micro::bit v2 Embedded Discovery Book

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 details such as linker scripts and the boot process. For example, we'll use existing tools to help get your code onto your board, but not go into detail about how those tools work.

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

Background

What's a microcontroller?

A microcontroller is a *system* on a chip. Whereas your computer is made up of several discrete components: a processor, RAM, storage, an Ethernet port, etc.; a microcontroller has all those types of components built into a single "chip" or package. This makes it possible to build systems with fewer parts.

What can you do with a microcontroller?

Lots of things! Microcontrollers are the central part of what are known as "*embedded* systems". Embedded systems are everywhere, but you don't usually notice them. They control the machines that wash your clothes, print your documents, and cook your food. Embedded systems keep the buildings that you live and work in at a comfortable temperature, and control the components that make the vehicles you travel in stop and go.

Most embedded systems 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.

Embedded systems are often used to *control* a physical process. To make this possible, they have one or more devices to tell them about the state of the world ("sensors"), and one or more devices which allow them to change things ("actuators"). For example, a building climate control system might have:

- Sensors which measure temperature and humidity in various locations.
- Actuators which control the speed of fans.
- Actuators which cause heat to be added or removed from the building.

When should I use a microcontroller?

Many of the embedded systems listed above could be implemented with a computer running Linux (for example a "Raspberry Pi"). Why use a microcontroller instead? Sounds like it might be harder to develop a program.

Some reasons might include:

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

Power consumption. Most microcontrollers consume a fraction of the power of a full blown processor. For applications which run on batteries, that makes a huge difference.

Responsiveness. To accomplish their purpose, some embedded systems must always react within a limited time interval (e.g. the "anti-lock" braking system of a car). If the system misses this type of *deadline*, a catastrophic failure might occur. Such a deadline is called a "hard real time" requirement. An embedded system which is bound by such a deadline is referred to as a "hard real-time system". A general purpose computer and OS usually has many software components which share the computer's processing resources. This makes it harder to guarantee execution of a program within tight time constraints.

Reliability. In systems with fewer components (both hardware and software), there is less to go wrong!

When should I *not* use a microcontroller?

Microcontrollers are often not great at heavy computational work. To keep their cost and power consumption low, microcontrollers have limited computational resources available to them.

Microcontrollers can typically execute fewer instructions per second than "big" processors. The slowest parts might run at "only" a few million instructions per second. In addition, the amount of work per instruction is typically lower. Microcontroller parts are typically "32 bit", but "16 bit" parts are not uncommon: this may mean more instructions to work with typical Rust datatypes. Most microcontrollers have no or little "cache", meaning instructions can run only as fast as main memory can be accessed.

Some microcontrollers don't have hardware support for floating point operations. On those devices, performing a simple addition of single precision numbers can take hundreds of CPU cycles.

Finally, microcontrollers typically come with limited memory. Memory sizes may be as small as 16KB for program instructions and 4KB for data, making programming for these systems quite challenging. While the internal memory size per unit cost and power consumption is constantly increasing, the processor we will work with still has "only" 512KB for program instructions and 256KB for data — far less than that of a "real computer".

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 more mature. Off-the-shelf solutions 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 (as of this writing) so you would have to either create one yourself or try one of the ones that are in development. You can find a list of those in the Awesome Embedded Rust repository.

Hardware/knowledge requirements

The primary knowledge requirement to read this book is to know *some* Rust. It's hard for me to quantify *some*. Being familiar with the basics of generics and traits is quite helpful. You do need to know how to *use* closures. You also need to be familiar with the idioms of the current Rust edition.

Also, to follow this material you'll need:

- A Micro:Bit v2 (MB2) board.

  You can purchase this board from many suppliers, including Amazon and Ali Baba. You can get a list of suppliers directly from the BBC, the manufacturers of MB2.

  There are several versions of the `V2` board available. While the material here was written for `V2.00`, things should work fine with with any `V2` board.

- A micro-B USB cable (nothing special — you probably have many of these). This is required to power the micro:bit board when not on

battery, and to communicate with it. Make sure that the cable supports data transfer, as some cables only support charging devices.

NOTE Some micro:bit kits ship with such cables. USB cables used with other mobile devices should work, if they are micro-B and have the capability to transmit data.

The official `micro:bit Go` kit provides both the USB cable and a nifty battery pack for powering the MB2 without USB.

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 a high level crate already exists for your development board somewhere. You probably want at least a HAL crate, like `nrf52-hal` used here. You may prefer a board with a Board Support crate, like [`microbit-v2`] used here. If you intend to use a different microcontroller, you can look through Awesome Embedded Rust or just search the web to find supported crates.

With a different development board, this text loses most if not all its beginner friendliness and "easy to follow"-ness, in my opinion: you have been warned.

If you have a different ARM-based development board and you don't consider yourself a total beginner, you might consider 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 from your computer'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. The official MB2 technical documentation is at https://tech.microbit.org. We will reference other technical documentation throughout the book.

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 1.79.0 or a newer toolchain.
- `gdb-multiarch`. This is a debugging tool. The oldest tested version is10.2, but other versions will most likely work as well. If your distribution/platform does not have `gdb-multiarch` available `arm-none-eabi-gdb` will do the trick as well. Furthermore, some normal `gdb` binaries are built with multiarch capabilities as well: you can find further information about this in the debugging chapter of this book.
- `cargo-binutils`. Version 0.3.6 or newer.
- `probe-rs-tools`. Version 0.24.0 or newer.
- `minicom` on Linux and macOS. Tested version: 2.7.1. Other versions will most likely work as well though.
- `PuTTY` on Windows.

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 and version no older than the one shown below:

```
$ rustc -V
rustc 1.79.0 (129f3b996 2024-06-10)
```

`cargo-binutils`

```
$ rustup component add llvm-tools
$ cargo install cargo-binutils --vers '^0.3'
$ cargo size --version
cargo-size 0.3.6
```

`probe-rs-tools`

NOTE If you already have old versions of `probe-run`, `probe-rs` or `cargo-embed` installed on your system, remove them before starting this step, as they could conceivably cause problems for you down the line. In particular, `probe-run` no longer officially exists. Try these as needed:

```
$ cargo uninstall cargo-embed
$ cargo uninstall probe-run
$ cargo uninstall probe-rs
$ cargo uninstall probe-rs-cli
```

In order to install `probe-rs-tools`, first install its prerequisites (note: these instructions are part of the more general `probe-rs` embedded debugging toolkit). Then install `probe-rs-tools` with Cargo.

```
$ cargo install --locked probe-rs-tools
```

NOTE This may fail due to frequent changes in `probe-rs`. If so, go to https://probe.rs and follow the current installation instructions there.

Installing `probe-rs-tools` will install several useful tools, including `probe-rs` and `cargo-embed` (which is normally run as a Cargo command). Check that things are working before proceeding.

```
$ cargo embed --version
cargo-embed 0.24.0 (git commit: crates.io)
```

This repository

This book also contains some small Rust codebases used in various chapters: the easiest way to use these is to download the book's source code. You can do this in one of the following ways:

- Visit the repository, click the green "Code" button and then the "Download Zip" one.
- Clone it using `git` (if you know `git` you presumably already have it installed) from the same repository as linked in the Zip approach.

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.

Ubuntu 20.04 or newer / Debian 10 or newer

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

```
$ sudo apt install gdb-multiarch minicom libunwind-dev
```

Fedora 32 or newer

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

```
$ sudo dnf install gdb minicom libunwind-devel
```

Arch Linux

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

```
$ sudo pacman -S arm-none-eabi-gdb minicom libunwind
```

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/XXX.tar.bz2
```

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/XXX/bin
```

udev rules

These rules let you use USB devices like the micro:bit without root privilege, i.e. `sudo`.

Create this file in `/etc/udev/rules.d` with the content shown below.

```
$ cat /etc/udev/rules.d/69-microbit.rules

# CMSIS-DAP for microbit
ACTION!="add|change", GOTO="microbit_rules_end"
SUBSYSTEM=="usb",                       ATTR{idVendor}=="0d28",
ATTR{idProduct}=="0204", TAG+="uaccess"
LABEL="microbit_rules_end"
```

Then reload the udev rules with:

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

If you had any board plugged to your computer, unplug them and then plug them in again, or run the following command.

```
$ sudo udevadm trigger
```

Verify permissions

Connect the micro:bit to your computer using a USB cable.

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

```
$ lsusb | grep -i "NXP ARM mbed"
Bus 001 Device 065: ID 0d28:0204 NXP ARM mbed
$ # ^^^        ^^^
```

In my case, the micro:bit got connected to the bus #1 and got enumerated as the device #65. This means the file `/dev/bus/usb/001/065` *is* the micro:bit. Let's check the file permissions:

```
$ ls -l /dev/bus/usb/001/065
crw-rw-r--+  1  nobody  nobody  189,  64  Sep    5  14:27
/dev/bus/usb/001/065
```

The permissions should be `crw-rw-r--+`, note the `+` at the end, then see your access rights by running the following command.

```
$ getfacl /dev/bus/usb/001/065
getfacl: Removing leadin '/' from absolute path names
# file: dev/bus/usb/001/065
# owner: nobody
# group: nobody
user::rw-
user:<YOUR-USER-NAME>:rw-
group::rw-
mask::rw-
other::r-
```

You should see your username in the list above with the `rw-` permissions, if not ... then check your udev rules and try re-loading them with:

```
$ sudo udevadm control --reload
$ sudo udevadm trigger
```

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) (..)
```

PuTTY

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

Now, go to the next section.

macOS

All the tools can be installed using Homebrew:

```
$ # ARM GCC debugger
$ brew install arm-none-eabi-gdb

$ # Minicom
$ brew install minicom

$ # lsusb lists USB ports
$ brew install lsusb
```

That's all! Go to the next section.

Verify the installation

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

Verifying cargo-embed

First, connect the micro:bit to your Computer using a USB cable.

At least an orange LED right next to the USB port of the micro:bit should light up. Furthermore, if you have never flashed another program on to your micro:bit, the default program the micro:bit ships with should start blinking the red LEDs on its back: you can ignore them, or you can play with the demo app.

Now let's see if probe-rs, and by extensions cargo-embed can see your micro:bit. You can do this by running the following command:

```
$ probe-rs list
The following debug probes were found:
[0]:         BBC         micro:bit         CMSIS-DAP         --
0d28:0204:990636020005282030f57fa14252d446000000006e052820
(CMSIS-DAP)
```

Or if you want more information about the micro:bits debug capabilities then you can run:

```
$ probe-rs info
Probing target via JTAG

Error identifying target using protocol JTAG: The probe does
not support the JTAG protocol.

Probing target via SWD

ARM Chip with debug port Default:
Debug Port: DPv1, DP Designer: ARM Ltd
├── 0 MemoryAP
│   └── ROM Table (Class 1), Designer: Nordic VLSI ASA
│       ├── Cortex-M4 SCS   (Generic IP component)
│       │   └── CPUID
│       │       ├── IMPLEMENTER: ARM Ltd
│       │       ├── VARIANT: 0
│       │       ├── PARTNO: Cortex-M4
```

```
│       │       └── REVISION: 1
│       ├── Cortex-M3 DWT   (Generic IP component)
│       ├── Cortex-M3 FBP   (Generic IP component)
│       ├── Cortex-M3 ITM   (Generic IP component)
│       ├── Cortex-M4 TPIU  (Coresight Component)
│       └── Cortex-M4 ETM   (Coresight Component)
└── 1 Unknown AP (Designer: Nordic VLSI ASA, Class: Undefined,
Type: 0x0, Variant: 0x0, Revision: 0x0)


Debugging RISC-V targets over SWD is not supported. For these
targets, JTAG is the only supported protocol. RISC-V specific
information cannot be printed.
Debugging Xtensa targets over SWD is not supported. For these
targets, JTAG is the only supported protocol. Xtensa specific
information cannot be printed.
```

Next, make sure you are in `src/03-setup` of this book's source code. Then run these commands:

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

If everything works correctly cargo-embed should first compile the small example program in this directory, then flash it and finally open a nice text based user interface that prints Hello World.

(If it does not, check out general troubleshooting instructions.)

This output is coming from the small Rust program you just flashed on to your micro:bit. Everything is working properly and you can continue with the next chapters!

Getting the most out of your IDE

All code in this book assumes that you use a simple terminal to build your code, run it, and interact with it. It also makes no assumption about your text editor.

However, you may have your favourite IDEs, providing you auto-complete, type annotation, your preferred shortcuts and much more. This section explains how to get the most out of your IDE using the code obtained from this book's repo.

IDE configuration

Below, we explain how to configure your IDE to get the most out of this book. If your IDE is not listed below, please improve this book by adding a section, so that the next reader can get the best experience out of it.

How to build with IntelliJ

When editing the IntelliJ build configuration, here are a few non-default values:

- You should edit the command. When this book tells you to run `cargo embed FLAGS`, You'll need to replace the default value `run` by the command `embed FLAGS`,
- You should enable "Emulate terminal in output console". Otherwise, your program will fail to print text to a terminal
- You should ensure that the working directory is `microbit/src/N-name`, with `N-name` being the directory of the chapter you are reading. You can not run from the `src` directory since it contains no cargo file.

Meet your hardware

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

micro:bit

Here are some of the many components on the board:

- A microcontroller.
- A number of LEDs, most notably the LED matrix on the back
- Two user buttons as well as a reset button (the one next to the USB port).
- One USB port.
- A sensor that is both a magnetometer and an accelerometer

Of these components, the most important is the microcontroller (sometimes shortened to "MCU" for "microcontroller unit"), which is the bigger of the two black squares sitting on the side of the board with the USB port. The MCU is what runs your code. You might sometimes read about "programming a board", when in reality what we are doing is programming the MCU that is installed on the board.

If you happen to be interested in a more detailed description of the board you can checkout the micro:bit website.

Since the MCU is so important, let's take a closer look at the one sitting on our board.

Nordic nRF52833 (the "nRF52", micro:bit v2)

Our MCU has 73 tiny metal **pins** sitting right underneath it (it's a so called aQFN73 chip). These pins are connected to **traces**, the little "roads" that act as the wires connecting components together on the board. The MCU can dynamically alter the electrical properties of the pins. This works similarly to a light switch, altering how electrical current flows through a circuit. By enabling or disabling electrical current to flow through a specific pin, an LED attached to that pin (via the traces) can be turned on and off.

Each manufacturer uses a different part numbering scheme, but many will allow you to determine information about a component simply by looking at the part number. Looking at our MCU's part number we find `N52833 QIAAA0 2024AL`: you probably cannot see it with your bare eye, but it is on the chip. (If you have a later revision of MB2, your number may vary somewhat. This not an issue. The `N52833` part should be there, though.) The `N` at the front hints to us that this is a part manufactured by Nordic Semiconductor. Looking up the part number on their website we quickly find the product page. There we learn that our chip's main marketing point is that it is a "Bluetooth Low Energy and 2.4 GHz SoC" (SoC being short for "System on a Chip"), which explains the RF in the product name since RF is short for radio frequency. If we search through the documentation of the chip linked on the product page for a bit we find the product specification which contains chapter 10 "Ordering Information" dedicated to explaining the weird chip naming. Here we learn that:

- The `N52` is the MCU's series, indicating that there are other `nRF52` MCUs
- The `833` is the part code
- The `QI` is the package code, short for `aQFN73`
- The `AA` is the variant code, indicating how much RAM and flash memory the MCU has, in our case 512 kilobyte flash and 128 kilobyte

RAM

- The `A0` is the build code, indicating the hardware version (`A`) as well as the product configuration (`0`)
- The `2024AL` is a tracking code, hence it might differ on your chip

The product specification does of course contain a lot more useful information about the chip: for example, that the chip is an ARM® Cortex™-M4 32-bit processor.

Arm? Cortex-M4?

If our chip is manufactured by Nordic, then who is Arm? And if our chip is the nRF52833, what is the Cortex-M4?

You might be surprised to hear that while "Arm-based" chips are quite popular, the company behind the "Arm" trademark (Arm Holdings) doesn't actually manufacture chips for purchase. Instead, their primary business model is to just *design* parts of chips. They will then license those designs to manufacturers, who will in turn implement the designs (perhaps with some of their own tweaks) in the form of physical hardware that can then be sold. Arm's strategy here is different from companies like Intel, which both designs *and* manufactures their chips.

Arm licenses a bunch of different designs. Their "Cortex-M" family of designs are mainly used as the core in microcontrollers. For example, the Cortex-M4 (the core our chip is based on) is designed for low cost and low power usage. The Cortex-M7 is higher cost, but with more features and performance.

Luckily, you don't need to know too much about different types of processors or Cortex designs for the sake of this book. However, you are hopefully now a bit more knowledgeable about the terminology of your device. While you are working specifically with an nRF52833, you might find yourself reading documentation and using tools for Cortex-M-based chips, as the nRF52833 is based on a Cortex-M design.

Rust Embedded terminology

Before we dive into programming the micro:bit let's have a quick look at the libraries and terminology that will be important for all the future chapters.

Abstraction layers

For any fully supported microcontroller/board with a microcontroller, you will usually hear the following terms being used for their levels of abstraction:

Peripheral Access Crate (PAC)

The job of the PAC is to provide a safe (ish) direct interface to the peripherals of the chip, allowing you to configure every last bit however you want (of course also in wrong ways). Usually you only ever have to deal with the PAC if either the layers that are higher up don't fulfill your needs or when you are developing higher-level code for them. Unsurprisingly, the PAC we are (mostly implicitly) going to use is for the nRF52.

Hardware Abstraction Layer (HAL)

The job of the HAL is to build up on top of the chip's PAC and provide an abstraction that is actually usable for someone who does not know about all the special behaviour of this chip. Usually a HAL abstracts whole peripherals away into single structs that can, for example, be used to send data around via the peripheral. We are going to use the nRF52-hal.

Board Support Crate (BSP)

(In non-Rust situations this is usually called the Board Support Package, hence the acronym.)

The job of the BSP is to abstract a whole board (such as the micro:bit) away at once. That means it has to provide abstractions to use both the microcontroller as well as the sensors, LEDs etc. that might be present on the board. Quite often (especially with custom-made boards) no pre-built BSP will be available. Instead you will be working with a HAL for the chip and build the drivers for the sensors either yourself or search for them on `crates.io`. Luckily for us though, the micro:bit does have a BSP, so we are going to use that on top of our HAL as well.

Unifying the layers

Next we are going to have a look at a very central piece of software in the Rust Embedded world: `embedded-hal`. As its name suggests it relates to the 2nd level of abstraction we got to know: the HALs. The idea behind `embedded-hal` is to provide a set of traits that describe behaviour which is usually shared across all implementations of a specific peripheral in all the HALs. For example one would always expect to have functions that are capable of turning the power on a pin either on or off: to switch an LED on and off on the board or whatever.

`embedded-hal` allows us to write a driver for some piece of hardware, for example a temperature sensor, that can be used on any chip for which an implementation of the `embedded-hal` traits exists. This is accomplished by writing the driver in such a way that it only relies on the `embedded-hal` traits. Drivers that are written in such a way are called *platform-agnostic*. Luckily for us, the drivers we will be getting from `crates.io` are almost all platform agnostic.

Further reading

If you want to learn more about these levels of abstraction, Franz Skarman (a.k.a. TheZoq2) held a talk about this topic during Oxidize 2020: An Overview of the Embedded Rust Ecosystem.

Meet your software

In this chapter we will learn how to build, run and debug some *very* simple programs. The goal here is not to get into the details of MB2 Rust programming (yet), but to just familiarize yourself with the mechanics of the process.

First, a quick note about the conventions used in the rest of this book. We expect you to get a copy of the whole book with

```
git clone http://github.com/rust-embedded/discovery-mb2
```

The book's "source code" is in `discovery-mb2/mdbook/src`. You should go there in your copy and look around a bit. Each chapter directory has both the source Markdown text *and* the complete source for all the programs in that chapter. When we refer to some path like `src/main.rs`, we mean that place starting from the chapter you are working in. For example, your `discovery-mb2` has a file called `mdbook/src/05-meet-your-software/examples/init.rs`. We will refer to that file as just `examples/init.rs` in this chapter.

There are two basic kinds of Rust code: "binary" executable programs, and "library" code. The library code won't play a huge role in this book. Binary program source code can live in one of several places:

- A program in `src/main.rs` will be automatically compiled and run by `cargo embed` or `cargo run`. No special flags are needed.
- A program in `examples/foo.rs` can be compiled and run by `cargo embed --example foo` or `cargo run --example foo`.
- A program in `src/bin/bar.rs` can be compiled and run by `cargo embed --bin bar` or `cargo run --bin bar`.

This is confusing, but it's a standard convention of Cargo.

Now let's move on and work with all this.

Build it

The first step is to build our "binary" crate. Because the microcontroller has a different architecture than your computer 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.

As we already know the microcontroller on the micro:bit v2 has a Cortex-M4F processor in it. `rustc` knows how to cross-compile to the Cortex-M architecture and provides several different targets that cover the different processors 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
- `thumbv8m.main-none-eabi`, for the Cortex-M33 and Cortex-M35P processors
- `thumbv8m.main-none-eabihf`, for the Cortex-M33**F** and Cortex-M35P**F** processors

"Thumb" here refers to a version of the Arm instruction set that has smaller instructions for reduced code size (it's a pun, see). The `hf`/`F` parts have hardware floating point acceleration. This will make numeric computations involving fractional ("floating decimal point") computations much faster.

For the micro:bit v2, we'll want the `thumbv7em-none-eabihf` target.

Before cross-compiling you have to download a 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 then update this target (re-installing a new standard library `rust-std` component that contains the `core` library we use) whenever you update your toolchain. Therefore you can skip this step if you have already added the necessary target while verifying your setup.

With the `rust-std` component in place you can now cross compile the program using Cargo. Make sure you are in the `mdbook/src/05-meet-your-software` directory in the Git repo, then build. This initial code is an example, so we compile it as such.

```
$ cargo build --example init
   Compiling semver-parser v0.7.0
   Compiling proc-macro2 v1.0.86
   ...

    Finished dev [unoptimized + debuginfo] target(s) in 33.67s
```

NOTE Be sure to compile this crate *without* optimizations. The provided `Cargo.toml` file and build command above will ensure optimizations are off as long as you *don't* pass `cargo` the `--release` flag.

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. (The command below is equivalent to

```
readelf -h ../../../target/thumbv7em-none-eabihf/debug/examples/init
```

on systems that have `readelf`.)

```
$ cargo readobj --example init -- --file-headers
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
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:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x117
  Start of program headers:          52 (bytes into file)
  Start of section headers:          793112 (bytes into file)
  Flags:                             0x5000400
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         4
  Size of section headers:           40 (bytes)
  Number of section headers:         21
  Section header string table index: 19
```

If your numbers don't exactly match these, don't worry: a lot of this is quite dependent on the current build environment.

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.

Our 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. Our program has full control over the device.

Flashing the binary itself is quite simple, thanks to `cargo embed`.

Before executing that command though, let's look into what it actually does. If you look at the side of your micro:bit with the USB connector facing upwards, you will notice that there are actually three black squares on there. The biggest one is a speaker. Another is our MCU we already talked about… but what purpose does the remaining one serve? This chip is *another* MCU, an NRF52820 almost as powerful as the NRF52833 we will be programming! This chip has three main purposes:

1. Enable power and reset control of our NRF52833 MCU from the USB connector.
2. Provide a serial to USB bridge for our MCU (we will look into that in a later chapter).
3. Provide an interface for programming and debugging our NRF52833 (this is the relevant purpose for now).

This chip acts as sort of bridge between our computer (to which it is connected via USB) and the MCU (to which it is connected via traces and communicates with using the SWD protocol). This bridge enables us to flash new binaries on to the MCU, inspect a program's state via a debugger and do other useful things.

So lets flash it!

```
$ cargo embed --example init
  (...)
                      Erasing    sectors    ✔    [00:00:00]
[########################################################
```

```
########################################################################
########################]  2.00KiB/ 2.00KiB @  4.21KiB/s (eta
0s )
      Programming      pages                 ✔      [00:00:00]
[########################################################################
########################################################################
########################]  2.00KiB/ 2.00KiB @  2.71KiB/s (eta
0s )
    Finished flashing in 0.608s
```

You will notice that `cargo-embed` does not exit after outputting the last line. This is intended: you should not close `cargo-embed`, since we need it in this state for the next step — debugging it! Furthermore, you will have noticed that `cargo build` and `cargo embed` are actually passed the same flags. This is because `cargo embed` actually executes the build and then flashes the resulting binary on to the chip. This means you can leave out the `cargo build` step in the future if you want to flash your code right away.

Debug it

Let's figure out how to debug our little program. It doesn't really have any interesting bugs yet, but that's the best kind of program to learn debugging on.

How does this even work?

Before we debug our program let's take a moment to quickly understand what is actually happening here. In the previous chapter we already discussed the purpose of the second chip on the board, as well as how it talks to our computer, but how can we actually use it?

The little option `default.gdb.enabled = true` in `Embed.toml` made `cargo embed` open a so-called "GDB stub" after flashing. This is a server that our GDB can connect to and send commands like "set a breakpoint at address X". The server can then decide on its own how to handle this command. In the case of the `cargo embed` GDB stub it will forward the command via USB to the "debugging probe" on the second chip. This chip does the job of talking to the MCU for us.

Let's debug!

`cargo-embed` is running in our current shell. We can open a new shell and go back into our project directory. Once we are there we first have to open the binary in gdb like this:

```
$ gdb ../../../target/thumbv7em-none-eabihf/debug/examples/init
```

> **NOTE**: Depending on which GDB you installed you will have to use a different command to launch it. Check out chapter 3 if you forgot which one it was.

The `../../..` in this command is needed, since each example project is in a "workspace" that contains the entire book. Workspaces have a single shared `target` directory. Check out Workspaces chapter in Rust Book for more.

> **NOTE** If `cargo-embed` prints a lot of warnings here don't worry about it. As of now it does not fully implement the GDB protocol, and thus might not recognize all the commands your GDB is sending to it. As long as GDB does not crash, you are fine.

Next we will have to connect to the GDB stub. It runs on `localhost:1337` by default so in order to connect to it run the following:

```
(gdb) target remote :1337
Remote debugging using :1337
0x00000116 in nrf52833_pac::{{impl}}::fmt (self=0xd472e165,
f=0x3c195ff7) at /home/nix/.cargo/registry/src/github.com-
1ecc6299db9ec823/nrf52833-pac-0.9.0/src/lib.rs:157
157     #[derive(Copy, Clone, Debug)]
```

NOTE: The example in the repository for this chapter may change over time. Line numbers and other source details may thus be different from what is shown here and below.

Next what we want to do is get to the `main` function of our program. We will do this by first setting a breakpoint there and then continuing program execution until we hit the breakpoint:

```
(gdb) break main
Breakpoint 1 at 0x104: file src/05-meet-your-
```

```
software/examples/init.rs, line 9.
Note: automatically using hardware breakpoints for read-only
addresses.
(gdb) continue
Continuing.

Breakpoint 1, init::__cortex_m_rt_main_trampoline () at src/05-
meet-your-software/examples/init.rs:9
9       #[entry]
```

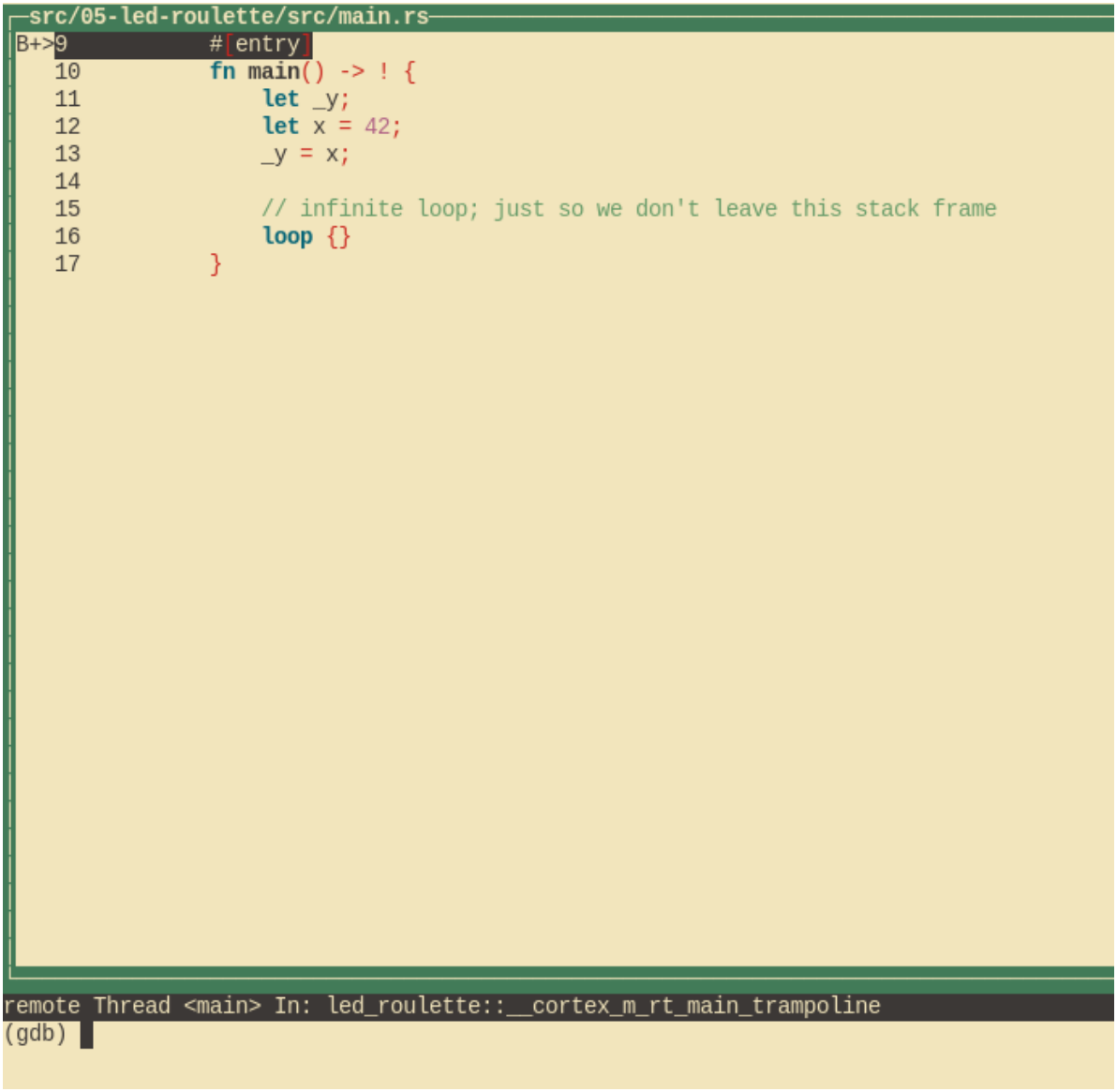
Breakpoints can be used to stop the normal flow of a program. The `continue` command will let the program run freely *until* it reaches a breakpoint. In this case, until it reaches the `main` function because there's a breakpoint there.

Note that GDB output says "Breakpoint 1". Remember that our processor can only use a limited amount of these breakpoints, so it's a good idea to pay attention to these messages. If you happen to run out of breakpoints, you can list all the current ones with `info break` and delete desired ones with `delete <breakpoint-num>`.

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's break command does works for more than just function names: it can also break at certain line numbers. If we want to break in line 13 we can simply do:

```
(gdb) break 13
Breakpoint 2 at 0x110: file src/05-meet-your-
software/examples/init.rs, line 13.
(gdb) continue
Continuing.

Breakpoint 2, init::__cortex_m_rt_main () at src/05-meet-your-
```

```
software/examples/init.rs:13
(gdb)
```

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

```
(gdb) tui disable
```

We are now "on" the `_y = x` statement; that statement hasn't been executed yet. This means that `x` is initialized but `_y` could contain anything. Let's inspect `x` using the `print` command:

```
(gdb) print x
$1 = 42
(gdb) print &x
$2 = (*mut i32) 0x20003fe8
(gdb)
```

As expected, `x` contains the value `42`. 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: `*mut i32`, a pointer to a mutable `i32` value.

If we want to continue the program execution line by line, we can do that using the `next` command. Let's proceed to the `loop {}` statement:

```
(gdb) next
16          loop {}
```

And `_y` should now be initialized.

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

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

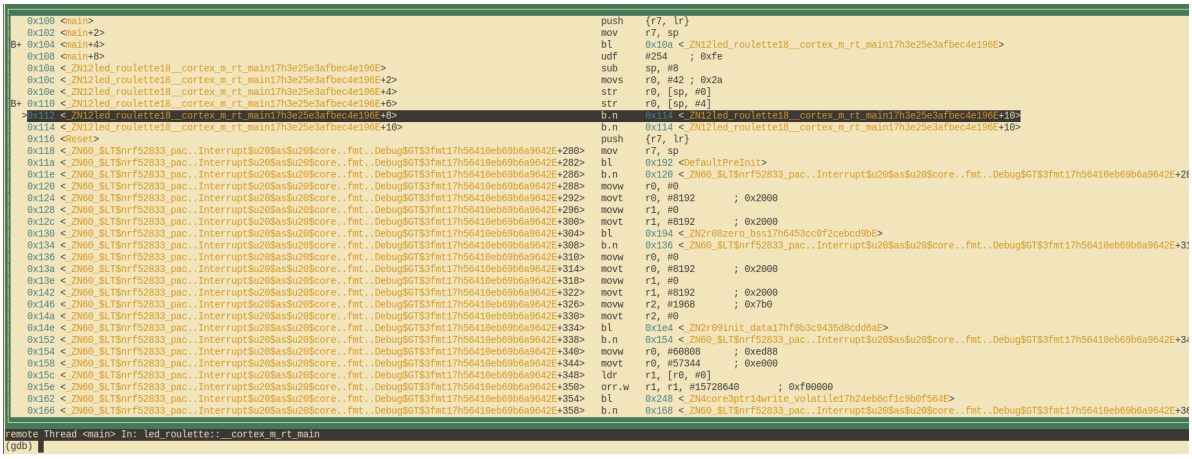
```
(gdb) info locals
x = 42
_y = 42
(gdb)
```

If we use `next` 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 `next` or `continue` command by mistake and GDB got stuck, you can get unstuck by hitting `Ctrl+C`.

```
(gdb) layout asm
```



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
_ZN12init18__cortex_m_rt_main17h3e25e3afbec4e196E:
10      fn main() -> ! {
   0x0000010a <+0>:     sub     sp, #8
   0x0000010c <+2>:     movs    r0, #42 ; 0x2a

11          let _y;
12          let x = 42;
   0x0000010e <+4>:     str     r0, [sp, #0]

13          _y = x;
   0x00000110 <+6>:     str     r0, [sp, #4]

14
15          // infinite loop; just so we don't leave this stack
frame
```

```
16          loop {}
=>   0x00000112   <+8>:                b.n                0x114
<_ZN12init18__cortex_m_rt_main17h3e25e3afbec4e196E+10>
         0x00000114   <+10>:             b.n                0x114
<_ZN12init18__cortex_m_rt_main17h3e25e3afbec4e196E+10>

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 and the line number of the instruction the processor will execute next.

```
(gdb) stepi
16          loop {}
(gdb) stepi
16          loop {}
```

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

```
(gdb) monitor reset
(gdb) c
Continuing.

Breakpoint 1, init::__cortex_m_rt_main_trampoline () at src/05-
meet-your-software/src/main.rs:9
9       #[entry]
(gdb)
```

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

`monitor reset` 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 on 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/meet-your-software, Remote target
Ending remote debugging.
[Inferior 1 (Remote target) detached]
```

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.

If you want to learn more about what GDB can do, check out the section How to use GDB.

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

Light it up

We will finish this chapter by making one of the many LEDs on the MB2 light up. In order to get this task done we will use one of the traits provided by `embedded-hal`, specifically the `OutputPin` trait which allows us to turn a pin on or off.

The micro:bit LEDs

On the back of the micro:bit you can see a 5x5 square of LEDs, usually called an LED matrix. This matrix alignment is used so that instead of having to use 25 separate pins to drive every single one of the LEDs, we can just use 10 (5+5) pins in order to control which column and which row of our matrix lights up.

Right now we will use the `microbit-v2` crate to manipulate the LEDs. In the next chapter we will go in detail through all of the options available.

Actually lighting it up!

The code required to light up an LED in the matrix is actually quite simple but it requires a bit of setup. First take a look at `examples/light-it-up.rs`; then we can go through it step by step.

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

use cortex_m_rt::entry;
use embedded_hal::digital::OutputPin;
use microbit::board::Board;
use panic_halt as _;

#[entry]
fn main() -> ! {
    let mut board = Board::take().unwrap();

    board.display_pins.col1.set_low().unwrap();
    board.display_pins.row1.set_high().unwrap();

    loop {}
}
```

The first few lines until the `main` function just do some basic imports and setup we mostly looked at before. However, the `main` function looks pretty different to what we have seen up to now.

The first line is related to how most HALs written in Rust work internally. As discussed before they are built on top of PAC crates which own (in the Rust sense) all the peripherals of a chip. When we say

```
let mut board = Board::take().unwrap();
```

We take all of these peripherals from the PAC and bind them to a variable. In this specific case we are not only working with a HAL but with

an entire BSP, so this also takes ownership of the Rust representation of the other chips on the board.

> **NOTE**: If you are wondering why we have to call `unwrap()` here, in theory it is possible for `take()` to be called more than once. This would lead to the peripherals being represented by two separate variables and thus lots of possible confusing behaviour because two variables modify the same resource. In order to avoid this, PACs are implemented in a way that it would panic if you tried to take the peripherals twice.

(Again, if you are confused by all of this, the next chapter will go through it all again in greater detail.)

Now we can light the LED connected to `row1`, `col1` up by setting the `row1` pin to high (i.e. switching it on). The reason we can leave `col1` set to low is because of how the LED matrix circuit works. Furthermore, `embedded-hal` is designed in a way that every operation on hardware can possibly return an error, even just toggling a pin on or off. Since that is highly unlikely in our case, we can just `unwrap()` the result.

Testing it

Testing our little program is quite simple. First put it into `src/main.rs`. Afterwards we simply have to run the `cargo embed` command from the last section again, and let it flash just like before. Then open our GDB and connect to the GDB stub:

```
$ # Your GDB debug command from the last section
(gdb) target remote :1337
Remote debugging using :1337
cortex_m_rt::Reset                    ()                    at
/home/nix/.cargo/registry/src/github.com-
1ecc6299db9ec823/cortex-m-rt-0.6.12/src/lib.rs:489
489     pub unsafe extern "C" fn Reset() -> ! {
(gdb)
```

We now let the program run via the GDB `continue` command: one of the LEDs on the front of the micro:bit should light up.

Hello World

In the last section, you wrote a sort of "Hello World" program. But for embedded programmers, the "real Hello World" is to blink an LED — any LED — on and off once per second. A program that does this is commonly known as a "blinky".

Why blinky? Because this shows that you have enough control of the board you're working with to perform this simple task. You can get a program loaded onto the machine and running, you can find and turn on the appropriate pin on the MCU, you can delay for a fixed amount of time. Once you have this much control, other tasks become much more straightforward.

In previous chapters, you found out several ways to load a program onto your MB2. Now it's just a question of which pin you turn on and off, and how you delay between these actions.

Let's start by finding out how to work with the needed pins. There's a path you can follow for this if you know how to read electronic circuit "schematic" diagrams. You can find the MB2 schematic, find an LED on that schematic that you want to turn on and off, and find what GPIO pins on the nRF52833 are attached to that LED. (The MB2 is a bit unusual in this regard: usually an LED is attached to just one pin that turns it on or off. The LED "display" on the MB2 is hooked up in a more complicated way to allow turning on and off combinations of LEDs at once: a feature that we will be using shortly.)

We will work with the LED in the upper-left corner of the MB2 display. Tracing the `ROW1` and `COL1` wires this LED is connected to, we can see that they go to pins on the nRF52833 labeled `AC17`/`P0.21` and `B11`/`AIN4`/`P0.28`. Digging further through the documentation we find that `AC17` and `B11` are the row and column indices of the physical pins (solder balls, really) on the bottom of the chip — useless to us. `AIN4` just means that this pin can act as an "Analog Input", which is also currently useless to us. (It will come into play later.)

This leaves `P0.21` and `P0.28`. These labels correspond to bits in the memory of the nRF52833 that can be turned on and off to get the LED to light up. Because electronics reasons, if pin `P0.21` is turned on (thus outputting 3.3V) and pin `P0.28` is turned off (thus accepting voltage) the LED will light up.

But what do we do in software to cause this to occur? We will work at the level of the `nrf52833-hal` crate. The Hardware Abstraction Layer (HAL) is a chunk of software designed to make a particular microcontroller easier to work with. As can be seen from the name, we have one for the microcontroller on the MB2. It happens to contain everything needed to turn our target LED on.

Take a look at `examples/light-up.rs` in this chapter's directory, and then try running it. You could use something fancy like before, but we have it set up so that

```
cargo run --example light-up
```

will load and run your program. That one LED should now be brightly lit!

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

use cortex_m_rt::entry;
use nrf52833_hal::{gpio, pac};
use panic_halt as _;

#[entry]
fn main() -> ! {
    let peripherals = pac::Peripherals::take().unwrap();
    let p0 = gpio::p0::Parts::new(peripherals.P0);
                                    let        _row1        =
p0.p0_21.into_push_pull_output(gpio::Level::High);
                                    let        _col1        =
p0.p0_28.into_push_pull_output(gpio::Level::Low);

    #[allow(clippy::empty_loop)]
```

```
    loop {}
}
```

Note that we access the Peripheral Access Crate (PAC) for this chip through our HAL crate. There's a complicated dance needed to get access to our pins. Finally, since we can just initialize the pins to the right levels, we don't need to set them. Wiggling the pins is a topic for the next section.

Toggle it

Let's turn the LED on and off repeatedly. That's how you make it blink, right?

In `examples/fast-blink.rs` you'll find the next iteration of our blinky. I've decided to make it blink the next LED over, while leaving the original LED on. That is an easy change.

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

use cortex_m_rt::entry;
use embedded_hal::digital::OutputPin;
use nrf52833_hal::{gpio, pac};
use panic_halt as _;

#[entry]
fn main() -> ! {
    let peripherals = pac::Peripherals::take().unwrap();
    let p0 = gpio::p0::Parts::new(peripherals.P0);
                                    let         _row1         =
p0.p0_21.into_push_pull_output(gpio::Level::High);
                                let        mut        row2        =
p0.p0_22.into_push_pull_output(gpio::Level::Low);
                                    let         _col1         =
p0.p0_28.into_push_pull_output(gpio::Level::Low);

    loop {
        row2.set_high().unwrap();
        row2.set_low().unwrap();
    }
}
```

The `embedded-hal` crate is being used here to provide the Rust traits needed to set and unset the LED. This means that this part of the code is

portable to any Rust HAL that implements the `embedded-hal` traits as ours does.

But wait: neither LED is blinking! The second one is slightly dimmer than the first one, but they are both solidly on… or are they? Out of the box, the MB2 executes 64 *million* instructions per second. Let's assume it takes a few dozen instructions under the hood to turn the LED on or off. (Maybe possibly that many compiled in debug mode, though way less in release mode. Though the pins take a while to change state. I don't know.) Anyhow, that second LED is actually turning on and off hundreds of thousands of times — perhaps millions of times — every second. Your eye just can't keep up.

We'll need to wait a while between toggles. Turns out waiting is the hardest part.

Spin wait

To blink the LED, we need to wait about a half-second between each change. How do we do that?

Well, here's the dumb way. It's not good, but it's a start. Take a look at `examples/spin-wait.rs`.

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

use cortex_m::asm::nop;
use cortex_m_rt::entry;
use embedded_hal::digital::OutputPin;
use nrf52833_hal::{gpio, pac};
use panic_halt as _;

fn wait() {
    for _ in 0..4_000_000 {
        nop();
    }
}

#[entry]
fn main() -> ! {
    let peripherals = pac::Peripherals::take().unwrap();
    let p0 = gpio::p0::Parts::new(peripherals.P0);
                                let       mut       row1       =
p0.p0_21.into_push_pull_output(gpio::Level::High);
                                      let         _col1        =
p0.p0_28.into_push_pull_output(gpio::Level::Low);

    loop {
        wait();
        row1.set_high().unwrap();
        wait();
```

```
        row1.set_low().unwrap();
    }
}
```

Run this with `cargo run --release --example spin-wait` — the `--release` is really important here — and you should see the LED on your MB2 flash on and off *about* once per second.

Things you might be wondering:

- **What are those `_` characters in that number?** Rust allows these in numbers and ignores them. It's really convenient to make big numbers more readable. Here we are using them as commas (or whatever the separator is for groups of three digits in your country).
- **If the nRF52833 is running at 64MHz, why is the wait loop iterating only 4M times? Shouldn't it be 32M?** The wait loop executes several instructions each time through: the `nop` (see next section), some bookkeeping, and a branch back to the start of the loop. The code generated is roughly this for the first `wait()` call

  ```
  .LBB1_4:
      adds r3, #1
      nop
      cmp  r3, r2
      bne  .LBB1_4
  ```

  and this for the second

  ```
  .LBB1_6:
      subs        r3, #1
      nop
      bne .LBB1_6
  ```

  This is only three or four instructions, but the backward branch may cost an extra bit. Notice that these *are not the same:* the compiler chooses to emit different instructions for the first and second wait loops. See "it varies depending" below.

  Still, we're executing about 4 instructions per loop iteration. This means that on our 64MHz CPU a half-second spin should take

64M/2/4 = 8M iterations to complete. So something is slowing us down by a factor of 2. What? I dunno. This whole thing is terrible.

- **Why is `--release` so all-important?** Try without it. Notice that the LED is still flashing on and off, but with a period of *many* seconds. The wait loop is now unoptimized and is taking many instructions each time through.
- **What is that `nop()` call and why is it there?** We shall answer this in the next section.
- **Why do you refer to this as "the dumb way"?**
    - **It isn't precise.** Trying to tune that loop to reliably hit exactly 0.5 seconds is… not really a thing.
    - **It varies depending.** Different CPU? Different compilation flags? Different anything really? Now the timing has changed.
    - **It sucks power.** The CPU is running instructions as fast as it can, just to stay in place. If there's nothing else for it to do, it should quietly sleep until it is needed again. This doesn't matter much if you have USB power. But if you hook up your MB2 using the battery pack you'll really feel this.

In the next section, we'll discuss `nop()`. After that, we'll talk more about the other things about our blinky that need improving.

For such a simple program, this is a pretty complicated program. That's why we start with blinky.

NOP

You might wonder what that `nop()` call is doing in the `wait()` loop in `src/bin/spin-wait.rs`.

The answer is that it literally does nothing. The `nop()` function causes the compiler to put a `NOP` ARM machine instruction at that point in the program. `NOP` is a special instruction that causes the CPU to skip it. To ignore it. To literally do No OPeration with it (hence the name).

So get rid of that line and recompile the program. Don't forget `--release` mode. Then run it.

We're back to a slightly darker solid LED again. With no loop body, the compiler's optimizer decided that `wait()` function wasn't doing anything. So it just removed it for you at compile time. Thanks optimizer. You have made my wait loop infinitely fast.

How does `nop()` do its job? Well, if you look at the implementation of `nop()` you will find (after a bunch of digging around) that it is implemented like this:

```
asm!("nop", options(nomem, nostack, preserves_flags));
```

The `nop()` function is "inlined", so when you "call" it an actual ARM `NOP` assembly instruction is inserted into your program's code at that point. Because details, this `NOP` will not be removed or moved around by the compiler: it will stay right there where you put it.

The ability to insert assembly code into your program where needed is sometimes quite important in embedded programming. Sometime a CPU will have instructions the compiler doesn't know about, but that you still need in order to use the CPU effectively. Rust's `asm!()` directive gives you a way to do that.

Our spin-wait is still terrible. Let's talk about doing better.

Timers

One of the big advantages of a "bare-metal" embedded system is that you control everything that happens on your machine. This allows you to have really precise control of time: nothing will slow you down unless you let it.

However, we've seen that if we really want to get time right, we probably need help. Embedded MCUs like the nRF52833 all provide this kind of help in the form of "timers". A timer is a peripheral that, as its name implies, acts like a little clock that keeps very precise track of time.

The nRF52833 contains four timers. If you look at the documentation for the chip, you'll find that they are pretty complicated to set up and use. Luckily, the HAL provides a wrapper around timers that makes common uses easy. The most common use of a timer is to delay for a precise amount of time: just what our `wait()` function of the previous sections was trying to do.

Take a look at `examples/timer-blinky.rs`. This code sets up a timer and uses it to delay for 500ms (0.5s) between each toggle.

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

use cortex_m_rt::entry;
use embedded_hal::{delay::DelayNs, digital::OutputPin};
use nrf52833_hal::{gpio, pac, timer};
use panic_halt as _;

#[entry]
fn main() -> ! {
    let peripherals = pac::Peripherals::take().unwrap();

    let p0 = gpio::p0::Parts::new(peripherals.P0);
                              let       mut       row1       =
p0.p0_21.into_push_pull_output(gpio::Level::High);
```

```
                                    let         _col1         =
p0.p0_28.into_push_pull_output(gpio::Level::Low);

    let mut timer0 = timer::Timer::new(peripherals.TIMER0);

    loop {
        timer0.delay_ms(500);
        row1.set_high().unwrap();
        timer0.delay_ms(500);
        row1.set_low().unwrap();
    }
}
```

Run this code with `cargo run --release --example timer-blinky` and time it with a stopwatch. You'll find that it is exactly one second for each on-off cycle.

Things you might notice:

- We need to use the `embedded_hal::Delay` trait to get the `delay_ms()` method we're using.
- As before, we dig the peripheral out of the PAC peripherals struct and give it to the HAL.

Now we have a production-quality blinky. Let's talk a bit about the implications of all this.

Portability

(This section is optional. Feel free to skip to the next section, where we clean our code up a bit and call it a day.)

You may wonder whether all this fancy ecosystem is worth its weight. The setup for our blinky is pretty fancy, and uses a lot of Rust crates and features for such a simple job.

One cool advantage, though, is that our code becomes really portable. On a different board, the setup may be different, but the actual blinky loop is identical!

Let's take a look at a blinky for the Sipeed Longan Nano. This is a little $5 board that, like the MB2, is an embedded board with an MCU. Otherwise, it is completely different: different processor (the GD32VF103, with a RISC-V instruction set entirely unlike the ARM instruction set we're using), different peripherals, different board. But it has an LED attached to a GPIO pin, so we can blinky it.

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

use panic_halt as _;
use riscv_rt::entry;
use gd32vf103xx_hal::{pac, prelude::*, delay::McycleDelay};
use                    embedded_hal::{blocking::delay::DelayMs,
digital::v2::OutputPin};

#[entry]
fn main() -> ! {
    let dp = pac::Peripherals::take().unwrap();

                                let       mut       rcu       =
dp.RCU.configure().ext_hf_clock(8.mhz()).sysclk(108.mhz()).fre
eze();

    let gpioc = dp.GPIOC.split(&mut rcu);
```

```
    let mut led = gpioc.pc13.into_push_pull_output();
    let mut delay = McycleDelay::new(&rcu.clocks);

    loop {
        delay.delay_ms(500);
        led.set_high().unwrap();
        delay.delay_ms(500);
        led.set_low().unwrap();
    }
}
```

The differences in setup here are partly because different hardware, and partly because this code uses an older HAL crate that hasn't yet been updated for `embedded-hal` 1.0. Yet the main loop is identical as advertised, and the rest of the code is pretty recognizable. Because of the portability provided by Rust's easy cross-compilation and the embedded Rust ecosystem, blinky is just blinky.

You can find a complete working nanoblinky example on GitHub, if you want to see all the details or even get your own board and try it yourself.

Board support crate

Working directly with the PAC and HAL is pretty neat. Most ARM MCUs and many other MCUs that Rust can compile for have a PAC crate. If you are working with one that does not, writing a PAC crate can be tedious but is pretty straightforward. Many MCUs that have a PAC crate also have a HAL crate — again, it's mostly just tedious work to build one if it is absent. Code written at the PAC and HAL level gives access to the fine details of the MCU.

As we have seen, though, it becomes pretty annoying to keep track of just what is going on at the interface between our nRF52833 and the rest of our MB2. We have had to read schematics and whatnot to see how to use our off-board hardware.

A "board support crate" — known in the non-Rust embedded community as a Board Support Package (BSP) — is a crate built on top of the HAL and PAC for a board to abstract away the details and provide conveniences. The board support crate we have been working with is the `microbit-v2` crate.

Let's use `microbit-v2` to get a final, cleaned up blinky (`src/main.rs`).

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

use cortex_m_rt::entry;
use embedded_hal::{delay::DelayNs, digital::OutputPin};
use microbit::hal::{gpio, timer};
use panic_halt as _;

#[entry]
fn main() -> ! {
    let board = microbit::Board::take().unwrap();

                                let       mut       row1       =
board.display_pins.row1.into_push_pull_output(gpio::Level::Hig
```

```
h);
                                    let         _col1         =
board.display_pins.col1.into_push_pull_output(gpio::Level::Low
);

    let mut timer0 = timer::Timer::new(board.TIMER0);

    loop {
        timer0.delay_ms(500);
        row1.set_high().unwrap();
        timer0.delay_ms(500);
        row1.set_low().unwrap();
    }
}
```

In this case, we haven't changed much. Our board support crate has hidden the PAC (for now). More importantly, it has done so by letting us just use reasonable names for the row and column GPIO pins for the LED.

The `microbit-v2` crate provides even fancier support for those "display" LEDs. We will see this support used soon to do things more fun than blinky.

Registers

This chapter is a technical deep-dive. You can safely skip it for now and come back to it later if you like. That said, there's a lot of good stuff in here, so I'd recommend you dive in.

It's time to explore what calling `display_pins.row1.set_high()` 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 (`src/main.rs`).

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

#[allow(unused_imports)]
use registers::entry;

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

    unsafe {
        // A magic address!
        const PORT_P0_OUT: u32 = 0x50000504;

        // Turn on the top row
        *(PORT_P0_OUT as *mut u32) |= 1 << 21;

        // Turn on the bottom row
        *(PORT_P0_OUT as *mut u32) |= 1 << 19;

        // Turn off the top row
        *(PORT_P0_OUT as *mut u32) &= !(1 << 21);
```

```
        // Turn off the bottom row
        *(PORT_P0_OUT as *mut u32) &= !(1 << 19);
    }

    loop {}
}
```

What's this magic?

The address `0x50000504` 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.

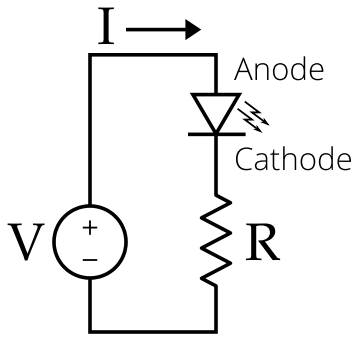
This particular register controls General Purpose Input/Output (GPIO) *pins* (GPIO *is* a peripheral) and can be used to *drive* each of those pins *low* or *high*.

(On the nRF52833 there are more than 32 GPIOs, yet the CPU is 32-bit. Thus, the GPIO pins are organized in two groups "P0" and "P1", with a set of registers for reading, writing and configuring each group. The address above is the address of the output register for the P0 pins.)

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 Light Emitting Diodes (LEDs). An LED will emit light when voltage is applied to it. As the name implies, an LED also acts as a "diode". A diode will only let electricity flow in one direction. Hook an LED up "forwards" and light comes out. Hook it up "backwards" and nothing happens.



Luckily for us, the microcontroller's pins are connected such that we can drive the LEDs the right way round. All that we have to do is apply enough voltage across the pins to turn the LED on. The pins attached to the LEDs are normally configured as *digital outputs* and can 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

We have previously seen the GPIO pins on the nRF52833. On this chip (and on many others) the GPIO pins are grouped into *ports*. There are two ports, Port 0 and Port 1, abbreviated to `P0` and `P1` respectively. The pins within each port are named with numbers starting from 0. Port 0 has 32 pins, named `P0.00` to `P0.31`, and Port 1 has 10 pins, named `P1.00` to `P1.09`.

The first thing we have to remember out is which pin is connected to which LED. We previously did this by tracing the schematic. That turns out to be hard mode: the required information is in the MB2 pinmap table.

The table says:

- `ROW1`, the top LED row, is connected to the pin `P0.21`. `P0.21` is the short form of: Pin 21 on Port 0.
- `ROW5`, the bottom LED row, is connected to the pin `P0.19`.

Up to this point, we know that we want to change the state of the pins `P0.21` and `P0.19` to turn the top and bottom rows on and off. These pins are part of Port 0 so we'll use the `P0` peripheral to set them up.

Each peripheral has a register *block* associated with 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 `P0` peripheral. That information is in the following section of the microcontroller Product Specification:

> Section 4.2.4 Instantiation - Page 22

The table says that base address of the `P0` register block is `0x5000_0000`.

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 6.8.2 Registers - Page 144

`OUT` is the register which we will be using to set/reset. Its offset value is `0x504` from the base address of the `P0`. We can look up `OUT` in the reference manual.

That register is specified right under the `GPIO` registers table:

> Subsection 6.8.2.1 OUT - Page 145

Anyway, `0x5000_0000` + `0x504` = `0x50000504`. That looks familiar! Finally!

This is the register we were writing to. The documentation says some interesting things. First, this register can both be written to and read from. Next, the register is a 32-bit piece of memory, and each bit represents the state of the corresponding pin. That means that bit 19 matches pin 19, for instance. Setting the bit to 1 will enable the pin output, and setting it to 0 will reset it. Furthermore, we can see that all pin outputs are disabled by default, as the reset value of all bits is 0.

We'll use GDB's `examine` command: `x`. Depending on the configuration of your GDB server, GDB will refuse to read memory that isn't specified. You can disable this behaviour by running:

```
set mem inaccessible-by-default off
```

So here we go. First turn off the `inaccessible-by-default` flag, then set a couple of breakpoints, reset the device and halt.

```
(gdb) set mem inaccessible-by-default off
(gdb) break 16
Breakpoint 1 at 0x172: file src/07-registers/src/main.rs, line
16.
Note: automatically using hardware breakpoints for read-only
addresses.
(gdb) break 19
Breakpoint 2 at 0x17c: file src/07-registers/src/main.rs, line
19.
(gdb) break 22
```

```
Breakpoint 3 at 0x184: file src/07-registers/src/main.rs, line
22.
(gdb) break 25
Breakpoint 4 at 0x18c: file src/07-registers/src/main.rs, line
25.
(gdb) monitor reset halt
Resetting and halting target
Target halted
```

All right. Let's continue until the first breakpoint, right before line 16, and print the contents of the register at address `0x50000504`.

```
(gdb) c
Continuing.

Breakpoint 1, registers::__cortex_m_rt_main () at src/07-
registers/src/main.rs:16
16              *(PORT_P0_OUT as *mut u32) |= 1 << 21;
(gdb) x 0x50000504
0x50000504:     0x00000000
```

Ok, we see that the register's value is `0x00000000` or `0` at this point. This corresponds with the data in the product specification, which says that `0` is the 'reset value' of this register. That means that once the MCU resets, the register will have `0` as its value.

Let's go on. This line consists of multiple instructions (reading, bitwise ORing and writing), so we need to instruct the debugger to continue execution more than once, until we hit the next breakpoint.

```
(gdb) c
Continuing.

Program received signal SIGINT, Interrupt.
0x00000174 in registers::__cortex_m_rt_main () at src/07-
registers/src/main.rs:16
16              *(PORT_P0_OUT as *mut u32) |= 1 << 21;
(gdb) c
Continuing.
```

```
Breakpoint 2, registers::__cortex_m_rt_main () at src/07-
registers/src/main.rs:19
19              *(PORT_P0_OUT as *mut u32) |= 1 << 19;
```

We've stopped right before line 19, meaning that line 16 is fully executed at this point. Let's have a look at the `OUT` register's contents again:

```
(gdb) x 0x50000504
0x50000504:     0x00200000
```

The value of the `OUT` register is `0x00200000` at this point, which is `2097152` in decimal, or `2^21`. That means that bit 21 is set to 1, and the rest of the bits is set to 0. That corresponds to the code on line 16, which writes `1 << 21`, or a 1 shifted left 21 positions, bitwise ORed with `OUT`s current value (which was 0), to the `OUT` register.

Writing `1 << 21` (`OUT[21]= 1`) to `OUT` sets `P0.21` *high*. That turns the top LED row *on*. Check that the top row is now indeed lit up.

```
(gdb) c
Continuing.
```

Yeah, I was gonna say that. Now, hit 'c' another time to continue execution up to the next breakpoint and print its value.

```
Program received signal SIGINT, Interrupt.
0x0000017e in registers::__cortex_m_rt_main () at src/07-
registers/src/main.rs:19
19              *(PORT_P0_OUT as *mut u32) |= 1 << 19;
(gdb) c
Continuing.

Breakpoint 3, registers::__cortex_m_rt_main () at src/07-
registers/src/main.rs:22
22              *(PORT_P0_OUT as *mut u32) &= !(1 << 21);
(gdb) x 0x50000504
0x50000504:     0x00280000
```

On line 19, we've set bit 21 of `OUT` to 1, keeping bit 19 as is. The result is `0x00280000`, which is `2621440` in decimal, or `2^19 + 2^21`, meaning that both bit 19 and bit 21 is set to 1.

Writing `1 << 19` (`OUT[19]= 1`) to `OUT` sets `P0.19` *high*. That turns the bottom LED row *on*. As such, the bottom row should now be lit up.

The following lines turn the rows off again. First the top row, then the bottom row. This time, we're doing a bitwise AND operation, combined with a bitwise NOT. We calculate `!(1 << 21)`, which is all bits set to 1, except for bit 21. Next, we bitwise AND that with the current value of `OUT`, ensuring that only bit 21 is set to 0, keeping the value of the other bits intact.

Continue execution and check that the reported values of the `OUT` register matches what you expect. You can press `CTRL+C` to pause execution once the device enters the endless loop at the end of the `main` function.

```
(gdb) c
Continuing.

Program received signal SIGINT, Interrupt.
0x00000186 in registers::__cortex_m_rt_main () at src/07-
registers/src/main.rs:22
22              *(PORT_P0_OUT as *mut u32) &= !(1 << 21);
(gdb) c
Continuing.

Breakpoint 4, registers::__cortex_m_rt_main () at src/07-
registers/src/main.rs:25
25              *(PORT_P0_OUT as *mut u32) &= !(1 << 19);
(gdb) x 0x50000504
0x50000504:     0x00080000
(gdb) c
Continuing.

Program received signal SIGINT, Interrupt.
0x0000018e in registers::__cortex_m_rt_main () at src/07-
```

```
registers/src/main.rs:25
25              *(PORT_P0_OUT as *mut u32) &= !(1 << 19);
(gdb) c
Continuing.
^C
Program received signal SIGINT, Interrupt.
0x00000196  in  registers::__cortex_m_rt_main  ()  at  src/07-
registers/src/main.rs:28
28          loop {}
(gdb) x 0x50000504
0x50000504:     0x00000000
```

And at this points all LEDs should be turned off again!

(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 `0x00000000` 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.

First, we'll use cargo objdump to get us the assembly of the build artifacts from both the optimized and the non-optimized build.

```
# Non-optimized
cargo objdump -- --disassemble --no-show-raw-insn --source > debug.dump
# Optimized
cargo objdump --release -- --disassemble --no-show-raw-insn --source > release.dump
```

Let's see what's in there. Specifically, let's try to find the assembly that manipulates the `OUT` register.

First, let's have a look at the contents of `debug.dump`, the assembly from the non-optimized build. I skipped a bunch and added my comments behind the `; <--`, indicating the line number in the source code that corresponds to the instruction.

```
$ cat debug.dump
[...]
00000158 <main>:
     158:       push    {r7, lr}
     15a:       mov     r7, sp
                  15c:                            bl      0x160 <registers::__cortex_m_rt_main::h0b7888ca966441cf>  @  imm  = #0x0
```

```
00000160 <registers::__cortex_m_rt_main::h0b7888ca966441cf>:
     160:       push    {r7, lr}
     162:       mov     r7, sp
     164:       sub     sp, #0x8
                  166:                          bl      0x198
<registers::init::hb6346637538e8ec5> @ imm = #0x2e
      16a:              movw    r1, #0x504         ; <-- Load
half of `OUT` register address into register `r1`
      16e:              movt    r1, #0x5000        ; <-- Load
half of `OUT` register address into register `r1`
     172:       str     r1, [sp, #0x4]
      174:              ldr     r0, [r1]           ; <-- (16)
value at the address in `r1` into `r0`.
     176:              orr     r0, r0, #0x200000 ; <-- (16) B
OR the value in `r0` with `0x200000`, and store in `r0`
      17a:              str     r0, [r1]           ; <-- (16)
contents of `r0` in memory at address from `r1`
      17c:              ldr     r0, [r1]           ; <-- (19)
value at the address in `r1` into `r0`.
     17e:              orr     r0, r0, #0x80000  ; <-- (19) B
OR the value in `r0` with `0x80000`, and store in `r0`
      182:              str     r0, [r1]           ; <-- (19)
contents of `r0` in memory at address from `r1`
      184:              ldr     r0, [r1]           ; <-- (22)
value at the address in `r1` into `r0`.
     186:              bic     r0, r0, #0x200000 ; <-- (22) B
AND the value in `r0` with bitwise complement of `0x200000`,
and store in `r0`
      18a:              str     r0, [r1]           ; <-- (22)
contents of `r0` in memory at address from `r1`
      18c:              ldr     r0, [r1]           ; <-- (25)
value at the address in `r1` into `r0`.
     18e:              bic     r0, r0, #0x80000  ; <-- (25) B
AND the value in `r0` with bitwise complement of `0x80000`,
and store in `r0`
      192:              str     r0, [r1]           ; <-- (25)
```

```
contents of `r0` in memory at address from `r1`
                 194:                           b       0x196
<registers::__cortex_m_rt_main::h0b7888ca966441cf+0x36> @ imm
= #-0x2
                 196:                           b       0x196
<registers::__cortex_m_rt_main::h0b7888ca966441cf+0x36> @ imm
= #-0x4
[...]
```

As you can see, the non-optimized assembly contains 4 loads, 4 stores, and 4 bit manipulation instructions. Those correspond nicely with the code we wrote. Now, let's have a look at the optimized assembly.

```
$ cat release.dump
[...]
00000158 <main>:
     158:       push    {r7, lr}
     15a:       mov     r7, sp
                 15c:                           bl      0x160
<registers::__cortex_m_rt_main::h1f38525e07b97485>  @  imm  =
#0x0

00000160 <registers::__cortex_m_rt_main::h1f38525e07b97485>:
     160:       push    {r7, lr}
     162:       mov     r7, sp
                 164:                           bl      0x17a
<registers::init::h4390f1d4f8a071f7> @ imm = #0x12
     168:               movw    r0, #0x504          ; <-- Loa
half of `OUT` register address into register `r0`
     16c:               movt    r0, #0x5000         ; <-- Loa
half of `OUT` register address into register `r0`
     170:       ldr     r1, [r0]                ; <-- (?) Load
value at the address in `r0` into `r1`.
      172:              bic     r1, r1, #0x280000        ; <-
Bitwise AND the value in `r1` with bitwise complement of
`0x280000`, and store in `r1`
     176:               str     r1, [r0]                 ; <-
```

```
Store contents of `r0` in memory at address from `r0`
                 178:                           b       0x178
<registers::__cortex_m_rt_main::h1f38525e07b97485+0x18> @ imm
= #-0x4
[...]
```

Huh? Just a single load - bit manipulate - store? 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.

How do we prevent LLVM from misoptimizing our program? We use *volatile* operations instead of plain reads/writes (`examples/volatile.rs`):

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

use core::ptr;

#[allow(unused_imports)]
use registers::entry;

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

    unsafe {
        // A magic address!
        const PORT_P0_OUT: u32 = 0x50000504;

        // Turn on the top row
        let out = ptr::read_volatile(PORT_P0_OUT as *mut u32);
         ptr::write_volatile(PORT_P0_OUT as *mut u32, out | 1
<< 21);

        // Turn on the bottom row
        let out = ptr::read_volatile(PORT_P0_OUT as *mut u32);
```

```
         ptr::write_volatile(PORT_P0_OUT as *mut u32, out | 1
<< 19);

        // Turn off the top row
        let out = ptr::read_volatile(PORT_P0_OUT as *mut u32);
        ptr::write_volatile(PORT_P0_OUT as *mut u32, out & !(1
<< 21));

        // Turn off the bottom row
        let out = ptr::read_volatile(PORT_P0_OUT as *mut u32);
        ptr::write_volatile(PORT_P0_OUT as *mut u32, out & !(1
<< 19));
    }

    loop {}
}
```

Let's run cargo objdump once again, with optimizations enabled.

```
cargo objdump -q --release --bin volatile -- --disassemble --
no-show-raw-insn  > release.volatile.dump
```

All right, now have a look at what's inside:

```
$ cat release.volatile.dump
[...]
00000158 <main>:
     158:       push    {r7, lr}
     15a:       mov     r7, sp
                  15c:                          bl      0x160
<registers::__cortex_m_rt_main::h1f38525e07b97485>  @  imm  =
#0x0

00000160 <registers::__cortex_m_rt_main::h1f38525e07b97485>:
     160:       push    {r7, lr}
     162:       mov     r7, sp
                  164:                          bl      0x192
<registers::init::h4390f1d4f8a071f7> @ imm = #0x2a
```

```
     168:       movw    r0, #0x504
     16c:       movt    r0, #0x5000
     170:       ldr     r1, [r0]
     172:       orr     r1, r1, #0x200000
     176:       str     r1, [r0]
     178:       ldr     r1, [r0]
     17a:       orr     r1, r1, #0x80000
     17e:       str     r1, [r0]
     180:       ldr     r1, [r0]
     182:       bic     r1, r1, #0x200000
     186:       str     r1, [r0]
     188:       ldr     r1, [r0]
     18a:       bic     r1, r1, #0x80000
     18e:       str     r1, [r0]
                  190:                            b       0x190
<registers::__cortex_m_rt_main::h1f38525e07b97485+0x30> @ imm
= #-0x4
[...]
```

Hey, look at that! Now we've got our four load - manipulate - store cycles back. Step through the code once again using GDB to see the volatile operations in action!

`0xBAAAAAAD` address

Not all the peripheral memory can be accessed. Look at this program (`examples/bad.rs`).

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

use core::ptr;

#[allow(unused_imports)]
use registers::entry;

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

    unsafe {
        ptr::read_volatile(0x5000_A784 as *const u32);
    }

    loop {}
}
```

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

Now, let's try it.

```
$ cargo run
(..)
Resetting and halting target
Target halted
(gdb) continue
Continuing.

Breakpoint 1, registers::__cortex_m_rt_main_trampoline () at
```

```
src/07-registers/src/main.rs:9
9       #[entry]
(gdb) continue
Continuing.

Program received signal SIGINT, Interrupt.
registers::__cortex_m_rt_main       ()       at       src/07-
registers/src/main.rs:10
10      fn main() -> ! {
(gdb) continue
Continuing.

Breakpoint  3,  cortex_m_rt::HardFault_  (ef=0x2001ffb8)  at
src/lib.rs:1046
1046        loop {}
(gdb)
```

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 `registers` crate depends on the `cortex-m-rt` crate which defines a default *hard fault* handler, named `HardFault_`, that handles the "invalid memory address" exception. `embed.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
1040  #[allow(unused_variables)]
1041    #[doc(hidden)]
```

```
1042    #[cfg_attr(cortex_m,        link_section        =
".HardFault.default")]
1043    #[no_mangle]
1044    pub    unsafe    extern    "C"    fn    HardFault_(ef:
&ExceptionFrame) -> ! {
1045        #[allow(clippy::empty_loop)]
1046        loop {}
1047    }
1048
1049    #[doc(hidden)]
1050    #[no_mangle]
```

`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: 0x5000a784,
  r1: 0x3,
  r2: 0x2001ff24,
  r3: 0x0,
  r12: 0x1,
  lr: 0x4403,
  pc: 0x43ea,
  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<u32>:
1654    pub unsafe fn read_volatile<T>(src: *const T) -> T {
   0x000043d2 <+0>:     push    {r7, lr}
   0x000043d4 <+2>:     mov     r7, sp
```

```
   0x000043d6 <+4>:     sub     sp, #16
   0x000043d8 <+6>:     str     r0, [sp, #4]
   0x000043da <+8>:     str     r0, [sp, #8]

1655          // SAFETY: the caller must uphold the safety
contract for `volatile_load`.
1656        unsafe {
1657            assert_unsafe_precondition!(
               0x000043dc     <+10>:    b.n     0x43de
<core::ptr::read_volatile<u32>+12>
   0x000043de <+12>:    ldr     r0, [sp, #4]
   0x000043e0 <+14>:    movs    r1, #4
               0x000043e2     <+16>:    bl      0x43f4
<core::ptr::read_volatile::precondition_check>
               0x000043e6     <+20>:    b.n     0x43e8
<core::ptr::read_volatile<u32>+22>

1658                check_language_ub,
1659                    "ptr::read_volatile requires that the
pointer argument is aligned and non-null",
1660                (
1661                    addr: *const () = src as *const (),
1662                    align: usize = align_of::<T>(),
1663                ) => is_aligned_and_not_null(addr, align)
1664            );
1665            intrinsics::volatile_load(src)
   0x000043e8 <+22>:    ldr     r0, [sp, #4]
   0x000043ea <+24>:    ldr     r0, [r0, #0]           ; <--
That's the one!
   0x000043ec <+26>:    str     r0, [sp, #12]
   0x000043ee <+28>:    ldr     r0, [sp, #12]

1666        }
1667    }
   0x000043f0 <+30>:    add     sp, #16
   0x000043f2 <+32>:    pop     {r7, pc}
```

```
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` *CPU register*. By the way, a CPU (processor) register not a memory mapped register; it doesn't have an associated address like, say, `OUT`.

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) print/x *ef
$1 = cortex_m_rt::ExceptionFrame {
  r0: 0x5000a784,
  r1: 0x3,
  r2: 0x2001ff24,
  r3: 0x0,
  r12: 0x1,
  lr: 0x4403,
  pc: 0x43ea,
  xpsr: 0x1000000
}
```

`r0` contains the value `0x5000_A784` which is the invalid address we called the `read_volatile` function with.

Spooky action at a distance

`OUT` is not the only register that can control the pins of Port E. The `OUTSET` register also lets you change the value of the pins, as can `OUTCLR`. However, `ODRSET` and `OUTCLR` don't let you retrieve the current output status of Port E.

`OUTSET` is documented in:

> Subsection 6.8.2.2. OUTSET - Page 145

Let's look at below program. The key to this program is `fn print_out`. This function prints the current value in `OUT` to the `RTT` console (`examples/spooky.rs`):

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

use core::ptr;

#[allow(unused_imports)]
use registers::{entry, rprintln};

// Print the current contents of P0.OUT
fn print_out() {
    const P0_OUT: u32 = 0x5000_0504;

      let out = unsafe { ptr::read_volatile(P0_OUT as *const
u32) };

    rprintln!("P0.OUT = {:#08x}", out);
}

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

```
    unsafe {
        // A bunch of magic addresses!
        const P0_OUTSET: u32 = 0x5000_0508;
        const P0_OUTCLR: u32 = 0x5000_050C;

        // Print the initial contents of OUT
        print_out();

        // Turn on the top LED row
        ptr::write_volatile(P0_OUTSET as *mut u32, 1 << 21);
        print_out();

        // Turn on the bottom LED row
        ptr::write_volatile(P0_OUTSET as *mut u32, 1 << 19);
        print_out();

        // Turn off the top LED row
        ptr::write_volatile(P0_OUTCLR as *mut u32, 1 << 21);
        print_out();

        // Turn off the bottom LED row
        ptr::write_volatile(P0_OUTCLR as *mut u32, 1 << 19);
        print_out();
    }

    loop {}
}
```

You'll see this if you run this program:

```
$ cargo embed
# cargo-embed's console
(..)
15:13:24.055: P0.OUT = 0x000000
15:13:24.055: P0.OUT = 0x200000
15:13:24.055: P0.OUT = 0x280000
```

```
15:13:24.055: P0.OUT = 0x080000
15:13:24.055: P0.OUT = 0x000000
```

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

Type safe manipulation

One of the registers of `P0`, the `IN` register, is documented as a read-only register.

> 6.8.2.4 IN - Pages 145 and 146

Note that in the 'Access' column of the table, only the 'R' is given for this register. We are not supposed to write to this register or Bad Stuff May Happen.

Registers have different read/write permissions. Some of them are write only, others can be read and written to and there must be others that are read only.

Directly working with hexadecimal addresses is also error-prone. You already saw that trying to access an invalid memory address caused an exception which disrupted 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! `registers::init()` actually returns a value that provides a type safe API to manipulate the registers of the `P0` and `P1` ports.

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 (`examples/type-safe.rs`).

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

#[allow(unused_imports)]
use registers::entry;

#[entry]
fn main() -> ! {
    let (p0, _p1) = registers::init();

    // Turn on the top row
    p0.out.modify(|_, w| w.pin21().set_bit());

    // Turn on the bottom row
    p0.out.modify(|_, w| w.pin19().set_bit());

    // Turn off the top row
    p0.out.modify(|_, w| w.pin21().clear_bit());

    // Turn off the bottom row
    p0.out.modify(|_, w| w.pin19().clear_bit());

    loop {}
}
```

First thing you notice: There are no magic addresses involved. Instead we use a more human friendly way, `p0.out`, to refer to the `OUT` register in the `P0` port register block.

The register block has a `modify` method that takes a closure. Before this closure is called, the `OUT` register's value is read and passed to the closure as the `r` parameter. Given the value of `r`, you can manipulate `w` to the desired new value of the register using its methods. The result is written to the register once the closure returns. In our case, the current value of the register is also passed in the `w` parameter, allowing us to just manipulate `w` when we want to keep the rest of the register bits as is.

The `modify` method is defined for registers that allow both write and read access. If you'd like to just read a register's value, but not update it, you can use the `read` method. Or, if you simply want to write a register value without reading, there's the `write` method.

Read-only registers only expose `read`, and write-only registers only expose `write`. This prevents users from accessing a register in a way that's not allowed, and therefore you don't need to wrap the calls in an `unsafe` block. And you don't need to figure out the exact register address and bit positions yourself!

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

`p0` is a reference to the `P0` port's register block. `print p0` will return the base address of the register block, and `print *p0` will print its value.

```
$ cargo run
(..)
Target halted
(gdb) set mem inaccessible-by-default off
(gdb) break main.rs:12
Breakpoint 4 at 0x162: main.rs:12. (2 locations)
(gdb) continue
Continuing.

Program received signal SIGINT, Interrupt.
cortex_m_rt::DefaultPreInit () at src/lib.rs:1058
1058    pub unsafe extern "C" fn DefaultPreInit() {}
(gdb) continue
Continuing.

Breakpoint 1, registers::__cortex_m_rt_main_trampoline () at
src/07-registers/src/main.rs:7
7       #[entry]
(gdb) continue
Continuing.
```

```
Program received signal SIGINT, Interrupt.
registers::__cortex_m_rt_main       ()       at       src/07-
registers/src/main.rs:8
8       fn main() -> ! {
(gdb) continue
Continuing.

Breakpoint 4.2, registers::__cortex_m_rt_main () at src/07-
registers/src/main.rs:12
12          p0.out.modify(|_, w| w.pin21().set_bit());
(gdb)                          print                          *p0
; ⬅️ Printing `*p0` here!
$1 = nrf52833_pac::p0::RegisterBlock {
  _reserved0: [0 <repeats 1284 times>],
                                                              out:
nrf52833_pac::generic::Reg<nrf52833_pac::p0::out::OUT_SPEC> {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0
      }
    },
                                                           _marker:
core::marker::PhantomData<nrf52833_pac::p0::out::OUT_SPEC>
  },
                                                            outset:
nrf52833_pac::generic::Reg<nrf52833_pac::p0::outset::OUTSET_SP
EC> {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0
      }
    },
                                                           _marker:
core::marker::PhantomData<nrf52833_pac::p0::outset::OUTSET_SPE
C>
  },
```

```
                                                                outclr:
nrf52833_pac::generic::Reg<nrf52833_pac::p0::outclr::OUTCLR_SP
EC> {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0
      }
    },
                                                              _marker:
core::marker::PhantomData<nrf52833_pac::p0::outclr::OUTCLR_SPE
C>
  },
                                                                in_:
nrf52833_pac::generic::Reg<nrf52833_pac::p0::in_::IN_SPEC> {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0
      }
    },
                                                              _marker:
core::marker::PhantomData<nrf52833_pac::p0::in_::IN_SPEC>
  },
                                                                dir:
nrf52833_pac::generic::Reg<nrf52833_pac::p0::dir::DIR_SPEC> {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 3513288704
      }
    },
                                                              _marker:
core::marker::PhantomData<nrf52833_pac::p0::dir::DIR_SPEC>
  },
                                                                dirset:
nrf52833_pac::generic::Reg<nrf52833_pac::p0::dirset::DIRSET_SP
EC> {
    register: vcell::VolatileCell<u32> {
```

```
      value: core::cell::UnsafeCell<u32> {
        value: 3513288704
      }
    },
                                                          _marker:
core::marker::PhantomData<nrf52833_pac::p0::dirset::DIRSET_SPE
C>
  },
                                                           dirclr:
nrf52833_pac::generic::Reg<nrf52833_pac::p0::dirclr::DIRCLR_SP
EC> {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 3513288704
      }
    },
                                                          _marker:
core::marker::PhantomData<nrf52833_pac::p0::dirclr::DIRCLR_SPE
C>
  },
                                                            latch:
nrf52833_pac::generic::Reg<nrf52833_pac::p0::latch::LATCH_SPEC
> {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
        value: 0
      }
    },
                                                          _marker:
core::marker::PhantomData<nrf52833_pac::p0::latch::LATCH_SPEC>
  },
                                                       detectmode:
nrf52833_pac::generic::Reg<nrf52833_pac::p0::detectmode::DETEC
TMODE_SPEC> {
    register: vcell::VolatileCell<u32> {
      value: core::cell::UnsafeCell<u32> {
```

```
        value: 0
      }
    },
                                                          _marker:
core::marker::PhantomData<nrf52833_pac::p0::detectmode::DETECT
MODE_SPEC>
  },
  _reserved9: [0 <repeats 472 times>],
                                                          pin_cnf:
[nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF
_SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 2
        }
      },
                                                          _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                          }      <repeats      11      times>,
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 3
        }
      },
                                                          _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                                                                },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 2
```

```
        }
      },
                                                          _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                                                        },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 2
        }
      },
--Type <RET> for more, q to quit, c to continue without
paging--c
                                                          _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                                                        },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 2
        }
      },
                                                          _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                                                        },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 3
        }
```

```
      },
                                                                  _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                                                                        },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 2
        }
      },
                                                                  _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                                                                        },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 2
        }
      },
                                                                  _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                                                                        },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 2
        }
      },
                                                                  _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
```

```
PEC>
                                                                    },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 3
        }
      },
                                                          _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                                                                    },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 2
        }
      },
                                                          _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                                                                    },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 3
        }
      },
                                                          _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                                                                    },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
```

```
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 3
        }
      },
                                                          _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                                                        },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 2
        }
      },
                                                          _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                                                        },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 3
        }
      },
                                                          _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                                                        },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
```

```
          value: 2
        }
      },
                                                            _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                                                                  },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 2
        }
      },
                                                            _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                                                                  },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 2
        }
      },
                                                            _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                                                                  },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 3
        }
      },
```

```
                                                                _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                                                              },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 2
        }
      },
                                                                _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                                                              },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 3
        }
      },
                                                                _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
                                                              },
nrf52833_pac::generic::Reg<nrf52833_pac::p0::pin_cnf::PIN_CNF_
SPEC> {
      register: vcell::VolatileCell<u32> {
        value: core::cell::UnsafeCell<u32> {
          value: 3
        }
      },
                                                                _marker:
core::marker::PhantomData<nrf52833_pac::p0::pin_cnf::PIN_CNF_S
PEC>
```

```
    }]
}
```

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

Use `cargo objdump` to grab the assembler code to `release.type-safe.dump`:

```
cargo objdump -q --release --bin type-safe -- --disassemble --no-show-raw-insn  > release.type-safe.dump
```

Then search for `main` in `release.type-safe.dump`

```
00000158 <main>:
     158:       push    {r7, lr}
     15a:       mov     r7, sp
                  15c:                              bl      0x160 <registers::__cortex_m_rt_main::h0e9b57c6799332fd>  @  imm  = #0x0

00000160 <registers::__cortex_m_rt_main::h0e9b57c6799332fd>:
     160:       push    {r7, lr}
     162:       mov     r7, sp
                  164:                              bl      0x192 <registers::init::hec71dddc40be11b5> @ imm = #0x2a
     168:       movw    r0, #0x504
     16c:       movt    r0, #0x5000
     170:       ldr     r1, [r0]
     172:       orr     r1, r1, #0x200000
     176:       str     r1, [r0]
     178:       ldr     r1, [r0]
     17a:       orr     r1, r1, #0x80000
     17e:       str     r1, [r0]
     180:       ldr     r1, [r0]
```

```
     182:       bic     r1, r1, #0x200000
     186:       str     r1, [r0]
     188:       ldr     r1, [r0]
     18a:       bic     r1, r1, #0x80000
     18e:       str     r1, [r0]
                  190:                            b       0x190
<registers::__cortex_m_rt_main::h0e9b57c6799332fd+0x30>  @  imm
= #-0x4
```

You can validate that this yields the exact same binary as the one with the calls to `ptr::read_volatile` and `ptr::write_volatile`.

The best part of all this is that nobody had to write a single line of code to implement the GPIO API. All the code 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.

LED roulette

Alright, let's build a "real" application. The goal is to get to this display of spinning lights:

Since working with the LED pins separately is quite annoying (especially if you have to use basically all of them like here) you can use the `microbit-v2` BSP crate, discussed previously, to work with the MB2's LED "display". It works like this (`examples/light-it-all.rs`):

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

use cortex_m_rt::entry;
use embedded_hal::delay::DelayNs;
use    microbit::{board::Board,    display::blocking::Display,
hal::Timer};
use panic_rtt_target as _;
use rtt_target::rtt_init_print;

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

    let board = Board::take().unwrap();
```

```
    let mut timer = Timer::new(board.TIMER0);
    let mut display = Display::new(board.display_pins);
    let light_it_all = [
        [1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1],
    ];

    loop {
        // Show light_it_all for 1000ms
        display.show(&mut timer, light_it_all, 1000);
        // clear the display again
        display.clear();
        timer.delay_ms(1000_u32);
    }
}
```

The Rust array `light_it_all` shown in the example contains 1 where the LED is on and 0 where it is off. The call to `show()` takes a timer for the BSP display code to use for delaying, a *copy* of the array, and a length of time in milliseconds to show this display before returning.

The challenge

You are now well armed to face our challenge! Again, your application should look like this:

If you can't exactly see what's happening here it is in a much slower version:

If you need a hint, `templates/solution.rs` provides a mostly-filled-out chunk of code to finish. I would suggest you try it on your own first, though: it should be doable by now…

Got it?

My solution

What solution did you come up with?

Here's mine. It's probably one of the simplest (but of course not most beautiful) ways to generate the required matrix:

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

use cortex_m_rt::entry;
use    microbit::{board::Board,    display::blocking::Display,
hal::Timer};
use panic_rtt_target as _;
use rtt_target::rtt_init_print;

#[rustfmt::skip]
const PIXELS: [(usize, usize); 16] = [
    (0, 0),
    (0, 1),
    (0, 2),
    (0, 3),
    (0, 4),
    (1, 4),
    (2, 4),
    (3, 4),
    (4, 4),
    (4, 3),
    (4, 2),
    (4, 1),
    (4, 0),
    (3, 0),
    (2, 0),
    (1, 0),
];
```

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

    let board = Board::take().unwrap();
    let mut timer = Timer::new(board.TIMER0);
    let mut display = Display::new(board.display_pins);
    #[rustfmt::skip]
    let mut leds = [
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
    ];

    let mut last_led = (0, 0);

    loop {
        for current_led in PIXELS.iter() {
            leds[last_led.0][last_led.1] = 0;
            leds[current_led.0][current_led.1] = 1;
            display.show(&mut timer, leds, 30);
            last_led = *current_led;
        }
    }
}
```

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

```
$ cargo embed --release
```

If you want to debug your "release" mode binary you'll have to use a different GDB command:

```
$    gdb    ../../../target/thumbv7em-none-eabihf/release/led-
roulette
```

The Rust compiler modifies the machine instructions generated in a release build (sometimes by a lot) in order to try to make the code faster or smaller. Unfortunately, GDB has a hard time figuring out what is going on after this. As a result, debugging release builds with GDB can be difficult.

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:

```
$ cargo size --release -- -A
      Finished release [optimized + debuginfo] target(s) in
0.02s
led-roulette  :
section               size        addr
.vector_table          256         0x0
.text                 6332       0x100
.rodata                648      0x19bc
.data                    0  0x20000000
.bss                  1076  0x20000000
.uninit                  0  0x20000434
.debug_loc            9036         0x0
.debug_abbrev         2754         0x0
.debug_info          96460         0x0
.debug_aranges        1120         0x0
.debug_ranges        11520         0x0
.debug_str           71325         0x0
.debug_pubnames      32316         0x0
.debug_pubtypes      29294         0x0
.ARM.attributes         58         0x0
.debug_frame          2108         0x0
.debug_line          19303         0x0
.comment               109         0x0
Total                283715
```

Your numbers may differ somewhat depending on how your code is built: this is OK.

Know how to read this output? The `text` section contains the program instructions. The `rodata` section contains read-only data stored with the program instructions. The `data` and `bss` sections contain variables statically allocated in RAM (`static` variables). If you remember the specification of the microcontroller on your micro:bit, you should notice that its flash memory is less than double the size of this extremely simple binary: can this be right? As we can see from the size statistics most of the binary is actually made up of debugging related sections. However, those are not flashed to the microcontroller at any time — after all they aren't relevant for the execution.

Serial communication

This is what we'll be using. I hope your computer 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 computer. Now you might be asking yourself why exactly we aren't using RTT for this like we did before. RTT is a protocol that is meant to be used solely for debugging. You will most definitely not be able to find a device that actually uses RTT to communicate with some other device in production. However, serial communication is used quite often. For example some GPS receivers send the positioning information they receive via serial communication. In addition RTT, like many debugging protocols, is slow even compared to serial transfer rates.

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 computers don't usually support the serial communication protocol, and even if they do the voltage they use, ±5..12V, may be higher than the micro:bit will accept and may result in damaging it. You can't directly connect your computer to the microcontroller. You *can* buy very inexpensive USB←→serial converters that will support the 0..3V most modern microntroller boards need. While a serial converter is not necessary for the MB2, as shown below, it can be handy for inexpensive boards that have few communications options other than serial.)

The debug probe on the micro:bit itself can act as a USB←→serial converter. This means that the converter will sit between the two and expose a serial interface to the microcontroller and a USB interface to your computer. The microcontroller will see your computer as another serial device and your computer 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:

- Linux/UNIX
- Windows

For MacOS check out the Linux documentation, although your experience may differ somewhat.

Linux USB ← → serial tooling

The micro:bit's USB emulated serial device shows up in Linux when you connect the MB2 to a Linux USB port.

Connecting the micro:bit board

If you connect the micro:bit board to your computer you should see a new TTY device appear in `/dev`.

```
$ sudo dmesg -T | tail | grep -i tty
[63712.446286] cdc_acm 1-1.7:1.1: ttyACM0: USB ACM device
```

This is the USB←→serial device. On Linux, it's named `tty` (for "TeleTYpe", believe it or not). It should show up as `ttyACM0`, or maybe `ttyUSB0`. If other "ACM" devices are plugged in, the number will be higher. (On Mac OS `ls /dev/cu.usbmodem*` will show the serial device.)

But what exactly is `ttyACM0`? It's a file of course! Everything is a file in Unix:

```
$ ls -l /dev/ttyACM0
crw-rw----+ 1 root plugdev 166, 0 Jan 21 11:56 /dev/ttyACM0
```

Note that you will need to be either running as `root` (not advised) or a member of the group `plugdev` to read and write this device. You can then send out data by simply writing to this file:

```
$ echo 'Hello, world!' > /dev/ttyACM0
```

You should see the orange LED on the micro:bit, right next to the USB port, blink for a moment, whenever you enter this command.

minicom

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 the last line.

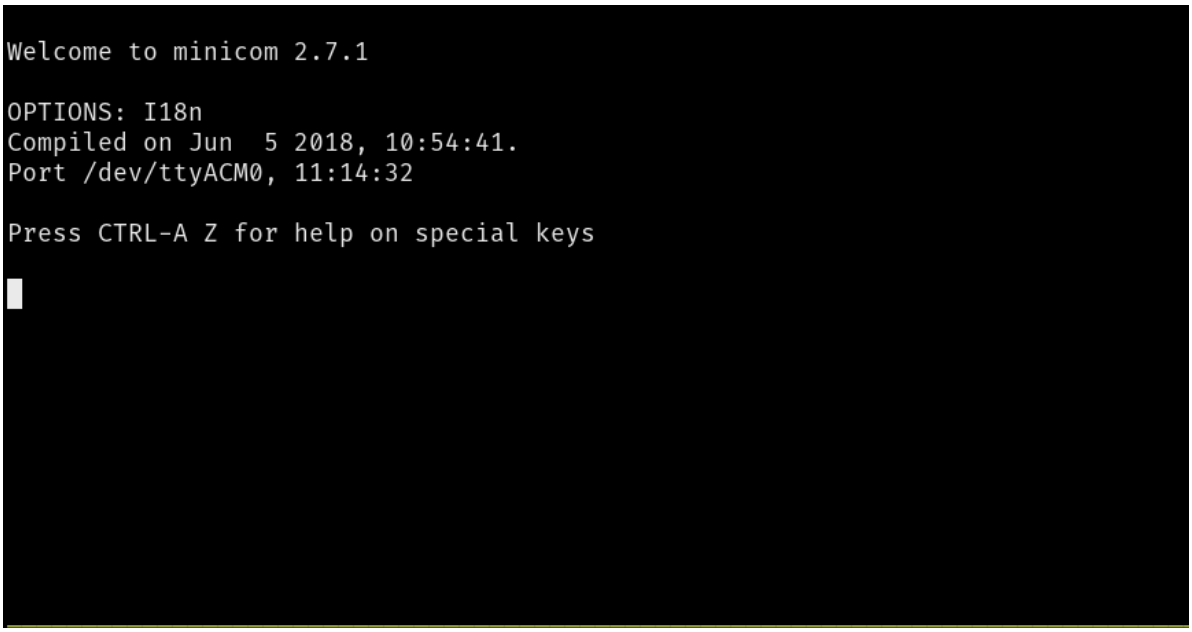
That file should be straightforward to read (except for the last two lines), but nonetheless let's go over it line by line:

- `pu baudrate 115200`. Sets baud rate to 115200 bps.
- `pu bits 8`. 8 bits per frame.
- `pu parity N`. No "parity check bit", which would be used for error detection.
- `pu stopbits 1`. 1 stop bit.
- `pu rtscts No`. No hardware flow control.
- `pu xonxoff No`. No software flow control.

Once that's in place, we can launch `minicom` on our ACM device, for example:

```
$ minicom -D /dev/ttyACM0 -b 115200
```

This tells `minicom` to open the serial device at `/dev/ttyACM0` and set its baud rate to 115200. A text-based user interface (TUI) will pop out.



You can now send data using the keyboard! Go ahead and type something. Note that the text UI will *not* echo back what you type. If you pay attention to the yellow LED on top of the micro:bit though, you will notice that it blinks whenever you type something.

`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

Start by unplugging your micro:bit.

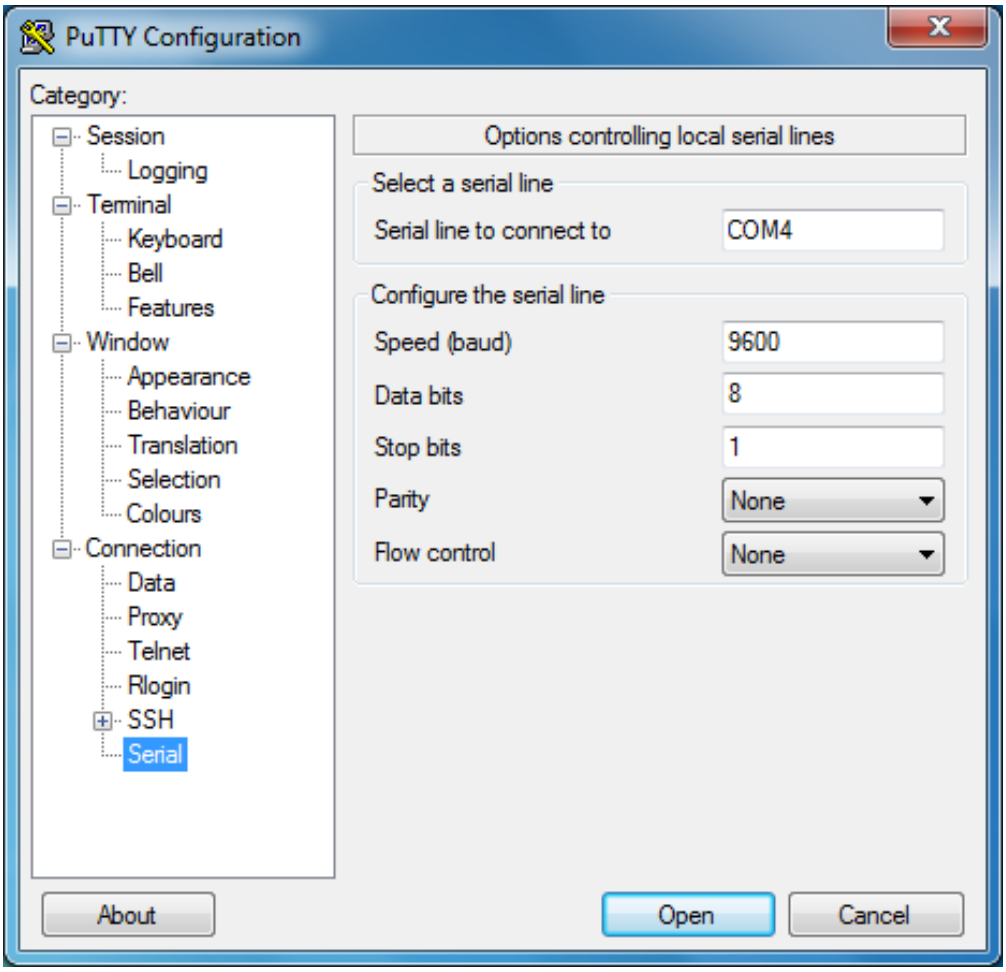
Before plugging the micro:bit back in, run the following command on the terminal:

```
$ mode
```

It will print a list of devices that are connected to your computer. 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 in the micro:bit and run the `mode` command again. If you see a new `COM` port appear on the list, then that's the COM port assigned to the serial functionality on the micro:bit.

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

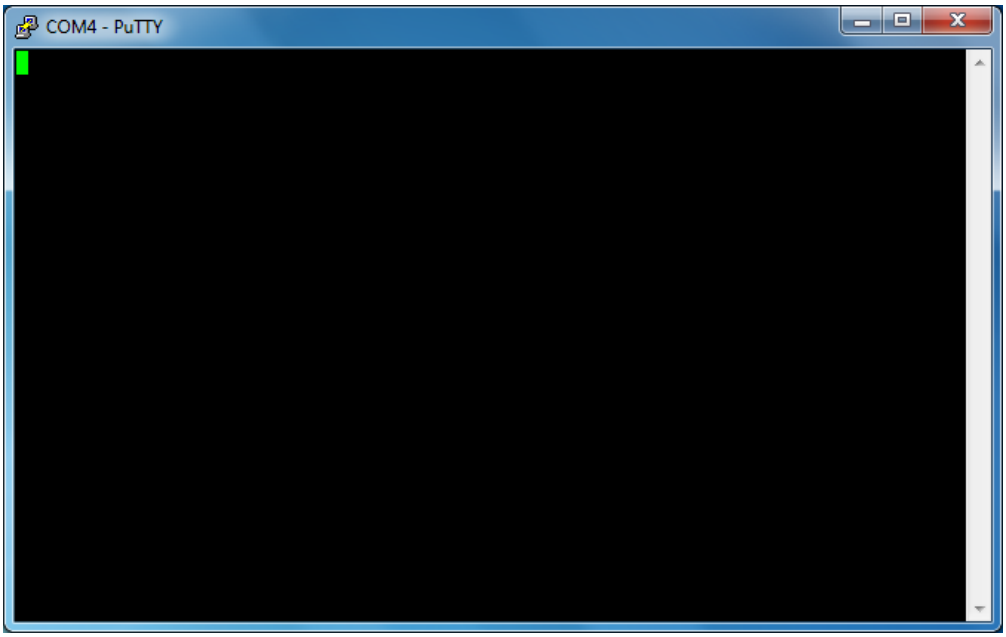


On the starter screen, which should have the "Session" category open, pick "Serial" as the "Connection type". On the "Serial line" field enter the `COM` device you got on the previous step, for example `COM3`.

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:



If you type on this console, the yellow LED on top of the micro:bit will blink. Each keystroke should make the LED blink once. Note that the console won't echo back what you type so the screen will remain blank.

UART

Our microcontroller (like most) has a peripheral called a UART (for "Universal Asynchronous Receiver/Transmitter). This peripheral can be configured to work with several serial communication protocols. The peripheral we will be working with is named UARTE (for "UART with Easy DMA", a topic outside the scope of this chapter).

Throughout this chapter, we'll use serial communication to exchange information between the microcontroller and your computer.

Send a single byte

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

In order to do that we will use the following snippet (this one is already in `10-uart/examples/send-byte.rs`):

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

use cortex_m::asm::wfi;
use cortex_m_rt::entry;
use panic_rtt_target as _;
use rtt_target::rtt_init_print;

use microbit::{
    hal::uarte,
    hal::uarte::{Baudrate, Parity},
};

use serial_setup::UartePort;

#[entry]
fn main() -> ! {
    rtt_init_print!();
    let board = microbit::Board::take().unwrap();

    let mut serial = {
        let serial = uarte::Uarte::new(
            board.UARTE0,
            board.uart.into(),
            Parity::EXCLUDED,
            Baudrate::BAUD115200,
        );
        UartePort::new(serial)
```

```
    };

    serial.write(b'X').unwrap();
    serial.flush().unwrap();

    loop {
        wfi();
    }
}
```

You might notice that one of the libraries used here, the `serial_setup` module, is not from `crates.io`, but was written for this project. The purpose of `serial_setup` is to provide a nice wrapper around the UARTE peripheral. If you want, you can check out what exactly the module does, but it is not required to understand this chapter in general.

We'll next discuss the initialization of UARTE. The UARTE is initialized with this piece of code:

```
uarte::Uarte::new(
    board.UARTE0,
    board.uart.into(),
    Parity::EXCLUDED,
    Baudrate::BAUD115200,
);
```

This function takes ownership of the UARTE peripheral representation in Rust (`board.UARTE0`) and the TX/RX pins on the board (`board.uart.into()`) so nobody else can mess with either the UARTE peripheral or our pins while we are using them. After that we pass two configuration options to the constructor: the baud rate (that one should be familiar) as well as an option called "parity". Parity is a way to allow serial communication lines to check whether the data they received was corrupted during transmission. We don't want to use that here so we simply exclude it. Then we wrap it up in the `UartePort` type so we can use it.

After the initialization, we send our `X` via the newly created uart instance. These serial functions are "blocking": they wait for the data to be

sent before returning. This is not always what is wanted: the microcontroller can do a lot of work while waiting for the byte to go out on the wire. However, in our case it is convenient and we didn't have other work to do anyway.

Last but not least, we `flush()` the serial port. This is because the UARTE may decide to buffer output until it has received a certain number of bytes to send. Calling `flush()` forces it to write the bytes it currently has right now instead of waiting for more.

Testing it

Before flashing this you should make sure to start your minicom/PuTTY as the data we receive via our serial communication is not backed up or anything: we have to view it live. Once your serial monitor is up you can flash the program just like in chapter 5:

```
$ cargo embed --example send-byte
  (...)
```

And after the flashing is finished, you should see the character `X` show up on your minicom/PuTTY terminal, congrats!

If you missed it, you can hit the reset button on the back of the MB2. This will cause the program to start from the beginning and send an `X` again.

Send a string

The next task will be to send a whole string from the microcontroller to your computer.

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

It's your turn to write the program.

Naive approach and `write!`

Naive approach

You probably came up with a program similar to the following (`examples/naive-send-string.rs`):

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

use cortex_m::asm::wfi;
use cortex_m_rt::entry;
use panic_rtt_target as _;
use rtt_target::rtt_init_print;

use microbit::hal::uarte::{self, Baudrate, Parity};

use serial_setup::UartePort;

#[entry]
fn main() -> ! {
    rtt_init_print!();
    let board = microbit::Board::take().unwrap();

    let mut serial = {
        let serial = uarte::Uarte::new(
            board.UARTE0,
            board.uart.into(),
            Parity::EXCLUDED,
            Baudrate::BAUD115200,
        );
        UartePort::new(serial)
    };

     for byte in b"The quick brown fox jumps over the lazy dog.\r\n".iter() {
        serial.write(*byte).unwrap();
    }
```

```
    serial.flush().unwrap();

    loop {
        wfi();
    }
}
```

While this is a perfectly valid implementation, at some point you might want to have all the nice perks of `print!` such as argument formatting and so on. If you are wondering how to do that, read on.

`write!` and `core::fmt::Write`

The `core::fmt::Write` trait allows us to use any struct that implements it in basically the same way as we use `print!` in the `std` world. In this case, the `Uart` struct from the `nrf` HAL does implement `core::fmt::Write` so we can refactor our previous program into this (`examples/send-string.rs`):

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

use core::fmt::Write;
use cortex_m::asm::wfi;
use cortex_m_rt::entry;
use panic_rtt_target as _;
use rtt_target::rtt_init_print;

use microbit::hal::uarte::{self, Baudrate, Parity};

use serial_setup::UartePort;

#[entry]
fn main() -> ! {
    rtt_init_print!();
    let board = microbit::Board::take().unwrap();

    let mut serial = {
        let serial = uarte::Uarte::new(
            board.UARTE0,
            board.uart.into(),
            Parity::EXCLUDED,
            Baudrate::BAUD115200,
        );
        UartePort::new(serial)
    };
```

```
     write!(serial, "The quick brown fox jumps over the lazy
dog.\r\n").unwrap();
    serial.flush().unwrap();

    loop {
        wfi();
    }
}
```

If you flash this program onto your micro:bit, you'll see that it is functionally equivalent to the iterator-based program you came up with.

Receive a single byte

So far we can send data from the microcontroller to your computer. It's time to try the opposite: receiving data from your computer (`examples/receive-byte.rs`).

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

use cortex_m_rt::entry;
use panic_rtt_target as _;
use rtt_target::{rprintln, rtt_init_print};

use microbit::hal::uarte::{self, Baudrate, Parity};

use serial_setup::UartePort;

#[entry]
fn main() -> ! {
    rtt_init_print!();
    let board = microbit::Board::take().unwrap();

    let mut serial = {
        let serial = uarte::Uarte::new(
            board.UARTE0,
            board.uart.into(),
            Parity::EXCLUDED,
            Baudrate::BAUD115200,
        );
        UartePort::new(serial)
    };

    loop {
        let byte = serial.read().unwrap();
        rprintln!("{}", byte);
```

```
    }
}
```

The only part that changed, compared to our send byte program, is the loop at the end of `main()`. Here we use the `serial.read()` function in order to wait until a byte is available and read it. Then we print that byte into our RTT debugging console to see whether stuff is actually arriving.

Note that if you flash this program and start typing characters inside `minicom` to send them to your microcontroller you'll only be able to see numbers inside your RTT console since we are not converting the `u8` we received into an actual `char`. Since the conversion from `u8` to `char` is quite simple, I'll leave this task to you if you really do want to see the characters inside the RTT console.

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 receives. For this application, the microcontroller will be the server and you and your computer 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:

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

use cortex_m_rt::entry;
use core::fmt::Write;
use heapless::Vec;
use rtt_target::rtt_init_print;
use panic_rtt_target as _;

use microbit::{
    hal::prelude::*,
    hal::uarte,
    hal::uarte::{Baudrate, Parity},
};

mod serial_setup;
use serial_setup::UartePort;

#[entry]
fn main() -> ! {
    rtt_init_print!();
    let board = microbit::Board::take().unwrap();

    let mut serial = {
        let serial = uarte::Uarte::new(
            board.UARTE0,
```

```
            board.uart.into(),
            Parity::EXCLUDED,
            Baudrate::BAUD115200,
        );
        UartePort::new(serial)
    };

    // A buffer with 32 bytes of capacity
    let mut buffer: Vec<u8, 32> = 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

You will find my solution in `src/main.rs`:

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

use core::fmt::Write;
use cortex_m_rt::entry;
use heapless::Vec;
use microbit::hal::uarte::{self, Baudrate, Parity};
use panic_rtt_target as _;
use rtt_target::rtt_init_print;
use serial_setup::UartePort;

#[entry]
fn main() -> ! {
    rtt_init_print!();
    let board = microbit::Board::take().unwrap();

    let mut serial = {
        let serial = uarte::Uarte::new(
            board.UARTE0,
            board.uart.into(),
            Parity::EXCLUDED,
            Baudrate::BAUD115200,
        );
        UartePort::new(serial)
    };

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

    loop {
        buffer.clear();
```

```
        loop {
            // We assume that the receiving cannot fail
            let byte = serial.read().unwrap();

            if buffer.push(byte).is_err() {
                                write!(serial,  "error:  buffer
full\r\n").unwrap();
                break;
            }

            if byte == b'\r' {
                for byte in buffer.iter().rev().chain(&[b'\n',
b'\r']) {
                    serial.write(*byte).unwrap();
                }
                break;
            }
        }
        serial.flush().unwrap()
    }
}
```

I2C

We just saw the UART serial communication format. UART serial is widely used because it is simple and has been around almost forever. (Remember how the host device is called a "tty" for "TeleTYpe"? Yeah, that.) This ubiquity and simplicity makes it a popular choice for simple communications.

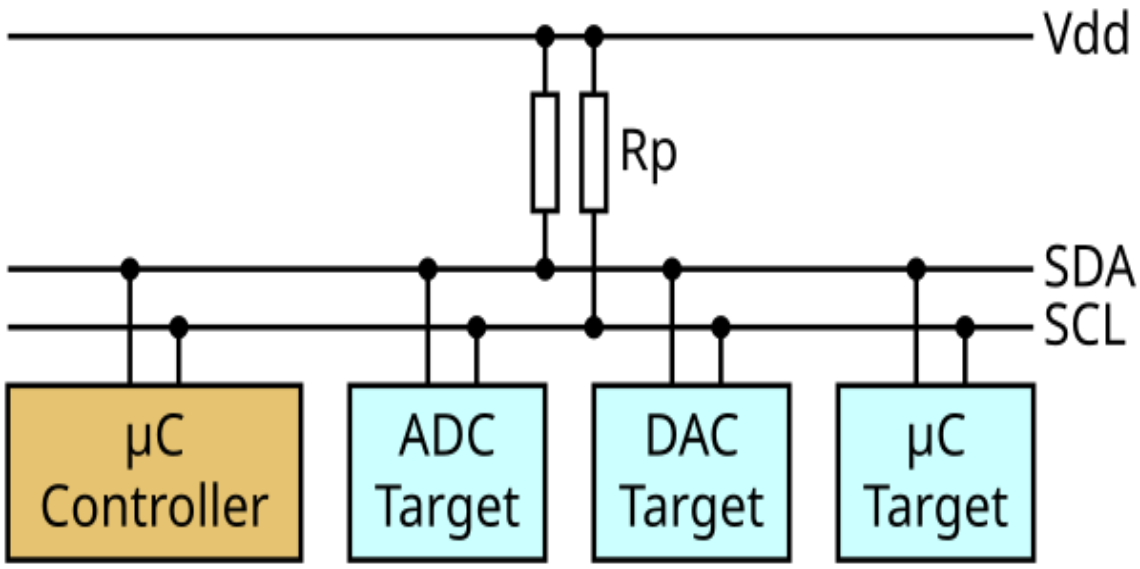
Because of hardware limitations on line length *vs* signal quality and because of difficulty of accurate decoding, UART serial typically caps out at about 115200 baud under ideal conditions. A UART serial port has both low bandwidth (11.5KB/s) and high latency (87µs/byte).

UART serial is point-to-point: there is no way to connect three or more devices to the same wire, and each wire requires a dedicated hardware device on each end.

The good news (and the bad news) is that there are *plenty* of other hardware-assisted serial communication protocols in the embedded space that overcome these limitations. Some of them are widely used in digital sensors.

The micro:bit board we are using has two motion sensors in it: an accelerometer and a magnetometer. Both of these sensors are packaged into a single component and can be accessed via an I2C bus.

I2C is pronounced "EYE-SQUARED-CEE" and stands for Inter-Integrated Circuit. I2C is a *synchronous* serial *bus* communication protocol: it uses two lines to exchange data: a data line (SDA) and a clock line (SCL). The clock line is used to synchronize the communication. Synchronous serial can run faster and more reliably than async serial. I2C devices have *bus addresses*: the hardware implementation allows sending bytes to a particular device, with other devices connected to the same wires ignoring this communication.



I2C uses a *controller*/*target* model: the controller is the device that *starts* and drives the communication with a target device. Several devices can be connected to the same bus at the same time, and can choose to act either as a controller or as a target. A controller device can communicate with a specific target device by first broadcasting the target address to the bus. This address can be 7 bits or 10 bits long. Once a controller has started a communication with a target, no device is other than the controller and target is allowed to transmit on the bus until the controller ends the communication.

> **NOTE** "Controller/target" was formerly referred to as "master/slave". You may still see that in literature or as labeling on boards. This terminology is now deprecated both in official standards and newer documents, but is used in the Nordic manual for our nRF52833 part and in some embedded Rust documentation.

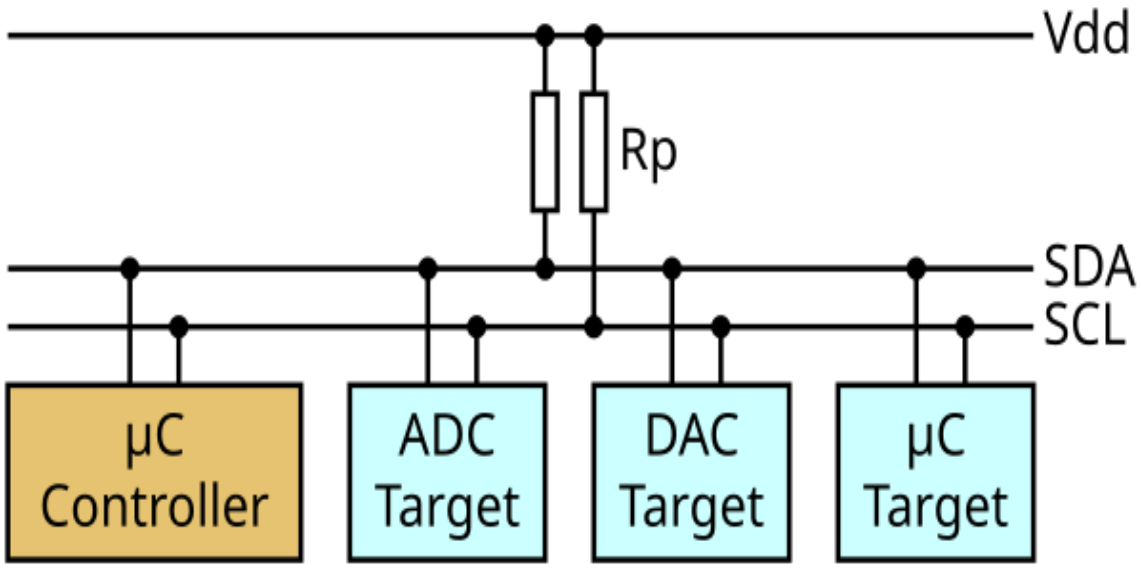
The clock line determines how fast data can be exchanged. The MB2 I2C interface can operate at speeds of 100, 250 or 400 Kbps. With other devices even faster modes are possible.

General protocol

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

Controller → Target

If the Controller wants to send data to the Target:

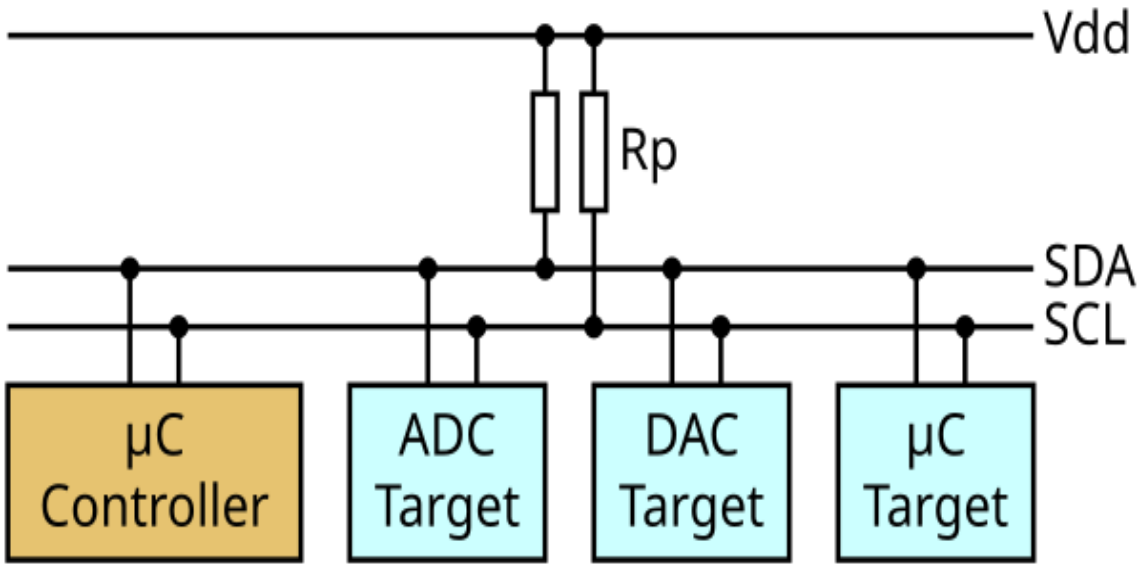


1. Controller: Broadcast START
2. C: Broadcast target address (7 bits) + the R/W (8th) bit set to WRITE
3. Target: Responds ACK (ACKnowledgement)
4. C: Send one byte
5. T: Responds ACK
6. Repeat steps 4 and 5 zero or more times
7. C: Broadcast STOP OR (broadcast RESTART and go back to (2))

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

Controller ← Target

If the controller wants to read data from the target:



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

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

LSM303AGR

Both of the motion sensors on the micro:bit, the magnetometer and the accelerometer, are packaged in a single component: the LSM303AGR integrated circuit. These two sensors can be accessed via an I2C bus. Each sensor behaves like an I2C target 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 LSM303AGR is its Data Sheet. Read through it to see how one can read the sensors' registers. That part is in:

> Section 6.1.1 I2C Operation - Page 38 - LSM303AGR Data Sheet

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

> Section 8 Register description - Page 46 - LSM303AGR Data Sheet

Read a single register

Let's put all that theory into practice!

First things first we need to know the target addresses of both the accelerometer and the magnetometer inside the chip, these can be found in the LSM303AGR's datasheet on page 39 and are:

- 0011001 for the accelerometer
- 0011110 for the magnetometer

> **NOTE** Remember that these are only the 7 leading bits of the address, the 8th bit is going to be the bit that determines whether we are performing a read or write.

Next up we'll need a register to read from. Lots of I2C chips out there will provide some sort of device identification register for their controllers to read. Considering the thousands (or even millions) of I2C chips out there it is highly likely that at some point two chips with the same address will end up being built (after all the address is "only" 7 bit wide). With this device ID register a driver can make sure that it is indeed talking to a LSM303AGR and not some other chip that just happens to have the same address. As you can read in the LSM303AGR's datasheet (specifically on page 46 and 61) this part does provide two registers — `WHO_AM_I_A` at address `0x0f` and `WHO_AM_I_M` at address `0x4f` — which contain some bit patterns that are unique to the device. (The "A" is for "Accelerometer" and the "M" is for "Magnetometer".)

The only thing missing now is the software part: we need to determine which API of the `microbit` or a HAL crate we should use for this. If you read through the datasheet of the nRF chip you are using you will soon find out that it doesn't actually have an I2C-specific peripheral. Instead, it has more general-purpose I2C-compatible peripherals called TWI ("Two-Wire Interface"), TWIM ("Two-Wire Interface Master") and TWIS ("Two-Wire Interface Slave"). We will normally be operating in controller mode and will use the newer TWIM, which supports "Easy DMA" — the TWI is provided mostly for backward compatibility with older devices.

Now if we put the documentation of the `twi(m)` module from the `microbit` crate together with all the other information we have gathered so far we'll end up with this piece of code to read out and print the two device IDs (`examples/chip-id.rs`):

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

use cortex_m::asm::wfi;
use cortex_m_rt::entry;
use panic_rtt_target as _;
use rtt_target::{rprintln, rtt_init_print};

use embedded_hal::i2c::I2c;
use microbit::{hal::twim, pac::twim0::frequency::FREQUENCY_A};

const ACCELEROMETER_ADDR: u8 = 0b0011001;
const MAGNETOMETER_ADDR: u8 = 0b0011110;

const ACCELEROMETER_ID_REG: u8 = 0x0f;
const MAGNETOMETER_ID_REG: u8 = 0x4f;

#[entry]
fn main() -> ! {
    rtt_init_print!();
    let board = microbit::Board::take().unwrap();

          let   mut   i2c   =   {   twim::Twim::new(board.TWIM0,
board.i2c_internal.into(), FREQUENCY_A::K100) };

    let mut acc = [0u8];
    let mut mag = [0u8];

    // First write the address + register onto the bus, then
read the chip's responses
```

```
                    i2c.write_read(ACCELEROMETER_ADDR,     &
[ACCELEROMETER_ID_REG], &mut acc)
        .unwrap();
     i2c.write_read(MAGNETOMETER_ADDR, &[MAGNETOMETER_ID_REG],
&mut mag)
        .unwrap();

       rprintln!("The  accelerometer  chip's  id  is:  {:#b}",
acc[0]);
    rprintln!("The magnetometer chip's id is: {:#b}", mag[0]);

    loop {
        wfi();
    }
}
```

Apart from the initialization, this piece of code should be straight forward if you understood the I2C protocol as described before. The initialization here works similarly to the one from the UART chapter. We pass the peripheral as well as the pins that are used to communicate with the chip to the constructor; and then the frequency we wish the bus to operate on, in this case 100 kHz (`K100`, since identifiers can't start with a digit).

Testing it

As usual

```
$ cargo embed --example chip-id
```

in order to test our little example program.

Using a driver

As we already discussed in chapter 5 `embedded-hal` provides abstractions which can be used to write platform independent code that can interact with hardware. In fact all the methods we have used to interact with hardware in chapter 7 and up until now in chapter 8 were from traits, defined by `embedded-hal`. Now we'll make actual use of the traits `embedded-hal` provides for the first time.

It would be pointless to implement a driver for our LSM303AGR for every platform embedded Rust supports (and new ones that might eventually pop up). To avoid this a driver can be written that consumes generic types that implement `embedded-hal` traits in order to provide a platform agnostic version of a driver. Luckily for us this has already been done in the `lsm303agr` crate. Hence reading the actual accelerometer and magnetometer values will now be basically a plug and play experience (plus reading a bit of documentation). In fact the `crates.io` page already provides us with everything we need to know in order to read accelerometer data but using a Raspberry Pi. We'll just have to adapt it to our chip:

Take a look at the linked page for the Raspberry Pi Linux sample code.

Because we already know how to create an instance of an object that implements the `embedded_hal::blocking::i2c` traits from the previous page, adapting the sample code is straightforward (`examples/show-accel.rs`):

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

use cortex_m_rt::entry;
use panic_rtt_target as _;
use rtt_target::{rprintln, rtt_init_print};

use microbit::{
    hal::{twim, Timer},
```

```
    pac::twim0::frequency::FREQUENCY_A,
};

use lsm303agr::{AccelMode, AccelOutputDataRate, Lsm303agr};

#[entry]
fn main() -> ! {
    rtt_init_print!();
    let board = microbit::Board::take().unwrap();

            let   i2c   =   {   twim::Twim::new(board.TWIM0,
board.i2c_internal.into(), FREQUENCY_A::K100) };
    let mut timer0 = Timer::new(board.TIMER0);

    // Code from documentation
    let mut sensor = Lsm303agr::new_with_i2c(i2c);
    sensor.init().unwrap();
    sensor
        .set_accel_mode_and_odr(
            &mut timer0,
            AccelMode::HighResolution,
            AccelOutputDataRate::Hz50,
        )
        .unwrap();
    loop {
        if sensor.accel_status().unwrap().xyz_new_data() {
                                      let   (x,   y,   z)   =
sensor.acceleration().unwrap().xyz_mg();
            // RTT instead of normal print
              rprintln!("Acceleration: x {} y {} z {}", x, y,
z);
        }
    }
}
```

Just like the last snippet you should just be able to try this out like this:

```
$ cargo embed --example show-accel
```

Furthermore if you (physically) move around your micro:bit a little you should see the acceleration numbers that are being printed change.

The challenge

The challenge for this chapter is, to build a small application that communicates with the outside world via the serial interface introduced in the last chapter. It should be able to receive the commands "magnetometer" as well as "accelerometer" and then print the corresponding sensor data in response. This time no template code will be provided since all you need is already provided in the UART and this chapter. However, here are a few clues:

- You might be interested in `core::str::from_utf8` to convert the bytes in the buffer to a `&str`, since we need to compare with `"magnetometer"` and `"accelerometer"`.
- You will (obviously) have to read the documentation of the magnetometer API, however it's more or less equivalent to the accelerometer one

My solution

My solution is in `src/main.rs`.

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

use core::str;

use cortex_m_rt::entry;
use embedded_hal::delay::DelayNs;
use panic_rtt_target as _;
use rtt_target::rtt_init_print;

use microbit::{
    hal::uarte::{self, Baudrate, Parity},
    hal::{twim, Timer},
    pac::twim0::frequency::FREQUENCY_A,
};

use core::fmt::Write;
use heapless::Vec;
use  lsm303agr::{AccelMode,  AccelOutputDataRate,  Lsm303agr,
MagMode, MagOutputDataRate};

use serial_setup::UartePort;

#[entry]
fn main() -> ! {
    rtt_init_print!();
    let board = microbit::Board::take().unwrap();

    let serial = uarte::Uarte::new(
        board.UARTE0,
        board.uart.into(),
```

```
        Parity::EXCLUDED,
        Baudrate::BAUD115200,
    );
    let mut serial = UartePort::new(serial);

            let   i2c   =   {   twim::Twim::new(board.TWIM0,
board.i2c_internal.into(), FREQUENCY_A::K100) };
    let mut timer0 = Timer::new(board.TIMER0);

    let mut sensor = Lsm303agr::new_with_i2c(i2c);
    sensor.init().unwrap();
    sensor
        .set_accel_mode_and_odr(
            &mut timer0,
            AccelMode::HighResolution,
            AccelOutputDataRate::Hz50,
        )
        .unwrap();
    sensor
        .set_mag_mode_and_odr(
            &mut timer0,
            MagMode::HighResolution,
            MagOutputDataRate::Hz50,
        )
        .unwrap();
                            let       mut       sensor       =
sensor.into_mag_continuous().ok().unwrap();
    let mut buffer: Vec<u8, 32> = Vec::new();

    loop {
        buffer.clear();

        loop {
            let byte = serial.read().unwrap();

            if byte == b'\r' {
```

```
                break;
            }

            if buffer.push(byte).is_err() {
                                write!(serial, "error: buffer
full\r\n").unwrap();
                break;
            }
        }

               if str::from_utf8(&buffer).unwrap().trim() ==
"accelerometer" {
                                                          while
!sensor.accel_status().unwrap().xyz_new_data() {
                timer0.delay_ms(1u32);
            }

                                       let (x, y, z) =
sensor.acceleration().unwrap().xyz_mg();
                 write!(serial, "Accelerometer: x {} y {} z
{}\r\n", x, y, z).unwrap();
         } else if str::from_utf8(&buffer).unwrap().trim() ==
"magnetometer" {
            while !sensor.mag_status().unwrap().xyz_new_data()
{
                timer0.delay_ms(1u32);
            }

                                       let (x, y, z) =
sensor.magnetic_field().unwrap().xyz_nt();
            write!(serial, "Magnetometer: x {} y {} z {}\r\n",
x, y, z).unwrap();
        } else {
                        write!(serial, "error: command not
detected\r\n").unwrap();
        }
```

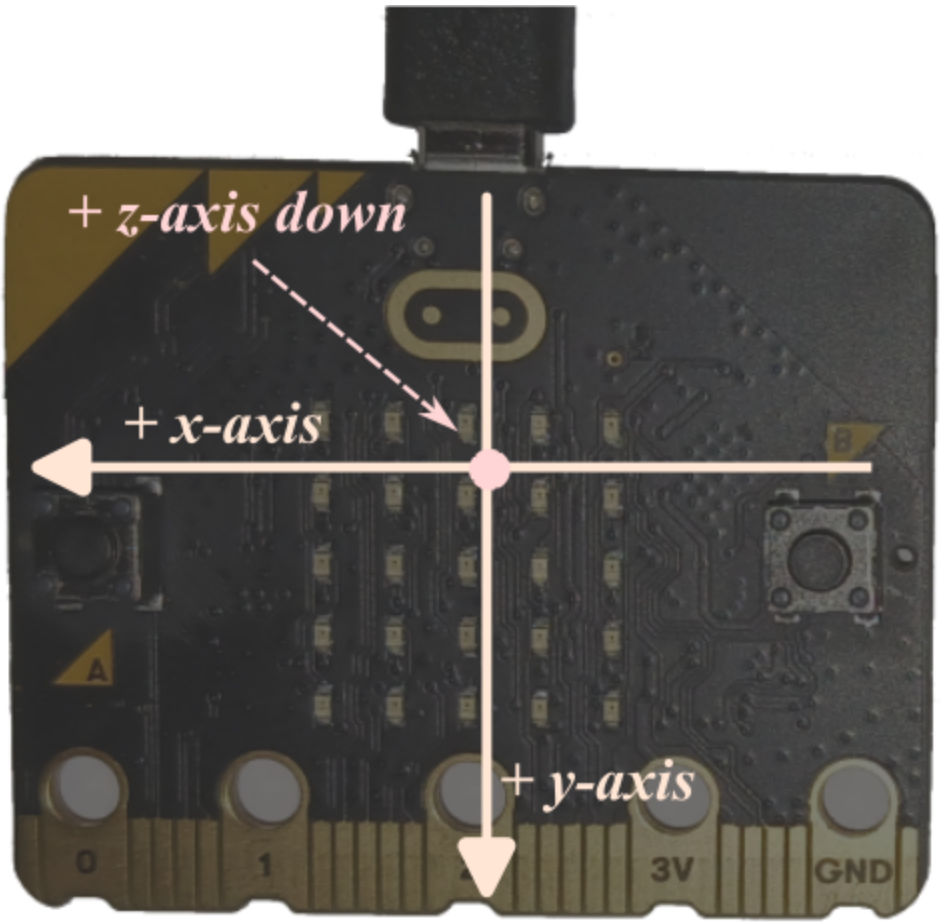
```
    }
}
```

LED compass

In this section, we'll implement a compass using the LEDs on the micro:bit. Like proper compasses, our LED compass must point north somehow. It will do that by turning on one of its outer LEDs; the LED turned on should point towards north.

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

The magnetometer has three axes associated with it. When the board is held flat with the LEDs facing uupward and the logo facing forward, the X and Y axes span the plane that is the floor. The X axis points to the left edge of the board. The Y axis points to the bottom (card connector) edge of the board. The Z axis points "into the floor", so downwards: "upside down" since the chip is mounted on the back. This is a "right-handed" coordinate system. It's all a bit confusing, since the reported field strengths are components of the magnetic field vector.



You should already be able to write a program that continuously prints the magnetometer data on the RTT console from the I2C chapter. After you write that program (`examples/show-mag.rs`), locate where north is at your current location. Then line up your micro:bit with that direction and observe how the sensor's X and Y measurements look.

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?

> **NOTE** Of the two MB2s I have handy at the time of this writing, one of them seems to have a somewhat broken magnetometer: the Z-axis is unusably offset. The manufacturer has a self-test process for detecting this and a calibration process for mitigating this kind of "hard iron" fault, which is usually the result of exposing the MB2 to a strong magnetic field at some point. However, the `lsm303agr` crate currently doesn't support either of these, and it seems like a lot for an

introductory guide to embedded systems. If you have only one MB2 and it doesn't seem to be working, you may just want to skip to the next chapter. Cheap hardware: whatcha gonna do?

The Earth's magnetic north is a fickle thing: it differs from true north in most places on Earth, sometimes substantially. It can point down into the ground quite a bit. It changes over time. Without allowing for all this, you won't get a very accurate compass even if your MB2 magnetometer is perfect (it's not). This US NOAA calculator https://www.ngdc.noaa.gov/geomag/calculators/mobileDeclination.shtml can be visited on your mobile device to get a good estimate of true north as well as magnetic north; you can give this UK BGS calculator your latitude, longitude and altitude to get both declination and inclination. At my location the "declination" (difference between true and magnetic north) is about 15°; the "inclination" is an astonishing 67° down into the ground.

NOTE The LSM303AGR magnetometer is not a particularly accurate device out-of-the box. The manufacturer recommends a fancy calibration procedure for finding adjustments to the magnetometer readings. You can find further information, a sample calibration implementation and some fancier compass graphics in appendix 3: since we're doing something fairly basic with the magnetometer we won't worry about it in this chapter.

Magnitude

How strong is the Earth's magnetic field? According to the documentation about the `magnetic_field()` method the `x` `y` `z` values we are getting are in nanoteslas. That means the only thing we have to compute in order to get the magnitude of the magnetic field in nanoteslas is the magnitude of the 3D vector that our `x` `y` `z` values describe. As you might remember from school this is simply:

```
use libm::sqrtf;
let magnitude = sqrtf(x * x + y * y + z * z);
```

Rust does not have floating-point math functions such as `sqrtf()` in `core`, so our `no_std` program has to get an implementation from somewhere. We use the libm crate for this.

Putting all this together in a program (`examples/magnitude.rs`):

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

use cortex_m_rt::entry;
use embedded_hal::delay::DelayNs;
use panic_rtt_target as _;
use rtt_target::{rprintln, rtt_init_print};

use libm::sqrtf;

use microbit::{
    hal::{twim, Timer},
    pac::twim0::frequency::FREQUENCY_A,
};

use lsm303agr::{Lsm303agr, MagMode, MagOutputDataRate};

#[entry]
```

```
fn main() -> ! {
    rtt_init_print!();
    let board = microbit::Board::take().unwrap();

            let   i2c   =   {   twim::Twim::new(board.TWIM0,
board.i2c_internal.into(), FREQUENCY_A::K100) };

    let mut timer0 = Timer::new(board.TIMER0);

    let mut sensor = Lsm303agr::new_with_i2c(i2c);
    sensor.init().unwrap();
    sensor
        .set_mag_mode_and_odr(
            &mut timer0,
            MagMode::HighResolution,
            MagOutputDataRate::Hz10,
        )
        .unwrap();
                            let       mut       sensor       =
sensor.into_mag_continuous().ok().unwrap();

    loop {
        while !sensor.mag_status().unwrap().xyz_new_data() {
            timer0.delay_ms(1u32);
        }
                                    let    (x,    y,    z)    =
sensor.magnetic_field().unwrap().xyz_nt();
        let (x, y, z) = (x as f32, y as f32, z as f32);
        let magnitude = sqrtf(x * x + y * y + z * z);
        rprintln!("{} mG", magnitude / 100.0);
    }
}
```

Run this with `cargo run --example magnitude`.

This program will report the magnitude (strength) of the magnetic field in nanotesla (`nT`) and milligauss (`mG`, where 1 `mG` = 100 `nT`). 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 ideally would see a value vaguely in that range. Your value will likely be off quite a bit because the sensor has not been calibrated: see appendix 3 for calibration. With calibration, I see a magnitude of around `340 mG`.

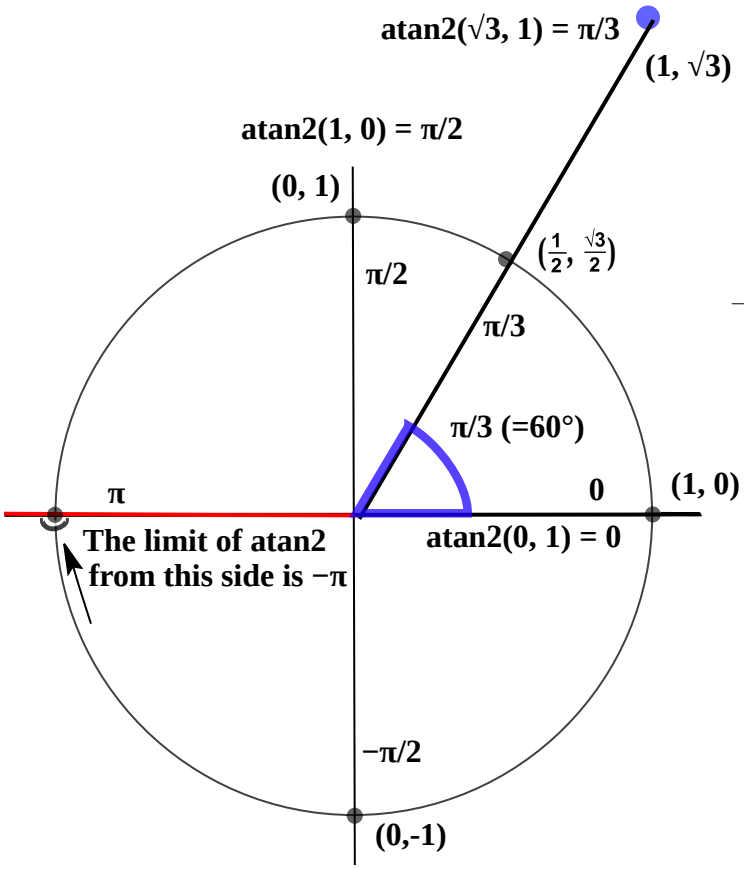
Some questions:

- 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?

The Challenge

We'll use some fancy math to get the precise angle that the magnetic field forms with the X and Y axes of the magnetometer. This will allow us to figure out which LED is pointing north.

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. Note that our coordinate system is rotated 180° from this.

Here's the starter code (in `templates/compass.rs`). `theta`, in radians, has already been computed. You need to pick which LED to turn on based on the value of `theta`.

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

```
#![no_std]

use cortex_m_rt::entry;
use embedded_hal::delay::DelayNs;
use panic_rtt_target as _;
use rtt_target::rtt_init_print;

// You'll find these useful ;-).
use core::f32::consts::PI;
use libm::{atan2f, floorf};

use microbit::{
    display::blocking::Display,
    hal::{Timer, twim},
    pac::twim0::frequency::FREQUENCY_A,
};

use lsm303agr::{Lsm303agr, MagMode, MagOutputDataRate};

#[entry]
fn main() -> ! {
    rtt_init_print!();
    let board = microbit::Board::take().unwrap();

            let   i2c   =   {   twim::Twim::new(board.TWIM0,
board.i2c_internal.into(), FREQUENCY_A::K100) };

    let mut timer0 = Timer::new(board.TIMER0);
    let mut display = Display::new(board.display_pins);

    let mut sensor = Lsm303agr::new_with_i2c(i2c);
    sensor.init().unwrap();
    sensor.set_mag_mode_and_odr(
        &mut timer0,
        MagMode::HighResolution,
        MagOutputDataRate::Hz10,
```

```
    ).unwrap();
                            let       mut       sensor       =
sensor.into_mag_continuous().ok().unwrap();

    let mut leds = [[0u8; 5]; 5];

    // Indexes of the 16 LEDs to be used in the display, and
their
    // compass directions.
    #[rustfmt::skip]
    let indices = [
         (2, 0) /* W */, (3, 0) /* W-SW */, (3, 1) /* SW */,
(4, 1) /* S-SW */,
         (4, 2) /* S */, (4, 3) /* S-SE */, (3, 3) /* SE */,
(3, 4) /* E-SE */,
         (2, 4) /* E */, (1, 4) /* E-NE */, (1, 3) /* NE */,
(0, 3) /* N-NE */,
         (0, 2) /* N */, (0, 1) /* N-NW */, (1, 1) /* NW */,
(1, 0) /* W-NW */,
    ];

    loop {
        // Measure the magnetic field.
        let (x, y) = todo!();

          // Get an angle between -180° and 180° from the x
axis.
        let theta = atan2f(y as f32, x as f32);

        // Figure out what LED index to blink.
        let index = todo!();

        // Blink the given LED.
        let (r, c) = indices[index];
        leds[r][c] = 255u8;
        display.show(&mut timer0, leds, 50);
```

```
        leds[r][c] = 0u8;
        display.show(&mut timer0, leds, 50);
    }
}
```

Suggestions/tips:

- A whole circle rotation equals 360 degrees.
- `PI` radians is equivalent to 180 degrees.
- If `theta` is zero, which direction are you pointing at?
- If `theta` is instead very close to zero, which direction are you pointing at?
- If `theta` keeps increasing, at what value should you change the direction

My Solution

Here's my solution (in `src/main.rs`):

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

use cortex_m_rt::entry;
use embedded_hal::delay::DelayNs;
use panic_rtt_target as _;
use rtt_target::rtt_init_print;

// You'll find these useful ;-).
use core::f32::consts::PI;
use libm::{atan2f, floorf};

use microbit::{
    display::blocking::Display,
    hal::{twim, Timer},
    pac::twim0::frequency::FREQUENCY_A,
};

use lsm303agr::{Lsm303agr, MagMode, MagOutputDataRate};

#[entry]
fn main() -> ! {
    rtt_init_print!();
    let board = microbit::Board::take().unwrap();

            let   i2c   =   {   twim::Twim::new(board.TWIM0,
board.i2c_internal.into(), FREQUENCY_A::K100) };

    let mut timer0 = Timer::new(board.TIMER0);
    let mut display = Display::new(board.display_pins);
```

```
    let mut sensor = Lsm303agr::new_with_i2c(i2c);
    sensor.init().unwrap();
    sensor
        .set_mag_mode_and_odr(
            &mut timer0,
            MagMode::HighResolution,
            MagOutputDataRate::Hz10,
        )
        .unwrap();
                            let       mut       sensor       =
sensor.into_mag_continuous().ok().unwrap();

    let mut leds = [[0u8; 5]; 5];

     // Indexes of the 16 LEDs to be used in the display, and
their
    // compass directions.
    #[rustfmt::skip]
    let indices = [
        (2, 0), /* W */
        (3, 0), /* W-SW */
        (3, 1), /* SW */
        (4, 1), /* S-SW */
        (4, 2), /* S */
        (4, 3), /* S-SE */
        (3, 3), /* SE */
        (3, 4), /* E-SE */
        (2, 4), /* E */
        (1, 4), /* E-NE */
        (1, 3), /* NE */
        (0, 3), /* N-NE */
        (0, 2), /* N */
        (0, 1), /* N-NW */
        (1, 1), /* NW */
        (1, 0), /* W-NW */
```

```
    ];

    loop {
        while !sensor.mag_status().unwrap().xyz_new_data() {
            timer0.delay_ms(1u32);
        }
                                let    (x,    y,    _)    =
sensor.magnetic_field().unwrap().xyz_nt();

          // Get an angle between -180° and 180° from the x
axis.
        let theta = atan2f(y as f32, x as f32);

        // Cut the unit circle into thirty-two segments,
        // with pairs of adjacent segments corresponding to
        // each compass direction.
        let seg = floorf(16.0 * theta / PI) as i8;

        // Figure out what LED index to blink.
        let index = if seg >= 15 || seg <= -15 {
            8
        } else if seg >= 0 {
            (seg / 2) as usize
        } else {
            ((31 + seg) / 2) as usize
        };

        // Blink the given LED.
        let (r, c) = indices[index];
        leds[r][c] = 255u8;
        display.show(&mut timer0, leds, 50);
        leds[r][c] = 0u8;
        display.show(&mut timer0, leds, 50);
    }
}
```

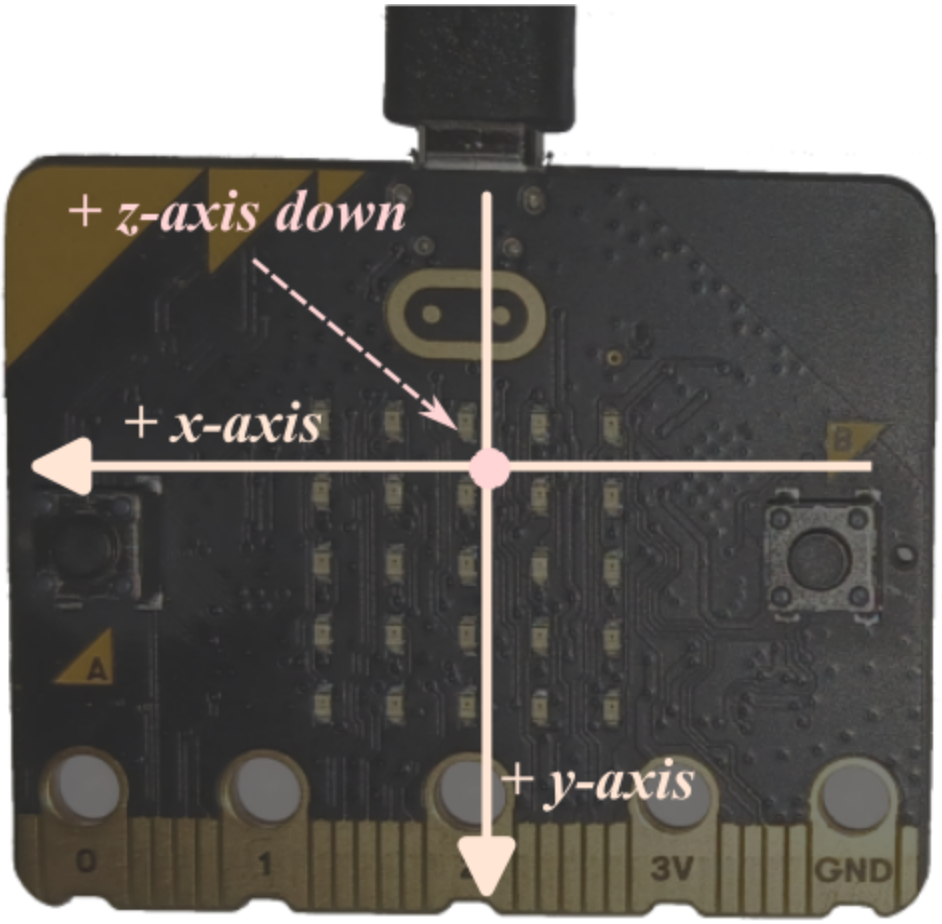
Punch-o-meter

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.

As we already know from previous chapters the accelerometer is built inside the LSM303AGR package. And just like the magnetometer, it is accessible using the I2C bus.

The accelerometer also has the same coordinate system as the magnetometer. Here's a reminder:



Gravity is up?

What's the first thing we'll do?

Perform a sanity check!

You should already be able to write a program that continuously prints the accelerometer data on the RTT console from the I2C chapter. Mine is in `examples/show-accel.rs`. Do you observe something interesting even when holding the board parallel to the floor with the back side facing up? (Remember that the accelerometer is mounted on the back of the board, so holding it upside-down like this makes the Z axis point up.)

What you should see when holding the board like this is that both the X and Y values are rather close to 0, while the Z value is at around 1000. Which is weird: 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` (aha, `1 g` = -1000 from the accelerometer). But the gravity pulls objects downwards so the acceleration along the Z axis should be positive, not negative.

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 an observer that's in free fall. An observer that's in free fall is moving toward the center of 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. Or do, if you're willing to risk your MB2 by dropping it.

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 `rprintln!` macro.

Give it a try and let me know how hard you can punch `;-)`.

NOTE There is an additional API that should be useful for this task that we haven't discussed yet: the `set_accel_scale` one which you need to measure high g values.

My solution

Here's my solution (`src/main.rs`). Note that you can get quite high G values by rapping the edge of your MB2 on a table. Note also that this can break the accelerometer, so probably don't?

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

const TICKS_PER_SEC: u32 = 400;
const THRESHOLD: f32 = 1.5;

use cortex_m::asm::nop;
use cortex_m_rt::entry;
use panic_rtt_target as _;
use rtt_target::{rprintln, rtt_init_print};

use microbit::{
    hal::{twim, Timer},
    pac::twim0::frequency::FREQUENCY_A,
};

use lsm303agr::{AccelMode, AccelOutputDataRate, AccelScale,
Lsm303agr};

#[entry]
fn main() -> ! {
    rtt_init_print!();
    let board = microbit::Board::take().unwrap();

            let i2c = { twim::Twim::new(board.TWIM0,
board.i2c_internal.into(), FREQUENCY_A::K100) };

    let mut delay = Timer::new(board.TIMER0);
```

```
    let mut sensor = Lsm303agr::new_with_i2c(i2c);
    sensor.init().unwrap();
    sensor
        .set_accel_mode_and_odr(&mut delay, AccelMode::Normal,
AccelOutputDataRate::Hz400)
        .unwrap();
     // Allow the sensor to measure up to 16 G since human
punches
    // can actually be quite fast
    sensor.set_accel_scale(AccelScale::G16).unwrap();

    let mut max_g = 0.;
    let mut countdown_ticks = None;

    loop {
        while !sensor.accel_status().unwrap().xyz_new_data() {
            nop();
        }
        // x acceleration in g
                                  let    (x,    _,    _)    =
sensor.acceleration().unwrap().xyz_mg();
        let g_x = x as f32 / 1000.0;

        if let Some(ticks) = countdown_ticks {
            if ticks > 0 {
                // countdown isn't done yet
                if g_x > max_g {
                    max_g = g_x;
                }
                countdown_ticks = Some(ticks - 1);
            } else {
                // Countdown is done: report max value
                rprintln!("Max acceleration: {}g", max_g);

                // Reset
                max_g = 0.;
```

```
                countdown_ticks = None;
            }
        } else {
                // If acceleration goes above a threshold, we
start measuring
            if g_x > THRESHOLD {
                rprintln!("START!");

                max_g = g_x;
                countdown_ticks = Some(TICKS_PER_SEC);
            }
        }
    }
}
```

Snake game

We're now going to implement a basic snake game that you can play on an MB2 using its 5×5 LED matrix as a display and its two buttons as controls. In doing so, we will build on some of the concepts covered in the earlier chapters of this book, and also learn about some new peripherals and concepts.

In particular, we will be using the concept of hardware interrupts to allow our program to interact with multiple peripherals at once. Interrupts are a common way to implement concurrency in embedded contexts. There is a good introduction to concurrency in an embedded context here that you might read through before proceeding.

Modularity

The source code here is more modular than it probably should be. This fine-grained modularity allows us to look at the source code a little at a time. We will build the code bottom-up: we will first build three modules — `game`, `controls` and `display`, and then compose these to build the final program. Each module will have a top-level source file and one or more included source files: for example, the `game` module will consist of `src/game.rs`, `src/game/coords.rs`, `src/game/movement.rs`, etc. The Rust `mod` statement is used to combine the various components of the module. *The Rust Programming Language* has a good description of Rust's module system.

Game logic

The first module we will build is the game logic. You are probably familiar with snake games, but if not, the basic idea is that the player guides a snake around a 2D grid. At any given time, there is some "food" at a random location on the grid and the goal of the game is to get the snake to "eat" as much food as possible. Each time the snake eats food it grows in length. The player loses if the snake crashes into its own tail.

In some variants of the game, the player also loses if the snake crashes into the edge of the grid, but given the small size of our grid we are going to implement a "wraparound" rule: if the snake goes off one edge of the grid, it will continue from the opposite edge.

The `game` module

We will build up the game mechanics in the `game` module.

Coordinates

We start by defining a coordinate system for our game (`src/game/coords.rs`).

```
use super::Prng;

use heapless::FnvIndexSet;

/// A single point on the grid.
#[derive(Debug, Copy, Clone, Eq, PartialEq, Hash)]
pub struct Coords {
      // Signed ints to allow negative values (handy when
checking if we have gone
    // off the top or left of the grid)
    pub row: i8,
    pub col: i8,
}

impl Coords {
    /// Get random coordinates within a grid. `exclude` is an
optional set of
    /// coordinates which should be excluded from the output.
            pub fn random(rng: &mut Prng, exclude:
Option<&FnvIndexSet<Coords, 32>>) -> Self {
        let mut coords = Coords {
            row: ((rng.random_u32() as usize) % 5) as i8,
            col: ((rng.random_u32() as usize) % 5) as i8,
        };
        while exclude.is_some_and(|exc| exc.contains(&coords))
{
            coords = Coords {
                row: ((rng.random_u32() as usize) % 5) as i8,
```

```
                col: ((rng.random_u32() as usize) % 5) as i8,
            }
        }
        coords
    }

    /// Whether the point is outside the bounds of the grid.
    pub fn is_out_of_bounds(&self) -> bool {
          self.row < 0 || self.row >= 5 || self.col < 0 ||
self.col >= 5
    }
}
```

We use a `Coords` struct to refer to a position on the grid. Because `Coords` only contains two integers, we tell the compiler to derive an implementation of the `Copy` trait for it, so we can pass around `Coords` structs without having to worry about ownership.

Random Number Generation

We define an associated function, `Coords::random`, which will give us a random position on the grid. We will use this later to determine where to place the snake's food.

To generate random coordinates, we need a source of random numbers. The nRF52833 has a hardware random number generator (HWRNG) peripheral, documented at section 6.19 of the nRF52833 spec. The HAL gives us a simple interface to the HWRNG via the `microbit::hal::rng::Rng` struct. The HWRNG may not be fast enough for a game; it is also convenient for testing to be able to replicate the sequence of random numbers produced by the generator between runs, which is impossible for the HWRNG by design. We thus also define a pseudo-random number generator (PRNG). The PRNG uses an xorshift algorithm to generate pseudo-random `u32` values. The algorithm is basic and not cryptographically secure, but it is efficient, easy to implement and good enough for our humble snake game. Our `Prng` struct requires an initial seed value, which we do get from the RNG peripheral.

All of this makes up `src/game/rng.rs`.

```
use crate::Rng;

/// A basic pseudo-random number generator.
pub struct Prng {
    value: u32,
}

impl Prng {
    pub fn seeded(rng: &mut Rng) -> Self {
        Self::new(rng.random_u32())
    }

    pub fn new(seed: u32) -> Self {
        Self { value: seed }
    }

            ///   Basic   xorshift   PRNG   function:   see
<https://en.wikipedia.org/wiki/Xorshift>
    fn xorshift32(mut input: u32) -> u32 {
        input ^= input << 13;
        input ^= input >> 17;
        input ^= input << 5;
        input
    }

    /// Return a pseudo-random u32.
    pub fn random_u32(&mut self) -> u32 {
        self.value = Self::xorshift32(self.value);
        self.value
    }
}
```

Movement

We also need to define a few `enum`s that help us manage the game's state: direction of movement, direction to turn, the current game status and the outcome of a particular "step" in the game (ie, a single movement of the snake). `src/game/movement.rs` contains these.

```
use super::Coords;

/// Define the directions the snake can move.
pub enum Direction {
    Up,
    Down,
    Left,
    Right,
}

/// What direction the snake should turn.
#[derive(Debug, Copy, Clone)]
pub enum Turn {
    Left,
    Right,
    None,
}

/// The current status of the game.
pub enum GameStatus {
    Won,
    Lost,
    Ongoing,
}

/// The outcome of a single move/step.
pub enum StepOutcome {
    /// Grid full (player wins)
    Full,
    /// Snake has collided with itself (player loses)
    Collision,
```

```
    /// Snake has eaten some food
    Eat(Coords),
    /// Snake has moved (and nothing else has happened)
    Move(Coords),
}
```

A Snake (*A Snaaake!*)

Next up we define a `Snake` struct, which keeps track of the coordinates occupied by the snake and its direction of travel. We use a queue (`heapless::spsc::Queue`) to keep track of the order of coordinates and a hash set (`heapless::FnvIndexSet`) to allow for quick collision detection. The `Snake` has methods to allow it to move. `src/game/snake.rs` gets this.

```
use super::{Coords, Direction, FnvIndexSet, Turn};

use heapless::spsc::Queue;

pub struct Snake {
    /// Coordinates of the snake's head.
    pub head: Coords,
    /// Queue of coordinates of the rest of the snake's body.
The end of the tail is
    /// at the front.
    pub tail: Queue<Coords, 32>,
    /// A set containing all coordinates currently occupied by
the snake (for fast
    /// collision checking).
    pub coord_set: FnvIndexSet<Coords, 32>,
    /// The direction the snake is currently moving in.
    pub direction: Direction,
}

impl Snake {
    pub fn make_snake() -> Self {
        let head = Coords { row: 2, col: 2 };
        let initial_tail = Coords { row: 2, col: 1 };
```

```
        let mut tail = Queue::new();
        tail.enqueue(initial_tail).unwrap();
              let mut coord_set: FnvIndexSet<Coords, 32> =
FnvIndexSet::new();
        coord_set.insert(head).unwrap();
        coord_set.insert(initial_tail).unwrap();
        Self {
            head,
            tail,
            coord_set,
            direction: Direction::Right,
        }
    }

    /// Move the snake onto the tile at the given coordinates.
If `extend` is false,
    /// the snake's tail vacates the rearmost tile.
    pub fn move_snake(&mut self, coords: Coords, extend: bool)
{
        // Location of head becomes front of tail
        self.tail.enqueue(self.head).unwrap();
        // Head moves to new coords
        self.head = coords;
        self.coord_set.insert(coords).unwrap();
        if !extend {
            let back = self.tail.dequeue().unwrap();
            self.coord_set.remove(&back);
        }
    }

    fn turn_right(&mut self) {
        self.direction = match self.direction {
            Direction::Up => Direction::Right,
            Direction::Down => Direction::Left,
            Direction::Left => Direction::Up,
            Direction::Right => Direction::Down,
```

```
        }
    }

    fn turn_left(&mut self) {
        self.direction = match self.direction {
            Direction::Up => Direction::Left,
            Direction::Down => Direction::Right,
            Direction::Left => Direction::Down,
            Direction::Right => Direction::Up,
        }
    }

    pub fn turn(&mut self, direction: Turn) {
        match direction {
            Turn::Left => self.turn_left(),
            Turn::Right => self.turn_right(),
            Turn::None => (),
        }
    }
}
```

Game Module Top-Level

The `Game` struct keeps track of the game state. It holds a `Snake` object, the current coordinates of the food, the speed of the game (which is used to determine the time that elapses between each movement of the snake), the status of the game (whether the game is ongoing or the player has won or lost) and the player's score.

This struct contains methods to handle each step of the game, determining the snake's next move and updating the game state accordingly. It also contains two methods--`game_matrix` and `score_matrix`--that output 2D arrays of values which can be used to display the game state or the player score on the LED matrix (as we will see later).

We put the `Game` struct at the top of the `game` module, in `src/game.rs`.

```
mod coords;
mod movement;
mod rng;
mod snake;

use crate::Rng;

pub use coords::Coords;
pub use movement::{Direction, GameStatus, StepOutcome, Turn};
pub use rng::Prng;
pub use snake::Snake;

use heapless::FnvIndexSet;

/// Struct to hold game state and associated behaviour
pub struct Game {
    pub status: GameStatus,
    rng: Prng,
    snake: Snake,
    food_coords: Coords,
    speed: u8,
    score: u8,
}

impl Game {
    pub fn new(rng: &mut Rng) -> Self {
        let mut rng = Prng::seeded(rng);
        let snake = Snake::make_snake();
                let  food_coords  =  Coords::random(&mut  rng,
Some(&snake.coord_set));
        Self {
            rng,
            snake,
            food_coords,
            speed: 1,
```

```
            status: GameStatus::Ongoing,
            score: 0,
        }
    }

    /// Reset the game state to start a new game.
    pub fn reset(&mut self) {
        self.snake = Snake::make_snake();
        self.place_food();
        self.speed = 1;
        self.status = GameStatus::Ongoing;
        self.score = 0;
    }

    /// Randomly place food on the grid.
    fn place_food(&mut self) -> Coords {
                let  coords  =  Coords::random(&mut  self.rng,
Some(&self.snake.coord_set));
        self.food_coords = coords;
        coords
    }

        ///  "Wrap  around"  out  of  bounds  coordinates  (eg,
coordinates that are off to the
    /// left of the grid will appear in the rightmost column).
Assumes that
    /// coordinates are out of bounds in one dimension only.
    fn wraparound(&self, coords: Coords) -> Coords {
        if coords.row < 0 {
            Coords { row: 4, ..coords }
        } else if coords.row >= 5 {
            Coords { row: 0, ..coords }
        } else if coords.col < 0 {
            Coords { col: 4, ..coords }
        } else {
            Coords { col: 0, ..coords }
```

```
        }
    }

    /// Determine the next tile that the snake will move on to
(without actually
    /// moving the snake).
    fn get_next_move(&self) -> Coords {
        let head = &self.snake.head;
        let next_move = match self.snake.direction {
            Direction::Up => Coords {
                row: head.row - 1,
                col: head.col,
            },
            Direction::Down => Coords {
                row: head.row + 1,
                col: head.col,
            },
            Direction::Left => Coords {
                row: head.row,
                col: head.col - 1,
            },
            Direction::Right => Coords {
                row: head.row,
                col: head.col + 1,
            },
        };
        if next_move.is_out_of_bounds() {
            self.wraparound(next_move)
        } else {
            next_move
        }
    }

     /// Assess the snake's next move and return the outcome.
Doesn't actually update
    /// the game state.
```

```
    fn get_step_outcome(&self) -> StepOutcome {
        let next_move = self.get_next_move();
        if self.snake.coord_set.contains(&next_move) {
            // We haven't moved the snake yet, so if the next
move is at the end of
            // the tail, there won't actually be any collision
(as the tail will have
            // moved by the time the head moves onto the tile)
            if next_move != *self.snake.tail.peek().unwrap() {
                StepOutcome::Collision
            } else {
                StepOutcome::Move(next_move)
            }
        } else if next_move == self.food_coords {
            if self.snake.tail.len() == 23 {
                StepOutcome::Full
            } else {
                StepOutcome::Eat(next_move)
            }
        } else {
            StepOutcome::Move(next_move)
        }
    }

     /// Handle the outcome of a step, updating the game's
internal state.
    fn handle_step_outcome(&mut self, outcome: StepOutcome) {
        self.status = match outcome {
            StepOutcome::Collision => GameStatus::Lost,
            StepOutcome::Full => GameStatus::Won,
            StepOutcome::Eat(c) => {
                self.snake.move_snake(c, true);
                self.place_food();
                self.score += 1;
                if self.score % 5 == 0 {
                    self.speed += 1
```

```
                }
                GameStatus::Ongoing
            }
            StepOutcome::Move(c) => {
                self.snake.move_snake(c, false);
                GameStatus::Ongoing
            }
        }
    }

    pub fn step(&mut self, turn: Turn) {
        self.snake.turn(turn);
        let outcome = self.get_step_outcome();
        self.handle_step_outcome(outcome);
    }

     /// Calculate the length of time to wait between game steps, in milliseconds.
     /// Generally this will get lower as the player's score increases, but need to
    /// be careful it cannot result in a value below zero.
    pub fn step_len_ms(&self) -> u32 {
        let result = 1000 - (200 * ((self.speed as i32) - 1));
        if result < 200 {
            200u32
        } else {
            result as u32
        }
    }

    /// Return an array representing the game state, which can be used to display the
    /// state on the microbit's LED matrix. Each `_brightness` parameter should be a
    /// value between 0 and 9.
    pub fn game_matrix(
```

```
        &self,
        head_brightness: u8,
        tail_brightness: u8,
        food_brightness: u8,
    ) -> [[u8; 5]; 5] {
        let mut values = [[0u8; 5]; 5];
                        values[self.snake.head.row   as   usize]
[self.snake.head.col as usize] = head_brightness;
        for t in &self.snake.tail {
                    values[t.row  as  usize][t.col  as  usize]  =
tail_brightness
        }
                       values[self.food_coords.row   as   usize]
[self.food_coords.col as usize] = food_brightness;
        values
    }

    /// Return an array representing the game score, which can
be used to display the
     /// score on the microbit's LED matrix (by illuminating
the equivalent number of
    /// LEDs, going left->right and top->bottom).
    pub fn score_matrix(&self) -> [[u8; 5]; 5] {
        let mut values = [[0u8; 5]; 5];
        let full_rows = (self.score as usize) / 5;
        #[allow(clippy::needless_range_loop)]
        for r in 0..full_rows {
            values[r] = [1; 5];
        }
        for c in 0..(self.score as usize) % 5 {
            values[full_rows][c] = 1;
        }
        values
    }
}
```

Next we will add the ability to control the snake's movements.

Controls

Our protagonist will be controlled by the two buttons on the front of the micro:bit. Button A will turn to the snake's left, and button B will turn to the snake's right.

We will use the `microbit::pac::interrupt` macro to handle button presses in a concurrent way. The interrupt will be generated by the MB2's General Purpose Input/Output Tasks and Events (GPIOTE) peripheral.

The `controls` module

We will need to keep track of two separate pieces of global mutable state: A reference to the `GPIOTE` peripheral, and a record of the selected direction to turn next.

Shared data is wrapped in a `RefCell` to permit interior mutability and locking. You can learn more about `RefCell` by reading the RefCell documentation and the interior mutability chapter of the Rust Book]. The `RefCell` is, in turn, wrapped in a `cortex_m::interrupt::Mutex` to allow safe access. The Mutex provided by the `cortex_m` crate uses the concept of a critical section. Data in a Mutex can only be accessed from within a function or closure passed to `cortex_m::interrupt:free` (renamed here to `interrupt_free` for clarity), which ensures that the code in the function or closure cannot itself be interrupted.

Initialization

First, we will initialise the buttons (`src/controls/init.rs`).

```
use super::{Buttons, GPIO};

use cortex_m::interrupt::free as interrupt_free;
use microbit::{
    hal::{
        gpio::{Floating, Input, Pin},
        gpiote::{Gpiote, GpioteChannel},
    },
    pac,
};

/// Initialise the buttons and enable interrupts.
pub fn init_buttons(board_gpiote: pac::GPIOTE, board_buttons:
Buttons) {
    let gpiote = Gpiote::new(board_gpiote);

        fn init_channel(channel: &GpioteChannel<'_>, button:
```

```
&Pin<Input<Floating>>) {

channel.input_pin(button).hi_to_lo().enable_interrupt();
        channel.reset_events();
    }

    let channel0 = gpiote.channel0();
                                        init_channel(&channel0,
&board_buttons.button_a.degrade());

    let channel1 = gpiote.channel1();
                                        init_channel(&channel1,
&board_buttons.button_b.degrade());

    interrupt_free(move |cs| {
        *GPIO.borrow(cs).borrow_mut() = Some(gpiote);

        unsafe {
            pac::NVIC::unmask(pac::Interrupt::GPIOTE);
        }
        pac::NVIC::unpend(pac::Interrupt::GPIOTE);
    });
}
```

The `GPIOTE` peripheral on the nRF52 has 8 "channels", each of which can be connected to a `GPIO` pin and configured to respond to certain events, including rising edge (transition from low to high signal) and falling edge (high to low signal). A button is a `GPIO` pin which has high signal when not pressed and low signal otherwise. Therefore, a button press is a falling edge.

Note the awkward use of the function `init_channel()` in initialization to avoid copy-pasting the button initialization code. The types that the various embedded crates for the MB2 have been hiding from you are sometimes a bit scary. I would encourage you to explore the type structure of the HAL and PAC crates at some point, as it is a bit odd and takes getting used to. In particular, note that each pin on the microbit has *its own unique*

type. The purpose of the `degrade()` function in initialization is to convert these to a common type that can reasonably be used as an argument to `init_channel()` and thence to `input_pin()`.

We connect `channel0` to `button_a` and `channel1` to `button_b`. In each case, we set the button up to generate events on a falling edge (`hi_to_lo`). We store a reference to our `GPIOTE` peripheral in the `GPIO` Mutex. We then `unmask` `GPIOTE` interrupts, allowing them to be propagated by the hardware, and call `unpend` to clear any interrupts with pending status (which may have been generated prior to the interrupts being unmasked).

Interrupt handler

Next, we write the code that handles the interrupt. We use the `interrupt` macro re-exported from the `nrf52833_hal` crate. We define a function with the same name as the interrupt we want to handle (you can see them all here) and annotate it with `#[interrupt]` (`src/controls/interrupt.rs`).

```
use super::{Turn, GPIO, TURN};

use cortex_m::interrupt::free as interrupt_free;
use microbit::pac::{self, interrupt};

#[pac::interrupt]
fn GPIOTE() {
    interrupt_free(|cs| {
                                    if    let    Some(gpiote)    =
GPIO.borrow(cs).borrow().as_ref() {
                                              let    a_pressed    =
gpiote.channel0().is_event_triggered();
                                              let    b_pressed    =
gpiote.channel1().is_event_triggered();

            let turn = match (a_pressed, b_pressed) {
                (true, false) => Turn::Left,
```

```
                (false, true) => Turn::Right,
                _ => Turn::None,
            };

            gpiote.channel0().reset_events();
            gpiote.channel1().reset_events();

            *TURN.borrow(cs).borrow_mut() = turn;
        }
    });
}
```

When a `GPIOTE` interrupt is generated, we check each button to see whether it has been pressed. If only button A has been pressed, we record that the snake should turn to the left. If only button B has been pressed, we record that the snake should turn to the right. In any other case, we record that the snake should not make any turn. (Having both buttons pressed "at the same time" is exceedingly unlikely: button presses are noted almost instantly, and this interrupt handler runs very fast — it would be hard to get both buttons down in time for this to happen. Similarly, it would be hard to press a button for a short enough time for this code to miss it and report that neither button is pressed. Still, Rust enforces that you plan for these unexpected cases: the code will not compile unless you check all the possibilities.) The relevant turn is stored in the `TURN` Mutex. All of this happens within an `interrupt_free` block, to ensure that we cannot be interrupted by some other event while handling this interrupt.

Finally, we expose a simple function to get the next turn (`src/controls.rs`).

```
mod init;
mod interrupt;

pub use init::init_buttons;

use crate::game::Turn;
use core::cell::RefCell;
```

```
use cortex_m::interrupt::{free as interrupt_free, Mutex};
use microbit::{board::Buttons, hal::gpiote::Gpiote};
pub    static    GPIO:    Mutex<RefCell<Option<Gpiote>>>    =
Mutex::new(RefCell::new(None));
pub       static       TURN:       Mutex<RefCell<Turn>>       =
Mutex::new(RefCell::new(Turn::None));

/// Get the next turn (ie, the turn corresponding to the most
recently pressed button).
pub fn get_turn(reset: bool) -> Turn {
    interrupt_free(|cs| {
        let turn = *TURN.borrow(cs).borrow();
        if reset {
            *TURN.borrow(cs).borrow_mut() = Turn::None
        }
        turn
    })
}
```

This function simply returns the current value of the `TURN` Mutex. It takes a single boolean argument, `reset`. If `reset` is `true`, the value of `TURN` is reset, i.e., set to `Turn::None`.

Next we will build support for a high-fidelity game display.

Using the non-blocking display

We will next display the snake and food on the LEDs of the MB2 screen. So far, we have used the blocking interface, which provides for LEDs to be either maximally bright or turned off. With this, a basic functioning snake game would be possible. But you might find that when the snake got a bit longer, it would be difficult to tell the snake from the food, and to tell which direction the snake was heading. Let's figure out how to allow the LED brightness to vary: we can make the snake's body a bit dimmer, which will help sort out the clutter.

The `microbit` library makes available two different interfaces to the LED matrix. There is the blocking interface we've already seen in previous chapters. There is also a non-blocking interface which allows you to customise the brightness of each LED. At the hardware level, each LED is either "on" or "off", but the `microbit::display::nonblocking` module simulates ten levels of brightness for each LED by rapidly switching the LED on and off.

(There is no great reason the two display modes of the `microbit` library crate have to be separate and use separate code. A more complete design would allow either non-blocking or blocking use of a single display API with variable brightness levels and refresh rates specified by the user. Never assume that the stuff you have been handed is perfected, or even close. Always think about what you might do differently. For now, though, we'll work with what we have, which is adequate for our immediate purpose.)

The code to interact with the non-blocking interface (`src/display.rs`) is pretty simple and will follow a similar structure to the code we used to interact with the buttons. This time we'll start at the top level.

Display module

```
pub mod interrupt;
pub mod show;

pub use show::{clear_display, display_image};

use core::cell::RefCell;
use cortex_m::interrupt::{free as interrupt_free, Mutex};
use microbit::display::nonblocking::Display;
use microbit::gpio::DisplayPins;
use microbit::pac;
use microbit::pac::TIMER1;

static DISPLAY: Mutex<RefCell<Option<Display<TIMER1>>>> =
Mutex::new(RefCell::new(None));

pub fn init_display(board_timer: TIMER1, board_display:
DisplayPins) {
    let display = Display::new(board_timer, board_display);

    interrupt_free(move |cs| {
        *DISPLAY.borrow(cs).borrow_mut() = Some(display);
    });
    unsafe { pac::NVIC::unmask(pac::Interrupt::TIMER1) }
}
```

First, we initialise a `microbit::display::nonblocking::Display` struct representing the LED display, passing it the board's `TIMER1` and `DisplayPins` peripherals. Then we store the display in a Mutex. Finally, we unmask the `TIMER1` interrupt.

Display API

We then define a couple of convenience functions which allow us to easily set (or unset) the image to be displayed (`src/display/show.rs`).

```
use super::DISPLAY;

use cortex_m::interrupt::free as interrupt_free;

use tiny_led_matrix::Render;

/// Display an image.
pub fn display_image(image: &impl Render) {
    interrupt_free(|cs| {
                                if    let    Some(display)    =
DISPLAY.borrow(cs).borrow_mut().as_mut() {
            display.show(image);
        }
    })
}

/// Clear the display (turn off all LEDs).
pub fn clear_display() {
    interrupt_free(|cs| {
                                if    let    Some(display)    =
DISPLAY.borrow(cs).borrow_mut().as_mut() {
            display.clear();
        }
    })
}
```

`display_image` takes an image and tells the display to show it. Like the `Display::show` method that it calls, this function takes a struct that implements the `tiny_led_matrix::Render` trait. That trait ensures that the struct contains the data and methods necessary for the `Display` to render it on the LED matrix. The two implementations of `Render` provided by the

`microbit::display::nonblocking` module are `BitImage` and `GreyscaleImage`. In a `BitImage`, each "pixel" (or LED) is either illuminated or not (like when we used the blocking interface), whereas in a `GreyscaleImage` each "pixel" can have a different brightness.

`clear_display` does exactly as the name suggests.

Display interrupt handling

Finally, we use the `interrupt` macro to define a handler for the `TIMER1` interrupt. This interrupt fires many times a second, and this is what allows the `Display` to rapidly cycle the different LEDs on and off to give the illusion of varying brightness levels. All our handler code does is call the `Display::handle_display_event` method, which handles this (`src/display/interrupt.rs`).

```
use super::DISPLAY;

use cortex_m::interrupt::free as interrupt_free;
use microbit::pac::{self, interrupt};

#[pac::interrupt]
fn TIMER1() {
    interrupt_free(|cs| {
                                if    let    Some(display)    =
DISPLAY.borrow(cs).borrow_mut().as_mut() {
            display.handle_display_event();
        }
    })
}
```

Now we can understand how our `main` function will do display: we will call `init_display` and use the new functions we have defined to interact with it.

Snake game: final assembly

The code in our `src/main.rs` file brings all the previously-discussed machinery together to make our final game.

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

mod controls;
mod display;
pub mod game;

use controls::{get_turn, init_buttons};
use display::{clear_display, display_image, init_display};
use game::{Game, GameStatus};

use cortex_m_rt::entry;
use embedded_hal::delay::DelayNs;
use microbit::{
    display::nonblocking::{BitImage, GreyscaleImage},
    hal::{Rng, Timer},
    Board,
};
use panic_rtt_target as _;
use rtt_target::rtt_init_print;

#[entry]
fn main() -> ! {
    rtt_init_print!();
    let board = Board::take().unwrap();
    let mut timer = Timer::new(board.TIMER0).into_periodic();
    let mut rng = Rng::new(board.RNG);
    let mut game = Game::new(&mut rng);

    init_buttons(board.GPIOTE, board.buttons);
```

```
    init_display(board.TIMER1, board.display_pins);

    loop {
        loop {
            // Game loop
                                                let    image    =
GreyscaleImage::new(&game.game_matrix(6, 3, 9));
            display_image(&image);
            timer.delay_ms(game.step_len_ms());
            match game.status {
                                          GameStatus::Ongoing  =>
game.step(get_turn(true)),
                _ => {
                    for _ in 0..3 {
                        clear_display();
                        timer.delay_ms(200u32);
                        display_image(&image);
                        timer.delay_ms(200u32);
                    }
                    clear_display();

display_image(&BitImage::new(&game.score_matrix()));
                    timer.delay_ms(2000u32);
                    break;
                }
            }
        }
        game.reset();
    }
}
```

After initialising the board and its timer and RNG peripherals, we initialise a `Game` struct and a `Display` from the `microbit::display::blocking` module.

In our "game loop" (which runs inside of the "main loop" we place in our `main` function), we repeatedly perform the following steps:

1. Get a 5×5 array of bytes representing the grid. The `Game::get_matrix` method takes three integer arguments (which should be between 0 and 9, inclusive) which will, eventually, represent how brightly the head, tail and food should be displayed.
2. Display the matrix, for an amount of time determined by the `Game::step_len_ms` method. As currently implemented, this method basically provides for 1 second between steps, reducing by 200ms every time the player scores 5 points (eating 1 piece of food = 1 point), subject to a floor of 200ms.
3. Check the game status. If it is `Ongoing` (which is its initial value), run a step of the game and update the game state (including its `status` property). Otherwise, the game is over, so flash the current image three times, then show the player's score (represented as a number of illuminated LEDs corresponding to the score), and exit the game loop.

Our main loop just runs the game loop repeatedly, resetting the game's state after each iteration.

What's left for you to explore

We have barely scratched the surface! There's lots of stuff left for you to explore.

> **NOTE:** If you're reading this, and you'd like to help add examples or exercises to the Discovery book for any of the items below, or any other relevant embedded topics, we'd love to have your help!
>
> Please open an issue if you would like to help, but need assistance or mentoring for how to contribute this to the book, or open a Pull Request adding the information!

Topics about embedded software

These topics discuss strategies for writing embedded software. Although many problems can be solved in different ways, these sections talk about some strategies, and when they make sense (or don't make sense) to use.

Multitasking

Most of 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*. We were introduced to interrupts when we built our snake game in chapter 14.

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.

Sleeping

All our programs have been continuously polling peripherals to see if there's anything that needs to be done. However, sometimes 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. The ARM machine

instructions `wfi` and `wfe` are the instructions that make the processor "sleep" waiting for an interrupt or event.

Topics related to microcontroller capabilities

Microcontrollers (like our nRF52/nRF51) have many capabilities. However, many share similar capabilities that can be used to solve all sorts of different problems.

These topics discuss some of those capabilities, and how they can be used effectively in embedded development.

Direct Memory Access (DMA).

Some peripherals have DMA, a kind of *asynchronous* `memcpy` that allows the peripheral to move data into or out of memory without the CPU being involved.

If you are working with a micro:bit v2 you have actually already used DMA: the HAL does this for you with the UARTE and TWIM peripherals. A DMA peripheral can be used to perform bulk transfers of data: either from RAM to RAM, from a peripheral like a UARTE, to RAM, or from RAM to a peripheral. You can schedule a DMA transfer — for example "read 256 bytes from UARTE into this buffer" — and leave it running in the background. You can check some register later to see if the transfer has completed, or you can ask to receive an interrupt when the transfer completes. Thus, you can schedule the DMA transfer and do other work while the transfer is ongoing.

The details of low-level DMA can be a bit tricky. We hope to add a chapter covering this topic in the near future.

Interrupts

We saw button interrupts briefly in chapter 14. This introduced the key idea: in order to interact with the real world, it is often necessary for the microcontroller to respond *immediately* when some kind of event occurs.

Microcontrollers have the ability to be interrupted, meaning when a certain event occurs, it will stop whatever it is doing at the moment, to instead respond to that event. This can be very useful when we want to stop a motor when a button is pressed, or measure a sensor when a timer finishes counting down.

Although these interrupts can be very useful, they can also be a bit difficult to work with properly. We want to make sure that we respond to events quickly, but also allow other work to continue as well.

In Rust, we model interrupts similar to the concept of threading on desktop Rust programs. This means we also must think about the Rust concepts of `Send` and `Sync` when sharing data between our main application, and code that executes as part of handling an interrupt event.

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.

There are some abstraction for working with PWM in the `embedded-hal` `pwm` module and you will find implementations of these traits in `nrf52833-hal`.

Digital inputs and outputs

We have used the microcontroller pins as digital outputs, to drive LEDs. When building our snake game, we also caught a glimpse of how these pins can be configured as digital inputs. As digital inputs, these pins can read the binary state of switches (on/off) or buttons (pressed/not pressed).

Digital inputs and outputs are abstracted within the `embedded-hal` `digital` module and [`nrf52833-hal`] does have an implementation for

them.

(*spoilers* reading the binary state of switches / buttons is not as straightforward as it sounds ;-))

Analog-to-Digital Converters (ADC)

There are a lot 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 reading to the CPU of the voltage they are sensing at an ADC input pin.

The ADC peripheral can thus be used to measure an "analog" voltage level — for example, `1.25` Volts — as a "digital" number — for example, `24824` — that the processor can use in its calculations.

There were generic ADC traits in `embedded-hal`, but they were removed for `embedded-hal` 1.0: see issue #377. The `nrf52833-hal` crate provides a nice interface to the specific ADC built into the nRF52833.

Digital-to-Analog Converters (DAC)

As you might expect a DAC is exactly the opposite of ADC. You can write some digital number into a register to produce a specific voltage on some analog output pin. When this analog output pin is connected to some appropriate electronics and the register is written to quickly with the right values you can do things like produce sounds or music.

Neither the nRF52833 nor the MB2 board has a dedicated DAC. One typically gets a kind of DAC effect by outputting PWM and using a bit of electronics on the output (RC filter) to "smooth" out the PWM waveform.

Real Time Clock

A Real-Time Clock peripheral keeps track of time under its own power, usually in "human format": seconds, minutes, hours, days, months and years. Some Real-Time Clocks even handle leap years and Daylight Saving Time automatically.

Neither the nRF52833 nor the MB2 board contains a Real-Time Clock. The nRF52833 does contain "Real-Time Counter" (RTC), a low-frequency ticking clock that is supported by `nrf52833-hal`. This counter can be

dedicated to serve as a synthesized real-time clock. The key requirement, of course, is to keep the RTC peripheral powered even when the MB2 is not in use. While the MB2 does not have an on-board battery, the RTC should be able to run for a long time (possibly years) with a battery plugged into the battery port on the MB2 (for example, the battery pack provided with the micro::bit Go kit).

Other communication protocols

- I2C: discussed in earlier chapters of this book
- SPI: abstracted within the `embedded-hal` `spi` module and implemented by the [`nrf52-hal`]
- I2S: currently not abstracted within the `embedded-hal` but implemented by the [`nrf52-hal`]
- Ethernet: there does exist a small TCP/IP stack named `smoltcp` which is implemented for some chips. The MB2 does not have an Ethernet peripheral
- USB: there is some experimental work on this, for example with the `usb-device` crate
- Bluetooth: the `nrf-softdevice` wrapper provided by the `Embassy` MB2 runtime is probably the easiest entry into MB2 Bluetooth right now
- CAN, SMBUS, IrDA, etc: All kinds of specialty interfaces exist in the world; Rust sometimes has support for them. Please investigate the current situation for the interface you need

Different applications use different communication protocols. User facing applications usually have a USB connector because USB is a ubiquitous protocol in PCs and smartphones. Whereas inside cars you'll find plenty of CAN buses. Some digital sensors use SPI, I2C or SMBUS.

If you happen to be interested in developing abstractions in the `embedded-hal` or implementations of peripherals in general, don't be shy to open an issue in the HAL repositories. Alternatively you could also join the Rust Embedded matrix channel and get into contact with most of the people who built the stuff from above.

General Embedded-Relevant Topics

These topics cover items that are not specific to our device, or the hardware on it. Instead, they discuss useful techniques that could be used on embedded systems. Most of what we will discuss here is not available on the MB2 — but most of it could easily be added by connecting a cheap piece of hardware to the MB2 edge-card connector, either driving it directly or using something like SPI or I2C to control it.

Gyroscopes

As part of our Punch-o-meter exercise, we used the Accelerometer to measure changes in acceleration in three dimensions. But there are other motion sensors such as gyroscopes, which allows us to measure changes in "spin" in three dimensions.

This can be very useful when trying to build certain systems, such as a robot that wants to avoid tipping over. Additionally, the data from a sensor like a gyroscope can also be combined with data from accelerometer using a technique called Sensor Fusion (see below for more information).

Servo and Stepper Motors

While some motors are used primarily just to spin in one direction or the other, for example driving a remote control car forwards or backwards, it is sometimes useful to measure more precisely how a motor rotates.

A microcontroller can be used to drive Servo or Stepper motors, which allow for more precise control of how many turns are being made by the motor, or can even position the motor in one specific place, for example if we wanted to move the arms of a clock to a particular direction.

Sensor fusion

The micro:bit contains two motion sensors: an accelerometer and a magnetometer. On their own these measure (proper) acceleration 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, with less measurement error than that of any single sensor.

This idea of deriving more reliable data from different sources is known as sensor fusion.

So where to next?

First and foremost, join us on the Rust Embedded matrix channel. Lots of people who contribute or work on embedded software hang out there, including, for example, the people who wrote the `microbit` BSP, the `nrf52-hal` crate, the `embedded-hal` crates, etc. We are happy to help you get started or move on with embedded programming in Rust!

There are many other options:

- You could check out the examples in the `microbit-v2` board support crate. All those examples work for the micro:bit board you have.
- If you are looking for a general overview of what is available in Rust Embedded right now check out the Awesome Rust Embedded list.
- You could check out Embassy. This is a modern efficient cooperative multitasking framework that supports concurrent execution using Rust `async/await`.
- You could check out Real-Time Interrupt-driven Concurrency RTIC. RTIC is a very efficient preemptive multitasking framework that supports task prioritization and deadlock-free execution.
- You could check out more abstractions of the `embedded-hal` project and maybe even try and write your own platform agnostic driver based on it.
- 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.

General troubleshooting

`cargo-embed` problems

Most `cargo-embed` problems are related to not having installed the `udev` rules properly on Linux, so make sure you got that right.

If you are stuck, you can open an issue in the `discovery` issue tracker or visit the Rust Embedded matrix channel or the probe-rs matrix channel and ask for help there.

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 forgot to install the proper target for your microcontroller `thumbv7em-none-eabihf`.

Fix:

Install the proper target.

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

How to use GDB

Below are some useful GDB commands that can help us debug our programs. This assumes you have flashed a program onto your microcontroller and attached GDB to a `cargo-embed` session.

General Debugging

NOTE: Many of the commands you see below can be executed using a short form. For example, `continue` can simply be used as `c`, or `break $location` can be used as `b $location`. Once you have experience with the commands below, try to see how short you can get the commands to go before GDB doesn't recognize them!

Dealing with Breakpoints

- `break $location`: Set a breakpoint at a place in your code. The value of `$location` can include:
    - `break *main` - Break on the exact address of the function `main`
    - `break *0x080012f2` - Break on the exact memory location `0x080012f2`
    - `break 123` - Break on line 123 of the currently displayed file
    - `break main.rs:123` - Break on line 123 of the file `main.rs`
- `info break`: Display current breakpoints
- `delete`: Delete all breakpoints
    - `delete $n`: Delete breakpoint `$n` (`n` being a number. For example: `delete $2`)
- `clear`: Delete breakpoint at next instruction
    - `clear main.rs:$function`: Delete breakpoint at entry of `$function` in `main.rs`
    - `clear main.rs:123`: Delete breakpoint on line 123 of `main.rs`
- `enable`: Enable all set breakpoints
    - `enable $n`: Enable breakpoint `$n`
- `disable`: Disable all set breakpoints
    - `disable $n`: Disable breakpoint `$n`

Controlling Execution

- `continue`: Begin or continue execution of your program
- `next`: Execute the next line of your program
    - `next $n`: Repeat `next` `$n` number times
- `nexti`: Same as `next` but with machine instructions instead
- `step`: Execute the next line, if the next line includes a call to another function, step into that code
    - `step $n`: Repeat `step` `$n` number times
- `stepi`: Same as `step` but with machine instructions instead
- `jump $location`: Resume execution at specified location:
    - `jump 123`: Resume execution at line 123
    - `jump 0x080012f2`: Resume execution at address 0x080012f2

Printing Information

- `print /$f $data` - Print the value contained by the variable `$data`. Optionally format the output with `$f`, which can include:

```
x: hexadecimal
d: signed decimal
u: unsigned decimal
o: octal
t: binary
a: address
c: character
f: floating point
```

    - `print /t 0xA`: Prints the hexadecimal value `0xA` as binary (0b1010)
- `x /$n$u$f $address`: Examine memory at `$address`. Optionally, `$n` define the number of units to display, `$u` unit size (bytes,

halfwords, words, etc.), `$f` any `print` format defined above

    - `x /5i 0x080012c4`: Print 5 machine instructions staring at address `0x080012c4`
    - `x/4xb $pc`: Print 4 bytes of memory starting where `$pc` currently is pointing
- `disassemble $location`

    - `disassemble /r main`: Disassemble the function `main`, using `/r` to show the bytes that make up each instruction

Looking at the Symbol Table

- `info functions $regex`: Print the names and data types of functions matched by `$regex`, omit `$regex` to print all functions

    - `info functions main`: Print names and types of defined functions that contain the word `main`
- `info address $symbol`: Print where `$symbol` is stored in memory

    - `info address GPIOC`: Print the memory address of the variable `GPIOC`
- `info variables $regex`: Print names and types of global variables matched by `$regex`, omit `$regex` to print all global variables
- `ptype $data`: Print more detailed information about `$data`

    - `ptype cp`: Print detailed type information about the variable `cp`

Poking around the Program Stack

- `backtrace $n`: Print trace of `$n` frames, or omit `$n` to print all frames

    - `backtrace 2`: Print trace of first 2 frames

- `frame $n`: Select frame with number or address `$n`, omit `$n` to display current frame
- `up $n`: Select frame `$n` frames up
- `down $n`: Select frame `$n` frames down
- `info frame $address`: Describe frame at `$address`, omit `$address` for currently selected frame
- `info args`: Print arguments of selected frame
- `info registers $r`: Print the value of register `$r` in selected frame, omit `$r` for all registers
    - `info registers $sp`: Print the value of the stack pointer register `$sp` in the current frame

Controlling `cargo-embed` Remotely

- `monitor reset`: Reset the CPU, starting execution over again

Magnetometer Calibration

One very important thing to do before using a sensor and trying to develop an application using it is verifying that it's output is actually correct. If this does not happen to be the case we need to calibrate the sensor. Alternatively the sensor could be broken: health-checking sensors before and during use is a really good idea when possible.

In my case, on two different MB2s the LSM303AGR's magnetometer without calibration is quite a bit off. (I also have one where the z-axis appears to be broken; the manufacturer has some extra hardware and a process to help detect this, but we won't deal with that complexity here.)

There is a manufacturer-specified procedure for calibrating the magnetometer. The calibration involves quite a bit of math (matrices) so we won't cover it in detail here: this Design Note describes the procedure if you are interested in the details.

Luckily for us, the CODAL group that built the original C++ software for the micro:bit already implemented the manufacturer calibration mechanism (or something similar) in C++ over here.

You can find a translation of this C++ calibration to Rust in `src/lib.rs`. Note that this is a translation from Matlab to C++ to Rust, and that it makes some interesting choices. In particular, when reading calibrated values *the axes are flipped* so that viewed from the top with the USB connector forward the X, Y and Z axes of the calibrated value are in "standard" (right, forward, up) orientation.

The usage of this calibrator is demonstrated in `src/main.rs` here.

The way the user does the calibration is shown in this video from the C++ version. (Ignore the initial printing — the calibration starts about halfway through.)

You have to tilt the micro:bit until all the LEDs on the LED matrix light up. The blinking cursor shows the current target LED.

Note that the calibration matrix is printed by the demo program. This matrix can be hard-coded into a program such as the chapter 12 compass program (or stored in flash somewhere somehow) to avoid the need to recalibrate every time the user runs the program.