, we'll be moving the board around quite a bit.

Your task is to write a program that reads the contents of the magnetometer's `IRA_REG_M` register. This register is read only and always contains the value `0b01001000`.

The microcontroller will be taking the role of the I2C master and the magnetometer inside the LSM303DLHC will be the I2C slave.

Here's the starter code. You'll have to implement the `TODO`s.

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

#[allow(unused_imports)]
use aux14::{entry, iprint, iprintln, prelude::*};

// Slave address
const MAGNETOMETER: u8 = 0b001_1110;

// Addresses of the magnetometer's registers
const OUT_X_H_M: u8 = 0x03;
const IRA_REG_M: u8 = 0x0A;

#[entry]
fn main() -> ! {
    let (i2c1, _delay, mut itm) = aux14::init();

    // Stage 1: Send the address of the register we want to read to the
    // magnetometer
    {
        // TODO Broadcast START

        // TODO Broadcast the MAGNETOMETER address with the R/W bit set to Write

        // TODO Send the address of the register that we want to read: IRA_REG_M
    }

    // Stage 2: Receive the contents of the register we asked for
    let byte = {
        // TODO Broadcast RESTART

        // TODO Broadcast the MAGNETOMETER address with the R/W bit set to Read

        // TODO Receive the contents of the register

        // TODO Broadcast STOP
        0
    };

    // Expected output: 0x0A - 0b01001000
    iprintln!(&mut itm.stim[0], "0x{:02X} - 0b{:08b}", IRA_REG_M, byte);

    loop {}
}
```

To give you some extra help, these are the exact bitfields you'll be working with:

- `CR2`: `SADD1`, `RD_WRN`, `NBYTES`, `START`, `AUTOEND`
- `ISR`: `TXIS`, `RXNE`, `TC`
- `TXDR`: `TXDATA`
- `RXDR`: `RXDATA`

The solution

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

#[allow(unused_imports)]
use aux14::{entry, iprint, iprintln, prelude::*};

// Slave address
const MAGNETOMETER: u8 = 0b001_1110;

// Addresses of the magnetometer's registers
const OUT_X_H_M: u8 = 0x03;
const IRA_REG_M: u8 = 0x0A;

#[entry]
fn main() -> ! {
    let (i2c1, _delay, mut itm) = aux14::init();

    // Stage 1: Send the address of the register we want to read to the
    // magnetometer
    {
        // Broadcast START
        // Broadcast the MAGNETOMETER address with the R/W bit set to Write
        i2c1.cr2.write(|w| {
            w.start().set_bit();
            w.sadd1().bits(MAGNETOMETER);
            w.rd_wrn().clear_bit();
            w.nbytes().bits(1);
            w.autoend().clear_bit()
        });

        // Wait until we can send more data
        while i2c1.isr.read().txis().bit_is_clear() {}

        // Send the address of the register that we want to read: IRA_REG_M
        i2c1.txdr.write(|w| w.txdata().bits(IRA_REG_M));

        // Wait until the previous byte has been transmitted
        while i2c1.isr.read().tc().bit_is_clear() {}
    }

    // Stage 2: Receive the contents of the register we asked for
    let byte = {
        // Broadcast RESTART
        // Broadcast the MAGNETOMETER address with the R/W bit set to Read
        i2c1.cr2.modify(|_, w| {
            w.start().set_bit();
            w.nbytes().bits(1);
            w.rd_wrn().set_bit();
            w.autoend().set_bit()
        });

        // Wait until we have received the contents of the register
        while i2c1.isr.read().rxne().bit_is_clear() {}

        // Broadcast STOP (automatic because of `AUTOEND = 1`)
        i2c1.rxdr.read().rxdata().bits()
    };

    // Expected output: 0x0A - 0b01001000
    iprintln!(&mut itm.stim[0], "0x{:02X} - 0b{:08b}", IRA_REG_M, byte);

    loop {}
}
```

Read several registers

Reading the `IRA_REG_M` register was a good test of our understanding of the I2C protocol but that register contains uninteresting information.

This time, we'll read the registers of the magnetometer that actually expose the sensor readings. Six contiguous registers are involved and they start with `OUT_X_H_M` at address `0x03`.

We'll modify our previous program to read these six registers. Only a few modifications are needed.

We'll need to change the address we request from the magnetometer from `IRA_REG_M` to `OUT_X_H_M`.

```
    // Send the address of the register that we want to read: OUT_X_H_M
    i2c1.txdr.write(|w| w.txdata().bits(OUT_X_H_M));
```

We'll have to request the slave for six bytes rather than just one.

```
    // Broadcast RESTART
    // Broadcast the MAGNETOMETER address with the R/W bit set to Read
    i2c1.cr2.modify(|_, w| {
        w.start().set_bit();
        w.nbytes().bits(6);
        w.rd_wrn().set_bit();
        w.autoend().set_bit()
    });
```

And fill a buffer rather than read just one byte:

```
    let mut buffer = [0u8; 6];
    for byte in &mut buffer {
        // Wait until we have received the contents of the register
        while i2c1.isr.read().rxne().bit_is_clear() {}

        *byte = i2c1.rxdr.read().rxdata().bits();
    }

    // Broadcast STOP (automatic because of `AUTOEND = 1`)
```

Putting it all together inside a loop alongside a delay to reduce the data throughput:

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

#[allow(unused_imports)]
use aux14::{entry, iprint, iprintln, prelude::*};

// Slave address
const MAGNETOMETER: u8 = 0b001_1110;

// Addresses of the magnetometer's registers
const OUT_X_H_M: u8 = 0x03;
const IRA_REG_M: u8 = 0x0A;

#[entry]
fn main() -> ! {
    let (i2c1, mut delay, mut itm) = aux14::init();

    loop {
        // Broadcast START
        // Broadcast the MAGNETOMETER address with the R/W bit set to Write
        i2c1.cr2.write(|w| {
            w.start().set_bit();
            w.sadd1().bits(MAGNETOMETER);
            w.rd_wrn().clear_bit();
            w.nbytes().bits(1);
            w.autoend().clear_bit()
        });

        // Wait until we can send more data
        while i2c1.isr.read().txis().bit_is_clear() {}

        // Send the address of the register that we want to read: OUT_X_H_M
        i2c1.txdr.write(|w| w.txdata().bits(OUT_X_H_M));

        // Wait until the previous byte has been transmitted
        while i2c1.isr.read().tc().bit_is_clear() {}

        // Broadcast RESTART
        // Broadcast the MAGNETOMETER address with the R/W bit set to Read
        i2c1.cr2.modify(|_, w| {
            w.start().set_bit();
            w.nbytes().bits(6);
            w.rd_wrn().set_bit();
            w.autoend().set_bit()
        });

        let mut buffer = [0u8; 6];
        for byte in &mut buffer {
            // Wait until we have received something
            while i2c1.isr.read().rxne().bit_is_clear() {}

            *byte = i2c1.rxdr.read().rxdata().bits();
        }
        // Broadcast STOP (automatic because of `AUTOEND = 1`)

        iprintln!(&mut itm.stim[0], "{:?}", buffer);
        delay.delay_ms(1_000_u16);
    }
}
```

If you run this, you should printed in the `itmdump`'s console a new array of six bytes every second. The values within the array should change if you move around the board.

```
$ # itmdump terminal
(..)
[0, 45, 255, 251, 0, 193]
[0, 44, 255, 249, 0, 193]
[0, 49, 255, 250, 0, 195]
```

But these bytes don't make much sense like that. Let's turn them into actual readings:

```

        let x_h = u16::from(buffer[0]);
        let x_l = u16::from(buffer[1]);
        let z_h = u16::from(buffer[2]);
        let z_l = u16::from(buffer[3]);
        let y_h = u16::from(buffer[4]);
        let y_l = u16::from(buffer[5]);

        let x = ((x_h << 8) + x_l) as i16;
        let y = ((y_h << 8) + y_l) as i16;
        let z = ((z_h << 8) + z_l) as i16;

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

Now it should look better:

```
$ # `itmdump terminal
(..)
(44, 196, -7)
(45, 195, -6)
(46, 196, -9)
```

This is the Earth's magnetic field decomposed alongside the XYZ axis of the magnetometer.

In the next section, we'll learn how to make sense of these numbers.

LED compass

In this section, 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 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*.

See below, the magnetometer has three axes associated to it.

Magnetometer axes

Only the X and Y axes are shown above. The Z axis is pointing "out" of your screen.

Let's get familiar with the readings of the magnetometer by running the following starter code:

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

#[allow(unused_imports)]
use aux15::{entry, iprint, iprintln, prelude::*};

#[entry]
fn main() -> ! {
    let (_leds, mut lsm303dlhc, mut delay, mut itm) = aux15::init();

    loop {
        iprintln!(&mut itm.stim[0], "{:?}", lsm303dlhc.mag().unwrap());
        delay.delay_ms(1_000_u16);
    }
}
```

This `lsm303dlhc` module provides high level API over the LSM303DLHC. Under the hood it does the same I2C routine that you implemented in the last section but it reports the X, Y and Z values in a `I16x3` struct instead of a tuple.

Locate where north is at your current location. Then rotate the board such that it's aligned "towards north": the North LED (LD3) should be pointing towards north.

Now run the starter code and observe the output. What X, Y and Z values do you see?

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

Now rotate the board 90 degrees while keeping it parallel to the ground. What X, Y and Z values do you see this time? Then rotate it 90 degrees again. What values do you see?

Take 1

What's the simplest way in which we can implement the LED compass? Even if it's not perfect.

For starters, we'd only care about the X and Y components of the magnetic field because when you look at a compass you always hold it in horizontal position thus the compass is in the XY plane.

For example, what LED would you turn on in the following case. EMF stands for Earth's Magnetic Field and green arrow has the direction of the EMF (it points north).

Quadrant I

The `Southeast` LED, right?

What *signs* do the X and Y components of the magnetic field have in that scenario? Both are positive.

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

Quadrants

In the previous example, the magnetic field was in the first quadrant (x and y were positive) and it made sense to turn on the `SouthEast` LED. Similarly, we could turn a different LED if the magnetic field was in a different quadrant.

Let's try that logic. Here's the starter code:

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

#[allow(unused_imports)]
use aux15::{entry, iprint, iprintln, prelude::*, Direction, I16x3};

#[entry]
fn main() -> ! {
    let (mut leds, mut lsm303dlhc, mut delay, _itm) = aux15::init();

    loop {
        let I16x3 { x, y, .. } = lsm303dlhc.mag().unwrap();

        // Look at the signs of the X and Y components to determine in which
        // quadrant the magnetic field is
        let dir = match (x > 0, y > 0) {
            // Quadrant ???
            (true, true) => Direction::Southeast,
            // Quadrant ???
            (false, true) => panic!("TODO"),
            // Quadrant ???
            (false, false) => panic!("TODO"),
            // Quadrant ???
            (true, false) => panic!("TODO"),
        };

        leds.iter_mut().for_each(|led| led.off());
        leds[dir].on();

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

There's a `Direction` enum in the `led` module that has 8 variants named after the cardinal points: `North`, `East`, `Southwest`, etc. Each of these variants represent one of the 8 LEDs in the compass. The `Leds` value can be indexed using the `Direction` `enum`; the result of indexing is the LED that points in that `Direction`.

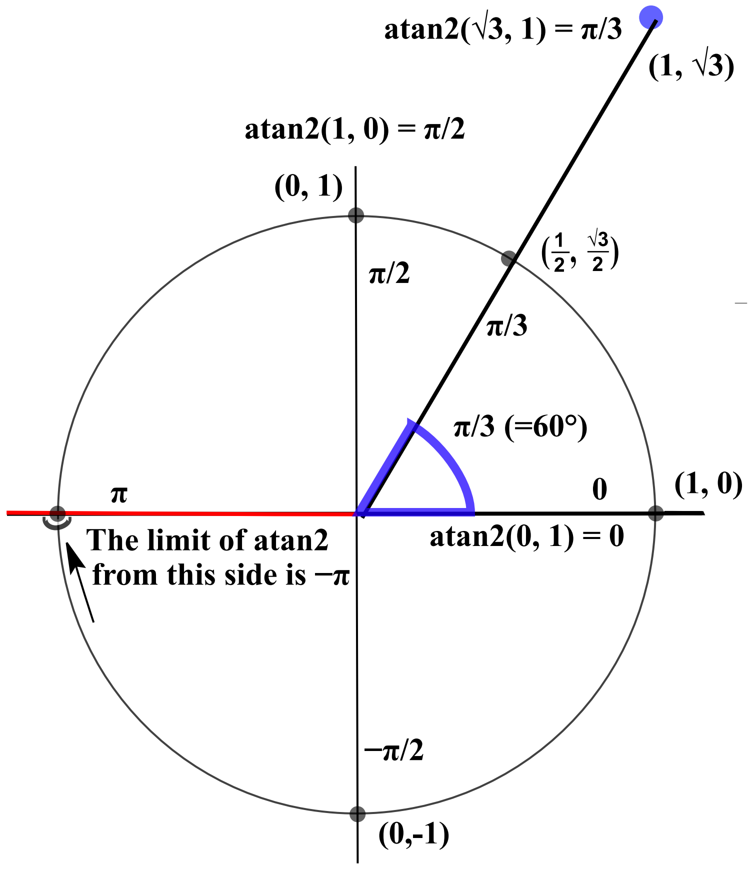
Solution 1

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

#[allow(unused_imports)]
use aux15::{entry, iprint, iprintln, prelude::*, Direction, I16x3};

#[entry]
fn main() -> ! {
    let (mut leds, mut lsm303dlhc, mut delay, _itm) = aux15::init();

    loop {
        let I16x3 { x, y, .. } = lsm303dlhc.mag().unwrap();

        // Look at the signs of the X and Y components to determine in which
        // quadrant the magnetic field is
        let dir = match (x > 0, y > 0) {
            // Quadrant I
            (true, true) => Direction::Southeast,
            // Quadrant II
            (false, true) => Direction::Northeast,
            // Quadrant III
            (false, false) => Direction::Northwest,
            // Quadrant IV
            (true, false) => Direction::Southwest,
        };

        leds.iter_mut().for_each(|led| led.off());
        leds[dir].on();

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

Take 2

This time, 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. The graphic below shows how this angle is measured:



Although not explicitly shown in this graph the X axis points to the right and the Y axis points up.

Here's the starter code. `theta`, in radians, has already been computed. You need to pick which LED to turn on based on the value of `theta`.

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

// You'll find this useful ;-)
use core::f32::consts::PI;

#[allow(unused_imports)]
use aux15::{entry, iprint, iprintln, prelude::*, Direction, I16x3};
// this trait provides the `atan2` method
use m::Float;

#[entry]
fn main() -> ! {
    let (mut leds, mut lsm303dlhc, mut delay, _itm) = aux15::init();

    loop {
        let I16x3 { x, y, .. } = lsm303dlhc.mag().unwrap();

        let _theta = (y as f32).atan2(x as f32); // in radians

        // FIXME pick a direction to point to based on `theta`
        let dir = Direction::Southeast;

        leds.iter_mut().for_each(|led| led.off());
        leds[dir].on();

        delay.delay_ms(100_u8);
    }
}
```

Suggestions/tips:

- A whole circle rotation equals 360 degrees.
- `PI` radians is equivalent to 180 degrees.
- If `theta` was zero, what LED would you turn on?
- If `theta` was, instead, very close to zero, what LED would you turn on?
- If `theta` kept increasing, at what value would you turn on a different LED?

Solution 2

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

// You'll find this useful ;-)
use core::f32::consts::PI;

#[allow(unused_imports)]
use aux15::{entry, iprint, iprintln, prelude::*, Direction, I16x3};
use m::Float;

#[entry]
fn main() -> ! {
    let (mut leds, mut lsm303dlhc, mut delay, _itm) = aux15::init();

    loop {
        let I16x3 { x, y, .. } = lsm303dlhc.mag().unwrap();

        let theta = (y as f32).atan2(x as f32); // in radians

        let dir = if theta < -7. * PI / 8. {
            Direction::North
        } else if theta < -5. * PI / 8. {
            Direction::Northwest
        } else if theta < -3. * PI / 8. {
            Direction::West
        } else if theta < -PI / 8. {
            Direction::Southwest
        } else if theta < PI / 8. {
            Direction::South
        } else if theta < 3. * PI / 8. {
            Direction::Southeast
        } else if theta < 5. * PI / 8. {
            Direction::East
        } else if theta < 7. * PI / 8. {
            Direction::Northeast
        } else {
            Direction::North
        };

        leds.iter_mut().for_each(|led| led.off());
        leds[dir].on();

        delay.delay_ms(100_u8);
    }
}
```

Magnitude

We have been working with the direction of the magnetic field but what's its real magnitude? The number that the `magnetic_field` function reports are unit-less. How can we convert those values to Gauss?

The documentation will answer that question.

Section 2.1 Sensor characteristics - Page 10 - LSM303DLHC Data Sheet

The table in that page shows a *magnetic gain setting* that has different values according to the values of the GN bits. By default, those GN bits are set to `001`. That means that magnetic gain of the X and Y axes is `1100 LSB / Gauss` and the magnetic gain of the Z axis is `980 LSB / Gauss`. LSB stands for Least Significant Bits and the `1100 LSB / Gauss` number indicates that a reading of `1100` is equivalent to `1 Gauss`, a reading of `2200` is equivalent to 2 Gauss and so on.

So, what we need to do is divide the X, Y and Z values that the sensor outputs by its corresponding *gain*. Then, we'll have the X, Y and Z components of the magnetic field in Gauss.

With some extra math we can retrieve the magnitude of the magnetic field from its X, Y and Z components:

```
let magnitude = (x * x + y * y + z * z).sqrt();
```

Putting all this together in a program:

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

#[allow(unused_imports)]
use aux15::{entry, iprint, iprintln, prelude::*, I16x3};
use m::Float;

#[entry]
fn main() -> ! {
    const XY_GAIN: f32 = 1100.; // LSB / G
    const Z_GAIN: f32 = 980.; // LSB / G

    let (_leds, mut lsm303dlhc, mut delay, mut itm) = aux15::init();

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

        let x = f32::from(x) / XY_GAIN;
        let y = f32::from(y) / XY_GAIN;
        let z = f32::from(z) / Z_GAIN;

        let mag = (x * x + y * y + z * z).sqrt();

        iprintln!(&mut itm.stim[0], "{} mG", mag * 1_000.);

        delay.delay_ms(500_u16);
    }
}
```

This program will report the magnitude (strength) of the magnetic field in milligauss (`mG`). 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) so you should see a value in that range or close to that range -- I see a magnitude of around 210 mG.

Some questions:

Without moving the board, what value do you see? Do you always see the same value?

If you rotate the board, does the magnitude change? Should it change?

Calibration

If we rotate the board, the direction of the Earth's magnetic field with respect to the magnetometer should change but its magnitude should not! Yet, the magnetometer indicates that the magnitude of the magnetic field changes as the board rotates.

Why's that the case? Turns out the magnetometer needs to be calibrated to return the correct answer.

The calibration involves quite a bit of math (matrices) so we won't cover it here but this Application Note describes the procedure if you are interested. Instead, what we'll do in this section is *visualize* how off we are.

Let's try this experiment: Let's record the readings of the magnetometer while we slowly rotate the board in different directions. We'll use the `iprintln` macro to format the readings as Tab Separated Values (TSV).

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

#[allow(unused_imports)]
use aux15::{entry, iprint, iprintln, prelude::*, I16x3};

#[entry]
fn main() -> ! {
    let (_leds, mut lsm303dlhc, mut delay, mut itm) = aux15::init();

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

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

        delay.delay_ms(100_u8);
    }
}
```

You should get an output in the console that looks like this:

```
$ # itmdump console
-76     213     -54
-76     213     -54
-76     213     -54
-76     213     -54
-73     213     -55
```

You can pipe that to a file using:

```
$ # Careful! Exit any running other `itmdump` instance that may be running
$ itmdump -F -f itm.txt > emf.txt
```

Rotate the board in many different direction while you log data for a several seconds.

Then import that TSV file into a spreadsheet program (or use the Python script shown below) and plot the first two columns as a scatter plot.

```
#!/usr/bin/python

import csv
import math
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import sys

# apply plot style
sns.set()

x = []
y = []

with open(sys.argv[1], 'r') as f:
    rows = csv.reader(f, delimiter='\t')

    for row in rows:
        # discard rows that are missing data
        if len(row) != 3 or not row[0] or not row[1]:
            continue

        x.append(int(row[0]))
        y.append(int(row[1]))

r = math.ceil(max(max(np.abs(x)), max(np.abs(y))) / 100) * 100

plt.plot(x, y, '.')
plt.xlim(-r, r)
plt.ylim(-r, r)
plt.gca().set_aspect(1)
plt.tight_layout()

plt.savefig('emf.svg')
plt.close
```

Earth's magnetic field

If you rotated the board on a flat horizontal surface, the Z component of the magnetic field should have remained relatively constant and this plot should have been a circumference (not a ellipse) centered at the origin. If you rotated the board in random directions, which was the case of plot above, then you should have gotten a circle made of a bunch of points centered at the origin. Deviations from the circle shape indicate that the magnetometer needs to be calibrated.

Take home message: Don't just trust the reading of a sensor. Verify it's outputting sensible values. If it's not, then calibrate it.

Punch-o-meter

In this section we'll be playing with the accelerometer that's in the board.

What are we building this time? A punch-o-meter! We'll be measuring the power of your jabs. Well, 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.

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. Here's the coordinate system again:

Magnetometer axes

Just like in the previous unit, we'll be using a high level API to directly get the sensor readings in a nicely packaged `struct`.

Gravity is up?

What's the first thing we'll do?

Perform a sanity check!

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 `g`s. Where `1 g` is equal to the acceleration of the gravity, about `9.8` meters per second squared.

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

#[allow(unused_imports)]
use aux16::{entry, iprint, iprintln, prelude::*, I16x3, Sensitivity};

#[entry]
fn main() -> ! {
    let (mut lsm303dlhc, mut delay, _mono_timer, mut itm) = aux16::init();

    // extend sensing range to `[-12g, +12g]`
    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);
    }
}
```

The output of this program with the board sitting still will be something like:

```
$ # 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)
```

Which is weird because the board is not moving yet its acceleration is non-zero. What's going on? This must be related to the gravity, right? Because the acceleration of gravity is `1 g`. But the gravity pulls objects downwards so the acceleration along the Z axis should be negative not positive ...

Did the program get the Z axis backwards? Nope, you can test rotating the board to align the gravity to the X or Y axis but the acceleration measured by the accelerometer is always pointing up.

What happens here is that the accelerometer is measuring the *proper acceleration* of the board not the acceleration *you* are observing. This proper acceleration is the acceleration of the board as seen from a observer that's in free fall. An observer that's in free fall is moving toward the center of the the Earth with an acceleration of `1g`; from its point of view the board is actually moving upwards (away from the center of the Earth) with an acceleration of `1g`. And that's why the proper acceleration is pointing up. This also means that if the board was in free fall, the accelerometer would report a proper acceleration of zero. Please, don't try that at home.

Yes, physics is hard. Let's move on.

The challenge

To keep things simple, we'll measure the acceleration only in the X axis while the board remains horizontal. That way we won't have to deal with subtracting that *fictitious* `1g` we observed before which would be hard because that `1g` could have X Y Z components depending on how the board is oriented.

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

- By default, the app is not "observing" the acceleration of the board.
- When a significant X acceleration is detected (i.e. the acceleration goes above some threshold), the app should start a new measurement.
- During that measurement interval, the app should keep track of the maximum acceleration observed
- After the measurement interval ends, the app must report the maximum acceleration observed. You can report the value using the `iprintln` macro.

Give it a try and let me know how hard you can punch `;-)`.

My solution

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

#[allow(unused_imports)]
use aux16::{entry, iprint, iprintln, prelude::*, I16x3, Sensitivity};
use m::Float;

#[entry]
fn main() -> ! {
    const SENSITIVITY: f32 = 12. / (1 << 14) as f32;
    const THRESHOLD: f32 = 0.5;

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

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

    let measurement_time = mono_timer.frequency().0; // 1 second in ticks
    let mut instant = None;
    let mut max_g = 0.;
    loop {
        let g_x = f32::from(lsm303dlhc.accel().unwrap().x).abs() * SENSITIVITY;

        match instant {
            None => {
                // If acceleration goes above a threshold, we start measuring
                if g_x > THRESHOLD {
                    iprintln!(&mut itm.stim[0], "START!");

                    max_g = g_x;
                    instant = Some(mono_timer.now());
                }
            }
            // Still measuring
            Some(ref instant) if instant.elapsed() < measurement_time => {
                if g_x > max_g {
                    max_g = g_x;
                }
            }
            _ => {
                // Report max value
                iprintln!(&mut itm.stim[0], "Max acceleration: {}g", max_g);

                // Measurement done
                instant = None;

                // Reset
                max_g = 0.;
            }
        }

        delay.delay_ms(50_u8);
    }
}
```

What's left for you to explore

We have barely scratched the surface! There's lots of stuff left for you to explore:

Multitasking

All our programs executed a single task. How could we achieve multitasking in a system with no OS, and thus no threads. There are two main approaches to multitasking: preemptive multitasking and cooperative multitasking.

In preemptive multitasking a task that's currently being executed can, at any point in time, be *preempted* (interrupted) by another task. On preemption, the first task will be suspended and the processor will instead execute the second task. At some point the first task will be resumed. Microcontrollers provide hardware support for preemption in the form of *interrupts*.

In cooperative multitasking a task that's being executed will run until it reaches a *suspension point*. When the processor reaches that suspension point it will stop executing the current task and instead go and execute a different task. At some point the first task will be resumed. The main difference between these two approaches to multitasking is that in cooperative multitasking *yields* execution control at *known* suspension points instead of being forcefully preempted at any point of its execution.

Direct Memory Access (DMA).

This peripheral is a kind of *asynchronous* `memcpy`. So far our programs have been pumping data, byte by byte, into peripherals like UART and I2C. This DMA peripheral can be used to perform bulk transfers of data. Either from RAM to RAM, from a peripheral, like a UART, to RAM or from RAM to a peripheral. You can schedule a DMA transfer, like read 256 bytes from USART1 into this buffer, leave it running in the background and then poll some register to see if it has completed so you can do other stuff while the transfer is ongoing.

Sleeping

All our programs have been continuously polling peripherals to see if there's anything that needs to be done. However, some times there's nothing to be done! At those times, the microcontroller should "sleep".

When the processor sleeps, it stops executing instructions and this saves power. It's almost always a good idea to save power so your microcontroller should be sleeping as much as possible. But, how does it know when it has to wake up to perform some action? "Interrupts" are one of the events that wake up the microcontroller but there are others and the `wfi` and `wfe` are the instructions that make the processor "sleep".

Pulse Width Modulation (PWM)

In a nutshell, PWM is turning on something and then turning it off periodically while keeping some proportion ("duty cycle") between the "on time" and the "off time". When used on a LED with a sufficiently high frequency, this can be used to dim the LED. A low duty cycle, say 10% on time and 90% off time, will make the LED very dim wheres a high duty cycle, say 90% on time and 10% off time, will make the LED much brighter (almost as if it were fully powered).

In general, PWM can be used to control how much *power* is given to some electric device. With proper (power) electronics between a microcontroller and an electrical motor, PWM can be used to control how much power is given to the motor thus it can be used to control its torque and speed. Then you can add an angular position sensor and you got yourself a closed loop controller that can control the position of the motor at different loads.

Digital input

We have used the microcontroller pins as digital outputs, to drive LEDs. But these pins can also be configured as digital inputs. As digital inputs, these pins can read the binary state of switches (on/off) or buttons (pressed/not pressed).

(*spoilers* reading the binary state of switches / buttons is not as straightforward as it sounds ;-)

Sensor fusion

The STM32F3DISCOVERY contains three motion sensors: an accelerometer, a gyroscope and a magnetometer. On their own these measure: (proper) acceleration, angular speed and (the Earth's) magnetic field. But these magnitudes can be "fused" into something more useful: a "robust" measurement of the orientation of the board. Where robust means with less measurement error than a single sensor would be capable of.

This idea of deriving more reliable data from different sources is known as sensor fusion.

Analog-to-Digital Converters (ADC)

There are a lots of digital sensors out there. You can use a protocol like I2C and SPI to read them. But analog sensors also exist! These sensors just output a voltage level that's proportional to the magnitude they are sensing.

The ADC peripheral can be use to convert that "analog" voltage level, say `1.25` Volts,into a "digital" number, say in the `[0, 65535]` range, that the processor can use in its calculations.

Digital-to-Analog Converters (DAC)

As you might expect a DAC is exactly the opposite of ADC. You can write some digital value into a register to produce a voltage in the `[0, 3.3V]` range (assuming a `3.3V` power supply) on some "analog" pin. When this analog pin is connected to some appropriate electronics and the register is written to at some constant, fast rate (frequency) with the right values you can produce sounds or even music!

Real Time Clock (RTC)

This peripheral can be used to track time in "human format". Seconds, minutes, hours, days, months and years. This peripheral handles the translation from "ticks" to these human friendly units of time. It even handles leap years and Daylight Save Time for you!

Other communication protocols

SPI, I2S, SMBUS, CAN, IrDA, Ethernet, USB, Bluetooth, etc.

Different applications use different communication protocols. User facing applications usually have an USB connector because USB is an ubiquitous protocol in PCs and smartphones. Whereas inside cars you'll find plenty of CAN "buses". Some digital sensors use SPI, others use I2C and others, SMBUS.

So where to next? There are several options:

- You could check out the examples in the `f3` board support crate. All those examples work for the STM32F3DISCOVERY board you have.
- You could try out this motion sensors demo. Details about the implementation and source code are available in this blog post.
- You could check out Real Time for The Masses. A very efficient preemptive multitasking framework that supports task prioritization and dead lock free execution.

- You could try running Rust on a different development board. The easiest way to get started is to use the `cortex-m-quickstart` Cargo project template.

- You could check out this blog post which describes how Rust type system can prevent bugs in I/O configuration.

- You could check out my blog for miscellaneous topics about embedded development with Rust.

- You could check out the `embedded-hal` project which aims to build abstractions (traits) for all the embedded I/O functionality commonly found on microcontrollers.

- You could join the Weekly driver initiative and help us write generic drivers on top of the `embedded-hal` traits and that work for all sorts of platforms (ARM Cortex-M, AVR, MSP430, RISCV, etc.)

General troubleshooting

OpenOCD problems

can't connect to OpenOCD - "Error: open failed"

Symptoms

Upon trying to establish a *new connection* with the device you get an error that looks like this:

```
$ openocd -f (..)
(..)
Error: open failed
in procedure 'init'
in procedure 'ocd_bouncer'
```

Cause + Fix

- All: The device is not (properly) connected. Check the USB connection using `lsusb` or the Device Manager.
- Linux: You may not have enough permission to open the device. Try again with `sudo`. If that works, you can use these instructions to make OpenOCD work without root privilege.
- Windows: You are probably missing the ST-LINK USB driver. Installation instructions here.

can't connect to OpenOCD - "Polling again in X00ms"

Symptoms

Upon trying to establish a *new connection* with the device you get an error that looks like this:

```
$ openocd -f (..)
(..)
Error: jtag status contains invalid mode value - communication failure
Polling target stm32f3x.cpu failed, trying to reexamine
Examination failed, GDB will be halted. Polling again in 100ms
Info : Previous state query failed, trying to reconnect
Error: jtag status contains invalid mode value - communication failure
Polling target stm32f3x.cpu failed, trying to reexamine
Examination failed, GDB will be halted. Polling again in 300ms
Info : Previous state query failed, trying to reconnect
```

Cause

The microcontroller may have get stuck in some tight infinite loop or it may be continuously raising an exception, e.g. the exception handler is raising an exception.

Fix

- Close OpenOCD, if running
- Press and hold the reset (black) button
- Launch the OpenOCD command
- Now, release the reset button

OpenOCD connection lost - "Polling again in X00ms"

Symptoms

A *running* OpenOCD session suddenly errors with:

```
# openocd -f (..)
Error: jtag status contains invalid mode value - communication failure
Polling target stm32f3x.cpu failed, trying to reexamine
Examination failed, GDB will be halted. Polling again in 100ms
Info : Previous state query failed, trying to reconnect
Error: jtag status contains invalid mode value - communication failure
Polling target stm32f3x.cpu failed, trying to reexamine
Examination failed, GDB will be halted. Polling again in 300ms
Info : Previous state query failed, trying to reconnect
```

Cause

The USB connection was lost.

Fix

- Close OpenOCD
- Disconnect and re-connect the USB cable.
- Re-launch OpenOCD

Can't flash the device - "Ignoring packet error, continuing..."

Symptoms

While flashing the device, you get:

```
$ arm-none-eabi-gdb $file
Start address 0x8000194, load size 31588
Transfer rate: 22 KB/sec, 5264 bytes/write.
Ignoring packet error, continuing...
Ignoring packet error, continuing...
```

Cause

Closed `itmdump` while a program that "printed" to the ITM was running. The current GDB session will appear to work normally, just without ITM output but the next GDB session will error with the message that was shown in the previous section.

Or, `itmdump` was called **after** the `monitor tpiu` was issued thus making `itmdump` delete the file / named-pipe that OpenOCD was writing to.

Fix

- Close/kill GDB, OpenOCD and `itmdump`
- Remove the file / named-pipe that `itmdump` was using (for example, `itm.txt`).
- Launch OpenOCD
- Then, launch `itmdump`
- Then, launch the GDB session that executes the `monitor tpiu` command.

Cargo problems

"can't find crate for `core`"

Symptoms

```
   Compiling volatile-register v0.1.2
   Compiling rlibc v1.0.0
   Compiling r0 v0.1.0
error[E0463]: can't find crate for `core`

error: aborting due to previous error

error[E0463]: can't find crate for `core`

error: aborting due to previous error

error[E0463]: can't find crate for `core`

error: aborting due to previous error

Build failed, waiting for other jobs to finish...
Build failed, waiting for other jobs to finish...
error: Could not compile `r0`.

To learn more, run the command again with --verbose.
```

Cause

You are using a toolchain older than `nightly-2018-04-08` and forgot to call `rustup target add thumbv7em-none-eabihf`.

Fix

Update your nightly and install the `thumbv7em-none-eabihf` target.

```
$ rustup update nightly

$ rustup target add thumbv7em-none-eabihf
```