

The micro:bit, Properly

Every part of the board, driven from Rust. A picture first, then a program you run, then the explanation, and the questions last.

14 chapters **13** programs, type-checked Built for teaching it on

CHAPTER 0

How to use this book

YOU ALREADY KNOW

You have a micro:bit, a laptop, and a first program that already ran on the board.

YOU WILL BE ABLE TO

Know how each chapter works, and what to trust and what to check.

This book teaches you to drive every part of the micro:bit from Rust, until you can do it from memory and teach it to someone else.

PICTURE THIS

A good mechanic does not learn a vehicle from the manual. He learns it by driving it. He turns the key, sees what happens, then opens the bonnet to find out why. After a week he can find any part with his eyes closed.

This book works the same way. Every chapter starts with a picture from the workshop. Then you run a program and something real happens on the board. Only then do we open the bonnet and explain why. The questions come last, and they only ask about things the chapter has already told you.

How a chapter runs

1. **A picture.** A workshop story, so you can see the idea before any jargon.
2. **A program.** A short Rust file, ready in your project folder. You run it and watch the board.
3. **The explanation.** What each part of the program does, in the order you read it.
4. **From memory.** A small job to rebuild the program with the book closed. This is how it sticks.
5. **Check yourself.** Questions you can answer from the chapter, and no trick questions.

What you can trust, and what you should check

Every program in this book was run through the Rust compiler's type checker, against the real libraries for the micro:bit, before it was put on the page. That catches mistakes in the way the code is written, such as a wrong name, a missing import or a mismatched type.

It does **not** prove the board will behave as described. The compiler that builds the final file for the chip, the programmer and the board were not in the room. So each program is honestly labelled, and if something does not behave as described, that is information. Tell me what you saw and we fix it.

Some programs are adapted from the examples in the *micro:bit v2 Embedded Discovery Book*, a free book by the Rust Embedded project. Those programs are marked. The explanations, pictures and questions are written for this book.

The chapters

#	PICTURE	WHAT IT TEACHES
1	The engine on the stand	The board, the chip and its memory
2	Your workbench	The project folder, and how a program gets onto the board
3	The wall of windows	Writing on the 25 LEDs
4	The paper strip	Scrolling numbers
5	The doorbell	Reading the two buttons
6	The indicator stalk	A car's turn signal and hazard lights
7	The stopwatch	Time, delays and a steady blink
8	The speaker cone	Notes and a scale
9	Talking to the laptop	Printing with RTT and the serial port
10	The spirit level	The tilt sensor
11	The thermometer	The chip's own temperature
12	The outside world	The pins on the edge
13	What is not here yet	The honest list

SAY IT BACK

- Picture, program, explanation, from memory, then the questions.
- A program that passes the type-check is written correctly. Only the board tells you what it does.

CHECK YOURSELF

1. In what order does each chapter go, and where do the questions sit?
2. What does the type-check prove, and what does it not prove?

The engine on the stand

YOU ALREADY KNOW

A micro:bit is a small board with lights and buttons on it.

YOU WILL BE ABLE TO

Point at the engine, the second chip and the speaker, and say what memory the engine has.

PICTURE THIS

Bolt a small engine to a stand in the corner of the workshop. It has no seats and no steering wheel. It has one job, and it does that job the moment power arrives, every time, and it never asks a question.

That engine is your micro:bit. The thinking part is one small chip called a **microcontroller**. Everything else on the board is there to help that chip see, press, light up and make noise.

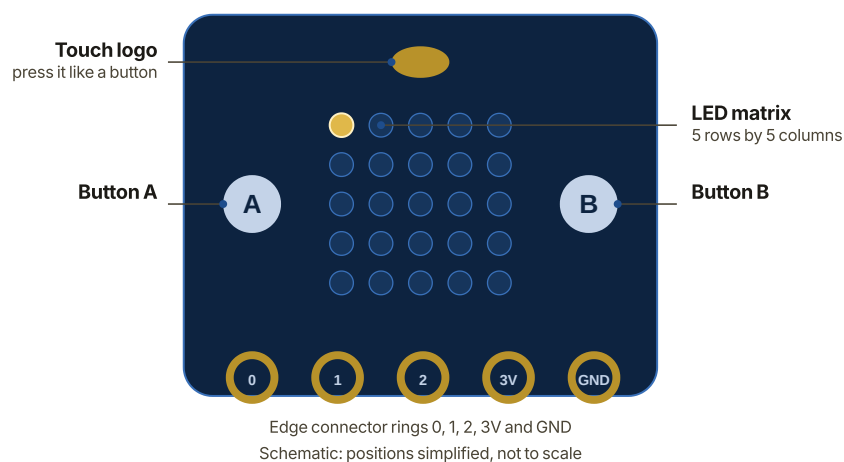


Figure 1.1 The front of the micro:bit v2. Everything you touch and see is here.

On the front you can see the touch logo, the grid of 25 lamps, the two buttons A and B, and the gold rings along the bottom edge. The rings are where wires can be clipped on.

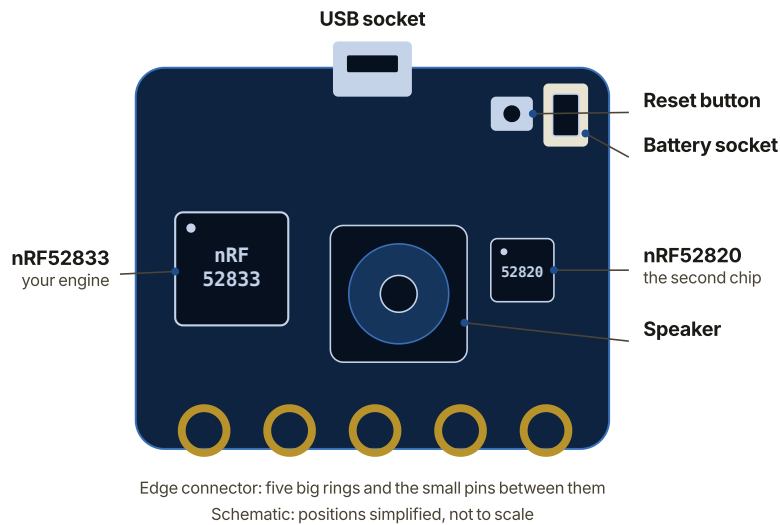


Figure 1.2 The back. The chips live here. There are three black squares: the speaker, the engine, and the second chip.

On the back are three black squares. The biggest one is the speaker. The next is the engine, the chip called the nRF52833, which runs the programs you write. The smallest, the nRF52820, is a second chip that looks after power and reset, carries the serial text to your laptop, and acts as the programmer that puts your program into the engine.

Reading the number plate

Every engine has a plate. Your chip has one too, and it is tiny: `N52833 QIAAA0 2024AL`. If you cannot read it with your eyes, do not worry. The maker's data sheet explains the code, and the Discovery book passes it on:

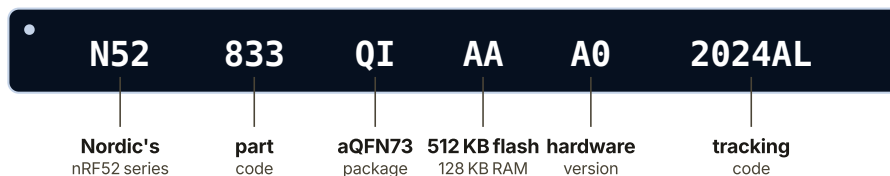


Figure 1.3 The part number is a code. Each piece answers one question.

PIECE	WHAT IT TELLS YOU
N52	Made by Nordic Semiconductor, in their nRF52 series
833	The part code, so this is the nRF52833
QI	The package: a small square with 73 tiny pins (called aQFN73)

PIECE	WHAT IT TELLS YOU
AA	The variant: 512 KB of flash and 128 KB of RAM
A0	The hardware version and configuration
2024AL	A tracking code, which may differ on your chip

Flash and RAM

The chip keeps two kinds of memory, and the difference matters. **Flash** is the workshop manual on the shelf. It keeps your program when the power is off. **RAM** is the whiteboard on the wall. It is quick to write on, and it is wiped when the power goes.

Flash: the workshop manual

512 KB. Keeps your program when the power is off.

512 KB

RAM: the whiteboard

128 KB. Quick to write on. Wiped when the power goes.

128 KB

a quarter of the length of the manual

Figure 1.4 The chip has far more manual than whiteboard.

So your program lives in flash, and the numbers it works on while it runs live in RAM. When you press reset, the whiteboard is wiped and the chip starts reading the manual again from the first page.

Who designed the engine?

The chip says Nordic, but the core inside it that does the calculating is called a **Cortex-M4**, and that name belongs to a company called Arm. Arm does not make chips. It designs processor cores and licenses the designs to makers like Nordic, who build them into their own chips. That is why the same Rust tools work on chips from many different makers.

KEY IDEA

The micro:bit is one small chip, the engine, with helpers around it. Your program lives in the chip's flash memory, and its numbers live in RAM.

SAY IT BACK

- The nRF52833 is the engine and the nRF52820 is the helper that talks to the laptop.
- Flash keeps the program, RAM holds working numbers and is wiped when power goes.

CHECK YOURSELF

1. Which black square on the back is the engine, and which is the helper?
2. What is the difference between flash and RAM, in the picture of the manual and the whiteboard?
3. Who designs the Cortex-M4 core, and who builds the chip?

Your workbench

YOU ALREADY KNOW

You have installed the tools and run a first program from a project folder.

YOU WILL BE ABLE TO

Set up one project folder for the whole book and run any program in it with one command.

PICTURE THIS

A good workshop has one bench with every tool within reach. You do not walk to another building to fetch a spanner. This book gives you one bench: one project folder with every library the micro:bit needs already listed, and one command that builds a program and puts it into the chip.

The folder

Unzip `microbit-properly` somewhere simple, like `D:\Projects\`, and open the folder in VS Code. Inside you will see:

FILE OR FOLDER	WHAT IT IS
<code>Cargo.toml</code>	The shopping list. Which libraries this project uses.
<code>.cargo\config.toml</code>	The fitter's instructions. Which chip to build for, and which tool to use to put the program in it.
<code>examples\</code>	Every program in this book. One file for each.
<code>src\main.rs</code>	A tiny empty program, so the folder is a complete project.

The shopping list

Cargo.toml (the libraries)

```
[dependencies]
cortex-m = { version = "0.7", features = ["critical-section-single-core"] }
cortex-m-rt = "0.7"
embedded-hal = "1.0"
microbit-v2 = "0.16"
panic-halt = "1.0"
panic-rtt-target = "0.2"
rtt-target = "0.6"
lsm303agr = "1.1"
```

Each line is a library. `microbit-v2` is the dashboard for this one board. `embedded-hal` is the standard shape for pedals and levers. `cortex-m-rt` starts the program properly when the chip wakes. `panic-halt` says what to do if the program breaks: stop. `rtt-target` prints messages to your laptop. `lsm303agr` talks to the tilt sensor.

The fitter's instructions

.cargo/config.toml

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

[target.thumbv7em-none-eabihf]
runner = "probe-rs run --chip nRF52833_xxAA"
rustflags = [
    "-C", "linker=rust-ld",
]
```

The first two lines say the program must be built for the engine's kind of processor, not for your laptop. The `runner` line is the fitter: after building, run `probe-rs` to put the file in the chip. This is exactly the setup you already used.

Running a program

Open the terminal inside the project folder and type one command, with the program's name at the end:

```
terminal
```

```
cargo run --example all-on
```

The very first run downloads the libraries and builds them, and it takes about a minute. Later runs reuse that work and take a couple of seconds. The program then runs on the board. To stop watching, press **Ctrl+C** in the terminal. The program keeps running on the chip.

If a build fails, read the first `error` line and the file name beside it. Copy that block to me. A compiler message is a fault code, and the first line is the one that matters.

Everything in one place

From here on, every chapter names a program and a command. The command is always the same shape, `cargo run --example` and then the name. Learn that shape once.

KEY IDEA

One folder, one shopping list, one fitter. `cargo run --example NAME` builds the program and puts it into the chip.

FROM MEMORY

Without looking: write the command that runs a program called `blink`. Then say what the `runner` line is for.

SAY IT BACK

- Cargo.toml lists the libraries and config.toml says how to build and fit.
- `cargo run --example NAME` builds a program and puts it into the chip.

CHECK YOURSELF

1. What does `Cargo.toml` hold, and what does `.cargo\config.toml` hold?
2. Why does the first run take a minute and later runs take seconds?
3. When you press Ctrl+C, does the program stop on the board?

The wall of windows

03

YOU ALREADY KNOW

You can run any program in the folder with one command.

YOU WILL BE ABLE TO

Draw a picture on the 25 LEDs, and draw any digit from memory.

PICTURE THIS

Imagine a wall of 25 windows, five rows of five, with a lamp behind each. A sign painter is told to write the number 7 on the wall. He does not paint anything. He decides which lamps to switch on, and the rest stay dark.

A picture on the micro:bit is exactly that: a list of which lamps are on.

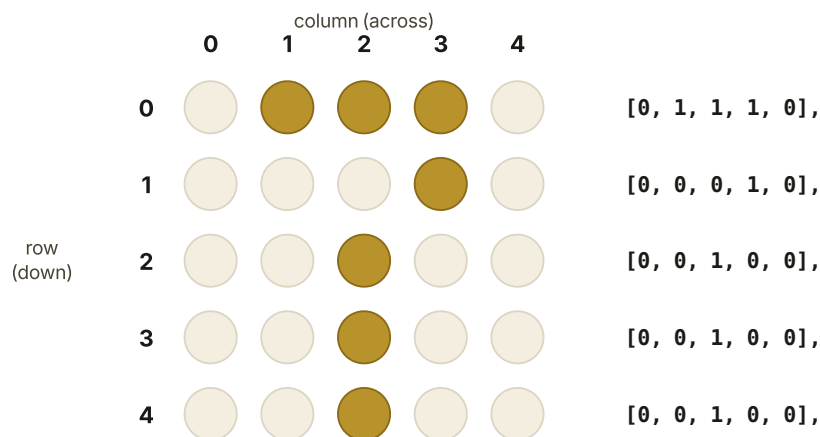


Figure 3.1 The number 7 as a list of rows. A 1 is a lamp on, a 0 is a lamp off. Row 0 is the top, column 0 is the left.

First, light every lamp

Run the first program and watch the grid:

```
terminal
```

```
cargo run --example all-on
```

Program adapted from the Rust Embedded Discovery book examples.

examples/all-on.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_halt as _;

#[entry]
fn main() -> ! {
    let board = Board::take().unwrap();
    let mut timer = Timer::new(board.TIMER0);
    let mut display = Display::new(board.display_pins);

    // A picture is 5 rows of 5 windows: 1 = lamp on, 0 = lamp off.
    let every_lamp = [
        [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 the picture for one second...
        display.show(&mut timer, every_lamp, 1000);
        // ...then switch every lamp off and wait one second.
        display.clear();
        timer.delay_ms(1000_u32);
    }
}
```

WHAT YOU SHOULD SEE

All 25 lamps light for one second, go dark for one second, and repeat.

Reading the program, in order

- `Board::take()` asks for the board: every pin, every part. You can take it only once, so two parts of a program cannot fight over the same pin.

- `Timer::new(board.TIMER0)` gets a stopwatch. The display library needs one to know how long a picture should stay up.
- `Display::new(board.display_pins)` hands the 25 lamps to the display library. It switches rows on and off so fast that your eye sees the whole picture.
- `every_lamp` is the picture: five rows of five numbers.
- `display.show(&mut timer, every_lamp, 1000)` shows the picture for 1000 milliseconds, which is one second.
- `display.clear()` switches every lamp off.

Now write with it

The next program is an alphabet: ten pictures, one for each digit. It shows them one after another, 0 to 9.

```
terminal
```

```
cargo run --example digits
```

Program written for this book.

```

#![no_main]
#![no_std]

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

// The sign painter's alphabet: ten pictures, one for each digit.
// Each picture is 5 rows of 5 windows. 1 = light on, 0 = light off.
// Row 0 is the top row. Column 0 is the left column.
// Our digits are 3 windows wide and sit in the middle, columns 1 to 3.
const DIGITS: [[u8; 5]; 5]; 10] = [
    // 0
    [
        [0, 1, 1, 1, 0],
        [0, 1, 0, 1, 0],
        [0, 1, 0, 1, 0],
        [0, 1, 0, 1, 0],
        [0, 1, 1, 1, 0],
    ],
    // 1
    [
        [0, 0, 1, 0, 0],
        [0, 1, 1, 0, 0],
        [0, 0, 1, 0, 0],
        [0, 0, 1, 0, 0],
        [0, 1, 1, 1, 0],
    ],
    // 2
    [
        [0, 1, 1, 1, 0],
        [0, 0, 0, 1, 0],
        [0, 1, 1, 1, 0],
        [0, 1, 0, 0, 0],
        [0, 1, 1, 1, 0],
    ],
    // 3
    [
        [0, 1, 1, 1, 0],
        [0, 0, 0, 1, 0],
        [0, 1, 1, 1, 0],
        [0, 0, 0, 1, 0],
        [0, 1, 1, 1, 0],
    ],
    // 4

```

```

[
    [0, 1, 0, 1, 0],
    [0, 1, 0, 1, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 0, 0, 1, 0],
],
// 5
[
    [0, 1, 1, 1, 0],
    [0, 1, 0, 0, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 1, 1, 0],
],
// 6
[
    [0, 1, 1, 1, 0],
    [0, 1, 0, 0, 0],
    [0, 1, 1, 1, 0],
    [0, 1, 0, 1, 0],
    [0, 1, 1, 1, 0],
],
// 7
[
    [0, 1, 1, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 1, 0, 0],
],
// 8
[
    [0, 1, 1, 1, 0],
    [0, 1, 0, 1, 0],
    [0, 1, 1, 1, 0],
    [0, 1, 0, 1, 0],
    [0, 1, 1, 1, 0],
],
// 9
[
    [0, 1, 1, 1, 0],
    [0, 1, 0, 1, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 1, 1, 0],
],

```

```

];

#[entry]
fn main() -> ! {
    let board = Board::take().unwrap();
    let mut timer = Timer::new(board.TIMER0);
    let mut display = Display::new(board.display_pins);

    loop {
        // Show each picture in turn, 800 milliseconds each, then start again.
        for picture in DIGITS {
            display.show(&mut timer, picture, 800);
        }
    }
}

```

WHAT YOU SHOULD SEE

The digits count 0, 1, 2 up to 9 and start again, each one for under a second. Each digit is three windows wide and sits in the middle three columns.

The alphabet is a table of ten pictures, `DIGITS`. Each entry is a picture exactly like the 7 above. To show digit 4, you hand the library the fifth entry, because counting starts at zero. The loop `for picture in DIGITS` walks down the table, one picture at a time.

That is the heart of every display you will ever build: a table of pictures, and a rule for picking which one to show.

KEY IDEA

A picture is five rows of five numbers. Writing on the board means choosing which picture to show.

FROM MEMORY

1. Close the file. On paper, draw the picture for the digit 2 as five rows of five numbers.
2. Change the 7 in `digits.rs` so its top bar is two lamps long instead of three. Run it.
3. Then write a new program that shows only your initial, drawn by you.

SAY IT BACK

- A picture is five rows of five numbers: 1 is on, 0 is off.
- `display.show` takes the stopwatch, a picture, and how long to show it.

CHECK YOURSELF

1. In `[0, 1, 1, 1, 0]` , which lamps are on?
2. Why can you only call `Board::take()` once?
3. What does `display.show` need from you? Name its three inputs.

The paper strip

YOU ALREADY KNOW

You can draw a picture on the 25 lamps.

YOU WILL BE ABLE TO

Scroll a number across the grid, and say why two digits do not fit at once.

PICTURE THIS

A shop has a narrow window and a long paper banner on a roller behind it. Only a short piece of the banner can be seen at one time. To read the whole sign, the shopkeeper winds the roller a little at a time and the words slide past.

The micro:bit's grid is that window. It is five columns wide. A two-digit number is wider than that, so we put the number on a long strip and slide the window along it.

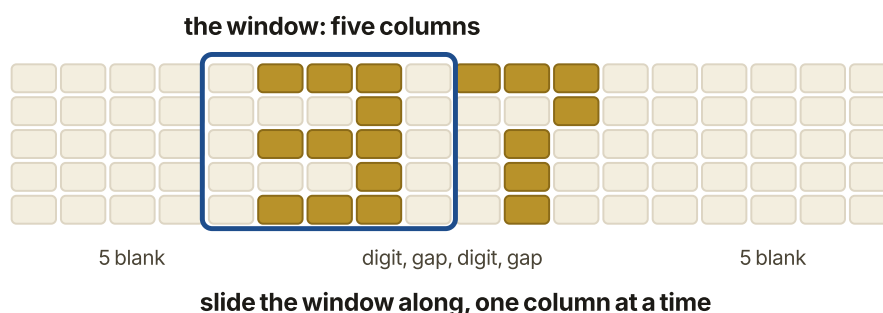


Figure 4.1 A long strip with five blank columns, the digits, and five more blank columns. The window shows five columns at a time.

A smaller alphabet

In the last chapter each digit was five rows of five numbers. That is a lot of typing, so the next program shortens each row to three bits. A row such as `0b101` means lamp on, off, on. The whole digit is five of those. The program works out the picture from it.

```
terminal
```

```
cargo run --example counter
```

Program written for this book.

```

#![no_main]
#![no_std]

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

// Digits 0 to 9, three windows wide. Each row is 3 bits: 1 = lamp on, left bit
// first.
const FONT: [[u8; 5]; 10] = [
    [0b111, 0b101, 0b101, 0b101, 0b111], // 0
    [0b010, 0b110, 0b010, 0b010, 0b111], // 1
    [0b111, 0b001, 0b111, 0b100, 0b111], // 2
    [0b111, 0b001, 0b111, 0b001, 0b111], // 3
    [0b101, 0b101, 0b111, 0b001, 0b001], // 4
    [0b111, 0b100, 0b111, 0b001, 0b111], // 5
    [0b111, 0b100, 0b111, 0b101, 0b111], // 6
    [0b111, 0b001, 0b010, 0b010, 0b010], // 7
    [0b111, 0b101, 0b111, 0b101, 0b111], // 8
    [0b111, 0b101, 0b111, 0b001, 0b111], // 9
];

// Picture a long strip of paper behind a window five columns wide.
// On the strip: 5 blank columns, then two digits (3 columns + 1 gap each), then 5
// blank.
// Sliding the window along the strip makes the number scroll past.
const STRIP_LEN: usize = 5 + 8 + 5;

/// What is on the strip at column `v`, row `row`? 1 = lamp on.
fn strip_pixel(v: usize, row: usize, digits: [usize; 2]) -> u8 {
    if v < 5 {
        return 0;
    }
    let w = v - 5;
    let which = w / 4; // 0 = tens digit, 1 = units digit, 2 = past the end
    let within = w % 4; // 0, 1, 2 = the digit's columns, 3 = the gap
    if which >= 2 || within == 3 {
        return 0;
    }
    (FONT[digits[which]][row] >> (2 - within)) & 1
}

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

```

```

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

loop {
    for number in 0..100usize {
        let digits = [number / 10, number % 10];
        // Slide the window along the strip, one column at a time.
        for offset in 0..(STRIP_LEN - 5) {
            let mut picture = [[0u8; 5]; 5];
            for row in 0..5 {
                for col in 0..5 {
                    picture[row][col] = strip_pixel(offset + col, row, digits);
                }
            }
            display.show(&mut timer, picture, 120);
        }
    }
}

```

WHAT YOU SHOULD SEE

The numbers 00 up to 99 scroll across the grid from right to left, one after another, and then it starts again.

Reading the program

- `FONT` is the alphabet: ten digits, each five rows of three bits.
- `strip_pixel(v, row, digits)` answers one question: on the long strip, in column `v`, is the lamp at this row on? The first five columns are blank. Then each digit takes three columns plus one blank gap. Anything past the end is blank.
- The inner loop builds a picture of five by five by asking that question 25 times, with the window starting at column `offset`.
- The loop over `offset` slides the window along by one column each time, and `display.show(..., 120)` holds each frame for 120 milliseconds.

Slide the window by a different amount and you get a different speed. Try a shorter frame time and the number rushes past.

KEY IDEA

To show something wider than the grid, put it on a long strip and slide a five-column window along it.

FROM MEMORY

1. Make the numbers scroll twice as fast.
2. Change the program so it counts only from 0 to 9, and stops on each digit for one second without scrolling.
3. Rebuild `strip_pixel` from memory on paper: what are the three cases that return 0?

SAY IT BACK

- The grid is five columns wide, so a number scrolls by sliding a window along a strip.
- The frame time in `display.show` sets the scroll speed.

CHECK YOURSELF

1. How many columns wide is the grid, and how wide is one digit in this alphabet?
2. What is the gap column for?
3. Which number controls how fast the scroll goes?

The doorbell

YOU ALREADY KNOW

You can put any picture on the grid.

YOU WILL BE ABLE TO

Read button A and button B, and make the display answer them.

PICTURE THIS

A doorbell has two wires. While nobody presses it, the wires are apart and nothing happens. When a finger presses it, the wires touch and the circuit completes. The bell does not know who pressed it. It only knows the wires touch.

A button on a chip works the same way, with one twist. The chip pin is gently held up at high voltage. Pressing the button joins the pin to ground, which pulls it low. So on this board, **pressed means low**.

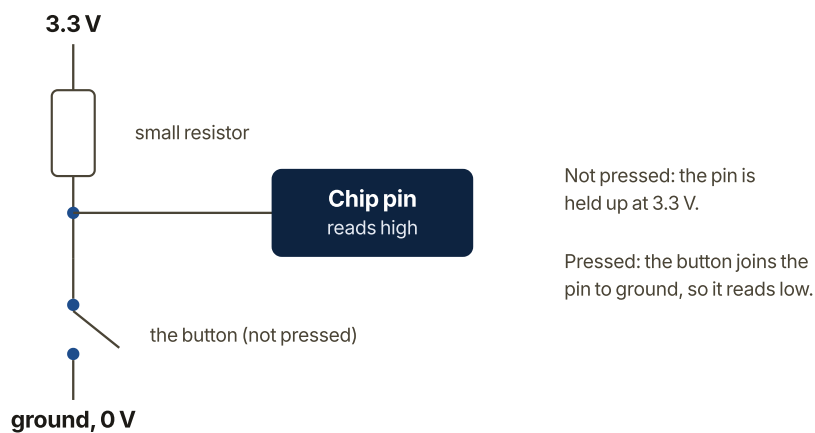


Figure 5.1 Not pressed, the pin sits high. Pressed, the button joins it to ground and it reads low.

The program cannot feel a finger. All it can do is ask the pin, "are you high or low?" Asking over and over, many times a second, is called **polling**. It is like a guard walking back and forth past a door to check whether it is open.

Turn signals on the grid

The next program does a small, useful job. Press A and a left arrow appears. Press B and a right arrow appears. Press neither and a single dot sits in the middle.

```
terminal
```

```
cargo run --example arrows
```

Program adapted from the Rust Embedded Discovery book examples.

```

#![no_main]
#![no_std]

use cortex_m_rt::entry;
use embedded_hal::digital::InputPin;
use microbit::{board::Board, display::blocking::Display, hal::Timer};
use panic_halt as _;

const LEFT_ARROW: [[u8; 5]; 5] = [
    [0, 0, 1, 0, 0],
    [0, 1, 0, 0, 0],
    [1, 1, 1, 1, 1],
    [0, 1, 0, 0, 0],
    [0, 0, 1, 0, 0],
];

const RIGHT_ARROW: [[u8; 5]; 5] = [
    [0, 0, 1, 0, 0],
    [0, 0, 0, 1, 0],
    [1, 1, 1, 1, 1],
    [0, 0, 0, 1, 0],
    [0, 0, 1, 0, 0],
];

const CENTRE_DOT: [[u8; 5]; 5] = [
    [0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0],
];

#[entry]
fn main() -> ! {
    let board = Board::take().unwrap();
    let mut timer = Timer::new(board.TIMER0);
    let mut display = Display::new(board.display_pins);
    let mut button_a = board.buttons.button_a;
    let mut button_b = board.buttons.button_b;

    loop {
        // A pressed button reads "low". Ask both buttons, over and over.
        if button_a.is_low().unwrap() {
            display.show(&mut timer, LEFT_ARROW, 10);
        } else if button_b.is_low().unwrap() {

```

```

        display.show(&mut timer, RIGHT_ARROW, 10);
    } else {
        display.show(&mut timer, CENTRE_DOT, 10);
    }
}
}

```

WHAT YOU SHOULD SEE

A dot in the middle. Hold A and it becomes a left arrow. Hold B and it becomes a right arrow.

Reading the program

- `board.buttons.button_a` and `button_b` are the two buttons. You take them from the board the same way as the display.
- `button_a.is_low().unwrap()` asks "is button A pressed?" and answers true or false. The `unwrap()` handles a rare error by stopping, which is fine for a small program.
- The loop checks A first, then B. If neither is pressed, it shows the dot.
- Each pass shows a picture for 10 milliseconds, so the loop goes round about a hundred times a second. That is quick enough that the board feels instant.

KEY IDEA

A pressed button reads low. Polling means asking the pin again and again.

FROM MEMORY

1. Swap the arrows, so A shows right and B shows left. Run it.
2. Add a third picture, a ring of lamps round the edge, that shows when you hold both buttons together.
3. Close the file and write the loop on paper: which question is asked first, and what happens last?

SAY IT BACK

- On this board a pressed button reads low.
- Polling means asking a pin its state over and over.

CHECK YOURSELF

1. When button A is pressed, does its pin read high or low?
2. What is polling, in the picture of the guard and the door?
3. What happens in `arrows.rs` when both buttons are pressed together?

The indicator stalk

YOU ALREADY KNOW

You can read the two buttons and show a picture.

YOU WILL BE ABLE TO

Make arrows blink like a car's indicators, and say why counting passes of the loop works as a clock.

PICTURE THIS

On a car, the lever beside the steering wheel flicks up for the right indicator and down for the left. The lamp does not stay on. It blinks: on for about half a second, off for about half a second. Press the hazard button and both sides blink together.

We have two buttons and a grid. That is enough to make the same thing.

The problem: blinking while watching

To blink a lamp, you switch it on, wait, switch it off, wait. But if the program is busy waiting, it is not looking at the buttons. If you let go of the button in the middle of a wait, the program will not notice until the wait is over.

There is a neat trick for small programs. Do not wait for a long time. Take one short step, about ten milliseconds, and each time round the loop, count one tick. The count is your clock. When the count says it is time to switch, switch. Because each step is short, the buttons are checked a hundred times a second.

```
terminal
```

```
cargo run --example indicator
```

Program written for this book.

```

#![no_main]
#![no_std]

use cortex_m_rt::entry;
use embedded_hal::digital::InputPin;
use microbit::{board::Board, display::blocking::Display, hal::Timer};
use panic_halt as _;

const BLANK: [[u8; 5]; 5] = [[0; 5]; 5];

const LEFT: [[u8; 5]; 5] = [
    [0, 0, 1, 0, 0],
    [0, 1, 0, 0, 0],
    [1, 1, 1, 1, 1],
    [0, 1, 0, 0, 0],
    [0, 0, 1, 0, 0],
];

const RIGHT: [[u8; 5]; 5] = [
    [0, 0, 1, 0, 0],
    [0, 0, 0, 1, 0],
    [1, 1, 1, 1, 1],
    [0, 0, 0, 1, 0],
    [0, 0, 1, 0, 0],
];

// Both arrows drawn on top of each other: the hazard lights.
const HAZARD: [[u8; 5]; 5] = [
    [0, 0, 1, 0, 0],
    [0, 1, 0, 1, 0],
    [1, 1, 1, 1, 1],
    [0, 1, 0, 1, 0],
    [0, 0, 1, 0, 0],
];

#[entry]
fn main() -> ! {
    let board = Board::take().unwrap();
    let mut timer = Timer::new(board.TIMER0);
    let mut display = Display::new(board.display_pins);
    let mut button_a = board.buttons.button_a;
    let mut button_b = board.buttons.button_b;

    // Each pass round the loop takes about 10 milliseconds.
    // We count the passes. That count is our clock.

```

```

let mut ticks: u32 = 0;

loop {
    let left = button_a.is_low().unwrap();
    let right = button_b.is_low().unwrap();

    // 40 passes of 10 ms = 400 ms on, then 400 ms off.
    let lit = (ticks / 40) % 2 == 0;

    let picture = match (left, right) {
        (true, false) if lit => LEFT,
        (false, true) if lit => RIGHT,
        (true, true) if lit => HAZARD,
        _ => BLANK,
    };

    display.show(&mut timer, picture, 10);
    ticks = ticks.wrapping_add(1);
}
}

```

WHAT YOU SHOULD SEE

Hold A and a left arrow blinks, on for about 0.4 of a second and off for the same. Hold B and a right arrow blinks. Hold both and the two arrows blink together, like hazard lights.

Reading the program

- `ticks` counts passes round the loop. Each pass takes about 10 milliseconds.
- `(ticks / 40) % 2 == 0` is true for 40 passes, then false for 40 passes, then true again. That is 400 milliseconds on and 400 off.
- The `match` looks at both buttons at once. Left alone and lit gives the left arrow. Right alone and lit gives the right arrow. Both and lit gives the hazard picture. Anything else gives a blank grid.
- `wrapping_add(1)` adds one, and if the count ever reaches its maximum, it quietly starts again at zero instead of crashing.

Where this method runs out

This works for small jobs. As soon as the program has many things to do, each with its own timing, counting passes by hand becomes tangled. Later you will meet a better way, where the chip itself taps the program on the shoulder when something happens. That is called an interrupt. For now, notice what the trick teaches: a program has no clock of its own, so it makes one by counting.

KEY IDEA

Take short steps and count them. The count is your clock, and the buttons stay watched the whole time.

FROM MEMORY

1. Make the indicator blink faster, about three times a second.
2. Make the indicator latch: press A once and it keeps blinking, press A again and it stops.
3. Explain on paper why the program uses 10 ms steps and not one long wait.

SAY IT BACK

- Counting short steps gives the program a clock while it keeps watching the buttons.

CHECK YOURSELF

1. Why is the program not blinking by waiting for half a second?
2. What does `(ticks / 40) % 2 == 0` do over time?
3. Which picture shows when both buttons are held?

The stopwatch

YOU ALREADY KNOW

You have made a program count steps to keep time.

YOU WILL BE ABLE TO

Use the timer to wait exactly, and blink a single lamp at a steady rate.

PICTURE THIS

A mechanic who checks a tightening sequence uses a stopwatch, not a guess. "Hold for ten seconds." He does not count in his head. He starts the watch, glances at it and stops when it says so.

The chip has stopwatches built into it, called **timers**. They count steadily whatever the program is doing. A wait that uses a timer is exact. A wait that uses a loop which counts is only as exact as how fast the loop happens to run.

The lamp at the crossing

The simplest light on the grid is the top-left one. It sits where row 1 crosses column 1, like a lamp at a crossing of two streets. A lamp lights when its row is driven high and its column is driven low. This program drives the column low and then switches the row on and off every half second.

```
terminal
```

```
cargo run --example blink
```

Program adapted from the Rust Embedded Discovery book examples.

```

#![no_main]
#![no_std]

use embedded_hal::delay::DelayNs;
use embedded_hal::digital::OutputPin;

use cortex_m_rt::entry;
use microbit::{hal::gpio, hal::Timer, Board};
use panic_halt as _;

#[entry]
fn main() -> ! {
    let board = Board::take().unwrap();
    let mut timer = Timer::new(board.TIMER0);

    // The top-left LED sits where row 1 meets column 1.
    // A lamp lights when its row is high and its column is low.
    let mut row1 = board
        .display_pins
        .row1
        .into_push_pull_output(gpio::Level::Low);
    let _col1 = board
        .display_pins
        .col1
        .into_push_pull_output(gpio::Level::Low);

    loop {
        row1.set_high().unwrap();
        timer.delay_ms(500_u32);
        row1.set_low().unwrap();
        timer.delay_ms(500_u32);
    }
}

```

WHAT YOU SHOULD SEE

The top-left lamp is on for half a second, off for half a second, and repeats.

Reading the program

- `board.display_pins.row1` and `col1` are the two wires that meet at the top-left lamp.
- `into_push_pull_output(gpio::Level::Low)` turns a pin into an output, a switch your program controls, and sets its starting level.

- `row1.set_high()` puts the row at full voltage and the lamp lights. `set_low()` puts it at zero and the lamp goes out.
- `timer.delay_ms(500_u32)` waits exactly 500 milliseconds on the timer.

Notice that this program does **nothing else** while it waits. It cannot look at the buttons during a wait. That is why the last chapter used the counting trick. Which method to use depends on the job: a pure blink can just wait, and a program that must stay watchful cannot.

KEY IDEA

A timer gives exact waits. A pin set as an output is a switch your program controls.

FROM MEMORY

1. Change the blink to two quick flashes and a long gap, like a signal.
2. Make the lamp blink exactly once a second, on for 100 ms and off for 900 ms.
3. Write the four steps of the loop on paper without looking.

SAY IT BACK

- A timer gives exact waits.
- An output pin is a switch your program controls, high or low.

CHECK YOURSELF

1. What two conditions make a lamp on the grid light?
2. What does `into_push_pull_output` do to a pin?
3. Why can this blink not check the buttons during a wait?

The speaker cone

YOU ALREADY KNOW

You can switch a pin high and low on a timer.

YOU WILL BE ABLE TO

Play a note of any pitch, and play a scale.

PICTURE THIS

Hold a tin lid against your chest and tap it quickly with a finger. The lid moves in and out. The air moves with it. If you tap fast enough, your ear hears a note.

The micro:bit's speaker is a small cone that moves in and out. When the pin is high, the cone is pushed out. When the pin is low, the cone is pulled in. Do that a few hundred times a second and you get a note.

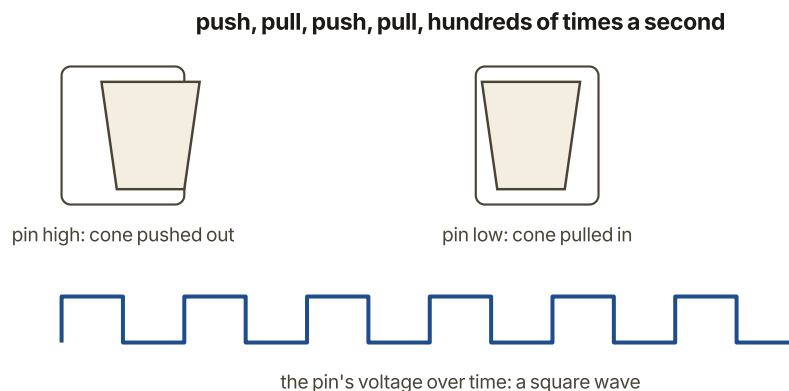


Figure 8.1 The pin goes high and low. The cone follows it, and the air moves.

The number of pushes and pulls per second is the **pitch**, measured in hertz. The ordinary A above middle C is 440 hertz: 440 pushes and 440 pulls every second.

One note

terminal

```
cargo run --example tone
```

Program adapted from the Rust Embedded Discovery book examples.

examples/tone.rs

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

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

// A 440 Hz note: 440 pushes out and 440 pulls in every second.
// One whole cycle takes 1,000,000 / 440 = about 2273 microseconds.
// Half of that, pushed out, and half pulled in.
const HALF_CYCLE_US: u32 = 1136;
const CYCLES: u32 = 440 * 2; // two seconds

#[entry]
fn main() -> ! {
    let board = Board::take().unwrap();
    let mut speaker = board.speaker_pin.into_push_pull_output(gpio::Level::Low);
    let mut timer = Timer::new(board.TIMER0);

    for _ in 0..CYCLES {
        speaker.set_high().unwrap();
        timer.delay_us(HALF_CYCLE_US);
        speaker.set_low().unwrap();
        timer.delay_us(HALF_CYCLE_US);
    }

    // The program lives in flash. Without this stop it would play again
    // every time you power the board. Sleep until something wakes the chip.
    loop {
        asm::wfi();
    }
}
```

WHAT YOU SHOULD SEE

A steady note, the A above middle C, for two seconds, and then silence.

Reading the program

- One cycle of 440 hertz lasts about 2273 microseconds. A microsecond is a millionth of a second. Half of that is 1136 microseconds, which is the time for each push and each pull.
- The loop pushes the cone out, waits half a cycle, pulls it in, waits half a cycle, 880 times. That is two seconds of note.
- The last loop with `asm:wfi()` parks the chip asleep once the note is done. A program stored in flash starts again from the top every time the board is powered or reset, so this keeps it from running off the end of `main`.

A whole scale

A function lets you write the "play one note" job once and use it for every note. The scale program plays do, re, mi, fa, sol, la, ti, do.

```
terminal
```

```
cargo run --example scale
```

Program written for this book.

```

#![no_main]
#![no_std]

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

// Notes, in cycles per second (Hertz): do re mi fa sol la ti do
const SCALE: [u32; 8] = [262, 294, 330, 349, 392, 440, 494, 523];

/// Play one note: push and pull the speaker `hz` times a second, for `ms`
/// milliseconds.
fn play<P: OutputPin, T: DelayNs>(speaker: &mut P, timer: &mut T, hz: u32, ms: u32)
{
    let half_us = 500_000 / hz;
    let cycles = hz * ms / 1000;
    for _ in 0..cycles {
        speaker.set_high().unwrap();
        timer.delay_us(half_us);
        speaker.set_low().unwrap();
        timer.delay_us(half_us);
    }
}

#[entry]
fn main() -> ! {
    let board = Board::take().unwrap();
    let mut speaker = board.speaker_pin.into_push_pull_output(gpio::Level::Low);
    let mut timer = Timer::new(board.TIMER0);

    for hz in SCALE {
        play(&mut speaker, &mut timer, hz, 400);
        timer.delay_ms(50_u32); // a short gap so the notes do not run together
    }

    loop {
        asm::wfi();
    }
}

```

WHAT YOU SHOULD SEE

Eight rising notes, each for 400 milliseconds with a short gap, and then silence.

The function `play` takes the speaker, the timer, the pitch in hertz, and how long to play for. It works out the half cycle for itself, `500_000 / hz` microseconds. Change the numbers in `SCALE` and you change the tune.

The speaker on this board is a simple on-off kind: the cone is either fully out or fully in. That is why the sound is bright and buzzy and cannot reproduce speech or music with soft tones. That is the speaker, not your program.

KEY IDEA

Pitch is how many pushes and pulls per second. Half a cycle is the wait between a push and a pull.

FROM MEMORY

1. Play your name's first letter as a tune of four notes you choose.
2. Make the scale go down as well as up.
3. Write on paper how to work out the half cycle for 330 hertz.

SAY IT BACK

- Pitch is the number of pushes and pulls per second, in hertz.
- The half cycle is 500,000 divided by the pitch, in microseconds.

CHECK YOURSELF

1. What does the cone do when the speaker pin is high?
2. What is the half cycle for a note of 500 hertz, in microseconds?
3. Why does the program end in an endless `wfi` loop?

Talking to the laptop

09

YOU ALREADY KNOW

You can make the board act, but you cannot yet hear it say anything.

YOU WILL BE ABLE TO

Print messages from the board to your laptop in two ways.

PICTURE THIS

A mechanic listens to an engine, but a good mechanic also has a scan tool that tells him what the engine thinks. The engine cannot talk, so the scan tool reads its numbers and writes them on a screen.

Your board can write numbers and words to your laptop in the same way. There are two routes, and each is useful for a different reason.

Route one: RTT, the quick line

RTT is a way for the program to drop short messages into a small memory buffer, and for the programmer tool to read them out through the same two wires used to load the program. There is no extra cable and no extra setup. When you start the program with `cargo run`, the messages show up in your terminal.

```
terminal
```

```
cargo run --example hello-rtt
```

Program written for this book.

```

#![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};

#[entry]
fn main() -> ! {
    rtt_init_print!();
    rprintln!("Hello from the micro:bit");

    let mut count: u32 = 0;
    while count < 5 {
        count += 1;
        rprintln!("count is {}", count);
    }

    loop {
        wfi();
    }
}

```

WHAT YOU SHOULD SEE

In the terminal: the line "Hello from the micro:bit", then "count is 1" up to "count is 5".

`rtt_init_print!()` sets the line up. `rprintln!` works like the `println!` you know from ordinary Rust, with curly brackets for the values. The messages are for you while you work. Nobody sees them once the board runs alone.

Route two: the serial port

The serial port is a true line from the engine to your laptop, through the second chip and the USB cable. A program on your laptop, such as PuTTY, shows what arrives. Unlike RTT, serial is the same line a finished product might use to talk to a computer.

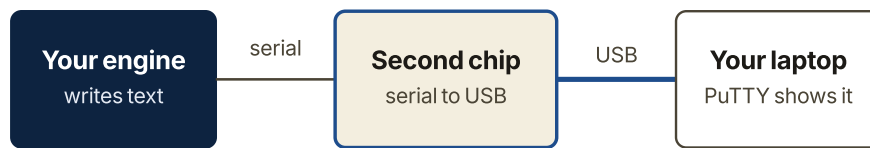


Figure 9.1 Text goes from the engine, through the second chip, over USB, and into a terminal program.

On Windows, open a terminal and type `mode` before you plug the board in, and again after. The new COM port that appears, such as COM3, is the board. Open PuTTY, choose **Serial**, type that COM name, and set the speed to **115200**, with 8 data bits, 1 stop bit, no parity and no flow control. Click Open.

```
terminal
```

```
cargo run --example hello-serial
```

Program adapted from the Rust Embedded Discovery book examples.

```

#![no_main]
#![no_std]

use core::fmt::Write;

use cortex_m::asm::wfi;
use cortex_m_rt::entry;
use embedded_hal::delay::DelayNs;
use microbit::{
    hal::{
        uarte,
        uarte::{Baudrate, Parity},
        Timer,
    },
    Board,
};
use panic_halt as _;

#[entry]
fn main() -> ! {
    let board = Board::take().unwrap();
    let mut timer = Timer::new(board.TIMER0);

    // The UART is the serial port. The second chip carries it to the laptop over USB.
    let mut serial = uarte::Uarte::new(
        board.UARTE0,
        board.uart.into(),
        Parity::EXCLUDED,
        Baudrate::BAUD115200,
    );

    let mut n: u32 = 0;
    loop {
        writeln!(serial, "line {} from the micro:bit", n).unwrap();
        n += 1;
        timer.delay_ms(1000_u32);
        if n == 10 {
            break;
        }
    }

    loop {
        wfi();
    }
}

```

```
}  
}
```

WHAT YOU SHOULD SEE

In the PuTTY window, a new line every second: "line 0 from the micro:bit", "line 1 ...", up to line 9, then no more. Press the reset button on the back to run it again.

`Uarte::new` builds the serial port. The speed, 115200, is called the baud rate: the number of signal changes per second. Both ends must agree on it, or you get nonsense. `writeln!` writes a line of text to the port, with values in curly brackets like `println!`.

Start PuTTY **before** you flash the program. The serial line does not keep old messages. If PuTTY is not listening when the board speaks, the words are gone.

KEY IDEA

RTT is quick and shows up in your terminal while you work.
Serial is a real line to the laptop and needs a program such as PuTTY to read it.

FROM MEMORY

1. Make `hello-rtt` print the numbers 1 to 10 and their squares.
2. Make `hello-serial` send your name, and the count, once a second.
3. Explain on paper why the baud rate must match at both ends.

SAY IT BACK

- RTT prints to the terminal while you work. Serial goes through the second chip to PuTTY.
- Both ends must use the same baud rate.

CHECK YOURSELF

1. Which route needs no extra program on the laptop?
2. What speed do we set in PuTTY?
3. Why start PuTTY before flashing?

The spirit level

10

YOU ALREADY KNOW

You can print numbers to your laptop and show a picture on the grid.

YOU WILL BE ABLE TO

Read the tilt sensor and make a lamp slide like the bubble in a spirit level.

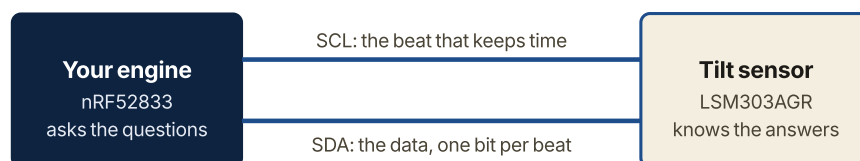
PICTURE THIS

A builder's spirit level has a small bubble in a tube of liquid. Tilt the level and the bubble drifts to the high end. When the surface is flat, the bubble sits in the middle.

The micro:bit has a sensor inside it that feels which way is down. We will use it to make one lamp on the grid behave like that bubble.

The sensor and the two wires

The sensor is a small chip called the LSM303AGR. It can feel tilt and shake, and it can feel the direction of north. It is a separate chip from the engine, so the engine has to ask it questions. They talk over a pair of wires called **I2C**. One wire, the clock, ticks like a metronome. The other wire carries the data, one bit on each tick.



Two wires carry the whole conversation

Figure 10.1 Two wires carry the whole conversation between the engine and the sensor.

You do not have to build the conversation by hand. A ready-made driver library, `Lsm303agr`, knows the sensor's language. Your program only says: start the sensor, then give me the acceleration.

```
terminal
```

```
cargo run --example tilt
```

Program adapted from the Rust Embedded Discovery book examples.

Adapted from the Discovery book's `show-accel` example, with a spirit-level display written for this book.

```

#![no_main]
#![no_std]

use cortex_m_rt::entry;
use microbit::{
    display::blocking::Display,
    hal::{twim, Timer},
    pac::twim0::frequency::FREQUENCY_A,
    Board,
};
use lsm303agr::{AccelMode, AccelOutputDataRate, Lsm303agr};
use panic_halt as _;

/// Turn a tilt reading (in thousandths of g) into a column or row, 0 to 4.
/// Flat means about 0 and lands in the middle, 2.
fn to_cell(mg: i32) -> usize {
    let step = mg / 200; // every 200 mg moves the dot by one window
    let cell = 2 + step;
    if cell < 0 {
        0
    } else if cell > 4 {
        4
    } else {
        cell as usize
    }
}

#[entry]
fn main() -> ! {
    let board = Board::take().unwrap();
    let mut timer = Timer::new(board.TIMER0);
    let mut display = Display::new(board.display_pins);

    // The tilt sensor talks on a two-wire road called I2C, inside the board.
    let i2c = twim::Twim::new(board.TWIM0, board.i2c_internal.into(),
FREQUENCY_A::K100);

    let mut sensor = Lsm303agr::new_with_i2c(i2c);
    sensor.init().unwrap();
    sensor
        .set_accel_mode_and_odr(
            &mut timer,
            AccelMode::HighResolution,
            AccelOutputDataRate::Hz50,
        )
}

```

```

        .unwrap();

    loop {
        if sensor.accel_status().unwrap().xyz_new_data() {
            let (x, y, _z) = sensor.acceleration().unwrap().xyz_mg();

            // One lit window, the bubble in the spirit level.
            let mut picture = [[0u8; 5]; 5];
            let col = to_cell(x);
            let row = to_cell(y);
            picture[row][col] = 1;
            display.show(&mut timer, picture, 20);
        }
    }
}

```

WHAT YOU SHOULD SEE

One lit lamp. Flat on the table, it sits in the middle. Tilt the board and the lamp slides toward the low side.

Reading the program

- `Twim::new(board.TWIM0, board.i2c_internal.into(), K100)` builds the two-wire link to the sensor, running at 100,000 bits per second.
- `Lsm303agr::new_with_i2c(i2c)` and `sensor.init()` start up the driver and the sensor. `set_accel_mode_and_odr` chooses how precise and how often to measure: here, high resolution, 50 times a second.
- `sensor.acceleration().unwrap().xyz_mg()` gives three numbers: how hard the sensor is being pulled in the x, y and z directions, in thousandths of the pull of gravity. On a flat table, x and y are near zero and z is near 1000, because gravity pulls straight down through the board.
- `to_cell` turns a tilt number into a grid position from 0 to 4. Every 200 units of tilt moves the lamp one window, and the result is held between 0 and 4.

I do not know from here which way each axis points on your board. If the lamp slides the wrong way when you tilt, put a minus sign in front of `x` or `y`. Finding and fixing that is a good first experiment.

KEY IDEA

The sensor is a separate chip. The engine asks it questions over two wires, and a driver library does the talking.

FROM MEMORY

1. Make the lamp move twice as sensitive to tilt.
2. Make the whole grid flash when the board is shaken hard.
3. Draw on paper the board lying flat, and say what x, y and z should read.

SAY IT BACK

- I2C uses a clock wire and a data wire.
- Acceleration comes in thousandths of gravity: flat means x and y near zero.

CHECK YOURSELF

1. What does I2C use two wires for?
2. What does the tilt sensor measure, and in what unit?
3. What would x and y read on a flat table, roughly?

The thermometer

YOU ALREADY KNOW

You can show digits on the grid and read a number from a part of the chip.

YOU WILL BE ABLE TO

Read the chip's temperature and show it on the grid.

PICTURE THIS

Many engines have a temperature sender screwed into the block. It does not measure the air in the workshop. It measures the engine itself. If the engine has been working hard, the reading is high.

The nRF52833 has a thermometer built into the silicon. It measures the chip's own temperature. On a board that has been sitting idle, that is close to the air around it. On a board that has been working hard, it is warmer.

```
terminal
```

```
cargo run --example temperature
```

Program written for this book.

```

#![no_main]
#![no_std]

use cortex_m_rt::entry;
use embedded_hal::delay::DelayNs;
use microbit::{
    display::blocking::Display,
    hal::{temp::Temp, Timer},
    Board,
};

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

// Digits 0 to 9, three windows wide. Each row is 3 bits: 1 = lamp on, left bit first.
const FONT: [[u8; 5]; 10] = [
    [0b111, 0b101, 0b101, 0b101, 0b111], // 0
    [0b010, 0b110, 0b010, 0b010, 0b111], // 1
    [0b111, 0b001, 0b111, 0b100, 0b111], // 2
    [0b111, 0b001, 0b111, 0b001, 0b111], // 3
    [0b101, 0b101, 0b111, 0b001, 0b001], // 4
    [0b111, 0b100, 0b111, 0b001, 0b111], // 5
    [0b111, 0b100, 0b111, 0b101, 0b111], // 6
    [0b111, 0b001, 0b010, 0b010, 0b010], // 7
    [0b111, 0b101, 0b111, 0b101, 0b111], // 8
    [0b111, 0b101, 0b111, 0b001, 0b111], // 9
];

/// Draw one digit in the middle three columns of the grid.
fn picture_of(digit: usize) -> [[u8; 5]; 5] {
    let mut picture = [[0u8; 5]; 5];
    for row in 0..5 {
        for col in 0..3 {
            picture[row][col + 1] = (FONT[digit][row] >> (2 - col)) & 1;
        }
    }
    picture
}

#[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);

```

```
// The chip has a thermometer inside it. It reads the chip's own temperature,  
// which is close to the air around it when the board has not been working hard.  
let mut thermometer = Temp::new(board.TEMP);
```

```
loop {  
    // The reading comes in steps of a quarter of a degree.  
    let celsius: f32 = thermometer.measure().to_num();  
    rprintln!("{}", degrees C", celsius);  
  
    let whole = celsius as i32;  
    let whole = if whole < 0 { 0 } else if whole > 99 { 99 } else { whole };  
    let tens = (whole / 10) as usize;  
    let units = (whole % 10) as usize;  
  
    display.show(&mut timer, picture_of(tens), 800);  
    display.clear();  
    timer.delay_ms(200_u32);  
    display.show(&mut timer, picture_of(units), 800);  
    display.clear();  
    timer.delay_ms(800_u32);  
}  
}
```

WHAT YOU SHOULD SEE

In the terminal, a line such as "24.5 degrees C" every few seconds. On the grid, the tens digit, a pause, then the units digit, then a longer pause.

Reading the program

- `Temp::new(board.TEMP)` takes the thermometer. `thermometer.measure()` asks it for a reading.
- The reading comes in steps of a quarter of a degree, and `.to_num()` turns it into an ordinary number with a decimal point.
- `rprintln!` prints it for you to read in the terminal.
- The digits are split into tens and units, and each is drawn from the same alphabet idea as chapter 4: a table of pictures and a rule for picking one. `picture_of` builds a picture from three-bit rows.

This is the temperature of the chip and not a laboratory instrument. Do not use it to set a medical or safety limit. It is good for watching how a board warms and cools.

KEY IDEA

**The chip senses its own temperature, in quarter-degree steps.
The table-of-pictures idea shows any number on the grid.**

FROM MEMORY

1. Blow gently on the chip, or hold a finger over the back of the board, and watch the reading rise.
2. Show the whole number and one decimal place.
3. Make the grid flash when the reading goes above a number you choose.

SAY IT BACK

- The thermometer measures the chip itself, in quarter-degree steps.

CHECK YOURSELF

1. What does the chip's thermometer measure?
2. In what steps does the reading change?
3. How does the program turn 27 into two pictures?

The outside world

YOU ALREADY KNOW

You can switch a pin high and low, and you have a breakout board with labelled pins.

YOU WILL BE ABLE TO

Drive a pin on the edge connector, and say what a pin can and cannot carry.

PICTURE THIS

Every machine eventually has to touch something that is not itself. A lamp, a relay, a sensor, a motor. The edge of the micro:bit, the gold rings and the row of tiny connections beside them, is where the engine reaches out.

Your breakout board takes each of those connections and brings it out to a row of sockets in groups of three, one for the signal, one for power and one for ground. The labels printed on the back, such as P0, P1, P2 and so on, are the names the program uses.

```
terminal
```

```
cargo run --example edge-pin
```

Program written for this book.

```

#![no_main]
#![no_std]

use embedded_hal::delay::DelayNs;
use embedded_hal::digital::OutputPin;

use cortex_m_rt::entry;
use microbit::{hal::gpio, hal::Timer, Board};
use panic_halt as _;

#[entry]
fn main() -> ! {
    let board = Board::take().unwrap();
    let mut timer = Timer::new(board.TIMER0);

    // Pin 0 on the edge connector (the big gold ring marked 0).
    // On your breakout board it is the row labelled P0.
    let mut pin0 = board.edge.e00.into_push_pull_output(gpio::Level::Low);

    loop {
        pin0.set_high().unwrap(); // 3.3 volts on the pin
        timer.delay_ms(1000_u32);
        pin0.set_low().unwrap(); // 0 volts on the pin
        timer.delay_ms(1000_u32);
    }
}

```

WHAT YOU SHOULD SEE

Pin 0 goes to 3.3 volts for one second, then to 0 volts for one second, repeating. You can see it with a small lamp and a resistor between the P0 signal socket and ground, or with a multimeter on the pin.

Reading the program

- `board.edge.e00` is edge pin 0, the big gold ring marked 0. Your breakout labels it P0.
- `into_push_pull_output` turns it into a switch your program controls, just like the lamp pin in chapter 7.
- `set_high()` puts 3.3 volts on it and `set_low()` puts 0 volts on it.

What a pin can and cannot do

A pin is a whisper, not a shout. It can switch a small lamp with a resistor, or tell something bigger to switch on. It cannot carry the current of a motor, a relay coil or a horn. If you connect one of those directly, you can damage the chip.

So the pin whispers to a driver, and the driver, with its own power supply, does the heavy work. That is the way every big machine in this book will be moved. Three rules go with it:

1. The heavy load gets its own supply, and never draws from the micro:bit.
2. The driver and the micro:bit must share a ground, or the whisper cannot be understood.
3. Do not put 5 volts on any signal pin. The micro:bit's pins work at 3.3 volts.

KEY IDEA

A pin carries a signal, not a load. Drivers do the heavy work, and every circuit shares a ground.

FROM MEMORY

1. Make pin 0 blink at five times a second.
2. Make pins 0 and 1 take turns: when one is high the other is low.
3. Write on paper the three rules for connecting something heavy.

SAY IT BACK

- A pin carries a signal. A driver with its own power carries the load.
- Share a ground. Keep 5 volts off the signal pins.

CHECK YOURSELF

1. Which name on your breakout is the same pin as edge pin 0?
2. What voltage does a pin put out when it is high?
3. Why must the driver and the micro:bit share a ground?

What is not here yet

YOU ALREADY KNOW

You have driven the lights, buttons, timer, speaker, serial port, tilt sensor, thermometer and edge pins.

YOU WILL BE ABLE TO

Know what is missing from this book, and why.

An honest book lists what it has not covered. Here is the list.

PICTURE THIS

A new vehicle comes with a list of features. A careful dealer shows you what he has driven and tested, and tells you plainly which ones he has not.

PART OF THE BOARD	WHY IT IS NOT HERE YET
The microphone	It needs the chip's analog input and a power pin for the microphone. I could not find a tested example to adapt, and I will not guess.
The touch logo	The gold logo senses a finger with a touch circuit. I could not find a tested example for it in these libraries.
The compass	It uses the same sensor and the same two wires as the tilt sensor, but it needs calibration to give a true heading. It comes next.
The radio	One micro:bit talking to another. A chapter of its own, after the others are solid.
Interrupts	The chip tapping the program on the shoulder. This is the better answer to the problem in chapter 6, and it is the next big idea.
Motors	Needs a driver board with its own power. We start with the signals in chapter 12 and build on them.

How to know you are ready

The test is simple. For each chapter, can you rebuild its program from a blank file without opening the book? Can you draw its picture for a technician? Can you break it on purpose and say why it broke? If you can, you can teach it.

KEY IDEA

This book covers what it has tested. The rest is on the list, and it gets done properly.

SAY IT BACK

- Microphone, touch logo, compass, radio, interrupts and motors come next.

CHECK YOURSELF

1. Name three parts of the board this book has not covered yet.
2. What is the reason for each gap?

code::core

Every program in this book passed the Rust type-checker against the real micro:bit v2 libraries before it was published. None was run on the board by the author of this page, so the first run on your board is the real test. Some programs are adapted from the micro:bit v2 Embedded Discovery Book by the Rust Embedded project; those are marked, and the book's own licence applies to them. The explanations, pictures and questions are new.