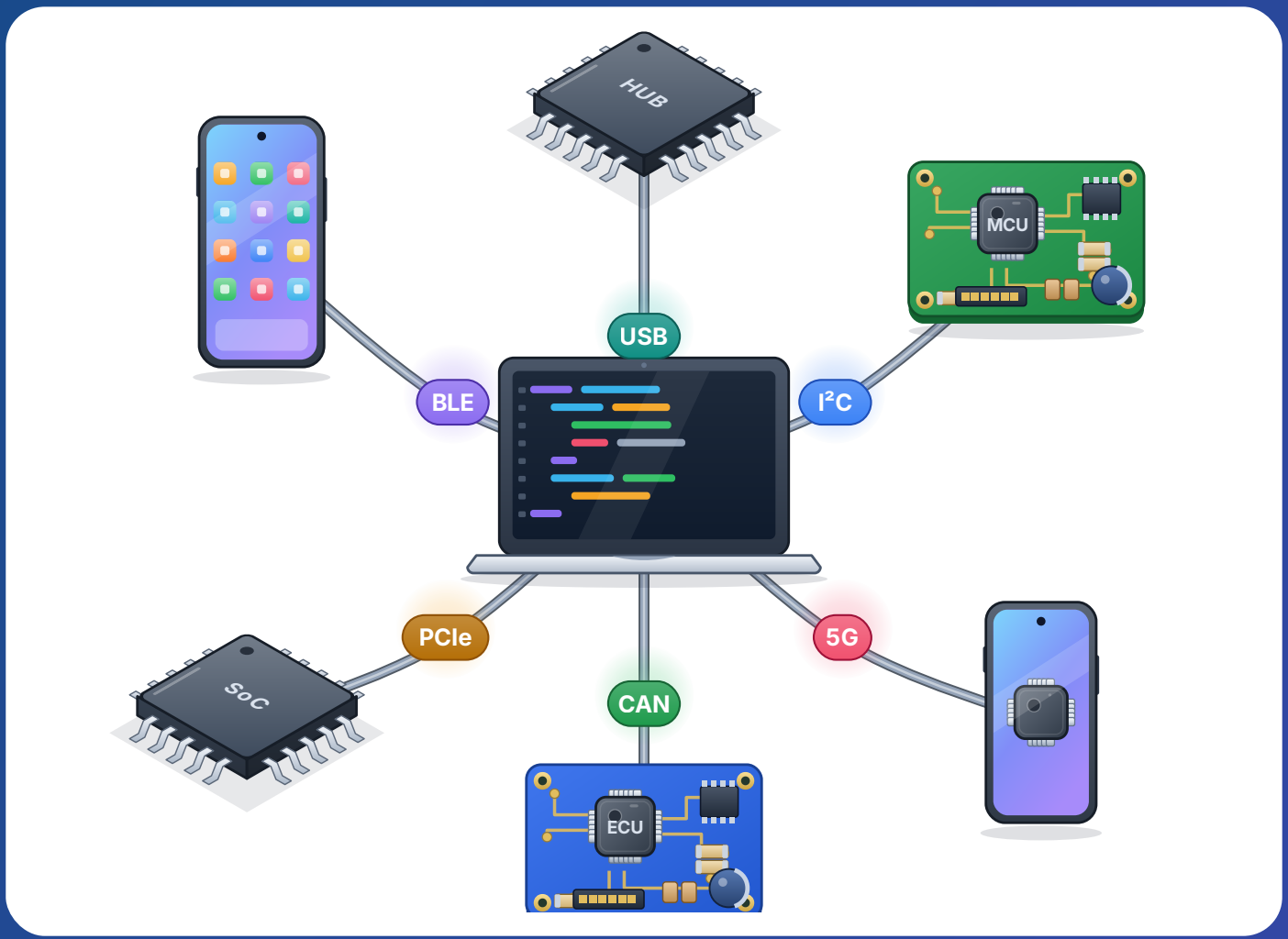
THE PROTOCOL LIBRARY

START HERE · FREE GUIDE

The Protocol Map

How devices talk, the map of all 40 books, and your free protocol bench

What a protocol is · Where protocols live · The library map · The Protocol Passport · UART in one chapter · Reading a spec · Your bench



Labs run on Python 3.13, OSS CAD Suite 2026-11-12 (Icarus Verilog 14), sigrok-cli 0.7.2 and tshark 4.4.

FIRST EDITION · 2026

THE PROTOCOL LIBRARY · START HERE

The Protocol Map

What protocols are, where they live, how this library teaches them, and your free bench

First Edition · v1.0 · 2026

The Protocol Map: What protocols are, where they live, how this library teaches them, and your free bench

First Edition (v1.0), 2026. Labs run on Python 3.13, OSS CAD Suite 2026-11-12 (Icarus Verilog 14), sigrok-cli 0.7.2 and tshark 4.4.

Copyright © 2026. Published by The Protocol Library.

This guide is free. You may copy and share it, in full and unchanged, but you may not sell it or publish changed versions of it. All other rights reserved.

The information in this book is provided "as is", without warranty of any kind. The author and publisher assume no responsibility for errors or omissions, or for damages resulting from the use of the information herein. Specifications are explained in the author's own words; for conformance and certification work, always use the official specification of the standards body.

Bluetooth, Wi-Fi, USB, HDMI, DisplayPort, Thunderbolt, Zigbee, Thread, Matter, EtherCAT, PROFINET, PROFIBUS, Modbus, CANopen, MIPI and other names in this book are trademarks or registered trademarks of their respective owners. They are used only to name the technology. This book is independent and is not affiliated with, sponsored or endorsed by any standards body or company.

Typeset in Source Serif 4, Inter and JetBrains Mono. Diagrams are original and drawn for this book.

About This Book

This free guide opens **The Protocol Library**: forty books that teach every electronics and communication protocol an engineer meets, deeply enough to answer any question about it. Before you study any one protocol, it shows you the whole map, the method every book uses, and one complete protocol, UART, taught the way the series teaches all of them.

- **Every example was really run.** Each code listing, simulation log, decoded capture and packet dissection in this book comes from a real run of free, open-source tools. No output was typed in by hand. Labs for real hardware are marked *On your bench · not run by the author*.
- **Tools used:** Labs run on Python 3.13, OSS CAD Suite 2026-11-12 (Icarus Verilog 14), sigrok-cli 0.7.2 and tshark 4.4.
- **The code bundle** that comes with this book holds every lab, ready to run, in the folder `labs/00/`.
- **Answers** to every "Check yourself" question and every exercise are in Appendix B, away from the questions, so you can try first.

What you need before you start

This guide is where the series starts, so it needs nothing before it except school mathematics and curiosity. Every term is explained the first time it appears.

This is the first edition (v1.0). Tool versions and industry facts were checked in November 2026. Tools change quickly. Where a detail is likely to change, the text says so.

Contents

PART I · THE BIG PICTURE

1 **What a Protocol Is (and Is Not)** 16
- 1.1 The idea in plain words 17
- 1.2 Six words that people mix up 18
- 1.3 One reading, traced through every layer 21
- 1.4 The questions every protocol answers 24
- 1.5 Layers: a short look 25
- 1.6 See it on the wire 26
- 1.7 The Protocol Passport 28

2 **Where Protocols Live** 32
- 2.1 The idea in plain words 33
- 2.2 Inside a smartphone 34
- 2.3 Inside a car 36
- 2.4 Inside a factory 39
- 2.5 Inside a data centre 41
- 2.6 In the sky and in orbit 43
- 2.7 Why each place picks its protocols 44

3 **The Library Map** 48
- 3.1 The idea in plain words 49
- 3.2 Thirteen shelves, forty books 49
- 3.3 One home for every protocol 52
- 3.4 Depth tiers: how much each protocol gets 53
- 3.5 How every book teaches 54
- 3.6 How the books keep their promise 56
- 3.7 Book 1, the foundation 57
- 3.8 Study paths by career 58
- 3.9 When the books appear 59
- 3.10 How to use the series 59

PART II · THE METHOD

4 **UART in One Chapter: A Complete Sample Dossier** 63
- 4.1 How a protocol dossier works 64
- 4.2 The idea in plain words 65
- 4.3 Why the UART exists 66
- 4.4 Where you meet it 67
- 4.5 Roles and architecture 68
- 4.6 Physical layer: wires and voltage levels 69
- 4.7 Bit-level signalling 70
- 4.8 Frame format 72
- 4.9 Transactions and state machines 74
- 4.10 Timing and performance 75
- 4.11 Errors and recovery 79
- 4.12 Hardware design 81
- 4.13 Implementation 83
- 4.14 See it on the wire 84
- 4.15 Debugging playbook 85
- 4.16 Compare and choose 86
- 4.17 In industry 87
- 4.18 Hands-on labs 89
- 4.19 Where to go next 92

4.20 Cheat sheet 92

5 Reading Specifications and Datasheets 96

5.1 The idea in plain words 97
5.2 Who writes the standards 98
5.3 Free, sold or members first 98
5.4 Editions, amendments and errata 99
5.5 Normative language 101
5.6 How a specification is organised 102
5.7 How a datasheet is laid out 103
5.8 Three kinds of limits 103
5.9 Timing and characteristics tables 104
5.10 Worked example: finding one number 106
5.11 Register maps 107
5.12 A checklist for reading any spec 109
5.13 Lab: recompute the data sheet 109

6 Your Protocol Bench 112

6.1 The idea in plain words 113
6.2 The four free tools 113
6.3 How the tools connect 114
6.4 Install the bench 115
6.5 When setup goes wrong 118
6.6 Your first signal capture 119
6.7 Your first packet capture 122
6.8 An optional hardware kit 125
6.9 Safety first 126
6.10 What to read next 128

APPENDICES

A Final Test 130

A.1 What a protocol is (Chapter 1) 130
A.2 Where protocols live (Chapter 2) 131
A.3 The library map (Chapter 3) 132
A.4 UART in one chapter (Chapter 4) 133
A.5 Reading specifications and datasheets (Chapter 5) 134
A.6 Your protocol bench (Chapter 6) 135

B Answers 137

C Revision Cards 152

D Glossary 155

List of Figures

1.1 A radio call between two guards. 17

1.2 The same signal read with two sets of rules (Lab 1.1). 18

1.3 Six different things that all get called “protocols”. 19

1.4 The greenhouse link. 21

1.5 One reading in five forms, from Lab 1.1. 22

1.6 The five questions, answered for the greenhouse link. 24

1.7 The greenhouse exchange as a sequence chart. 25

1.8 Two stacks in the same system. 26

1.9 The capture from Lab 1.1. 27

1.10 The 16 fields of the Protocol Passport, grouped by the question they answer. 29

1.11 This chapter at a glance: a protocol is an agreement; six words that are often mixed up; one reading in five forms; the five questions; layers; a capture read with the right and wrong rules; the Protocol Passport. 29

2.1 Protocols sorted by the distance they cover, from inside a chip to across the world, with the books that teach them. 33

2.2 Inside a smartphone (simplified). 34

2.3 The phone’s radio links. 35

2.4 Inside a car, seen from above (simplified). 37

2.5 The OBD-II socket seen from the front. 38

2.6 A small factory (simplified). 39

2.7 The levels of factory automation and the protocols typical of each. 40

2.8 Inside one server (simplified). 41

2.9 A data-centre hall (simplified). 42

2.10 An airliner, a satellite and a drone, with their ground stations. 43

2.11 Where this chapter’s tours met some well-known protocols, with the book that teaches each one. 45

2.12 This chapter at a glance: protocols sorted by distance, and the main links of a phone, a car, a factory, a data centre, and an aircraft and a satellite. 45

3.1 Finding one fact in the library takes three steps: the shelf for the subject, the one book that teaches the protocol, and the part of the dossier that holds the answer (here P6, the frame format). 49

3.2 The library wall: this free guide and the forty books on their thirteen shelves. 50

3.3 A single-board computer of the Raspberry Pi 5 kind, drawn simply and not to scale. 52

3.4 One home per protocol. 53

3.5 The four depth tiers. 54

3.6 The 22 parts of the Protocol Dossier, plus the Passport (P0), grouped by what they do. 55

3.7 How every chapter is built: an opener, teaching sections with checks and worked examples, a lab, and a review. 55

3.8 The blind test. 56

3.9 Book 1 at a glance: 974 pages in five parts and 22 chapters, with 502 figures and 71 desk labs. 57

3.10 Study paths for eight careers. 58

3.11 The order in which the books appear: three waves, essentials first. 59

3.12 This chapter at a glance: the wall of books, one home per protocol, the tiers, the dossier, the proof behind each book, Book 1, the study paths and the waves. 60

4.1 The 22 parts of the Protocol Dossier, plus the Passport (P0), in reading order. 64

4.2 The lamp code for the letter ‘H’. 66

4.3 Milestones from start-stop telegraphy to the UARTs inside every microcontroller. 67

4.4 A typical board and its UART links: a GPS module, a radio module, a console header and a PC through a USB–UART bridge. 67
4.5 Inside a UART. 68
4.6 Wiring two UART devices. 69
4.7 The same frame (the letter 'H') at logic level and on an RS-232 cable. 70
4.8 The letter 'H' (0x48) sent at 115 200 baud, 8N1. 71
4.9 16× oversampling. 72
4.10 The letter 'H' in three formats. 73
4.11 The UART receiver's state machine. 74
4.12 A host and a module exchange AT-style commands over a UART. 75
4.13 Where the receiver samples when its clock is slow. 77
4.14 The same captured "Hi!" decoded by sigrok with the receiver's baud rate offset from −7 % to +7 %. 78
4.15 The four UART errors. 80
4.16 A real break captured in Lab 4.1. 81
4.17 Three common UART hardware fixes. 82
4.18 The real TX line from Lab 4.1 and sigrok's decode. 85
4.19 UART, SPI, I²C and USB side by side. 87
4.20 The UART frame (here 8E1, the letter 'H'): idle high, start 0, data LSB first, parity, stop 1. 93
4.21 This chapter at a glance: the UART's passport, frame, maths, sampling, errors, wiring and capture, and where to go next. 94
5.1 A specification is one rule book that every product of a kind follows; a datasheet is one maker's promise about one product, with its limits. 97
5.2 Who writes the specifications in this library, with two examples each and the books that teach them. 98
5.3 Three kinds of access. 99
5.4 How to read six document identifiers. 100
5.5 A mock page of a made-up standard. 101
5.6 The normative words, from binding to not binding, with what each one asks of an implementer. 102
5.7 How a datasheet is laid out, from the front page to the back pages. 103
5.8 The three kinds of limits, drawn for the TMP117 supply. 104
5.9 A mock electrical-characteristics table for a made-up part. 105
5.10 The receiver's operating range for each frame size, computed by Lab 5.1 from the data sheet's formula. 107
5.11 A real register map, redrawn: the 16550 UART's registers by address, and the bit fields of its line control register (LCR). 108
5.12 Two checklists: one for a specification and one for a datasheet. 109
5.13 This chapter at a glance: specifications and datasheets, who writes them, how to get them, editions, the normative words, the three kinds of limits, register maps and checking a number. 110
6.1 The four jobs of every protocol bench, with the reporter's tools in brackets. 113
6.2 The four tools of the desk bench and the command you type for each. 114
6.3 The two pipelines of the desk bench. 115
6.4 Three ways to the same desk bench. 115
6.5 Your first capture, drawn from Lab 6.2's VCD file and sigrok's annotations. 119
6.6 The first frame of the capture as the VCD describes it. 120
6.7 Lab 6.3's capture as tshark dissects it. 123
6.8 Packet 1 as nested envelopes, drawn from tshark's PDML. 124
6.9 A low-cost kit for the bench boxes of this library: a development board, an 8-channel USB logic analyzer, a USB–UART adapter, a multimeter, a breadboard and jumper wires. 126
6.10 Which signals the kit may touch. 126

6.11 The wiring for your first real capture. 127

6.12 This chapter at a glance: the four jobs of a bench, the four free tools, their two pipelines, installing and checking them, your first two captures, the hardware kit and its safety rules, and Book 1 as the next step. 128

List of Labs

1.1 The greenhouse capture 28
4.1 UART on the wire 89
4.2 UART timing maths 90
4.3 Two programs on a virtual serial cable 91
5.1 Recompute the numbers in a data sheet 109
6.1 Check your bench 117
6.2 Your first signal capture 121
6.3 Your first packet capture 124

How to Use This Book

This book is written so that a careful reader keeps about **90 %** of it after one pass. That only works if you *do* things while you read. Read the chapters in order, and in each chapter follow the same rhythm:

1. **Warm up.** Answer the three warm-up questions from memory before you read on.
2. **Read with a pencil.** When a *Check yourself* box appears, stop and answer it before you continue.
3. **Do the lab.** Type the commands and compare your output with the output printed in the book.
4. **Do the exercises.** At least the Recall and Apply groups. Then compare with Appendix B.
5. **Finish the book with the final test** (Appendix A). Score 90 % or more and you are ready for the next book. Below that, each answer tells you exactly which section to re-read.

Every chapter opens with a strip that shows which parts of the communication stack it works on, from the electrical signal up to the application:

WHERE WE ARE — Signal | Line coding | **Framing** | Error control | Access | Network | Transport | Application | Measure & debug

Section headings carry a level: BASIC INTERMEDIATE ADVANCED. On a first reading, read every Basic and Intermediate section. Advanced sections and *Deep dive* boxes can wait. The next chapter never depends on them.

You will meet these boxes:

> **WARM-UP**
>
> Three quick questions on earlier chapters, so old ideas stay fresh.

> **IN THIS CHAPTER**
>
> › 2.1 What you will be able to do by the end. Each line is a numbered **learning outcome**, tested in the final test.

> **Definition**
>
> A new term, explained in plain words the first time it appears. It is also in the glossary.

> **Check yourself**
>
> A question to answer before you read further. The answer is in Appendix B.

WORKED EXAMPLE

A problem with numbers, a circuit or code, solved one step at a time.

WHAT THE HARDWARE BECOMES

How the idea is built in real hardware: the circuit, the parts, the PCB rules or the logic inside a chip.

Common mistake

What beginners often get wrong, why it is wrong, and the right way.

In industry

How engineers use this at work: in which products, with which tools and test equipment.

PROTOCOL PASSPORT

A one-page identity card for each protocol: who made it, its wires, speed, distance, topology, error check and where you meet it.

SEE IT ON THE WIRE

A real capture, decoded field by field, exactly as a logic analyzer or Wireshark shows it.

DEBUGGING PLAYBOOK

Symptom, likely cause, what to measure and how to fix it.

On your bench · not run by the author

Steps for your own hardware. The author ran every other lab, but could not run this one on real hardware.

Commercial tool · shown for recognition, not run

Commands of a paid tool, shown so you recognise them at work. Every lab in the book runs on free tools.

HANDS-ON LAB

Goal, what you need, exact steps, the output you should see, and what to try if it does not work.

Interview

Questions an interviewer could ask on the topic, with model answers.

Commands you type are printed in a `monospaced font`. Blue references such as *Chapter 2* or *Figure 2.3* are links in the PDF.

Learning Outcomes

When you finish this book, you will be able to do these 35 things. Each one is taught in its chapter, practised in that chapter's exercises or lab, and tested in the final test (Appendix A).

Chapter 1 · What a Protocol Is (and Is Not)

1.1 Explain what a protocol is, and name its three kinds of rule: format, meaning and timing.

1.2 Tell a protocol from an interface, an electrical standard, a network standard, a data format and a security algorithm, and correct the common mix-ups.

1.3 Trace one sensor reading through every layer of a real link: data format, Modbus RTU, CRC, UART and RS-485.

1.4 Answer the five questions every protocol answers (who, when, how fast, what format, what if it fails) for a real link.

1.5 Explain why protocols are built in layers, and say which layers change when a link moves to a new medium.

1.6 Say how the Protocol Passport of this library turns the five questions into 16 fields.

Chapter 2 · Where Protocols Live

2.1 Name the main links inside a smartphone and the protocol on each, and the phone's radio links.

2.2 Name the buses of a modern car and say what each carries, including the diagnostics port.

2.3 Describe the levels of a factory network, from 4–20 mA sensors up to OPC UA and MQTT.

2.4 Name the protocols inside a server and between the servers of a data centre.

2.5 Name the data links of an airliner, a satellite and a drone.

2.6 Explain how distance, speed, cost, power and safety decide which protocol a link uses, and find the book that teaches it.

Chapter 3 · The Library Map

3.1 Find the shelf and the book that teach a protocol, using the library wall and the home-book rule.

3.2 Explain what the depth tiers A, B, C and D give a protocol, and what each means for you as a reader.

3.3 Say where in the Protocol Dossier to look for a frame format, a timing number, a fix or a summary.

3.4 Describe how each book proves its promise: outcomes, desk labs with real output, question universes, blind tests and a final test.

3.5 Choose a study path for a career and plan the order in which to read the books.

Chapter 4 · UART in One Chapter: A Complete Sample Dossier

4.1 Fill in the Protocol Passport of a UART link and explain each entry.

4.2 Draw a UART frame (8N1, 7E1, 8O2) bit by bit and explain why it starts low and ends high.

4.3 Compute bit time, frame time, throughput efficiency and the largest baud-rate mismatch a receiver can tolerate.

4.4 Explain how a receiver finds the start bit and samples each bit (16× oversampling, mid-bit sampling).

4.5 Recognise parity, framing, overrun and break conditions and say what causes each.

4.6 Wire two UART devices correctly (TX to RX, common ground, matching logic levels) and spot the classic wiring mistakes.

4.7 Read a decoded UART capture and use the debugging playbook to find a fault.

Chapter 5 · Reading Specifications and Datasheets

5.1 Tell a specification from a datasheet, and name the body that writes a given standard.

- **5.2** Say whether a specification is free, sold or for members first, and find its edition, amendments and errata.
- **5.3** Read normative language (shall, should, may, can, NOTE) and the structure of a standard.
- **5.4** Find your way around a datasheet: absolute maximum ratings, operating conditions, electrical and timing characteristics, register maps.
- **5.5** Find one number in a datasheet, check it against its formula and use it in a design.

Chapter 6 · Your Protocol Bench

- **6.1** Name the four free tools of the desk bench, say what each one does, and draw how they connect.
- **6.2** Install the bench on Debian or Ubuntu, or on Windows through WSL 2, and check that it works.
- **6.3** Run a simulated UART capture, decode it with sigrok, and read its VCD times and sample numbers by hand.
- **6.4** Build packets with scapy, dissect them with tshark, and count a packet's bytes layer by layer.
- **6.5** Choose a low-cost hardware kit and connect it safely: voltage levels, ground first, static, never mains.
- **6.6** Fix the common setup problems with the troubleshooting table.

PART I

The Big Picture

What a protocol is, where protocols live, and how this library is organised.

1 What a Protocol Is (and Is Not)

2 Where Protocols Live

3 The Library Map

CHAPTER 1

What a Protocol Is (and Is Not)

WHERE WE ARE: **Signal** | Line coding | Framing | Error control | Access | Network | Transport | Application | Measure & debug



Every device you own talks to other devices. A sensor talks to a controller, a phone talks to a headset, a car's brake unit talks to its engine unit, and a server talks to the whole world. None of this works unless both sides follow the same rules. This chapter explains what those rules are, what they are called, and why engineers so often use the wrong name for them. You will follow one real temperature reading from a greenhouse sensor all the way to the wire and back, and in the lab you will decode it yourself.

WARM-UP

1. What number is 0xF5 in decimal?
2. Eight switches can each be on or off. How many different patterns can they show together?
3. Name two devices in your home that send information to each other without a cable.

IN THIS CHAPTER

1.1 Explain what a protocol is, and name its three kinds of rule: format, meaning and timing.

1.2 Tell a protocol from an interface, an electrical standard, a network standard, a data format and a security algorithm, and correct the common mix-ups.

1.3 Trace one sensor reading through every layer of a real link: data format, Modbus RTU, CRC, UART and RS-485.

1.4 Answer the five questions every protocol answers (who, when, how fast, what format, what if it fails) for a real link.

1.5 Explain why protocols are built in layers, and say which layers change when a link moves to a new medium.

1.6 Say how the Protocol Passport of this library turns the five questions into 16 fields.

1.1 The idea in plain words BASIC

Picture two guards at a large farm, one in the control room and one out in the fields. They talk by walkie-talkie. Only one of them can speak at a time, because a walkie-talkie sends only while its button is pressed. So they follow a few rules. Each call starts with a name, so the right person answers. Each speaker ends with "over", which hands the turn to the other side. Both speak slowly and clearly. Numbers are said digit by digit. And if a word is lost in noise, the listener says "say again".

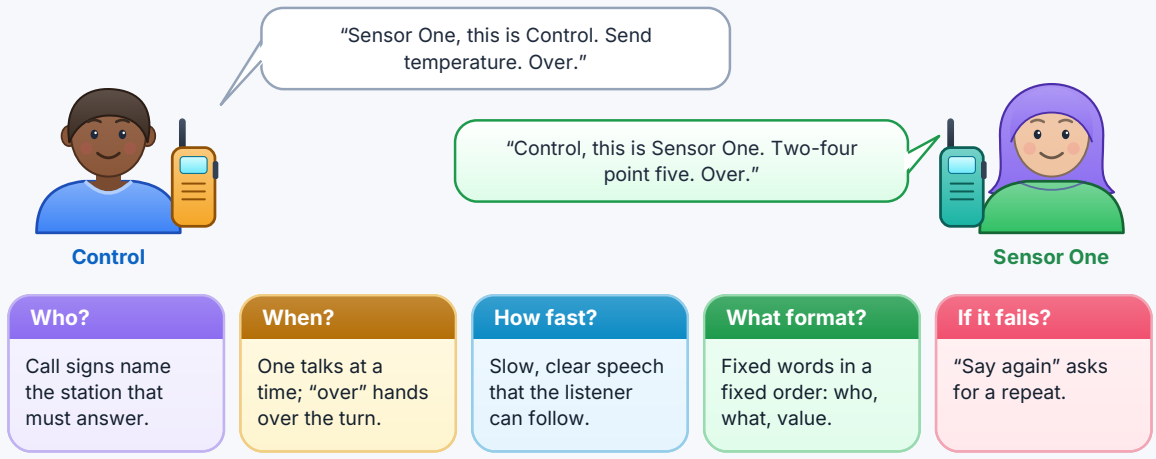


Figure 1.1 A radio call between two guards. Before they speak, they already share five rules: whom a call is for, whose turn it is, how fast to speak, what words to use and in what order, and what to do when a word is lost.

People learn rules like these quickly and can fix small mistakes with common sense. Machines have no common sense. Two chips that must exchange data need *every* rule written down and built in before they start: how a message begins, how long each bit lasts, which bit comes first, what each byte means, who may send when, and what to do when something goes wrong. That full set of rules is a **protocol**.

Definition **Protocol**

A **protocol** (more fully, a communication protocol) is a set of rules, agreed in advance by every side, that says how messages are formatted, what they mean, in what order they are exchanged, with what timing, and what each side does when something goes wrong.

The rules come in three kinds:

- **Format** (engineers also say **syntax**): which parts a message has, how long each part is, in what order the bits go.
- **Meaning** (**semantics**): what each part says, and what the receiver must do about it.
- **Timing**: how fast bits are sent, how long to wait for an answer, and how long a pause ends a message.

What happens when two devices do *not* share the rules? Lab 1.1 tries it. A temperature sensor sends its answer at 9600 bits per second. A receiver set to the same rules reads the reading back. A receiver set to twice that speed sees exactly the same voltages on the same wire, and turns them into nonsense.

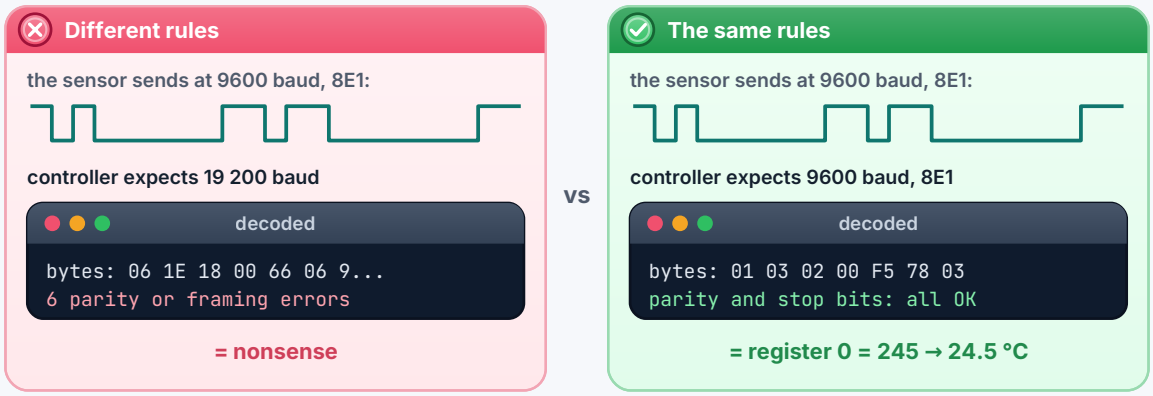


Figure 1.2 The same signal read with two sets of rules (Lab 1.1). With the sensor's own rules the receiver gets 01 03 02 00 F5 78 03, which holds the value 245, or 24.5 °C. A receiver that expects twice the speed sees the same wire and reports different bytes and six errors.

KEY IDEA

A protocol is an agreement made before the first bit is sent. A receiver understands a signal only if it applies the same rules of format, meaning and timing that the sender used, and it cannot ask the sender which rules those were.

Check yourself

1. In the radio call, which rule answers "whose turn is it?", and which one handles a lost word?
2. Give one rule of format and one rule of timing from the radio call.

1.2 Six words that people mix up BASIC

Engineers often say "protocol" for anything with a name: "the RS-485 protocol", "the USB-C protocol", "the AES protocol". These phrases sound harmless, but they cause real mistakes, such as buying the wrong part or wiring two devices that can never talk. Figure 1.3 separates six words that the owner's list of topics for this library sets side by side.

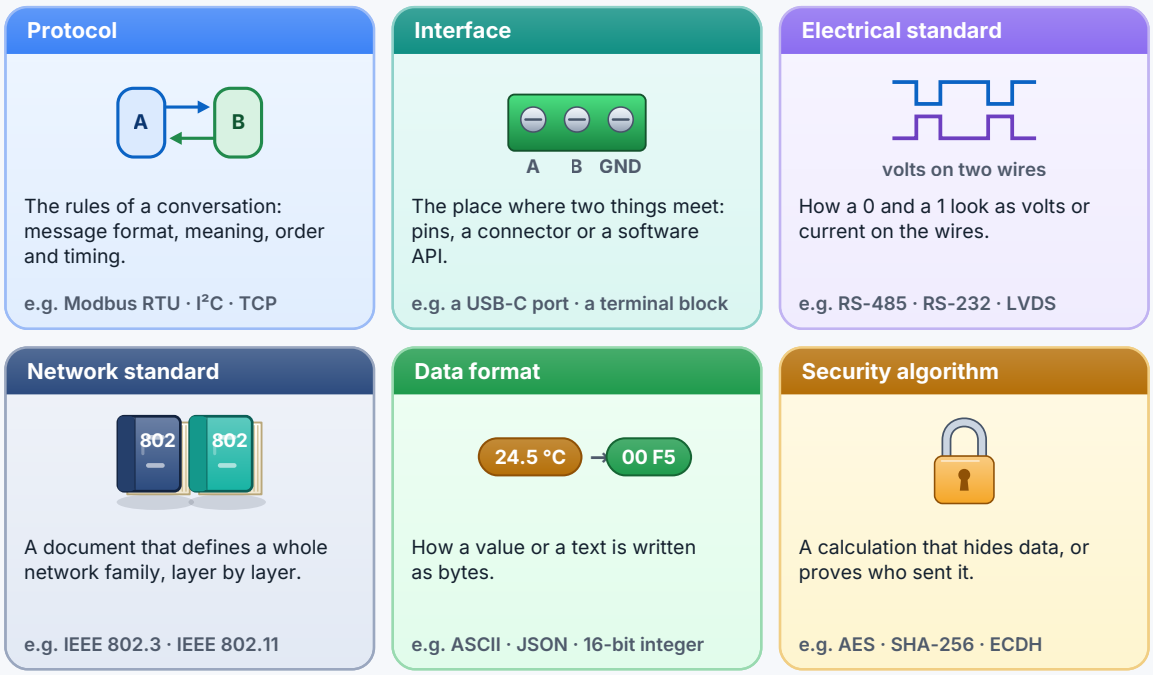


Figure 1.3 Six different things that all get called "protocols". Only the first one is a protocol. The others are a meeting place (interface), a voltage rule (electrical standard), a whole family of network rules (network standard), a way to write values as bytes (data format) and a calculation (security algorithm).

Word	It defines	It does not define
Protocol	messages: their format, meaning, order and timing, and error handling	the voltages or the connector
Interface	where two things meet: pins, terminals, a connector, or a set of software functions	what is said through it
Electrical standard	how a 0 and a 1 look: voltages or currents, thresholds, sometimes cable and distance	what the bits mean, who talks when
Network standard	a whole network family in one document: physical layers, frames, access rules, management	the applications on top
Data format	how a value or a text is written as bytes	how the bytes travel
Security algorithm	a calculation that hides data (encryption) or proves who sent it and that it was not changed	messages, turns or timing

Now the common mix-ups, corrected.

"The RS-485 protocol." RS-485 (formally TIA-485-A) is an **electrical standard**. It says how a driver puts a difference voltage on a pair of wires, and how much difference a receiver needs. It says nothing about bytes, addresses or turns. Two RS-485 devices can run completely different protocols on top, such as Modbus RTU, BACnet MS/TP or a maker's own, and then they never understand each other.

"The USB-C protocol." USB-C is a connector and cable: an **interface**. Many protocols travel through it: USB 2.0 and USB 3.x data, USB4, DisplayPort video in some ports, and USB Power

Delivery, the protocol a phone and a charger use to agree on a higher voltage. Two USB-C ports can look the same and support different protocols. Book 30 teaches them all.

"The Wi-Fi protocol." Wi-Fi products follow IEEE 802.11, a **network standard**: one large document with several radio physical layers, the frame formats, the rules for sharing the air, and the joining and security procedures. Each of those is a protocol of its own. "Wi-Fi" itself is the name used by the Wi-Fi Alliance, the industry group that certifies products; "Wi-Fi 6" is its name for 802.11ax. Ethernet works the same way, with IEEE 802.3 as its network standard (Books 13 and 19).

"The JSON protocol." JSON, like ASCII, is a **data format**. It says how to write values as text; an application protocol such as HTTP or MQTT carries it.

"The AES protocol." AES is a **security algorithm**, an encryption method. Security *protocols* such as TLS and WPA3 decide when to use it, with which key, and what to do with the result (Book 35).

Definition **Interface**

An **interface** is the boundary where two things connect: in hardware, a set of pins or terminals and their signals, often with a connector; in software, the set of functions one part of a program offers to another. The interface says *where* the conversation happens; the protocol says *what is said*.

Two more words help. A **standard** or **specification** is the *document* that writes rules down; one document can define several of the six things at once. USB 2.0, for example, defines its connectors, its voltages and its protocols. And a **checksum** or **CRC** (cyclic redundancy check) is an **error-check algorithm**: a calculation that catches accidental damage. It is not a security algorithm, because anyone who changes the data can simply compute a new CRC.

Common mistake **It is RS-485, so it will work**

Wrong: "The new meter has an RS-485 port, and so does our controller, so they will talk."

Why: RS-485 fixes only the voltages on the pair. The meter and the controller must also agree on the protocol, the speed, the character format and the addresses.

Right: "The meter speaks Modbus RTU at 9600 baud, 8E1, address 5, on a two-wire RS-485 pair." Now every layer is named.

Common mistake **A CRC keeps the data safe**

Wrong: "Our frames end with a CRC, so nobody can change them."

Why: A CRC catches noise, not attackers. Its calculation is public, so anyone who changes the data can attach a new, correct CRC.

Right: Use a CRC against noise, and a security protocol that uses a security algorithm (for example a message authentication code with a secret key) against attackers.

REAL-WORLD EXAMPLE

The EU charger law names an interface and a protocol

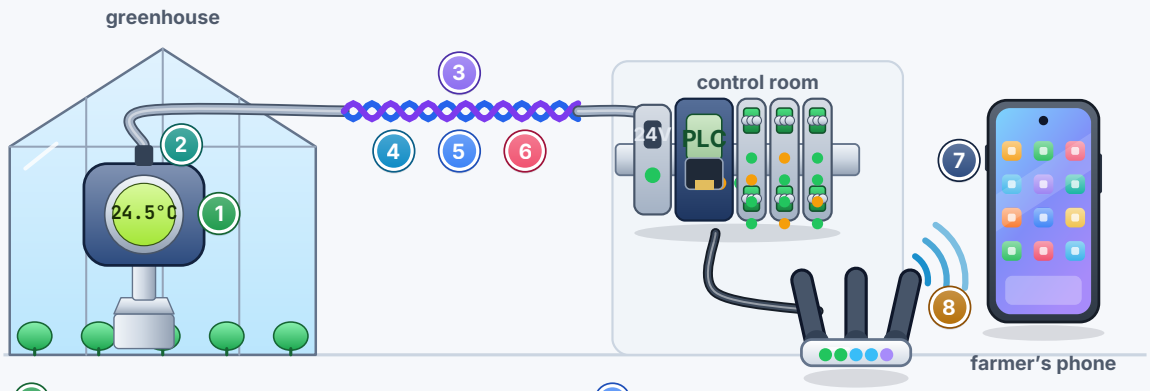
Since 28 December 2024 the European Union's common-charger rules (Directive (EU) 2022/2380) require phones, tablets, cameras and similar devices sold there to have a USB Type-C charging port, and since 28 April 2026 the same applies to laptops. The law asks for two separate things. The **interface** is the USB Type-C receptacle. For fast charging, a device that charges at more than 5 V, 3 A or 15 W must also support the **protocol** USB Power Delivery (EN IEC 62680-1-2), so that any compliant charger can agree on a higher voltage with it. The connector alone would not have been enough. (European Commission and ComReg guidance, checked November 2026.)

Check yourself

3. A data sheet says: "The module supports the TTL serial protocol at 3.3 V." Which part of the sentence is about a protocol, and which part is about an electrical level?

1.3 One reading, traced through every layer BASIC

Here is a real kind of system, used in farms, factories and buildings. A temperature sensor hangs inside a greenhouse. A controller in a control room, about 200 m away, reads it every few seconds. One twisted pair of wires joins them, and the farmer checks the value on a phone through the controller's Wi-Fi. Figure 1.4 marks where each of the six words lives.



1. Data format: 24.5 °C is sent as 245, the bytes 00 F5
2. Interface: the terminals A, B and GND on each device
3. Electrical standard: RS-485 on one twisted pair
4. Character format: UART bits, 9600 baud, 8E1
5. Protocol: Modbus RTU, one request and one answer
6. Algorithm: a CRC-16 checks every frame
7. Network standard: IEEE 802.11 (Wi-Fi) to the phone
8. Security: WPA2/WPA3 protect the radio hop with AES

Figure 1.4 The greenhouse link. The sensor and the controller share a twisted pair; the farmer's phone reaches the controller over Wi-Fi. Each number marks where one of this chapter's words lives in the system.

The controller wants the temperature, so it sends a request: “sensor 1, send me register 0”. The sensor answers with its reading. Follow that answer, 24.5 °C, from the top of Figure 1.5 to the bottom.

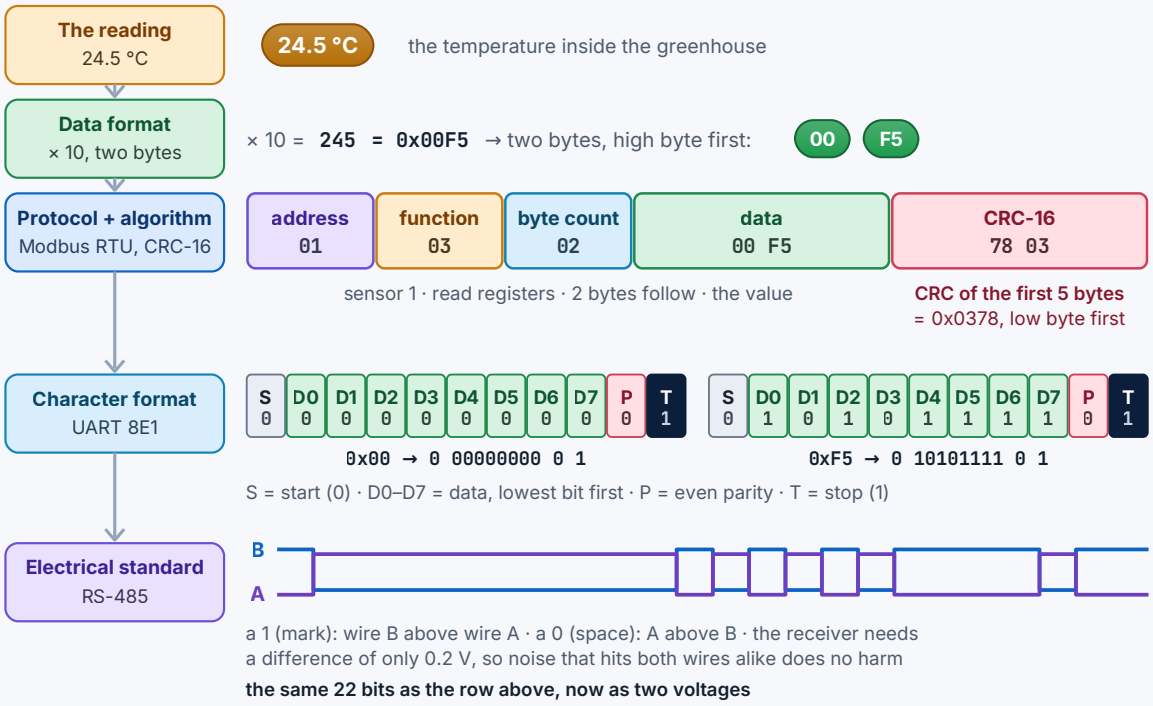


Figure 1.5 One reading in five forms, from Lab 1.1. The value becomes two bytes, the bytes go into a Modbus RTU frame with a CRC, each byte becomes an 11-bit UART character, and each bit becomes a voltage difference on the RS-485 pair.

1. **Data format.** The sensor stores temperatures as whole numbers of tenths of a degree, so 24.5 °C becomes 245. In hexadecimal that is 0x00F5, and it is sent as two bytes, high byte first: 00, then F5. This rule belongs to the sensor’s maker, and its manual lists it in a **register map**.
2. **Protocol.** The sensor wraps the two bytes in a **Modbus RTU** answer: its own address (01), the function it is answering (03, “read holding registers”), the number of data bytes (02) and the data (00 F5). Modbus RTU is a protocol from 1979 that is still used in meters, drives and sensors; Book 10 teaches it.
3. **Algorithm.** Last comes a 16-bit CRC, worked out from the first five bytes with the algorithm called CRC-16/MODBUS. It is sent low byte first: 78 03. The controller repeats the calculation; if the result differs, it throws the frame away.
4. **Character format.** Each byte travels as one 11-bit **UART character**: a start bit (0), the eight data bits with the lowest bit first, an even-parity bit and a stop bit (1). This is the default format of Modbus RTU, written **8E1**, at 9600 bits per second. The UART character format is a small protocol of its own, and Chapter 4 teaches it completely.
5. **Electrical standard. RS-485** turns each bit into a voltage *difference* between the two wires, called A and B. For a 1, B is above A; for a 0, A is above B. A driver must make a difference of at least 1.5 V, and a receiver needs only 0.2 V, so a long cable that loses voltage,

or noise that hits both wires alike, does no harm. That is why RS-485 can reach about a kilometre.

6. **Interface.** The pair lands on screw terminals marked A, B and GND on both devices. Those terminals, with their names and functions, are the interface.

The controller does the same steps in reverse: voltages to bits, bits to bytes, bytes to a checked frame, the frame to the value 245, and the value to 24.5 °C. Lab 1.1 computes every form:

OUTPUT layers.txt

```
One reading, layer by layer (Lab 1.1)
reading          24.5 degC
data format      x10 -> 245 = 0x00F5 -> bytes 00 F5 (high byte first)
request  PDU     03 00 00 00 01  (function 3, from register 0, 1 register)
request  RTU     01 03 00 00 00 01 84 0A
response PDU     03 02 00 F5  (function 3, 2 bytes follow)
response RTU     01 03 02 00 F5 78 03
CRC-16/MODBUS    0x0378 over 01 03 02 00 F5, sent low byte first: 78 03
receiver check   CRC over all 7 bytes = 0x0000 -> no error found
UART 8E1         byte -> start | D0..D7 | parity | stop
```

The trip to the farmer's phone uses different layers. The controller sends the value over Wi-Fi, which follows the **network standard** IEEE 802.11. WPA2 or WPA3, the Wi-Fi security protocols, protect the radio hop with encryption based on the **security algorithm** AES. Nothing on the RS-485 pair is encrypted, which is one reason such cables are kept inside a plant.

Tip Labels on RS-485 terminals

Makers do not all use the names A and B the same way. If an RS-485 link gives garbage, try swapping the pair first; it does no damage. Lab 1.1 shows what a swapped pair decodes as.

KEY IDEA

One reading passes through several rule sets, each with its own job: a data format writes the value as bytes, a protocol wraps and checks the bytes, a character format turns them into bits, and an electrical standard turns the bits into voltages. Name each one separately.

Check yourself

4. A second sensor reads 21.0 °C and also stores tenths of a degree. Which two data bytes does it send?
5. Which part of the greenhouse link would change if the cable were replaced by a radio link, and which parts could stay the same?

1.4 The questions every protocol answers BASIC

The radio call answered five questions, and every protocol, from a two-wire sensor link to 5G, must answer the same five. When you meet a new protocol, look for its answer to each one. When two devices refuse to work together, one of these answers usually differs.

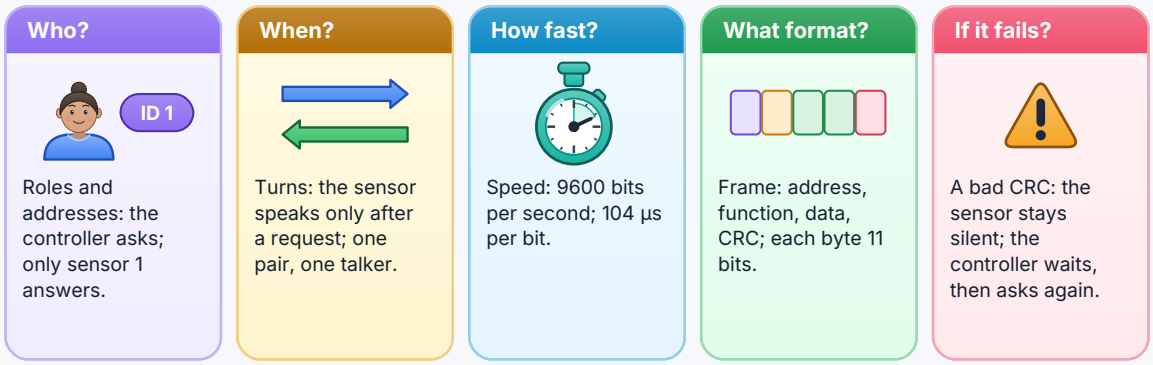


Figure 1.6 The five questions, answered for the greenhouse link. Who may speak, when, how fast, in what format, and what happens when a frame is damaged.

The table answers them for two real links: the greenhouse link, and a GPS module that sends its position to a microcontroller board, which Chapter 4 also uses.

Question	Greenhouse sensor (Modbus RTU over RS-485)	GPS module (NMEA 0183 text over UART)
Who? Roles and addresses	one controller (the client) asks; each sensor (a server) has an address from 1 to 247	one sender, one listener; no addresses needed
When? Who may send	a sensor sends only when asked; one talker at a time on the pair	the module sends whenever it has new data, typically once a second
How fast? Speed and timing	9600 bit/s here; a silence of 3.5 characters (4.01 ms) ends a frame	an agreed baud rate, often 9600
What format? Framing and data	address, function, data, CRC; 11 bits per byte	lines of text: `$`, comma-separated fields, `*`, a 2-digit checksum
What if it fails? Errors	a damaged request gets no answer; the controller times out and asks again	the listener drops a line with a bad checksum and waits for the next one

The two links answer “who?” and “when?” in opposite ways. The GPS module simply *pushes* data. The Modbus sensor is *polled*: it speaks only when asked, so two devices on the shared pair never talk at once. That simple rule also answers “when?” without any extra mechanism.

A **sequence chart** shows the order of messages in time, top to bottom, with one vertical line per device. Figure 1.7 shows the greenhouse exchange, and what happens when noise damages a request.

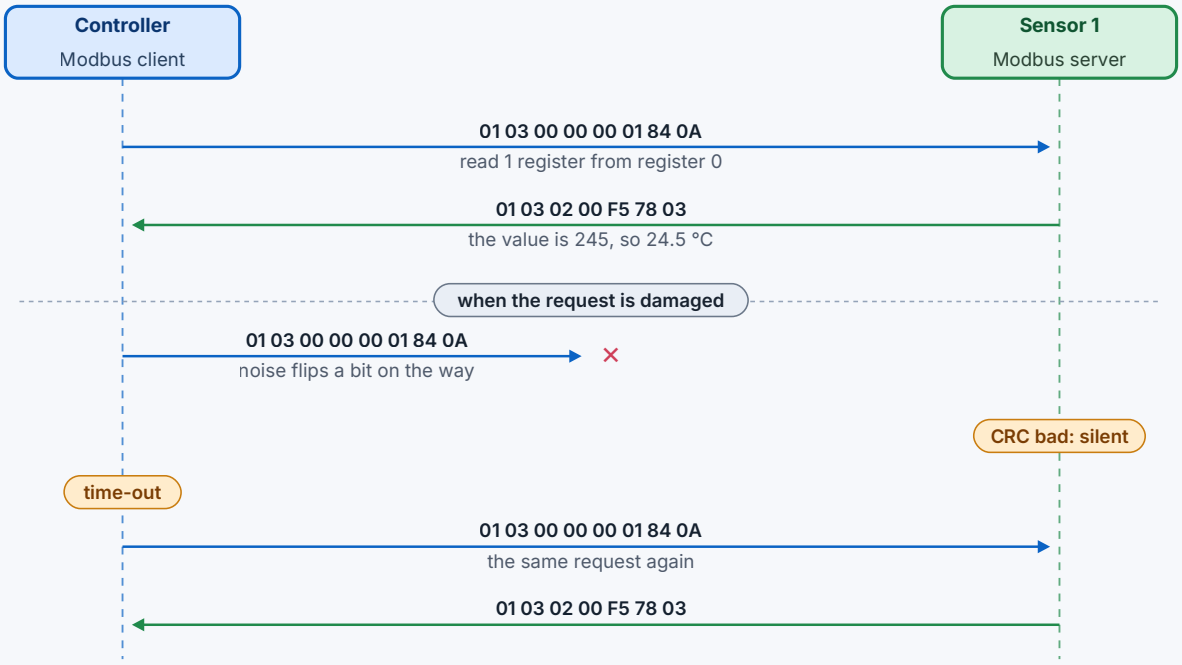


Figure 1.7 The greenhouse exchange as a sequence chart. Top: a request and its answer. Bottom: noise damages a request, so the sensor's CRC check fails and it stays silent; the controller waits for its time-out and sends the request again. Blue arrows come from the controller, green arrows from the sensor.

Why does the sensor stay silent instead of saying "your request was damaged"? Because it cannot trust anything in a damaged frame, not even the address. If it answered, two sensors might answer at once. Silence is safe, and the controller's **time-out** turns silence into a clear signal: "ask again".

KEY IDEA

Every protocol answers five questions: who may speak, when, how fast, in what format, and what happens when something fails. Learn a new protocol by finding its five answers.

Check yourself

6. On the greenhouse link, why can two sensors never answer at the same time?

1.5 Layers: a short look BASIC

You have seen that one link uses several rule sets at once. Engineers stack them as **layers**. Each layer solves one problem, uses the layer below it, and offers a service to the layer above. The value layer does not care how bits become volts; the RS-485 layer does not care what the bytes mean. So one layer can change without touching the others.

Definition **Layer and encapsulation**

A **layer** is one level of a communication system that solves one problem, such as turning bits into voltages or adding addresses. **Encapsulation** means that each layer takes the data of the layer above, adds its own parts (a header in front, sometimes a check at the end), and hands the result down.

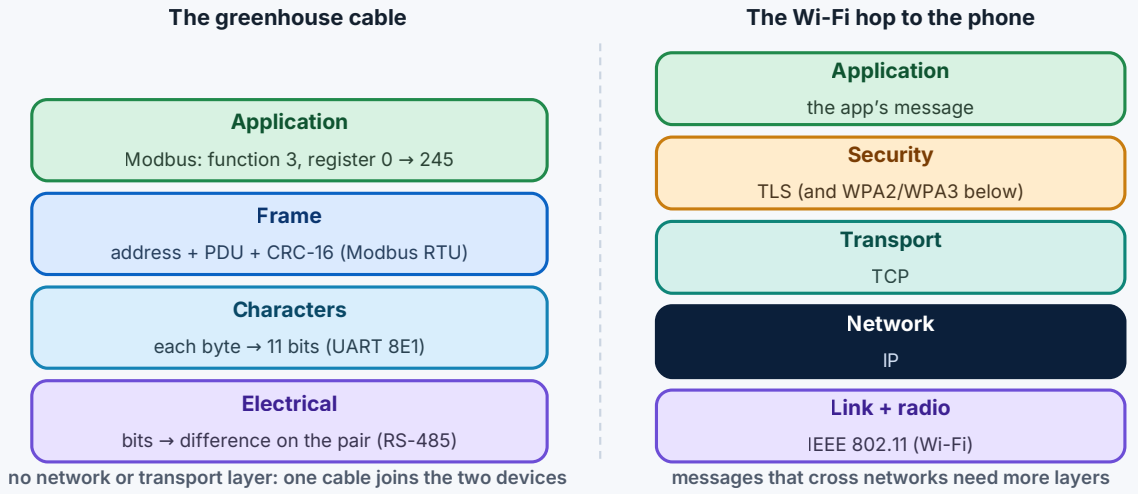


Figure 1.8 Two stacks in the same system. The greenhouse cable needs only four layers, because one cable joins the two devices. The Wi-Fi hop to the phone crosses networks, so it adds a network layer (IP), a transport layer (TCP) and a security layer.

Encapsulation is easy to see in the greenhouse answer. The Modbus part, 03 02 00 F5, is wrapped by the serial-line frame, which adds the address in front and the CRC behind; each byte is wrapped again by a start, parity and stop bit. The same Modbus part can also travel inside Ethernet and TCP, as **Modbus TCP**: then the lower layers change completely, and the top layer does not change at all.

The best-known layer model is the seven-layer **OSI model**. The Internet uses the four-layer **TCP/IP model**. Book 1 teaches both, and real stacks, in detail. In this library, a small strip under every chapter title, called **"Where we are"**, shows which layers that chapter works on: signal, line coding, framing, error control, access, network, transport and application, plus measuring and debugging.

1.6 See it on the wire BASIC

You can watch the whole greenhouse exchange in a simulated capture. In Lab 1.1, two small transmitters written in Verilog, a hardware description language, send the request and the answer. A simulator saves the line levels in the kind of file a logic analyzer makes, and the free decoder sigrok reads it with its `uart` and `modbus` decoders.

SEE IT ON THE WIRE

The greenhouse exchange, decoded

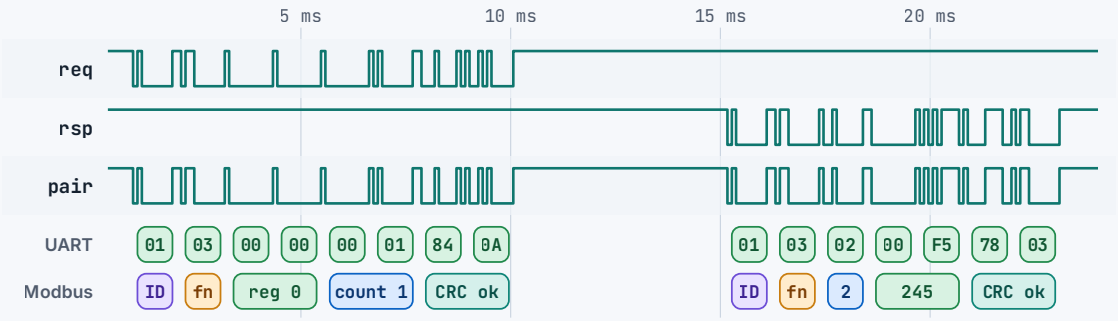


Figure 1.9 The capture from Lab 1.1. The controller's request leaves on req; 5 ms later the sensor's answer comes back on rsp; the pair row shows both, one after the other, as every receiver on the RS-485 pair sees them. The UART row shows each byte and the Modbus row the fields sigrok found, including both CRC checks.

DECODED CAPTURE modbus.txt

```
modbus-1: Slave ID: 1
modbus-1: Function 3: Read Holding Registers
modbus-1: Byte count: 2
modbus-1: 0x00F5 / 245
modbus-1: CRC correct
```

These are the decoder's lines for the answer; the request gets five similar lines. sigrok still uses the old word "Slave" for the Modbus server. The value 0x00F5 = 245 is the reading: 24.5 °C.

Now read the sensor's answer with wrong rules. Only one row gives the reading back:

OUTPUT wrong-settings.txt

```
Receiver's rules               Bytes decoded           Errors
-----------------------------  ----------------------  ------
9600 8E1 (the sensor's rules)  01 03 02 00 F5 78 03         0
19200 8E1 (twice as fast)      06 1E 18 00 66 06 9...       6
4800 8E1 (half as fast)        14 04 26                     8
9600 8N1 (no parity bit)       01 03 02 00 F5 78 03         5
9600 8E1, A and B swapped      7F FE 3F 15 0F F9 00         7
```

A wrong speed gives wrong bytes and many errors. A receiver that does not expect a parity bit reads it as the stop bit, so it gets the right bytes but reports a framing error on every byte whose parity bit is 0. A swapped pair inverts every bit and gives seven wrong bytes. The wire was the same each time; only the rules changed.

HANDS-ON LAB 1.1

The greenhouse capture

GOAL

Build the greenhouse frames, simulate them on the wire, and decode them with the right and wrong rules.

YOU NEED

Python 3.13, Icarus Verilog 14 or later (OSS CAD Suite), sigrok-cli 0.7.2 with its `uart` and `modbus` decoders, and the lab folder `labs/00/ch01/greenhouse-capture/`. Chapter 6 shows how to install them.

STEPS

1. Read `layers.py`: it builds both frames, computes the CRC and prints every form of the reading.
2. Read `uart_tx_8e1.v`, the transmitter from Chapter 4 with a parity bit added.
3. Run `bash labs/00/ch01/greenhouse-capture/run.sh`. It runs `layers.py`, simulates the two transmitters, decodes the capture and runs `wrong.py`.

EXPECTED RESULT

The files shown in this chapter appear in `out/`, and the script ends with:

OUTPUT result.txt

```
PASS greenhouse-capture: decoded 01 03 02 00 F5 78 03 = 245 = 24.5 degC
```

IF IT DOESN'T WORK

- sigrok prints nothing: the VCD must first pass through `vcd-for-sigrok`, as `run.sh` does.
- No Modbus fields: the `modbus` decoder must be stacked on `uart` (`-P uart:…,modbus`), with `parity=even`.

TRY THIS NEXT

Change `TEMP_C` in `layers.py` to 18.7 and run the lab again. Which bytes and which CRC change?

1.7 The Protocol Passport BASIC

The five questions are the start of a method that this library uses for every protocol. Each protocol opens with a one-page card called the **Protocol Passport**. It has 16 fields, always in the same order, and each field answers part of one question (Figure 1.10). With the same card for every protocol, you can compare two protocols field by field and see at once what is different.

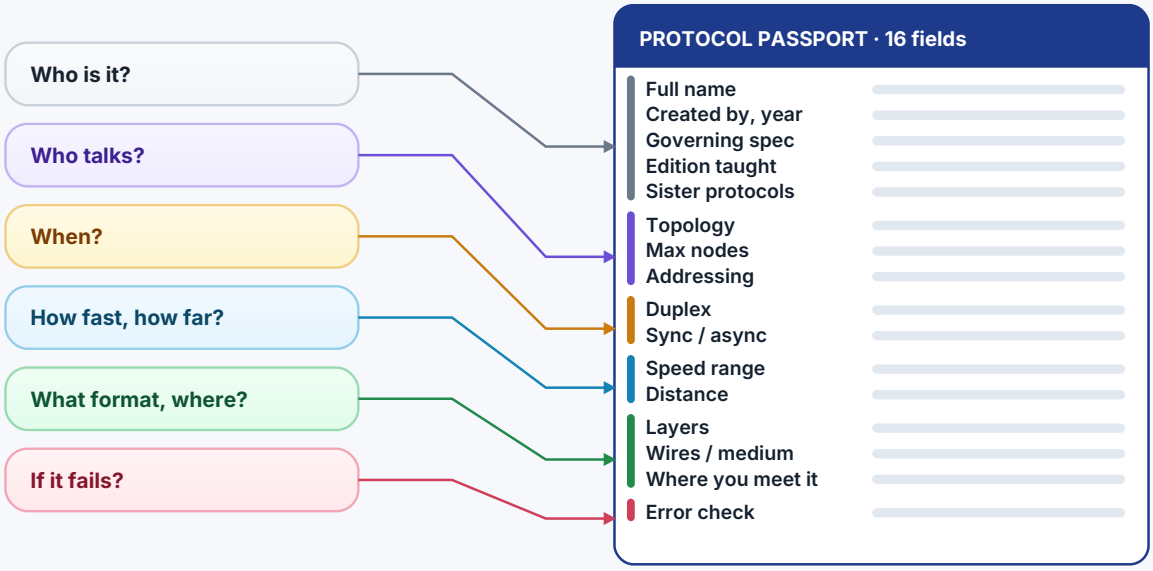


Figure 1.10 The 16 fields of the Protocol Passport, grouped by the question they answer. "Who is it?" adds the identity fields: the protocol's full name, who made it, the document that defines it and the edition the book teaches.

The passport is the first page of a longer, fixed plan called the **Protocol Dossier**: 22 parts, from why the protocol exists to a debugging playbook and a one-page cheat sheet. Chapter 4 shows a complete passport and dossier for the UART, and Chapter 3 shows where each of the 40 books uses it. At work, when two companies connect their equipment, engineers write the same answers into an **interface control document**: speeds, character format, addresses, register map and time-outs.

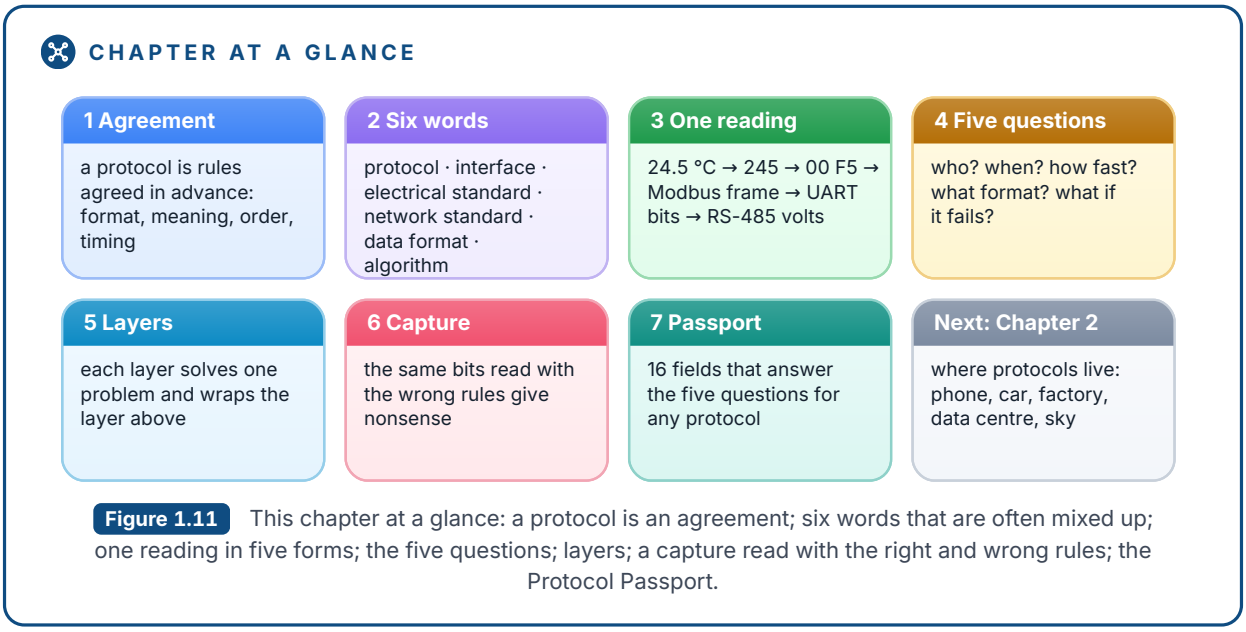


Figure 1.11 This chapter at a glance: a protocol is an agreement; six words that are often mixed up; one reading in five forms; the five questions; layers; a capture read with the right and wrong rules; the Protocol Passport.

REVIEW QUESTIONS

Recall

1. What is a protocol? Name its three kinds of rule, with one example of each from the greenhouse link.
2. In one sentence each, say what an interface, an electrical standard, a network standard, a data format and a security algorithm define, and give one example of each.
3. List the five questions every protocol answers, and name the passport fields that answer “Who talks?”.

Apply

4. Sort these names into the six words of this chapter: RS-232, JSON, I^2C, IEEE 802.3, SHA-256, a 3-pin header, Modbus RTU, a USB-C receptacle.
5. A second sensor reads 18.7 °C and also stores tenths of a degree. What register value and which two data bytes does it send? How long does its 7-byte answer take at 9600 bit/s with 11 bits per byte?
6. The farm moves the controller to Ethernet and reads the sensor with Modbus TCP through a gateway. Which layers of Figure 1.8 change on that new path, and which stay the same?
7. Answer the five questions for the GPS module link in your own words. How does it differ from the greenhouse link on “who?” and “when?”?

Debug or design

8. A technician says: “The new meter speaks RS-485, so it will work with our controller.” The readings are garbage. Name three things to check, using this chapter’s words.
9. In Lab 1.1, a receiver set to 9600 8N1 got the right bytes but five errors. Explain why the bytes were right and where the errors came from.

Answers: Appendix B.

Interview

1. What is the difference between a protocol and an interface? An interface is where two things connect: pins, a connector or a software API. A protocol is the set of rules for what is said through it: message format, meaning, order, timing and error handling. One interface, such as USB-C, can carry several protocols.

2. Is RS-485 a protocol? No. RS-485 (TIA-485-A) is an electrical standard: it defines the difference voltages on a twisted pair and the receiver threshold. Protocols such as Modbus RTU or BACnet MS/TP run on top of it, usually with UART characters.

3. Why are protocols built in layers? Each layer solves one problem and hides it from the others, so one layer can be changed without touching the rest. Modbus messages, for example, travel over RS-485 with RTU framing or over Ethernet with TCP, and the application layer stays the same.

4. What happens on a Modbus RTU link when a request arrives with a bad CRC? The server ignores it and sends nothing, because it cannot trust any field, not even the address. The client waits for its response time-out and then decides what to do, usually to send the request again.

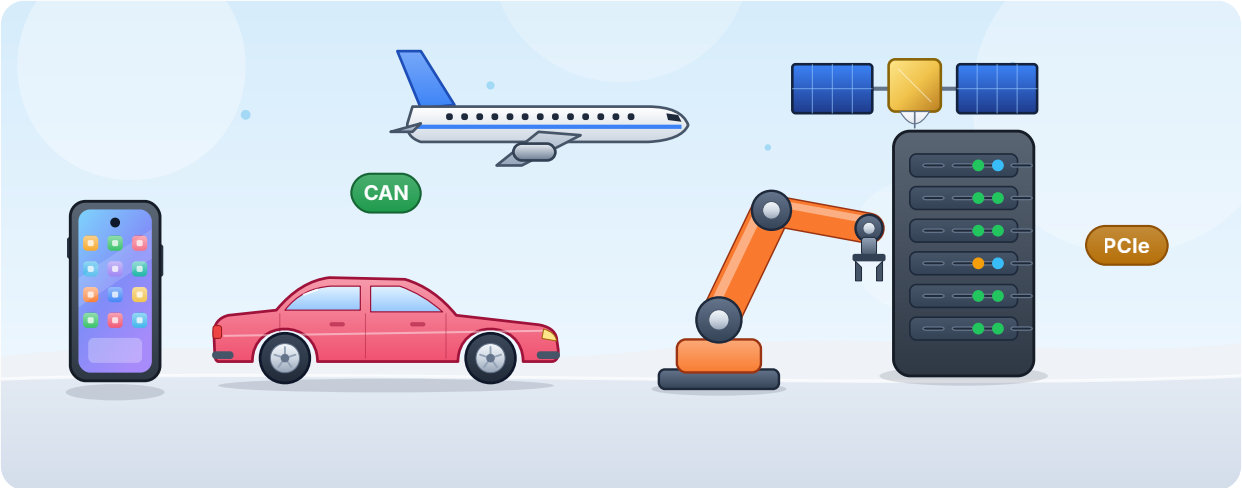
KEY TAKEAWAYS

- A protocol is a set of rules, agreed in advance, for the format, meaning, order and timing of messages and for what happens on errors.
- An interface is where devices connect; an electrical standard says how 0 and 1 look as voltages; a network standard is a document that defines a whole network family; a data format writes values as bytes; a security algorithm hides or signs data.
- RS-485 is an electrical standard, USB-C is an interface, Wi-Fi products follow the network standard IEEE 802.11, JSON is a data format and AES is a security algorithm; none of them is “a protocol” on its own. A CRC is an error-check algorithm: it catches noise, not attackers.
- In the greenhouse link, 24.5 °C becomes 245 (00 F5), a Modbus RTU frame with a CRC-16, 11-bit UART characters at 9600 bit/s, and voltage differences on an RS-485 pair.
- Every protocol answers five questions: who, when, how fast, what format, and what if it fails.
- A polled protocol such as Modbus RTU avoids collisions because a server speaks only when asked; a damaged request gets no answer, and the client times out and retries.
- Layers split the work; encapsulation wraps each layer’s data with the next layer’s header and checks.
- The Protocol Passport answers the five questions in 16 fixed fields for every protocol in this library.

CHAPTER 2

Where Protocols Live

WHERE WE ARE: Signal | Line coding | Framing | Error control | Access | Network | Transport | Application | Measure & debug



Chapter 1 followed one reading along one cable. Real products have dozens of links like that one, each with its own protocol. This chapter takes you on five illustrated tours: inside a phone, a car, a factory, a data centre, and an aircraft and a satellite. On every tour, each link is numbered and named, and the text says in a sentence or two what it carries and which book of this library teaches it. You do not need to understand every name yet. The goal is a map: after this chapter, when you meet a protocol, you will know where it lives.

WARM-UP

1. Is RS-485 a protocol, an interface or an electrical standard (Chapter 1)?
2. Name the five questions that every protocol answers.
3. Why does a Modbus RTU sensor stay silent when a request arrives with a bad CRC?

IN THIS CHAPTER

2.1 Name the main links inside a smartphone and the protocol on each, and the phone's radio links.

2.2 Name the buses of a modern car and say what each carries, including the diagnostics port.

2.3 Describe the levels of a factory network, from 4–20 mA sensors up to OPC UA and MQTT.

2.4 Name the protocols inside a server and between the servers of a data centre.

2.5 Name the data links of an airliner, a satellite and a drone.

2.6 Explain how distance, speed, cost, power and safety decide which protocol a link uses, and find the book that teaches it.

2.1 The idea in plain words BASIC

Think about how people talk at different distances. At a table, you just speak: it is fast, and nobody needs to say who they are. Across a building, you use an intercom with a button for each room. Across a city, you make a phone call through an exchange. Across the world, you send a message that is passed from network to network. Each distance has its own rules, because each has different problems.

Electronics works the same way. Inside a chip, blocks are millimetres apart and move huge amounts of data. On a board, chips are centimetres apart. In a car or a machine, devices are metres apart in an electrically noisy place. Across a farm, a factory or a city, signals must travel hundreds of metres or kilometres, often by radio, sometimes from a battery that must last for years. Across the world, data crosses many networks. Figure 2.1 sorts some well-known protocols by the distance they cover.

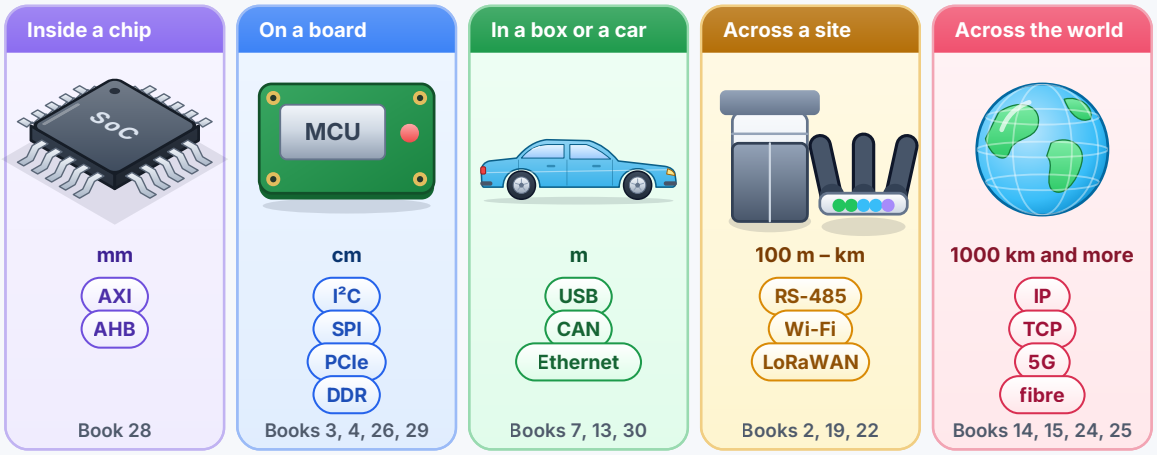


Figure 2.1 Protocols sorted by the distance they cover, from inside a chip to across the world, with the books that teach them. The longer the distance, the more a protocol has to deal with: noise, delay, sharing the medium, and finding the way through other networks.

Definition Link

A **link** is one connection that carries data between two or more devices, such as the wires between a processor and its memory, a cable between two units, or a radio channel. Each link uses its own protocol, or its own stack of protocols.

On the tour figures that follow, every link has a colour, a number and the name of its protocol. The numbered list under each figure says what the link carries, with the book that teaches it. The figures are simplified: real products have more chips and more links, and makers choose different parts.

KEY IDEA

The distance a link must cover, the speed it needs, its cost, its power budget and how safe it must be decide which protocols can live there. That is why one product uses many different protocols.

2.2 Inside a smartphone BASIC

A smartphone is built around one large chip, the **application processor** or **SoC** (system on a chip). It holds the processor cores and the graphics unit and, in many phones, the cellular **modem** and the satellite-navigation receiver. Everything else connects to it.

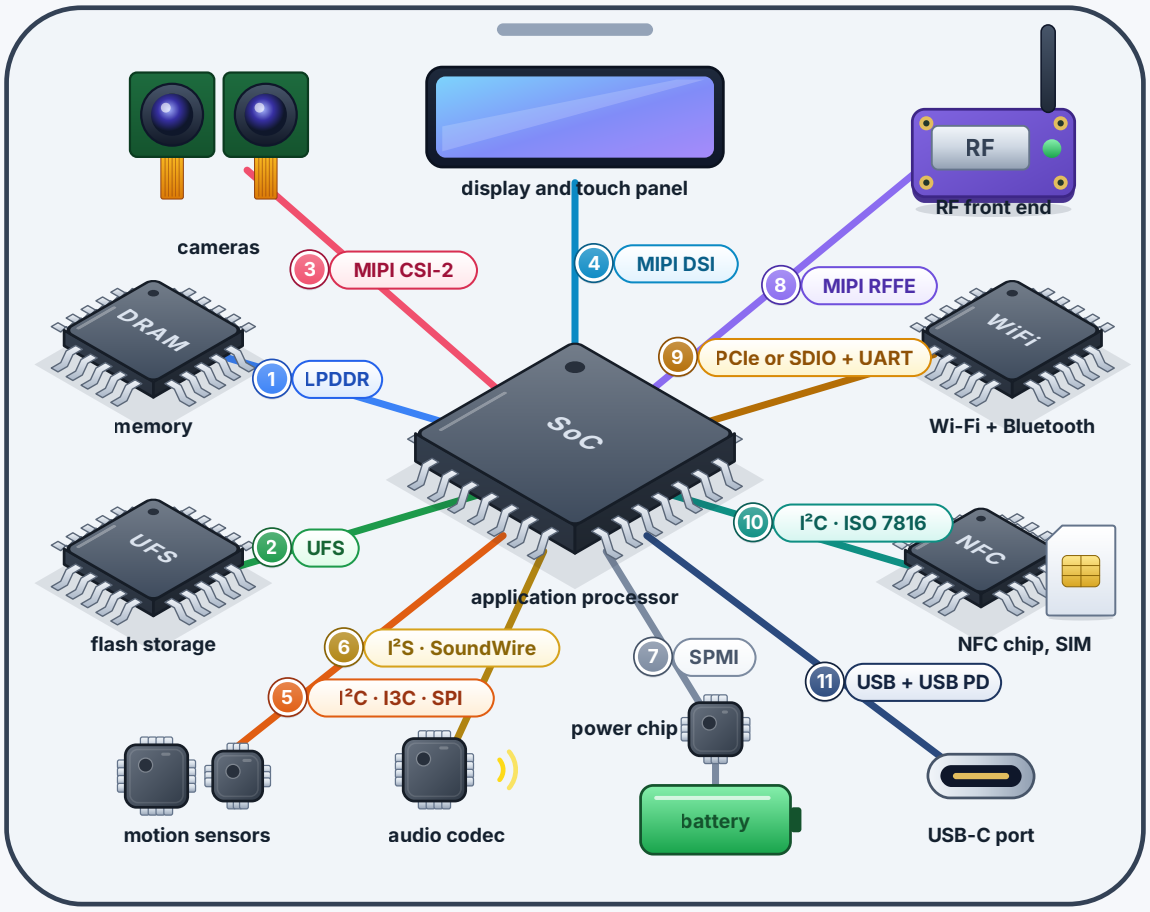


Figure 2.2 Inside a smartphone (simplified). The SoC sits in the middle; each numbered link to memory, storage, cameras, display, sensors, audio, power, radios and the USB-C port uses its own protocol.

1. **Memory: LPDDR.** The SoC keeps its working data in low-power DRAM. **LPDDR** (low-power double data rate) moves data on both edges of a fast clock over many parallel wires; the memory chips are often mounted on top of or right next to the SoC. Book 26.
2. **Storage: UFS.** Apps, photos and the operating system live in flash memory, reached over **UFS** (Universal Flash Storage) or, in many lower-cost phones, the older eMMC. Book 27.

3. **Cameras: MIPI CSI-2.** Each camera sends its pictures over **MIPI CSI-2**, a fast serial link on a few wire pairs. The SoC sets the camera up over a slower I²C-style control bus. Book 33 (the wire pairs, MIPI D-PHY and C-PHY, are in Book 32).
4. **Display: MIPI DSI.** The picture goes to the screen over **MIPI DSI**, on the same kinds of wire pairs. Book 32.
5. **Touch and sensors: I²C, I3C, SPI.** The touch controller and the motion sensors report over **I²C**, **SPI** or the newer **I3C**: slow, cheap buses with few wires. Books 3 and 4.
6. **Audio: I²S, TDM, SoundWire.** Sound travels as digital samples to the audio chips over **I²S** or **TDM**, or over **MIPI SoundWire** in some designs. Book 34.
7. **Power: SPMI.** The SoC tells its power-management chip which voltages to make over a control bus such as **MIPI SPMI**; a fuel-gauge chip often reports the battery's charge over I²C. Books 4 and 36.
8. **Radio front end: MIPI RFFE.** The modem controls the amplifiers, switches and filters near the antennas over **MIPI RFFE** (Book 4); the radio signals themselves are 4G and 5G (Books 23, 24).
9. **Wi-Fi and Bluetooth chip: PCIe or SDIO, and UART.** A combined Wi-Fi and Bluetooth chip usually moves Wi-Fi data over **PCIe** or **SDIO**, and Bluetooth over a **UART** that carries the Bluetooth **HCI** (host controller interface) messages. Books 29, 27 and 20.
10. **NFC and SIM: I²C and ISO/IEC 7816.** The NFC chip talks to the SoC over I²C or SPI. The SIM card, or the eSIM chip soldered to the board, speaks the smart-card protocol **ISO/IEC 7816**. Book 38.
11. **USB-C: USB and USB Power Delivery.** The port carries USB data (USB 2.0, or USB 3.x in some phones). On a separate wire, the phone and the charger exchange **USB Power Delivery** messages to agree on a higher charging voltage. Book 30.

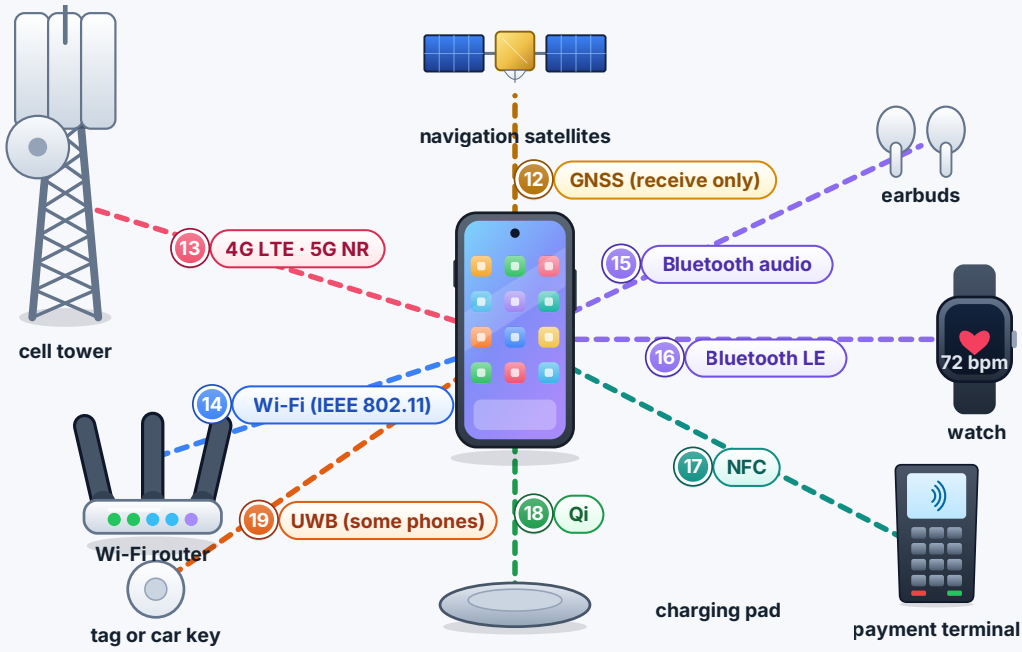


Figure 2.3 The phone's radio links. The numbers continue from the previous figure. Dashed lines are radio links; the phone only listens to the navigation satellites.

Almost all of the phone's links to the world around it are radio links (Figure 2.3):

12. **Navigation: GNSS.** The phone listens to navigation satellites, such as GPS, Galileo, GLONASS, BeiDou and India's NavIC, and works out its own position from their signals. It never transmits to them. Book 37.
13. **Mobile network: 4G LTE and 5G NR.** Calls and data go to the cell tower. Books 23 and 24.
14. **Wi-Fi.** At home and at work the phone joins a router using IEEE 802.11. Book 19.
15. **Bluetooth audio.** Music and calls go to earbuds over Bluetooth Classic or the newer LE Audio. Book 20.
16. **Bluetooth Low Energy.** A watch or a fitness band syncs over **BLE**, which saves battery by sleeping between short bursts. Book 20.
17. **NFC.** A tap on a payment terminal uses **NFC**, a radio link that works over a few centimetres. Book 38.
18. **Qi.** A charging pad and the phone exchange small messages inside the wireless power itself to agree on how much power to send. Book 36.
19. **UWB.** Some phones measure distance to a tag or a car key to within centimetres with **ultra-wideband** radio. Book 21.

Check yourself

1. Which two protocols carry the picture from a phone camera to its screen?
2. Which protocol does a phone use to agree a charging voltage with a USB-C charger?

2.3 Inside a car BASIC

A modern car has dozens of **ECUs** (electronic control units): small computers for the engine, the brakes, the airbags, the doors, the seats, the lights and the screens. They share data on several kinds of bus. In many cars a central **gateway** joins the buses and passes messages between them; newer designs group the ECUs by where they sit in the car, in **zones** joined by an Ethernet backbone.

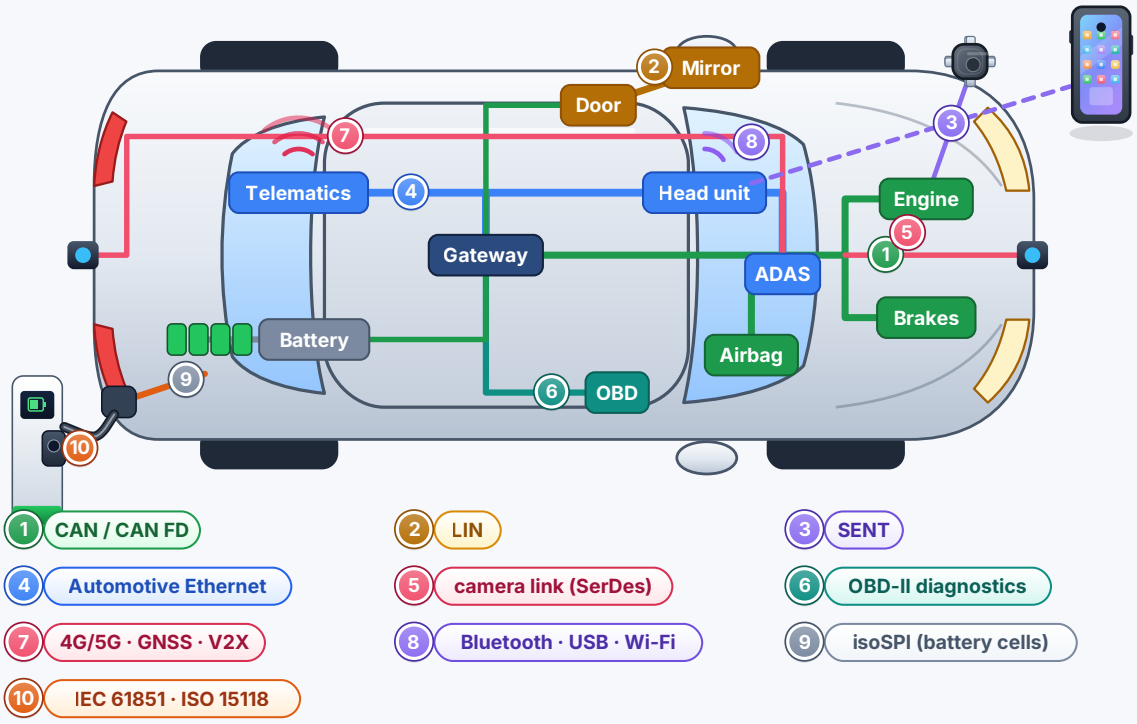


Figure 2.4 Inside a car, seen from above (simplified). Control units share CAN buses; small motors hang on LIN; cameras, the head unit and the driver-assistance computer use faster links; the gateway joins them all and serves the diagnostics port.

1. **CAN and CAN FD.** The workhorse bus of the car, on two twisted wires: engine, gearbox, brakes, airbags and body units send short messages many times a second, and any unit may send when the bus is free. **CAN FD** is its faster successor. Book 7.
2. **LIN.** A cheap, slow bus on one signal wire (at most 20 kbit/s) where one master unit controls simple devices: window lifters, mirrors, seat motors, wipers. Book 8.
3. **SENT.** Some sensors, such as pedal-position and pressure sensors, send their readings one way to their ECU over **SENT**, a simple one-way protocol on a single signal wire. Book 8.
4. **Automotive Ethernet.** The driver-assistance computer, the head unit and the telematics unit need much more data, so they use Ethernet on a single twisted pair (**100BASE-T1** and **1000BASE-T1**), carrying service messages such as **SOME/IP**. Book 9.
5. **Camera links.** Cameras send raw video to the driver-assistance computer over fast serial links, such as **GMSL** or **FPD-Link**. Book 33.
6. **OBD-II diagnostics.** A workshop's scan tool plugs into the diagnostics socket and talks to the ECUs, today over CAN using the **UDS** diagnostic services, and in newer cars also over Ethernet (**DoIP**). Book 9 (CAN in Book 7).

7. **Telematics: 4G/5G, GNSS, V2X.** A telematics unit with its own modem and GNSS receiver connects the car to the maker's servers; some cars also talk directly to other cars and to roadside units with **V2X** radio. Books 23, 24, 37 and 9.
8. **Phone: Bluetooth, USB, Wi-Fi.** The head unit takes calls and music from the driver's phone. Books 20, 30 and 19.
9. **Battery cells: isoSPI.** In an electric car, monitoring chips on the battery modules measure every cell and pass the readings along an isolated daisy chain, such as Analog Devices' **isoSPI**, to the battery-management unit, which reports on CAN. Book 36.
10. **Charging: IEC 61851 and ISO 15118.** At a charging station, a simple control signal (the IEC 61851 control pilot) says whether the plug is connected and how much current is allowed; **ISO 15118** adds digital messages, for example so the car can identify itself and pay automatically. Book 36.

Some cars also use **FlexRay**, a faster bus with fixed time slots, for chassis control; the BMW X5 of 2006 was the first production car to use it (Book 8).

REAL-WORLD EXAMPLE

The one link you can touch: the OBD-II socket

Every car and light truck sold in the United States since the 1996 model year must have an OBD-II diagnostics socket, a 16-pin connector under the dashboard, and since the 2008 model year it must support CAN diagnostics (ISO 15765-4) on pins 6 and 14. CAN itself reached production in 1991, in the Mercedes-Benz S-Class (W140). In 2013 BMW put Ethernet on a single twisted pair into series production, for the surround-view cameras of the X5. (US EPA and CARB information, CiA and BMW material, checked November 2026.)

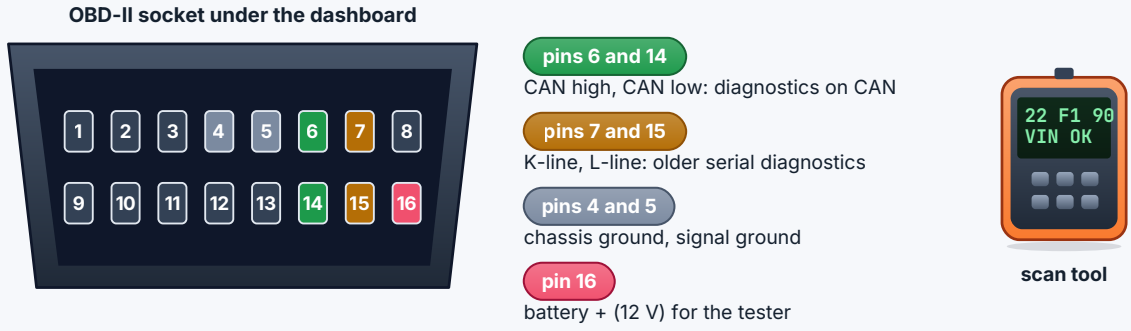


Figure 2.5 The OBD-II socket seen from the front. Pins 6 and 14 carry the CAN bus that a scan tool uses for diagnostics; pins 7 and 15 are older serial lines; pin 16 brings battery power to the tester.

Check yourself

3. Why do window and mirror motors hang on LIN instead of CAN?

2.4 Inside a factory BASIC

A factory must move the right signal at the right moment: a valve that opens 50 ms late can spoil a batch, and an emergency stop must never be missed. So factory networks are built in levels, with simple, robust links at the bottom and IT-style protocols at the top.

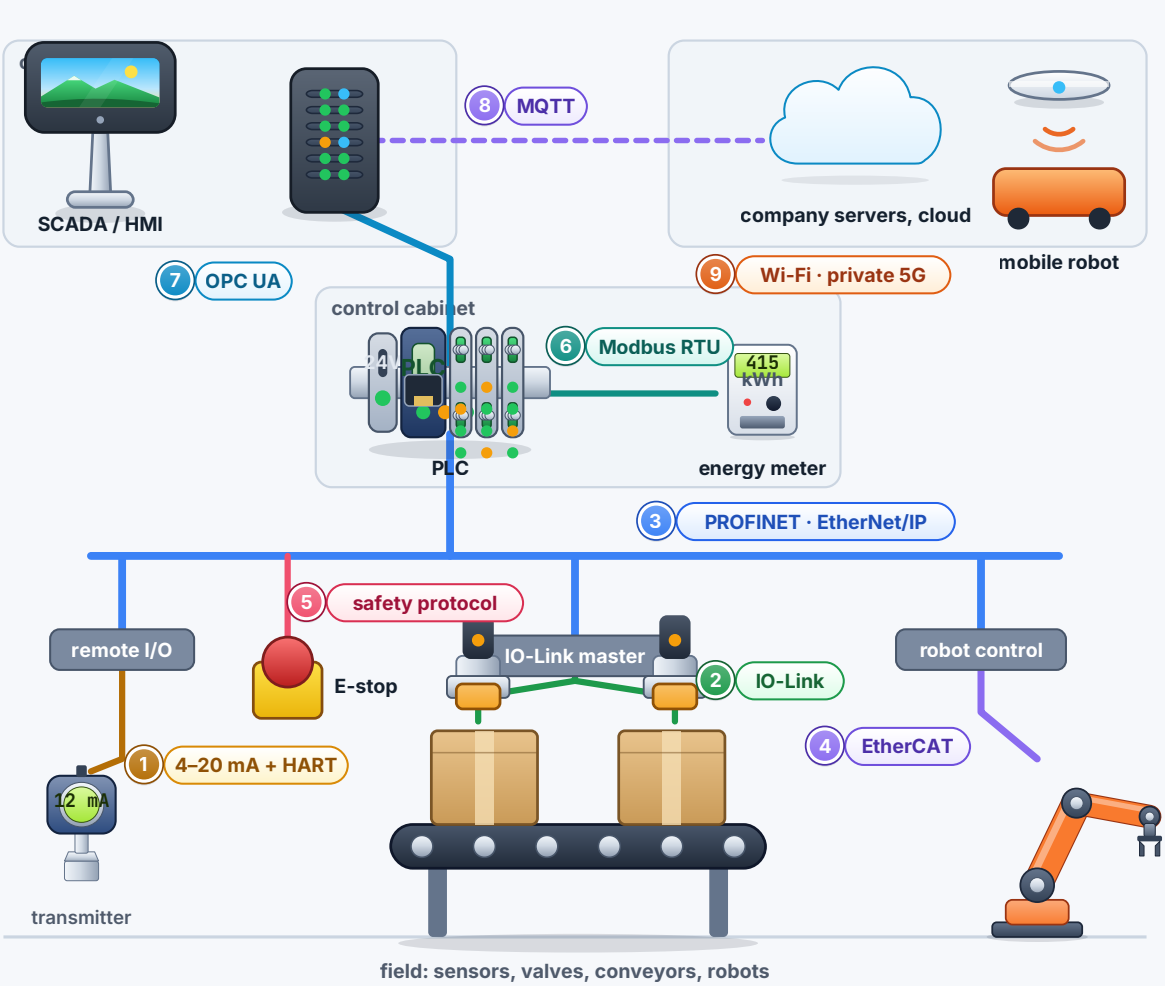


Figure 2.6 A small factory (simplified). Sensors and actuators at the bottom, a control cabinet with the PLC in the middle, and the control room and company servers at the top. One industrial Ethernet network joins the PLC to remote I/O, an IO-Link master and a robot.

1. **4–20 mA and HART.** Many process sensors (pressure, flow, level, temperature) still send their value as a current on two wires: 4 mA means the bottom of the range and 20 mA the top. The same current flows all round the loop, so a long cable does not change the reading, and 0 mA clearly means “wire broken”. **HART** adds a small digital signal on the same two wires, for setup and diagnostics. Book 10.
2. **IO-Link.** A point-to-point digital link between an **IO-Link master** and one smart sensor or actuator, on the usual three-wire sensor cable. Book 10.
3. **PROFINET and EtherNet/IP.** Industrial Ethernet protocols that carry inputs and outputs between the **PLC** (programmable logic controller), remote I/O, drives and IO-Link masters on a fixed cycle. Book 11.

4. **EtherCAT.** For fast motion control, such as robots and servo drives: one Ethernet frame passes through every device, and each one reads and writes its own part on the fly. Book 11.
5. **Safety protocols.** Emergency stops and light curtains send their signals over the same network with a safety protocol, such as **PROFIsafe**, **CIP Safety** or **FSoE**, which adds its own checks, counters and time-outs. Book 11.
6. **Modbus RTU.** Energy meters and many older devices still answer Modbus RTU on RS-485, as in Chapter 1; older plants also run fieldbuses such as **PROFIBUS DP**. Book 10.
7. **OPC UA.** The PLC shares its data with the SCADA and HMI screens in the control room, and with plant software, through **OPC UA**, which also describes what each value means. Book 11.
8. **MQTT.** Data for company servers or the cloud is often published with **MQTT**, a light publish-and-subscribe protocol. Book 17.
9. **Wi-Fi and private 5G.** Mobile robots and tablets use radio: Wi-Fi, or a factory's own private 5G network. Books 19 and 24.

Newer factory networks also add **TSN** (time-sensitive networking), which lets urgent control traffic and ordinary traffic share one Ethernet with guaranteed timing, and **PTP**, which keeps the clocks of all devices in step to well under a microsecond (Books 11 and 37).

Engineers often draw a factory as a pyramid of levels (Figure 2.7). Near the bottom, messages are small, frequent and must arrive on time. Near the top, messages are larger, rarer and less urgent, so the protocols there look like those of office IT.

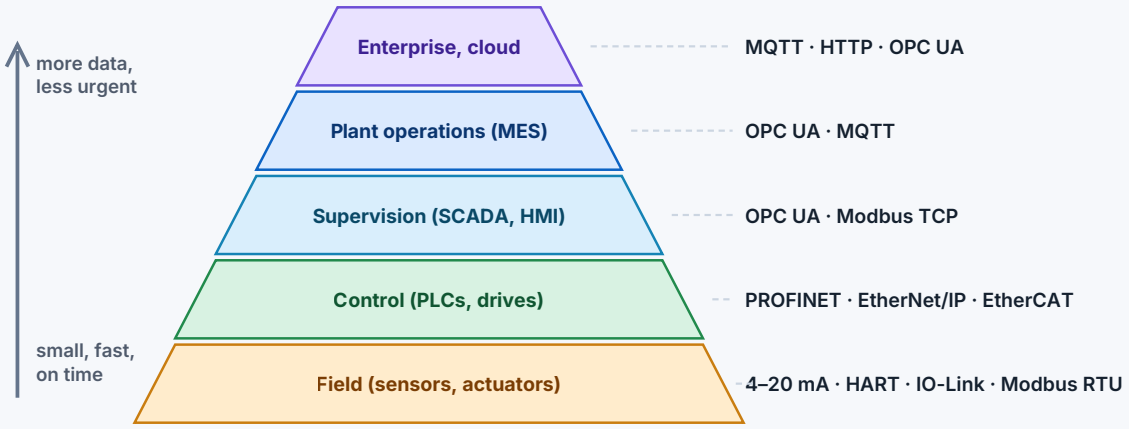


Figure 2.7 The levels of factory automation and the protocols typical of each. Messages grow larger and less time-critical from the field level at the bottom to the enterprise and cloud level at the top.

Check yourself

4. A temperature transmitter with a range of 0 °C to 100 °C sends 4 mA. What temperature does it report, and what would 0 mA mean?

2.5 Inside a data centre BASIC

A data centre is a building full of servers. Inside each server, the links are short and extremely fast; between servers, Ethernet and Internet protocols take over. Start with one server.

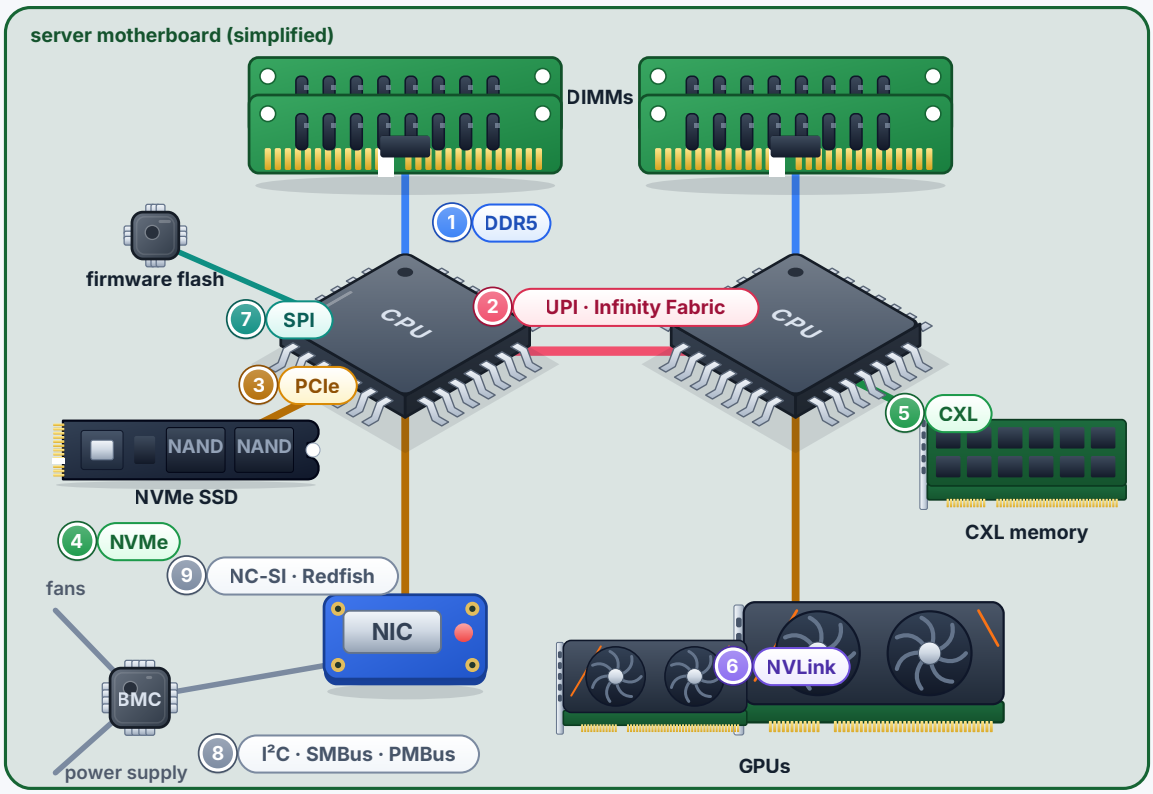


Figure 2.8 Inside one server (simplified). Two CPUs share memory and talk to each other; PCIe links reach the storage, the network card and the GPUs; a small management controller watches everything.

1. **DDR5.** Each CPU reads its main memory, the DIMMs, over **DDR5**. Book 26.
2. **UPI or Infinity Fabric.** The two CPU sockets share work and memory over a processor link, such as Intel's **UPI** or AMD's **Infinity Fabric**. Book 29.
3. **PCIe.** The main expansion link of every server: point-to-point lanes from the CPU to GPUs, network cards and SSDs. Book 29.
4. **NVMe.** The SSD's own command protocol, carried over PCIe. Book 27.
5. **CXL.** It runs on the same wires as PCIe and lets the CPU use extra memory on an add-in card. Book 29.
6. **NVLink.** GPUs exchange data with each other over a faster, dedicated link; inside each GPU package, stacked **HBM** memory sits next to the chip (Book 26). Book 29.
7. **SPI.** The firmware that starts the server is stored in a small flash chip on an SPI bus. Book 3.
8. **I²C, SMBus and PMBus.** The **BMC** (baseboard management controller), a small computer of its own, reads temperature sensors and fan speeds over I²C and SMBus, and controls the power supplies over **PMBus**. Book 4.

9. **NC-SI and Redfish.** The BMC can share the server's network card (**NC-SI**), and operators manage it remotely with **Redfish** or the older **IPMI**, even when the server is switched off. Book 29.

Now step back and look at the hall (Figure 2.9).

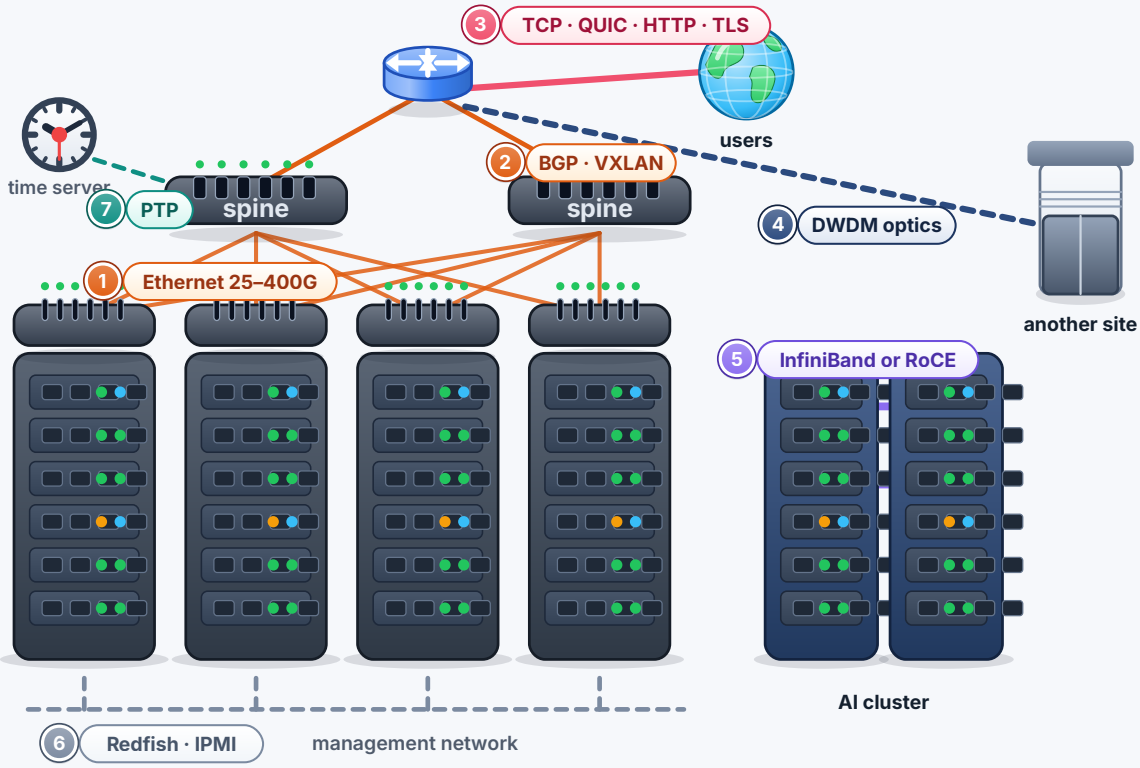


Figure 2.9 A data-centre hall (simplified). Racks of servers connect to top-of-rack switches, which connect to every spine switch; a router links the hall to users and to other sites. An AI cluster has its own fast network, and a separate management network reaches every server's BMC.

1. **Ethernet, 25 to 400 Gbit/s.** Each server connects to a switch at the top of its rack, and each of those switches connects to every **spine** switch, over copper cables or optical fibre. Books 13 and 25.
2. **BGP and VXLAN.** The switches route traffic with IP routing protocols such as **BGP**, and build virtual networks for different customers with **VXLAN**. Book 14.
3. **TCP, QUIC, HTTP and TLS.** Users reach the services over the Internet stack: TCP or QUIC for transport, HTTP for web requests, TLS for security. Books 15, 16 and 35.
4. **DWDM optics.** Links to other data centres send many light wavelengths down one fibre. Book 25.
5. **InfiniBand or RoCE.** AI clusters connect their GPU servers with very low-delay networks: **InfiniBand**, or **RoCE** (RDMA over Ethernet), which let one server write into another's memory directly. Book 29.
6. **Redfish and IPMI.** A separate management network reaches every BMC. Book 29.
7. **PTP.** A time server keeps the clocks of the switches and servers in step. Book 37.

Check yourself

5. Which link does a CPU use to reach an NVMe SSD, and which protocol does the SSD itself speak?

2.6 In the sky and in orbit BASIC

Aircraft and spacecraft need links that never fail, can be certified, and keep working for decades. Many of their protocols are old, simple and very well tested.

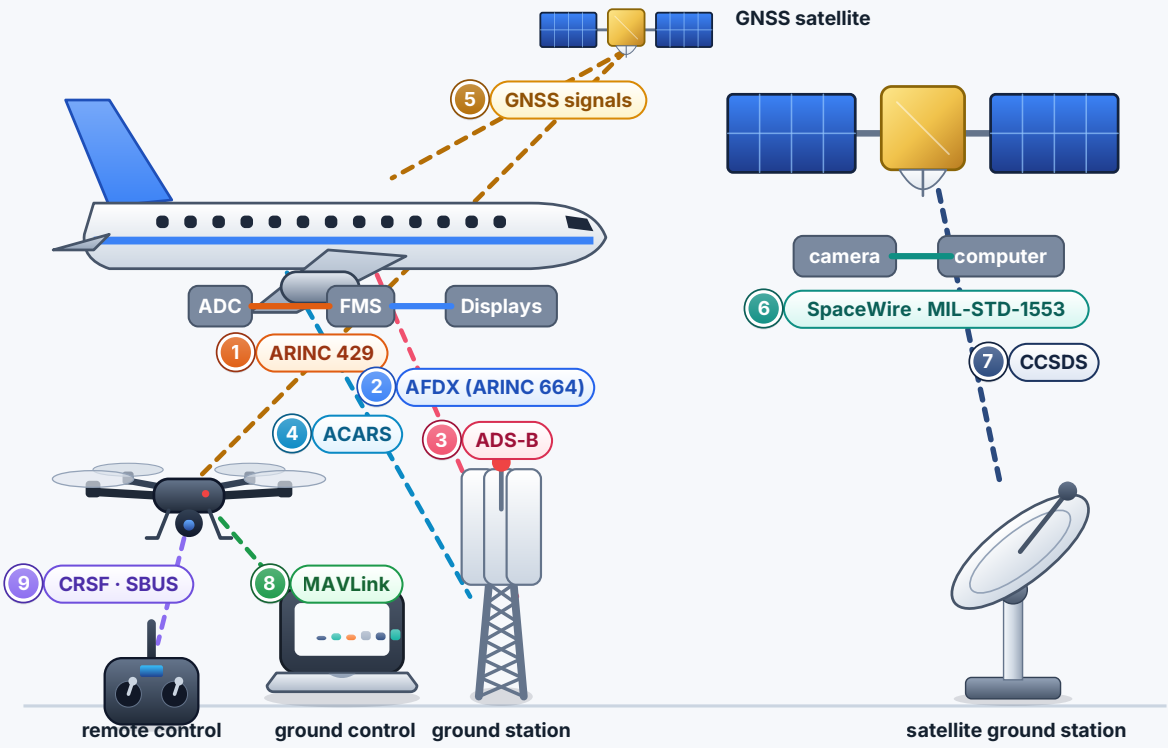


Figure 2.10 An airliner, a satellite and a drone, with their ground stations. Solid lines are wired links inside a vehicle; dashed lines are radio links.

1. **ARINC 429.** In many airliners, sensor units such as the air data computer send their values to the flight management system and the displays over **ARINC 429**: one transmitter, several receivers, on a twisted pair. Book 39.
2. **AFDX (ARINC 664).** Newer airliners, such as the Airbus A380 and A350 and the Boeing 787, add a switched Ethernet backbone with guaranteed timing, defined in ARINC 664 Part 7 (**AFDX** is Airbus's name for it). Book 39.
3. **ADS-B.** The aircraft keeps broadcasting its position, height and speed on 1090 MHz. Ground stations and other aircraft receive it. Book 39.

4. **ACARS.** Short text messages between the aircraft and its airline, such as flight plans, weather and engine reports, travel over **ACARS** by VHF radio, HF radio or satellite. Book 39.
5. **GNSS.** Aircraft and drones, like phones, listen to navigation satellites. Book 37.
6. **SpaceWire and MIL-STD-1553.** Inside a spacecraft, instruments send data to the on-board computer over **SpaceWire**, as on the James Webb Space Telescope, or over **MIL-STD-1553**, a bus that began in military aircraft (first used across a whole aircraft in the F-16) and is used on many spacecraft. Small satellites (CubeSats) often use I^2C and CAN (Books 4 and 7). Book 39.
7. **CCSDS.** Space agencies send telemetry down and commands up with the **CCSDS** space-data standards. Book 39.
8. **MAVLink.** A drone's flight controller and the ground-control software exchange telemetry and commands over **MAVLink**, on a telemetry radio or a network. Book 39.
9. **CRSF and SBUS.** The pilot's sticks reach the flight controller through the radio-control receiver, which passes them on over a serial protocol such as **CRSF** or **SBUS**; the flight controller then drives each motor's speed controller, for example with **DShot**. Book 39.

Check yourself

6. Which link lets air-traffic services and other aircraft see where an airliner is?

2.7 Why each place picks its protocols BASIC

The tours showed very different protocols, but the reasons behind them repeat.

What matters most	Where you saw it	Typical choice
Very high speed over centimetres	memory, GPUs, SSDs, cameras	wide or very fast links: LPDDR, DDR5, PCIe, MIPI CSI-2
Low cost for many slow devices	sensors, motors, power chips	simple buses with few wires: I^2C, SPI, LIN, IO-Link
Noise, long cables, many nodes	cars, factories	robust differential buses: CAN, RS-485, industrial Ethernet
On-time delivery and safety	brakes, robots, aircraft	fixed schedules and extra checks: EtherCAT, TSN, AFDX, safety protocols
Battery life	watches, tags, field sensors	radios that sleep most of the time: BLE, UWB, LoRaWAN
Reaching anyone, anywhere	phones, data centres	the Internet stack: IP, TCP, QUIC, HTTP, TLS
Decades of service	factories, aircraft, spacecraft	well-tested older protocols: 4–20 mA, Modbus, ARINC 429, MIL-STD-1553

A few protocols turned up on almost every tour (Figure 2.11). I^2C hides inside phones, servers and satellites; Ethernet runs in cars, factories, data centres and aircraft; CAN moved

from cars into small satellites. So a protocol you learn for one job is very likely to be useful in another.

Protocol	Book	Phone	Car	Factory	Data centre	Sky, space
I²C · I3C	4	●	·	·	●	●
SPI	3	●	·	·	●	·
UART	2	●	·	●	·	·
CAN	7	·	●	·	·	●
Ethernet	13	·	●	●	●	●
PCIe	29	●	·	·	●	·
USB	30	●	●	·	·	·
Bluetooth	20	●	●	·	·	·
Wi-Fi	19	●	●	●	·	·
4G / 5G	23, 24	●	●	●	·	·
GNSS	37	●	●	·	·	●

Figure 2.11 Where this chapter's tours met some well-known protocols, with the book that teaches each one. A big dot means the protocol appeared on that tour.

KEY IDEA

A few protocols live almost everywhere: I²C, SPI, UART, CAN, Ethernet, PCIe, USB and the common radios. Learn them first, and each new place you work in will already look familiar.

Chapter 3 turns these tours into the full map of the library: the 13 shelves, the 40 books, and a study path for each kind of engineering job.

CHAPTER AT A GLANCE

1 Distance
chip, board, box, site, world: each distance has its own protocols

2 Phone
MIPI links, LPDDR, UFS, I²C/SPI inside; 5G, Wi-Fi, Bluetooth, NFC outside

3 Car
CAN, LIN and SENT for control; Ethernet backbone; OBD-II for diagnostics

4 Factory
4–20 mA, IO-Link, PROFINET, EtherCAT at the bottom; OPC UA, MQTT on top

5 Data centre
DDR5, PCIe, NVMe, CXL in a server; Ethernet, BGP, RDMA between servers

6 Sky and space
ARINC 429, AFDX, ADS-B; SpaceWire, CCSDS; MAVLink on drones

7 Everywhere
I²C, Ethernet, CAN, USB and radios appear in many places

Next: Chapter 3
the library map: 13 shelves, 40 books and study paths

Figure 2.12 This chapter at a glance: protocols sorted by distance, and the main links of a phone, a car, a factory, a data centre, and an aircraft and a satellite.

REVIEW QUESTIONS

Recall

1. Name four links inside a smartphone, the protocol on each, and what it carries.
2. Which bus usually joins a door module to its window and mirror motors, and which one joins the engine unit to the brake unit?
3. Which link does an airliner use to broadcast its position, and which one carries instrument data inside the James Webb Space Telescope?

Apply

4. Trace a photo from a phone camera to the phone's screen, naming the protocol on each link, and say how the SoC sets the camera up.
5. A mechanic plugs a scan tool into the OBD-II socket of a 2015 car sold in the United States. Which pins carry the diagnostic data, on which bus, and which book teaches the diagnostic messages?
6. A level transmitter has a range of 0 m to 10 m and sends 12 mA. What level does it report, and which protocol could carry its diagnostics on the same two wires?
7. A web request arrives at a server's network card and needs a file from its NVMe SSD. Name the protocols the request and the data pass through inside the server.

Debug or design

8. A drone answers its pilot's sticks, but the ground-control software shows no telemetry. Which two links are involved, which protocol does each use, and which one would you check?
9. Give two reasons why car makers use LIN for window motors instead of putting every device on CAN or Ethernet.
10. You must read a battery-powered weather station 3 km from a farmhouse, once every ten minutes. Using Figure 2.1, which kind of protocol fits, and why is Wi-Fi a poor choice?

Answers: Appendix B.

Interview

1. Which protocols connect a phone's application processor to its camera and display? MIPI CSI-2 carries pictures from the camera, and MIPI DSI carries the picture to the display, both on MIPI D-PHY or C-PHY wire pairs. The camera's settings go over a slower I²C-style control bus.

2. What is CAN used for in a car, and how does it differ from LIN? CAN joins the main control units (engine, brakes, airbags, body) on a two-wire differential bus where any unit may send. LIN is a cheaper, slower bus on one signal wire, where one master controls simple devices such as window and mirror motors.

3. How does a server's CPU talk to an NVMe SSD? Over PCIe lanes. The SSD's commands, queues and data transfers follow the NVMe protocol, which runs on top of PCIe.

4. What is the difference between 4–20 mA and HART? 4–20 mA sends one value as a current on two wires, with 4 mA as the bottom of the range and 20 mA as the top. HART adds a small digital signal on the same wires, so the device can also be set up and report diagnostics.

KEY TAKEAWAYS

- Every product is a set of links, and each link uses a protocol chosen for its distance, speed, cost, power budget and safety needs.
- Inside a phone: LPDDR, UFS, MIPI CSI-2 and DSI, I²C/I3C/SPI, I²S, SPMI, RFFE, PCIe or SDIO with UART, ISO/IEC 7816 and USB with USB PD; its radios: GNSS, 4G/5G, Wi-Fi, Bluetooth, NFC, Qi, UWB.
- In a car: CAN and CAN FD for control units, LIN for simple motors, SENT for sensors, automotive Ethernet for cameras, screens and the backbone, and OBD-II diagnostics on CAN pins 6 and 14.
- In a factory: 4–20 mA and HART, IO-Link, PROFINET or EtherNet/IP, EtherCAT, safety protocols, Modbus, OPC UA and MQTT, from the field level up to the cloud.
- Inside a server: DDR5, UPI or Infinity Fabric, PCIe with NVMe and CXL, NVLink, SPI flash, and I²C, SMBus and PMBus for the BMC; between servers: Ethernet, BGP, VXLAN and RDMA networks.
- In the sky: ARINC 429 and AFDX in airliners, ADS-B and ACARS to the ground, SpaceWire, MIL-STD-1553 and CCSDS in space, MAVLink and CRSF or SBUS on drones.
- A few protocols, such as I²C, CAN, Ethernet, PCIe and USB, appear in many places: learn them first.

CHAPTER 3

The Library Map

WHERE WE ARE: Signal | Line coding | Framing | Error control | Access | Network | Transport | Application | Measure & debug



The first two chapters showed what a protocol is and where protocols live, from a phone to a satellite. You have now met dozens of protocol names, and there are hundreds more. This chapter is the map of the library that teaches them: thirteen shelves, this free guide and forty books, the rule that gives every protocol one home, and the method that every book uses. It ends with study paths for eight careers and the order in which the books appear, so you can plan your own route.

WARM-UP

1. What is the difference between a protocol and an interface (Chapter 1)?
2. A sensor sends readings as Modbus frames over a UART at RS-485 levels. Which of the three is the electrical standard (Chapter 1)?
3. Name three protocols you would find inside a phone (Chapter 2).

IN THIS CHAPTER

3.1 Find the shelf and the book that teach a protocol, using the library wall and the home-book rule.

3.2 Explain what the depth tiers A, B, C and D give a protocol, and what each means for you as a reader.

3.3 Say where in the Protocol Dossier to look for a frame format, a timing number, a fix or a summary.

3.4 Describe how each book proves its promise: outcomes, desk labs with real output, question universes, blind tests and a final test.

3.5 Choose a study path for a career and plan the order in which to read the books.

3.1 The idea in plain words BASIC

Think of a large library. Its books stand on shelves, and each shelf holds one subject. Every book has exactly one place, so you never search the whole building. And inside every book of a series the chapters follow the same plan, so once you know one book you can find your way in all of them.

Suppose you want to know how an I²C target's address is sent. You go to the shelf for chips on a board, take down the I²C book, and open the part that every book uses for frame formats (Figure 3.1). Three steps, and no guessing.



Figure 3.1 Finding one fact in the library takes three steps: the shelf for the subject, the one book that teaches the protocol, and the part of the dossier that holds the answer (here P6, the frame format).

This chapter gives you the three things that make those steps work: the shelves, the home-book rule and the dossier.

3.2 Thirteen shelves, forty books BASIC

The Protocol Library is this free guide and forty paid books, Book 1 to Book 40, on thirteen **shelves**. A shelf is a group of books about one kind of system. Figure 3.2 shows the whole wall: each colour is one shelf, and the same colours mark the shelves in every figure of this guide.

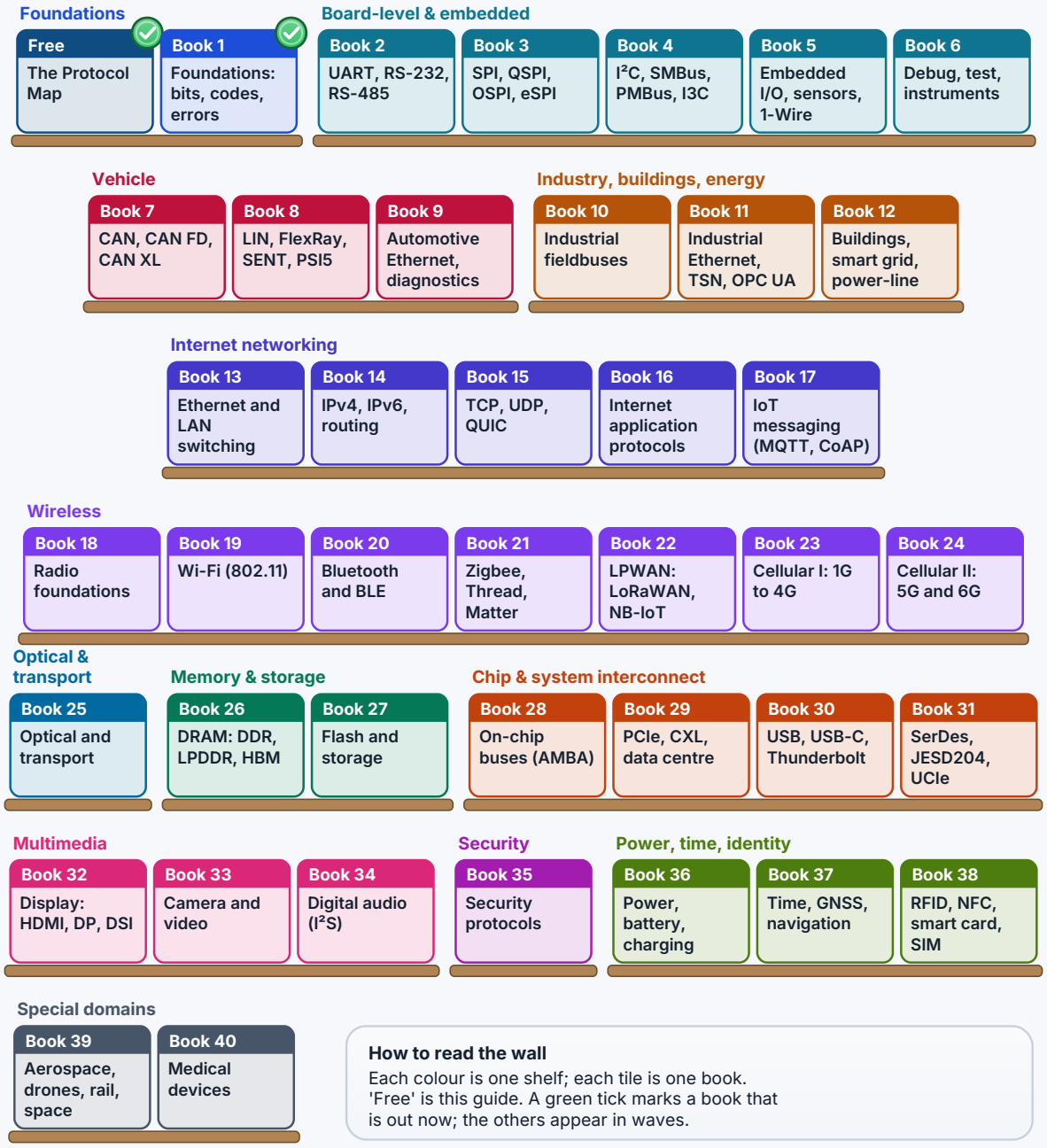


Figure 3.2 The library wall: this free guide and the forty books on their thirteen shelves. Each colour is one shelf; a green tick marks a book that is already out.

Read the wall from the wire outwards. Foundations come first. Then come the links between chips on one board; the networks inside vehicles, factories and buildings; the networks between machines, wired, wireless and optical; the links inside a computer and a chip; and finally media, security, power, time and identity, and special fields such as aircraft and medicine. The table gives the full titles.

Shelf	Books	What you meet there
Foundations	Start Here: The Protocol Map (free) · 1 Digital Communication Foundations	what protocols are; bits, clocks, line codes, signal integrity, CRCs, framing, access, tools

Shelf	Books	What you meet there
Board-level & embedded	2 UART, RS-232, RS-422 and RS-485 · 3 SPI and Its Family · 4 I²C, SMBus, PMBus and I3C · 5 Embedded I/O, Sensor and Single-Wire Protocols · 6 Debug, Programming, Test Access and Instrument Control	links between chips on a board, and to the lab bench: UART, SPI, I²C, I3C, 1-Wire, JTAG, SWD, SCPI
Vehicle	7 CAN, CAN FD, CAN XL and the CAN Protocol Family · 8 LIN, FlexRay, SENT, PSI5 and Classic Vehicle Buses · 9 Automotive Ethernet, SOME/IP, V2X and Vehicle Diagnostics	the networks inside cars, trucks and tractors, and how they are diagnosed
Industry, buildings, energy	10 Industrial Fieldbuses · 11 Industrial Ethernet, TSN and OPC UA · 12 Building Automation, Smart Grid and Power-Line Communication	4–20 mA, Modbus, PROFIBUS, IO-Link, EtherCAT, PROFINET, BACnet, KNX, IEC 61850
Internet networking	13 Ethernet and LAN Switching · 14 IPv4, IPv6 and Routing Protocols · 15 TCP, UDP, QUIC and Transport Protocols · 16 Internet Application Protocols · 17 IoT Messaging	from the Ethernet frame to HTTP, DNS and MQTT
Wireless	18 Radio Foundations · 19 Wi-Fi · 20 Bluetooth Classic and Bluetooth Low Energy · 21 Zigbee, Thread, Matter and Low-Power Wireless · 22 LPWAN · 23 Cellular I: 1G to 4G LTE · 24 Cellular II: 5G NR, 5G Core, O-RAN and 6G	radio basics, then every common radio protocol up to 5G
Optical & transport	25 Optical and Transport Networks	SONET/SDH, OTN, WDM, PON, fronthaul
Memory & storage	26 DRAM Interfaces · 27 Flash and Storage Interfaces	DDR, LPDDR, HBM; NAND, SD, eMMC, UFS, SATA, NVMe
Chip & system interconnect	28 On-Chip Buses · 29 PCI Express, CXL and Data-Centre Interconnects · 30 USB, USB-C, USB Power Delivery and Thunderbolt · 31 SerDes, JESD204 and Chiplet Interconnects	AXI, PCIe, CXL, USB, UCIe
Multimedia	32 Display Interfaces · 33 Camera, Broadcast and Professional Video · 34 Digital Audio Interfaces	HDMI, DisplayPort, MIPI DSI and CSI-2, SDI, I²S
Security	35 Communication Security Protocols	TLS, SSH, IPsec, WireGuard, WPA3, MACsec
Power, time, identity	36 Power, Battery and Charging Communication · 37 Time, GNSS and Navigation Protocols · 38 RFID, NFC, Smart-Card and SIM Protocols	EV charging, PTP, GNSS, NFC, SIM cards
Special domains	39 Aerospace, Drones, Rail and Space Data Links · 40 Medical Device Communication	ARINC, MIL-STD-1553, MAVLink, CCSDS, HL7, FHIR

Check yourself

1. Which shelf and which book would you open to learn MQTT? And LoRaWAN?
2. Why does the Wireless shelf start with a book that teaches no single protocol?

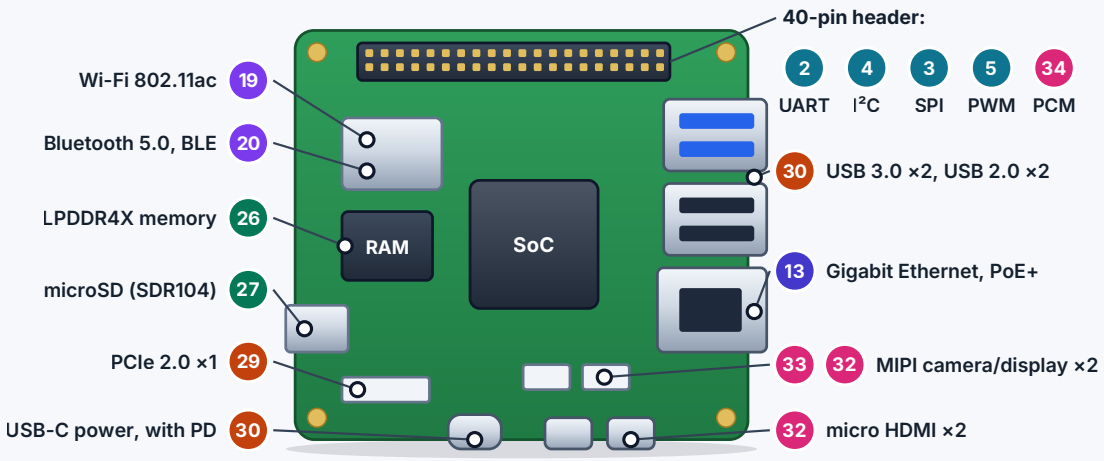


Figure 3.3 A single-board computer of the Raspberry Pi 5 kind, drawn simply and not to scale. Each interface carries the number of the book that teaches it: one board reaches fourteen books on six shelves.

REAL-WORLD EXAMPLE

One board, fourteen books

A single-board computer such as the Raspberry Pi 5 shows how widely one product reaches across the shelves (Figure 3.3 above). Its specification lists LPDDR4X memory (Book 26), a microSD slot with SDR104 mode (Book 27), two USB 3.0 and two USB 2.0 ports and USB-C power with USB Power Delivery (Book 30), Gigabit Ethernet with PoE+ through an add-on board (Book 13), dual-band 802.11ac Wi-Fi (Book 19), Bluetooth 5.0 and Bluetooth Low Energy (Book 20), two micro HDMI outputs (Book 32), two 4-lane MIPI camera/display connectors (Books 33 and 32) and a PCIe 2.0 ×1 interface (Book 29). Its 40-pin header carries UART, I²C, SPI, PWM and PCM signals at 3.3 V (Books 2, 4, 3, 5 and 34). (Raspberry Pi product page and documentation, checked November 2026.)

3.3 One home for every protocol BASIC

Many protocols are used in several fields. TLS protects web pages, but also IoT messages and industrial servers. If every book taught TLS, you would read it five times, a little differently each time. So the library follows one rule.

Definition **Home book**

A protocol's **home book** is the one book that teaches it in full, through every part of the dossier. Any other book that needs it uses it and points to the home book with a cross-reference.

Figure 3.4 shows four examples from the plan of the series. Book 16 teaches HTTPS but sends you to Book 35 for TLS itself. Book 36 explains chargers but leaves USB Power Delivery to its home, Book 30. Book 9 uses TSN in vehicles, which Book 11 teaches. Book 4 shows DDC/EDID running over I²C, but its home is the display book, Book 32.

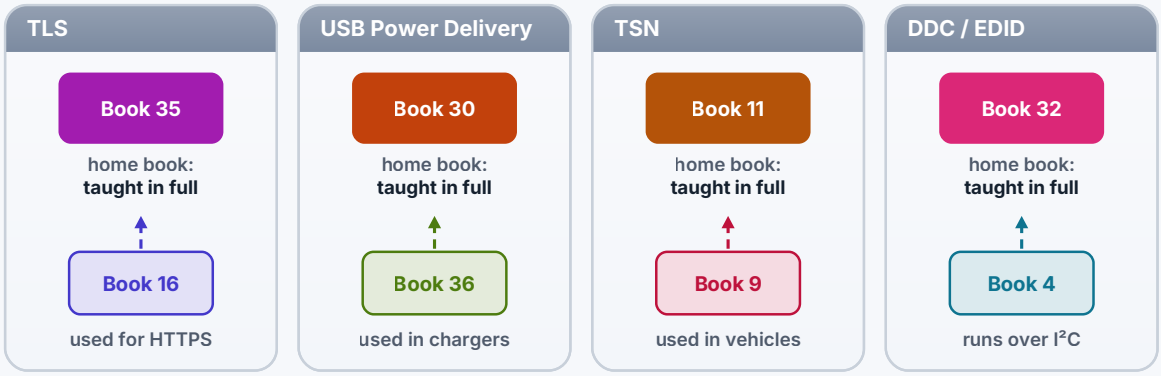


Figure 3.4 One home per protocol. Each protocol is taught in full in its home book (solid tile); a book that only uses it points there (dashed arrow) instead of teaching it again.

Behind the books sits a **register**: a list of every protocol the series covers, with its home book, its depth tier and the specification that defines it. The register also catches names that are not protocols at all. CSMA/CD is an access method, RINEX is a file format, and ISO 13485 is a quality-management standard. Each is still covered, as what it really is, in a short note with a pointer.

> KEY IDEA
>
> **Every protocol has exactly one home book that teaches it in full. Other books use it and point home, so you learn each protocol once, in one place, in one style.**

3.4 Depth tiers: how much each protocol gets BASIC

A protocol that engineers use every day needs more pages than one you meet once in a career. The library gives each protocol a **depth tier**, set in the register before its book is written. I²C, CAN, PCIe, TCP, AXI and DDR are examples of Tier A.

Tier	Size	Dossier parts	Labs	Printed questions
A: core, used daily	several chapters, 40–100 pages	all 22	2–4	100–200
B: important in one field	one chapter, 15–35 pages	all 22, shorter	1	40–80
C: specialist or legacy	a section, 5–12 pages	the core parts	optional	15–25
D: not a protocol	a note, 1–2 pages	none	none	none

What does a tier mean for you (Figure 3.5)? After a **Tier A** protocol you can design with it, write its driver or RTL, capture it and debug it. After a **Tier B** protocol you can do the same at a smaller scale, in one chapter. After a **Tier C** section you can explain the protocol, recognise and decode it, and you know where its full details live; legacy protocols such as PS/2, LPC and Token Ring are here. A **Tier D** note tells you what the item really is and where it is used.

Each protocol book opens with a map of its protocols and their tiers, so you can see the depth before you start.

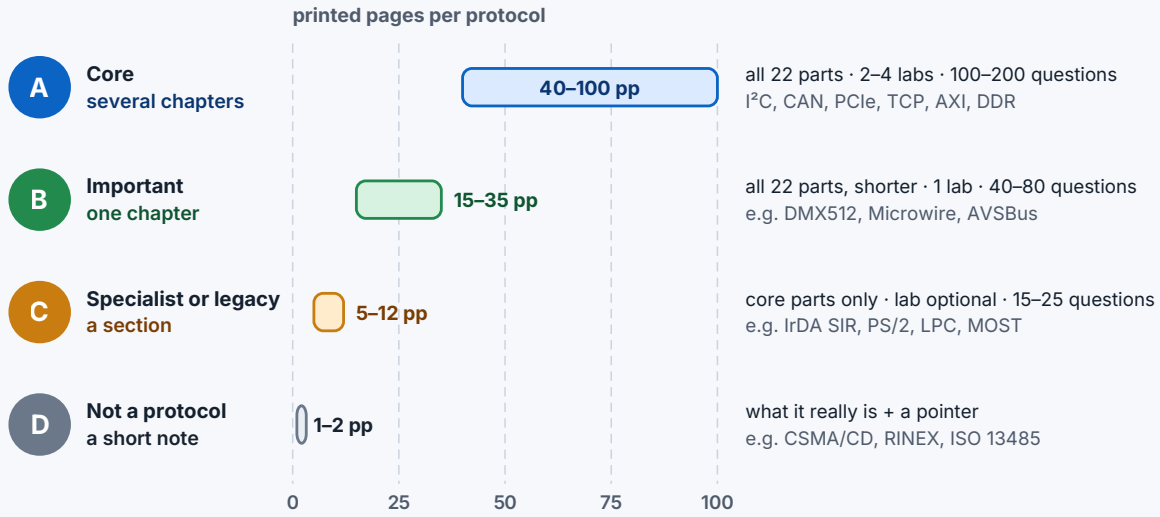


Figure 3.5 The four depth tiers. A Tier A protocol gets several chapters, a Tier B protocol one full chapter, a Tier C protocol a compact section, and a Tier D item, which is not really a protocol, a short note with a pointer.

Check yourself

1. Which tier is a protocol taught in one full chapter?
2. Why is CSMA/CD Tier D, even though it appears in many exam syllabuses?

3.5 How every book teaches BASIC

Every protocol, in every book, is taught to the same plan, called the **Protocol Dossier**. It opens with a one-page identity card, the **Protocol Passport** (P0), and then has 22 parts, P1 to P22, always in the same order (Figure 3.6). The fixed order has two benefits. You always know where to look: the frame format is P6, the timing maths P9, the errors P10, the debugging playbook P16 and the cheat sheet P22. And a missing part shows at once. A Tier C section keeps only the core parts, marked with a dot.

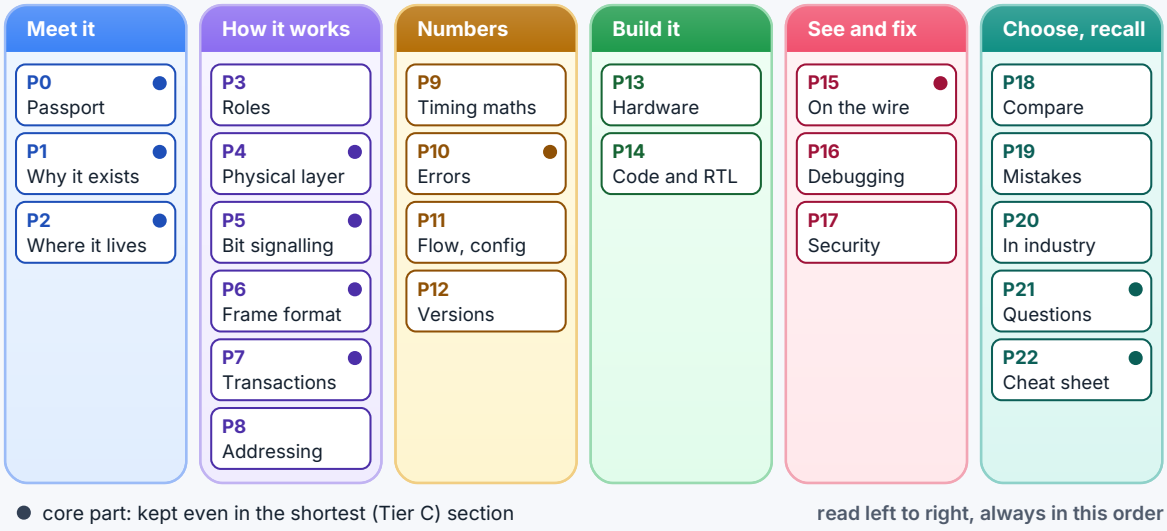


Figure 3.6 The 22 parts of the Protocol Dossier, plus the Passport (P0), grouped by what they do. The order never changes; a dot marks the core parts that even a Tier C section keeps.

Chapter 4 of this guide teaches one complete dossier, for the UART, so you can see every part at work before you buy a book.

Every chapter is also built the same way (Figure 3.7). It opens with a picture, a "Where we are" strip that shows which layers of the communication stack it covers, three warm-up questions and its learning outcomes. Sections carry a level, **Basic**, **Intermediate** or **Advanced**; Advanced sections and Deep dive boxes can wait for a second reading. Check-yourself questions, key ideas and worked examples follow each idea. A desk lab, a one-picture summary ("at a glance"), exercises, interview questions and a summary close the chapter. The answers, a final test, revision cards and a glossary are at the back of the book.

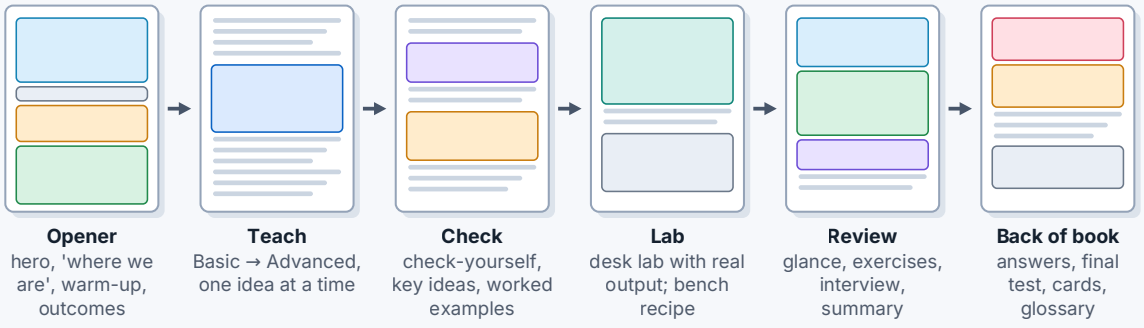


Figure 3.7 How every chapter is built: an opener, teaching sections with checks and worked examples, a lab, and a review. The answers and the final test are at the back of the book, never on the same page as the questions.

KEY IDEA

Every protocol is taught in the same 22 parts, in the same order. Learn the order once and you can find any answer in any book.

3.6 How the books keep their promise BASIC

The series promises that after a protocol's chapters you can answer any question that a working engineer, an interviewer, an exam or a design review would ask about it. Four things back that promise.

- **Learning outcomes.** Each chapter lists what you will be able to do, as numbered outcomes. Each outcome points to the exercises that practise it, and the book's final test has at least one question for every outcome.
- **Desk labs with real output.** A **desk lab** runs on an ordinary computer with free tools: simulations, decoders and network stacks. The author has run every one, and every output printed in a book is copied from a real run, never typed by hand. A **bench lab** is a recipe for your own hardware; it is marked "not run by the author" and describes what you should see.
- **A question universe and a blind test.** Before a protocol is written, its **question universe** is collected: at least 200 questions for Tier A, 80 for Tier B and 30 for Tier C, from concepts to numbers, debugging and interviews. The chapter is written to answer all of them. Then a fresh reviewer who sees only the finished chapter answers a held-out quarter of the questions (Figure 3.8). Tier A and B chapters must score at least 95 %; every miss becomes new text, and the chapter is tested again.
- **Sources for every number.** Each limit and rule is traced to the specification, datasheet or RFC that defines it, with its section number, and facts about products are checked and dated.

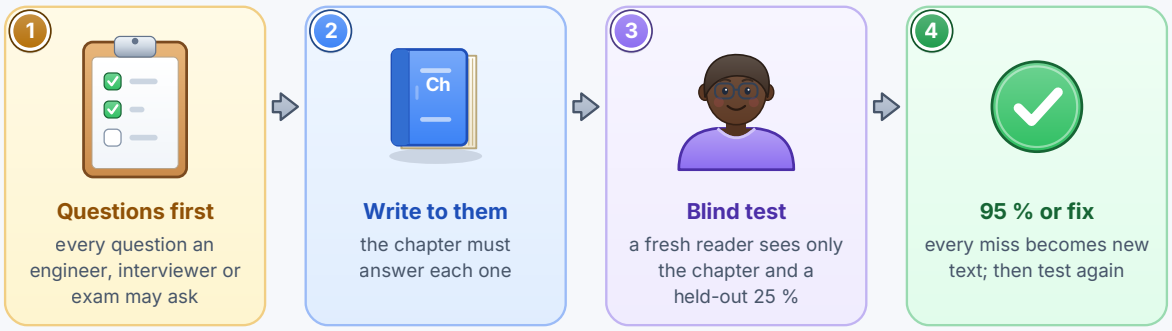


Figure 3.8 The blind test. Questions come first; the chapter is written to answer them; a reader who sees only the chapter answers a held-out sample; anything below 95 % sends the chapter back for more text.

Check yourself

1. What is the difference between a desk lab and a bench lab?
2. Who answers the questions in a blind test, and what may that reviewer read?

3.7 Book 1, the foundation BASIC

Every protocol is built from the same small set of ideas: a way to put bits on a wire, a way to keep two clocks in step, a code that makes the signal easy to receive, a check that catches errors, a frame that marks where a message starts, and a rule for who may talk when. **Book 1, Digital Communication Foundations**, teaches those ideas once, completely, so that every later book can build on them. It is out now (Figure 3.9).

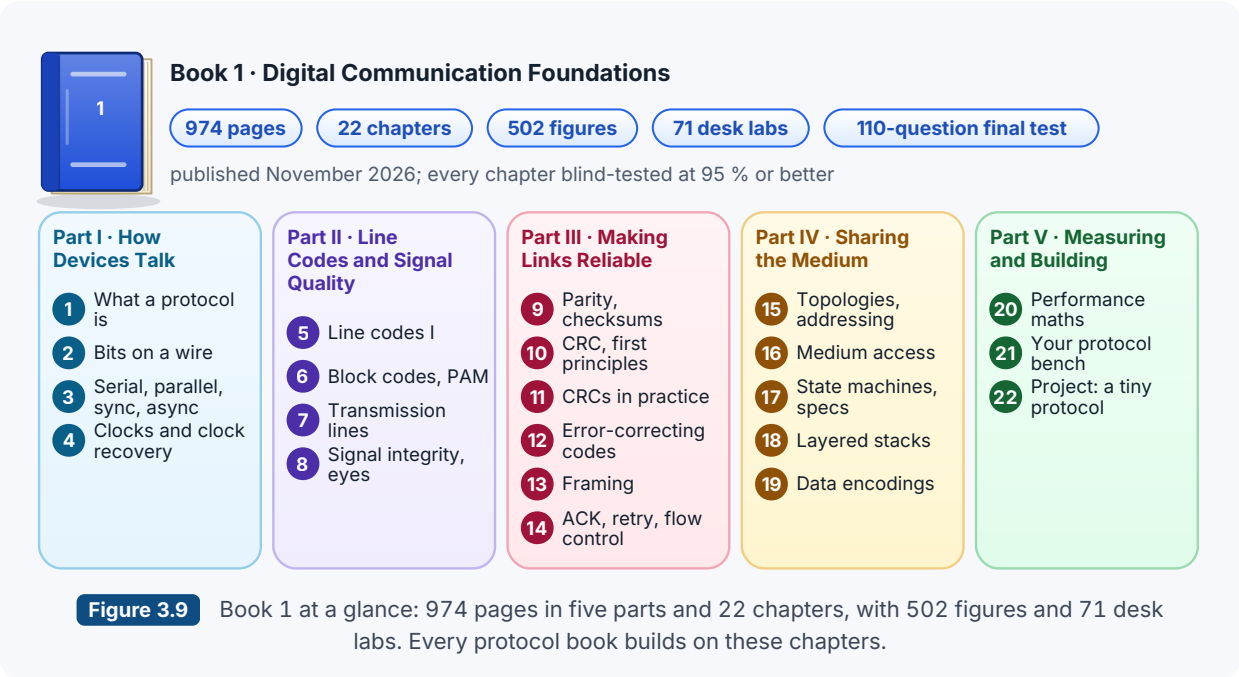


Figure 3.9 Book 1 at a glance: 974 pages in five parts and 22 chapters, with 502 figures and 71 desk labs. Every protocol book builds on these chapters.

Book 1 has 974 pages, 22 chapters in five parts, 502 figures and 71 desk labs. You compute CRCs by hand and build a CRC circuit in Verilog, watch an eye diagram close as a cable gets longer, decode real captures with a logic analyzer and Wireshark, and finish by designing and debugging a small protocol of your own. It ends with a final test of 110 questions (99 or more means you are ready for the protocol books), the worked answers, revision cards and a glossary of about 960 terms. Each protocol book opens with a short readiness check that points back to the Book 1 chapters it needs.

Common mistake **Skipping the foundations**

Wrong: "I only need PCIe, so I will start with Book 29."

Why: PCIe uses block codes such as 128b/130b, CRCs, credit-based flow control and a link state machine. Book 29 builds on these ideas; it does not teach them from zero.

Right: Read the Book 1 chapters that the book's readiness check names (here line codes, CRCs, flow control and state machines), then start Book 29.

3.8 Study paths by career BASIC

You do not need all forty books. Figure 3.10 gives a route for each of eight careers. Read each route from left to right; the colour of a stop is its shelf.



Figure 3.10 Study paths for eight careers. Every path starts with Book 1; the stops are in reading order, and each colour is a shelf of the library wall.

Career	Books in order
Embedded firmware	1 → 2 → 3 → 4 → 5 → 6 → 30 → 7 → 20 → 21
Automotive electronics	1 → 2 → 4 → 7 → 8 → 9 → 13 → 35
Industrial automation	1 → 2 → 10 → 13 → 11 → 12 → 17
Networking & telecom	1 → 13 → 14 → 15 → 16 → 35 → 25 → 23 → 24
Wireless & IoT	1 → 18 → 20 → 19 → 21 → 22 → 17
VLSI / SoC design & verification	1 → 3 → 4 → 6 → 28 → 29 → 30 → 26 → 31 → 32
Board / hardware design	1 → 2 → 3 → 4 → 5 → 13 → 30 → 26 → 32
Power electronics & EV	1 → 4 → 7 → 36 → 12

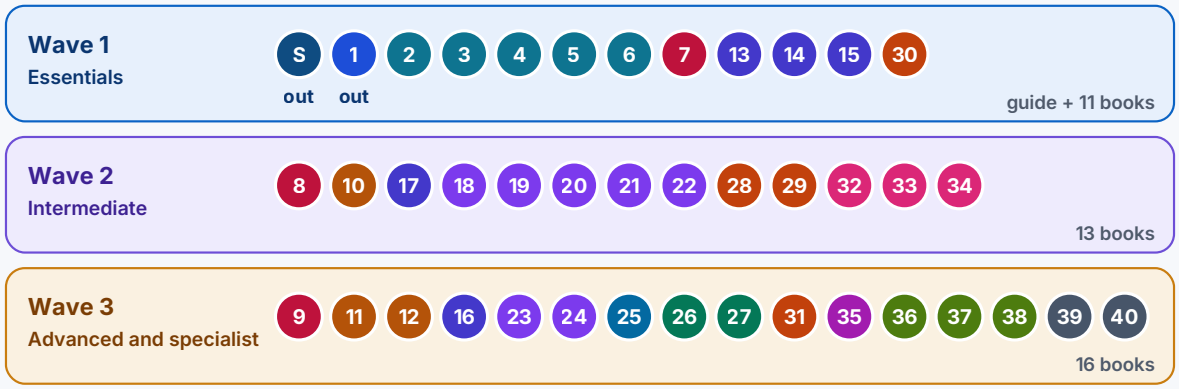
The paths are a starting point, not a cage. A firmware engineer who moves into cars adds Books 8 and 9; a verification engineer who works on networking chips adds Book 13. Because each protocol has one home book, mixing paths never makes you read the same material twice.

Check yourself

1. Which book is on every path, and why?
2. Which path would you give a student who wants to design PCIe and DDR controllers in RTL?

3.9 When the books appear BASIC

The books are written one at a time, in three **waves** (Figure 3.11). Wave 1 holds the essentials that most engineers need first: this guide, Book 1, the board-level books 2 to 6, CAN (Book 7), the Internet core (Books 13, 14 and 15) and USB (Book 30). Wave 2 adds intermediate subjects such as the radios, IoT messaging, on-chip buses, PCIe and the media books. Wave 3 holds advanced and specialist subjects, from 5G to aircraft data links. The guide and Book 1 are out now; the order of the others may still change, and no dates are promised.



Books appear one at a time, roughly in this order; S = this Start Here guide.

Figure 3.11 The order in which the books appear: three waves, essentials first. Each disc is a book, coloured by its shelf; the guide (S) and Book 1 are already out.

3.10 How to use the series BASIC

1. **Finish this guide.** Chapter 4 shows a complete dossier, Chapter 5 shows how to read the specifications behind every number, and Chapter 6 sets up your free bench.
2. **Read Book 1**, at least its Basic and Intermediate sections, and take its final test.
3. **Pick a study path** and read its books in order. Use the library wall to find a protocol's home book when a path leaves it out.
4. **Read actively.** Answer the warm-up from memory, stop at every check-yourself box, run the desk lab and compare your output with the book's, and do the exercises before you look at the answers.

5. **Finish each book with its final test.** A score of 90 % or more means you are ready for the next book; each missed question names the section to re-read.
6. **Keep the summaries.** The cheat sheet (P22) of each protocol and the revision cards at the back are made for quick review before an exam, an interview or a design review.

CHAPTER AT A GLANCE

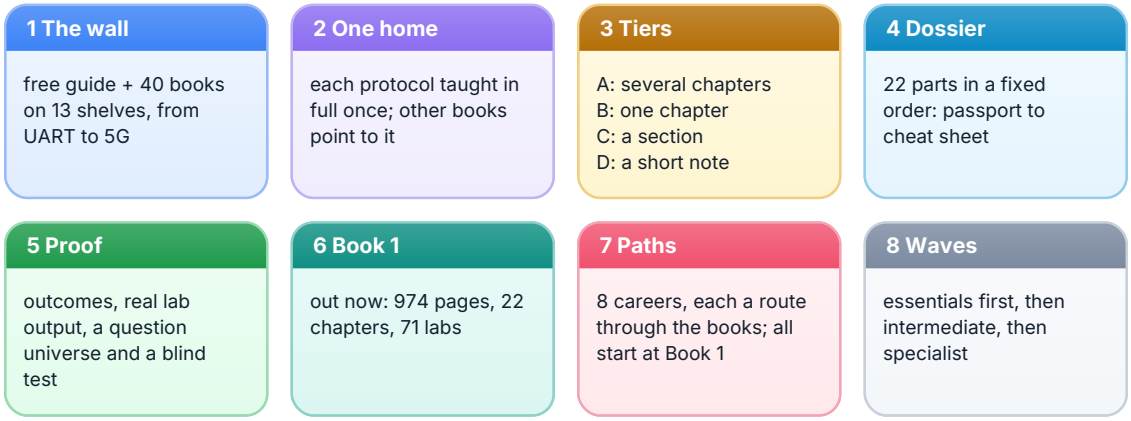


Figure 3.12 This chapter at a glance: the wall of books, one home per protocol, the tiers, the dossier, the proof behind each book, Book 1, the study paths and the waves.

REVIEW QUESTIONS

Recall

1. How many shelves and paid books does the library have? Name the shelf and the book for SPI, MQTT, HDMI and LoRaWAN.
2. For each tier, A to D, give the size, the dossier parts and the number of printed questions.
3. Which dossier parts hold the frame format, the timing maths, the debugging playbook, the versions and the cheat sheet?

Apply

4. Book 9 uses TSN in vehicles. Where is TSN taught in full, and what does Book 9 do with it? Give one more example of the same rule.
5. You meet CSMA/CD and RINEX in a list of “protocols”. What does the library do with each, and why?
6. A protocol gets a Tier C section. Name six dossier parts that the section must still contain.

Debug or design

7. A new chapter scores 88 % in its blind test. What happens next, and why is the reviewer not allowed to read anything except the chapter?

8. Plan the reading order for a new automotive engineer, and say which wave each book belongs to.
9. A reader asks how they can trust a lab output printed in a book. What do you tell them, and how is a bench lab different?
10. An embedded engineer wants to add USB and Bluetooth to their skills. Which path already holds both, and which of the two books appears first?

Answers: Appendix B.

Interview

1. Why teach every protocol to the same 22-part plan? The fixed order makes every book predictable: the frame format is always P6 and the debugging playbook P16, so a reader finds any answer quickly, and an author or reviewer sees at once when a part is missing.

2. How would you go from zero to designing an I²C sensor board? Learn the foundations first (logic levels, open-drain signalling, timing and state machines in Book 1), then the full I²C dossier in Book 4, with GPIO, level shifting and sensor interfaces from Book 5. That is the start of the board / hardware design path.

3. Is CSMA/CD a protocol? No. It is a medium-access method: a rule by which stations share one medium (listen before sending, detect collisions, back off). Classic Ethernet used it. The library teaches it as an access method, not as a protocol of its own.

KEY TAKEAWAYS

- The Protocol Library is this free guide and forty books, Book 1 to Book 40, on thirteen shelves, from Foundations to Special domains.
- Each protocol has one home book that teaches it in full; other books use it and point there.
- Depth tiers: A gets several chapters, B one chapter, C a section with the core parts, and D (not a protocol) a short note.
- Every protocol follows the 22-part Protocol Dossier in a fixed order, opened by the Passport (P0).
- Every chapter has outcomes, checks, worked examples, a desk lab with real output, exercises, answers at the back and a final test for the book.
- A question universe is written first, and Tier A and B chapters must pass a blind test at 95 % or better.
- Book 1, Digital Communication Foundations (974 pages, 22 chapters, 71 labs), is out now and is the first stop on every study path.
- Eight study paths give a reading order per career; the books appear in three waves, essentials first.

PART II

The Method

How every book teaches a protocol, one complete example, how to read a specification, and your own bench.

4 UART in One Chapter: A Complete Sample Dossier

5 Reading Specifications and Datasheets

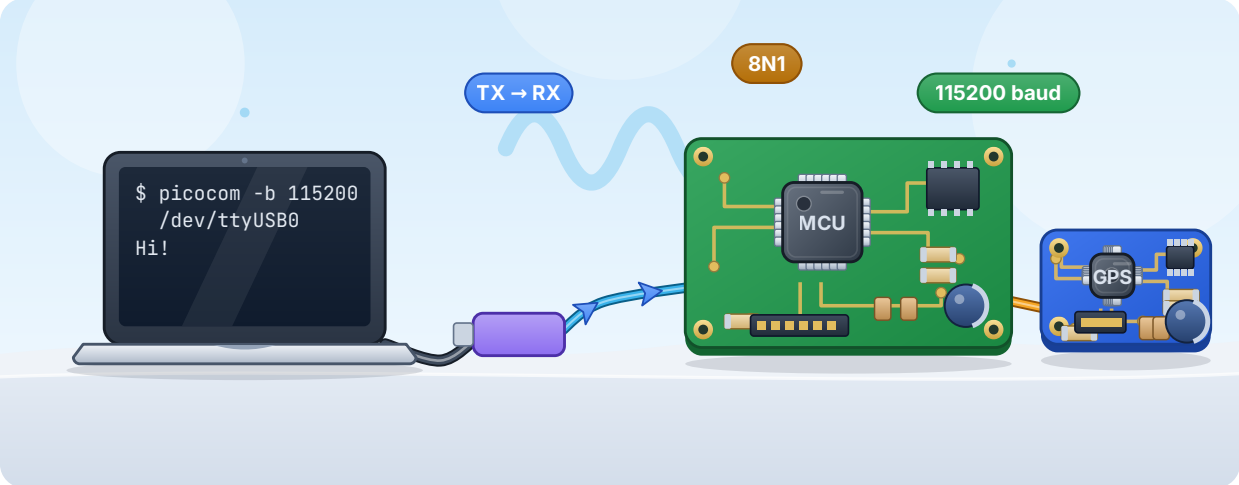
6 Your Protocol Bench

CHAPTER 4

UART in One Chapter: A Complete Sample Dossier

4

WHERE WE ARE Signal | Line coding | Framing | Error control | Access | Network | Transport | Application | Measure & debug



You now know what a protocol is, where protocols live, and how this library is organised. In this chapter you see the method at work on one real protocol: the **UART**. Almost every electronic board has at least one, so it is the protocol engineers meet first. It is also simple enough to teach completely in one chapter, part by part, in the same order that every book in the series uses. Book 2 returns to the UART in full depth and adds RS-232, RS-422 and RS-485.

WARM-UP

1. What is the difference between a protocol and an electrical standard (Chapter 1)?
2. Name two places inside a product where a small module talks to a main processor (Chapter 2).
3. What does a Tier B protocol get in this library: a section, a chapter or several chapters (Chapter 3)?

IN THIS CHAPTER

4.1 Fill in the Protocol Passport of a UART link and explain each entry.

4.2 Draw a UART frame (8N1, 7E1, 8O2) bit by bit and explain why it starts low and ends high.

4.3 Compute bit time, frame time, throughput efficiency and the largest baud-rate mismatch a receiver can tolerate.

4.4 Explain how a receiver finds the start bit and samples each bit (16× oversampling, mid-bit sampling).

4.5 Recognise parity, framing, overrun and break conditions and say what causes each.

4.6 Wire two UART devices correctly (TX to RX, common ground, matching logic levels) and spot the classic wiring mistakes.

4.7 Read a decoded UART capture and use the debugging playbook to find a fault.

4.1 How a protocol dossier works BASIC

Every protocol in this library is taught with the same checklist, called the **Protocol Dossier**. It has 22 parts, P1 to P22, and it opens with a one-page identity card, the Protocol Passport (P0). The order never changes. So when you want the frame format of any protocol, you look for part P6; when you want to debug it, you look for P16.

A big protocol (Tier A) spreads the dossier over several chapters; a smaller one (Tier B) fits it into one. The UART is small, so this chapter covers almost every part, and four parts get only one line. Each section starts with a small note that names its part, for example *Dossier part P6*.

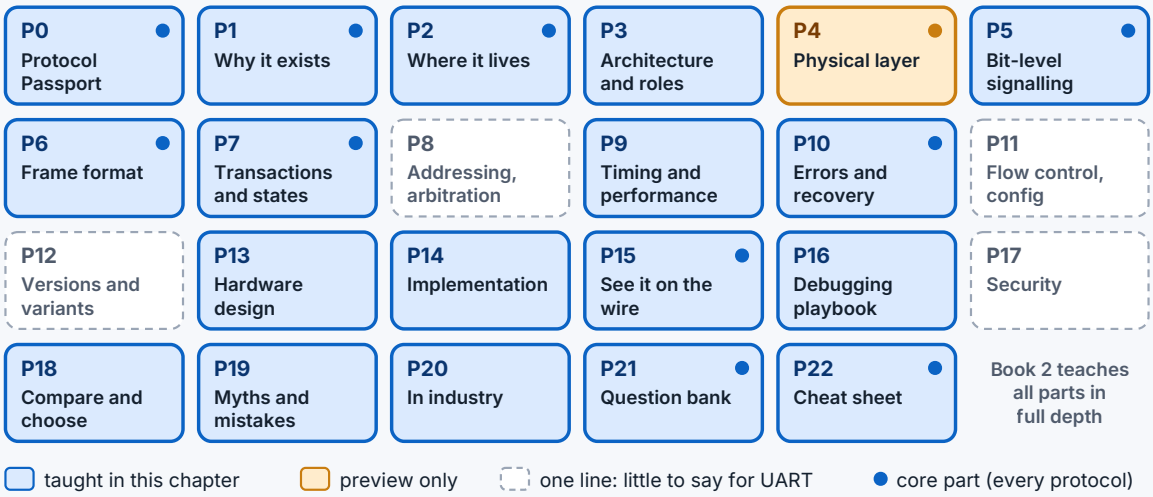


Figure 4.1 The 22 parts of the Protocol Dossier, plus the Passport (P0), in reading order. Blue parts are taught in this chapter, the amber part is a preview, and the dashed parts get one line because they hardly apply to a UART. A dot marks the core parts that even the shortest protocol section keeps.

PROTOCOL PASSPORT

UART

Full name	Universal Asynchronous Receiver-Transmitter
Born	Start-stop telegraphy (Krum, patent filed 1910); first single-chip UARTs around 1971
Governing spec	None for the character format itself; it is defined by UART datasheets. The wires use RS-232 (TIA-232-F), RS-422 or RS-485 (Book 2)
Edition taught	16550-class UART behaviour (PC16550D data sheet); RS-232 levels as a preview
Layers	Physical signalling and character framing (part of the data link)
Topology	Point-to-point: exactly one transmitter on each wire
Wires	TX, RX and ground; RTS and CTS are optional
Speed	Both ends agree on a baud rate; 9600 and 115 200 baud are the most common
Distance	Logic level: on a board or a short cable; RS-232: about 15 m; RS-485: much more (Book 2)
Max nodes	2
Duplex	Full duplex: TX and RX are separate wires
Sync / async	Asynchronous: no clock wire; each side has its own clock
Addressing	None
Error check	Optional parity bit; stop-bit (framing) check; break detection
Where you meet it	Debug consoles, GPS and radio modules, bootloaders, USB–UART bridges
Sister protocols	USART, RS-232, RS-422, RS-485, LIN; compare with SPI and I²C

4.2 The idea in plain words BASIC

Imagine two friends on two hills at night. They want to send letters with a lamp. They agree on three rules before they start. First, while nothing is being sent, the lamp stays **on**. Second, a message character begins with one beat of lamp **off**, which means “get set”. Third, after the “get set” beat come exactly eight beats of on or off, which carry the message, and then the lamp goes back on for at least one beat.

Each friend has a watch, and both watches tick at the same agreed pace. The receiver does not need to see the sender’s watch. She waits for the lamp to go off, starts her own watch, and looks at the lamp in the middle of each beat. Because she restarts her watch on every “get set”, a small difference between the two watches never has time to grow large.



Figure 4.2 The lamp code for the letter 'H'. The lamp is on while waiting, one 'off' beat says "get set", eight beats carry the message, and the lamp returns to on. Each friend keeps time with their own watch.

That is exactly how a UART works. The lamp is a wire. "On" is a high voltage, "off" is a low voltage. The agreed pace is the **baud rate**. The "get set" beat is the **start bit**, and the final "on" beat is the **stop bit**.

> **Definition UART**
> A **UART** (Universal Asynchronous Receiver-Transmitter) is a hardware block that turns bytes into a timed stream of bits on one wire, and turns such a stream back into bytes. People also say "UART" for the bit format it uses.

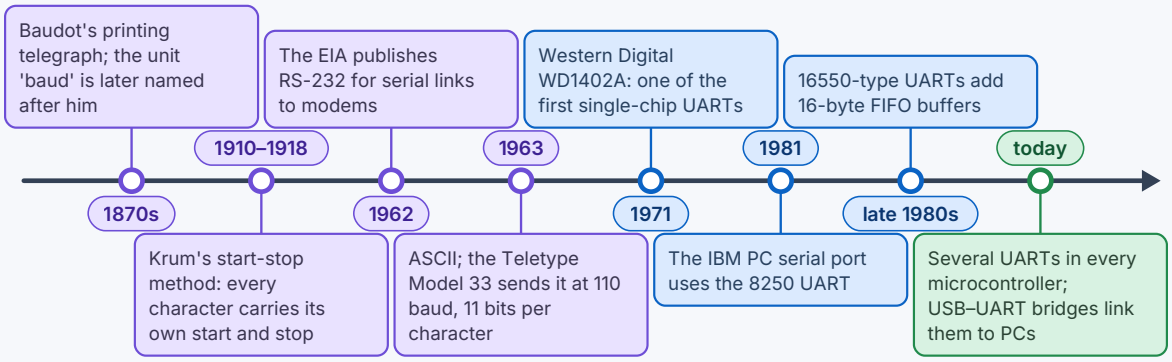
> **Definition Asynchronous**
> A link is **asynchronous** when no clock signal travels with the data. Each side uses its own clock, and the receiver finds the timing from the data itself.

4.3 Why the UART exists BASIC

Dossier part P1 · Why it exists

Before computers, teleprinters sent text over telegraph wires. Every character had to arrive at a machine whose motor was not locked to the sender's motor. The solution was the **start-stop** method: each character carries its own start signal and stop signal, so the receiver can re-align itself on every character. Howard Krum's patent for this method was filed in 1910 and issued in 1918. The unit "baud" honours Émile Baudot, who built a printing telegraph and its 5-unit code in the 1870s.

Computers kept the idea. In 1963 ASCII was published, and the Teletype Model 33 sent it at 110 baud with 11 bits per character: one start bit, eight bits for the character and two stop bits. So it printed 110 ÷ 11 = 10 characters per second. A UART was first a whole circuit board; around 1971 Western Digital put one on a single chip, the WD1402A. The IBM PC (1981) used the 8250 UART, and later 16550-type UARTs added 16-byte FIFO buffers. Today every micro-controller has one or more UARTs.



Not to scale. Dates checked November 2026; see the source log.

Figure 4.3 Milestones from start-stop telegraphy to the UARTs inside every microcontroller. The idea of a start and a stop around each character never changed; the hardware moved from motors to boards to single chips.

The problem the UART solves is still the same: send characters over very few wires, with no clock wire, between two devices that each have their own clock.

4.4 Where you meet it BASIC

Dossier part P2 · Where it lives

Open almost any product and you find UART links. A GPS module sends its position as lines of text. A Wi-Fi or Bluetooth module takes commands such as `AT` and answers `OK`. A router or single-board computer prints its boot messages on a **console** UART, so an engineer can watch it start even when the network is down. A **USB–UART bridge** chip lets a PC open the same link as a serial port.

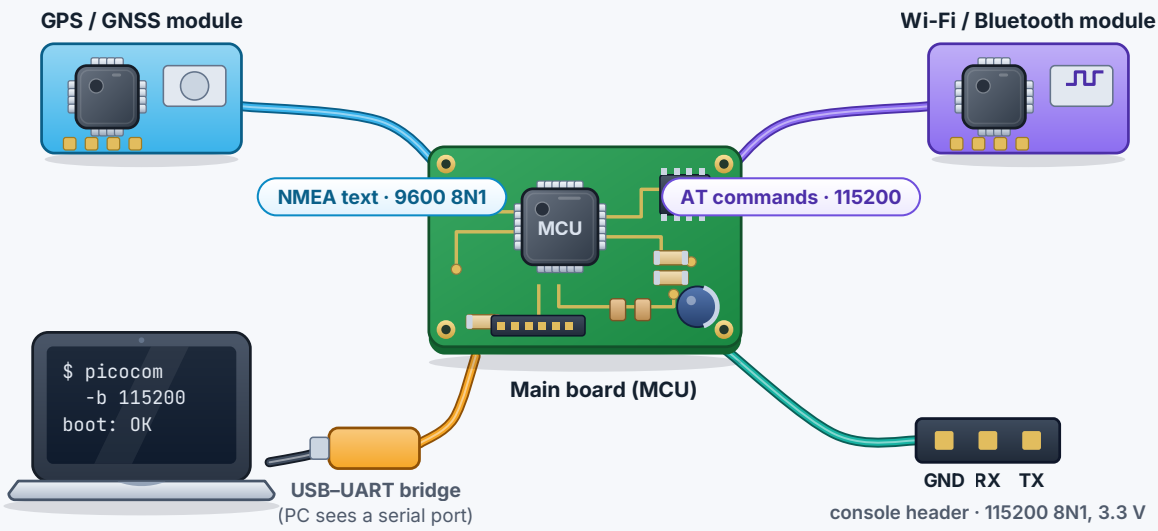


Figure 4.4 A typical board and its UART links: a GPS module, a radio module, a console header and a PC through a USB–UART bridge. Each link has two signal wires and a ground, and each has its own baud rate.

Notice that each link in Figure 4.4 is separate. A microcontroller with four UARTs can run four links at four different baud rates at the same time.

4.5 Roles and architecture BASIC

Dossier part P3 · Architecture and roles

A UART link has no controller and no target. It joins exactly two devices, and the link is symmetric: each device owns its own **TX** (transmit) pin and listens on its own **RX** (receive) pin. Because TX and RX are separate wires, both sides can send at the same time. This is **full duplex**.

Inside each device, the UART has a few simple blocks:

- a **baud-rate generator**, which divides a clock to make a tick, usually 16 ticks per bit;
- a **transmit FIFO** (a small queue of bytes) and a **transmit shift register**, which adds the start, parity and stop bits and pushes the bits out one by one;
- a **receive sampler** and **receive shift register**, which find each start bit, sample the bits and check the parity and stop bits;
- a **receive FIFO**, which holds received bytes together with their error flags for the software.

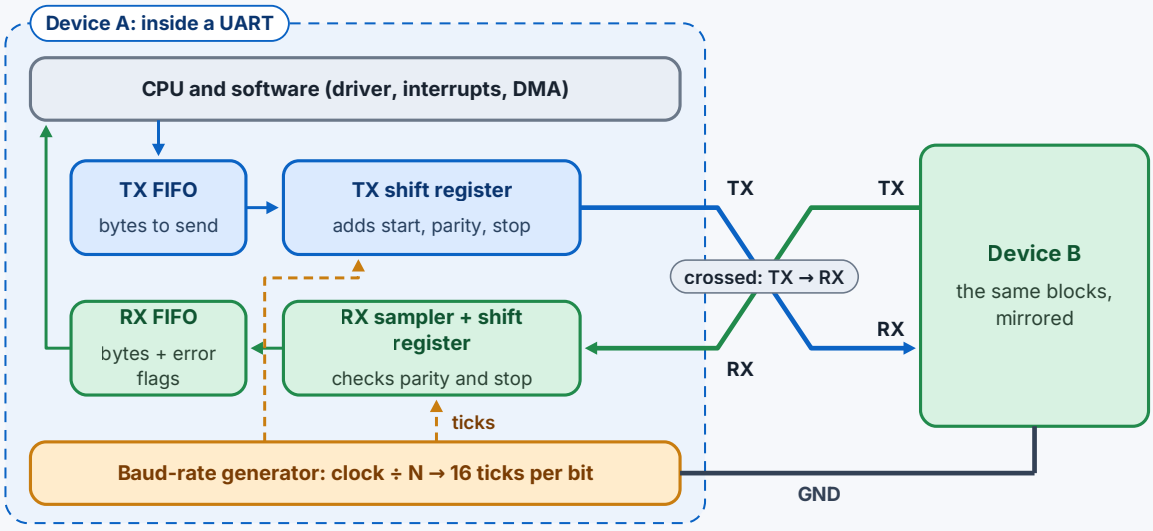


Figure 4.5 Inside a UART. Software writes bytes into the TX FIFO and reads them from the RX FIFO; the shift registers and the baud-rate generator do all the bit timing. Device A's TX wire goes to device B's RX pin, and the other way round.

Old RS-232 equipment adds two role names, **DTE** (data terminal equipment, such as a computer) and **DCE** (data communication equipment, such as a modem), which decide which connector pin is TX (Book 2).

Check yourself

1. Why does a UART link not need addresses?
2. What makes a UART link full duplex?

4.6 Physical layer: wires and voltage levels BASIC

Dossier part P4 · Physical layer (a preview; Book 2 has the full story)

At the chip pins, a UART uses **logic levels**: the same voltages as the chip's other pins. On a 3.3 V microcontroller, a 1 is about 3.3 V and a 0 is about 0 V; on a 5 V board, a 1 is about 5 V. The line **idles high**. The old telegraph names are still used: a 1 is called **mark** and a 0 is called **space**.

Wiring follows one rule: what one side transmits, the other side receives. So TX of A goes to RX of B, RX of A comes from TX of B, and the two grounds are joined. A ground wire is needed because a receiver measures the signal voltage against its own ground; without a shared ground, that voltage means nothing. Two optional wires, **RTS** and **CTS**, carry hardware flow control (Section 4.9).

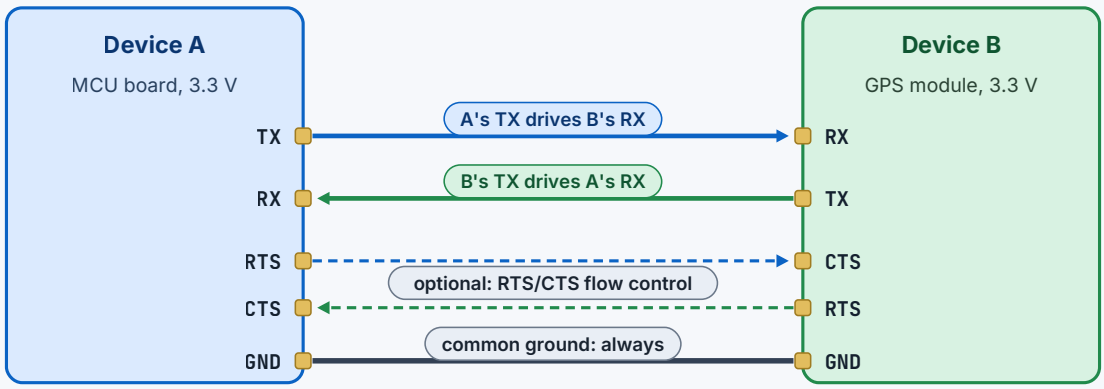


Figure 4.6 Wiring two UART devices. Each TX drives the other side's RX, the grounds are always joined, and RTS/CTS are optional. Blue lines are driven by device A, green lines by device B.

Logic levels are fine on a board or a short cable. They are single-ended (each signal is measured against ground), so ground noise and cable capacitance soon corrupt them on long cables. There, a transceiver chip changes the levels. **RS-232** uses inverted, bipolar levels. A driver sends a 0 (space) as +5 to +15 V and a 1 (mark) as −5 to −15 V. A receiver reads +3 to +15 V as 0 and −3 to −15 V as 1; the band from −3 V to +3 V is undefined. TIA-232-F allows up to 20 kbit/s and 2500 pF of cable load, which is about 15 to 20 m of cable. RS-422 and RS-485 use a differential pair and go much farther (Book 2).

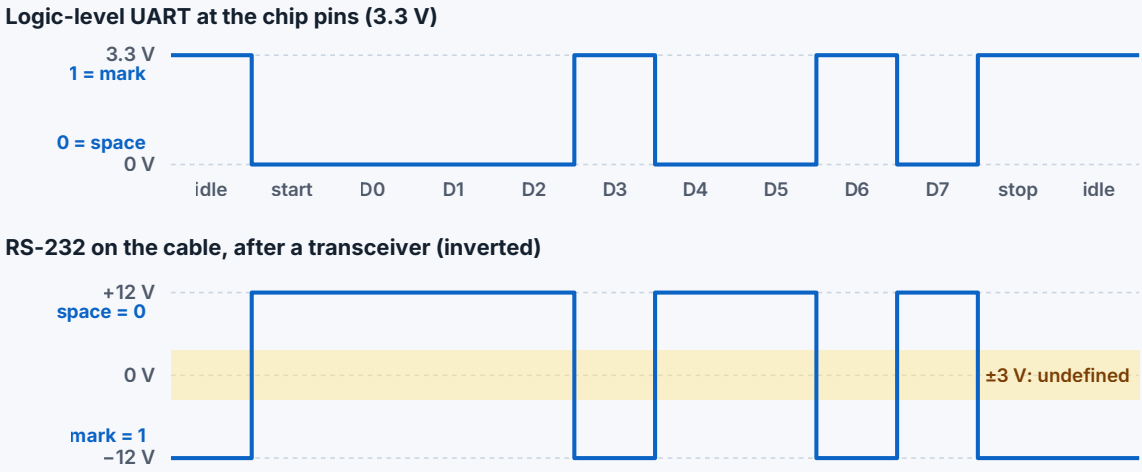


Figure 4.7 The same frame (the letter 'H') at logic level and on an RS-232 cable. RS-232 is inverted: the idle level (mark, 1) is a negative voltage and the start bit (space, 0) is positive. ±12 V is drawn as a typical example.

Common mistake **TX to TX**

Wrong: Connect TX to TX and RX to RX "because the names match".

Why: Two outputs then drive the same wire and two inputs listen to a wire that nobody drives. Nothing gets through, and two push-pull outputs fighting each other can be damaged.

Right: Cross the signals: TX → RX and RX ← TX. If a board labels its pins from the other side's point of view, check the schematic or measure which pin moves when the device sends.

Check yourself

1. A logic-level UART line sits at 3.3 V for a long time. Is it broken?
2. On an RS-232 cable, is the idle line positive or negative?

4.7 Bit-level signalling BASIC

Dossier part P5 · Bit-level signalling

A **bit** is one binary digit, 0 or 1. On a UART wire, each bit is one fixed slice of time with one voltage level. The length of that slice is the **bit time**, and the number of slices per second is the **baud rate**. A UART line has two levels, so one symbol carries one bit, and baud equals bits per second on the line. (Faster links send several bits per symbol; then baud and bit rate differ.)

Definition **Baud rate and bit time**

The **baud rate** is the number of symbols per second on the line. For a UART it is also the number of bits per second, counting start, parity and stop bits. The **bit time** is 1 ÷ baud rate.

A character travels like this:

1. **Idle.** The line is high (1).
2. **Start bit.** The transmitter pulls the line low (0) for one bit time. The falling edge tells the receiver "a character starts now".
3. **Data bits.** Usually 8 bits follow, **least significant bit (LSB) first**: D0, then D1, up to D7.
4. **Parity bit (optional).** One bit for a simple error check.
5. **Stop bit(s).** The line goes high (1) for at least one bit time, and stays high while idle.

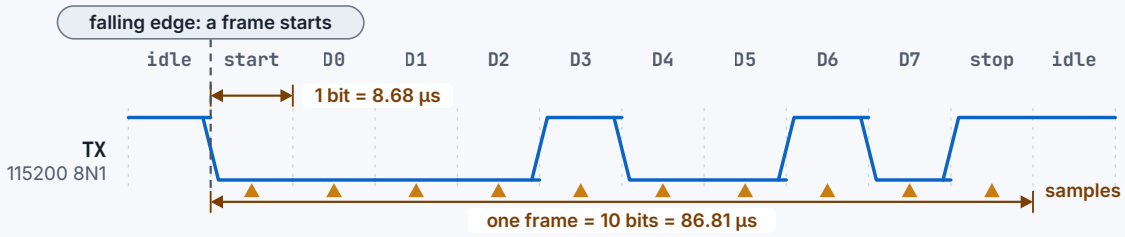


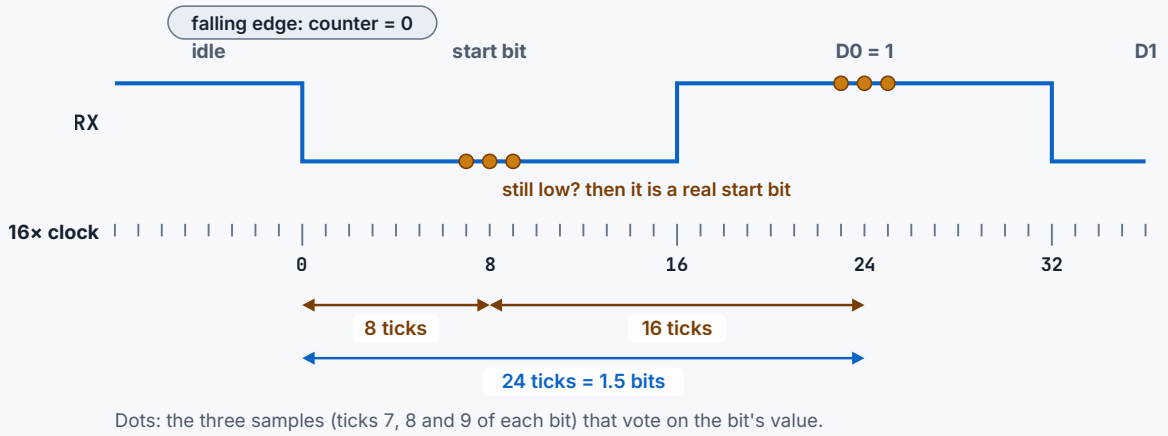
Figure 4.8 The letter 'H' (0x48) sent at 115 200 baud, 8N1. The falling edge starts the frame; the bits follow LSB first; the triangles show where the receiver samples, in the middle of each bit.

Read Figure 4.8 from left to right. 0x48 is 0100 1000 in binary, but the wire shows D0 first, so you see 0 0 0 1 0 0 1 0. Only D3 and D6 are high.

How the receiver samples. The receiver has no clock from the sender, so it uses its own clock, usually running at 16 times the baud rate. This is called **16× oversampling**. It works in four steps:

1. While idle, it watches the line on every tick, waiting for a falling edge.
2. When it sees the edge, it sets a counter to 0 and waits 8 ticks, which is half a bit, to reach the middle of the start bit.
3. It checks that the line is still low. If it is high again, the edge was noise, and it goes back to waiting. This is called false start-bit detection.
4. From there it samples every 16 ticks, so each later sample falls in the middle of a bit. The first data bit, D0, is sampled 8 + 16 = 24 ticks after the edge, which is 1.5 bit times.

The middle is the safest place to sample: it is farthest from both edges, so it tolerates the most clock error and the slowest edges. Many UARTs take three samples near the middle of each bit (ticks 7, 8 and 9 of 16 is a common choice) and use the majority. One noise spike then cannot change the bit.



Dots: the three samples (ticks 7, 8 and 9 of each bit) that vote on the bit's value.

Figure 4.9 16× oversampling. The receiver counts 8 ticks from the falling edge to reach the middle of the start bit, checks it is still low, then samples every 16 ticks. D0 is sampled 24 ticks (1.5 bits) after the edge.

KEY IDEA

A UART sends no clock. Both sides agree on the baud rate in advance, and the receiver re-synchronises on the falling edge of every start bit, then samples each bit in its middle.

Check yourself

1. At 16× oversampling, how many ticks after the start edge is D3 sampled?
2. Why does the receiver check the start bit again in its middle?

4.8 Frame format BASIC

Dossier part P6 · Frame format

One character with its start, parity and stop bits is called a **frame**. Engineers describe the frame format with a short code: data bits, parity, stop bits. So **8N1** means 8 data bits, no parity, 1 stop bit. It is by far the most common format.

Field	Bits	Value	Purpose
Start	1	always 0	makes a falling edge so the receiver can synchronise
Data	5 to 8 (usually 8)	the character, LSB first	the payload
Parity	0 or 1	even, odd or none	a simple error check
Stop	1 or 2 (1.5 with 5 data bits)	always 1	returns the line to idle; checked by the receiver



Figure 4.10 The letter 'H' in three formats. 8N1: a start bit (always 0), eight data bits with D0 first, and a stop bit (always 1). 7E1: only D0 to D6 are sent, and the even parity bit is 0 because those bits hold two 1s. 8O2: the odd parity bit is 1 and two stop bits follow. The value under each field is its level on the wire.

The table of common formats below shows how the bit count and efficiency change. The numbers come from Lab 4.2.

Format	Data	Parity	Stop	Bits on wire	Efficiency	Typical use
8N1	8	none	1	10	80.0 %	the default almost everywhere
7E1	7	even	1	10	70.0 %	7-bit ASCII text with a parity check
8E1	8	even	1	11	72.7 %	binary bytes with a parity check
8O2	8	odd	2	12	66.7 %	rare; it shows every option at once

Why a frame starts low and ends high. The line idles high. A start bit that is always 0 guarantees a falling edge at the start of every frame, even when the data begins with 0s. A stop bit that is always 1 guarantees that the line is high again before the next start bit, so the next frame also begins with a clean falling edge. The stop bit also gives the receiver one known value to check. If it reads 0 there, something is wrong; Section 4.11 names that error.

A classic 16550-type UART allows 5, 6, 7 or 8 data bits, parity even, odd or none, and 1 or 2 stop bits (1.5 stop bits with 5 data bits, a leftover from 5-bit teleprinter codes). Its receiver checks only the first stop bit.

Check yourself

1. How many bits does one 7E2 frame have on the wire?
2. Which data bit comes right after the start bit?

4.9 Transactions and state machines INTERMEDIATE

Dossier part P7 · Transactions

A UART has no handshake of its own. The transmitter sends a byte whenever software gives it one, and it never waits for an acknowledgement. Any reply, such as `OK` after a command, belongs to a higher protocol that rides on the UART. Figure 4.12 shows a host and a module talking in an **AT-style** command language: short text commands that start with `AT` and end with a carriage return and line feed (`\r\n`).

Sometimes a receiver cannot keep up. **Flow control** lets it ask the sender to pause:

- **RTS/CTS (hardware flow control).** Each side drives its RTS (request to send) output to say "I can receive", and the other side's CTS (clear to send) input sees it. A transmitter sends only while its CTS input is active. RTS of one side goes to CTS of the other.
- **XON/XOFF (software flow control).** The receiver sends the character XOFF (0x13, DC3) to say "pause" and XON (0x11, DC1) to say "go on". No extra wires are needed, but the data must never contain those two byte values, and the pause comes a few characters late.

Lab 4.3 shows real XON/XOFF behaviour: the operating system's serial layer holds the host's bytes while it is paused and releases them after XON.

The receiver's work is a small **state machine**. A state machine is a set of states, and rules for moving between them.

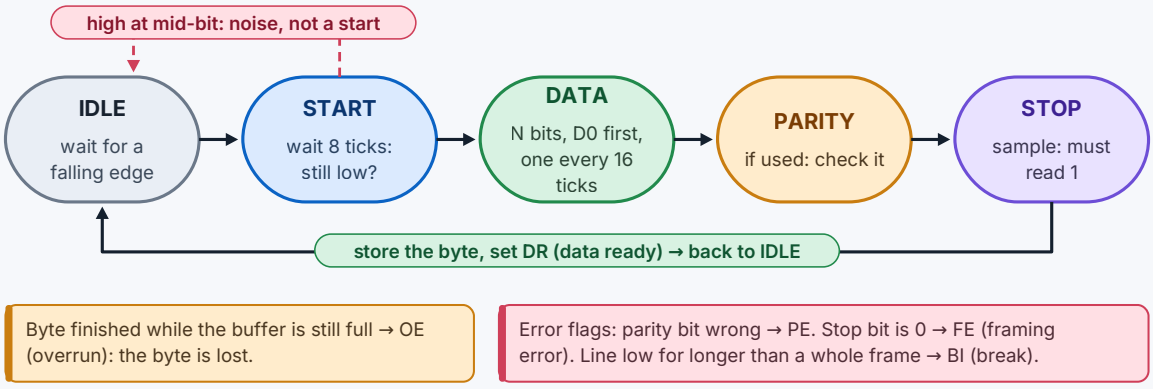


Figure 4.11 The UART receiver's state machine. It waits in IDLE for a falling edge, confirms the start bit, samples the data bits, checks parity if used, and checks the stop bit, setting an error flag when something is wrong.

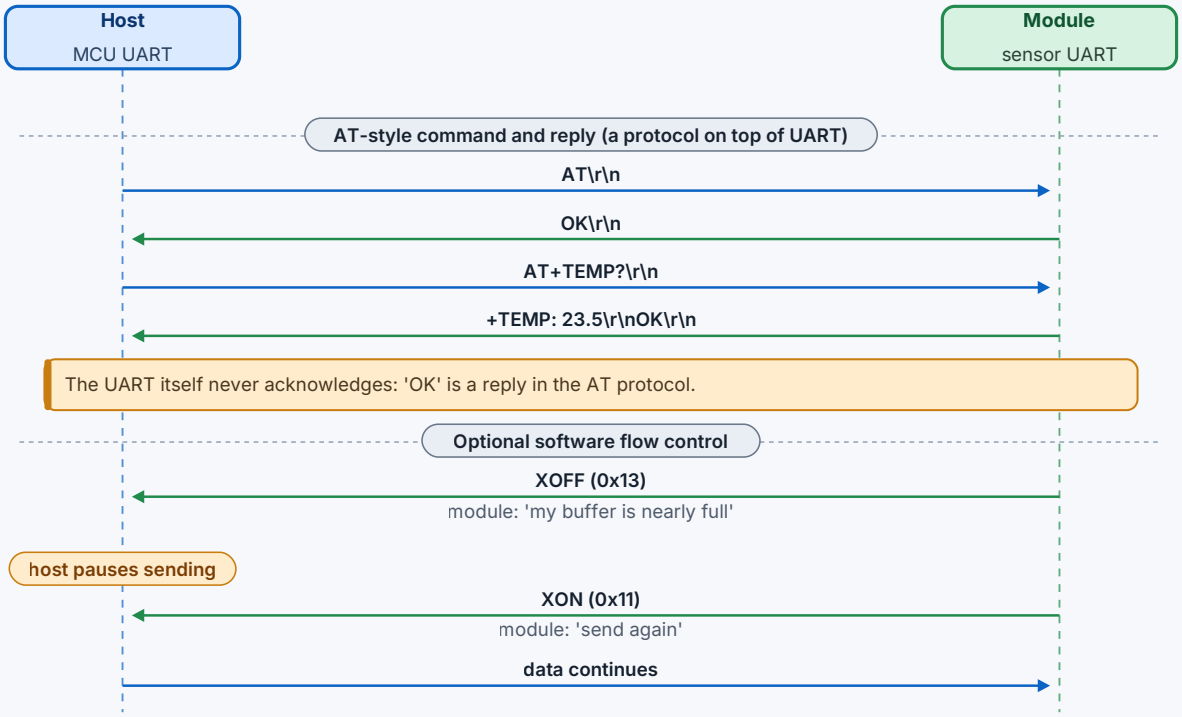


Figure 4.12 A host and a module exchange AT-style commands over a UART. The replies are part of the command protocol, not of the UART. In the second part the module uses XOFF and XON to pause and resume the host.

> **Note** **Parts P8, P11, P12 and P17 in one line each**
>
> **P8:** two devices and one transmitter per wire, so no addresses and no collisions (multidrop RS-485 adds them, Book 2). **P11:** no discovery or configuration protocol; both ends are set by hand or firmware. **P12:** the format has not changed in decades; only the electrical layers and features such as FIFOs and 9-bit modes vary (Book 2). **P17:** no security at all; anyone with a probe can read the line, and an open console can be a way into a product (Book 35).

4.10 Timing and performance INTERMEDIATE

Dossier part P9 · Timing and performance

Three formulas give every UART timing number.

$$T_{\text{bit}} = 1 / B \qquad T_{\text{frame}} = n \times T_{\text{bit}} \qquad \eta = d / n$$

where:

- T_{bit} is the bit time, in seconds;
- B is the baud rate, in baud (bits per second on a UART);
- T_{frame} is the frame time, in seconds;
- n is the number of bits in one frame (start + data + parity + stop);
- d is the number of data bits;

- η (eta) is the efficiency: the share of line time that carries data.

WORKED EXAMPLE 4.1

Bit time, frame time and throughput at 115 200 baud

Problem. A module sends text at 115 200 baud, 8N1. Find the bit time, the frame time, the efficiency and the largest number of bytes per second.

Step 1 — bit time. T_{bit} = 1 / 115 200 = 8.68 µs.

Step 2 — frame time. 8N1 has n = 1 + 8 + 0 + 1 = 10 bits, so T_{frame} = 10 × 8.68 µs = 86.81 µs.

Step 3 — efficiency. η = 8 / 10 = 80 %.

Step 4 — throughput. One frame carries one byte, so the link moves at most 115 200 / 10 = 11 520 bytes per second in each direction.

Answer. 8.68 µs per bit, 86.81 µs per byte, 80 %, and 11 520 bytes/s. At 9600 baud the same steps give 104.17 µs, 1.042 ms and 960 bytes/s (Lab 4.2, `out/bit-times.txt`, lists more rates). The result tells us that a "115 200 baud" link carries about 11.5 kB of data per second, not 115 kB.

How far can the two clocks differ? Each side makes its bit time from its own clock, so the two bit times are never exactly equal. The receiver re-synchronises on every start edge, so only the error inside one frame matters. The worst place is the last sample: the middle of the stop bit.

WORKED EXAMPLE 4.2

The baud-mismatch budget for 8N1

Problem. Find the largest total difference between the two clocks that still lets an 8N1 receiver read every frame.

Step 1 — where is the last sample? The stop bit is bit number 9 (start = 0, D0 to D7 = 1 to 8). Its middle is 9.5 bit times after the start edge. With 16 ticks per bit, that is 16 × 9.5 = 152 ticks.

Step 2 — how far may it move? A sample in the middle of a bit can move half a bit (8 ticks) either way before it falls into the next or the previous bit.

Step 3 — ideal limit. The error grows evenly over the frame, so the allowed clock difference is 8 / 152 = 0.5 / 9.5 = 5.26 %.

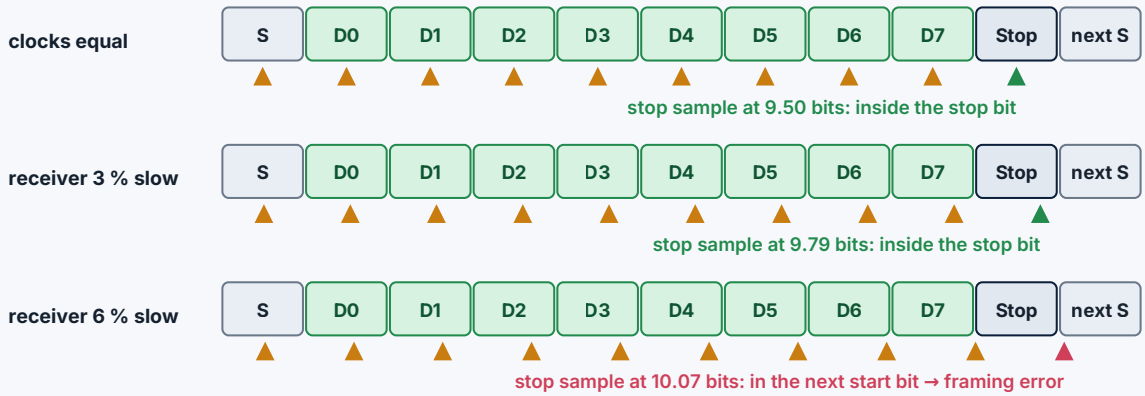
Step 4 — subtract the start-edge uncertainty. The receiver only notices the edge on its next tick, so its start can be up to one tick late. That leaves 7 / 152 = 4.61 %.

Answer. The total mismatch between the two clocks must stay below about 5 % (4.6 % with 16× sampling). Real edges are not sharp, so engineers keep a wider margin, as the lab table shows:

OUTPUT budget.txt

```
Format  Stop sample   Ideal    +/-1 tick  Normal   Nasty
        (bits/ticks)  +/-8 t   +/-7 t     +/-5 t   +/-3 t
------  ------------  -------  ---------  -------  -------
8N1      9.5 /   152    5.26%      4.61%    3.29%    1.97%
7E1      9.5 /   152    5.26%      4.61%    3.29%    1.97%
8E1     10.5 /   168    4.76%      4.17%    2.98%    1.79%
8O2     10.5 /   168    4.76%      4.17%    2.98%    1.79%
```

This tells us that more bits per frame means less tolerance: 8E1 and 8O2 have their stop-bit sample at 10.5 bits, so they tolerate only 4.76 % even in the ideal case.

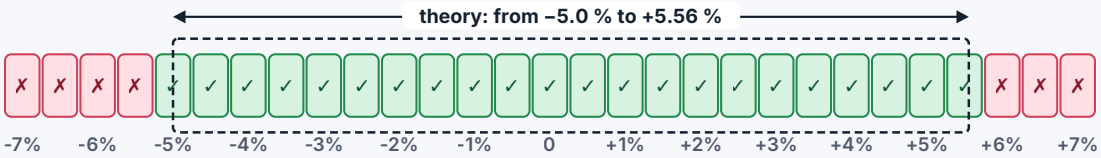


Triangles: where the receiver samples (middle of each of its own bits). The error adds up: 9.5 × the clock error at the stop bit.

Figure 4.13 Where the receiver samples when its clock is slow. With equal clocks the stop-bit sample sits in the middle of the stop bit. At 3 % slow it is 0.29 bit late but still inside. At 6 % slow it lands in the next start bit, and the receiver reports a framing error.

The “normal” and “nasty” columns follow Maxim (now Analog Devices) application note 2141: on a real link you can trust only the middle 75 % of each bit (a short buffered link) or the middle 50 % (a long, slow-edged RS-232 cable), which leaves a total budget of ±3.3 % or ±2 %, shared by both clocks. Microchip’s AVR data sheets recommend a receiver error of at most ±2.0 % for 8 data bits in normal-speed mode (16× sampling, U2X = 0). A practical rule: **keep each side within about ±2 %, and closer on long cables.** A crystal easily does this; an uncalibrated internal RC oscillator may not.

Lab 4.1 tested the rule on a real capture. It decoded the same “Hi!” while setting the receiver’s baud rate off by −7 % to +7 % in steps of 0.5 %.



Receiver baud offset (the sender stays at exactly 115200). Green: decodes 48 69 21 with no errors.
Red: wrong bytes or framing errors. Real decodes from Lab 4.1 (baud-sweep.txt).

Figure 4.14 The same captured "Hi!" decoded by sigrok with the receiver's baud rate offset from −7 % to +7 %. Every offset inside the theoretical window decodes correctly; every offset outside fails. Real decodes from Lab 4.1.

Why is the window from −5.0 % to +5.56 %, and not symmetric? In Lab 4.1 the frames follow each other with no gap. The stop-bit sample, 9.5 receiver bits after the edge, must land between 9 and 10 sender bits. A slow receiver reaches the next start bit at −5.0 %; a fast one reaches D7 at +5.56 %.

Making the baud rate from a clock. A baud-rate generator divides its clock by a whole number N:

$$B = f_{\text{clk}} / (k \times N) \qquad \text{error} = B_{\text{actual}} / B_{\text{wanted}} - 1$$

where f_{clk} is the clock frequency in hertz, k is the oversampling ratio (16, sometimes 8), and N is a whole number (the divisor). Because N must be whole, the actual baud rate is often a little off.

WORKED EXAMPLE 4.3

A 16 MHz clock at 115 200 baud

Problem. A microcontroller runs from a 16 MHz crystal and uses 16× oversampling. What divisor, actual baud rate and error do you get for 115 200 baud? Is it safe?

Step 1 — exact divisor. N = 16 000 000 / (16 × 115 200) = 8.68.

Step 2 — round. The nearest whole number is 9. (On AVR chips the register value is N − 1 = 8.)

Step 3 — actual baud. 16 000 000 / (16 × 9) = 111 111 baud.

Step 4 — error. 111 111 / 115 200 − 1 = −3.55 %.

Answer. −3.55 % uses most of the ~4.6 % budget by itself, so it is risky. Better choices: 8× oversampling (N = 17, +2.12 %), or a crystal that divides exactly, such as 18.432 MHz (N = 10, 0 %). Lab 4.2 checks several real clocks:

OUTPUT divisors.txt

```
Clock               OS   Baud    Exact N  N    Actual baud  Error    Verdict
------------------  ---  ------  -------  ---  -----------  -------  -------
1.8432 MHz crystal  16x    9600    12.00   12        9,600  +0.000%  OK
1.8432 MHz crystal  16x  115200     1.00    1      115,200  +0.000%  OK
16 MHz crystal      16x    9600   104.17  104        9,615  +0.160%  OK
16 MHz crystal      16x  115200     8.68    9      111,111  -3.549%  risky
16 MHz crystal       8x  115200    17.36   17      117,647  +2.124%  risky
18.432 MHz crystal  16x  115200    10.00   10      115,200  +0.000%  OK
48 MHz clock        16x  115200    26.04   26      115,385  +0.160%  OK
8 MHz clock         16x  115200     4.34    4      125,000  +8.507%  fails
50 MHz FPGA clock    1x  115200   434.03  434      115,207  +0.006%  OK

OS = oversampling (ticks per bit). OK: |error| <= 2 %;
risky: up to 4.6 %; fails: beyond 4.6 % (8N1 budget above).
```

The table tells us why “UART crystals” such as 1.8432 MHz exist: they divide exactly to the common baud rates (the 16550 data sheet lists divisor 12 for 9600 baud and divisor 1 for 115 200).

KEY IDEA

The stop-bit sample sits 9.5 bit times after the start edge, so the total clock mismatch must stay below about 5 %. Real designs keep each side within about ±2 % and check the divisor rounding error first.

4.11 Errors and recovery INTERMEDIATE

Dossier part P10 · Errors and recovery

A UART can notice four kinds of trouble. It does not correct any of them and never asks for a resend; recovery is the job of the higher protocol.

Parity error. With parity on, the transmitter adds one bit so that the total number of 1s in the data bits plus the parity bit is even (**even parity**) or odd (**odd parity**). The receiver counts again. If the count is wrong, it sets a parity error flag.

WORKED EXAMPLE 4.4

Parity by hand for “Hi!”

Problem. Find the even and odd parity bits for the three characters of “Hi!” in 8-bit frames.

Step 1 — write each byte in binary. ‘H’ = 0x48 = 0100 1000; ‘i’ = 0x69 = 0110 1001; ‘!’ = 0x21 = 0010 0001.

Step 2 — count the 1s. ‘H’ has 2, ‘i’ has 4 and ‘!’ has 2.

Step 3 — choose the parity bit. All three counts are already even, so the even parity bit is 0 and the odd parity bit is 1 for every character.

Answer. Even parity bits 0, 0, 0; odd parity bits 1, 1, 1. Lab 4.2 agrees, and also lists the order of the bits on the wire:

OUTPUT parity.txt

```
Char  Hex   Binary     On the wire (start, D0..D7, stop)  1s  Even  Odd
----  ----  ---------  ---------------------------------  --  ----  ---
'H'   0x48  0100 1000  0 0 0 0 1 0 0 1 0 1                 2     0    1
'i'   0x69  0110 1001  0 1 0 0 1 0 1 1 0 1                 4     0    1
'!'   0x21  0010 0001  0 1 0 0 0 0 1 0 0 1                 2     0    1
```

This tells us that parity depends only on how many 1s there are, not on where they are.

Parity catches any odd number of flipped bits (1, 3, 5 …). It misses any even number (2, 4 …), and it cannot tell which bit is wrong. Book 1 teaches stronger checks such as CRCs.

Framing error. The receiver read the stop bit as 0. The usual causes are a wrong baud rate, a wrong format (for example the sender uses 7 data bits), noise, or a break.

Overrun error. A new byte finished arriving while the receive buffer was still full, because software did not read the old byte in time. The new byte is lost. Nothing is wrong on the wire; the receiver was too slow.

Break. The line stays at 0 for longer than a whole frame (start, data, parity and stop bits). That is not a valid character, so it is used as a signal: to get attention, to reset a link, or to mark the start of a packet in some protocols (DMX512 lighting control, Book 2). A UART sends a break with a "set break" control bit that forces TX low.

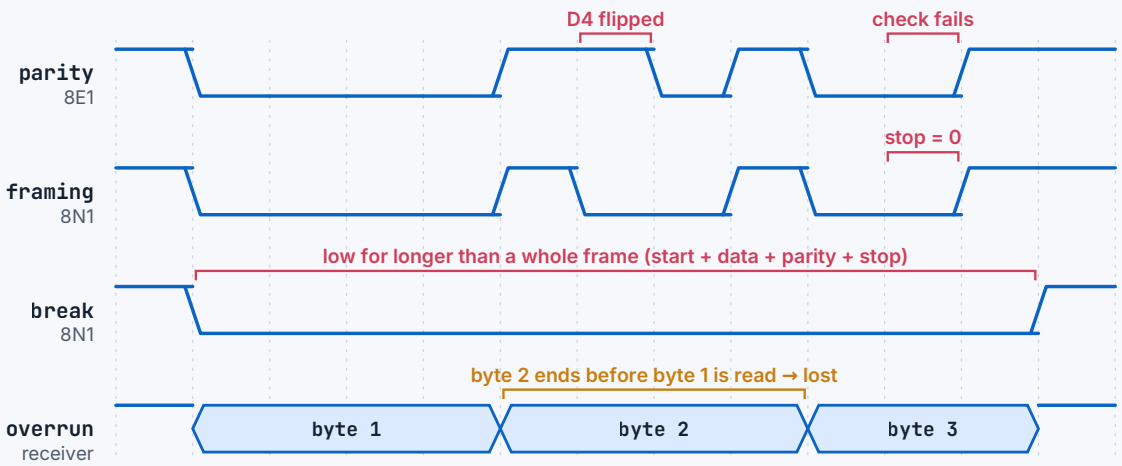


Figure 4.15 The four UART errors. Parity: noise flipped D4, so the parity check fails. Framing: the stop bit is 0. Break: the line stays low for longer than a frame. Overrun: byte 2 ends before software has read byte 1.

A break captured from the lab transmitter looks like this. The decoder first sees a start bit followed by all-zero data and a missing stop bit, so it reports `00` with a framing error. Then, when the line has stayed low for a whole frame, it reports the break:

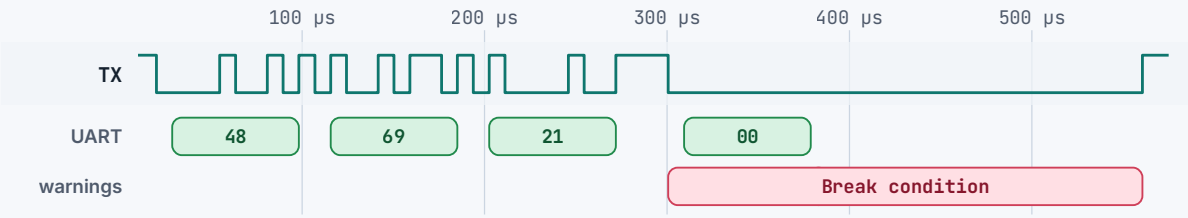


Figure 4.16 A real break captured in Lab 4.1. After "Hi!" the transmitter holds TX low for 260 µs, three frame times. sigrok decodes 00 with a framing error and then reports a break condition.

A 16550-type UART reports these conditions in its **line status register**, one bit per flag:

Flag	Name	Set when
DR	Data ready	a complete byte is waiting in the receive buffer or FIFO
OE	Overrun error	a byte arrived before the previous one was read, so one was lost
PE	Parity error	the parity bit does not match the data
FE	Framing error	the stop bit was 0
BI	Break interrupt	the input stayed at 0 for longer than a whole frame

In FIFO mode each error flag travels with the byte it belongs to, so software knows exactly which byte was bad.

Check yourself

1. Two bits of a byte flip on the way. Does even parity catch it?
2. Which error says nothing about the wire, only about the receiving software?

4.12 Hardware design INTERMEDIATE

Dossier part P13 · Hardware design (short)

Most UART hardware problems come from three things: voltage levels, ground and the bridge to a PC.

Voltage levels. Every input has an **absolute maximum** voltage, usually a little above its own supply, so a 5 V signal can damage a 3.3 V input unless the data sheet says the pin is 5 V tolerant. The other direction is a question of thresholds: an input reads a 1 only above its **VIH** (minimum input-high voltage).

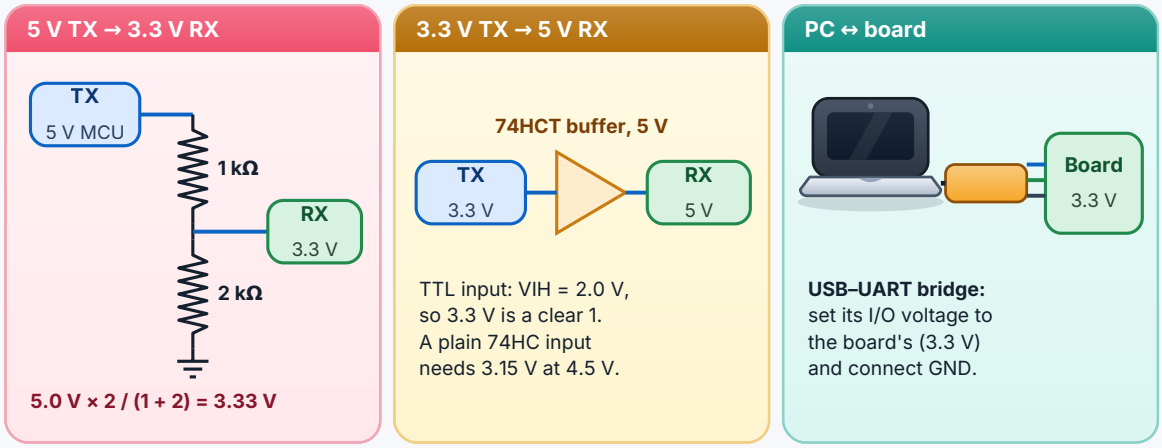


Figure 4.17 Three common UART hardware fixes. A resistor divider brings a 5 V TX down to 3.3 V. A 74HCT buffer, which has TTL-level inputs, turns a 3.3 V TX into a full 5 V signal. A USB–UART bridge set to the board's voltage links a PC to the board.

OUTPUT `levels.txt`

```
Divider 1 kOhm over 2 kOhm, 5.0 V in: 3.33 V at the RX pin

A 3.3 V high level into a 5 V input     VIH min   Margin
-----------------------------------  -------  -------
74HCT input (TTL levels)               2.00 V  +1.30 V
74HC input at VCC = 4.5 V              3.15 V  +0.15 V
CMOS input needing 0.7 x VCC (5.0 V)   3.50 V  -0.20 V
```

The lab numbers show the problem clearly. A 3.3 V high level gives a 74HCT input 1.3 V of margin, but only 0.15 V to a 74HC input at 4.5 V, and none at all to an input that needs 0.7 × 5 V. Always look up VIH in the receiver's data sheet.

IN HARDWARE

Level shifting, ground and bridges

- **5 V TX → 3.3 V RX:** a divider of 1 kΩ (top) and 2 kΩ (bottom) gives 5.0 × 2 / 3 = 3.33 V. It is cheap and fine at common baud rates, because the resistors and the pin capacitance slow the edges only a little. A level-translator chip is the cleaner choice at high speed.
- **3.3 V TX → 5 V RX:** works directly only if the 5 V input has TTL-level thresholds (VIH about 2.0 V). Otherwise add a buffer with TTL-level inputs, such as a 74HCT gate powered from 5 V.
- **Ground:** always join the grounds, with a wire that is as short as the signal wires.
- **USB–UART bridges:** chips such as the FTDI FT232R, Silicon Labs CP2102 and WCH CH340 appear on a PC as a serial port: `/dev/ttyUSB0` on Linux (or `/dev/ttyACM0` for USB CDC-ACM devices) and a `COM` port on Windows. Choose one whose I/O voltage matches your board.

- **Idle level at power-up:** a TX pin that floats before the firmware sets it up can look like start bits; a pull-up resistor on the RX input keeps the line idle.

Common mistake **5 V into a 3.3 V pin**

Wrong: Connect a 5 V Arduino-style board's TX straight to a 3.3 V module's RX.

Why: The 5 V level is above the 3.3 V pin's absolute maximum. The pin may work for a while and then fail, or fail at once.

Right: Use a divider or a level translator, or a module whose data sheet says the pin is 5 V tolerant.

4.13 Implementation INTERMEDIATE

Dossier part P14 · Implementation

A UART is simple enough to build from a counter and a shift register. Here is the transmitter used in Lab 4.1, written in Verilog. It makes one bit time by counting `CLKS_PER_BIT` clock cycles, 434 cycles at 50 MHz and 115 200 baud, and then moves to the next bit of a 10-bit frame register.

VERILOG uart_tx.v

```
    always @(posedge clk) begin
        if (rst) begin
            tx      <= 1'b1;
            active  <= 1'b0;
            clk_cnt <= 16'd0;
            bit_idx <= 4'd0;
            frame   <= 10'h3FF;
        end else if (!active) begin
            tx <= ~brk;                       // idle high, or break low
            if (start && !brk) begin
                frame   <= {1'b1, data, 1'b0};
                active  <= 1'b1;
                clk_cnt <= 16'd0;
                bit_idx <= 4'd0;
                tx      <= 1'b0;              // start bit begins now
            end
        end else if (clk_cnt == CLKS_PER_BIT - 1) begin
            clk_cnt <= 16'd0;
            if (bit_idx == 4'd9) begin
                active <= 1'b0;               // stop bit lasted one bit
                tx     <= 1'b1;
            end else begin
                bit_idx <= bit_idx + 4'd1;
                tx      <= frame[bit_idx + 4'd1];  // next bit
            end
        end else begin
            clk_cnt <= clk_cnt + 16'd1;
        end
    end
```

Read the code with the frame in mind. `frame` holds `{stop, data, start}`, so bit 0 is the start bit (0) and bit 9 is the stop bit (1). While idle, `tx` is high; the `brk` input forces it low, which is

how the lab makes a break. A counter, `clk_cnt`, runs from 0 to `CLKS_PER_BIT` − 1 for every bit, and `bit_idx` counts the bits of the frame. A receiver in Verilog is a little longer (it needs the 16× sampler and the state machine of Figure 4.11); Book 2 builds one and tests it with cocotb.

On a computer, software opens the serial port and sets its baud rate and format. In Python the `pyserial` package does this:

PYTHON

```python
import serial
port = serial.Serial("/dev/ttyUSB0", 9600, bytesize=7, parity="E",
                     stopbits=1, timeout=1)        # 9600 baud, 7E1
port.write(b"AT\r\n")
```

On a real UART, the driver turns these settings into the chip's control registers: the divisor, the data size, the parity and the stop bits. Lab 4.3 runs two Python programs that talk through a **pseudo-terminal** (pty): a pair of software serial ports inside the operating system. A pty is useful for testing programs, but it has no wire. It copies bytes and never times bits. The lab shows what that means:

OUTPUT run.log

```
2. 1000 bytes at '9600 baud 8N1'
  received 1000 bytes, identical: True
  a real 9600 8N1 wire needs 1.042 s (1000 x 10 bits / 9600)
  the pty took 0.2 ms  (your number will differ)

3. Host asks for 9600 7E1, module uses 115200 8N1
  host requested 9600 7E1, kernel stored: 9600 8N1
  module requested 115200 8N1, kernel stored: 115200 8N1
  module received b'Hi!'  (a real wire would give garbage)
```

Look at parts 2 and 3. (Part 1 is the AT exchange of Figure 4.12; part 4 is XON/XOFF.) A real 9600 baud link needs 1.042 s for 1000 bytes; the pty took well under a millisecond. And when the host asked for 9600 7E1, the Linux pty driver simply stored 8 data bits and no parity: it ignores data size and parity, because there is no wire to apply them to. So the two sides "agreed" even with different settings; on real hardware the same mismatch gives garbage. XON/XOFF in part 4 is different: the operating system's serial layer does it in software, so it works on a pty just as on a real port. So a pty is good for testing the logic of serial programs (commands, replies, timeouts, XON/XOFF), but it cannot show baud errors, framing, parity, breaks or timing. For those you need a signal-level model (Lab 4.1) or real hardware.

4.14 See it on the wire BASIC

Dossier part P15 · See it on the wire

SEE IT ON THE WIRE

Lab 4.1: “Hi!” at 115 200 baud, 8N1

The Verilog transmitter sent “Hi!” and the simulator saved the TX line to a VCD (value change dump) file, the kind of file a logic analyzer saves. sigrok’s `uart` decoder read it at 115 200 baud, 8N1: like a hardware receiver, it waits for a falling edge, samples the middle of each bit, and flags framing errors, parity errors and breaks. The frames follow each other with no gap, so “Hi!” takes 3 × 86.81 µs = 260.4 µs.

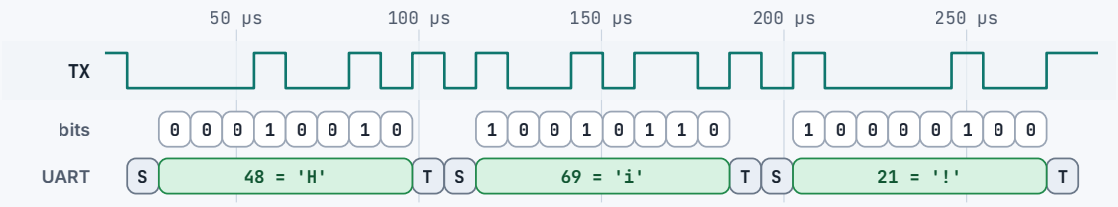


Figure 4.18 The real TX line from Lab 4.1 and sigrok’s decode. The ‘bits’ row shows each sampled data bit, D0 first; the ‘UART’ row shows the start bit (S), the byte with its ASCII character, and the stop bit (T, sigrok’s short name for stop).

DECODED CAPTURE `decoded-ascii.txt`

```
uart-1: H
uart-1: i
uart-1: !
```

Now the same capture goes through a receiver with wrong settings. These are the most common faults on real benches:

OUTPUT `wrong-settings.txt`

```
Receiver setting              Bytes decoded       Errors
----------------------------  ------------------  ------
115200 8N1 (correct)          48 69 21                 0
57600 8N1 (half speed)        CC                       2
115200 8E1 (expects parity)   48 5A                    4
115200 7N1 (7 data bits)      48 69 21                 3
115200 8N1, line inverted     AB A9                    0
```

Half the baud rate gives a wrong byte and framing errors. Expecting parity makes the receiver read the stop bit as parity. With 7 data bits it reads D7 as the stop bit: the right letters, but a framing error on every byte. An inverted line (RS-232 read as logic level) gives wrong bytes and no errors at all, the hardest case to spot.

4.15 Debugging playbook INTERMEDIATE

Dossier part P16 · Debugging playbook

The single most useful measurement is the width of the shortest pulse on the line. That is one bit time, so 1 ÷ width is the real baud rate: 8.68 µs means 115 200 baud and 104 µs means 9600 baud. It finds most “garbage” problems in a minute.

DEBUGGING PLAYBOOK

Symptom	Likely cause	What to measure	Fix
Garbage characters	baud rate or format differs; line inverted; wrong clock	bit width on a scope (1 ÷ baud); decode with an analyzer at several settings	set both ends the same; add or remove an RS-232 transceiver; fix the clock setting
Nothing received	TX to TX; no ground; wrong port; CTS not asserted; device not sending	does the TX pin move when the device sends? continuity of TX→RX and GND	cross TX/RX, join grounds, open the right port, turn off flow control or wire RTS/CTS
Every byte has a framing error	large baud error; sender uses parity or 7 bits; inverted line	stop-bit level in a capture; the analyzer's frame errors	match baud and format exactly
Missing characters at high speed	receive overrun: software reads too slowly	OE flag; interrupt latency; FIFO level	use the FIFO with interrupts or DMA; read in blocks; use RTS/CTS
Random characters at power-up	TX floats or glitches before setup; boot ROM prints at another baud (the ESP8266 boot log is at 74 880 baud)	line level during reset; capture the first second	pull-up on RX; ignore or decode the boot output at its own rate
Works at 9600, fails at 115 200	divisor error grows (16 MHz: +0.16 % at 9600 but −3.55 % at 115 200); slow edges; long cable	actual bit width; edge rise time	change oversampling or crystal; shorten the cable; lower the baud rate

Common mistake **Wrong baud rate**

Wrong: "The text is garbage, so the cable must be bad."

Why: A baud mismatch of more than about 5 % turns good signals into wrong bytes and framing errors, as Lab 4.1 shows. The cable is rarely the cause on a short link.

Right: Measure the bit width, set both ends to the same baud rate and format, then look at the cable.

4.16 Compare and choose BASIC

Dossier part P18 · Compare and choose

The UART has three close relatives on a circuit board: SPI, I²C and USB. Figure 4.19 shows their wiring, and the table compares them.

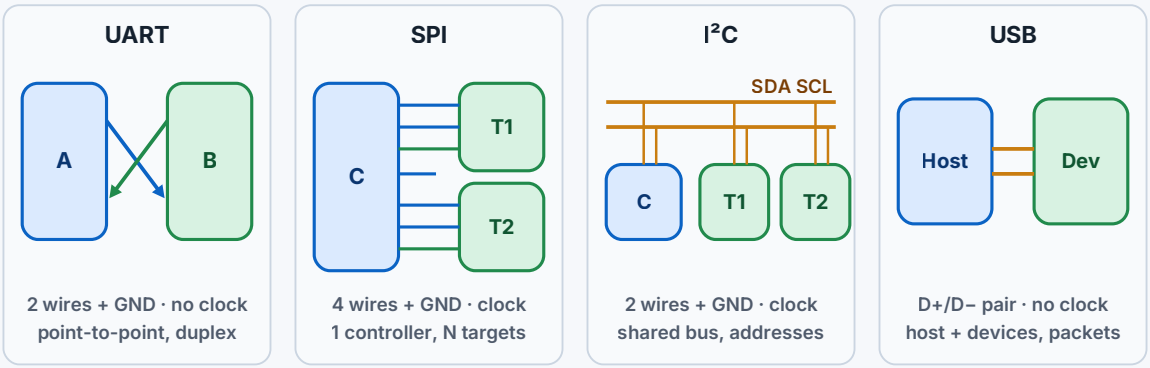


Figure 4.19 UART, SPI, I²C and USB side by side. UART is the only one with no clock and no shared bus: two devices, two crossed signal wires and a ground.

	UART	SPI	I²C	USB
Signal wires	2 (TX, RX) + GND	4 (clock, 2 data, select) + GND	2 (SDA, SCL) + GND	2 (D+, D−) + GND and power
Clock	none: each side has its own	from the controller	from the controller	recovered from the data
Topology	point-to-point	1 controller, many targets	shared bus with addresses	host, hubs and devices
Duplex	full	full	half	half
Typical speed	9600 to 115 200 baud (higher possible)	set by the controller; usually far above UART	100 kHz, 400 kHz, 1 MHz	12 Mbit/s or 480 Mbit/s (USB 2.0)
Typical reach	board or short cable; far with RS-485	on one board	on one board	cables up to a few metres
Learn it in	Book 2	Book 3	Book 4	Book 30

Choose a UART when two devices need a simple text or byte stream, when you want a console that works before anything else does, or when a module (GPS, radio, sensor) offers it. Choose SPI for speed on one board, I²C for many slow chips on two wires, and USB to talk to a PC or phone directly, faster and with power on the same cable.

4.17 In industry BASIC

Dossier parts P19 and P20 · Myths, mistakes and industry practice

The mistakes in this chapter (TX to TX, 5 V into 3.3 V, the wrong baud rate) are the ones engineers make most often, together with one myth: that 115 200 baud moves 115 200 bytes per second. It moves 11 520 bytes per second at 8N1, because every byte costs 10 bits. Here is how the UART is used at work.

In industry

- **Debug consoles.** Routers, set-top boxes, single-board computers and most embedded products have a UART console. It works before any network or display driver starts, so engineers use it to watch the bootloader and kernel, and to recover a product that will not boot.
- **Modules.** GPS, cellular, Wi-Fi and Bluetooth modules often take commands over a UART. Many use AT commands, a text command language that began with telephone modems (Book 2).
- **Bootloaders, test and industry.** Many microcontrollers can be programmed through a UART, factory test stations talk to products through one, and the same frames travel over RS-485 on long, noisy cables, for example in Modbus RTU networks (Books 2 and 10).
- **Tools.** Every logic analyzer decodes UART, and terminal programs such as `picocom`, `minicom` or PuTTY with a USB–UART cable are on every engineer's desk.

REAL-WORLD EXAMPLE

The Raspberry Pi console

On most Raspberry Pi models the primary UART uses GPIO 14 (TX) on header pin 8 and GPIO 15 (RX) on header pin 10. The pins work at 3.3 V, and the documentation warns that 5 V damages them. The Linux console runs at 115 200 baud. On the Raspberry Pi 5 the console moves to a separate three-pin header labelled UART. (Raspberry Pi documentation, checked November 2026.)

REAL-WORLD EXAMPLE

GPS modules and NMEA 0183

Most GPS receivers send their data as **NMEA 0183** sentences: lines of ASCII text that start with `$`, hold comma-separated fields, end with `*`, a two-digit hexadecimal checksum, and a carriage return and line feed. A sentence is at most 82 characters long, including `$` and the line ending. The NMEA standard rate is 4800 baud, 8N1; many modules, such as those built on u-blox 6-series receivers, start at 9600 baud, 8N1. The checksum is the XOR of every character between `$` and `*` (not including them), written as two hexadecimal digits. Lab 4.2 checks it on the classic example sentence:

OUTPUT nmea.txt

```
sentence : $GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47
XOR of the 61 characters between $ and * = 0x47
length with CR LF: 67 characters (limit 82)
time at 9600 baud 8N1: 69.79 ms
```

The first two letters after `$` name the talker (`GP` = GPS) and the next three the sentence type (`GGA` = position fix). Each character is one UART frame, so the receiver sees plain bytes; the NMEA layer sits on top. (gpsd "NMEA Revealed" and vendor documentation, checked November 2026.)

In the specification

There is no single standard for the UART character format. Its behaviour is defined by UART data sheets, such as National Semiconductor's PC16550D (1995), which this chapter follows. The electrical layers have their own standards: TIA-232-F (1997) for RS-232, and TIA-422 and TIA-485 for differential links. TIA standards are paid documents; Texas Instruments' design note SLLA037A gives a public summary of TIA-232-F. Book 2 lists every edition.

4.18 Hands-on labs BASIC

The three desk labs run on a normal computer, with no hardware. Lab 4.1 works at the signal level: Verilog transmitter → simulation → VCD file → sigrok decode → analyzer figure. Lab 4.2 computes every number in this chapter, and Lab 4.3 works at the software level.

HANDS-ON LAB 4.1

UART on the wire

GOAL

Simulate a transmitter, capture its TX line, decode it, and decode it again with wrong settings.

YOU NEED

Icarus Verilog 14 or later (OSS CAD Suite), sigrok-cli 0.7.2 with its `uart` decoder, Python 3.13, and the lab folder `labs/00/ch04/uart-on-the-wire/`.

STEPS

1. Read `uart_tx.v` and `tb_uart.v`: the testbench sends "Hi!" at 115 200 baud and saves only `tx`.
2. Run `bash labs/00/ch04/uart-on-the-wire/run.sh`. It simulates, cleans the VCD for sigrok, decodes it and runs the wrong-settings sweep. The decode step is:

SHELL run.sh

```
sigrok-cli -I vcd:downsample=1000 -i out/hi-sigrok.vcd \
  -P uart:tx=tx:baudrate=115200 -A uart=tx-data > out/decoded.txt
sigrok-cli -I vcd:downsample=1000 -i out/hi-sigrok.vcd \
  -P uart:tx=tx:baudrate=115200:format=ascii -A uart=tx-data \
  > out/decoded-ascii.txt
sigrok-cli -I vcd:downsample=1000 -i out/hi-sigrok.vcd \
  -P uart:tx=tx:baudrate=115200 -A uart \
  --protocol-decoder-samplenum > out/annotations.txt
```

3. In `out/annotations.txt`, find the start bit, data bits and stop bit of 'H'; in `out/baud-sweep.txt`, find where decoding starts to fail.

EXPECTED RESULT

The simulator log:

OUTPUT sim.log

```
VCD info: dumpfile out/hi.vcd opened for output.
TB: sent 0x48 'H' (done at 106.91 us)
TB: sent 0x69 'i' (done at 193.77 us)
TB: sent 0x21 '!' (done at 280.63 us)
PASS: testbench finished at 300.63 us
tb_uart.v:65: $finish called at 300630000 (1ps)
```

The decoded bytes are those of the "See it on the wire" box, and the script ends with a `PASS` line.

IF IT DOESN'T WORK

- sigrok prints nothing: libsigrok 0.5.2 stops at Icarus's `$comment` and `$dumpall` sections; pass the VCD through `vcd-for-sigrok` first.
- Garbage: check `-I vcd:downsample=1000` (the VCD unit is 1 ps, so this gives 1 sample per ns).
- No stop bits: this decoder version files them under "parity OK", so `-A uart=tx-stop` shows nothing; use `-A uart`.

TRY THIS NEXT

Send your own name at 9600 baud; then add a parity bit to `uart_tx.v` (an 11-bit frame register).

HANDS-ON LAB 4.2

UART timing maths

GOAL

Compute every timing number in this chapter and check the formulas against datasheet tables.

YOU NEED

Python 3.13 and the lab folder `labs/00/ch04/uart-timing-maths/`.

STEPS

1. Read `uart_maths.py`: each numbered section makes one table, and its comments give the formula.
2. Run `bash labs/00/ch04/uart-timing-maths/run.sh`.

EXPECTED RESULT

The tables shown in this chapter, in `out/`, then eleven checks against published numbers, ending:

OUTPUT checks.txt

```
ok   50 MHz 115200 count (uart_tx.v: 434): 434 (expected 434)
ok   even parity of 'H': 0 (expected 0)
ok   NMEA example checksum (NMEA Revealed: 47): 47 (expected 47)
```

IF IT DOESN'T WORK

- A check says `BAD`: you changed a formula. The expected values come from data sheets, so fix the formula, not the expected value.
- The divisor looks one too high: some chips store $N - 1$ (AVR), others N (16550).

TRY THIS NEXT

Add your board's clock to `pairs` and find the best oversampling ratio for 230 400 baud.

HANDS-ON LAB 4.3

Two programs on a virtual serial cable

GOAL

Run a host and a module over a pseudo-terminal "cable" and see what a pty can and cannot show.

YOU NEED

Python 3.13 with pyserial 3.5 on Linux, and the lab folder `labs/00/ch04/uart-pty-loopback/`.

STEPS

1. Read `loopback.py`: `VirtualCable` joins two pty pairs with a relay thread, like a null-modem cable, and `stored()` reads back the settings the kernel really kept.
2. Run `bash labs/00/ch04/uart-pty-loopback/run.sh`.

EXPECTED RESULT

Four parts: an AT exchange, the timing test and settings mismatch of Section 4.13, and XON/XOFF, then a `PASS` line for its ten checks.

IF IT DOESN'T WORK

- `Invalid argument` when you reopen a used pty with new settings: open a fresh pty pair, as the lab does.
- The program hangs: a read without `timeout=` waits for ever.

TRY THIS NEXT

Add a command `AT+NAME?` to the pretend module, and make the host retry once after a 0.5 s timeout.

On your bench · not run by the author **Read a real console**

Use a 3.3 V USB–UART adapter and any board with a UART console (for example a Raspberry Pi, or an ESP32, Pico or Nucleo board running a "hello" program at 115 200 baud).

Adapter pin	Board pin
GND	GND
RX	the board's TX
TX	the board's RX (optional: only to type commands)

1. Set the adapter to 3.3 V (never 5 V on 3.3 V pins). Connect GND first; touch grounded metal before handling the board, against static.
2. Open a terminal at 115 200 baud, 8N1 (`picocom -b 115200 /dev/ttyUSB0`) and reset the board: you should see its boot text. At 57 600 baud it should turn into garbage, as in Lab 4.1.
3. A logic analyzer on the board's TX pin, decoding `uart` at 115 200, should show the same bytes.

4.19 Where to go next BASIC

You have now seen one protocol taught through the whole dossier, and every protocol in the library follows the same path. For the UART itself, **Book 2** goes much deeper: RS-232 connectors and handshake lines, RS-422 and RS-485 multidrop networks with termination and biasing, a full UART receiver in Verilog with a cocotb testbench, autobaud, 9-bit modes, USB–UART bridges, and the protocols that ride on UART frames, such as AT commands and DMX512. The next chapter of this guide, Chapter 5, shows how to read the data sheets and specifications that this chapter's numbers came from.

4.20 Cheat sheet BASIC

Dossier part P22 · Cheat sheet

CHEAT SHEET

UART on one page

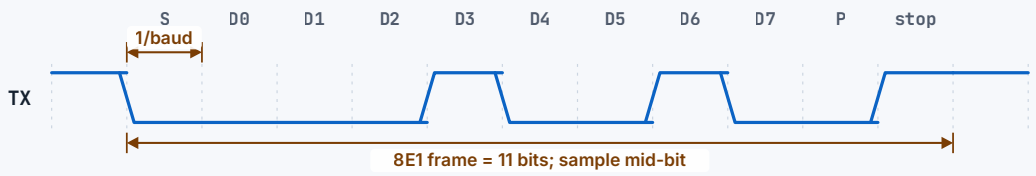


Figure 4.20 The UART frame (here 8E1, the letter 'H'): idle high, start 0, data LSB first, parity, stop 1. One bit lasts 1/baud, and the receiver samples in the middle of each bit.

Topic	Rule or number
Wires	TX → RX, RX ← TX, GND; RTS/CTS optional; one transmitter per wire
Levels	logic level idles high (1 = mark); RS-232 inverted: 1 = −3 to −15 V, 0 = +3 to +15 V
Frame	start (0) · 5–8 data bits, LSB first · optional parity · 1, 1.5 or 2 stop bits (1)
Names	8N1 = 8 data, no parity, 1 stop (10 bits); 7E1 (10 bits); 8E1 (11); 8O2 (12)
Timing	T_{bit} = 1/baud: 104.17 µs at 9600, 8.68 µs at 115 200; 8N1 byte = 10 bits
Throughput	efficiency = data bits / frame bits (8N1 = 80 %); 115 200 8N1 = 11 520 bytes/s
Sampling	16 ticks per bit; edge → 8 ticks → check start → every 16 ticks; D0 at 1.5 bits
Clock budget	stop sample at 9.5 bits: ideal 5.26 %, 4.6 % with 16× sampling (total); keep each side ≤ ±2 %
Divisor	N = f_{clk} / (16 × baud), rounded; check the error (16 MHz → 115 200: −3.55 %)
Errors	parity (PE), framing: stop bit 0 (FE), overrun: byte lost (OE), break: low > 1 frame (BI)
Flow control	RTS/CTS wires; XON 0x11 / XOFF 0x13 in-band
Debug first	measure the bit width; cross TX/RX; join GND; match baud, format and levels

CHAPTER AT A GLANCE

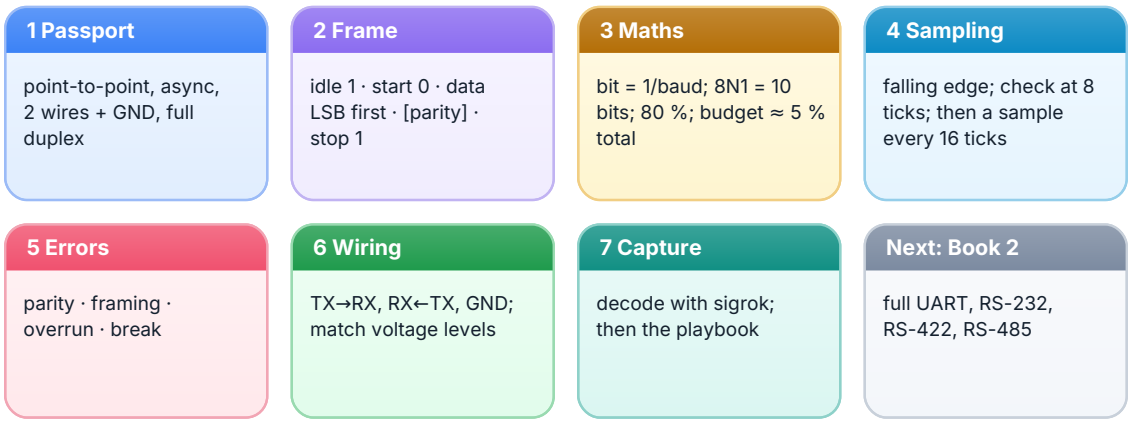


Figure 4.21 This chapter at a glance: the UART's passport, frame, maths, sampling, errors, wiring and capture, and where to go next.

REVIEW QUESTIONS

Recall

1. Give eight entries of the UART Protocol Passport, each in a few words.
2. A receiver uses 16× oversampling. Why does it wait 8 ticks after the falling edge, check the line, and only then start counting in steps of 16?
3. Why does every UART frame start with a 0 bit and end with a 1 bit?

Apply

4. Write the line levels, in time order, for the character '!' (0x21) in 7E1 and in 8O2.
5. A link runs at 57 600 baud, 8E1. Find the bit time, the frame time, the efficiency and the largest number of characters per second.
6. A microcontroller has a 12 MHz clock. With 16× oversampling, what divisor and error do you get for 115 200 baud and for 9600 baud? Would 8× oversampling help at 115 200?
7. In Lab 4.1 the decoder still read "Hi!" with its baud rate 5.5 % too high, but failed at 5.5 % too low. Use the stop-bit sample rule to explain why.

Debug or design

8. An analyzer set to 115 200 8E1 decodes `48 5A` with parity and framing errors on a link that you expected to read "Hi!". What is wrong, and how do you confirm it?
9. You must connect a 5 V microcontroller board to a 3.3 V GPS module that sends NMEA at 9600 baud. Draw the wiring and say what each connection needs.
10. A logic analyzer shows `AB A9` with no errors where you expected `48 69` ("Hi"). The baud rate is right. What is the most likely fault?

Answers: Appendix B.

Interview

1. Why is a UART called asynchronous if both sides need the same baud rate? No clock signal is sent. Each side times bits with its own clock, and the receiver re-synchronises on each start bit. The clocks only need to agree to within a few per cent for the length of one frame.

2. What happens if the two baud rates differ by 3 %? By 6 %? For 8N1, the stop-bit sample sits 9.5 bits after the start edge. A 3 % error moves it about 0.29 bit, still inside the stop bit, so it usually works, though with little margin on a poor link. A 6 % error moves it about 0.57 bit, past the end of the stop bit, so you get framing errors and wrong bytes.

3. Why do UART receivers oversample, often at 16×? To find the start edge within 1/16 of a bit, to place each sample near the middle of its bit, and to take several samples (majority vote) to reject noise.

4. How would you debug a UART that prints garbage? Measure the shortest pulse width on a scope or analyzer to find the real baud rate; check the format (data bits, parity, stop bits); check for an inverted RS-232 level; then check the divisor error of both clocks.

5. When would you choose a UART over SPI or I²C? For a link between two devices that may be on different boards or have different clocks, for a text console or a module interface, or when you need a link that is easy to watch with a terminal. SPI is faster on one board; I²C connects many slow chips on two shared wires.

KEY TAKEAWAYS

- A UART sends bytes on one wire per direction with no clock wire; the two devices agree on the baud rate and format in advance.
- Wiring is TX → RX, RX ← TX and a common ground; RTS/CTS are optional flow-control wires.
- A logic-level UART idles high (1, mark); RS-232 inverts the levels and uses ±3 to ±15 V at the receiver.
- A frame is a start bit (0), 5 to 8 data bits sent LSB first, an optional parity bit, and 1 or 2 stop bits (1); 8N1 is 10 bits long.
- Bit time = 1 / baud: 104.17 µs at 9600 and 8.68 µs at 115 200; 8N1 is 80 % efficient, so 115 200 baud carries 11 520 bytes/s.
- The receiver finds the falling edge, waits half a bit, checks the start bit, then samples every bit in its middle, usually with 16× oversampling.
- The stop-bit sample is 9.5 bits after the edge, so the total clock mismatch must stay below about 5 %; keep each side within about ±2 % and check the divisor rounding error.
- Errors: parity (wrong count of 1s), framing (stop bit 0), overrun (byte lost because software was slow) and break (line low for more than a frame).
- Debug by measuring the bit width, then checking format, level inversion, wiring and ground; a pseudo-terminal tests serial programs but never bit timing.

CHAPTER 5

Reading Specifications and Datasheets

WHERE WE ARE Signal | Line coding | Framing | Error control | Access | Network | Transport | Application | Measure & debug



Every number in the UART chapter came from a document. The 16-byte FIFO and the line control register came from a UART data sheet, the RS-232 voltages from a summary of the TIA-232-F standard, and the "±2 %" receiver limit from a microcontroller data sheet. Every book in this library rests on such documents, and as an engineer you will read them every week. This chapter shows who writes them, how to get them, how to find the right edition, how to read their strict language, and how to find one number in a datasheet and check it. Book 1, Chapter 17 takes the same skills further.

WARM-UP

1. In the ideal case, what total clock mismatch can an 8N1 UART receiver tolerate (Chapter 4)?
2. Which book is the home of TLS, and what does "home book" mean (Chapter 3)?
3. Is RS-232 a protocol or an electrical standard (Chapter 1)?

IN THIS CHAPTER

5.1 Tell a specification from a datasheet, and name the body that writes a given standard.

5.2 Say whether a specification is free, sold or for members first, and find its edition, amendments and errata.

5.3 Read normative language (shall, should, may, can, NOTE) and the structure of a standard.

5.4 Find your way around a datasheet: absolute maximum ratings, operating conditions, electrical and timing characteristics, register maps.

5.5 Find one number in a datasheet, check it against its formula and use it in a design.

5.1 The idea in plain words BASIC

Look at a phone charger. Its plug fits every USB-C phone, laptop and cable, because all of them follow the same rule book. Printed on the charger is a small label: 5 V, 3 A, 100–240 V input. That label is the maker's promise about this one product. You need both: the rule book to know what any charger must do, and the label to know what this one can do (Figure 5.1).

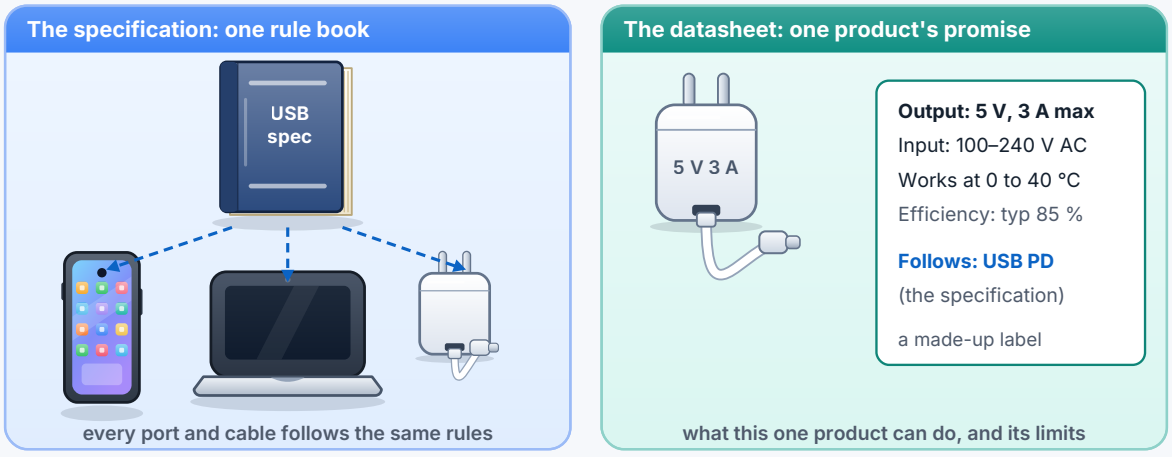


Figure 5.1 A specification is one rule book that every product of a kind follows; a datasheet is one maker's promise about one product, with its limits. The charger label here is made up.

Definition **Specification and datasheet**

A **specification** (or **standard**) is the agreed rule book for a protocol or interface, written by a standards body or industry group so that products from different makers work together. A **datasheet** (or data sheet) is one maker's document about one part: its pins, its guaranteed limits and how to use it.

5.2 Who writes the standards BASIC

A few kinds of organisation write almost every specification in this library (Figure 5.2).

- **Formal bodies. ISO** and **IEC** work through national members such as BIS (India), DIN (Germany) and ANSI (USA). The **ITU** writes telecom Recommendations (ITU-T) and radio rules (ITU-R).
- **IEEE and the IETF.** The **IEEE**, a professional association, writes Ethernet (802.3) and Wi-Fi (802.11). The **IETF** writes the Internet protocols, published as numbered **RFCs** (Requests for Comments).
- **3GPP** writes the mobile standards, GSM to 5G NR, for seven regional partners, among them ETSI (Europe) and TSDSI (India).
- **Industry groups. JEDEC** writes memory standards. Company groups, often called SIGs (special interest groups) or alliances, own many interfaces: **USB-IF**, **Bluetooth SIG**, **MIPI Alliance** and **PCI-SIG**. Trade associations such as **SAE** (vehicles) and **TIA** (RS-232) write their own.
- **Single companies.** The I^2C specification is NXP's user manual UM10204 (Philips, now NXP, created I^2C), and Arm publishes the AMBA buses.

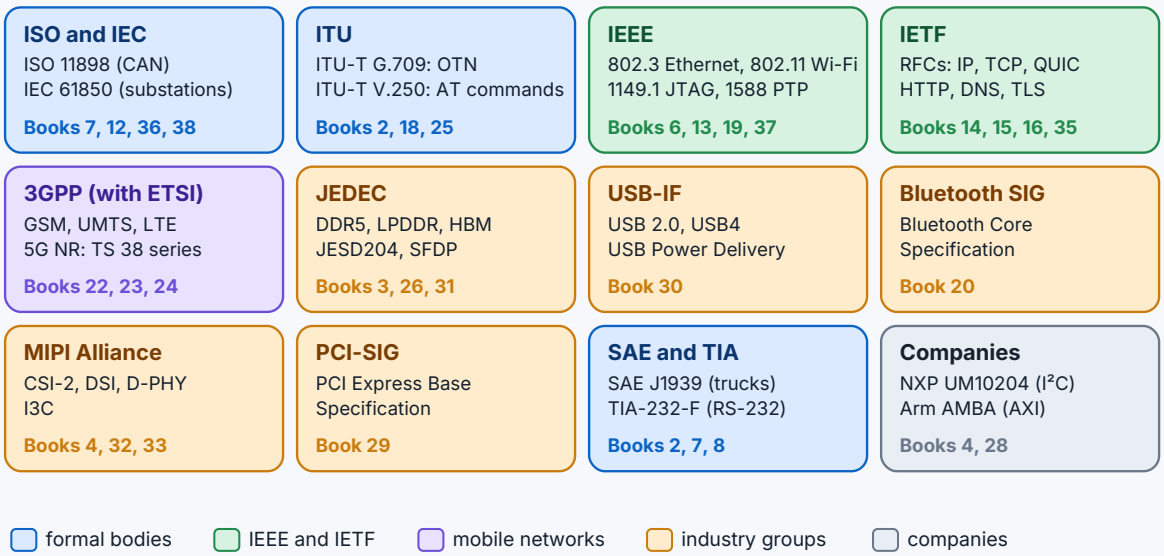


Figure 5.2 Who writes the specifications in this library, with two examples each and the books that teach them. The colour shows the kind of organisation.

5.3 Free, sold or members first BASIC

Before you can read a specification you must get it, and the rules differ (Figure 5.3; all checked November 2026).

- **Free:** RFCs, 3GPP specifications, ITU-T Recommendations, USB-IF specifications, NXP's UM10204, and IEEE 802 standards through the sponsored IEEE GET Program. The

Bluetooth Core Specification can be read free, but its notice says that non-members may not use it to build products.

- **Sold:** ISO and IEC standards (ISO 11898-1:2024 for CAN), SAE and TIA standards (J1939, TIA-232-F) and most other IEEE standards.
- **Members first:** PCI-SIG members read the PCI Express Base Specification free; others buy it. The full MIPI I3C specification is for members only, but a subset, MIPI I3C Basic, is public.

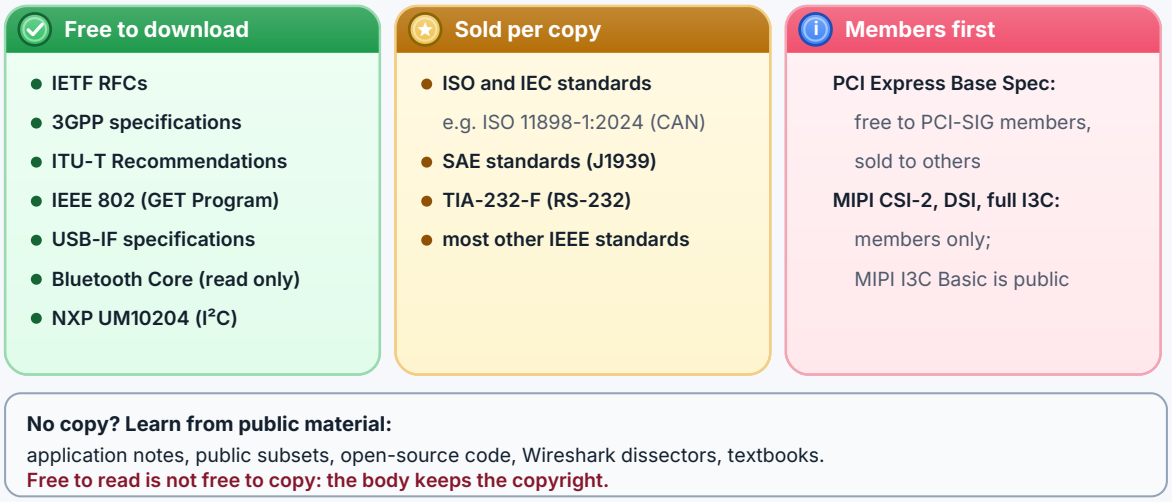


Figure 5.3 Three kinds of access. Free documents can be downloaded by anyone; sold ones cost money per copy; members-first documents go to members, and others buy them or use public material.

When a detail is not public, this library says so and teaches from public material; it never guesses. And free to read is not free to copy: the body keeps the copyright, so use your own words and cite the clause.

Check yourself

1. Name one specification that is free, one that is sold and one that is for members first.
2. You may download the Bluetooth Core Specification without paying. May you copy its figures into your product manual?

5.4 Editions, amendments and errata BASIC

Specifications change. Each document carries an identifier that tells you which version you hold (Figure 5.4).

- **RFCs never change.** A new version gets a new number, and its header says which RFCs it **obsoletes** (replaces) or **updates** (changes).
- **ISO and IEC** add the year after a colon: ISO 11898-1:2024 is part 1 of ISO 11898, published in 2024. An **amendment** adds or changes technical content; a **corrigendum** corrects errors; a new **edition** replaces the whole document.

- **IEEE** adds the year after a hyphen (IEEE Std 802.3-2022). Amendments get letters, such as IEEE 802.11ax-2021 (Wi-Fi 6), and are later merged into a new revision of the base standard.
- **3GPP** gives every specification a number and a version *x.y.z*. Once a specification is approved, *x* is its **release** (V18.y.z belongs to Release 18), *y* counts approved technical changes and *z* editorial fixes.
- **Datasheets** carry a revision: TI ends its literature number with a letter (SNOSD82**D** is revision D of the TMP117 data sheet, June 2018, revised September 2022), and a revision history lists every change.

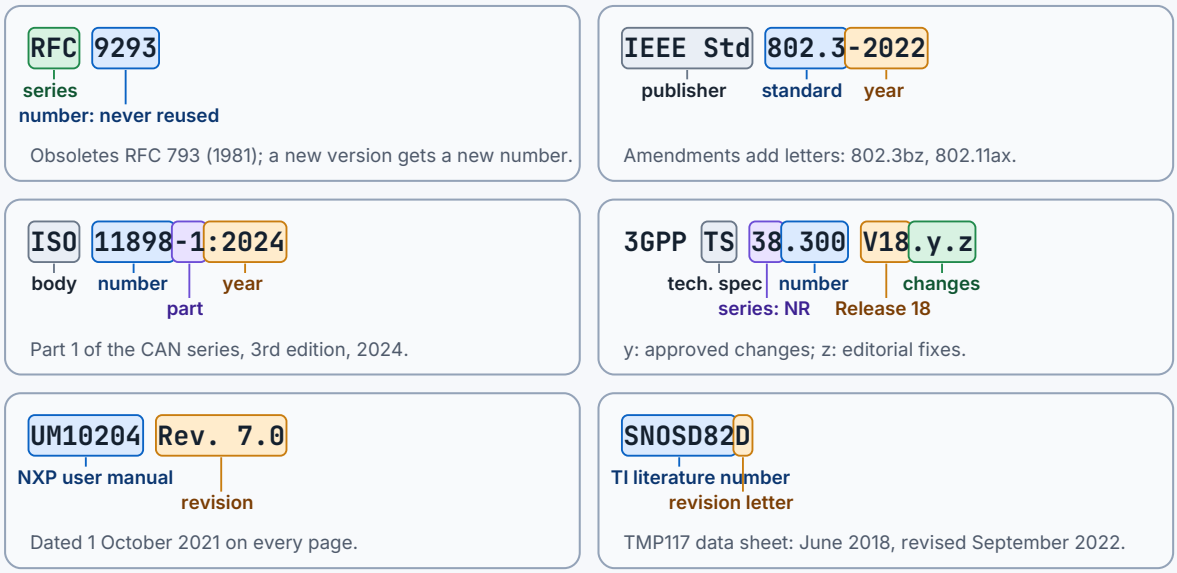


Figure 5.4 How to read six document identifiers. Each one names the publisher, the document, and its year, release or revision.

Even a good document has mistakes, so check its **errata**: published lists of corrections. The RFC Editor keeps errata for every RFC, each marked reported, verified, held for document update or rejected. The Bluetooth SIG publishes specification updates and lists which of them a product must include when it claims a given core version. Chip makers publish **errata sheets** (silicon errata) that list where the part differs from its own datasheet, with workarounds.

REAL-WORLD EXAMPLE

TCP: one protocol, two RFCs 41 years apart

TCP was specified in RFC 793 in September 1981, and many later RFCs fixed and extended it. In August 2022 RFC 9293 gathered them into one document, Internet Standard STD 7. Its header says it obsoletes RFC 793, 879, 2873, 6093, 6429, 6528 and 6691 and updates RFC 1011, 1122 and 5961. Anyone who still cites RFC 793 cites a replaced document. (RFC 9293 header, rfc-editor.org, checked November 2026.)

KEY IDEA

Quote every number with its document, edition and clause, and read the errata before you trust it. "Table 11 of UM10204 Rev. 7.0" can be checked; "the I²C spec says" cannot.

5.5 Normative language BASIC

A specification uses a few words with exact meanings (Figure 5.6). For Internet protocols they are defined by RFC 2119: **MUST** (or SHALL, REQUIRED) is an absolute requirement, **SHOULD** (RECOMMENDED) must be followed unless you have a valid reason and understand the consequences, and **MAY** (OPTIONAL) is truly optional. RFC 8174 added that the words have this meaning only when written in capitals: a lowercase "must" in a recent RFC is ordinary English. ISO, IEC and IEEE standards use lowercase **shall**, **should** and **may**, add **can** for what is possible, and keep **must** for external constraints such as laws of nature.

Figure 5.5 shows all of this on one page of a made-up standard.

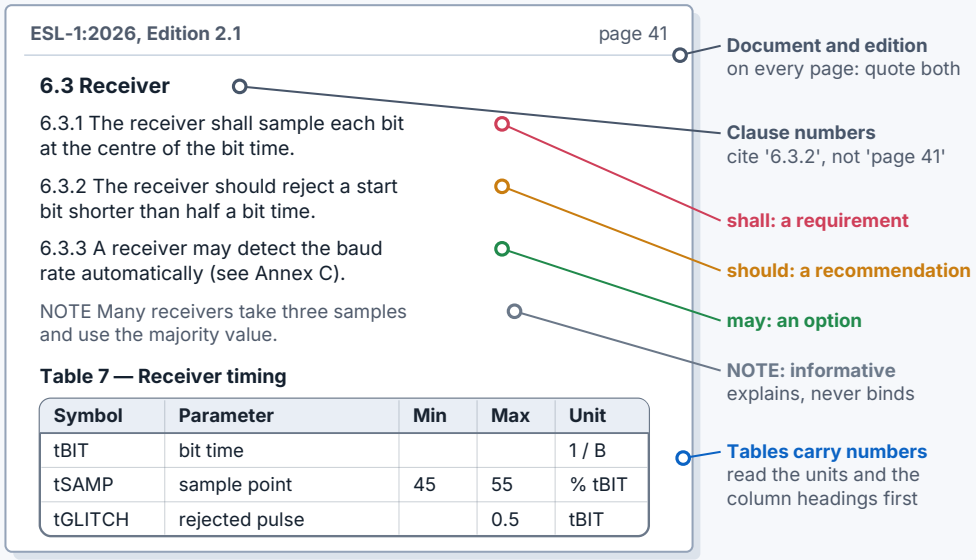
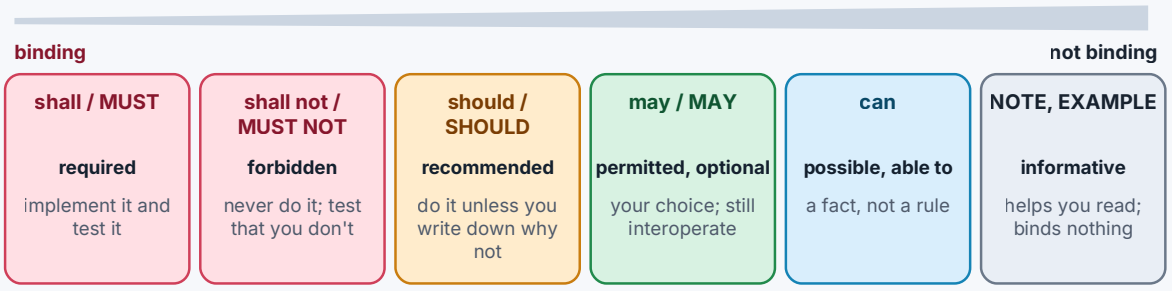


Figure 5.5 A mock page of a made-up standard. Every page names the document and its edition; clauses are numbered so that you can cite them; shall, should and may set the strength of each rule; a NOTE explains but binds nothing; tables carry the numbers.



IETF RFCs: the capital words of RFC 2119, special only in capitals (RFC 8174).
ISO, IEC and IEEE: shall, should, may, can; must only for external constraints.

Figure 5.6 The normative words, from binding to not binding, with what each one asks of an implementer. IETF RFCs write them in capitals; ISO, IEC and IEEE in lowercase.

Definition **Normative and informative**

Normative text is binding: the requirements you must meet to conform. **Informative** text only helps you understand: introductions, notes, examples and informative annexes. A NOTE never contains a requirement.

Check yourself

1. A recent RFC says "the client must close the connection" in lowercase. Is that a requirement in the RFC 2119 sense?
2. On the mock page, which of the three clauses must every receiver implement?

5.6 How a specification is organised BASIC

Most standards follow the same order: front matter (title, number, edition, foreword), the **scope** (what the document covers), **normative references** (other documents you must also follow), **terms and definitions**, the **technical clauses** (formats, state machines, timing), a **conformance** clause (what an implementation must do), **annexes** (each marked normative or informative) and a bibliography. You rarely read a 1,000-page specification from the start. Read the scope and the conformance clause first, then the terms, then only the clauses for your feature, and search for the exact term or symbol.

Tip **When a cross-reference does not match**

Documents are edited over many revisions, and a cross-reference can lag behind. In UM10204 Rev. 7.0, §7.1 says the maximum rise time comes "from Table 10", but in that revision the rise time t_r is in Table 11, the bus-line table. Search for the symbol, and cite the table you actually used.

5.7 How a datasheet is laid out BASIC

Datasheets follow a common plan too. Figure 5.7 shows the order Texas Instruments uses, for example in the TMP117 temperature sensor data sheet. The **Specifications** section holds the absolute maximum ratings, ESD ratings, recommended operating conditions, thermal information, electrical characteristics, timing and typical graphs; the **Detailed Description** holds the block diagram, modes, programming and register map. Other makers use similar sections.

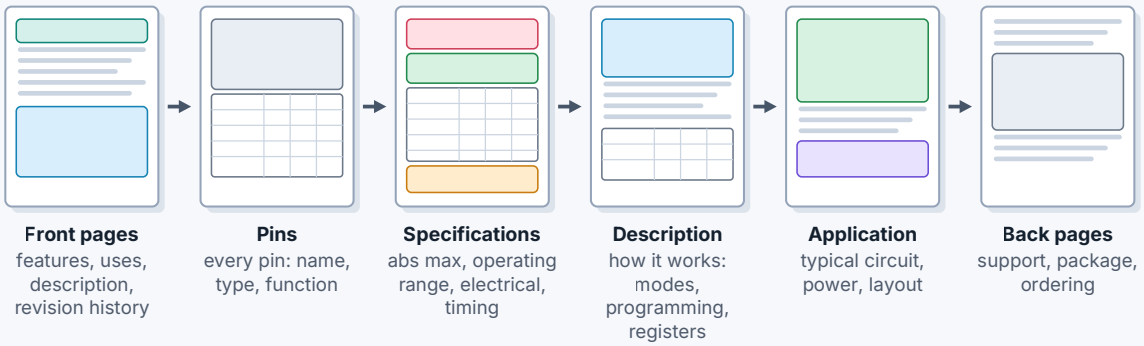


Figure 5.7 How a datasheet is laid out, from the front page to the back pages. Most design questions are answered in the specifications section and the detailed description.

A microcontroller usually has several documents. The **datasheet** gives pins, electrical limits and packages; a **reference manual** describes every peripheral and register (for the STM32F1 series, ST's RM0008); an **errata sheet** lists silicon bugs; **application notes** show how to use a feature. Small 8-bit parts such as the AVR family put everything in one long data sheet.

5.8 Three kinds of limits BASIC

Three tables in the specifications section look alike but mean very different things. Here they are for the TMP117's supply voltage V+ (Figure 5.8).

- **Absolute maximum ratings** are stress limits: −0.3 V to 6 V. Beyond them the part may be damaged. The data sheet's note says they are stress ratings only, that the part is not promised to work at them, and that long exposure may affect reliability.
- **Recommended operating conditions** are the range in which the part is meant to run: 1.8 V to 5.5 V from −55 °C to 150 °C (1.7 V to 5.5 V up to 70 °C), nominal 3.3 V.
- **Electrical characteristics** are the guaranteed performance, such as accuracy, currents and thresholds, and they hold only under the operating conditions in the table's header.

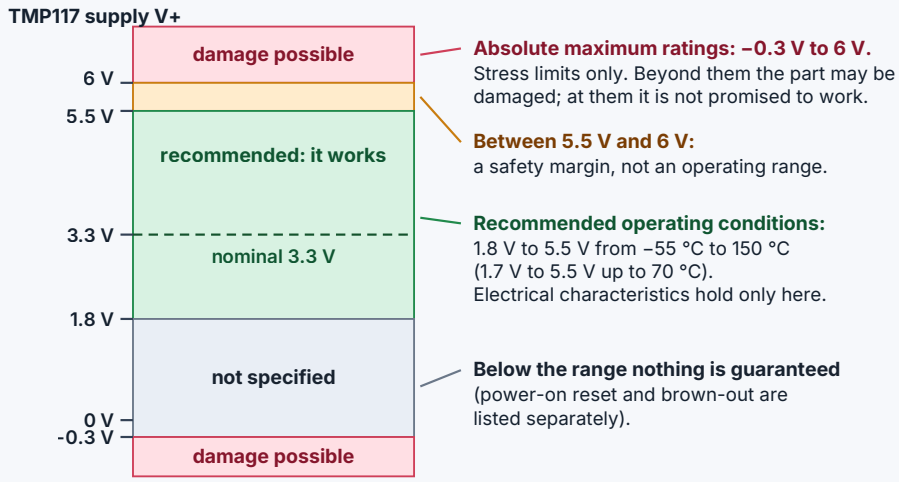


Figure 5.8 The three kinds of limits, drawn for the TMP117 supply. Damage is possible above 6 V; between 5.5 V and 6 V nothing breaks at once but nothing is promised; the part is guaranteed to work only from 1.8 V to 5.5 V.

KEY IDEA

Absolute maximum ratings say what can break the part. Recommended operating conditions say where it works. Electrical characteristics are guaranteed only inside those conditions.

Common mistake **Designing to the absolute maximum**

Wrong: "Its absolute maximum is 6 V, so the sensor is fine on our 5.8 V rail."

Why: 5.8 V is above the recommended maximum of 5.5 V. No characteristic is guaranteed there, and running close to the stress limit can shorten the part's life.

Right: Keep every supply and pin inside the recommended operating conditions, with margin for tolerance and ripple; use the absolute maximum only to judge faults and transients.

5.9 Timing and characteristics tables BASIC

Electrical and timing tables share one layout (Figure 5.9): a symbol, a parameter, the test conditions, then **min**, **typ** and **max** columns and a unit. Footnotes say how a limit is guaranteed; "guaranteed by design" means the maker does not test it on every part.

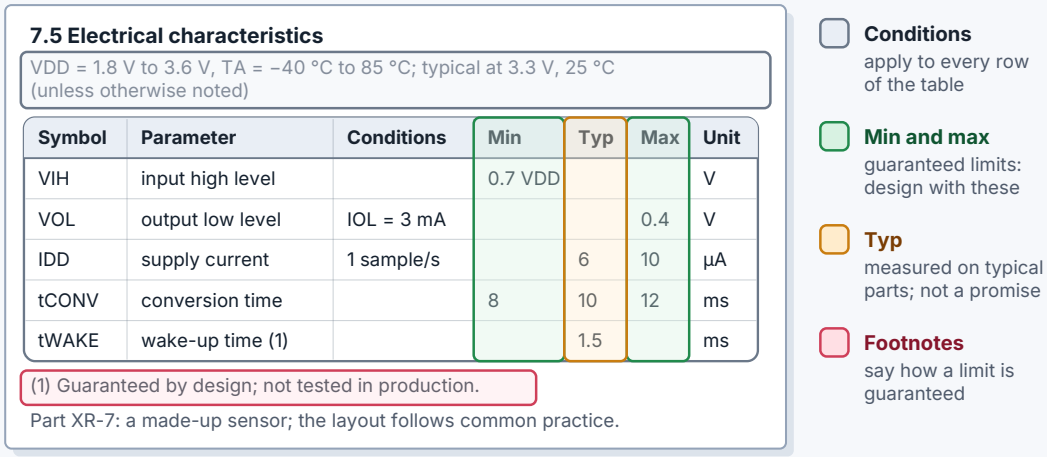


Figure 5.9 A mock electrical-characteristics table for a made-up part. The conditions at the top apply to every row; min and max are guarantees; typ is information only; footnotes change how far you can trust a row.

Min and max are the guarantees; **typ** is the value measured on typical parts at typical conditions, and is never a promise. The TMP117 lists its one-shot conversion time as min 13 ms, typ 15.5 ms and max 17.5 ms. Firmware must wait at least 17.5 ms before it reads the result, not 15.5 ms.

Timing diagrams name each parameter as the time between two edges, measured at stated voltage levels; I²C, for example, measures at 30 % and 70 % of VDD. Book 1, Chapter 17 teaches how to read timing diagrams and check a real capture against a table.

Some numbers come from a formula next to the table. UM10204 sizes the I²C pull-up resistor from the rise-time limit and the bus capacitance (§7.1):

$$R_{p(max)} = t_r / (0.8473 \times C_b) \qquad R_{p(min)} = (V_{DD} - V_{OL(max)}) / I_{OL}$$

where:

- R_p is the pull-up resistance, in ohms;
- t_r is the maximum rise time from 30 % to 70 % of VDD, in seconds (300 ns in Fast-mode, Table 11);
- 0.8473 is ln(0.7 / 0.3), the number of RC time constants between 30 % and 70 %;
- C_b is the bus capacitance, in farads (at most 400 pF in Fast-mode);
- V_{DD} is the supply, $V_{OL(max)}$ = 0.4 V the largest low level and I_{OL} = 3 mA the sink current a Fast-mode device must manage (Table 10).

For a Fast-mode bus with 200 pF at 3.3 V, Lab 5.1 gives $R_{p(max)}$ = 300 ns / (0.8473 × 200 pF) = 1.77 kΩ and $R_{p(min)}$ = 2.9 V / 3 mA = 967 Ω. So any standard value between them, such as 1.2 kΩ or 1.5 kΩ, meets both the rise-time and the sink-current limit.

5.10 Worked example: finding one number INTERMEDIATE

Chapter 4 quoted a ±2 % receiver limit from an AVR data sheet. Here is how to find that number, read it correctly and check it, in five steps: search, pick the row, read the columns, compare, and recompute.

The ATmega324PB data sheet gives the receiver's operating range as two ratios, from a formula in §23.9.3:

$$R_{slow} = (D+1)\,S \,/\, (S-1+D\cdot S+S_F) \qquad R_{fast} = (D+2)\,S \,/\, ((D+1)\,S+S_M)$$

where:

- R_{slow} and R_{fast} are the slowest and fastest incoming data rates the receiver accepts, as a ratio to its own baud rate (no unit; printed in %);
- D is the number of data bits plus parity bits (5 to 10);
- S is the number of samples per bit: 16 in normal-speed mode, 8 in double-speed mode;
- S_F is the first sample used for the majority vote (8 normal, 4 double) and S_M the middle one (9 normal, 5 double).

WORKED EXAMPLE 5.1

Is 115 200 baud safe on a 16 MHz AVR?

Problem. An ATmega324PB runs from a 16 MHz crystal in normal-speed mode (U2X = 0) and must receive 115 200 baud, 8N1, from a PC's USB–UART adapter. Is that within the data sheet's limits?

Step 1 — find the section. Search the PDF for "error". The baud-rate table in §23.11 gives UBRR = 8 with an error of −3.5 %, and §23.9.3, *Asynchronous Operational Range*, gives the limits.

Step 2 — pick the row. 8N1 has 8 data bits and no parity, so $D = 8$. Normal speed means Table 23-2, row $D = 8$.

Step 3 — read the columns. R_{slow} = 95.36 %, R_{fast} = 104.58 %, maximum total error +4.58/−4.54 %, recommended maximum receiver error ±2.0 %.

Step 4 — compare. The actual rate is 16 000 000 / (16 × 9) = 111 111 baud, an error of −3.55 %. That is outside the recommended ±2.0 %. If the PC side were exact, the incoming rate would be 103.68 % of the receiver's rate, still below R_{fast}, so the link would work, but with only 0.90 % left for every other error.

Step 5 — check the table. With $D = 8$, $S = 16$, $S_F = 8$ and $S_M = 9$: R_{slow} = 144 / 151 = 95.36 % and R_{fast} = 160 / 153 = 104.58 %. Both agree. But the negative total, 100 − 95.36, is 4.64 %, not the printed 4.54 %, while the other eleven rows agree with 100 − R_{slow}. Lab 5.1 recomputes every row:

OUTPUT avr-range.txt

```
AVR USART, normal (U2X=0): S = 16, SF = 8, SM = 9
    computed        | printed in the table        | check
 D  Rslow%  Rfast%  | Rslow  Rfast   total +/- % | 100-Rslow
--  ------  ------  | -----  ------  ----------- | ---------
 5   93.20  106.67  | 93.20  106.67  +6.67/-6.8  |   6.80
 6   94.12  105.79  | 94.12  105.79  +5.79/-5.88 |   5.88
 7   94.81  105.11  | 94.81  105.11  +5.11/-5.19 |   5.19
 8   95.36  104.58  | 95.36  104.58  +4.58/-4.54 |   4.64  <- differs
 9   95.81  104.14  | 95.81  104.14  +4.14/-4.19 |   4.19
10   96.17  103.78  | 96.17  103.78  +3.78/-3.83 |   3.83
```

Answer. No: −3.55 % is outside the recommended receiver error. Use a crystal that divides exactly, such as 18.432 MHz (UBRR = 9, 0.0 %), or a lower rate such as 38 400 baud (+0.16 %); Lab 5.1 checks both in `out/avr-check.txt`.

This tells us two things: read the row that matches your frame (*D* counts the parity bit too), and recompute any number your design depends on, because a single cell can be misprinted. When one is, design with the safer value, look for a newer revision or erratum, and tell the maker.

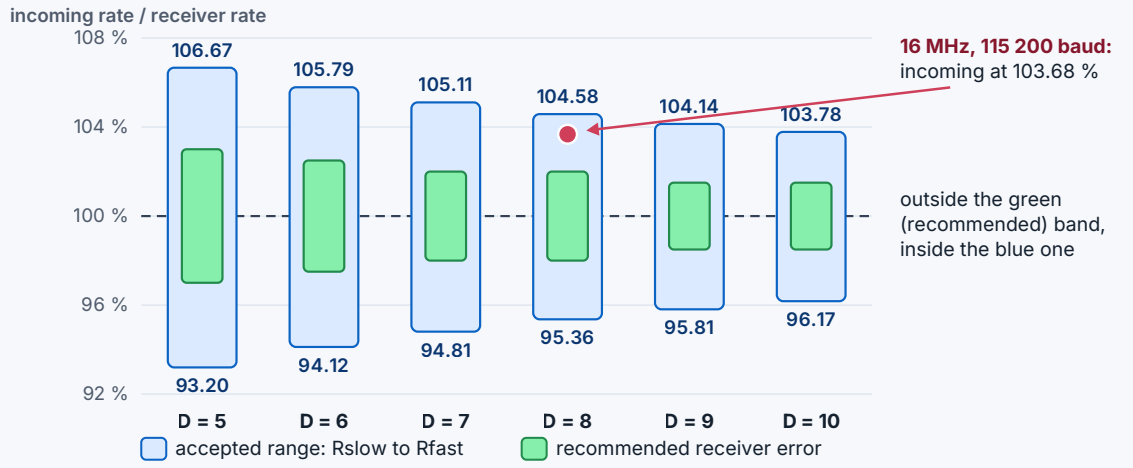


Figure 5.10 The receiver's operating range for each frame size, computed by Lab 5.1 from the data sheet's formula. Blue is everything the receiver accepts; green is the recommended error for the receiver's own clock. 16 MHz at 115 200 baud lands outside the green band, though still inside the blue one.

5.11 Register maps BASIC

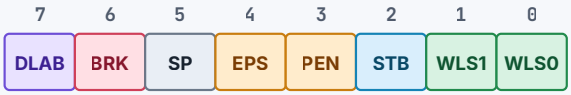
Software talks to a chip through its **registers**: small numbered storage places, each with bit fields that set a mode or report a status. A **register map** gives them in two levels. A summary lists each register's address (or offset), name, **access type** and **reset value**; then a field table describes every bit. The usual access codes are R (read only), W (write only), R/W (read and write) and RC (read to clear: reading it clears it). TI's data sheets write the access and reset value under each bit, so "R-0" means read-only with reset value 0. Bits marked reserved should be left as the data sheet says, usually written as 0.

Figure 5.11 shows a classic map: the 16550 UART from Chapter 4. Its eight addresses hold ten registers; the DLAB bit of the line control register (LCR) switches addresses 0 and 1 between the data registers and the baud-rate divisor latches. The LCR sets the frame format: bits 1–0 choose 5 to 8 data bits, bit 2 the stop bits, bits 3 and 4 the parity. So 8N1 is 0x03 and 7E1 is 0x1A.

16550 UART: register summary

Addr	DLAB	Name	Function
0	0	RBR / THR	receive / transmit buffer
1	0	IER	interrupt enable
2	x	IIR / FCR	interrupt ID / FIFO control
3	x	LCR	line control
4	x	MCR	modem control
5	x	LSR	line status
6	x	MSR	modem status
7	x	SCR	scratch
0	1	DLL	divisor latch, low byte
1	1	DLM	divisor latch, high byte

LCR (address 3): line control, R/W, reset 0x00



WLS1:0 word length: 00 = 5 ... 11 = 8 bits
STB: 1 = two stop bits
PEN: parity on
EPS: 1 = even parity
SP: stick parity
BRK: force a break
DLAB: 1 = divisor latches

Lab 5.1 (registers.txt):

8N1	0 0 0 0 0 0 1 1	0x03
7E1	0 0 0 1 1 0 1 0	0x1A

Figure 5.11 A real register map, redrawn: the 16550 UART's registers by address, and the bit fields of its line control register (LCR). The values for 8N1 and 7E1 come from Lab 5.1.

A data register needs its format too. The TMP117 keeps its result in a 16-bit register at address 00h, in two's complement, with 1 LSB = 7.8125 m°C. Its reset value, 8000h, reads as −256 °C until the first conversion is complete, so a reading of −256 °C means "no result yet", not a frozen sensor:

OUTPUT registers.txt

```
TMP117 Temp_Result (16-bit two's complement, 1 LSB = 7.8125 m°C)
raw     counts   temperature
------  ------   -----------
0x0C80    3200     25.0000 °C
0x0E80    3712     29.0000 °C
0x3200   12800    100.0000 °C
0xF380   -3200    -25.0000 °C
0xFFF0     -16     -0.1250 °C
0x8000  -32768   -256.0000 °C
0x8000 is the reset value: the register reads -256 °C
until the first conversion ends.
```

Check yourself

1. Which LCR value gives 8 data bits, odd parity and one stop bit?
2. A TMP117 returns 0xFFF0. What temperature is that?

5.12 A checklist for reading any spec BASIC

Use Figure 5.12 whenever you open a new specification or datasheet. It prevents the most common mistakes: the wrong edition, a missed erratum, a design built on a typical value, a wrong reset assumption.



Figure 5.12 Two checklists: one for a specification and one for a datasheet. Work through them in order before you design with a number.

5.13 Lab: recompute the data sheet BASIC

HANDS-ON LAB 5.1

Recompute the numbers in a data sheet

GOAL

Recompute the numbers this chapter reads from real data sheets, compare them with the published values, and find the cell that does not agree.

YOU NEED

Python 3.13 and the lab folder `labs/00/ch05/datasheet-numbers/`.

STEPS

1. Read `datasheet_numbers.py`. The block at the top holds the published numbers, each with its table; each numbered section computes one table and prints it.
2. Run `bash labs/00/ch05/datasheet-numbers/run.sh`.
3. Open `out/avr-range.txt` and find the row marked “differs”; open `out/i2c-pullup.txt` and find the largest pull-up for a 400 pF Fast-mode bus.

EXPECTED RESULT

Five tables in `out/`, then the checks against the published values, ending:

OUTPUT `checks.txt`

```
ok   Rp(min) 3.3 V, 3 mA in ohm: 967 (expected 967)
PASS: 51 checks match the published numbers
```

IF IT DOESN'T WORK

- A check says `BAD`: you changed a formula or a published value; the published numbers are the reference.
- A ratio looks 100 times too small: the script works in ratios and prints percentages.

TRY THIS NEXT

Add the double-speed case for 16 MHz at 57 600 baud and decide whether it meets Table 23-3.

CHAPTER AT A GLANCE

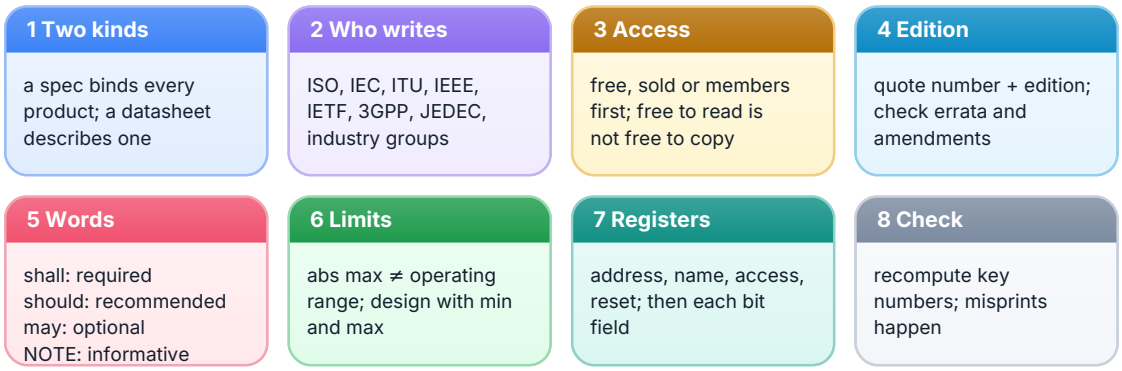


Figure 5.13 This chapter at a glance: specifications and datasheets, who writes them, how to get them, editions, the normative words, the three kinds of limits, register maps and checking a number.

REVIEW QUESTIONS

Recall

1. Which body writes each of these: Ethernet, CAN, TCP, 5G NR, DDR5, USB Power Delivery?
2. Give one specification that is free, one that is sold and one that is for members first. What does "free to read" still not allow you to do?
3. What do shall, should, may and can mean in an ISO or IEC standard? May a NOTE contain a requirement?

Apply

4. Decode these identifiers: ISO 11898-1:2024; 3GPP TS 38.331 V18.2.0; IEEE Std 802.11ax-2021; SNOSD82D.
5. Write the 16550 LCR value for 8E1 and for 7O2.

6. An ATmega324PB at 8 MHz must receive 38 400 baud, 8E1, in normal-speed mode. Which row of Table 23-2 applies, what UBRR and error do you get, and is it within the recommended limit?
7. A Fast-mode I^2C bus has 150 pF of capacitance and a 3.3 V supply. Find the largest and the smallest pull-up resistor.

Debug or design

8. A standard says: "6.2 The device should retry after 10 ms. NOTE: The device must not retry more than three times." What does a conforming device have to do?
9. Firmware waits 15.5 ms after starting a TMP117 one-shot conversion. Sometimes it reads the old value, and after power-up it reads −256 °C. Explain both and fix the firmware.

Answers: Appendix B.

Interview

1. How do you read the datasheet of a new part? Check the part number and revision; read the front page and pins; separate the absolute maximum ratings from the operating conditions; read each table with its conditions and footnotes, using min and max; then the registers and the errata sheet.

2. Absolute maximum ratings or recommended operating conditions? The first are stress limits beyond which the part may be damaged; the second are the range in which it is meant to run, where its characteristics are guaranteed.

3. A specification says SHOULD. Can you skip it? Only for a good reason that you have understood and written down. SHOULD is the recommended behaviour that others may rely on; it is not MAY.

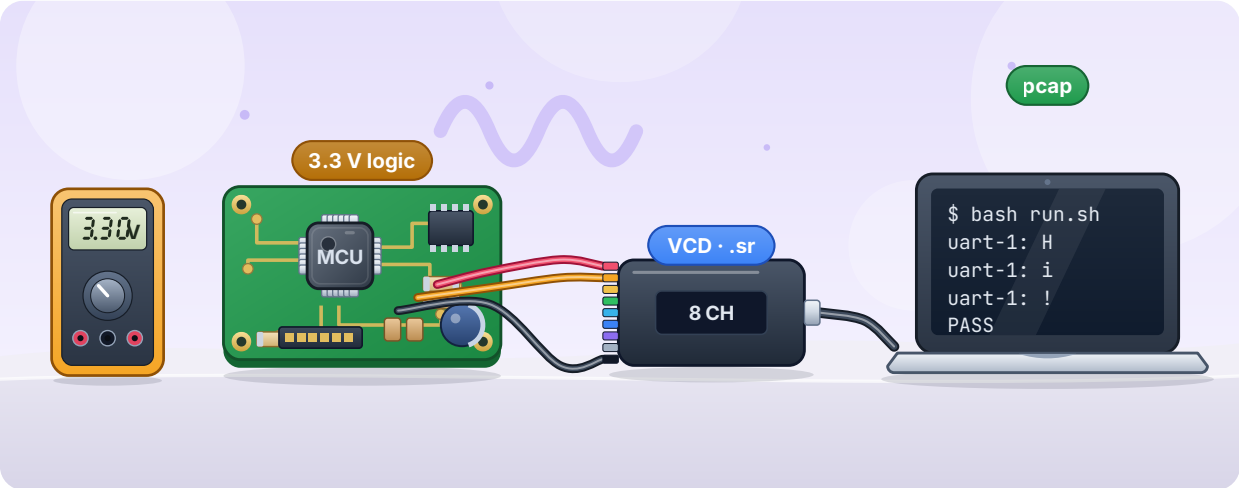
KEY TAKEAWAYS

- A specification is the rule book that every product of a kind follows; a datasheet is one maker's promise about one part.
- Standards come from ISO, IEC, ITU, IEEE, the IETF (RFCs), 3GPP, JEDEC and industry groups.
- Specifications are free (RFCs, 3GPP, ITU-T), sold (ISO, IEC) or members first (PCI-SIG, MIPI); free to read is never free to copy.
- Quote every number with its document, edition and clause; check errata and amendments.
- shall/MUST is required, should/SHOULD recommended, may/MAY optional; NOTEs are informative.
- Absolute maximum ratings are damage limits; characteristics hold only inside the recommended operating conditions. Design with min and max, never typ, and recompute key numbers.

CHAPTER 6

Your Protocol Bench

WHERE WE ARE: Signal | Line coding | Framing | Error control | Access | Network | Transport | Application | Measure & debug



In Chapter 4 a Verilog transmitter sent “Hi!”, and sigrok turned the captured wire back into the three letters. Every book in this library works like that: a free tool makes or captures a signal, and another free tool explains it. This chapter sets up that **bench** on your own computer. You install four free tools, check them, and make your first two captures: one of a wire and one of a network packet. Then you meet an optional low-cost hardware kit and its safety rules. Book 1, Chapter 21, teaches the instruments in depth; this chapter gets you set up and successful.

WARM-UP

1. In Chapter 4, what did sigrok’s `uart` decoder need to know before it could decode “Hi!”?
2. Why must two UART devices share a ground wire (Chapter 4)?
3. Which table of a datasheet gives the highest voltage a pin can survive (Chapter 5)?

IN THIS CHAPTER

6.1 Name the four free tools of the desk bench, say what each one does, and draw how they connect.

6.2 Install the bench on Debian or Ubuntu, or on Windows through WSL 2, and check that it works.

6.3 Run a simulated UART capture, decode it with sigrok, and read its VCD times and sample numbers by hand.

6.4 Build packets with scapy, dissect them with tshark, and count a packet's bytes layer by layer.

6.5 Choose a low-cost hardware kit and connect it safely: voltage levels, ground first, static, never mains.

6.6 Fix the common setup problems with the troubleshooting table.

6.1 The idea in plain words BASIC

Imagine a reporter at an interview in a language she does not speak. A microphone listens, a recorder keeps every word, and an interpreter translates; then the reporter writes what happened. A protocol bench does the same four jobs (Figure 6.1). A probe or a logic-analyzer lead listens to a wire. The samples, or the packets, go into a **capture file**. A **decoder** turns the recording into bits, bytes and named fields. And you explain what happened, and why.

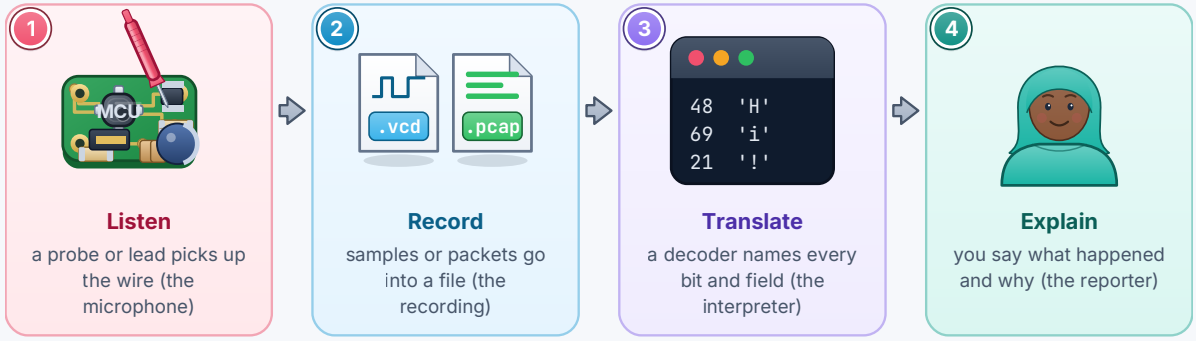


Figure 6.1 The four jobs of every protocol bench, with the reporter's tools in brackets. A lead listens to the wire, a file keeps the recording, a decoder names every field, and you explain the result.

The **desk bench** is the set of free programs this library uses on a normal computer, with no hardware, and a **desk lab** is a lab that runs on it. The author ran every desk lab in this guide, and the book prints their real output.

6.2 The four free tools BASIC

The desk bench has four tools (Figure 6.2). All are free and open source, and all have a command-line form, so a script can run them, check their output and save it for the book. **Python 3** runs the maths and builds packets; labs call it as `$EP_PYTHON`. **Icarus Verilog** is a **simulator**: it runs a hardware design written in Verilog with its **testbench**, a Verilog file that drives the design and prints messages (`iverilog` compiles, `vvp` runs). It comes in **OSS CAD Suite**, a free bundle of open-source hardware tools. **sigrok** (`sigrok-cli`, or the **PulseView** window) runs protocol decoders on logic captures. **Wireshark** (`tshark` on the command line) runs a **dissector** for each protocol layer of a packet.

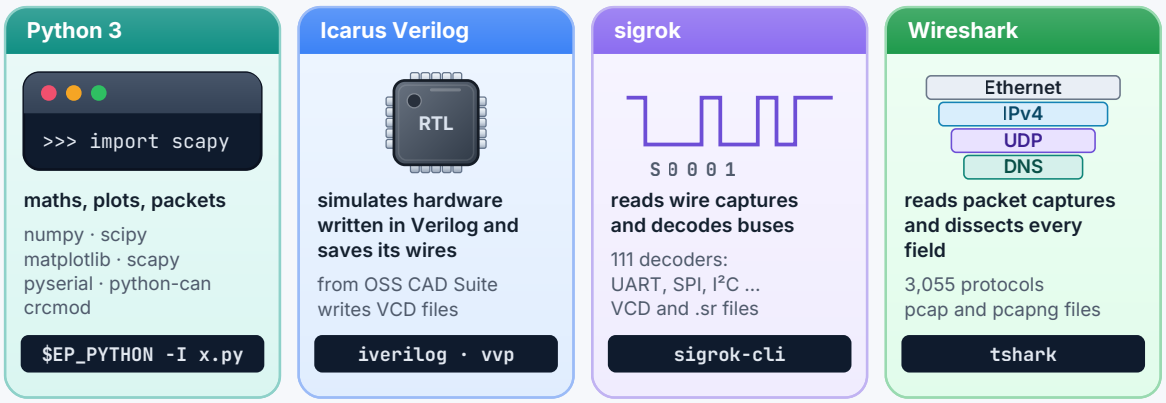


Figure 6.2 The four tools of the desk bench and the command you type for each. Python does the maths and builds packets, Icarus Verilog simulates hardware, sigrok decodes wires and Wireshark dissects packets. The numbers of decoders and protocols come from Lab 6.1.

Of the Python packages, numpy and scipy do the maths, matplotlib the plots, scapy builds packets, pyserial opens serial ports, python-can talks to CAN and crcmod computes CRCs. Later books add a few; each book's lab setup lists them with their versions.

Definition **VCD and pcap**

A VCD (value change dump) file is a text file of the time of every change of every signal. A **pcap** file is a binary file of network packets, each with a time stamp.

6.3 How the tools connect BASIC

The four tools form two pipelines (Figure 6.3). In the **signal pipeline**, Icarus simulates a design and writes a VCD file; a small helper, `vcd-for-sigrok`, removes two header sections that sigrok 0.5.2 cannot read; and `sigrok-cli` decodes the wire. In the **packet pipeline**, a Python script with scapy writes packets to a pcap file, and `tshark` dissects them.

Real hardware fits into the same pipelines: a real logic analyzer saves a sigrok session file (`.sr`), which sigrok decodes with the same command, and a real network capture is a pcap file.

KEY IDEA

A desk lab and a real bench differ only in where the samples or packets come from. The files (VCD, `.sr`, pcap) and the decoders are the same, so what you learn at your desk works on real hardware.

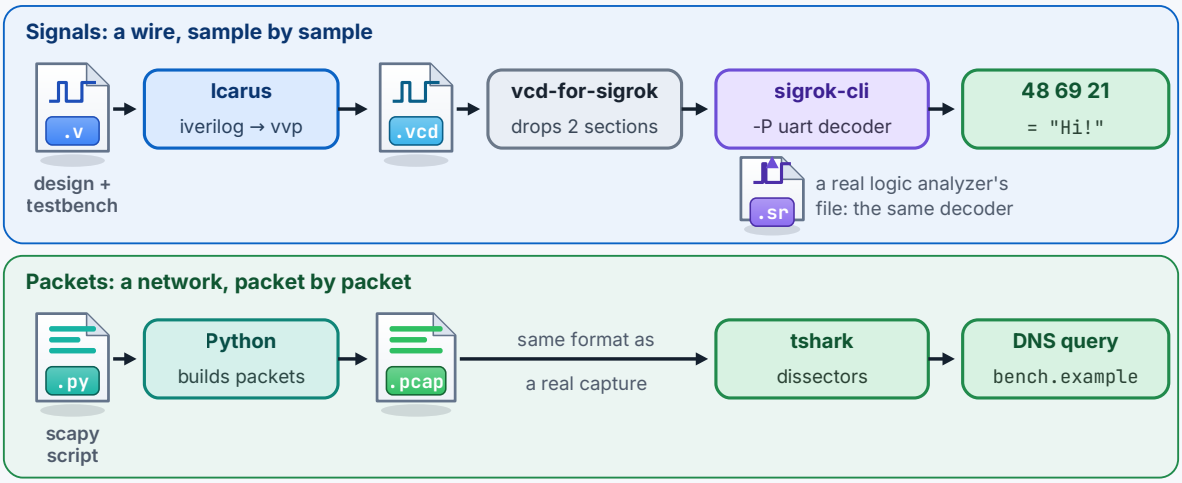


Figure 6.3 The two pipelines of the desk bench. Signals: Verilog → Icarus → VCD → vcd-for-sigrok → sigrok's decoder → bytes. Packets: a scapy script → pcap → tshark's dissectors → fields. A real logic analyzer's .sr file enters at the same decoder, and a real network capture is the same pcap format.

Check yourself

1. Which tool turns a simulated wire into bytes, and which turns a packet into fields?
2. Why does the signal pipeline need `vcd-for-sigrok`?

6.4 Install the bench BASIC

You need a 64-bit x86 computer, about 4 GB of free disk space, and internet for the installation (the labs never use the network). Linux is the tested route; Windows runs the same tools through WSL 2 (Figure 6.4). Each book's lab setup lists the versions it was tested with.

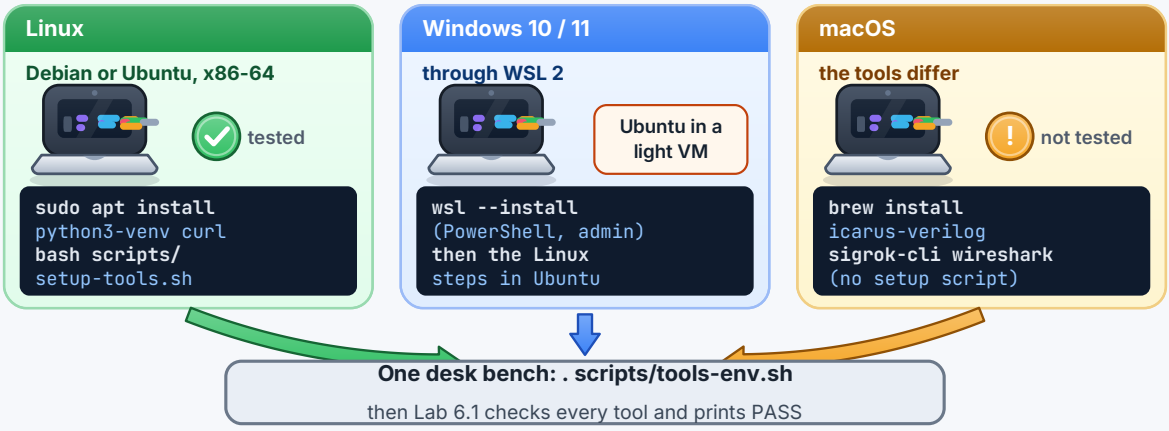


Figure 6.4 Three ways to the same desk bench. Linux (Debian or Ubuntu) is the tested route; Windows runs Ubuntu in a light virtual machine through WSL 2 and then follows the Linux steps; on macOS the tools have other names and the setup script does not run.

On Linux: Debian or Ubuntu

The **code bundle** that comes with this guide holds the folders `labs/` and `scripts/`. Unpack it in your home folder; here we call its top folder `bundle`. First install two small system packages: the module that makes Python **virtual environments** (a private Python with its own packages), and `curl` for downloads. Then, inside the bundle, run the series' setup script once, make the virtual environment `.venv`, and install the packages into it:

```shell
sudo apt update && sudo apt install python3-venv curl
cd ~/bundle                       # the code bundle's top folder
bash scripts/setup-tools.sh       # OSS CAD Suite, sigrok-cli, tshark -> .tools/
python3 -m venv .venv             # a private Python for the labs
.venv/bin/pip install numpy scipy matplotlib scapy pyserial python-can crcmod
```

`setup-tools.sh` needs no administrator rights. It downloads the pinned OSS CAD Suite release (about 750 MB; 2.5 GB unpacked), fetches the `sigrok-cli` and `tshark` packages with `apt-get download`, unpacks them into `.tools/`, and writes wrappers and the `vcd-for-sigrok` helper into `.tools/bin`. Run it **before** you install `sigrok-cli` or `tshark` as system packages: it installs only what your system lacks. If the download stops (or the disk is full), delete the partial file in `.tools/dl/` and run the script again. Then, in **every new terminal**, load the bench and check it with Lab 6.1:

```shell
. scripts/tools-env.sh                  # note the dot: it changes this shell
bash labs/00/ch06/bench-check/run.sh    # Lab 6.1: check the bench
```

The dot means "read this file into the current shell". `tools-env.sh` puts `.tools/bin` and the OSS CAD Suite folder first on the shell's search path (`PATH`) and sets `EP_PYTHON` to `.venv/bin/python`; run with `bash` instead, it would change only a short-lived child shell.

On Windows: WSL 2

WSL (Windows Subsystem for Linux) runs a real Linux system inside Windows; version 2 runs it in a light virtual machine with a Linux kernel. Open PowerShell as administrator, run one command, and restart:

```powershell
wsl --install
```

It turns on WSL and installs Ubuntu, set to WSL 2; it needs Windows 10 version 2004 (build 19041) or later, or Windows 11. Start Ubuntu, choose a Linux user name and password, and follow the Linux steps inside it. Keep the bundle under `/home/<you>`, not under `/mnt/c`: files on the Windows drive are much slower to use from Linux. `explorer.exe .` shows the folder in Windows, where the Windows Wireshark can open the pcap files. WSL does not pass USB devices to Linux by itself: for a real analyzer, install usbipd-win in Windows, share the device with `usbipd bind --busid <id>` (as administrator) and hand it to Linux with `usbipd`

`attach --wsl --busid <id>`. (Microsoft Learn, "Install WSL", "Working across file systems" and "Connect USB devices", checked November 2026.)

> **Note** **macOS: the tools differ**
>
> The setup script is for Debian and Ubuntu, so it does not run on a Mac. Homebrew has the tools as `icarus-verilog`, `sigrok-cli` and `wireshark` (the formula holds `tshark`), and OSS CAD Suite has macOS builds. The series does not test this route; for exactly the book's results, run Ubuntu in a virtual machine.

HANDS-ON LAB 6.1

Check your bench

GOAL

Prove that every tool of the desk bench is installed, found and working.

YOU NEED

The bench installed as above, and the lab folder `labs/00/ch06/bench-check/`.

STEPS

1. In a new terminal, go to the bundle and run `. scripts/tools-env.sh`.
2. Run `bash labs/00/ch06/bench-check/run.sh`. It finds every tool, prints its version and where it came from, and runs the simulator once on a tiny design, `hello.v`.

EXPECTED RESULT

On the author's computer it printed the report below. Paths that start with `.tools/` or `.venv/` are inside the bundle: the lab used the bench, not some other copy on your system.

OUTPUT bench-report.txt

```
Bench check: the free desk bench of The Protocol Library
  tool           version                      where
  python         3.13.5 (virtual env)         .venv/bin/python
  numpy          2.5.5
  scipy          1.18.1
  matplotlib     3.11.2
  scapy          2.8.0
  pyserial       3.5
  python-can     4.6.1
  crcmod         1.7
  iverilog       14.0 (devel)                 .tools/oss-cad-suite/bin/iverilog
  vvp            ran hello.v                  .tools/oss-cad-suite/bin/vvp
  sigrok-cli     0.7.2, 111 decoders          .tools/bin/sigrok-cli
  vcd-for-sigrok present                      .tools/bin/vcd-for-sigrok
  tshark         4.4.19, 3055 protocols       .tools/bin/tshark
PASS bench-check: 13 of 13 checks passed
```

IF IT DOESN'T WORK

- `EP_TOOLS` not set, or a tool `MISSING`: step 1 was skipped, or `setup-tools.sh` did not finish.

- `NOT a venv`: make `.venv` with `python3 -m venv .venv`. Anything else: Section 6.5.

TRY THIS NEXT

Run `which sigrok-cli` before and after `. scripts/tools-env.sh` and explain the difference.

6.5 When setup goes wrong BASIC

Most setup problems come from a terminal that has not loaded the bench, a missing package (on Ubuntu, enable the *universe* archive with `sudo add-apt-repository universe`), or the wrong Python.

Common mistake sudo pip install

Wrong: Install the packages with `sudo pip install numpy …` into the system's Python.

Why: Current Debian and Ubuntu releases mark their Python as managed by `apt`, so pip stops with `error: externally-managed-environment`; forcing it can break system programs.

Right: Make the virtual environment `.venv` once and install with `.venv/bin/pip`. The labs always call `$EP_PYTHON`, which is that Python.

Check yourself

1. Lab 6.1 worked yesterday; today a new terminal says `sigrok-cli: command not found`. Why?
2. Why must you never install the labs' packages with `sudo pip`?

DEBUGGING PLAYBOOK

Symptom	Likely cause	What to check	Fix
`command not found`	bench not loaded here	`echo $EP_TOOLS`	`. scripts/tools-env.sh`
`Unable to locate package`	old lists; universe off	`apt-cache policy`	`sudo apt update`
venv fails, names `ensurepip`	venv module missing	`dpkg -l python3-venv`	install `python3-venv`
`externally-managed-environment`	pip of the system	`which pip`	use `.venv/bin/pip`
no analyzer found	firmware, USB rights, WSL	`lsusb`	`sigrok-firmware-fx2lafw`; `usbipd`

6.6 Your first signal capture BASIC

Your first capture is the “Hi!” of Chapter 4, now run by you. Lab 6.2 uses the same Verilog transmitter and testbench, copied into its own folder. The testbench sends “Hi!” at 115 200 baud, 8N1, and saves only the `tx` line in a VCD file. Here is the decode step of its `run.sh`:

SHELL run.sh

```
sigrok-cli -I vcd:downsample=1000 -i out/hi-sigrok.vcd \
  -P uart:tx=tx:baudrate=115200:format=ascii -A uart=tx-data \
  > out/decoded.txt
sigrok-cli -I vcd:downsample=1000 -i out/hi-sigrok.vcd \
  -P uart:tx=tx:baudrate=115200 -A uart \
  --protocol-decoder-samplenum > out/annotations.txt
```

`-I vcd:downsample=1000` keeps every 1000th time step of the VCD. Icarus counts time in pico-seconds here, so sigrok sees one **sample** every nanosecond: a **sample rate** of 1 GS/s (giga-samples per second). `-P uart:…` starts the decoder on the channel `tx`, and `-A` chooses which of its annotations to print. The decoded text is `H`, `i`, `!`, one per line, and Figure 6.5 draws the capture as a logic analyzer shows it.

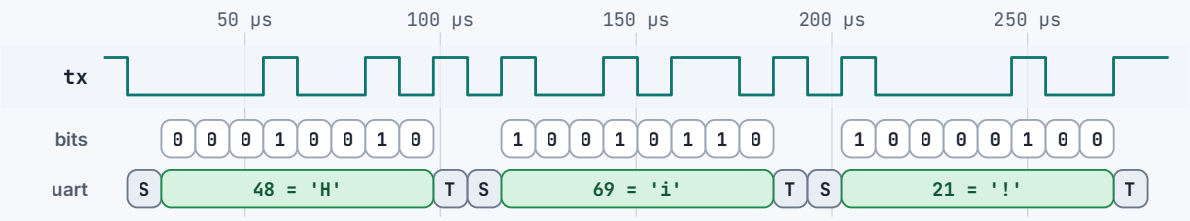


Figure 6.5 Your first capture, drawn from Lab 6.2’s VCD file and sigrok’s annotations. The ‘bits’ row shows each data bit sigrok sampled, D0 first; the ‘uart’ row shows the start bit (S), the byte and its character, and the stop bit (T).

A VCD file is plain text, so you can read it. After its header, a line `#t` gives a time and the next line gives the new value of a signal, by its one-character code; here `!` is `tx`:

OUTPUT vcd-head.txt

```
$timescale
        1ps
…
$var wire 1 ! tx $end
…
#20090000
0!
```

The `$timescale` is 1 ps, so `#20090000` means 20 090 000 ps = 20.09 µs, and `0!` says that `tx` fell to 0 there: the start bit of ‘H’. (`out/vcd-removed.txt` shows the `$comment` and `$dumpall` sections that `vcd-for-sigrok` removed.)

WORKED EXAMPLE 6.1

Read the first character by hand

Problem. Using only the VCD, find the first byte. The link runs at 115 200 baud, 8N1.

Step 1 — one bit time. T_{bit} = 1 / 115 200 = 8.681 µs.

Step 2 — measure each run. Divide the time from one change to the next by the bit time: from 20.09 µs to 54.81 µs is 34.72 µs = 4.00 bits of 0.

Step 3 — name the bits. The four 0s are the start bit, D0, D1 and D2; the next 1 is D3, and so on to the stop bit (Figure 6.6). Lab 6.2 does the same walk:

OUTPUT samples.txt

```
The first frame, read from the VCD (one bit = 8.681 us)
  VCD line    tx  lasts       bits  meaning
  #20090000   0   34.720 us   4.00  start, D0, D1, D2
  #54810000   1    8.680 us   1.00  D3
  #63490000   0   17.360 us   2.00  D4, D5
  #80850000   1    8.680 us   1.00  D6
  #89530000   0    8.680 us   1.00  D7
  #98210000   1    8.740 us   1.01  stop
  D0..D7 = 0 0 0 1 0 0 1 0 (LSB first) -> 0x48 = 'H'
```

Step 4 — assemble the byte. D0 is sent first, so D7…D0 = 0100 1000 = 0x48.

Answer. 0x48 = ‘H’. This tells us that a decoder does nothing magic: it measures runs of equal bits against the bit time, as you just did.

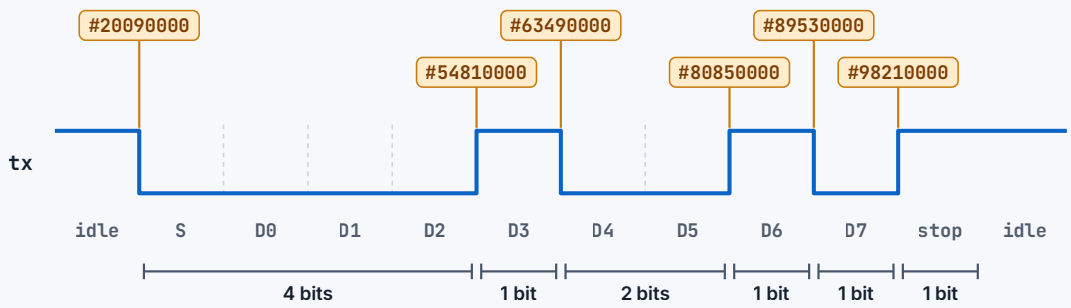


Figure 6.6 The first frame of the capture as the VCD describes it. Each amber label is the VCD time stamp, in picoseconds, of one change of tx; the brackets below give the length of each run in bit times (8.681 µs each).

sigrok reports the same frame in **sample numbers**: with `--protocol-decoder-samplenum`, each annotation starts with its first and last sample. Dividing by the sample rate gives a time:

$$t = n / f_s \qquad N_{bit} = f_s / B$$

where t is the time since the start of the capture in seconds, n the sample number, f_s the sample rate in samples per second (S/s), N_{bit} the number of samples in one bit, and B the baud rate (bits per second on a UART).

For the start bit, sigrok prints `20090-28771 uart-1: Start bit`: it begins at $t = 20\,090 / 10^9$ s = 20.090 µs, the time of `#20090000`, and lasts 28 771 − 20 090 = 8681 samples = 8.681 µs, one bit at 115 200 baud. A real logic analyzer samples far more slowly than 1 GS/s, so how fast must it be? $N_{bit} = f_s / 115\,200$ for each rate:

OUTPUT `samples.txt`

```
Samples per bit at other sample rates (rule: at least 4)
  1 GS/s      8680.6  ok
  24 MS/s      208.3  ok
  4 MS/s        34.7  ok
  1 MS/s         8.7  ok
  250 kS/s       2.2  too slow
```

A decoder needs at least about 4 samples per bit (Book 1, Chapter 21), so even 1 MS/s captures this console, and 250 kS/s does not: a low-cost analyzer is fast enough for UART consoles.

Check yourself

1. A VCD with `$timescale 1ps` says `#106950000` then `0!`. What time is that?
2. At 4 MS/s, how many samples does one 115 200-baud bit get?

HANDS-ON LAB 6.2

Your first signal capture

GOAL

Simulate a UART sending “Hi!”, decode the captured line with sigrok, and read it by hand.

YOU NEED

Icarus Verilog 14 or later (OSS CAD Suite), sigrok-cli 0.7.2, Python 3.13; `labs/00/ch06/first-uart-capture/`.

STEPS

1. Read `tb_uart.v` and `uart_tx.v` (Chapter 4 explains them).
2. Run `bash labs/00/ch06/first-uart-capture/run.sh`. It simulates, cleans the VCD, decodes it, saves `out/hi.sr` (the format PulseView opens), decodes that too, and runs `samples.py`. Then follow the first frame in `out/hi-sigrok.vcd`, as in the example.

EXPECTED RESULT

`out/decoded.txt` holds `H`, `i` and `!`; `out/samples.txt` holds the tables of this section; the last line is `PASS first-uart-capture: decoded 48 69 21 = "Hi!" from VCD and .sr`.

IF IT DOESN'T WORK

Nothing decoded: the VCD skipped `vcd-for-sigrok`. Wrong characters: check `downsample=1000` and the baud rate.

TRY THIS NEXT

Open `out/hi.sr` in PulseView, if you have it, and add the UART decoder. Then send your initials.

6.7 Your first packet capture BASIC

Your second capture is a network conversation. Lab 6.3 builds two packets with scapy: a **DNS** query that asks "what is the IPv4 address of `bench.example`?", and the server's answer. DNS is the internet's name-lookup protocol (Book 16). All names and addresses come from ranges reserved for documentation, and nothing is sent: the packets go straight into a pcap file.

PYTHON make_packets.py

```python
# scapy stacks layers with "/": each layer becomes the payload of the one
# on its left. Fields left out (lengths, checksums) are computed by scapy.
query = (
    Ether(src=PC["mac"], dst=DNS_SERVER["mac"])
    / IP(src=PC["ip"], dst=DNS_SERVER["ip"], id=1, ttl=64)
    / UDP(sport=PC["port"], dport=DNS_SERVER["port"])
    / DNS(id=0x1234, rd=1, qd=DNSQR(qname="bench.example", qtype="A"))
)
```

In scapy, `/` stacks **layers**: each layer on the right becomes the **payload** (the carried data) of the layer on its left. Ethernet carries IPv4, IPv4 carries UDP, and UDP carries the DNS message. You give the fields that matter; scapy computes the lengths and checksums.

Then tshark dissects the file. Figure 6.7 shows its three views, the same three panes the Wireshark window has: a one-line **summary** per packet, the **protocol tree** of one packet, and its bytes. The highlighted line and the highlighted bytes are the same field.

In the text form (`-V`), lines in square brackets are not bytes of the packet: Wireshark worked them out. For the answer it even matches the reply to its query by the transaction ID, and measures the delay:

DECODED CAPTURE answer-V.txt

```
    Answers
        bench.example: type A, class IN, addr 192.0.2.80
…
            Address: 192.0.2.80
    [Request In: 1]
    [Time: 0.004000000 seconds]
```

No.	Time	Source	Destination	Proto	Len	Info
1	0.000000	192.0.2.10	192.0.2.53	DNS	73	query 0x1234 A bench.example
2	0.004000	192.0.2.53	192.0.2.10	DNS	102	response 0x1234 A bench.example A 192.0.2.80

```
▸ Ethernet II, Src: 02:00:00:00:00:01 (02:00:00:00:00:01), Dst: 02:00:00:00:00:02 (02:00:00:00:00:…
▸ Internet Protocol Version 4, Src: 192.0.2.10, Dst: 192.0.2.53
▸ User Datagram Protocol, Src Port: 53000, Dst Port: 53
▾ Domain Name System (query)
    Transaction ID: 0x1234
    Flags: 0x0100 Standard query
    Questions: 1
    Answer RRs: 0
    Queries
      bench.example: type A, class IN
        Name: bench.example
        Name Length: 13
    … 3 more fields
```

```
0000   02 00 00 00 00 02 02 00  00 00 00 01 08 00 45 00   ··············E·
0010   00 3b 00 01 00 00 40 11  f6 71 c0 00 02 0a c0 00   ·;····@··q······
0020   02 35 cf 08 00 35 00 27  6f 18 12 34 01 00 00 01   ·5···5·'o··4····
0030   00 00 00 00 00 00 05 62  65 6e 63 68 07 65 78 61   ·······bench·exa
0040   6d 70 6c 65 00 00 01 00  01                        mple·····
```

Figure 6.7 Lab 6.3's capture as tshark dissects it. Top: the packet list (tshark -r; the Info column is shortened, 'query' for 'Standard query'). Middle: the protocol tree of packet 1 (tshark -V), with the DNS layer open. Bottom: its 73 bytes (tshark -x), coloured by layer; the highlighted field is the name bench.example.

A **display filter** (`-Y 'dns.flags.response == 1'`) keeps only the packets you ask for, and `-T fields` prints chosen fields as a table; the lab saves the responses' name, address and delay in `out/answers.txt`.

A pcap file adds a little to every packet: one 24-byte file header at the start, and a 16-byte record header (time stamp and lengths) before each packet. So its size is:

$$S = 24 + \Sigma\ (16 + L_i)$$

where S is the file size in bytes and L_i is the length of packet i in bytes.

WORKED EXAMPLE 6.2

Count a packet's bytes

Problem. Packet 1 is 73 bytes long. Split it into layers, compute the UDP and IPv4 length fields, and predict the size of the pcap file that holds both packets.

Step 1 — the layers. tshark's XML output (PDML) gives each layer's position and size (Figure 6.8): Ethernet 14, IPv4 20, UDP 8, DNS 31; 14 + 20 + 8 + 31 = 73.

Step 2 — the length fields. The UDP Length counts its own 8-byte header plus the DNS message: 8 + 31 = 39. The IPv4 Total Length counts its 20-byte header plus everything after it: 20 + 39 = 59. The Ethernet header is in neither.

Step 3 — the file. With L = 73 and 102 bytes: S = 24 + (16 + 73) + (16 + 102) = 231 bytes. The lab checks every count:

```
OUTPUT  layers.txt

  all layers                  73 bytes   (tshark frame.len: 73)
  UDP Length field  = 8 + 31 = 39   (in the packet: 39)
  IPv4 Total Length = 20 + 39 = 59   (in the packet: 59)
…
  file header 24 + packet headers 2 x 16 + packets 73 + 102 = 231 bytes
  size of dns.pcap on disk: 231 bytes
```

Answer. UDP Length 39, IPv4 Total Length 59, and a 231-byte file, exactly its size on disk. This tells us that every length field in a packet is a count of bytes you can check.

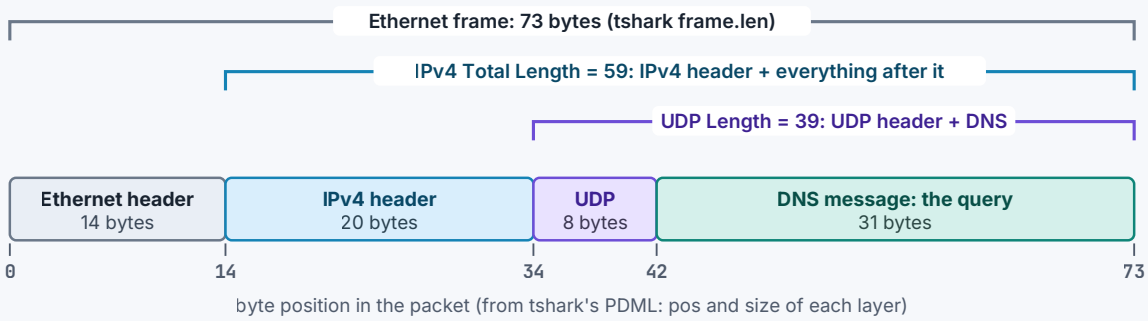


Figure 6.8 Packet 1 as nested envelopes, drawn from tshark's PDML. The UDP Length covers the UDP header and the DNS message; the IPv4 Total Length also covers the IPv4 header; only the Ethernet frame includes the 14-byte Ethernet header.

Check yourself

1. In tshark's output, what does a line in square brackets mean?
2. What does the `/` do in `IP() / UDP()` in scapy?

HANDS-ON LAB 6.3

Your first packet capture

GOAL

Build a DNS query and its answer with scapy, dissect them with tshark, and check every length.

YOU NEED

Python 3.13 with scapy, tshark 4.4 or later; `labs/00/ch06/first-packet-capture/`.

STEPS

1. Read `make_packets.py`: two packets, each a stack of four layers, with fixed time stamps.
2. Run `bash labs/00/ch06/first-packet-capture/run.sh`, then find each layer of packet 1 in `out/query-V.txt`. (It sets `XDG_CACHE_HOME` to `out/cache`, so scapy writes only in the lab.)

EXPECTED RESULT

The packet list of Figure 6.7, the counts of the worked example, and a last line `PASS first-packet-capture: query and answer dissected, sizes add up`.

IF IT DOESN'T WORK

`ModuleNotFoundError: scapy`: install scapy into `.venv`. `tshark: command not found`: source `tools-env.sh`.

TRY THIS NEXT

Rename the query `sensor.example`; predict the new sizes, then run the lab again.

In industry

- **Captures are evidence.** Engineers attach the `.sr`, VCD or pcap file to a bug report, so others can decode it again; the same command-line decoders run unattended in automated tests.
- **Simulate first, then measure.** Teams check a design's buses in simulation, as Lab 6.2 does, and use the same decoders on the first real board.

6.8 An optional hardware kit BASIC

The desk bench alone teaches every protocol; to see real wires, a small kit is enough (Figure 6.9). Choose a 3.3 V development board with USB (ESP32, Raspberry Pi Pico and STM32 Nucleo boards are common), an 8-channel USB logic analyzer that sigrok supports, with an input range that covers your logic levels, and a USB–UART adapter set to 3.3 V. Buy only what your next book needs.

Many low-cost analyzers keep no firmware of their own: sigrok loads its free `fx2lafw` firmware into them when it finds them (Book 1, Chapter 21). The desk bench in `.tools/` reads files only, so for real hardware also install your distribution's packages, which add the firmware and the USB permission rules (after the setup script):

SHELL

```
sudo apt install sigrok-cli sigrok-firmware-fx2lafw pulseview
```

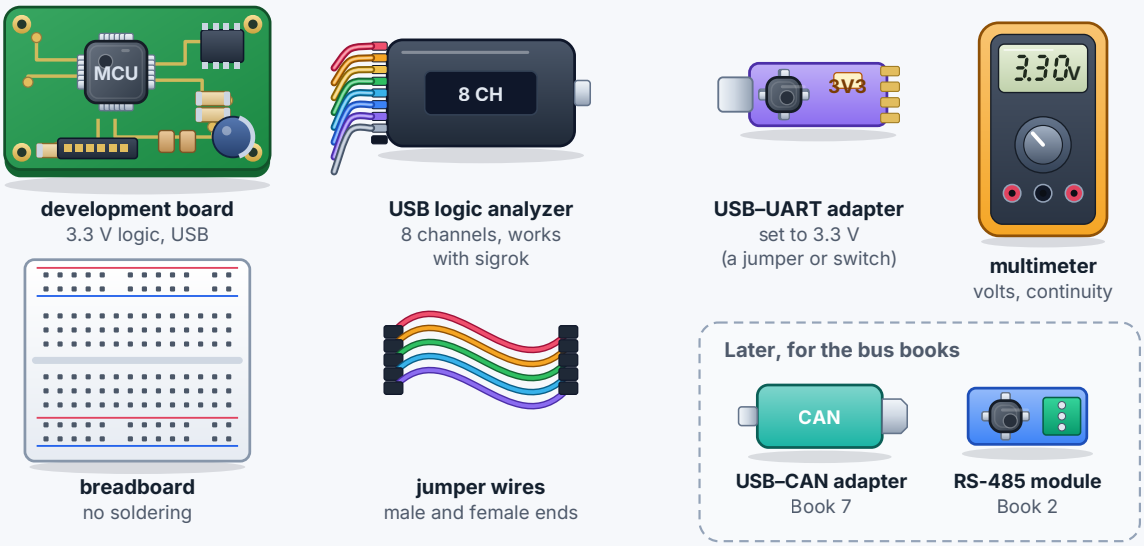


Figure 6.9 A low-cost kit for the bench boxes of this library: a development board, an 8-channel USB logic analyzer, a USB–UART adapter, a multimeter, a breadboard and jumper wires. A USB–CAN adapter and RS-485 modules come later, with the books that need them.

6.9 Safety first BASIC

Small boards are safe to touch, but the wrong voltage destroys a pin in a moment (Figure 6.10).

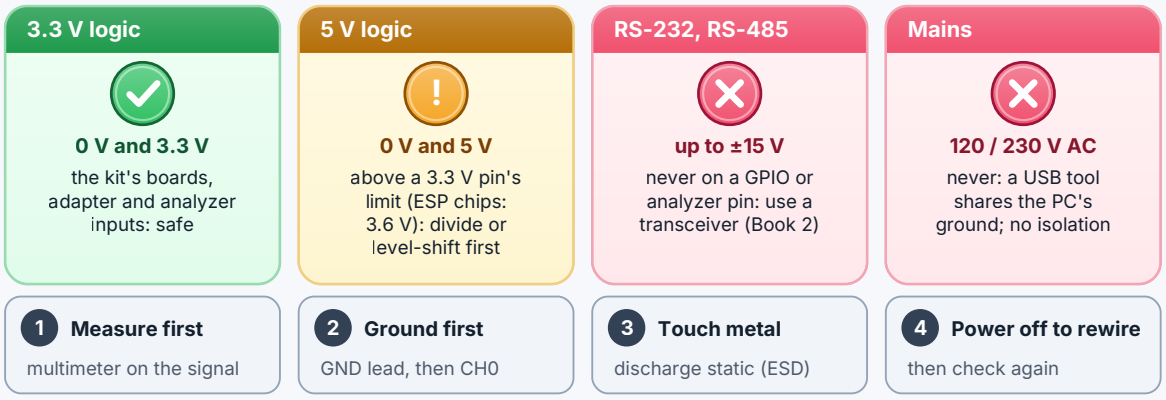


Figure 6.10 Which signals the kit may touch. 3.3 V logic is safe; 5 V logic needs a divider or level shifter before a 3.3 V pin; RS-232 and RS-485 lines and anything at mains voltage must never touch a GPIO or analyzer pin. Below: the four habits of a safe bench.

Most current boards use 3.3 V logic, and 5 V can damage a 3.3 V pin; use the divider or buffer of Chapter 4. Keep RS-232 (up to ±15 V) and RS-485 lines away from the analyzer, and check its documented input range. **Electrostatic discharge** (ESD), the spark from a charged body, can kill a pin without a visible mark: touch grounded metal first. And a USB analyzer shares its ground with your computer and has no isolation, so it must never touch mains wiring.

REAL-WORLD EXAMPLE

The 3.6 V limit of ESP chips

Espressif's hardware-design FAQ gives the GPIO voltage tolerance of its ESP chips as 3.6 V, and asks for a voltage divider when a signal is higher. So the 5 V TX of an older board must never go straight into an ESP32's RX pin: a divider of 1 kΩ over 2 kΩ brings it to 3.33 V (Chapter 4). (Espressif ESP-FAQ, "Hardware design", checked November 2026.)

KEY IDEA

Measure before you connect, connect ground first, and keep every signal inside the input range of what it meets: 3.3 V logic for this kit, never RS-232, RS-485 or mains.

Common mistake "The USB cable already shares ground"

Wrong: Clip only the analyzer's CH0 lead to TX and leave its GND lead off, because the board and the analyzer both plug into the same computer.

Why: The only ground path is then a long loop through two USB cables and the computer. Noise on that loop moves the analyzer's reference, and edges near the threshold turn into extra bits.

Right: Always connect the GND lead, first, to a ground pin next to the signal.

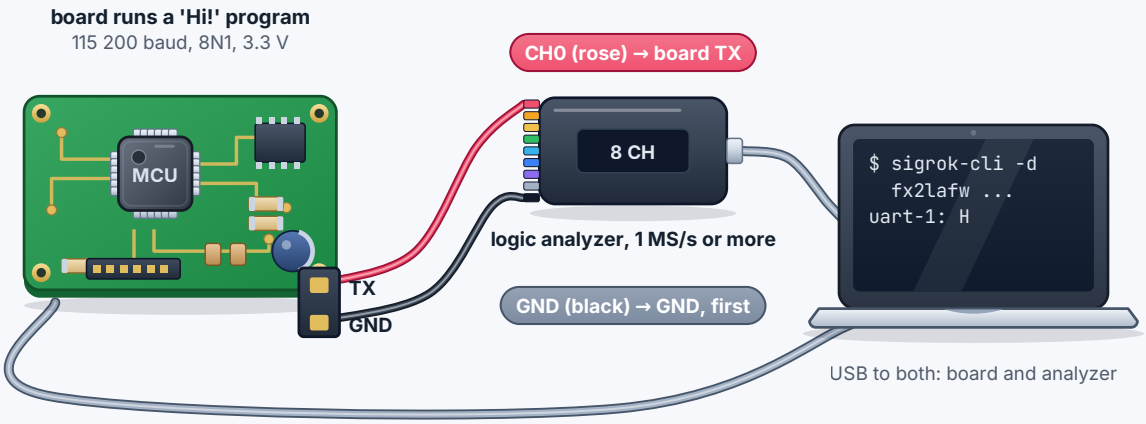


Figure 6.11 The wiring for your first real capture. The analyzer's GND lead goes to the board's ground first, CH0 goes to the board's TX pin, and both the board and the analyzer plug into the computer.

On your bench · not run by the author **First real capture on your own board**

Repeat Lab 6.2 on real hardware: a 3.3 V development board that prints “Hi!” once a second on a UART at 115 200 baud, 8N1 (the serial “hello” example of most boards), and an 8-channel USB logic analyzer that sigrok supports.

Analyzer	Board
GND (connect it first)	GND, next to the TX pin
CH0	the UART’s TX pin

1. Unpowered, wire GND, then CH0. Power the board; the multimeter should show TX idle at about 3.3 V.
2. In a new terminal (the system’s sigrok), `sigrok-cli --scan` should list the analyzer (in WSL, attach it with usbipd first).
3. Capture three seconds at 1 MS/s: `sigrok-cli -d fx2lafw -c samplerate=1m --time 3s -C D0 -o board.sr`.
4. Decode it with Lab 6.2’s settings: `sigrok-cli -i board.sr -P uart:tx=D0:baudrate=115200:format=ascii -A uart=tx-data`. You should see H, i and ! on lines of their own, a few times over, as in Lab 6.2. At `baudrate=57600` they should turn into wrong bytes.

6.10 What to read next BASIC

Your bench is ready. **Book 1** is the next step: it teaches the ideas every protocol shares, with a desk lab in every chapter, and its **Chapter 21** teaches the instruments in depth (scopes, logic analyzers, protocol analyzers, sigrok, Wireshark filters). Then pick a protocol book from the map of Chapter 3; Book 2 continues the UART.

CHAPTER AT A GLANCE

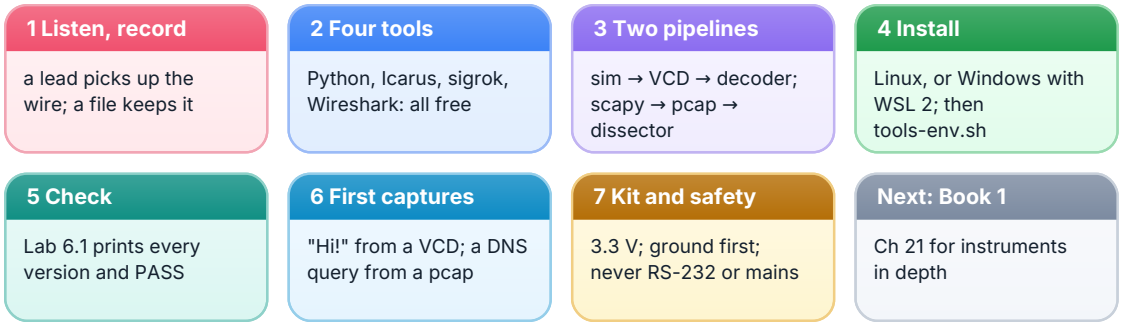


Figure 6.12 This chapter at a glance: the four jobs of a bench, the four free tools, their two pipelines, installing and checking them, your first two captures, the hardware kit and its safety rules, and Book 1 as the next step.

REVIEW QUESTIONS

Recall

1. Name the four tools of the desk bench and the file each one writes or reads.
2. Draw the two pipelines of the bench; mark where real captures enter.
3. What does `. scripts/tools-env.sh` change, and why does it start with a dot?

Apply

4. Lab 6.2's VCD has `#115630000` then `1!` : what time, and which bit starts?
5. Samples per 921 600-baud bit at 24, 4 and 1 MS/s? Which is enough?
6. Packet 2 of Lab 6.3 is 102 bytes. Find its DNS size, UDP Length and IPv4 Total Length, and the size of `dns.pcap` after you add a third packet of 90 bytes.

Debug or design

7. May the analyzer touch (a) a 3.3 V TX pin, (b) a 5 V TX pin, (c) RS-232, (d) mains?
8. Explain and fix `sigrok-cli: command not found` and `externally-managed-environment`.
9. Plan a real console capture: parts, wiring, sample rate, commands, result.

Answers: Appendix B.

Interview

1. No hardware yet: how do you practise? Decode simulated VCDs with sigrok, dissect scapy packets with tshark, and break settings on purpose to read the errors.

2. Why keep the capture file? To decode it again with other settings or tools, and check timing.

3. What do you check before probing an unknown board? The voltage, the analyzer's input range, ground and no mains; then connect ground first.

KEY TAKEAWAYS

- The free desk bench: Python 3 with its packages, Icarus Verilog (OSS CAD Suite), `sigrok-cli` and `tshark`.
- Signals: Verilog → VCD → `vcd-for-sigrok` → decoder; packets: scapy → pcap → tshark. Real captures use the same files.
- Install with `scripts/setup-tools.sh` and a `.venv` (Debian, Ubuntu or WSL 2); source `tools-env.sh` in every terminal.
- In a VCD, `#t` then `0!` means: at time t, ! became 0. Decoders need 4 or more samples per bit.
- UDP Length = 8 + payload; IPv4 Total Length = 20 + UDP Length; pcap = $24 + \Sigma(16 + L)$.
- Kit: 3.3 V board, 8-channel analyzer, 3.3 V USB–UART adapter, wires, meter; ground first, never RS-232 or mains.

APPENDIX A

Final Test

This test checks every learning outcome of the guide. Each question carries the number of the outcome it tests, so you can see at once which part of the guide it comes from.

- **How to take it.** Close the guide, take a sheet of paper and answer all 30 questions. Work the numbers by hand or with a calculator. You do not need a computer. Allow about two hours, in one or two sittings.
- **How to mark it.** Give yourself one mark for each question that you answered fully and correctly. A question with parts scores its mark only when every part is right. The worked answers are in Appendix Appendix B.
- **What your score means.** The pass mark is **27 out of 30** (90 %). With 27 or more, you are ready for Book 1, *Digital Communication Foundations*. Below 27, every answer you missed names the section to re-read. Re-read those sections, redo their exercises, and take the test again a few days later.

A.1 What a protocol is (Chapter 1)

1. A data logger reads a wind sensor on a mast. The sensor's manual lists six rules: (i) every message starts with the byte 0x02 and ends with 0x03; (ii) the speed field gives the wind speed in tenths of a metre per second; (iii) the sensor answers within 20 ms of a request; (iv) bytes travel at 4800 bit/s; (v) the command byte 0x57 asks the sensor to send its wind speed; (vi) the speed field is two bytes long, high byte first. (a) Is each rule a rule of format, of meaning or of timing? (b) A new logger set to 9600 bit/s reads nonsense, although the cable and the sensor are fine. Explain why, using the idea of a protocol as an agreement. Why can the logger not simply ask the sensor which rules it uses? (c) In Lab 1.1, a receiver set to half the sensor's speed (4800 baud, 8E1) read the greenhouse answer. What did it report? Outcome 1.1
2. A colleague's design note says: "Our gateway uses the USB-C protocol for power, the RS-485 protocol to reach the meters, the Wi-Fi protocol to reach the router and the JSON protocol to send readings to the cloud. Nobody can tamper with the readings, because every frame ends with a CRC-32 and the cloud link uses the AES protocol." (a) Correct each claim: say what each named thing really is, using this chapter's six words, and say what must be added so that each link is fully described. (b) What should the note say about tampering instead? Outcome 1.2
3. The greenhouse sensor of Chapter 1 now reads 31.6 °C. It still stores tenths of a degree and answers the same request. (a) What register value and which two data bytes does it send? (b) Write the whole Modbus RTU answer. The CRC-16/MODBUS over its first five bytes is 0xC5B9; in which order are the two CRC bytes sent? (c) Write the 11-bit UART character of the low data byte, in the order the bits leave the sensor, and say why its

parity bit has that value. (d) During that character's start bit, which RS-485 wire is higher, A or B, and by at least how much at the driver? (e) How long does the whole answer last on the pair at 9600 bit/s? Outcome 1.3

4. A barcode scanner is joined to a shop's till by a UART cable at 9600 baud, 8N1. Whenever it reads a code, it sends the digits as ASCII text followed by a carriage return. The till checks the code's last digit, a check digit; if it is wrong, the till beeps and the cashier scans again. (a) Answer the five questions every protocol answers for this link. (b) The shop wants four scanners on one shared RS-485 pair to the till, each still sending whenever it reads a code. Which answers break, and how does the polling of the greenhouse link avoid that? (c) In which Protocol Passport fields would you write your answers to "Who talks?" and "If it fails?"? Which group of passport fields is not one of the five questions, and why does every protocol in the library get the same 16 fields in the same order? Outcome 1.4, 1.6
5. (a) In the greenhouse answer 01 03 02 00 F5 78 03, which bytes are the Modbus part (the application layer), and which bytes does the serial-line framing add in front and behind? How many bits does the UART character format add to the whole answer, and what share of the 77 line bits carries the reading itself? (b) The cable must now cross a 600 m field where lightning often strikes, so the farm puts an RS-485-to-fibre converter at each end of that stretch: in the middle, light in an optical fibre replaces the voltages on the pair. Which layers change on that stretch, which stay the same, and why do the sensor and the controller need no change at all? Outcome 1.5

A.2 Where protocols live (Chapter 2)

6. A phone is in a video call over home Wi-Fi. The user wears Bluetooth earbuds, and the phone charges from a USB-C charger. Name the protocol on each link: (a) the front camera to the SoC; (b) the SoC to the screen; (c) the SoC to its working memory; (d) the call's data between the phone and the router; (e) the voice between the phone and the earbuds; (f) the Wi-Fi data, and the Bluetooth messages, between the combined Wi-Fi and Bluetooth chip and the SoC; (g) the phone and the charger agreeing on a higher voltage; (h) the SoC telling its power-management chip which voltages to make. (i) Which of the phone's radio links only ever receives, and why? Outcome 2.1
7. (a) Name the bus or link that carries each of these in a modern car: the engine unit's data to the gearbox unit, many times a second; a mirror motor's commands; a pedal-position sensor's reading, sent one way to its ECU; raw video from a surround-view camera to the driver-assistance computer; service messages between the driver-assistance computer and the head unit; the cell readings of an electric car's battery modules; the digital messages with which a car identifies itself to a charging station. (b) A scan tool plugged into the OBD-II socket reads a fault code from the brake unit, although the brake unit's own bus does not run to the socket. Which unit makes this possible? Which pins carry the data, what does pin 16 supply, which diagnostic services does the tool use, and over what can newer cars also be diagnosed? Outcome 2.2

8. A flow transmitter in a factory has a range of 0 to 50 m^3/h on a 4–20 mA loop. (a) It sends 14.4 mA. What flow does it report? (b) One morning the PLC reads 0 mA. What does that tell you, and why can a 4–20 mA loop tell it so clearly? (c) Follow the plant's data upwards. Name the protocol that carries the transmitter's setup and diagnostics on the same two wires; the one that links a smart sensor on a three-wire cable to its master; one that carries inputs and outputs between remote I/O and the PLC on a fixed cycle; the one that shares the PLC's data, with the meaning of each value, with the control room's screens; and the one that publishes data to the company's cloud. (d) How do the messages change from the bottom of the factory pyramid to the top, and how does an emergency stop travel on the same network? Outcome 2.3

9. A cloud company trains AI models in one data-centre hall. Name the link or protocol for each job: (a) a CPU reading its DIMMs; (b) the two CPU sockets of a server sharing work and memory; (c) a CPU reaching its GPUs and its network card; (d) GPUs exchanging data with each other; (e) a CPU using extra memory on an add-in card, on the same wires as (c); (f) the BMC reading fan speeds, and controlling the power supplies; (g) the flash chip that holds the firmware that starts the server; (h) a server's link to its top-of-rack switch; (i) routing between the switches, and virtual networks kept apart for different customers; (j) GPU servers writing directly into each other's memory; (k) a link to another data centre that sends many wavelengths of light down one fibre. (l) A server is switched off after a fault, but an operator must read its temperature log and power it on again from far away. Through which controller, and with which protocols? Outcome 2.4

10. (a) Name the data link for each: the switched Ethernet backbone of a Boeing 787; an air data computer sending values to the displays of an older airliner; flight plans and weather sent as text between an aircraft and its airline; the instruments of the James Webb Space Telescope sending data to its on-board computer; a space agency's commands sent up to a satellite; a drone's flight controller exchanging telemetry with ground-control software; a radio-control receiver passing the pilot's sticks to the flight controller; the flight controller driving each motor's speed controller. (b) For each new link, say what matters most, which kind of protocol fits, one example and the book that teaches it: (i) a dozen temperature sensors and a fan controller on one circuit board; (ii) a servo drive in a packaging machine that must receive a new position on time, every cycle; (iii) the brake, engine and gearbox units of a truck, on long cables near noisy motors; (iv) a phone app that must reach its company's server from anywhere in the world. Outcome 2.5, 2.6

A.3 The library map (Chapter 3)

11. (a) Name the shelf and the book that teach each of these: SWD, PROFINET, WireGuard, HBM, UCIe, DNS, FHIR. (b) Book 9, on vehicle diagnostics, needs CAN frames all the time. Where are they taught in full, and what does Book 9 do with them? Why does the library work this way? (c) Which depth tier does each of these get, how much text and how many labs does that tier mean, and what can you do after reading it: CAN; Token Ring; ISO 13485? Outcome 3.1, 3.2

12. You open the SPI book, Book 3, and want five things. In which dossier part, by number and name, do you look for each? (a) the order of the bits and fields in one transfer; (b) the highest clock rate that a given cable length allows; (c) the fix for a target that always reads 0xFF; (d) one page to revise before an interview; (e) how SPI differs from I^2C. (f) A later book gives a legacy protocol only a Tier C section. Which of these five parts are you sure to find there, and why? Outcome 3.3

13. A new Tier B protocol chapter is being written. (a) How many questions must its question universe hold at least, how many of them does the blind test use, and how many must the reviewer answer correctly for the chapter to pass? (b) The reviewer gets 17 right. What happens next? (c) Why may the reviewer read nothing but the finished chapter? (d) A reader asks: “How do I know the outputs printed in the desk labs are real?” and “How do I know the final test covers everything?” Answer both. Outcome 3.4

14. A graduate joins a company that builds battery systems and chargers for electric vehicles. (a) Which study path fits, with its books in reading order? Which of them are in Wave 1? (b) Their chargers talk to cars with ISO 15118 and send status to the company’s servers with MQTT. Which home book of these two protocols is already on the path, and which book would you add? (c) They know some CAN from work and want to start with Book 36 at once, skipping Book 1. What do you advise? (d) After Book 7 they score 84 % in its final test. What should they do? Outcome 3.5

A.4 UART in one chapter (Chapter 4)

15. A 3.3 V microcontroller talks to a 3.3 V Wi-Fi module at 115 200 baud, 8N1, with hardware flow control. (a) Fill in these passport fields for the link: topology, wires, max nodes, duplex, sync / async, addressing, error check. (b) List every connection between the two boards. (c) A technician wired TX to TX and RX to RX, and left out the ground wire “because both boards plug into the same USB hub”. What does each mistake cause? (d) The module is moved to an older 5 V board, whose RX input needs VIH = 0.7 × VCC and whose TX drives 5 V. What does each direction now need? Outcome 4.1, 4.6

16. (a) Write the line levels, in time order, of the letter ‘K’ (0x4B) in 8N1, in 7E1 and in 8O2, and give the number of bits on the wire for each. (b) A designer wants to save line time by dropping the stop bit. ‘K’ is followed at once by a second character. Using the idle level, explain what goes wrong, and what else the receiver loses. Outcome 4.2

17. A GPS module sends NMEA 0183 sentences at 38 400 baud, 8N1. (a) Find the bit time, the frame time, the efficiency and the largest number of characters per second. (b) How long does a 67-character sentence (with its CR LF) take? Lab 4.2 found 69.79 ms for it at 9600 baud: how much faster is it now? (c) The receiving microcontroller runs from an 8 MHz clock with 16× oversampling. Find the divisor, the actual baud rate and the error. Is that safe? (d) A colleague says that the module “moves 38 400 bytes per second”. Correct the statement. Outcome 4.3

18. A UART receiver uses 16× oversampling. (a) How many ticks after the falling edge of the start bit does it sample D0, D5 and the stop bit of an 8N1 frame? (b) A noise spike pulls the idle line low for 2 ticks. Why does the receiver not start a frame, and what else do

many UARTs do so that one spike cannot change a bit? (c) For 8E1, find the ideal limit for the total mismatch of the two clocks, and the limit once the receiver's one-tick uncertainty about the start edge is included. Why are they smaller than for 8N1? (d) An 8E1 receiver's bit time is 4 % longer than the sender's. In sender bit times, where does it sample the stop bit, and is that sample still inside the stop bit? Outcome 4.4, 4.3

19. (a) An 8E1 link runs beside a pump motor, and noise flips one bit of a byte. Which error does the receiver report, and with which flag of a 16550's line status register? What if the noise flips two bits? (b) A microcontroller with a one-byte receive buffer is busy for 5 ms while bytes arrive back to back at 115 200 baud, 8N1. How many bytes arrive in that time, what does the UART report, and is anything wrong on the wire? (c) A 9600-baud 8N1 transmitter holds its TX line low for 1.0 ms, and later for 3.0 ms. What does the receiver report each time, and why? Outcome 4.5

20. You debug a product's console with a logic analyzer whose UART decoder is set to 115 200 baud, 8N1. (a) After every reset, the first burst of characters decodes as garbage; then the console text is clean. The shortest pulse in the burst is 13.4 µs. What is happening, and what do you do? (b) A second product decodes as garbage, with framing errors, all the time, and its shortest pulse is 17.4 µs. Cause and fix? (c) On a third, the letters are right, but every byte has a framing error; the shortest pulse is 8.7 µs. Which wrong decoder setting fits, and why do the letters still come out right? (d) On a fourth, the capture decodes cleanly, but the product's own software loses characters at full load. Which flag do you read, and what are two fixes? Outcome 4.7, 4.5

A.5 Reading specifications and datasheets (Chapter 5)

21. (a) Is each of these a specification or a datasheet: RFC 9293; NXP's UM10204; National Semiconductor's PC16550D; TI's SNOSD82D for the TMP117? (b) Which body or group writes each of these: Wi-Fi; LPDDR5 memory; MIPI CSI-2; PCI Express; SAE J1939; RS-232; the AMBA buses; DNS; Bluetooth Low Energy? Outcome 5.1

22. (a) Is each of these free, sold or for members first, and how can a non-member read it: IEEE 802.3; ISO 11898-1:2024; the PCI Express Base Specification; a 3GPP technical specification; TIA-232-F; the MIPI I3C specification? (b) Decode these identifiers: IEEE Std 802.3-2022; a 3GPP specification at V17.3.0; ISO 11898-1:2024. (c) In a 2026 design review, an engineer quotes "RFC 793" for a TCP rule. What is wrong, and what should they quote? (d) A microcontroller's UART loses its first byte after waking from sleep, and its datasheet does not say why. Which document do you look for? Outcome 5.2

23. A made-up ISO-style standard says: "7.4.1 A receiver shall sample each bit at its centre. 7.4.2 A receiver should take three samples per bit and use the majority. 7.4.3 A transmitter may insert idle time between characters. 7.4.4 The idle time can exceed one second on slow links. NOTE A receiver must reject every character that has a framing error." (a) What does each of the five sentences ask of an implementer? (b) A supplier's receiver samples once per bit, at the centre, and passes characters with framing errors on to software with a flag. Does it conform? What should the supplier be ready to show? (c) The first rule appears in a recent RFC as "a receiver must sample each bit at its centre", in

lowercase. Is it a requirement in the RFC 2119 sense? (d) You must implement one feature from a 1,000-page standard. In what order do you read it, and which clause tells you what an implementation must do? Outcome 5.3

24. A board uses the TMP117 temperature sensor. (a) In which part of its datasheet do you find: the highest supply voltage the part survives; the supply range in which it is meant to run; its accuracy and its conversion time; the address and reset value of its result register? (b) A fault holds the board's 3.3 V rail at 5.9 V for ten minutes. Is the sensor likely to be damaged? Can you trust its readings during those minutes? (c) After power-up the firmware reads 0x8000 from the result register; later it reads 0xFE00. What does each mean? (The result is in two's complement, with 1 LSB = 7.8125 m°C.) (d) A status flag in another chip's register map is marked RC. What does that mean, and why can reading that register with a debugger confuse the firmware? Outcome 5.4

25. (a) A Fast-mode I^2C bus runs at 5.0 V, and its long traces and many devices give a bus capacitance of 400 pF. With UM10204's formulas (t_r = 300 ns, $V_{OL(max)}$ = 0.4 V, I_{OL} = 3 mA), find $R_{p(max)}$ and $R_{p(min)}$. Which standard resistor do you choose? Name two ways out. (Standard-mode allows t_r = 1000 ns, with the same I_{OL}.) (b) The ATmega324PB's table for double-speed mode (U2X = 1: S = 8, S_F = 4, S_M = 5) prints R_{slow} = 96.00 % and R_{fast} = 103.90 % for D = 8. Check both with the data sheet's formula. (c) In that mode, a 16 MHz AVR set up for 115 200 baud gets UBRR = 16, which gives 16 000 000 / (8 × 17) baud. Find the error. The table recommends at most ±1.5 % for this row. Is it safe, and what would you change? Outcome 5.5

A.6 Your protocol bench (Chapter 6)

26. (a) Name the desk-bench tool for each job, and the file it writes or reads: simulating a Verilog SPI transmitter with its testbench; decoding the simulated SPI data line into bytes; building an MQTT packet in a script; dissecting that packet layer by layer; computing a table of CRCs. (b) A colleague sends you `board.sr`, a capture from a real logic analyzer, and a customer sends `site.pcap` from a real network. Where does each file enter the bench's two pipelines, and which steps does each skip? (c) Why does the series use only tools with a command-line form? Outcome 6.1

27. A student has a Windows 11 laptop and the guide's code bundle. (a) List the steps, in order, from no Linux at all to a passing Lab 6.1, and say where the bundle must live. (b) Give the cause and the fix of each problem: (i) `python3 -m venv .venv` fails and names `ensurepip`; (ii) `apt` says `Unable to locate package`; (iii) `setup-tools.sh` stopped halfway when the disk filled up; (iv) Lab 6.1 reports `NOT a venv`; (v) a new terminal says `tshark: command not found`, although Lab 6.1 passed yesterday; (vi) the labs run very slowly from a folder under `/mnt/c/Users`. Outcome 6.2, 6.6

28. Lab 6.2's VCD file, with `$timescale 1ps`, holds these changes of the third character, in order (each time stamp has its own line, and the new value follows on the next): `#193810000` then `0!`; `#202490000` then `1!`; `#211170000` then `0!`; `#245890000` then `1!`; `#254570000` then `0!`; `#271930000` then `1!`. The link runs at 115 200 baud, 8N1, and `!` is the code of `tx`. (a) At what time does the frame start? (b) Measure each run in bit times,

name the bits and assemble the byte. Which character is it? (c) sigrok, at 1 GS/s, prints `193810-202491 uart-1: Start bit` and later `271935-280616 uart-1: Stop bit`. How long are the start bit and the whole frame, in µs? (d) You will capture a real 9600-baud console. Is 100 kS/s enough? And 20 kS/s? Outcome 6.3

29. A colleague builds a DNS query in scapy as `Ether() / IP() / UDP() / DNS(…)`, and its DNS message is 45 bytes long. (a) What does each `/` do, and which fields can you leave for scapy to fill in? (b) Find the UDP Length, the IPv4 Total Length and the length of the whole packet. (c) The pcap file holds this query and a 102-byte answer. How large is the file? (d) In tshark's `-V` output for the answer, a line in square brackets gives the time since the query. What kind of line is that? (e) Which display filter keeps only the answers? Outcome 6.4

30. You will make your first real capture: the UART from an older 5 V microcontroller board to an ESP32 board, which also drives a relay that switches a 230 V pump. (a) Which parts make up the low-cost kit, and what must the logic analyzer's input range cover? (b) What must sit between the 5 V board's TX and the ESP32's RX, and why? (c) In what order do you connect the analyzer, and what do you do before you handle the boards? (d) Which part of this setup must the analyzer never touch, and why? (e) A friend clips only CH0 to TX and leaves the GND lead off, "because both boards and the analyzer plug into the same laptop". The capture shows extra bits near the edges. Why? Outcome 6.5

APPENDIX B

Answers

Chapter 1 — What a Protocol Is (and Is Not)

Check yourself: the idea. 1. “Over” hands the turn to the other side; “say again” handles a lost word. 2. Format: a call starts with the name of the station it is for, and numbers are said digit by digit. Timing: speak slowly, and wait for “over” before you answer.

Check yourself: six words. 3. “Serial protocol” is about the protocol (the UART character format). “TTL” and “3.3 V” are about the electrical level: logic-level signals that swing between 0 V and 3.3 V.

Check yourself: the trace. 4. 21.0 °C × 10 = 210 = 0x00D2, so the data bytes are 00 and D2. 5. The electrical standard (RS-485) and the character timing would change, and the interface would become a radio module with an antenna. The data format and the Modbus messages could stay the same.

Check yourself: the questions. 6. A sensor speaks only when the controller asks it by its address, and the controller asks one sensor at a time.

Review questions

1. A protocol is a set of rules, agreed in advance by every side, for the format, meaning, order and timing of messages and for error handling. Format: the answer is address, function, byte count, data, CRC. Meaning: function 3 means “read holding registers”, and register 0 holds tenths of a degree. Timing: 9600 bit/s, and a silence of 3.5 characters ends a frame.
2. Interface: where two things meet (a terminal block). Electrical standard: how 0 and 1 look as voltages (RS-485). Network standard: a whole network family in one document (IEEE 802.11). Data format: how a value is written as bytes (245 as 00 F5). Security algorithm: a calculation that hides or signs data (AES).
3. Who may speak? When? How fast? In what format? What if it fails? “Who talks?” is answered by topology, max nodes and addressing.
4. Electrical standard: RS-232. Data format: JSON. Protocol: I^2C and Modbus RTU. Network standard: IEEE 802.3. Security algorithm: SHA-256. Interface: the 3-pin header and the USB-C receptacle.
5. 18.7 × 10 = 187 = 0x00BB, so the data bytes are 00 BB. The answer takes 7 × 11 = 77 bits; 77 / 9600 = 8.021 ms, the same as in Lab 1.1 (`timing.txt`), because the length does not depend on the value.
6. On the new path the electrical layer (Ethernet instead of RS-485) and the framing change, and a network layer (IP) and a transport layer (TCP) are added; the RTU address and CRC are replaced by the Modbus TCP header and by the checks of Ethernet and TCP. The

application layer, the Modbus part 03 02 00 F5, stays the same. The gateway still uses Modbus RTU on the pair to the sensor.

7. Who: one module talks and one board listens; no addresses. When: the module sends whenever it has new data, usually once a second. How fast: an agreed baud rate, often 9600. Format: lines of text with `$`, fields, `*` and a checksum. Failure: a line with a bad checksum is dropped. The GPS module pushes data to a single listener; the greenhouse sensor is one of many and speaks only when polled.
8. Any three of: the protocol (is it Modbus RTU at all?); the baud rate and character format (for example 9600 8E1 on both sides); the address the controller uses; the register map (which register holds which value, and in which data format); whether A and B are swapped.
9. A receiver set to 8N1 expects the stop bit right after D7, but the sensor sends a parity bit there. The eight data bits were read correctly, so the bytes were right. Where the parity bit was 0, the receiver saw a 0 where it expected the stop bit's 1 and reported a framing error. Five of the seven bytes (03, 00, F5, 78, 03) have parity 0.

Chapter 2 — Where Protocols Live

Check yourself: the phone. 1. MIPI CSI-2 from the camera to the SoC, and MIPI DSI from the SoC to the display. 2. USB Power Delivery.

Check yourself: the car. 3. They need only slow, simple commands, and LIN needs one signal wire and a cheaper chip than CAN, so it costs less for each of the many small motors.

Check yourself: the factory. 4. 4 mA is the bottom of the range: 0 °C. 0 mA means the loop is broken (a cut wire or a dead transmitter), because a working loop never goes below 4 mA.

Check yourself: the data centre. 5. PCIe; the SSD speaks NVMe over it.

Check yourself: the sky. 6. ADS-B, broadcast on 1090 MHz.

Review questions

1. Any four of: LPDDR (SoC to DRAM, working memory); UFS or eMMC (flash storage); MIPI CSI-2 (camera pictures); MIPI DSI (display picture); I^2C, I3C or SPI (touch and sensor readings); I^2S, TDM or SoundWire (audio samples); SPMI (power-chip control); MIPI RFFE (radio front-end control); PCIe or SDIO plus UART (Wi-Fi data and Bluetooth HCI); ISO/IEC 7816 (SIM); USB and USB Power Delivery (data and charging).
2. LIN joins the door module to its motors; CAN (or CAN FD) joins the engine and brake units.
3. ADS-B broadcasts the position; SpaceWire carries the instrument data.
4. Camera → SoC over MIPI CSI-2; SoC → display over MIPI DSI. The SoC sets the camera up over a slower I^2C-style control bus.
5. Pins 6 (CAN high) and 14 (CAN low), the CAN bus required for US OBD-II since the 2008 model year. The diagnostic services (UDS) and OBD-II are taught in Book 9, and CAN in Book 7.

6. 12 mA is (12 − 4) / (20 − 4) = 50 % of the range, so the level is 5 m. HART can carry the diagnostics on the same two wires.
7. Ethernet into the network card; PCIe from the network card to the CPU; PCIe again from the CPU to the SSD, carrying NVMe commands and data. On top of Ethernet, the request itself uses IP, TCP or QUIC, HTTP and usually TLS.
8. The stick commands use the radio-control link (the receiver passes them on with CRSF or SBUS); the telemetry uses MAVLink to the ground-control software. Since the sticks work, check the MAVLink link: the telemetry radio or network, and its settings.
9. Any two of: a window motor needs only slow, simple commands; LIN uses one signal wire instead of two; a LIN node is cheaper than a CAN or Ethernet node; keeping slow traffic off CAN leaves the CAN bus free for the units that need it.
10. A long-range, low-power radio such as LoRaWAN ("across a site" in Figure 2.1, Book 22). Wi-Fi reaches only tens of metres and uses far more power, so the battery would not last.

Chapter 3 — The Library Map

Check yourself: thirteen shelves. 1. MQTT: the Internet networking shelf, Book 17 (IoT Messaging). LoRaWAN: the Wireless shelf, Book 22 (LPWAN). 2. Book 18, Radio Foundations, teaches the ideas that every radio protocol uses (modulation, OFDM, spread spectrum, link budgets), so the protocol books on that shelf can build on it.

Check yourself: tiers. 1. Tier B. 2. CSMA/CD is not a protocol but an access method (a rule for sharing a medium), so it gets a short note saying what it is and where it is taught (medium access in Book 1, its history and maths in Book 13).

Check yourself: the promise. 1. A desk lab runs on a computer with free tools and was run by the author, so its printed output is real. A bench lab is a recipe for the reader's own hardware, marked "not run by the author", and describes the expected result. 2. A fresh reviewer who has not seen the question key; it may read only the finished chapter, so the test shows what the chapter itself teaches.

Check yourself: study paths. 1. Book 1, because every protocol is built from its ideas. 2. The VLSI / SoC design and verification path: 1 → 3 → 4 → 6 → 28 → 29 → 30 → 26 → 31 → 32, which holds PCIe (Book 29) and DRAM interfaces (Book 26).

Review questions

1. Thirteen shelves and forty paid books, plus the free guide. SPI: Board-level & embedded, Book 3. MQTT: Internet networking, Book 17. HDMI: Multimedia, Book 32. LoRaWAN: Wireless, Book 22.
2. A: several chapters, 40–100 pages, all 22 parts, 100–200 printed questions. B: one chapter, 15–35 pages, all 22 parts in shorter form, 40–80 questions. C: a section of 5–12 pages, the core parts only, 15–25 questions. D: a note of 1–2 pages saying what the item really is, with a pointer.
3. P6 frame format, P9 timing and performance, P16 debugging playbook, P12 versions and variants, P22 cheat sheet.

4. In Book 11, its home book. Book 9 uses TSN and points to Book 11 instead of teaching it again. Other examples: TLS lives in Book 35 and is used by Book 16 (HTTPS); USB Power Delivery lives in Book 30 and is used by Book 36; DDC/EDID lives in Book 32 and is used by Book 4.
5. Both are Tier D: CSMA/CD is an access method and RINEX is a file format. Each gets a short note that says what it really is and points to where it is used, so a reader is not misled into treating it as a protocol.
6. Any six of the core parts: P0 Passport, P1 why it exists, P2 where it lives, P4 physical layer, P5 bit-level signalling, P6 frame format, P7 transactions, P10 errors, P15 see it on the wire, P21 question bank, P22 cheat sheet.
7. Every question it missed becomes new or clearer text, and the chapter is tested again until it reaches at least 95 %. The reviewer reads only the chapter so that the score measures what the chapter teaches, not what the reviewer already knew from elsewhere.
8. 1 → 2 → 4 → 7 → 8 → 9 → 13 → 35. Books 1, 2, 4, 7 and 13 are in Wave 1 (Book 1 is out now), Book 8 is in Wave 2, and Books 9 and 35 are in Wave 3.
9. Every output in a desk lab was produced by a real run on free tools and copied from the lab's output files, and each lab checks its own result; the reader can run the same lab and compare. A bench lab is for the reader's own hardware; it is marked "not run by the author" and describes the expected result instead of printing an output.
10. The embedded firmware path: 1 → 2 → 3 → 4 → 5 → 6 → 30 → 7 → 20 → 21. Book 30 (USB) is in Wave 1; Book 20 (Bluetooth) is in Wave 2.

Chapter 4 — UART in One Chapter: A Complete Sample Dossier

Check yourself: roles and architecture. 1. There are only two devices and each wire has one transmitter, so every byte can only be for the other side. 2. TX and RX are separate wires, so both sides can send at the same time.

Check yourself: physical layer. 1. No. A logic-level UART idles high, so a long high level is just an idle line. 2. Negative: idle is mark (1), and RS-232 sends mark as −3 to −15 V.

Check yourself: bit-level signalling. 1. D3 is the fifth bit after the edge (start, D0, D1, D2, D3), so its middle is 8 + 4 × 16 = 72 ticks after the edge (Lab 4.2, `out/exercises.txt`). 2. To make sure the falling edge was a real start bit and not a short noise spike.

Check yourself: frame format. 1. 1 + 7 + 1 + 2 = 11 bits. 2. D0, the least significant bit.

Check yourself: errors. 1. No. Two flipped bits keep the count of 1s even (or odd), so parity cannot see them. 2. Overrun: the bits arrived correctly, but software read too slowly.

Review questions

1. Any eight of: full name Universal Asynchronous Receiver-Transmitter; born from start-stop telegraphy, single-chip around 1971; no governing standard for the format (data sheets), RS-232/422/485 for the wires; point-to-point; TX, RX, GND (RTS/CTS optional); common rates 9600 and 115 200 baud; 2 nodes; full duplex; asynchronous; no addressing;

parity and framing checks; met in consoles, modules, bootloaders and USB–UART bridges.

2. The 8 ticks bring it to the middle of the start bit. Checking that the line is still low rejects noise spikes (false start bits). After that, steps of 16 ticks land in the middle of every following bit, the point farthest from both edges.
3. The idle line is high, so a 0 start bit always makes a falling edge to synchronise on. The 1 stop bit returns the line to idle so that the next frame also starts with a falling edge, and it gives the receiver a known value to check (framing).
4. Lab 4.2 (`out/exercises.txt`) gives: 7E1: 0 | 1 0 0 0 0 1 0 | 0 | 1 (D0 to D6 hold two 1s, so the even parity bit is 0). 8O2: 0 | 1 0 0 0 0 1 0 0 | 1 | 1 1 (two 1s, so the odd parity bit is 1).
5. Bit time 17.36 µs; frame 11 bits = 190.97 µs; efficiency 8/11 = 72.7 %; 57 600 / 11 = 5236 characters per second (Lab 4.2).
6. 115 200 at 16×: exact $N = 6.51$; $N = 7$ gives 107 143 baud (−6.99 %) and $N = 6$ gives 125 000 baud (+8.51 %), so neither works. 9600 at 16×: $N = 78$ gives 9615 baud (+0.16 %), fine. 8× at 115 200: $N = 13$ gives 115 385 baud (+0.16 %), so yes, 8× oversampling fixes it (Lab 4.2).
7. The frames in the capture follow each other with no gap. The stop-bit sample is taken 9.5 receiver bits after the edge and must land between 9 and 10 sender bits. A fast receiver (+5.5 %) samples at 9.5 / 1.055 = 9.005 sender bits, just inside the stop bit. A slow receiver (−5.5 %) samples at 9.5 / 0.945 = 10.05 sender bits, inside the next start bit, so it sees 0 and reports a framing error (numbers from Lab 4.2, `out/exercises.txt`).
8. The sender uses 8N1 but the analyzer expects 8E1, so it reads the stop bit as a parity bit and the next start bit as the stop bit. Lab 4.1's wrong-settings table shows exactly `48 5A` with 4 errors. Confirm by decoding at 8N1 (clean "Hi!") or by counting bits between start edges on the capture.
9. Module TX → board RX directly (3.3 V is a valid 1 only if the board's input has a low enough VIH; check the data sheet, or add a 74HCT buffer). Board TX → module RX through a divider (1 kΩ over 2 kΩ gives 3.33 V) or a level translator. Join the grounds. Set the board's UART to 9600 8N1.
10. The line is inverted: it was decoded as logic level but is really RS-232 (or the analyzer's invert option is set wrongly). Lab 4.1 shows that an inverted "Hi!" decodes as `AB A9` with no errors. Add or remove the RS-232 transceiver, or set the decoder's invert option to check.

Chapter 5 — Reading Specifications and Datasheets

Check yourself: access. 1. For example: free, RFC 9293 or NXP UM10204; sold, ISO 11898-1:2024; members first, the PCI Express Base Specification. 2. No. Free access is not free reuse: the specification is copyrighted, and its notice limits use by non-members. Describe it in your own words, draw your own figures and cite the section.

Check yourself: normative language. 1. No. Since RFC 8174 the key words have their special meaning only in capitals; a lowercase “must” is ordinary English, although the authors probably meant it.

2. Only 6.3.1 (shall: sample at the centre). 6.3.2 is a recommendation and 6.3.3 an option.

Check yourself: registers. 1. 8 data bits = 11 in bits 1–0 (0x03), parity enable = bit 3 (0x08), odd parity means bit 4 = 0: 0x0B. 2. 0xFFF0 is −16 in two’s complement, so −16 × 7.8125 m°C = −0.125 °C (Lab 5.1).

Review questions

1. Ethernet: IEEE (802.3). CAN: ISO (ISO 11898). TCP: the IETF (RFC 9293). 5G NR: 3GPP. DDR5: JEDEC. USB Power Delivery: USB-IF.
2. Free: any IETF RFC, 3GPP specification, ITU-T Recommendation or NXP UM10204. Sold: ISO and IEC standards such as ISO 11898-1:2024, SAE J1939, TIA-232-F. Members first: PCI Express Base Specification; the full MIPI I3C specification. Free to read still does not let you copy the text or figures into your own documents, and some (the Bluetooth Core Specification) forbid non-members to use them for products.
3. shall: a requirement; should: a recommendation, to be followed unless there is a good reason; may: a permission (optional); can: a possibility or capability. No: notes and examples are informative and never contain requirements.
4. ISO 11898-1:2024: ISO standard 11898, part 1, published 2024. 3GPP TS 38.331 V18.2.0: technical specification 331 of the 38 series (5G NR), approved and in Release 18, with two rounds of approved technical changes and no editorial-only changes since 18.0.0. IEEE Std 802.11ax-2021: amendment “ax” to the IEEE 802.11 standard, approved in 2021. SNOSD82D: TI literature number SNOSD82, revision D.
5. 8E1: 0x1B (word length 11, parity enable, even parity). 7O2: 0x0E (word length 10, two stop bits, parity enable, odd parity) (Lab 5.1, `out/exercises.txt`).
6. D = 8 data + 1 parity = 9, so row D = 9 (recommended ±1.5 %). UBRR = 12 gives 8 000 000 / (16 × 13) = 38 461.5 baud, +0.16 %. That is well within ±1.5 % (Lab 5.1).
7. $R_{p(max)}$ = 300 ns / (0.8473 × 150 pF) = 2.36 kΩ; $R_{p(min)}$ = (3.3 − 0.4) V / 3 mA = 967 Ω. Any value between them works, for example 2.2 kΩ (Lab 5.1).
8. Nothing is strictly required by these two sentences. Retrying after 10 ms is a recommendation (should), so a device may deviate with a good reason. The NOTE is informative, so its “must not” binds nobody; under ISO rules it is also the wrong verb. Follow it as good practice, and report it to the standards body as an editorial problem.
9. 15.5 ms is the typical conversion time; the maximum is 17.5 ms, so some conversions are not finished when the firmware reads, and it gets the old result. After reset the result register holds its reset value 8000h, which reads as −256 °C until the first conversion is complete. Wait at least the maximum, 17.5 ms plus margin, and treat −256 °C as “no result yet”.

Chapter 6 — Your Protocol Bench

Check yourself: how the tools connect. 1. sigrok's decoder (`sigrok-cli -P uart`) turns the wire into bytes; tshark's dissectors turn a packet into fields. 2. sigrok 0.5.2 stops reading a VCD at Icarus's `$comment` and `$dumpall` sections and decodes nothing; the helper removes them.

Check yourself: when setup goes wrong. 1. A new terminal does not keep yesterday's setting: run `. scripts/tools-env.sh` again. 2. The system's Python belongs to the distribution's package manager; pip refuses (`externally-managed-environment`), and forcing it can break system programs. Use `.venv`.

Check yourself: first signal capture. 1. 106 950 000 ps = 106.95 µs (the start bit of 'i'). 2. 4 000 000 / 115 200 = 34.7 samples per bit (Lab 6.2, `out/samples.txt`).

Check yourself: first packet capture. 1. A value Wireshark worked out, not a byte of the packet, such as a link to the matching request or a time difference. 2. It stacks layers: the UDP layer becomes the payload of the IP layer.

Review questions

1. Python 3 (scripts; scapy writes pcap files), Icarus Verilog (writes VCD files), sigrok (reads VCD and `.sr` files), Wireshark/tshark (reads pcap and pcapng files).
2. Signals: Verilog design and testbench → `iverilog` / `vvp` → VCD → `vcd-for-sigrok` → `sigrok-cli` decoder → bytes; a real logic analyzer's `.sr` file enters at the decoder. Packets: scapy script → pcap → `tshark` dissectors → fields; a real network capture is a pcap that enters at tshark.
3. It puts `.tools/bin` and `.tools/oss-cad-suite/bin` first on `PATH`, and sets `EP_PYTHON` to `.venv/bin/python` (plus the sigrok decoder and Wireshark profile folders). The dot reads the file into the current shell; `bash scripts/tools-env.sh` would change only a child shell that ends at once.
4. 115 630 000 ps = 115.63 µs. The second frame's start bit began at 106.95 µs, so this is one bit (8.68 µs) later: D0 of 'i' (0x69, whose D0 is 1). sigrok's `out/annotations.txt` agrees: its line `115631-124312 uart-1: 1` is the first data bit of the second frame.
5. 26.04, 4.34 and 1.09 samples per bit (Lab 6.2, `out/samples.txt`). Trust 24 MS/s; 4 MS/s just meets the four-samples rule with no margin; 1 MS/s cannot capture it.
6. 102 − 14 − 20 − 8 = 60 bytes of DNS; UDP Length = 8 + 60 = 68; IPv4 Total Length = 20 + 68 = 88 (Lab 6.3, `out/layers.txt`, which checks them against the packet). The file grows to 231 + 16 + 90 = 337 bytes: one more record header and the packet's bytes (`out/layers.txt`).
7. (a) Fine: measure that TX idles near 3.3 V, connect GND first, then CH0. (b) Only if the analyzer's input range includes 5 V; never feed that line into a 3.3 V board pin without a divider or level shifter. (c) No: RS-232 swings up to ±15 V; use an RS-232 transceiver and measure its logic side. (d) Never: a USB analyzer is not isolated, and mains voltage is lethal.

8. The new terminal has not loaded the bench: run `. scripts/tools-env.sh` in it. The `pip` it found is the system's, which Debian and Ubuntu protect; use `.venv/bin/pip install scapy` (the labs use `$EP_PYTHON` from `.venv`).
9. Parts: a 3.3 V development board printing "Hi!" at 115 200 baud 8N1, an 8-channel USB analyzer that sigrok supports, jumper wires, a multimeter. Wiring: GND to GND first, CH0 to TX; check TX idles at about 3.3 V. Sample at 1 MS/s or more (8.7 samples per bit). Commands: `sigrok-cli -d fx2lafw -c samplerate=1m --time 3s -C D0 -o board.sr`, then decode with `-P uart:tx=D0:baudrate=115200`. Expect the bytes 48 69 21 ("Hi!") repeated, as in Lab 6.2, and wrong bytes at a wrong baud rate.

Appendix A — Final Test

1. (a) Format: (i) and (vi), because they fix the parts of a message, their length and their order. Meaning: (ii) and (v), because they say what a field or a byte says and what the receiver must do about it. Timing: (iii) and (iv), because they say how fast bits go and how long to wait. (b) A protocol is a set of rules agreed in advance by every side. The new logger applies a bit time half as long as the sensor's to the same voltages, so it cuts the signal into the wrong bits and builds wrong bytes. It cannot ask, because its question would itself have to follow rules that both sides already share: the rules must be agreed before the first bit is sent. (c) Only three bytes, 14 04 26, with 8 errors, instead of 01 03 02 00 F5 78 03 with none. The wire was the same; only the timing rule differed. *Re-read Sections 1.1 and 1.6 and Lab 1.1.*

2. (a) USB-C is an interface, a connector and cable; the protocol that agrees on a charging voltage is USB Power Delivery. RS-485 is an electrical standard: it fixes only the voltages on the pair, so the note must also name the protocol, the speed, the character format and the addresses, for example "Modbus RTU at 9600 baud, 8E1, addresses 1 to 5". Wi-Fi products follow IEEE 802.11, a network standard: one document with several protocols in it (radio physical layers, frame formats, rules for sharing the air, joining and security). JSON is a data format; an application protocol such as MQTT or HTTP must carry it. AES is a security algorithm; a security protocol such as TLS decides when to use it, with which key, and what to do with the result. (b) A CRC-32 is an error-check algorithm: it catches noise, not attackers, because anyone who changes a frame can attach a new, correct CRC. The note should name a security protocol, for example TLS on the cloud link, that uses a security algorithm with a secret key, such as a message authentication code. *Re-read Section 1.2.*

3. (a) 31.6 × 10 = 316 = 0x013C, sent high byte first: 01, then 3C. (b) 01 03 02 01 3C B9 C5: address 1, function 3, byte count 2, the two data bytes, and the CRC low byte first (B9, then C5). A receiver that runs the CRC over all seven bytes gets 0x0000. (c) 0x3C = 0011 1100, so D0 to D7 = 0 0 1 1 1 1 0 0. The character is 0 | 0 0 1 1 1 1 0 0 | 0 | 1: start bit, data LSB first, parity, stop bit. The data hold four 1s, already an even number, so the even-parity bit is 0. (d) The start bit is a 0, and for a 0 wire A is above wire B. The driver must make a difference of at least 1.5 V; a receiver needs only 0.2 V. (e) 7 characters × 11 bits = 77 bits, and 77 / 9600 = 8.021 ms, the same as for 24.5 °C, because the length of the answer does not depend on the value. *Re-read Section 1.3 and Lab 1.1.*

4. (a) Who: one sender (the scanner) and one listener (the till), so no addresses are needed. When: the scanner pushes a code whenever it reads one. How fast: an agreed 9600 baud, 8N1. What format: ASCII digits, ended by a carriage return. What if it fails: the till rejects a code with a wrong check digit and beeps, and the cashier scans again. (b) "Who?" and "When?" break. Four scanners that send whenever they like can talk at once on the shared pair, and the till cannot tell who sent a code, because there are no addresses. With polling, each device has an address and one client asks one server at a time; a server speaks only when asked, so two devices never talk at once. (c) "Who talks?": topology, max nodes and addressing. "If it fails?": error check. The group "Who is it?" holds the identity fields: the full name, who created the protocol and when, the governing specification and the edition taught. With the same card for every protocol, you can compare two protocols field by field and see at once what differs. *Re-read Sections 1.4 and 1.7.*

5. (a) The Modbus part is 03 02 00 F5: function, byte count and data. The framing adds the address 01 in front and the CRC 78 03 behind. The character format adds a start, a parity and a stop bit to each of the 7 bytes: 21 bits. The reading is 16 of the 77 bits, 20.8 %, because each layer wraps the data of the layer above with its own parts (encapsulation). (b) Only the lowest layer changes: on the fibre, each bit is light instead of a voltage difference, and the interface there is a fibre connector instead of the terminals A and B. The UART characters (8E1 at 9600 bit/s), the RTU frame with its address and CRC, the Modbus part and the data format all pass through unchanged. Each layer only uses the service of the layer below, so the devices at both ends still see the same RS-485 signals and the same frames. *Re-read Sections 1.5 and 1.3.*

6. (a) MIPI CSI-2; the SoC sets the camera up over a slower I^2C-style control bus. (b) MIPI DSI. (c) LPDDR, low-power DRAM. (d) Wi-Fi, IEEE 802.11. (e) Bluetooth: Bluetooth Classic or the newer LE Audio. (f) PCIe or SDIO for the Wi-Fi data, and a UART that carries the Bluetooth HCI messages. (g) USB Power Delivery. (h) A control bus such as MIPI SPMI. (i) GNSS: the phone listens to the navigation satellites and works out its own position from their signals, but it never transmits to them. *Re-read Section 2.2.*

7. (a) CAN or CAN FD; LIN; SENT; a fast camera link such as GMSL or FPD-Link; automotive Ethernet on a single twisted pair (100BASE-T1 or 1000BASE-T1), carrying SOME/IP; an isolated daisy chain such as isoSPI to the battery-management unit, which reports on CAN; ISO 15118, on top of the IEC 61851 control pilot. (b) The central gateway, which joins the buses, passes messages between them and serves the diagnostics port. Pins 6 and 14 carry CAN; pin 16 brings battery power to the tester. The tool uses the UDS diagnostic services over CAN, and newer cars can also be diagnosed over Ethernet (DoIP). *Re-read Section 2.3.*

8. (a) $(14.4 - 4) / (20 - 4) = 0.65$ of the range, and $0.65 \times 50 = 32.5$ m^3/h. (b) The loop is broken: a cut wire or a dead transmitter. A working loop never goes below 4 mA, because 4 mA already means the bottom of the range, so 0 mA can never be a real reading. (c) HART; IO-Link (to an IO-Link master); PROFINET or EtherNet/IP; OPC UA (to the SCADA and HMI screens); MQTT. (d) At the bottom, messages are small and frequent and must arrive on time. Towards the top they grow larger, rarer and less urgent, so the protocols there look like office IT. An emer-

gency stop uses a safety protocol, such as PROFIsafe, CIP Safety or FSoE, which adds its own checks, counters and time-outs. *Re-read Section 2.4.*

9. (a) DDR5. (b) A processor link, such as Intel's UPI or AMD's Infinity Fabric. (c) PCIe. (d) NVLink. (e) CXL. (f) I^2C and SMBus for the sensors and fans; PMBus for the power supplies. (g) SPI. (h) Ethernet, at 25 to 400 Gbit/s. (i) BGP for routing; VXLAN for the virtual networks. (j) InfiniBand or RoCE (RDMA over Ethernet). (k) DWDM optics. (l) The BMC, a small computer of its own that keeps working when the server is switched off. The operator manages it remotely with Redfish or the older IPMI, over the separate management network, or over the server's own network card shared through NC-SI. *Re-read Section 2.5.*

10. (a) AFDX (ARINC 664 Part 7); ARINC 429; ACARS; SpaceWire; CCSDS; MAVLink; CRSF or SBUS; DShot. Book 39 teaches them all. (b) (i) Low cost for many slow devices: a simple bus with few wires, such as I^2C (Book 4). (ii) On-time delivery: fixed schedules with guaranteed timing, such as EtherCAT (Book 11). (iii) Noise, long cables and many nodes: a robust differential bus, CAN (Book 7). (iv) Reaching anyone, anywhere: the Internet stack, that is IP (Book 14), TCP or QUIC (Book 15), HTTP (Book

16) and TLS (Book 35). *Re-read Sections 2.6 and 2.7.*

11. (a) SWD: Board-level & embedded, Book 6. PROFINET: Industry, buildings, energy, Book 11. WireGuard: Security, Book 35. HBM: Memory & storage, Book 26. UCIe: Chip & system interconnect, Book 31. DNS: Internet networking, Book 16. FHIR: Special domains, Book 40. (b) In Book 7, the home book of CAN, which teaches it through every part of the dossier. Book 9 uses CAN and points to Book 7 with a cross-reference instead of teaching it again, so you learn each protocol once, in one place and in one style. (c) CAN: Tier A, several chapters (40–100 pages) with all 22 dossier parts and 2–4 labs; afterwards you can design with it, write its driver or RTL, capture it and debug it. Token Ring: a legacy protocol, Tier C, a section of 5–12 pages with the core parts and an optional lab; afterwards you can explain, recognise and decode it, and you know where its full details live. ISO 13485: Tier D, because it is a quality-management standard, not a protocol; a note of 1–2 pages with no lab says what it really is and where it is used. *Re-read Sections 3.2, 3.3 and 3.4.*

12. (a) P6, frame format. (b) P9, timing and performance (the timing maths). (c) P16, the debugging playbook. (d) P22, the cheat sheet. (e) P18, compare and choose. (f) Only P6 and P22. They are core parts, which even a Tier C section keeps, together with P0 to P2, P4, P5, P7, P10, P15 and P21. P9, P16 and P18 are not core parts, so a short section may leave them out. *Re-read Section 3.5.*

13. (a) At least 80 questions. A held-out quarter, 20 questions, goes to the blind test. Tier A and B chapters must score at least 95 %, so the reviewer needs 19 of 20. (b) 17 / 20 = 85 %, below 95 %: every missed question becomes new or clearer text, and the chapter is tested again. (c) So that the score measures what the chapter itself teaches, not what the reviewer already knew from elsewhere. (d) The author ran every desk lab with free tools, and every printed output is copied from a real run, never typed by hand, so you can run the same lab and compare (a bench lab, for your own hardware, is marked "not run by the author"). And every chapter lists numbered learning outcomes, each outcome points to the exercises that

practise it, and the final test has at least one question for every outcome, as this test does. *Re-read Section 3.6.*

14. (a) Power electronics & EV: 1 → 4 → 7 → 36 → 12. Books 1, 4 and 7 are in Wave 1, and Book 1 is out now. (b) ISO 15118 lives in Book 36, which is on the path; MQTT lives in Book 17, IoT Messaging, which they would add. Because every protocol has one home book, adding it repeats nothing. (c) Read Book 1 first, at least the chapters that Book 36's readiness check names: the protocol books build on Book 1's ideas and do not teach them from zero. (d) 84 % is below 90 %. Re-read the sections that the missed answers name, redo their exercises, and take the test again a few days later before moving on to Book 36. *Re-read Sections 3.7, 3.8, 3.9 and 3.10.*

15. (a) Topology: point-to-point, with one transmitter on each wire. Wires: TX, RX and ground, here with RTS and CTS. Max nodes: 2. Duplex: full, because TX and RX are separate wires. Sync / async: asynchronous, with no clock wire; each side has its own clock. Addressing: none. Error check: optional parity (none in 8N1), the stop-bit (framing) check, and break detection. (b) Microcontroller TX → module RX; module TX → microcontroller RX; microcontroller RTS → module CTS; module RTS → microcontroller CTS; GND to GND. (c) TX to TX makes two outputs drive one wire while nobody drives the other: nothing gets through, and two push-pull outputs fighting each other can be damaged. Without a ground wire, each receiver measures the signal against its own ground, so the voltage means nothing; join the grounds with a wire as short as the signal wires. (d) 5 V TX → 3.3 V module RX: 5 V is above the module pin's absolute maximum, so use a divider of 1 kΩ over 2 kΩ (3.33 V) or a level translator. 3.3 V module TX → 5 V RX: VIH = 0.7 × 5 V = 3.5 V, so a 3.3 V high has −0.20 V of margin and is not a valid 1. Add a buffer with TTL-level inputs, such as a 74HCT gate powered from 5 V (VIH = 2.0 V, so +1.30 V of margin). *Re-read Sections 4.1, 4.6 and 4.12.*

16. (a) 0x4B = 0100 1011, so D0 to D7 = 1 1 0 1 0 0 1 0, with four 1s. 8N1: 0 | 1 1 0 1 0 0 1 0 | 1, which is 10 bits. 7E1: 0 | 1 1 0 1 0 0 1 | 0 | 1, which is 10 bits; D0 to D6 hold four 1s, so the even-parity bit is 0. 8O2: 0 | 1 1 0 1 0 0 1 0 | 1 | 1 1, which is 12 bits; four 1s, so the odd-parity bit is 1. (b) The line idles high, and the receiver finds each frame by the falling edge of its start bit. 'K' ends with D7 = 0. Without a stop bit, the next start bit (also 0) follows that 0 directly: there is no falling edge, so the receiver cannot find where the next frame begins. The receiver also loses the one known value it checks in every frame, so it can no longer detect framing errors. *Re-read Sections 4.7 and 4.8.*

17. (a) T_{bit} = 1 / 38 400 = 26.04 µs. 8N1 has 10 bits, so T_{frame} = 260.4 µs. η = 8 / 10 = 80 %. At most 38 400 / 10 = 3840 characters per second. (b) 67 × 260.4 µs = 17.45 ms: four times faster than 69.79 ms, because the baud rate is four times higher. (c) N = 8 000 000 / (16 × 38 400) = 13.02, rounded to 13. The actual rate is 8 000 000 / (16 × 13) = 38 462 baud, and the error is 38 462 / 38 400 − 1 = +0.16 %, well inside the practical ±2 % per side: safe. (d) Each byte costs 10 bits at 8N1, so the link carries at most 3840 bytes of data per second, not 38 400. *Re-read Section 4.10 and Lab 4.2.*

18. (a) The receiver waits 8 ticks to reach the middle of the start bit, then samples every 16 ticks. D0: 8 + 16 = 24 ticks (1.5 bits). D5 is bit 6 of the frame (start = 0, D0 = 1), so 8 + 6 × 16 = 104 ticks. The stop bit is bit 9: 8 + 9 × 16 = 152 ticks (9.5 bits). (b) At tick 8, the middle of the sup-

posed start bit, the line is high again, so the receiver treats the edge as noise and goes back to waiting (false start-bit detection). Many UARTs also take three samples near the middle of each bit, for example at ticks 7, 8 and 9, and use the majority. (c) The 8E1 stop bit is bit 10, so its middle is 10.5 bits = 168 ticks after the edge. Ideal limit: 8 / 168 = 4.76 %. With one tick of start-edge uncertainty: 7 / 168 = 4.17 %. The error grows over the frame, and this last sample comes one bit later than in 8N1 (152 ticks, 5.26 % and 4.61 %), so less clock error fits. (d) 10.5 × 1.04 = 10.92 sender bit times. The stop bit runs from 10 to 11, so the sample is still inside, but only 0.08 bit from its end: the start-edge uncertainty (up to 1/16 bit) and slow edges leave almost no margin. 4 % is far above the practical ±2 % per side. *Re-read Sections 4.7 and 4.10.*

19. (a) A parity error, flag PE: one flipped bit makes the count of 1s odd. Two flipped bits keep the count even, so parity sees nothing and the receiver delivers a wrong byte with no flag. Parity catches only odd numbers of flipped bits, and it never says which bit is wrong. (b) One 8N1 frame lasts 86.81 µs, so 5 ms / 86.81 µs = 57.6: 57 whole bytes arrive. Only one fits in the buffer, so the UART sets OE (overrun error) and the later bytes are lost. Nothing is wrong on the wire; the software read too slowly. (c) A frame lasts 10 bits = 1.042 ms, and the receiver samples the stop bit 9.5 bits = 0.990 ms after the falling edge. Low for 1.0 ms (9.6 bits): the stop bit is still 0, so the receiver reports the byte 00 with a framing error (FE); the line was not low for longer than a whole frame, so there is no break. Low for 3.0 ms (almost three frames): first 00 with a framing error, then a break (BI), because the line stayed at 0 for longer than a whole frame, as in the break capture of Lab 4.1. *Re-read Section 4.11 and Lab 4.1.*

20. (a) 1 / 13.4 µs ≈ 74 600 baud. The nearest known rate is 74 880 baud, at which an ESP8266's boot ROM prints its log: the boot output simply runs at another baud rate. Ignore it, or decode it at its own rate. (b) 1 / 17.4 µs ≈ 57 500 baud, so the product sends at 57 600 baud, half the decoder's rate. Set both ends to the same rate. Measuring the shortest pulse is the playbook's most useful first step. (c) 8.7 µs is one bit at 115 200 baud, so the rate is right and the format is wrong: the decoder expects 7 data bits (7N1) while the product sends 8. The decoder reads D7, which is 0 in every ASCII letter, as the stop bit: the seven bits before it still give the right letters, but every byte has a framing error. Set 8 data bits. (d) The OE (overrun) flag in the line status register: the bytes arrived, but the software read them too slowly. Use the FIFO with interrupts or DMA and read in blocks, or turn on RTS/CTS flow control. *Re-read Sections 4.14 and 4.15.*

21. (a) RFC 9293: a specification, the TCP standard of the IETF. UM10204: a specification, the I^2C rule book that every maker follows, although one company publishes it as a user manual. PC16550D: a datasheet, one maker's document about one part; because the UART character format has no standard of its own, such data sheets are also where its behaviour is defined. SNOSD82D: a datasheet, revision D of the TMP117 data sheet. (b) Wi-Fi: IEEE (802.11); the Wi-Fi Alliance certifies products. LPDDR5: JEDEC. MIPI CSI-2: the MIPI Alliance. PCI Express: PCI-SIG. J1939: SAE. RS-232: TIA (TIA-232-F). AMBA: Arm. DNS: the IETF, in RFCs. Bluetooth Low Energy: the Bluetooth SIG. *Re-read Sections 5.1 and 5.2.*

22. (a) IEEE 802.3: free, through the sponsored IEEE GET Program. ISO 11898-1:2024: sold. The PCI Express Base Specification: members first; PCI-SIG members read it free, and others buy it. A 3GPP specification: free. TIA-232-F: sold (TI's design note SLLA037A gives a public sum-

mary). MIPI I3C: the full specification is for members only, but its subset MIPI I3C Basic is public. (b) IEEE Std 802.3-2022: the IEEE 802.3 (Ethernet) standard, in its 2022 revision; IEEE puts the year after a hyphen. V17.3.0: Release 17, after three rounds of approved technical changes and no editorial-only fixes. ISO 11898-1:2024: part 1 of ISO 11898, published in 2024; ISO puts the year after a colon. (c) RFCs never change; a new version gets a new number. RFC 9293 (August 2022, Internet Standard STD 7) obsoletes RFC 793, so the engineer cites a replaced document. Quote the rule from RFC 9293, with its section number, and check its errata. (d) The maker's errata sheet (silicon errata), which lists where the part differs from its own datasheet, with workarounds; also check that you hold the latest datasheet revision. *Re-read Sections 5.3 and 5.4.*

23. (a) 7.4.1, shall: a requirement that every conforming receiver must meet. 7.4.2, should: a recommendation, to be followed unless there is a valid reason and its consequences are understood. 7.4.3, may: a permission, truly optional. 7.4.4, can: a statement of what is possible, not a rule. The NOTE is informative and binds nothing; under ISO rules its "must" is also the wrong verb, because ISO keeps "must" for external constraints. (b) Yes. It meets the only requirement, 7.4.1. It does not follow the recommendation 7.4.2, so the supplier should be able to give, and document, a good reason. The NOTE asks for nothing. (c) No. Since RFC 8174 the key words have their special meaning only in capitals (MUST); a lowercase "must" is ordinary English, although the authors probably meant it. (d) Read the scope and the conformance clause first, then the terms and definitions, then only the technical clauses for your feature, searching for the exact term or symbol. Also follow the normative references, and check whether each annex you use is normative or informative. The conformance clause says what an implementation must do. *Re-read Sections 5.5 and 5.6.*

24. (a) The Specifications section holds the absolute maximum ratings (V+ from −0.3 V to 6 V), the recommended operating conditions (1.8 V to 5.5 V, nominal 3.3 V) and the electrical and timing characteristics, such as accuracy and conversion time. The Detailed Description holds the register map, with each register's address, access type and reset value. (b) 5.9 V is below the 6 V absolute maximum, so damage is not expected. But it is above the recommended maximum of 5.5 V, where nothing is promised: no reading taken then is guaranteed, and long exposure near the stress limit may affect reliability. Distrust those readings and fix the rail. (c) 0x8000 is the reset value: −32 768 × 7.8125 m°C = −256 °C, which means "no result yet", not a frozen sensor. 0xFE00 is −512 in two's complement: −512 × 7.8125 m°C = −4.0 °C. (d) RC means read to clear: reading the register clears the flag. If a debugger reads it, the flag is gone before the firmware looks, and the firmware misses the event. *Re-read Sections 5.7, 5.8, 5.9 and 5.11.*

25. (a) $R_{p(max)}$ = 300 ns / (0.8473 × 400 pF) = 885 Ω, and $R_{p(min)}$ = (5.0 − 0.4) V / 3 mA = 1533 Ω. The smallest allowed value is larger than the largest, so no resistor meets both limits: any value low enough for the rise time would need more than 3 mA of sink current. Ways out: lower the bus capacitance (shorter traces, fewer devices, or a split bus), or run the bus in Standard-mode, where $R_{p(max)}$ = 1000 ns / (0.8473 × 400 pF) = 2.95 kΩ, so 2.2 kΩ fits between 1533 Ω and 2.95 kΩ. (b) R_{slow} = (8 + 1) × 8 / (8 − 1 + 8 × 8 + 4) = 72 / 75 = 96.00 %, and R_{fast} = (8 + 2) × 8 / ((8 + 1) × 8 + 5) = 80 / 77 = 103.90 %. Both agree with the table. (c) 16 000 000 / 136 = 117 647 baud, and 117 647 / 115 200 − 1 = +2.12 %. That is outside the recommended ±1.5 %, so it is not

safe. Use a crystal that divides exactly, such as 18.432 MHz in normal-speed mode (UBRR = 9, 0.0 %), or a lower rate such as 38 400 baud. *Re-read Sections 5.9 and 5.10 and Lab 5.1.*

26. (a) Icarus Verilog (`iverilog` compiles, `vvp` runs), which writes a VCD file. sigrok (`sigrok-cli` with a protocol decoder), which reads the VCD after `vcd-for-sigrok` has cleaned it. Python 3 with scapy, which writes a pcap file. Wireshark's `tshark`, whose dissectors read the pcap. Python 3 again, with a package such as crcmod. (b) `board.sr` enters the signal pipeline at sigrok's decoder, with the same command; it skips the Verilog design, Icarus, the VCD and `vcd-for-sigrok`. `site.pcap` enters the packet pipeline at tshark; it skips the scapy script. The files and decoders are the same, so what works at the desk works on real hardware. (c) So that a script can run each tool, check its output and save it: that is how every lab output in the books comes from a real run. *Re-read Sections 6.2 and 6.3.*

27. (a) In PowerShell as administrator, run `wsl --install` and restart; it needs Windows 10 version 2004 or later, or Windows 11. Start Ubuntu and choose a Linux user name and password. Unpack the bundle under `/home/<you>`, not under `/mnt/c`. Then follow the Linux steps: `sudo apt update && sudo apt install python3-venv curl`; inside the bundle, `bash scripts/setup-tools.sh`, then `python3 -m venv .venv`, then `.venv/bin/pip install numpy scipy matplotlib scapy pyserial python-can crcmod`. In every new terminal, run `. scripts/tools-env.sh`; then `bash labs/00/ch06/bench-check/run.sh` should end with `PASS bench-check: 13 of 13 checks passed`. (b) (i) The venv module is missing: install `python3-venv`. (ii) Old package lists, or the universe archive is off: run `sudo apt update`, and `sudo add-apt-repository universe` if needed. (iii) Free some disk space, delete the partial file in `.tools/dl/`, and run the script again. (iv) The virtual environment is missing: make it with `python3 -m venv .venv`. (v) A new terminal does not keep yesterday's setting: run `. scripts/tools-env.sh` in it (`echo $EP_TOOLS` shows whether it is loaded). (vi) Files on the Windows drive are much slower to use from Linux: move the bundle under `/home/<you>`. *Re-read Sections 6.4 and 6.5 and Lab 6.1.*

28. (a) 193 810 000 ps = 193.81 µs. (b) One bit lasts 1 / 115 200 = 8.681 µs. From 193.81 to 202.49 µs: 8.68 µs, 1 bit of 0, the start bit. From 202.49 to 211.17 µs: 1 bit of 1, D0. From 211.17 to 245.89 µs: 34.72 µs, 4 bits of 0, D1 to D4. From 245.89 to 254.57 µs: 1 bit of 1, D5. From 254.57 to 271.93 µs: 17.36 µs, 2 bits of 0, D6 and D7. From 271.93 µs the line is high: the stop bit. D0 to D7 = 1 0 0 0 0 1 0 0, so D7...D0 = 0010 0001 = 0x21, the character '!'. (c) The start bit lasts 202 491 − 193 810 = 8681 samples = 8.681 µs, one bit time. The frame lasts 280 616 − 193 810 = 86 806 samples = 86.806 µs, ten bit times. (d) $N_{bit} = f_s / B$: 100 000 / 9600 = 10.4 samples per bit, which is enough, because a decoder needs at least about 4; 20 000 / 9600 = 2.1, which is too few. *Re-read Section 6.6 and Lab 6.2.*

29. (a) It stacks layers: each layer on the right becomes the payload of the layer on its left, so Ethernet carries IPv4, IPv4 carries UDP and UDP carries the DNS message. You give the fields that matter; scapy computes the lengths and checksums. (b) UDP Length = 8 + 45 = 53 bytes. IPv4 Total Length = 20 + 53 = 73 bytes. The whole packet is 14 + 73 = 87 bytes, because the 14-byte Ethernet header is in neither length field. (c) $S = 24 + (16 + 87) + (16 + 102) = 245$ bytes. (d) A value that Wireshark worked out, not bytes of the packet: it matched the answer to its

query by the transaction ID and measured the delay. (e) `-Y 'dns.flags.response == 1'`. *Re-read Section 6.7 and Lab 6.3.*

30. (a) A 3.3 V development board with USB (here the ESP32), an 8-channel USB logic analyzer that sigrok supports, a 3.3 V USB–UART adapter, a multimeter, a breadboard and jumper wires; for real hardware, also the distribution's sigrok packages with the `fx2lafw` firmware. The analyzer's input range must cover every logic level you probe, here 5 V if you probe the 5 V board's TX. (b) A divider of 1 kΩ over 2 kΩ, which brings 5 V down to 3.33 V, or a level shifter: 5 V can damage a 3.3 V pin, and Espressif gives the GPIO tolerance of its ESP chips as 3.6 V. (c) With the boards unpowered, connect the analyzer's GND lead first, to a ground pin next to the signal, then CH0. Touch grounded metal before you handle the boards, because electrostatic discharge can kill a pin without a visible mark. After power-up, check with the multimeter that TX idles at its logic-high level. (d) The relay's 230 V mains side. A USB analyzer shares its ground with the laptop and has no isolation, and mains voltage is lethal. (e) The only ground path is a long loop through two USB cables and the laptop. Noise on that loop moves the analyzer's reference, so edges near the threshold turn into extra bits. Always connect the GND lead, first, next to the signal. *Re-read Sections 6.8 and 6.9.*

APPENDIX C

Revision Cards

Each card is one chapter's key takeaways. Read them the day after you finish a chapter and again before the final test. If a line on a card feels unfamiliar, re-read that chapter.

1 What a Protocol Is (and Is Not)

- A protocol is a set of rules, agreed in advance, for the format, meaning, order and timing of messages and for what happens on errors.
- An interface is where devices connect; an electrical standard says how 0 and 1 look as voltages; a network standard is a document that defines a whole network family; a data format writes values as bytes; a security algorithm hides or signs data.
- RS-485 is an electrical standard, USB-C is an interface, Wi-Fi products follow the network standard IEEE 802.11, JSON is a data format and AES is a security algorithm; none of them is "a protocol" on its own. A CRC is an error-check algorithm: it catches noise, not attackers.
- In the greenhouse link, 24.5 °C becomes 245 (00 F5), a Modbus RTU frame with a CRC-16, 11-bit UART characters at 9600 bit/s, and voltage differences on an RS-485 pair.
- Every protocol answers five questions: who, when, how fast, what format, and what if it fails.
- A polled protocol such as Modbus RTU avoids collisions because a server speaks only when asked; a damaged request gets no answer, and the client times out and retries.
- Layers split the work; encapsulation wraps each layer's data with the next layer's header and checks.
- The Protocol Passport answers the five questions in 16 fixed fields for every protocol in this library.

2 Where Protocols Live

- Every product is a set of links, and each link uses a protocol chosen for its distance, speed, cost, power budget and safety needs.
- Inside a phone: LPDDR, UFS, MIPI CSI-2 and DSI, I^2C/I3C/SPI, I^2S, SPMI, RFFE, PCIe or SDIO with UART, ISO/IEC 7816 and USB with USB PD; its radios: GNSS, 4G/5G, Wi-Fi, Bluetooth, NFC, Qi, UWB.
- In a car: CAN and CAN FD for control units, LIN for simple motors, SENT for sensors, automotive Ethernet for cameras, screens and the backbone, and OBD-II diagnostics on CAN pins 6 and 14.
- In a factory: 4–20 mA and HART, IO-Link, PROFINET or EtherNet/IP, EtherCAT, safety protocols, Modbus, OPC UA and MQTT, from the field level up to the cloud.
- Inside a server: DDR5, UPI or Infinity Fabric, PCIe with NVMe and CXL, NVLink, SPI flash, and I^2C, SMBus and PMBus for the BMC; between servers: Ethernet, BGP, VXLAN and RDMA networks.
- In the sky: ARINC 429 and AFDX in airliners, ADS-B and ACARS to the ground, SpaceWire, MIL-STD-1553 and CCSDS in space, MAVLink and CRSF or SBUS on drones.
- A few protocols, such as I^2C, CAN, Ethernet, PCIe and USB, appear in many places: learn them first.

3 The Library Map

- The Protocol Library is this free guide and forty books, Book 1 to Book 40, on thirteen shelves, from Foundations to Special domains.
- Each protocol has one home book that teaches it in full; other books use it and point there.
- Depth tiers: A gets several chapters, B one chapter, C a section with the core parts, and D (not a protocol) a short note.
- Every protocol follows the 22-part Protocol Dossier in a fixed order, opened by the Passport (P0).
- Every chapter has outcomes, checks, worked examples, a desk lab with real output, exercises, answers at the back and a final test for the book.
- A question universe is written first, and Tier A and B chapters must pass a blind test at 95 % or better.
- Book 1, Digital Communication Foundations (974 pages, 22 chapters, 71 labs), is out now and is the first stop on every study path.
- Eight study paths give a reading order per career; the books appear in three waves, essentials first.

4 UART in One Chapter: A Complete Sample Dossier

- A UART sends bytes on one wire per direction with no clock wire; the two devices agree on the baud rate and format in advance.
- Wiring is TX → RX, RX ← TX and a common ground; RTS/CTS are optional flow-control wires.
- A logic-level UART idles high (1, mark); RS-232 inverts the levels and uses ±3 to ±15 V at the receiver.
- A frame is a start bit (0), 5 to 8 data bits sent LSB first, an optional parity bit, and 1 or 2 stop bits (1); 8N1 is 10 bits long.
- Bit time = 1 / baud: 104.17 µs at 9600 and 8.68 µs at 115 200; 8N1 is 80 % efficient, so 115 200 baud carries 11 520 bytes/s.
- The receiver finds the falling edge, waits half a bit, checks the start bit, then samples every bit in its middle, usually with 16× oversampling.
- The stop-bit sample is 9.5 bits after the edge, so the total clock mismatch must stay below about 5 %; keep each side within about ±2 % and check the divisor rounding error.
- Errors: parity (wrong count of 1s), framing (stop bit 0), overrun (byte lost because software was slow) and break (line low for more than a frame).
- Debug by measuring the bit width, then checking format, level inversion, wiring and ground; a pseudo-terminal tests serial programs but never bit timing.

5 Reading Specifications and Datasheets

- A specification is the rule book that every product of a kind follows; a datasheet is one maker's promise about one part.
- Standards come from ISO, IEC, ITU, IEEE, the IETF (RFCs), 3GPP, JEDEC and industry groups.
- Specifications are free (RFCs, 3GPP, ITU-T), sold (ISO, IEC) or members first (PCI-SIG, MIPI); free to read is never free to copy.
- Quote every number with its document, edition and clause; check errata and amendments.
- shall/MUST is required, should/SHOULD recommended, may/MAY optional; NOTEs are informative.
- Absolute maximum ratings are damage limits; characteristics hold only inside the recommended operating conditions. Design with min and max, never typ, and recompute key numbers.

6 Your Protocol Bench

- The free desk bench: Python 3 with its packages, Icarus Verilog (OSS CAD Suite), `sigrok-cli` and `tshark`.
- Signals: Verilog → VCD → `vcd-for-sigrok` → decoder; packets: scapy → pcap → tshark. Real captures use the same files.
- Install with `scripts/setup-tools.sh` and a `.venv` (Debian, Ubuntu or WSL 2); source `tools-env.sh` in every terminal.
- In a VCD, `#t` then `0!` means: at time t, ! became 0. Decoders need 4 or more samples per bit.
- UDP Length = 8 + payload; IPv4 Total Length = 20 + UDP Length; pcap = 24 + $\Sigma(16 + L)$.
- Kit: 3.3 V board, 8-channel analyzer, 3.3 V USB–UART adapter, wires, meter; ground first, never RS-232 or mains.

APPENDIX D

Glossary

Every term this guide explains, in alphabetical order. Terms that begin with a digit come first. The number after each definition is the chapter that explains the term first. Book 1 explains all of these terms again in more depth, and every book of the series adds its own glossary.

0–9

16550 UART — the classic PC UART, described by National Semiconductor's PC16550D data sheet, with 16-byte FIFOs and a line status register; the UART chapter follows its behaviour. 4

16× oversampling — a receiver clock at 16 times the baud rate, so the receiver finds the start edge to within 1/16 of a bit and samples each bit near its middle. 4

3GPP (3rd Generation Partnership Project) — the partnership that writes the mobile standards, from GSM to 5G NR, for seven regional partners such as ETSI (Europe) and TSDSI (India). 5

4–20 mA loop — the current loop on which a process sensor sends its value on two wires, 4 mA for the bottom of the range and 20 mA for the top; 0 mA means a broken loop. 2

5 V tolerant — said of an input that its data sheet allows to receive 5 V although the chip runs from a lower supply. 4

7E1 — the UART frame with 7 data bits, even parity and 1 stop bit, 10 bits on the wire, used for 7-bit ASCII text with a parity check. 4

8E1 — the UART character format with 8 data bits, even parity and 1 stop bit, 11 bits in all; the default format of Modbus RTU. 1

8N1 — the UART frame with 8 data bits, no parity and 1 stop bit: 10 bits on the wire and 80 % efficient, the default almost everywhere. 4

8O2 — the UART frame with 8 data bits, odd parity and 2 stop bits, 12 bits on the wire; rare. 4

A

Absolute maximum ratings — the stress limits of a part, such as the highest voltage a pin can take, beyond which it may be damaged; the part is not promised to work at them. 4

ACARS — the link that carries short text messages, such as flight plans, weather and engine reports, between an aircraft and its airline by VHF radio, HF radio or satellite. 2

Access type — how software may use a register or field: R (read only), W (write only), R/W (read and write) or RC (read to clear: reading it clears it). 5

ADS-B — the aircraft's continuous broadcast of its position, height and speed on 1090 MHz, received by ground stations and other aircraft. 2

AES (Advanced Encryption Standard) — a security algorithm, an encryption method, which security protocols such as TLS and WPA3 use with their keys. 1

AFDX — Airbus's name for the switched Ethernet backbone with guaranteed timing defined in ARINC 664 Part 7, used in the Airbus A380 and A350 and the Boeing 787. 2

Amendment — a published change that adds to or changes the technical content of a standard. 5

Annex — a part at the end of a standard, marked either normative or informative. 5

ARINC 429 — the airliner data bus on which one transmitter, such as the air data computer, sends values to several receivers on a twisted pair. 2

ASCII — a data format, published in 1963, that gives each text character a number, such as 0x48 for 'H'. 1

Asynchronous — said of a link on which no clock signal travels with the data: each side uses its own clock, and the receiver finds the timing from the data itself. 4

AT commands — a text command language, begun with telephone modems, whose short commands start with AT and end with a carriage return and line feed, answered by OK or ERROR. 4

Automotive Ethernet — Ethernet on a single twisted pair, 100BASE-T1 at 100 Mbit/s or 1000BASE-T1 at 1 Gbit/s, for driver assistance, screens, telematics and the car's backbone. 2

B

Basic, Intermediate and Advanced — the levels of a section; read every Basic and Intermediate section on a first pass, while Advanced sections can wait. 3

Baud-mismatch budget — the largest total difference between the two clocks that still lets a receiver read every frame: for 8N1, 5.26 % in theory and 4.6 % with 16× sampling; keep each side within about ±2 %. 4

Baud rate — the number of symbols per second on a line; on a UART, the number of bits per second, counting the start, parity and stop bits. 4

Baud-rate generator — the UART block that divides a clock to make a tick, usually 16 ticks per bit. 4

Bench — the tools that capture or make a signal and explain it: a probe or lead listens, a capture file keeps the recording, a decoder names every field, and you explain the result. 6

Bench lab — a recipe for your own hardware, marked not run by the author, which describes what you should see. 3

BGP and VXLAN (Border Gateway Protocol, Virtual Extensible LAN) — data-centre network protocols: switches route traffic with the IP routing protocol BGP and build separate virtual networks for different customers with VXLAN. 2

Bit time — the length of one bit on the line, 1 ÷ baud rate: 104.17 µs at 9600 baud and 8.68 µs at 115 200 baud; a frame lasts its number of bits times the bit time. 4

BLE (Bluetooth Low Energy) — the Bluetooth radio that saves battery by sleeping between short bursts, used by watches and fitness bands. 2

Blind test — the check in which a fresh reviewer who sees only the finished chapter answers a held-out quarter of its question universe; Tier A and B chapters must score at least 95 %. 3

Bluetooth Classic and LE Audio — the two ways Bluetooth carries music and calls to earbuds: the older Bluetooth Classic and the newer LE Audio. 2

Bluetooth SIG — the group that writes the Bluetooth specifications; its Core Specification is free to read, but non-members may not use it to build products. 5

BMC (baseboard management controller) — a small computer of its own inside a server, which reads its sensors, controls its power supplies and lets operators manage it remotely, even when the server is switched off. 2

Break — the line held at 0 for longer than a whole frame (flag BI); it is not a valid character, so it is used to get attention, reset a link or mark the start of a packet. 4

C

CAN and CAN FD (Controller Area Network) — the workhorse bus of a car, a differential bus on two twisted wires on which any control unit may send short messages when the bus is free; CAN FD is its faster successor. 2

Capture file — a file that keeps the recording of a wire (samples) or of a network (packets), such as a VCD, .sr or pcap file. 6

CCSDS (Consultative Committee for Space Data Systems) — the space-data standards with which space agencies send telemetry down and commands up. 2

Cheat sheet — dossier part P22: one page with a protocol's key rules and numbers, for quick review. 3

Checksum — a value computed from a message and sent with it, so that the receiver can repeat the calculation and catch accidental damage; like a CRC, it is an error-check algorithm. 1

Client — in Modbus, the device that sends the requests and starts every exchange, such as the greenhouse controller. 1

Code bundle — the folders `labs/` and `scripts/` that come with this guide, which you unpack in your home folder. 6

Conformance clause — the clause of a standard that says what an implementation must do to conform. 5

Console — a UART on which a product prints its boot messages and takes commands, so an engineer can watch it start even when the network is down. 4

Core parts — the dossier parts that even a Tier C section keeps: P0, P1, P2, P4, P5, P6, P7, P10, P15, P21 and P22. 3

Corrigendum — a published correction of errors in a standard, with no new technical content. 5

CRC (cyclic redundancy check) — an error-check algorithm whose result is sent at the end of a frame; it catches noise, but anyone who changes the data can compute a new, correct CRC. 1

CRC-16/MODBUS — the 16-bit CRC that ends every Modbus RTU frame, computed over all the bytes before it and sent low byte first. 1

CRSF and SBUS — serial protocols with which a radio-control receiver passes the pilot's stick commands to a drone's flight controller. 2

CSMA/CD (carrier-sense multiple access with collision detection) — a medium-access method, not a protocol: stations listen before sending, detect collisions and back off; classic Ethernet used it. 3

CXL (Compute Express Link) — a link that runs on the same wires as PCIe and lets a CPU use extra memory on an add-in card. 2

D

Data format — a rule for how a value or a text is written as bytes, such as JSON, ASCII or the reading 245 sent as 00 F5; it says nothing about how the bytes travel. 1

Datasheet (data sheet) — one maker's document about one part: its pins, its guaranteed limits and how to use it. 5

DDR5 — the memory interface over which a server's CPU reads its main memory, the DIMMs. 2

Debugging playbook — dossier part P16: a table of symptoms, their likely causes, what to measure and the fixes. 3

Decoder — software that turns a recording of a wire into bits, bytes and named fields, such as sigrok's uart decoder. 6

Depth tier — the depth a protocol gets, Tier A to Tier D, set in the register before its book is written. 3

Desk bench — the set of free programs that this library uses on a normal computer, with no hardware: Python 3, Icarus Verilog, sigrok and Wireshark. 6

Desk lab — a lab that runs on an ordinary computer with free tools; the author ran every one, and every output printed in a book is copied from a real run. 3

Detailed Description — the datasheet section that holds the block diagram, the modes, the programming and the register map. 5

Differential signalling — sending each bit as the voltage difference between two wires, so that noise which hits both wires alike, or a long cable that loses voltage, does no harm. 1

Digital Communication Foundations — Book 1 of the series, which teaches once and completely the ideas every protocol is built from: signalling, clocks, line codes, CRCs, framing, access and tools. 3

Display filter — a Wireshark expression, such as `dns.flags.response == 1`, that keeps only the packets you ask for. 6

Dissector — the part of Wireshark that decodes one protocol layer of a packet into named fields. 6

Divisor — the whole number N by which a baud-rate generator divides its clock; because N must be whole, the actual baud rate is often a little off. 4

DLAB (divisor latch access bit) — bit 7 of the 16550's LCR, which switches addresses 0 and 1 between the data registers and the baud-rate divisor latches. 5

DNS (Domain Name System) — the Internet's name-lookup protocol, which answers questions such as what the IPv4 address of a name is. 6

Double-speed mode (U2X = 1) — the AVR USART mode with 8 samples per bit instead of the 16 of normal-speed mode. 5

DShot — a protocol with which a drone's flight controller drives each motor's speed controller. 2

DWDM (dense wavelength-division multiplexing) — optics that send many wavelengths of light down one fibre, as on the links between data centres. 2

E

ECU (electronic control unit) — one of the dozens of small computers in a car, for the engine, brakes, airbags, doors, seats, lights or screens. 2

Edition — a new version of a standard that replaces the whole document. 5

Efficiency — the share of line time that carries data, data bits ÷ frame bits: 80 % for 8N1, so 115 200 baud at 8N1 carries 11 520 bytes of data per second. 4

Electrical characteristics — the datasheet table of guaranteed performance, such as accuracy, currents and thresholds, which holds only under the operating conditions in the table's header. 5

Electrical standard — a rule set for how a 0 and a 1 look as voltages or currents, with thresholds and sometimes cable and distance, such as RS-485; it says nothing about what the bits mean or who talks when. 1

Encapsulation — each layer taking the data of the layer above, adding its own parts (a header in front, sometimes a check at the end) and handing the result down. 1

Errata — published lists of corrections to a document; a chip maker's errata sheet (silicon errata) lists where a part differs from its own datasheet, with workarounds. 5

Error-check algorithm — a calculation, such as a checksum or a CRC, that catches accidental damage to data but gives no protection against attackers. 1

ESD (electrostatic discharge) — the spark from a charged body, which can kill a pin without a visible mark; touch grounded metal first. 6

EtherCAT — an industrial Ethernet protocol for fast motion control, in which one frame passes through every device and each device reads and writes its own part on the fly. 2

Ethernet — the wired network family of IEEE 802.3, which this guide meets in cars, factories, data centres and aircraft. 1

Even and odd parity — the two kinds of parity bit, chosen so that the data bits and the parity bit together hold an even, or an odd, number of 1s. 4

externally-managed-environment — the error with which pip on current Debian and Ubuntu refuses to install packages into the system's Python; install into a virtual environment instead. 6

F

False start-bit detection — the receiver's check, half a bit after a falling edge, that the line is still low; if it is high again, the edge was noise. 4

Fast-mode — an I^2C speed mode in which UM10204 allows a rise time of at most 300 ns and a bus capacitance of at most 400 pF. 5

FIFO (first in, first out) — a small queue of bytes inside a UART; a 16550's FIFOs hold 16 bytes, and in FIFO mode each error flag travels with its byte. 4

Final test — the test at the back of every book, with at least one question for every learning outcome; 90 % or more means you are ready for the next book. 3

Flow control — a way for a receiver to make the sender pause, with RTS/CTS wires or with XON/XOFF characters. 4

Format (syntax) — the kind of protocol rule that fixes which parts a message has, how long each part is and in what order the bits go. 1

Frame — on a UART, one character together with its start, parity and stop bits. 4

Framing error — the receiver read the stop bit as 0 (flag FE), usually because of a wrong baud rate or format, noise or a break. 4

Full duplex — both sides can send at the same time, as on a UART, whose TX and RX are separate wires. 4

fx2lafw — sigrok's free firmware, which it loads into many low-cost logic analyzers when it finds them. 6

G

Gateway — in a car, the central unit that joins the buses, passes messages between them and serves the diagnostics port. 2

GMSL and FPD-Link — fast serial links that carry raw video from a car's cameras to the driver-assistance computer. 2

GNSS (global navigation satellite system) — navigation satellites such as GPS, Galileo, GLONASS, BeiDou and India's NavIC; a receiver works out its own position from their signals and never transmits to them. 2

Guaranteed by design — a datasheet footnote that means the maker does not test that limit on every part. 5

H

HART (Highway Addressable Remote Transducer) — a small digital signal added to a 4–20 mA loop, on the same two wires, for setup and diagnostics. 2

HCI (host controller interface) — the messages that travel between a Bluetooth chip and the host processor, often over a UART. 2

Home book — the one book that teaches a protocol in full, through every part of the dossier; any other book that needs it points there with a cross-reference. 3

HTTP (Hypertext Transfer Protocol) — the application protocol of web requests. 2

I

I²C (inter-integrated circuit) — a slow, cheap two-wire bus (SDA and SCL) with addresses, on which a controller drives the clock and many chips share the same wires; the newer I3C follows on from it. 2

I²S and TDM (inter-IC sound, time-division multiplexing) — the links that carry digital audio samples between a processor and its audio chips; some phones use MIPI SoundWire instead. 2

Icarus Verilog — a free simulator, part of the OSS CAD Suite bundle, that runs a Verilog design with its test-bench (`iverilog` compiles, `vvp` runs) and writes a VCD file. 6

IEEE — a professional association that writes standards such as Ethernet (802.3) and Wi-Fi (802.11). 5

IEEE 802.11 — the network standard that Wi-Fi products follow, with several radio physical layers, the frame formats, the rules for sharing the air and the joining and security procedures. 1

IETF (Internet Engineering Task Force) — the body that writes the Internet protocols and publishes them as numbered RFCs. 5

InfiniBand and RoCE (RDMA over Converged Ethernet) — very low-delay networks for AI clusters that let one server write directly into another server's memory (RDMA). 2

Interface — the boundary where two things connect: in hardware a set of pins or terminals and their signals, often with a connector; in software the set of functions one part of a program offers to another. 1

Interface control document — the document in which two companies that connect their equipment write down the speeds, character format, addresses, register map and time-outs. 1

IO-Link — a point-to-point digital link between an IO-Link master and one smart sensor or actuator, on the usual three-wire sensor cable. 2

IP (Internet Protocol) — the network layer of the Internet, which carries data across many networks. 1

IPv4 Total Length — the IPv4 header field that counts the 20-byte IPv4 header plus everything after it, but not the Ethernet header. 6

ISO 15118 — the digital messages between an electric car and a charging station, on top of the simple IEC 61851 control pilot, with which, for example, the car identifies itself and pays automatically. 2

ISO and IEC (International Organization for Standardization, International Electrotechnical Commission) — the formal standards bodies that work through national members such as BIS (India), DIN (Germany) and ANSI (USA). 5

ISO/IEC 7816 — the smart-card protocol spoken by a SIM card or by the eSIM chip soldered to a phone's board. 2

isoSPI — Analog Devices' isolated daisy chain, along which battery-monitoring chips pass every cell's reading to the battery-management unit. 2

ITU (International Telecommunication Union) — the formal body that writes telecom Recommendations (ITU-T) and radio rules (ITU-R). 5

J

JEDEC — the industry group that writes memory standards such as DDR5 and LPDDR. 5

JSON — a data format that writes values as text; an application protocol such as HTTP or MQTT carries it. 1

L

Layer — one level of a communication system that solves one problem, such as turning bits into voltages or adding addresses, using the layer below and offering a service to the layer above. 1

LCR (line control register) — the 16550 register that sets the frame format: bits 1–0 the data bits, bit 2 the stop bits, bits 3 and 4 the parity and bit 7 the DLAB; 8N1 is 0x03 and 7E1 is 0x1A. 5

Learning outcome — a numbered statement of what you will be able to do after a chapter; each outcome points to the exercises that practise it, and the final test has at least one question for it. 3

Level translator (level shifter) — a chip that converts a signal from one logic level to another; the cleaner choice at high speed. 4

LIN (Local Interconnect Network) — a cheap, slow car bus on one signal wire (at most 20 kbit/s), on which one master unit controls simple devices such as window, mirror and seat motors. 2

Line status register (LSR) — the 16550 register that reports the receiver's conditions, one bit for each flag: DR (data ready), OE, PE, FE and BI. 4

Link — one connection that carries data between two or more devices, such as the wires between a processor and its memory, a cable between two units or a radio channel. 2

Logic analyzer — an instrument that samples digital lines through its leads and saves the samples in a capture file for a decoder. 6

Logic level — the voltages a chip uses on its own pins, such as about 3.3 V for a 1 and 0 V for a 0 on a 3.3 V microcontroller. 4

LoRaWAN — a long-range, low-power radio protocol for battery devices that send small amounts of data, such as field sensors. 2

LPDDR (low-power double data rate) — the low-power DRAM interface of phones, which moves data on both edges of a fast clock over many parallel wires. 2

LSB first (least significant bit first) — the bit order of a UART: D0 is sent first and D7 last. 4

M

Majority vote — taking three samples near the middle of a bit, such as ticks 7, 8 and 9 of 16, and keeping the value most of them show, so one noise spike cannot change the bit. 4

Mark and space — the old telegraph names for a 1 (mark), the idle level of a UART line, and a 0 (space). 4

MAVLink (Micro Air Vehicle Link) — the protocol for telemetry and commands between a drone's flight controller and its ground-control software, over a telemetry radio or a network. 2

MAY (OPTIONAL) — a truly optional feature; ISO, IEC and IEEE standards write lowercase may for a permission. 5

Meaning (semantics) — the kind of protocol rule that says what each part of a message says and what the receiver must do about it. 1

Members first — said of a specification that members of a group read free, while others buy it or use public material, such as the PCI Express Base Specification. 5

Message authentication code — a check value computed with a secret key, which proves that a message was not changed; unlike a CRC, an attacker without the key cannot make a new one. 1

MIL-STD-1553 — a bus that began in military aircraft, first across a whole aircraft in the F-16, and is used on many spacecraft. 2

Min, typ and max — the columns of a datasheet table: min and max are guarantees, while typ is a value measured on typical parts and is never a promise. 5

MIPI Alliance — the group that owns many phone interfaces, such as CSI-2, DSI and I3C; its full I3C specification is for members only. 5

MIPI CSI-2 and DSI — the fast serial links on a few wire pairs (MIPI D-PHY or C-PHY) that carry pictures from a camera to the SoC (CSI-2) and from the SoC to the display (DSI). 2

MIPI SPMI and RFFE — two control buses of a phone: over SPMI the SoC tells its power-management chip which voltages to make, and over RFFE the modem sets the amplifiers, switches and filters near the antennas. 2

Modbus RTU — a protocol from 1979, still used in meters, drives and sensors, in which a client polls servers with frames of address, function, data and a CRC-16, usually over RS-485. 1

Modbus TCP — the Modbus part of a message carried inside Ethernet and TCP, with the RTU address and CRC replaced by the Modbus TCP header and the checks of Ethernet and TCP. 1

MQTT — a light publish-and-subscribe protocol, often used to send data to company servers or the cloud. 2

MUST (SHALL, REQUIRED) — in an RFC, an absolute requirement; ISO, IEC and IEEE standards keep lowercase must for external constraints such as laws of nature. 5

N

NC-SI (Network Controller Sideband Interface) — the link that lets a BMC share the server's own network card. 2

Network standard — one document that defines a whole network family, with its physical layers, frames, access rules and management, such as IEEE 802.11 or IEEE 802.3. 1

NFC (near-field communication) — a radio link that works over a few centimetres, used to pay with a tap. 2

NMEA 0183 — the text format of GPS receivers: sentences that start with `$`, hold comma-separated fields and end with `*`, a two-digit checksum (the XOR of the characters between `$` and `*`) and a carriage return and line feed, at most 82 characters long. 1

Normative and informative — normative text is binding, the requirements you must meet to conform; informative text, such as introductions, notes, examples and informative annexes, only helps you understand and never contains a requirement. 5

Normative references — the other documents that a standard says you must also follow. 5

NOTE — a note in a standard; it is informative, so it explains but never contains a requirement. 5

NVLink — the fast, dedicated link over which GPUs exchange data with each other. 2

NVMe (Non-Volatile Memory Express) — the SSD's own command protocol, with its commands, queues and data transfers, carried over PCIe. 2

O

OBD-II (on-board diagnostics) — the car's 16-pin diagnostics socket under the dashboard; in US cars since the 2008 model year it carries CAN diagnostics on pins 6 and 14, and pin 16 brings battery power to the tester. 2

Obsoletes and updates — the RFC header lines that name the older RFCs a new RFC replaces (obsoletes) or changes (updates). 5

OPC UA (Open Platform Communications Unified Architecture) — the protocol with which a PLC shares its data, together with what each value means, with SCADA and HMI screens and plant software. 2

OSI and TCP/IP models — the two best-known layer models: the seven-layer OSI (Open Systems Interconnection) model and the four-layer TCP/IP model of the Internet. 1

Overrun error — a new byte finished arriving while the receive buffer was still full, so a byte was lost (flag OE); nothing is wrong on the wire, the software read too slowly. 4

P

Parity bit — an optional bit after the data bits that makes the number of 1s even or odd, as a simple error check. 4

Parity error — the count of 1s in the data and parity bits does not match the agreed parity (flag PE); parity catches odd numbers of flipped bits only. 4

Payload — the data a layer carries; in scapy, each layer on the right of a `/` becomes the payload of the layer on its left. 6

pcap — a binary file of network packets, each with a time stamp; it adds a 24-byte file header and a 16-byte record header before each packet. 6

PCIe (PCI Express) — the point-to-point lanes that link a CPU to GPUs, network cards and SSDs: the main expansion link of every server. 2

PCI-SIG — the group that writes PCI Express; its members read the Base Specification free, and others buy it. 5

PLC (programmable logic controller) — the controller in a factory's control cabinet that reads the inputs and drives the outputs of a machine. 2

Point-to-point — a topology that joins exactly two devices, as a UART does with one transmitter on each wire. 4

Polling — a way of sharing a line in which a client asks one server at a time and a server speaks only when asked, so two devices never talk at once; a GPS module instead pushes its data whenever it has some. 1

PROFINET and EtherNet/IP — industrial Ethernet protocols that carry inputs and outputs between a PLC, remote I/O, drives and IO-Link masters on a fixed cycle. 2

Protocol — a set of rules, agreed in advance by every side, that says how messages are formatted, what they mean, in what order they are exchanged, with what timing, and what each side does when something goes wrong. 1

Protocol Dossier — the fixed plan of 22 parts, P1 to P22, opened by the Passport (P0), with which every protocol in the library is taught, from why it exists to its cheat sheet. 1

Protocol Library — the series of this free guide and forty books, Book 1 to Book 40, on thirteen shelves, which teach the electronics and communication protocols an engineer meets. 3

Protocol Passport — the one-page card of 16 fields, always in the same order, that opens every protocol in the library and answers the five questions together with the protocol's identity. 1

Protocol tree — Wireshark's view of one packet as nested layers and their fields. 6

Pseudo-terminal (pty) — a pair of software serial ports inside the operating system; it copies bytes but never times bits, so it cannot show baud errors, framing, parity or breaks. 4

PTP (Precision Time Protocol) — the protocol that keeps the clocks of devices in step, in factories to well under a microsecond. 2

Pull-up resistor — a resistor from a line to the supply that holds the line high while nothing drives it low, such as on a UART's RX input or on an I^2C bus. 4

Python 3 — the desk bench's language for the maths and for building packets; the labs call it as `$EP_PYTHON`. 6

Q

Question universe — the questions collected before a protocol is written, at least 200 for Tier A, 80 for Tier B and 30 for Tier C; the chapter is written to answer all of them. 3

QUIC — a transport protocol that, like TCP, carries traffic to Internet services. 2

R

Readiness check — the short check at the start of every protocol book that names the Book 1 chapters it needs. 3

Receiver operating range — the slowest and fastest incoming data rates a receiver accepts, as ratios to its own baud rate; the AVR data sheets compute them as Rslow and Rfast. 5

Recommended operating conditions — the datasheet range in which a part is meant to run, such as 1.8 V to 5.5 V for the supply of TI's TMP117 sensor; outside it nothing is promised. 5

Redfish and IPMI — the protocols, Redfish the newer and IPMI the older, with which operators manage a server remotely through its BMC. 2

Reference manual — a microcontroller document that describes every peripheral and register, such as ST's RM0008 for the STM32F1 series. 5

Register (in a chip) — a small numbered storage place in a chip, whose bit fields set a mode or report a status. 5

Register (of the library) — the list of every protocol the series covers, with its home book, its depth tier and the specification that defines it. 3

Register map — the part of a manual or datasheet that lists a chip's registers, each with its address, name, access type and reset value, followed by a table of every bit field. 5

Release — in 3GPP, the x of an approved version x.y.z, so V18.y.z belongs to Release 18; y counts approved technical changes and z editorial fixes. 5

Reset value — the value a register holds after reset, such as 8000h in the TMP117 result register, which reads as −256 °C. 5

Resistor divider — two resistors that scale a voltage down, such as 1 kΩ over 2 kΩ, which turn a 5 V signal into 3.33 V for a 3.3 V input. 4

Revision — the version of a datasheet, such as the D at the end of TI's literature number SNOSD82D; a revision history lists every change. 5

RFC (Request for Comments) — a numbered IETF document; an RFC never changes, so a new version gets a new number. 5

RFC 2119 — the RFC that defines the key words MUST, SHOULD and MAY for Internet protocols; RFC 8174 adds that they have this meaning only in capitals. 5

Rise time — the time an edge takes to rise between two levels; on I^2C, from 30 % to 70 % of VDD, at most 300 ns in Fast-mode. 5

RS-232 (TIA-232-F) — an electrical standard with inverted, bipolar levels: a driver sends a 0 as +5 to +15 V and a 1 as −5 to −15 V, for about 15 to 20 m of cable. 4

RS-485 (TIA-485-A) — an electrical standard that sends each bit as the voltage difference between wires A and B, B above A for a 1 and A above B for a 0; a driver makes at least 1.5 V and a receiver needs only 0.2 V, so it reaches about a kilometre. 1

RTS and CTS (request to send, clear to send) — the optional hardware flow-control wires of a UART: each side drives its RTS to say it can receive, and a transmitter sends only while its CTS input is active. 4

S

SAE — the trade association that writes vehicle standards such as J1939. 5

Safety protocol — a protocol, such as PROFIsafe, CIP Safety or FSoE, that carries emergency stops and light-curtain signals over the normal factory network and adds its own checks, counters and time-outs. 2

Sample number — the count of samples from the start of a capture; dividing it by the sample rate gives the time. 6

Sample rate — the number of samples per second (S/s), a sample being one reading of a line's level; a decoder needs at least about 4 samples per bit. 6

SCADA and HMI (supervisory control and data acquisition, human–machine interface) — the control-room software and operator screens that show a plant's values and take commands. 2

scapy — a Python package that builds packets by stacking layers with `/` and writes them to pcap files. 6

Scope — the clause of a standard that says what the document covers. 5

SDIO (Secure Digital input/output) — an interface over which a Wi-Fi chip can move its data to the SoC instead of PCIe. 2

Security algorithm — a calculation that hides data (encryption) or proves who sent it and that it was not changed, such as AES; it has no messages, turns or timing of its own. 1

SENT (Single Edge Nibble Transmission) — a simple one-way protocol on a single signal wire, with which sensors such as pedal-position and pressure sensors send their readings to their ECU. 2

Sequence chart — a drawing of the order of messages in time, from top to bottom, with one vertical line for each device. 1

Server — in Modbus, a device with an address from 1 to 247 that answers only the requests sent to that address; sigrok still uses the old word Slave. 1

setup-tools.sh — the series' setup script, which installs OSS CAD Suite, `sigrok-cli` and `tshark` into `.tools/` without administrator rights. 6

Shall, should, may and can — the normative words of ISO, IEC and IEEE standards, in lowercase: shall is a requirement, should a recommendation, may a permission and can a statement of what is possible. 5

Shelf — a group of books in the library about one kind of system, such as Vehicle or Wireless; each shelf has its own colour in every figure. 3

SHOULD (RECOMMENDED) — a rule to follow unless there is a valid reason and its consequences are understood; written in capitals in RFCs and in lowercase in ISO, IEC and IEEE standards. 5

Signal and packet pipelines — the bench's two paths: for wires, Icarus writes a VCD, `vcd-for-sigrok` cleans it and `sigrok-cli` decodes it; for networks, a scapy script writes a pcap file and tshark dissects it. 6

sigrok — free software that runs protocol decoders, such as uart and modbus, on logic captures; `sigrok-cli` is its command-line form and PulseView its window. 1

sigrok session file (.sr) — the capture file of a real logic analyzer, which sigrok and PulseView open and decode with the same decoders as a VCD. 6

Single-ended — said of a signal measured against ground, such as a UART at logic level; ground noise and cable capacitance soon corrupt it on a long cable. 4

Sink current (IOL) — the current an output must be able to pull in while it holds a low level; 3 mA for a Fast-mode I^2C device. 5

SMBus and PMBus — buses close to I^2C over which a BMC reads temperature sensors and fan speeds (SMBus) and controls the power supplies (PMBus). 2

SoC (system on a chip) — one large chip that holds the processor cores and the graphics unit and, in many phones, the cellular modem and the navigation receiver; in a phone it is also called the application processor. 2

SOME/IP — service messages that car units exchange over automotive Ethernet. 2

SpaceWire — a link over which spacecraft instruments send data to the on-board computer, as on the James Webb Space Telescope. 2

Specification (standard) — the agreed rule book for a protocol or interface, written down by a standards body or an industry group so that products from different makers work together; one document can define several things at once. 1

SPI (Serial Peripheral Interface) — a fast bus on one board, with a clock from the controller, two data lines and a select line. 2

Start bit — the 0 that begins every UART frame; its falling edge from the idle high level tells the receiver that a character starts now. 4

State machine — a set of states and the rules for moving between them, such as the UART receiver's idle, start, data, parity and stop states. 4

Stop bit — the 1 (or two 1s) that ends every UART frame, returns the line to idle and gives the receiver a known value to check. 4

Study path — the recommended reading order of the books for one career; every path starts with Book 1. 3

T

TCP (Transmission Control Protocol) — the Internet's main transport protocol, specified today in RFC 9293. 1

Testbench — a Verilog file that drives a design during simulation and prints messages. 6

TIA (Telecommunications Industry Association) — the trade association that writes RS-232 (TIA-232-F), TIA-422 and TIA-485. 5

Tier A — the tier of core protocols used daily, such as I^2C, CAN, PCIe and TCP: several chapters of 40–100 pages with all 22 dossier parts and 2–4 labs, after which you can design with the protocol, write its driver or RTL, capture it and debug it. 3

Tier B — the tier of protocols important in one field: one chapter of 15–35 pages with all 22 dossier parts in shorter form and one lab. 3

Tier C — the tier of specialist or legacy protocols: a section of 5–12 pages with only the core dossier parts, after which you can explain, recognise and decode the protocol. 3

Tier D — the tier of items that are not protocols, such as CSMA/CD or RINEX: a note of 1–2 pages that says what the item really is and where it is used. 3

Time-out — the longest time a device waits for an answer; when it expires, the client treats the silence as a failure and usually sends the request again. 1

Timescale — the line in a VCD header that gives its time unit, such as 1 ps, for the time stamps (`#t`) that follow. 6

Timing — the kind of protocol rule that says how fast bits are sent, how long to wait for an answer and how long a pause ends a message. 1

Timing diagram — a drawing of signals against time, in which each timing parameter is the time between two edges, measured at stated voltage levels. 5

TLS (Transport Layer Security) — the security protocol of web and other Internet traffic, which decides when to use a security algorithm such as AES, with which key, and what to do with the result. 1

tools-env.sh — the script you read into every new terminal with a dot (`. scripts/tools-env.sh`); it puts the bench's tools first on `PATH` and sets `EP_PYTHON`. 6

Top-of-rack and spine switches — the two levels of switches in a data-centre hall: each server connects to the switch at the top of its rack, and each of those connects to every spine switch. 2

TSN (time-sensitive networking) — Ethernet extensions that let urgent control traffic and ordinary traffic share one network with guaranteed timing. 2

TTL-level inputs — inputs with the thresholds of the old TTL logic family, VIH about 2.0 V, so a 3.3 V high reads as a 1 even on a 5 V chip; a 74HCT buffer powered from 5 V has such inputs. 4

Two's complement — the usual way of storing signed numbers, in which a 16-bit value from 0x8000 to 0xFFFF is negative, so 0xFFF0 is −16. 5

TX and RX (transmit and receive) — a UART's two signal pins; the TX of one device goes to the RX of the other. 4

U

UART (Universal Asynchronous Receiver-Transmitter) — a hardware block that turns bytes into a timed stream of bits on one wire and back into bytes; people also say UART for the bit format it uses. 4

UART character — one byte sent with its start bit, its data bits lowest bit first, an optional parity bit and its stop bit; 11 bits in the 8E1 format. 1

UART crystal — a crystal, such as 1.8432 MHz or 18.432 MHz, whose frequency divides exactly to the common baud rates. 4

UBRR (USART baud rate register) — the AVR's baud-rate register, which holds the divisor minus one. 5

UDP (User Datagram Protocol) — a simple transport protocol with an 8-byte header, whose Length field counts that header plus the payload. 6

UDS and DoIP (Unified Diagnostic Services, diagnostics over IP) — the diagnostic services with which a scan tool talks to a car's ECUs, today over CAN, and in newer cars also over Ethernet (DoIP). 2

UFS and eMMC (Universal Flash Storage, embedded MultiMediaCard) — the interfaces over which a phone reaches its flash memory; many lower-cost phones use the older eMMC. 2

UM10204 — NXP's user manual that is the I²C specification, written by a single company because Philips, now NXP, created I²C. 5

UPI and Infinity Fabric — processor links, Intel's and AMD's, over which the CPU sockets of a server share work and memory. 2

USB-C — a connector and cable, so an interface, through which several protocols travel: USB data, USB4, DisplayPort video in some ports and USB Power Delivery. 1

USB-IF (USB Implementers Forum) — the group that writes the USB specifications, which anyone can download free. 5

USB Power Delivery (USB PD) — the protocol with which a device and a charger agree, over USB-C, on a charging voltage higher than 5 V. 1

USB–UART bridge — a chip, such as the FT232R, CP2102 or CH340, that a PC sees as a serial port (`/dev/ttyUSB0` on Linux, a `COM` port on Windows) and that drives a UART link. 4

V

VCD (value change dump) — a text file that holds the time of every change of every signal, written by a simulator and read by sigrok like a logic analyzer's capture. 4

vcd-for-sigrok — a small helper that removes from an Icarus VCD the `$comment` and `$dumpall` sections that sigrok 0.5.2 cannot read. 6

Verilog — a hardware description language, in which the labs of this library write their transmitters and testbenches. 1

VIH (input-high voltage) — the lowest voltage that an input is guaranteed to read as a 1. 4

Virtual environment (.venv) — a private Python with its own packages, made once with `python3 -m venv .venv`, so the labs never touch the system's Python. 6

W

Wave — one of the three groups in which the books appear, essentials first. 3

Wi-Fi — the name that the Wi-Fi Alliance, which certifies products, uses for products that follow IEEE 802.11; its security protocols are WPA2 and WPA3. 1

Wireshark — free software that runs a dissector for each protocol layer of a packet; `tshark` is its command-line form. 6

WSL (Windows Subsystem for Linux) — a real Linux system inside Windows; WSL 2 runs it in a light virtual machine with a Linux kernel, and usbipd-win can hand it a USB device such as a logic analyzer. 6

X

XON/XOFF — software flow control: the receiver sends XOFF (0x13) to pause the sender and XON (0x11) to resume; no extra wires are needed, but the data must never contain those bytes. 4

The Protocol Library

A free Start Here guide and forty books that teach every electronics and communication protocol an engineer meets, from UART to 5G, deeply enough to answer any question about it. All books share one format.

Free **The Protocol Map** ← this book
Free guide: what protocols are, the map of all 40 books, the Protocol Passport method and your bench.

1 **Digital Communication Foundations**
Bits, clocks, line codes, signal integrity, CRC and error control, framing, access and tools.

2 **UART, RS-232, RS-422 and RS-485**
Asynchronous serial from the bit to multidrop RS-485 networks, AT commands and DMX512.

3 **SPI and Its Family**
SPI modes, QSPI, OSPI, xSPI, HyperBus, SPI flash command sets and eSPI.

4 **I²C, SMBus, PMBus and I3C**
The two-wire bus family: I²C, SMBus, PMBus, AVSBus, I3C, SPMI and RFFE.

5 **Embedded I/O, Sensor and Single-Wire Protocols**
GPIO, PWM, ADC/DAC interfaces, 1-Wire, IR remotes, LED and encoder protocols.

6 **Debug, Programming, Test Access and Instrument Control**
JTAG, boundary scan, SWD, CoreSight, RISC-V debug, bootloaders and SCPI/GPIB.

7 **CAN, CAN FD, CAN XL and the CAN Protocol Family**
Classical CAN to CAN XL, ISO-TP, J1939, CANopen, NMEA 2000 and ISOBUS.

8 **LIN, FlexRay, SENT, PSI5 and Classic Vehicle Buses**
LIN, CXPI, FlexRay, SENT, PSI5, K-Line, MOST and J1850.

9 **Automotive Ethernet, SOME/IP, V2X and Vehicle Diagnostics**
T1 Ethernet, SOME/IP, V2X, DoIP, UDS, OBD-II, XCP and SecOC.

10 **Industrial Fieldbuses**
4–20 mA, HART, Modbus, PROFIBUS, FOUNDATION Fieldbus, DeviceNet and IO-Link.

11 **Industrial Ethernet, TSN and OPC UA**
EtherNet/IP, PROFINET, EtherCAT, TSN, redundancy, safety protocols and OPC UA.

12 **Building Automation, Smart Grid and Power-Line Communication**
BACnet, KNX, DALI, DNP3, IEC 61850, DLMS/COSEM and power-line links.

13 **Ethernet and LAN Switching**
802.3 PHYs, MII family, PoE, switching, VLANs, spanning tree and LACP.

14 **IPv4, IPv6 and Routing Protocols**
IP, ICMP, NAT, IPv6, MPLS, OSPF, IS-IS, BGP, VXLAN and Segment Routing.

15 **TCP, UDP, QUIC and Transport Protocols**
TCP in depth, congestion control, UDP, SCTP, QUIC and MPTCP.

16 **Internet Application Protocols**
HTTP/1.1 to HTTP/3, DNS, DHCP, mail, SNMP, SIP, RTP and network automation.

17 **IoT Messaging: MQTT, CoAP, LwM2M, AMQP and DDS**
Publish/subscribe and device-management protocols for IoT and robotics.

18 **Radio Foundations for Wireless Protocols**
Modulation, OFDM, spread spectrum, link budgets, MIMO and spectrum rules.

19 **Wi-Fi: IEEE 802.11 from Wi-Fi 4 to Wi-Fi 7**
The 802.11 PHY and MAC, management, roaming, QoS and Wi-Fi 7.

20 **Bluetooth Classic and Bluetooth Low Energy**
BR/EDR, BLE link layer, GATT, pairing, LE Audio and Bluetooth Mesh.

21 **Zigbee, Thread, Matter and Low-Power Wireless**
802.15.4, 6LoWPAN, Zigbee, Thread, Matter, Z-Wave, UWB and more.

22 **LPWAN: LoRaWAN, Sigfox, NB-IoT and LTE-M**
Long-range, low-power networks and their battery maths.

23 **Cellular I: From 1G to 4G LTE**
Telephone basics, GSM, UMTS, LTE, the EPC and VoLTE.

24 **Cellular II: 5G NR, 5G Core, O-RAN and 6G**
NR PHY and stack, the 5G Core, O-RAN, RedCap, NTN and 6G.

25 **Optical and Transport Networks**
PDH, SONET/SDH, OTN, WDM, PON, Ethernet optics, CPRI and eCPRI.

26 **DRAM Interfaces: DDR, LPDDR, GDDR and HBM**
DRAM commands and timing, DDR5, LPDDR, GDDR, HBM and controllers.

27 **Flash and Storage Interfaces**
NAND, SD, eMMC, UFS, SATA, SAS, NVMe and storage networks.

28 **On-Chip Buses: AMBA and Open Buses**
APB, AHB, AXI, ACE, CHI, Wishbone, Avalon, TileLink and OBI.

29 **PCI Express, CXL and Data-Centre Interconnects**
PCIe in depth, CXL, NVLink, UALink, InfiniBand, RoCE and platform management.

30 **USB, USB-C, USB Power Delivery and Thunderbolt**
USB 2.0 to USB4, Type-C, Power Delivery, BC1.2 and Thunderbolt.

31 **SerDes, JESD204 and Chiplet Interconnects**
SerDes equalization, JESD204, Aurora, Interlaken, UCIe, BoW and AIB.

32 **Display Interfaces: HDMI, DisplayPort, MIPI DSI and LVDS**
Video timing, HDMI, DisplayPort, eDP, MIPI DSI, D-PHY/C-PHY and HDCP.

33 **Camera, Broadcast and Professional Video**
MIPI CSI-2, UVC, GigE Vision, SDI, ST 2110, DVB and GMSL.

34 **Digital Audio Interfaces**
I²S, TDM, PDM, S/PDIF, AES3, MIDI, A2B, AVB and AES67.

35 **Communication Security Protocols**
Crypto building blocks, TLS, SSH, IPsec, WireGuard, WPA3, MACsec and FIDO2.

36 **Power, Battery and Charging Communication**
Smart batteries, BMS links, Qi, EV charging, ISO 15118 and OCPP.

37 **Time, GNSS and Navigation Protocols**
NTP, PTP, SyncE, GNSS signals, NavIC, NMEA, RTCM and NTRIP.

38 **RFID, NFC, Smart-Card and SIM Protocols**
ISO 14443, 15693, EPC Gen2, NFC, ISO 7816, SIM and eSIM.

39 **Aerospace, Drones, Rail and Space Data Links**
MIL-STD-1553, ARINC, ADS-B, MAVLink, rail TCN, SpaceWire and CCSDS.

40 **Medical Device Communication**
IEEE 11073, health profiles, body-area networks, HL7, FHIR and DICOM.

Thank you for reading.