FIRST EDITION · 2026

CROSS-COMPILER CONSTRUCTION *for* EMBEDDED SYSTEMS

A Hands-On Guide with GCC, Clang, and Docker

WRITTEN BY

Abdollah Ebadi

Cross-Compiler Construction for Embedded Systems

A Hands-On Guide with GCC, Clang, and Docker

Abdollah Ebadi

This book is available at https://leanpub.com/cross-compilerconstructionforembeddedsystemsahands-onguidewithgccclanganddocker

This version was published on 2026-05-21

In the name of He who taught humanity what they did not know through writing. To the love of my life, Narges, my compass in every storm and my calm in every harbour.

Contents

Copyright

Cover design: Abdollah Ebadi **Cover background:** "Technology background in abstract style" designed by Freepik – Magnific.com

Published on Leanpub: https://leanpub.com/cross-compilerconstructionforembeddedsonguidewithgccclanganddocker

Preface

Why This Book Exists

Every embedded system developer has encountered a cross-compiler. Most have used one without thinking too deeply about it — downloaded a pre-built tool-chain, set a handful of environment variables, and moved on. That works, until it doesn't. A glibc version mismatch produces a cryptic linker error on the target. A pre-built Clang tool-chain silently uses the wrong sysroot. A Docker image ships a compiler that nobody on the team can reproduce from source. At that point, not understanding what is inside the tool-chain stops being acceptable.

This book is to help the day to day developer get into the understanding of how cross-compilers are put together and thus we are going to treat them as a first-class topic. A lot of the books on embedded systems briefly touch this topic by introducing automation tools like Crosstool-NG and stop there. None cover Clang cross-compilation in depth. None explain glibc version matching as a deliberate production practice. None walk you through building a tool-chain by hand and then containerising it for distribution. That is the gap this book fills. And the reason it does is that one needs to have a good understanding of what is going under the hood of tools like Crosstool-NG to be able to debug issues when they occur.

Who This Book Is For

This book is written for developers who are comfortable at the Linux command line and have some prior exposure to embedded development or cross-compilation — even if only as a user of pre-built tool-chains. You do not need to have built a compiler before. You do not need a deep background in compiler theory. Having exposure to basic computer related topics such as the general compilation steps that a source code goes through is a must. In addition, what you need is the willingness to follow a build process step by step, read terminal output carefully, and understand why each step exists

rather than just executing it. The understanding is a must – do not just blindly copy everything and paste it in your system. Have the due diligence to read, question, experiment and understand.

If you have ever wondered what `arm-linux-gnueabihf-gcc` actually is, where it comes from, and why the name looks the way it does – this book answers that question in full.

What This Book Is Not

This book does not teach embedded Linux application development, kernel configuration, or device driver writing. This book is specifically about the toolchain – the infrastructure you need before you can compile anything for a target board.

A Note on the Hardware

All demonstrations in this book use two ARM boards: a TinkerBoard S R2.0 (ARMv7-A, 32-bit) and a Raspberry Pi 4 (ARMv8-A, 64-bit). The BeagleBone Black (ARMv7-A, 32-bit, 512MB RAM, 4GB eMMC) is introduced in Part 0 and serves as an additional ARMv7-A target – its resource constraints make it a realistic representation of production embedded hardware. It becomes the primary board in Book 2. ARM is the dominant architecture in embedded systems development and makes for concrete, realistic demonstrations. However, the concepts in this book are not ARM-specific. Everything you learn here applies equally to other modern processors such as RISC-V, MIPS, and PowerPC, and the reader is encouraged to take the knowledge established through the pages of this book and apply it to other architectures. Remember – as an engineer, experimentation is your best friend.

If you do not have access to physical hardware, Chapter 2 of Part 0 establishes a Docker-based ARM testing environment that lets you run cross-compiled binaries without a physical board. See also Appendix A for a quick reference.

A Note on the Series

This book is a self-contained technical book on cross-compiler construction. It covers the tool-chain from foundations to containerised distribution. But it can also be considered as part of a series of books – a series that delves into the foundations of hardware-level computer software and tools. This book is the foundation for that series, and when fully grasped, the interested reader can follow along with the series on the author's Leanpub page.

How to Use This Book

Read Part 0 even if you are experienced. It is short – just two chapters – and it establishes the vocabulary, mental models, and environment that the rest of the book assumes. Experienced readers will move through it quickly and it will help them review topics they are already familiar with; newer readers will find it essential.

Follow the builds in sequence. Parts 1 and 2 build on each other deliberately. Part 1 uses Crosstool-NG to build a tool-chain automatically. Part 2 builds the same tool-chain by hand. Doing them in order means that when you build manually in Part 2, you already know what the finished product should look like. From my personal experience this type of flow creates the "Ah, I get it now...." moment and then that knowledge sticks with you for a longer time.

Use the appendices as a reference. The appendices in this book are for later reference – the troubleshooting appendix (Appendix B) collects every real error encountered during the builds in Parts 1, 2, and 3, with causes and fixes. It is not meant to be read sequentially – it is a lookup resource for when something goes wrong.

The code repository accompanying this book mirrors the chapter structure exactly. Scripts are numbered in execution order within each chapter folder so the sequence is self-evident. All code is available at:

https://github.com/abdollahebadi/Cross-Compiler-Construction-for-Embedded-Systems

What You Will Have Built by the End

By the time you finish this book you will have:

- A generic ARMv7-A GCC cross-compiler built manually from source, matched to glibc 2.28 on the target OS, capable of producing binaries for both the TinkerBoard and the BeagleBone Black
- A CPU-specific ARMv8-A GCC cross-compiler built manually from source using musl as the C library, targeted to the Raspberry Pi 4
- The same ARMv8-A musl tool-chain built with Crosstool-NG, compared against the manual build
- A Clang/LLVM cross-compiler reusing the GCC sysroot, configured for both ARM targets
- Containerised versions of all three tool-chains, versioned by digest, distributable with a single `docker pull`

These are not toy tool-chains. They are production-matched, reproducible, and ready to use as the foundation for everything that comes next.

A Guide to the Contents

Part 0 — Foundations

Part 0 covers the foundational ground and knowledge for the rest of the book. It provides the conceptual mental model and the environment that everything else depends on.

Chapter 1 – Foundations A single chapter covering the essential mental models: what a cross-compiler is and why embedded development requires one; tool-chain anatomy; CPU architecture and what it mandates for the tool-chain; target triples decoded field by field; a comparison of the five main C libraries (glibc, musl, uClibc-ng, Newlib, and Picolibc) with a decision framework; the bootstrap problem and why it is the foundational challenge of compiler construction; and an introduction to the three target boards used throughout the book.

Chapter 2 – Setting Up the Development Environment The practical setup chapter. Covers the rationale for using a virtual machine, platform options (Linux, Windows with WSL2, macOS with Intel and Apple Silicon considerations), the reference environment used for all demonstrations, prerequisite installation, the `~/cross/` directory structure used throughout the book, Docker Engine setup for both binary testing and tool-chain containerisation, and a baseline VM snapshot before the build chapters begin.

Part 1 — Crosstool-NG

Part 1 is the guided tour. Crosstool-NG automates the tool-chain build – the goal here is not just to produce a working compiler but to understand everything the tool does on your behalf.

This part covers what Crosstool-NG is and where it sits in the tool-chain landscape; how to install and configure it; how to determine the correct target triple from hardware; a detailed walkthrough of the configuration menu; building a cross-compiler for the TinkerBoard; reading and understanding the terminal output at each build phase; and inspecting the finished sysroot. The part closes by comparing the Crosstool-NG output to what Part 2 will build manually – establishing what the tool automated and what it deliberately hid.

Part 2 — Manual GCC Cross-Compiler

Part 2 is the deep technical core of the book. Building the tool-chain by hand – step by step, stage by stage – is what transforms a tool-chain user into someone who genuinely understands the system.

Chapter 1 – Understanding the Sysroot Before the manual build begins, this chapter explains what the sysroot actually is. By this point the reader has seen Crosstool-NG produce one automatically; now it gets a full treatment. The chapter covers how the compiler finds headers and libraries on a native system, why that same mechanism cannot work when host and target are different machines, what the sysroot contains and deliberately does not contain, how the compiler is told where it is, and the common mistakes that produce confusing build errors.

Chapter 2 – Matching the Tool-chain to the Target OS A production practice made explicit before the build begins. Covers why glibc version matters through GLIBC symbol versioning; what happens when the tool-chain glibc is newer than the target OS glibc; how to inspect a target OS to determine its glibc version; the decision to build against glibc 2.28 to match TinkerOS; and how to verify the resulting binary carries no GLIBC requirements above the target version. Includes a reusable inspection workflow applicable to any target OS.

Chapter 3 – Building the ARMv7-A Generic Tool-chain (TinkerBoard S R2.0) The five-stage manual build: binutils, GCC stage 1, glibc 2.28 headers and startup files, GCC stage 2, and the full glibc 2.28. Covers the design decision

to build a generic tool-chain not locked to a specific CPU, what that means in practice, and how to target specific boards at compile time. The chapter closes by verifying the tool-chain produces binaries that run correctly on the TinkerBoard.

Chapter 4 — Building the ARMv8-A CPU-Specific Tool-chain with musl (Raspberry Pi 4) The same five-stage build for a 64-bit target using musl instead of glibc. Covers the decision to lock the tool-chain to `cortex-a72`, what musl changes in the build process, musl's stricter POSIX compliance and its practical implications, and a side-by-side comparison of the glibc and musl sysroots by size and contents.

Chapter 5 — Revisiting Crosstool-NG: Building the musl ARMv8-A Tool-chain With the manual build complete, the reader returns to Crosstool-NG with fully informed eyes. Covers the configuration changes needed to switch from the ARMv7-A glibc build to the ARMv8-A musl build, comparing the Crosstool-NG output to the manually built tool-chain, and when to reach for Crosstool-NG versus building manually in a real project.

Chapter 6 — Inspecting and Understanding the Output A systematic examination of what was built. Covers the anatomy of the finished sysroot, what each library is and why it is there, and using `file`, `objdump`, and `readelf` to verify cross-compiled binaries and read GLIBC version tags.

Chapter 7 — Troubleshooting Every real error encountered during the builds in this part, with causes and fixes. Covers the `stdio.h` libgcc bootstrap failure, glibc version mismatches, missing headers, misconfigured sysroot paths, and compiler version conflicts.

Part 3 — Clang/LLVM Cross-Compiler

Part 3 is all about the newer kid around the block. Clang cross-compilation for embedded Linux receives almost no coverage in existing resources – this part fills that gap.

This part covers why Clang cannot bootstrap itself and why GCC is always the foundational "chicken"; Clang's natively multi-target architecture; reusing the GCC sysroot built in Part 2; building LLVM and Clang from source; the full LLVM runtime stack (compiler-rt, LLD, libc++, libunwind); configuring Clang for ARM cross-compilation; the difference between compiler-rt and libgcc; C++ runtime choices between libstdc++ and libc++; kernel cross-compilation with

Clang and the `LLVM=1` flag; and an honest comparison of when to use GCC versus Clang in production.

Part 4 — Docker and Tool-chain Containerisation

Part 4 takes the tool-chains built in Parts 2 and 3 and packages them for distribution, reproducibility, and CI/CD integration.

A Docker introduction covers core concepts, installation, key commands, and Dockerfile syntax. The tool-chain containerisation chapters cover designing and building containers for the GCC ARMv7-A, GCC ARMv8-A musl, and Clang tool-chains; a versioning and tagging strategy; and comparing container sizes. The distribution chapter covers running a local registry, pushing to Docker Hub, using image digests as a provenance record, onboarding a new developer with a single `docker pull`, and a minimal CI/CD integration example using GitHub Actions.

Part 5 — Where to Go Next

Part 5 closes the book by pointing outward. The tool-chain is the foundation – this part maps where it leads.

The skills built across this book do not exist in isolation. They are the prerequisite for a much broader journey into embedded systems engineering. This part explores how cross-compiler knowledge becomes the foundation for operating system development, hypervisor construction, and mixed-criticality systems design – areas where the tool-chain decisions you make have direct consequences on system safety, performance, and architecture. It closes with an explicit look at where this series goes next and the titles that continue from where this book leaves off.

Appendices

The appendices are reference material, not sequential reading – reach for them when you need them, not before.

Appendix A – Testing Without Hardware is a quick reference for readers who do not have physical boards available. It covers the one-line commands for running ARMv7-A and AArch64 binaries in a Docker container, choosing the

right base image for glibc version matching, and mounting a local directory into the container for binary transfer. Useful from Part 1 onwards.

Appendix B – Troubleshooting Reference collects every real error encountered during the builds in Parts 1, 2, and 3 – with the cause and fix for each. If something breaks during a build, check here before spending an hour on Stack Overflow. It is not meant to be read sequentially – it is a lookup resource for when something goes wrong.

Part 0: Foundations

The objective of *Part 0* is to establish the conceptual and practical foundation that everything else in this book depends on. Whether you are new to cross-compilation or an experienced embedded developer looking to fill in the gaps, this part gives you the mental models and the working environment you need before a single line of build output appears on your screen.

This part comprises the following chapters:

- *Chapter 1*, *Foundations*
- *Chapter 2*, *Setting Up the Development Environment*

Chapter 1 — Foundations

What is a Cross-Compiler and Why Do We Need One?

For the reader that is not familiar with the term "Compiler" – sometimes referred to as a tool-chain – it's a complex set of software tools in which it compiles source code into an executable for a target platform. This leads naturally to a distinction between the two fundamental modes of compilation:

Cross-compilation is the mode this book is concerned with. The toolchain runs on a host machine that differs from the target, allowing development to happen on a fast, well-resourced workstation while the resulting binary runs on a constrained embedded system. This is the standard approach in professional development; for example, an Android application is built on a powerful PC using tools provided by Google, but the resulting app runs on a mobile phone or tablet.

Native compilation is where the toolchain runs on the same type of system – sometimes the very same machine – as the programs it produces. This is how software is typically built for desktop and server platforms. For high-end embedded systems without significant resource constraints, native compilation on the device itself can be a viable option, and it carries the advantage of simplicity: there is no ABI mismatch to worry about, no sysroot to maintain, and the compiler naturally sees the same system libraries as the running programs. However, for the vast majority of embedded targets this book addresses, native compilation is not a practical option.

In either case, the word **toolchain** is worth emphasising. It is not merely a synonym for "compiler." This term reflects the fact that turning source code into an executable is a multi-stage pipeline rather than a single operation. It is a series of steps where each tool's output acts as the input for the next – from the preprocessor to the compiler, assembler, and finally the linker. One can think of the cross-compiler as the **orchestrator** of this entire show, or refer to the whole machinery as the "cross-compiler" itself. Regardless of the viewpoint, understanding the individual parts of this machine is vital for both constructing a toolchain and using one effectively.

The Purpose: Separating the Build Environment from the Target

One can argue that one of the fundamental purposes of a cross-compiler is to formally separate the *build environment* from the *target environment*. This separation is not just about convenience – in most cases, it is a hard requirement enforced by the system constraints.

In all cases, logically speaking, one of the factors below dictates to us the need for separating the build from the target environment.

The target often cannot build itself. Entry-level embedded systems, such as the popular Arduino boards, may lack the resource capacity to run a compiler at all. A microcontroller with 256KB of flash and 64KB of RAM has no realistic path to run the compilation process natively. The separation is not a design choice in these cases – it is a constraint imposed by the hardware itself.

The build environment can be controlled independently. Realistically speaking, in many projects – such as in **aerospace or automotive engineering** – the application-level developers or even the system-level developers might not have physical access to the target hardware, or the hardware may be a **fixed, certified entity** that cannot be altered. In such cases, it is necessary that the development environment be separated from the target to give control to the developer to configure the setup based on the needs of the application. One might say that in such circumstances, the target still limits what the developer can do, and that is, of course, true; however, even within those boundaries, the developer retains the freedom to experiment with configurations. For instance, they can toggle between **optimisation levels** (prioritising execution speed versus code size for limited flash memory) or modify **sysroot contents** to test different library versions without needing to "install" anything on the actual device. The target only ever receives the output – a binary compiled to its exact instruction set and ABI. This independence is what makes **reproducible, auditable builds** possible, ensuring that a firmware image can be perfectly recreated years later in a virtualised build environment, even if the original prototype hardware is long gone.

Maintaining Target Integrity

In critical embedded systems such as automotive ECUs, medical devices, and industrial controllers, the more the target environment is leaner, the better.

This reduces the possibility of bugs and failures and thus increases system robustness and health. In order to achieve this, cleaning the target from clutter, such as unused and unwanted libraries, is key, and one of the ways to achieve this is to avoid developing natively. Development tools, headers, and build infrastructure can all increase the footprint of the runtime environment and increase the possibility of failure. The cross-compiler enforces this discipline structurally: the toolchain lives entirely on the host, and only the final artifact crosses the boundary.

Toolchain Anatomy — Compiler, Binutils, Sysroot, and Runtime Libraries

A tool-chain is not a monolithic application. It is a collection of cooperating tools, each responsible for a specific stage of the build process. Understanding what each component does and what role it plays is essential before attempting to build one from scratch.

The Compiler

The compiler is the most visible part of the tool-chain, which is why the entire tool-chain is loosely referred to as the "Compiler". Its job is to translate higher-level source code, such as C, C++, or other supported languages, into assembly language for the target architecture. When we refer to *the compiler* as a single entity, we are really referring to the entire front-to-back pipeline that accepts source code as input and produces assembly as output. Internally, this pipeline consists of various sections, including a pre-processor, a parser, a semantic analyser, an intermediate representation layer, an optimiser, and a code generator. The internals of these steps are outside the scope of this book, and the curious reader is directed to the further reading section for references on the internals of compilers.

What matters here is the *boundary* the compiler operates at: it knows the target architecture, the calling conventions, the register set of the target processor, the instruction set, and the ABI. Every decision it makes about how to lay out data, pass arguments, and generate instructions is shaped by those target-specific facts.

The C Library

The C library – commonly abbreviated as *libc* – is the bridge between application code and the underlying system, whether that system be a Linux kernel or bare-metal hardware. It is the component most developers interact with constantly without thinking about it: every call to the famous `printf`, `malloc`, `memcpy`, or `strlen` functions and alike goes through the C library.

The cross-compiler must know about the C library at *build time* because the C library defines the ABI, which we will get to in a minute, but roughly speaking, it determines calling conventions, the layout of standard types, the interface to the operating system, and the startup sequence that runs before `main()` is called. Get the C library wrong – or mix incompatible versions – and the resulting binary will fail to link, fail to load, or fail at runtime in ways that are difficult to diagnose.

The role of the C library differs depending on the type of the target environment:

On a **Linux-based target**, the C library (typically glibc, musl, or uClibc-ng) serves as the interface to the kernel. Application code calls standard functions; the C library translates those calls into the correct system calls for the target architecture.

On a **bare-metal target**, there is no operating system and therefore no kernel to call into. The C library's role shifts: it still provides the standard library functions your code depends on, but the OS-facing layer is either stubbed out or replaced by hardware-specific implementations you provide yourself. The standard choice for bare-metal work is newlib or its size-optimised variant newlib-nano, both designed specifically for this environment. In the most minimal case – boot-loaders, OS kernels – you can instruct GCC to operate in fully freestanding mode (`-ffreestanding`), making no assumptions about any C library at all.

Binary Utilities (Binutils)

The binary utilities – universally known as *binutils* – are the set of tools that operate on assembly and machine-level code. The two most important are the assembler and the linker.

The **assembler** (`as`) takes the assembly output produced by the compiler and converts it into object files containing machine code. Each translation unit

produces one object file; those files contain code and data, but are not yet a complete executable.

The **linker** (`ld`) takes all the object files produced by the assembler, along with any required static libraries, and combines them into a single executable or shared library. For embedded targets, the linker script is particularly important: it tells the linker where in memory to place code, initialised data, the stack, and other sections, reflecting the actual memory map of the target hardware.

Beyond the assembler and linker, binutils also includes tools for inspecting and manipulating binary artifacts: `objdump` for disassembly and section inspection, `readelf` for examining ELF file headers and symbol tables, `nm` for listing symbols, `strip` for removing debug information, and `objcopy` for converting between binary formats – for example, converting an ELF executable into a raw binary image for flashing to a microcontroller.

The Sysroot

The sysroot is the physical manifestation of the separation between build and target environments, which was mentioned earlier. It is a directory on the host machine that mirrors the root file-system of the target – containing the target's C library headers, system headers, and pre-built libraries – organised exactly as they would appear on a running target system.

To understand what a sysroot is and why it exists, it helps to first consider what happens during native compilation – the familiar case where the compiler and the target are the same machine.

When you compile a program natively on a Linux system, the compiler needs to find two categories of things: **headers** and **libraries**. Headers are consulted at compile time – they tell the compiler what functions, types, and constants exist and what their signatures look like. Libraries are consulted at link time – they provide the actual compiled implementations that the linker will bind your code against. At run-time, shared libraries are found again by the dynamic linker, but that is a separate concern from the build process itself.

On a native Linux system, the compiler knows exactly where to find these things because the build machine *is* the target machine. Headers live in `/usr/include`. Libraries live in `/usr/lib` or `/lib`. The compiler has these paths baked in as defaults, and they are correct because the system the

compiler runs on and the system the binary will run on are one and the same. The root of the filesystem is simultaneously the root of the build environment and the root of the target environment. There is no distinction to make.

Cross-compilation breaks this assumption entirely. The host machine and the target machine are different systems, with different architectures, different C library versions, and different system library layouts. If the cross-compiler were allowed to look in the host's `/usr/include` and `/usr/lib` for headers and libraries, it would find the host's own files – compiled for the host architecture, reflecting the host's C library version – and silently use them to build code intended for an entirely different system. The resulting binary might link, but it would encode the wrong ABI, reference the wrong symbol versions, and very likely crash or misbehave on the target.

What is needed is a mirror: a directory on the host machine that contains the headers and libraries of the *target* system, laid out in the same structure as they would appear on a real target filesystem. This mirror is the **sysroot**.

Structurally, a sysroot looks like a minimal root filesystem for the target:

```
sysroot/
├── usr/
│   ├── include/       ← target system headers and C library headers
│   └── lib/           ← target libraries for linking
└── lib/               ← target C library and dynamic linker
```

The cross-compiler is configured to treat this directory as the root for all header and library searches. When it needs `<stdio.h>`, it looks in `sysroot/usr/include`. When the linker needs `libc.so`, it looks in `sysroot/usr/lib` or `sysroot/lib`. The host system's own headers and libraries are never consulted.

One point worth emphasising: the sysroot is a **compile-time and link-time** construct only. It exists entirely on the host machine and plays no role whatsoever at runtime on the target. The target does not need the sysroot – it has its own real filesystem with its own real libraries. The sysroot is simply the host's *model* of that target filesystem, present only so the compiler and linker can make correct decisions during the build. Once the binary is built and transferred to the target, the sysroot's job is done.

This distinction matters from a practical perspective. The sysroot does not need to be a complete replica of the target filesystem. It does not need

structures such as device nodes, configuration files, application binaries, or anything that is only relevant at runtime. It needs precisely and only what the compiler and linker require during a build: the C library headers, the system headers, and the compiled libraries to link against. A minimal, correct sysroot is sufficient – and in fact preferable, since a leaner sysroot is easier to version-control, reproduce, and reason about.

When the cross-compiler builds your code, it looks inside the sysroot rather than at the host system's own headers and libraries.

Debugging Utilities

Although not strictly required to produce a binary, debugging utilities are an essential part of any practical tool chain and deserve mention here. The nature of software development and, in particular, embedded systems is such that bugs are inevitable, and the ability to diagnose them efficiently is not optional.

The primary debugging tool in the GNU toolchain is `gdb`, or the GNU Debugger. Cross-debugging typically involves `gdb` running on the host, communicating with a `gdbserver` process or a hardware debug probe on the target. This allows full source-level debugging using breakpoints, watchpoints, register inspection, and memory examination across the host-target boundary.

These tools complete the picture of what a full tool-chain provides: not just the ability to build for a target, but the ability to build *and debug* for one, which in practice is the capability you actually need, as your eyes are inside the embedded system during development.

CPU Architecture and What It Mandates

When you build a cross-compiler, one of the first things you must decide is the exact CPU architecture you are targeting. This is not a vague, high-level choice; the compiler needs a precise answer to a set of questions about the target hardware before it can generate a single correct instruction for it. Those questions fall into the following categories:

- What is the target instruction set (ISA)?
- Which calling convention should be used?

- What rules should be considered concerning data layout?
- Do we have access to hardware-accelerating units like Floating Point Units (FPUs), or do we need to use software-based ones?
- What byte ordering does the target processor support?

Together, these form the foundation of the tool chain to generate its binaries. Get any of them wrong, and the result ranges from a binary that runs slightly inefficiently to one that crashes immediately or silently produces wrong results.

The Instruction Set Architecture (ISA)

The instruction set architecture is the most fundamental description of the target processor; it defines the instructions the processor can execute, the registers it exposes, and the memory model it operates under.

Common ISA families you will encounter in the wild include ARM, x86_64 (AMD64), MIPS, PowerPC, and RISC-V, to name a few. Each of these ISAs comes with distinct instruction encodings, register conventions, and memory models. RISC-V deserves particular mention as it represents a new generation of open, royalty-free ISA design that has gained significant traction in both academic and commercial embedded systems – from micro-controllers to application processors – and is increasingly appearing as a first-class target in production tool-chains.

Within any given ISA family, there is often significant variation between generations. ARM is a particularly clear example. ARMv7-A is the 32-bit architecture found in processors like the Cortex-A8, the processor found in the famous SBC BeagleBone Black, or Cortex-A17, an upgraded variant of the same processor, and the core inside the TinkerBoard SBC used throughout this book.

ARMv8-A introduces a completely new 64-bit execution state alongside a 32-bit compatibility mode, and the Cortex-A72 inside the Raspberry Pi 4 is an ARMv8-A implementation. These two generations are not the same architecture in any practical sense; their instruction encodings differ, their register files differ, and as a result, the binaries created for them differ in structure from one another as well. That being said, because of the backward compatibility that ARMv8 processors have, you can still run an ARMv7-based

binary on such processors, but care needs to be given to the target system for it to be able to run successfully.

MIPS has a similar generational split between MIPS32 and MIPS64, and between Release 1 and Release 6, which introduced incompatible instruction encoding changes. PowerPC spans decades of evolution from the original 32-bit Book E embedded cores used in networking equipment and automotive systems through to the 64-bit POWER cores used in servers. RISC-V follows a modular design philosophy where the base integer instruction set is minimal and well-defined, and additional capabilities — multiplication, atomic operations, floating-point, vector extensions — are added as optional, independently versioned modules. This modularity means that two RISC-V processors can share the same base ISA but differ significantly in what extensions they support, and the compiler must know the exact extension profile of the target to generate correct and efficient code. In every case, the compiler must know not just the broad ISA family but the specific generation and what we call "**micro-architecture**" it is targeting, so it can take advantage of what is actually available and avoid emitting instructions the CPU cannot execute.

The Application Binary Interface (ABI)

If the ISA describes what instructions a CPU can execute, the ABI describes the *rules* that compiled code must follow so that separately compiled pieces can work together correctly. The ABI is a contract between your code and the C library, between your application and the operating system, between one shared library and another. These rules cover several areas of concern:

Calling conventions define which registers are used to pass function arguments, which register holds the return value, and which registers a called function must preserve across a call versus which it is free to use freely. If the caller and the function to be called follow different calling conventions, arguments arrive in the wrong registers, and return values are read from the wrong place, leading to silent memory corruption and unpredictable execution. And given that the ABI is out of the direct control of the normal developer, identifying such runtime bugs and tracking down their root cause is incredibly difficult.

Data type sizes and alignment define how wide, for example, an integer is going to be or how a `long long` data type is aligned in memory, and how structures are padded. If two compilation units compiled under different ABI

assumptions disagree on the size of these types of decisions, one will read fields at the wrong offsets, which again will result in unwanted and unknown results, which in turn is again hard to resolve.

Stack layout and frame conventions define how the stack grows, where the frame pointer lives, and how the return address is stored and recovered.

Name mangling is particularly relevant for C++, where the ABI defines how function names are encoded to include type and namespace information, ensuring that a function compiled in one translation unit can be correctly referenced from another.

The practical consequence is that the ABI is not negotiable once chosen – and once set in stone, every component that will interact at runtime, such as your application, the C library, third-party libraries, the operating system, and so forth, must comply with the same set of rules and thus must have been compiled against the same ABI. The cross-compiler enforces this by encoding the ABI into its configuration at build time, and it will refuse to link object files that violate the contract.

As an example of the ABI and in relation to the processor architecture we will target in this book, we will have a closer look at some of the available ARM ABIs.

The ARM Embedded ABI (EABI)

The ARM architecture has a specific ABI story worth addressing directly, because it is what you will be working with throughout this book on both the TinkerBoard and the BeagleBone Black.

Early ARM Linux systems used what is now called the **Old ABI (OABI)**. It was functional but had several shortcomings – inefficient system call conventions, ambiguity around struct layout, and inconsistent floating-point handling across implementations. As ARM moved into more serious embedded and mobile deployments, these inconsistencies became a practical problem.

The **Embedded ABI (EABI)** was introduced to address this. It is a comprehensive, precisely specified ABI designed specifically for ARM processors in the context of embedded systems, and it became the standard for all modern ARM Linux distributions. Among other things, EABI defines:

- A clean, efficient system call convention that passes arguments in registers rather than on the stack.

- Precise struct packing and alignment rules, eliminating ambiguity across compilers and compiler versions.
- A standardised set of helper routines for operations the hardware may not support directly, such as integer division and certain floating-point operations, collected in a library called **libgcc** or its ARM-specific counterpart **libgcc_s**.
- Clear rules for how software and hardware floating-point interoperate, which directly underpins the soft, softfp, and hard-float distinctions discussed in the next section.

EABI is not optional for modern ARM Linux work; it is considered the baseline. Every component in the system, from the kernel to the C library to your application code, must be compiled to EABI conventions. When you build a cross-compiler targeting ARM Linux today, you are building an EABI toolchain, and this will be reflected explicitly in the toolchain configuration you work through in later chapters.

It is worth noting that EABI was defined specifically for 32-bit ARM. When ARM introduced the 64-bit ARMv8-A architecture, the ABI was redesigned from scratch and given a new name – the **AArch64 Procedure Call Standard (AAPCS64)**, commonly referred to simply as the ARM64 ABI. The Raspberry Pi 4, which runs a 64-bit ARMv8-A configuration, uses this new and extended ABI. The two are not compatible – they are distinct architectures with distinct ABIs – but the underlying principles of calling conventions, register usage, and data layout rules apply equally to both.

Floating-Point: Hard, Soft, and Soft-fp

Floating-point handling is one of the most practically consequential choices when configuring a cross-compiler for an embedded processor. Especially on ARM, it is a frequent source of ABI incompatibility encountered in practice. The central question is straightforward: does your target CPU have a hardware floating-point unit (FPU), and if so, how should the compiler pass those arguments during function calls?

- **Soft-float** means the compiler generates no hardware floating-point instructions at all; all arithmetic is emulated through software library calls. While this is the safest option because it runs on any processor regardless of whether an FPU is present, it carries a severe performance

penalty. For systems requiring real-time math execution, this overhead can be fatal to baseline performance.

- **Soft-fp** occupies a middle ground. The compiler uses hardware floating-point instructions for localised arithmetic, but it still passes floating-point arguments via standard integer registers at function boundaries. This preserves compatibility with soft-float code while recovering significant computational performance.
- **Hard-float** means the compiler leverages the FPU for both arithmetic and passing function arguments directly within FPU registers. This is the fastest option and the standard for modern Linux distributions. However, it is completely ABI-incompatible with both soft-float and soft-fp layouts. A binary compiled as hard-float cannot call into a soft-float library; because the calling conventions disagree, the floating-point arguments land in the wrong registers entirely.

This incompatibility is not theoretical. It is the reason the float ABI of your toolchain must precisely match the libraries residing in your sysroot. Mixing them can produce binaries that compile and link cleanly, only to trigger silent computation errors or immediate crashes at runtime.

The specific FPU variant also matters. For example, ARM hardware implementations are identified by generations such as VFPv3, VFPv4, and NEON – ARM's SIMD extension providing wide vector operations. The compiler uses this profile to determine exactly which instructions it can safely emit. Specifying a more capable FPU allows for highly optimised code generation, but executing that binary on a processor lacking that specific hardware extension will instantly trigger an illegal instruction fault.

Endianness

Endianness describes the byte ordering a processor uses when storing multi-byte values in memory. For example, a 32-bit integer with the value `0x12345678` occupies four bytes of memory; the fundamental question is which of those bytes resides at the lowest memory address.

- **Big-endian** processors store the most significant byte first: `0x12` sits at the lowest address, and `0x78` sits at the highest. Traditional MIPS configurations, many PowerPC embedded cores, and most network processors

use big-endian ordering. Because this is also the standard format for network protocols, it is frequently referred to as network byte order.

- **Little-endian** processors store the least significant byte first; in our example, `0x78` is placed at the lowest address, and `0x12` is at the highest. The x86/x86_64 architectures and ARM cores in their default configurations all operate in little-endian mode.

Some architectures are bi-endian, meaning they can be configured to support either byte order. While ARM and MIPS both possess this capability, the vast majority of modern ARM Linux deployments run strictly little-endian. PowerPC embedded cores, by contrast, have historically defaulted to big-endian, though little-endian PowerPC Linux configurations do exist.

RISC-V stands apart by being exclusively little-endian by specification. Omitting a big-endian mode was a deliberate design decision by its creators to reduce hardware implementation complexity and completely eliminate an entire class of toolchain configuration errors.

Endianness is not just a runtime hardware trait – it directly alters the structure of object files and executables. A big-endian object file cannot be linked by a little-endian linker; every single tool in your cross-compilation pipeline must agree on the target's byte order. Mismatching this setting results in immediate linker failures or, in the absolute worst-case scenario, a binary that executes but interprets all multi-byte data as garbled values.

The Compounding Effect

These properties do not exist in isolation; they interact and compound. A choice of ISA generation constrains which FPU variants are available. The FPU variant constrains the float ABI choices that make sense. The ABI must be consistent across every compiled component in the system. Endianness must be consistent across the entire toolchain and all libraries.

This is why a cross-compiler system is never a generic, one-size-fits-all tool. It is important to make a distinction here regarding modern compiler design: while a modern compiler binary like Clang is internally "multi-target" – meaning a single executable can technically generate instructions for ARM, MIPS, or x86 on the fly using command-line flags – the broader toolchain

environment itself cannot be generic. Even if Clang emits the correct raw ARM instructions, it still absolutely depends on a target-specific sysroot, matching architecture-specific libraries, and a definitive float strategy to produce a working executable.

Whether you are using a traditional GCC setup where the target is permanently baked into the compiler binary, or a modern LLVM setup where the target is specified at runtime, the reality remains the same. A cross-compiler toolchain is always tied to a specific configuration of a given architecture – a specific ISA generation, a specific ABI, a specific float strategy, and a specific endianness. Every binary it produces carries those assumptions baked in. The next section will show how all of these choices are expressed concisely and unambiguously in the target tuple – the compact string that serves as the identity of a toolchain and encodes everything discussed here into a single, parseable form.

The Target Triple: Structure and Valid Values

The target triple is a string structure that encodes and compacts the identity of a target platform into a parseable form that the entire toolchain can utilise. The word “triple” (or “tuple” in some contexts) is technically a carryover originating from a time when the string had exactly three fields. Modern target triples often have four, and some fields carry compound information within them. The name has stuck regardless, and you will encounter it used universally across GCC, Clang, binutils, and the broader toolchain ecosystem.

The general form is:

```
<architecture>-<vendor>-<operating-system>-<abi>
```

In practice, the vendor field is frequently omitted or replaced with a placeholder, giving the more common three-field form:

```
<architecture>-<operating-system>-<abi>
```

The Fields Are Not Arbitrary

A critical point that is easy to miss: the fields of a target triple are not free-form strings you can populate with whatever seems descriptive. With one exception, every field draws its value from a defined, finite set of recognised identifiers that the GNU build system and LLVM both understand and act upon. Supplying an unrecognised value does not simply produce a warning — it causes the toolchain build system to fail, often deep into the compilation process with a cryptic error message that does not obviously point back to the malformed triple.

- **The architecture field** must be a value recognised by the toolchain's configuration machinery. Examples include `arm`, `aarch64`, `mips`, `mipsel`, `powerpc`, `riscv32`, `riscv64`, and `x86_64`. These map directly to the code generation back-end inside the compiler. An unrecognised architecture field means the build system cannot select a back-end, and the compilation fails immediately.
- **The operating system field** must similarly be a recognised value — such as `linux`, `none` (for bare-metal), `darwin`, or `windows`. This field tells the toolchain which OS-specific code paths to activate, which binary format to produce, and what runtime assumptions to make. It acts as a strict selector rather than a descriptive label.
- **The ABI field** is a compound token that simultaneously identifies the C library, the ABI specification, and, in the ARM case, the floating-point calling convention. `gnu` means glibc with standard ABI conventions; `gnueabi` means glibc with the ARM Embedded ABI; `gnueabihf` adds hard-float to that configuration. Similarly, `musl` and `musleabihf` select the musl C library with the corresponding float policy. Each of these is a recognised token with strict semantic meaning.
- **The vendor field** is the sole exception to the non-arbitrary rule. It is a free-form field with no semantic meaning to the build system — it exists purely as a human-readable label and is ignored by configure scripts and compiler back-ends alike. Common values are `unknown`, `none`, `pc`, or a manufacturer name like `broadcom`. You may set it to anything meaningful to you, or omit it entirely, without affecting the behaviour of the toolchain. In most embedded cross-compilation work, it is simply omitted.

The practical consequence of this rigidity is that constructing a target triple is an act of discovery; you must identify the correct recognised values for your specific hardware and software configuration before writing a single line of code. How to perform that discovery in practice is covered in Part 1 for GCC-based toolchains and revisited in Part 3 for Clang-based ones.

To make this field structure concrete, the following two examples decode the exact triples used for the target boards throughout this book.

Decoding `arm-linux-gnueabihf`

For our specific use cases, this is the triple targeting the TinkerBoard and the BeagleBone Black — 32-bit ARM processors running Linux with a hardware floating-point ABI.

Field 1: `arm` — Architecture

The first field identifies the ISA family and, implicitly, its default variant. Here, `arm` means 32-bit ARM, operating in the traditional ARM or Thumb-2 instruction set. It does not by itself specify the exact micro-architecture — that level of detail is supplied separately through compiler flags at build time. What it does establish is the broad instruction set the compiler will target: 32-bit ARM instructions, 32-bit general-purpose registers, and the ARM memory model.

Field 2: `linux` — Operating System

The second field tells the toolchain that the target runs a Linux kernel, which carries several concrete consequences. It instructs the compiler to route system calls through the Linux kernel ABI and directs the linker to produce ELF executables. It also tells the toolchain to expect a C library that provides POSIX interfaces and wraps Linux system calls. Finally, it signals to the build system that concepts like processes, virtual memory, shared libraries, and a dynamic linker exist on the target — assumptions that would be entirely incorrect for a bare-metal target.

Field 3: `gnueabihf` — C Library, ABI, and Float Policy

This field packs the most information, and is best understood by decomposing it character by character:

- **`gnu`** identifies the C library as `glibc` (the GNU C Library). This is the most fully featured C library in the Linux ecosystem, implementing the full POSIX standard and a significant body of GNU extensions. Choosing `gnu` here tells the toolchain to expect `glibc` on the target, to link against its headers and libraries within the sysroot, and to generate code compatible with its internal conventions.
- **`eabi`** specifies the Embedded ABI discussed in the previous section — the modern, precisely specified ARM ABI that governs calling conventions, struct layout, and system call boundaries. As established earlier, EABI is the non-negotiable baseline for modern ARM Linux work.
- **`hf`** stands for hard-float. It specifies that floating-point arguments are passed directly in FPU registers at function call boundaries. This is the most performant option, appropriate for ARM Cortex-A class processors that carry a capable VFP or NEON FPU. As discussed previously, this choice must be consistent across the entire system: the toolchain, the C library, and every library in the sysroot must be built with this exact float ABI.

The full field `gnueabihf` therefore simultaneously compresses the C library, the ABI specification, and the floating-point calling convention into a single compound token.

Decoding `aarch64-linux-gnu`

This is the triple for the Raspberry Pi 4 — a 64-bit ARMv8-A processor running Linux.

Field 1: `aarch64` — Architecture

`aarch64` identifies the 64-bit ARM execution state introduced with ARMv8-A. This is a fundamentally different architecture from 32-bit `arm`, not merely an extension of it. The instruction set is entirely new, the register file is wider (31 general-purpose 64-bit registers versus 16 32-bit registers in ARMv7), and

the calling conventions, stack layout, and system call interfaces are completely redesigned. The compiler treats `aarch64` as a distinct target from `arm`, utilising separate code generation back-ends, separate default flags, and separate ABI rules.

Field 2: `linux` — Operating System

Identical in meaning to the same field in the 32-bit triple. The target runs a Linux kernel, expects ELF binaries, and relies on a POSIX-compatible C library providing system call wrappers.

Field 3: `gnu` — C Library and ABI

In the 64-bit triple, the third field simplifies to just `gnu` — denoting `glibc` without an explicit EABI or float suffix. This omission reflects two streamlined facts about the AArch64 architecture:

1. The ABI for AArch64 — the AAPCS64 — is the sole standard ABI defined for this architecture. There is no legacy "old" ABI to distinguish from, meaning an explicit `eabi` marker is redundant.
2. Hardware floating-point support is mandatory in the AArch64 specification; the architecture natively requires a floating-point and SIMD unit. Because there are no soft-float or soft-fp variants to choose between, the float ABI is always hard-float, making the `hf` suffix unnecessary.

The compactness of `gnu` in this position reflects a cleaner, more uniform architecture specification with fewer legacy variants to track and fewer configuration dimensions to manage.

How the Triple Drives Toolchain Configuration

The triple is not merely a label — it is an active input to the toolchain build system. When you configure GCC, binutils, or cross compiler generation tools like Crosstool-NG with a target triple, the build system uses it to:

- **Select the correct code generation backend.**

  - *In GCC and Monolithic Toolchains*: GCC contains separate backend implementations for each supported architecture. The `arm` versus `aarch64` distinction in the first field routes the entire compilation pipeline to a completely different set of instruction selection, register allocation, and code emission logic.
  - *In Clang and LLVM-Based Toolchains*: Unlike GCC's compile-time lock, Clang natively ships with all code generation backends built into a single compiler executable. When you pass the target triple to Clang via the `--target` flag at runtime, the triple acts as a dynamic switch. It immediately tells the compiler which internal backend to activate for that specific file, shifts its header search paths to look for the matching target sysroot, and alters its internal assumptions regarding default macros, structural alignment, and calling conventions on the fly.
- **Set default compiler flags.** The triple establishes the baseline configuration that applies when no explicit flags are given. A toolchain built for `arm-linux-gnueabihf` will default to hard-float, EABI conventions, and 32-bit ARM instruction generation without any additional flags being specified. These defaults can be overridden per-compilation, but they represent the expected common case for that target.
- **Determine the sysroot layout expectations.** The C library field tells the toolchain where to look for headers and libraries within the sysroot, and which library filenames to expect. A `gnu` toolchain expects `libc.so.6`, glibc's standard filename, whereas a `musl` toolchain would expect different library names and a different header layout entirely.
- **Name the toolchain executables.** The cross-compiler binaries themselves are prefixed with the triple: `arm-linux-gnueabihf-gcc`, `arm-linux-gnueabihf-ld`, `arm-linux-gnueabihf-objdump`. This naming convention allows multiple toolchains targeting different architectures to coexist on the same host machine without conflict, and it allows build systems to select the correct toolchain by simply constructing the right executable name from the known target triple.
- **Validate binary compatibility.** The linker uses the triple encoded in object files to verify that all inputs to a link step were compiled for the same target. Attempting to link an `arm` object file into an `aarch64` executable produces an error immediately – the triple mismatch is caught before a broken binary can be produced.

Summary

To summarise — the target triple is ultimately a precision instrument for eliminating ambiguity. Embedded systems work involves an enormous range of hardware configurations, C library choices, ABI variants, and float policies — far more variation than desktop or server development ever encounters. The triple is the mechanism by which all of that variation is named, tracked, and kept consistent across the compiler, the linker, the C library, the sysroot, and the build system.

When something goes wrong in a cross-compilation setup — a symbol not found, a binary that crashes on startup, a library that refuses to link — the triple is often the first place to look. Does the triple of the toolchain match the triple the C library was built for? Does the float suffix match what the sysroot libraries expect? Is the architecture field consistent with the actual CPU on the target board? These are the questions the triple is designed to make answerable quickly and unambiguously.

You will work with these triples constantly throughout the chapters ahead. By the time you have built your first toolchain from scratch, the fields will be second nature.

C Library Choices — glibc, musl, uClibc-ng, Newlib, and Picolibc

The C library is one of the most consequential choices you make when configuring a cross-compiler. It shapes the ABI of every binary the toolchain produces, determines which standard interfaces are available to application code, and sets a floor on the runtime footprint of anything that runs on the target. Yet in practice, many developers treat it as an afterthought, accepting whatever default a pre-built toolchain ships with — and then encounter mysterious runtime failures or bloated binaries that trace directly back to a C library mismatch or an inappropriate choice for the target environment.

This section surveys the C libraries you will encounter most frequently in embedded Linux and bare-metal work, examines the design philosophy behind each, and provides a decision framework for choosing the right one before you begin a toolchain build. The choice made here propagates through every subsequent decision, sysroot construction, ABI flags, link-time options, and

ultimately the behaviour of deployed software, so it would be well worth the time.

glibc — The GNU C Library

glibc has been a cornerstone of the GNU project since 1988 and is the de facto standard C library for the vast majority of mainstream Linux distributions — Debian, Ubuntu, Fedora, Red Hat Enterprise Linux, and virtually every other general-purpose Linux system. It is the reference implementation against which POSIX compliance is most thoroughly measured, and it is the C library against which most Linux software in the world has been built and tested.

Design Philosophy

glibc was designed to be complete before it was designed to be small. Its philosophy is one of maximalism and backward compatibility — supporting the full sweep of C standards from C89 through C11, the complete POSIX specification, and a large body of GNU-specific extensions that have become de facto standards in the Linux ecosystem. Performance is treated as a first-class concern: glibc contains architecture-specific implementations of critical routines such as `memcpy` and `strlen` that take advantage of SIMD instructions on each supported architecture.

This philosophy produces a library that is extraordinarily capable and battle-tested, but one that makes significant assumptions about its environment — virtual memory, a dynamic linker, a writable heap, and thread-local storage support in the kernel. These assumptions are valid on any modern Linux system running on application-class hardware, but they rule glibc out for many constrained embedded targets. It also carries run-time dependencies that make truly portable static binaries impractical, which has licensing implications discussed below.

Footprint

A minimal glibc installation occupies several megabytes of storage. The shared library alone is typically 1.5MB to 2MB on ARM. Dynamic linking is the practical norm for glibc-based systems, which requires the shared library to be present on the target filesystem.

POSIX Compliance

glibc's POSIX compliance is essentially complete. It implements the full POSIX.1-2017 standard, large file support, thread safety across the entire API, robust locale and internationalisation support, and a comprehensive set of mathematical functions.

Typical Use Cases

glibc is the right choice when the target is an application-class Linux system with sufficient RAM and storage, when binary compatibility with mainstream Linux distributions is required, or when the software being built depends on third-party libraries that were themselves built against glibc.

Licensing

glibc is licensed under the **LGPL-2.1**. Dynamic linking in proprietary applications is permitted. Static linking carries obligations around providing relinkable object files — in commercial products, this warrants legal review, and is one practical reason musl is increasingly preferred where static linking is desirable.

musl — The Lightweight POSIX C Library

musl was released in 2011 as a clean implementation of the C standard library with a clear design philosophy in mind: correctness, simplicity, and a small static footprint. It is the default C library for Alpine Linux and an increasingly common choice for embedded Linux work where glibc's weight is unjustifiable and too much for the system.

Design Philosophy

Where glibc targets completeness and performance, musl aims for correctness and simplicity. Every interface is implemented to the letter of the relevant standard, without extensions or legacy compatibility shims. The entire library — standard C, pthreads, and the dynamic linker — are compiled into a single binary, preventing library version mismatching and allowing atomic upgrades. The codebase is deliberately small and auditable, designed to be something a competent engineer can read and understand in its entirety.

Footprint

musl's static footprint advantage over glibc is substantial and too much to ignore. A statically linked hello-world binary against musl is approximately 132KB on ARM. The equivalent glibc binary is approximately 892KB — representing a roughly seven-fold increase in binary overhead. This makes static linking a practical deployment strategy for musl-based systems, which simplifies deployment since a statically linked binary carries no runtime library dependencies and thus makes execution much simpler, robust, and less error-prone.

POSIX Compliance

musl implements the full POSIX.1-2008 standard and most of POSIX.1-2017, with thread safety throughout. It deliberately omits GNU extensions and legacy glibc-specific interfaces. Software written to standard POSIX interfaces will compile and run correctly against musl; software that relies on glibc-specific behaviour may require porting work.

Typical Use Cases

musl is the right choice when static linking is preferred, when the target has limited storage, when security auditability matters, or when licensing simplicity is a priority. It is the modern default for constrained embedded Linux work where glibc is too heavy.

Licensing

musl is licensed under the **MIT license**. There are no copyleft obligations of any kind — proprietary applications may link against musl statically or dynamically without restriction, making it the cleanest option for commercial embedded products.

uClibc-ng — The Embedded Linux C Library

uClibc-ng is the actively maintained fork of the original uClibc project, established in 2014 after the original stalled. It has accumulated more than two decades of embedded-specific optimisation and is the C library of choice for several important embedded Linux environments.

Design Philosophy

uClibc-ng's main and defining characteristic is configurability. Using the same Kconfig system as the Linux kernel, individual features such as IPv6, locale handling, wide character support, floating-point stdio, and thread support can be enabled or disabled at build time. This allows you to produce a C library tailored precisely to the requirements of a specific target, omitting everything the software does not use. This configurability is both its strength and its complexity: a uClibc-ng build with a required feature disabled will produce runtime failures that are not immediately obvious from the build output and thus very hard to debug.

Footprint

When configured aggressively for a specific use case, uClibc-ng can produce the smallest runtime footprint of any Linux-targeted C library discussed here. On MIPS and older ARM targets, a shared library in the low hundreds of kilobytes is achievable.

POSIX Compliance

uClibc-ng targets ANSI/ISO C99 and SUSv3 compliance, with some POSIX features disabled by default. Compliance is configuration-dependent, and certain interfaces have known behavioural differences from the standard. It is best suited for environments where the software stack is well-understood and controlled.

Typical Use Cases

uClibc-ng is the right choice when the target runs Linux on a processor **without a memory management unit (MMU)** — uClibc-ng supports no-MMU Linux configurations that neither glibc nor musl can handle, making it the only viable option for certain older processors, like older MIPS processors, MMU-less ARM variants, and legacy industrial controllers. It is also the historical C library for OpenWrt-based targets.

Licensing

uClibc-ng is licensed under the **LGPL-2.1**, with the same static linking considerations as glibc.

Newlib and Picolibc — The Bare-Metal C Libraries

Newlib and its modern derivative, Picolibc, occupy a fundamentally different category from the three libraries mentioned above. They are not Linux C libraries – they have no concept of Linux system calls, processes, or a dynamic linker. They are designed specifically for bare-metal and deeply embedded environments where there is no operating system, and they are the standard choices for cross-compilers targeting Cortex-M micro-controllers, RISC-V embedded cores, and DSPs.

Newlib

Newlib's design is built around a syscall stub mechanism where the developer implements a small set of hardware-specific bridge functions that connect the C library to the target hardware. The library calls `_write` when it needs to output characters, so you implement `_write` to send bytes to a UART, for example. It calls `_sbrk` when it needs heap memory, and you implement `_sbrk` to carve memory from a region defined in your linker script. This model gives you the entire C standard library implemented and tested, requiring you to provide only a handful of functions that are genuinely hardware-specific.

Newlib-nano is a size-optimised variant with simplified `printf` and `malloc` implementations, and is the default choice for Cortex-M targets where code size is a primary constraint.

Picolibc

Picolibc is a modern fork of newlib created in 2018 that pushes bare-metal optimisation further for 32-bit and 64-bit embedded systems. It delivers a meaningfully smaller footprint than newlib through a more efficient stdio implementation, and its native compiler TLS support reduces per-task RAM overhead in RTOS environments. For new bare-metal designs, particularly those targeting RISC-V cores, Picolibc is the modern recommendation. It also carries uniformly clean BSD licensing, making legal review for commercial products straightforward.

Typical Use Cases

Newlib or Picolibc is the right choice when the target has no operating system, when building an RTOS-based system that needs the C standard library without a full POSIX environment, or when code size is tightly constrained.

Licensing

Newlib uses a collection of BSD and MIT-style permissive licenses with no copyleft obligations. Picolibc has explicitly removed all non-BSD code, giving it a uniformly clean licensing status that simplifies commercial use.

C Library Selection: A Practical Decision Framework

Choosing a C library should not be a matter of guesswork. By evaluating your target platform through the following five sequential questions, you can precisely isolate the correct library for your environment.

Step 1: Does the target platform run an Operating System?

- **No (Bare-Metal / RTOS-only):** Your choice is immediately narrowed to **Newlib** or **Picolibc**. For new designs, **Picolibc** is the modern recommendation. **Newlib-nano** remains highly appropriate for legacy platforms or environments where upstream vendor toolchain integration is already firmly established.
- **Yes (Linux Kernel):** Proceed to Step 2.

Step 2: Does the target CPU possess a Memory Management Unit (MMU)?

- **No: uClibc-ng** is your only viable choice. Both `glibc` and `musl` natively require virtual memory management and an MMU to function.
- **Yes:** Proceed to Step 3.

Step 3: Is the choice already dictated by a specific Linux Distribution?

- **Yes:** If your target hardware runs an established distribution or firmware framework, your toolchain's C library **must** match that distribution's runtime exactly. Alpine Linux strictly uses `musl`. OpenWrt relies on `musl` in modern releases. If you are using Yocto or Buildroot, they default to `glibc` but allow you to explicitly select `musl` or `uClibc-ng`. Matching this is non-negotiable; mixing them at build time creates a systemic ABI mismatch that will instantly fail at runtime.
- **No (Custom RootFS / From Scratch):** Proceed to Step 4.

Step 4: Are Storage and RAM heavily constrained?

- **No (Ample Resources):** Choose **glibc**. It is the natural default for application-class hardware. It provides the richest API surface, the absolute best compatibility with third-party pre-compiled binaries, and the most thoroughly optimised, battle-tested performance routines.
- **Yes (Resource-Constrained):** Proceed to Step 5.

Step 5: Is the software stack entirely known and controlled?

- **Yes: musl** is the definitive choice for modern, resource-constrained embedded Linux targets. Its clean static footprint, permissive MIT license, and strict standard correctness make it the ideal default when `glibc` is simply too heavy.
- **No (Legacy / Proprietary Binaries Present):** If your software stack relies on closed-source third-party binaries or legacy applications built specifically with older `uClibc` compatibility in mind, then **uClibc-ng** remains your best alternative to ensure backward compatibility.

Decision Table

Scenario	Recommended Library
Bare-metal / RTOS only	Picolibc (new) or Newlib-nano (legacy)
Linux, no MMU	uClibc-ng
Linux, distro-based	Match the distro
Linux, custom, ample resources	glibc
Linux, custom, constrained, controlled stack	musl
Linux, custom, constrained, legacy binaries	uClibc-ng

Comparison Table

	glibc	musl
Target environment	Linux	Linux
Design priority	Completeness + perf	Correctness + size
Static footprint	Large (~892KB hello)	Small (~132KB hello)
POSIX compliance	Full	High
Third-party compat	Excellent	Good
License	LGPL-2.1	MIT
Typical use	Application-class SBCs	Constrained Linux

Continued — uClibc-ng and Newlib / Picolibc:

	uClibc-ng	Newlib / Picolibc
Target environment	Linux (incl. no-MMU)	Bare-metal / RTOS
Design priority	Configurability	Bare-metal fit
Static footprint	Very small (configured)	Very small
POSIX compliance	Partial	ISO C only
Third-party compat	Fair	N/A
License	LGPL-2.1	BSD / MIT
Typical use	Routers, IoT, no-MMU	MCUs, DSPs, RISC-V

What This Book Uses

Throughout Parts 1 through 3, the toolchains we construct will target `glibc`, which we will use to build the software stack for our 32-bit TinkerBoard single-board computer (SBC) demonstrations. Later, in Part 4, we will shift our focus to constructing a streamlined, 64-bit `aarch64` `musl`-based toolchain specifically tailored for our Raspberry Pi 4 projects.

The decision framework detailed above is provided so that when you move beyond this book's specific hardware platforms to your own production

designs, you possess the structural reasoning required to select the optimal C library independently, rather than simply defaulting to `glibc` out of familiarity.

The Bootstrap Problem

Before writing a single configuration flag or downloading a single source tarball, there is a conceptual problem worth confronting directly — one that sits at the foundation of everything this book covers. It is called the bootstrap problem, and understanding it is what separates developers who can reason about toolchain failures from those who simply follow recipes and hope for the best.

A Simplified Explanation of the Problem

To compile a program, you need a compiler. To build a compiler, you need to compile it. This is not a paradox in general — your host machine already has a compiler, so you can use it to build another one. But cross-compilation introduces a twist that makes the problem significantly more interesting.

When you build a cross-compiler for ARM on an x86-64 host, you are building a compiler that will run on x86-64 but produce code for ARM. The compiler itself is an x86-64 binary — your host's existing compiler can build it without difficulty. So far, no problem.

The complication arises when that cross-compiler needs to build things that depend on the target's runtime environment. Specifically, it needs to build the C library for the target. And the C library, as established in the previous section, is not just a collection of functions — it is deeply intertwined with the compiler itself. The C library depends on the compiler's internal headers. The compiler depends on the C library to build a fully functional runtime. Each one, in its complete form, requires the other to exist first. This is the classic chicken-and-egg problem at the heart of cross-compiler construction.

This is the bootstrap problem: the mutual dependency between the compiler and the C library creates a situation where neither can be fully built without the other already being present.

Why You Cannot Simply Build the Compiler Once

A natural response is to ask why the compiler cannot simply be built in one pass and then used to compile the target's C library. The reason comes back to the target architecture and the ABI.

The host's C library is built for the host architecture – x86-64, with the host's calling conventions and the host's system call interface. The target's C library needs to be built for the target architecture – in this book's case ARM and AArch64 – with their respective calling conventions and system calls. These are not compatible. As established earlier in this chapter, a compiler built with knowledge of only the host environment cannot correctly produce a C library for a fundamentally different architecture and ABI.

More concretely: when GCC builds the C library, it requires a set of internal compiler support routines – low-level code that handles operations the target hardware may not support natively, such as software floating-point emulation or integer division. For those routines to work correctly on the target, they must themselves be compiled for the target. But compiling them for the target requires a working compiler for the target. And a fully working compiler for the target requires those routines to already exist.

The circularity is real and unavoidable. The solution is not to break the circle – it is to traverse it in carefully ordered stages.

The Three-Stage Bootstrap for GCC Based Toolchains

The three-stage bootstrap is the standard solution to this problem, and it is the architectural pattern underlying every from-scratch toolchain build covered in this book. Each stage produces something that makes the next stage possible, with dependencies resolved incrementally rather than all at once.

Stage 1 — A Minimal Compiler With No C Library

The first stage builds a minimal GCC that can compile C code for the target architecture, but makes no assumptions about a C library existing on the target at all. GCC is specifically designed to support this – it can be configured to operate without any knowledge of a target C library, producing just enough compiler machinery to process C code and generate target assembly. The exact

mechanics of how this is configured are covered in detail when you perform the build in later chapters.

The result is a cross-compiler that is deliberately limited. It cannot link complete programs. It cannot use standard library headers. What it can do is compile low-level C code that makes no use of the standard library — which is precisely what is needed to build the next component.

Stage 2 — The C Library

With a minimal compiler in hand, Stage 2 builds the C library for the target. The Stage 1 compiler is used to compile glibc's source code for the ARM target, producing the actual target C library — the headers, the compiled library, the startup objects, and the dynamic linker.

This is the pivotal stage. Once the C library exists for the target, the fundamental dependency is resolved — there is now a real C library present in the sysroot, built for the correct architecture with the correct ABI. Every subsequent compilation step has a complete runtime environment to build against.

It is worth noting why Stage 2 is possible at all. glibc's own source code is designed to be bootstrappable — it can be compiled by a minimal compiler that lacks a complete runtime environment, because glibc is itself the thing that provides that runtime environment. This careful design in glibc's internals is what makes the staged approach work.

Stage 3 — The Full Compiler

With a complete C library now present in the sysroot, Stage 3 rebuilds GCC from scratch — this time fully configured, with full knowledge of the target's C library, headers, and ABI. This is no longer the limited Stage 1 compiler. This is a complete, production-grade cross-compiler capable of building real software for the target: programs that use the full standard library, C++ with exception handling, shared libraries, and optimised code with full knowledge of the target's ABI and CPU capabilities.

The Stage 3 compiler is the toolchain you will actually use to build software for the target. Stages 1 and 2 were not wasted work — they were the scaffolding required to make Stage 3 possible.

Why This Matters Beyond Theory

The three-stage bootstrap is not just an intellectual curiosity — it is the direct explanation for a significant portion of the complexity you will encounter throughout this book.

It explains why building a cross-compiler takes considerably longer than installing one. Each stage requires a full GCC configure and build cycle, and building GCC more than once is not inefficiency — it is the irreducible minimum imposed by the dependency structure.

It explains why the order of operations in a toolchain build is rigid and unforgiving. The binary utilities must exist before the Stage 1 compiler because the compiler needs an assembler and linker during its own build. The Stage 1 compiler must exist before the C library because the C library needs a cross-compiler to build against. The C library must exist before the Stage 3 compiler because the full compiler needs a complete runtime to link against. Violating this order does not produce a slightly broken result — it produces a build that fails with errors that are genuinely confusing if you do not understand why the order matters.

It explains why tools like Crosstool-NG exist. The three-stage bootstrap involves downloading, patching, configuring, and building multiple large packages in a precise order with flags that must remain consistent across all stages. Automating this is not laziness — it is sound engineering. The Crosstool-NG build chapters use it for exactly that reason before the manual GCC build chapters take you through the same process by hand so you understand every step.

It explains why Clang's relationship with GCC is what it is. Clang cannot perform this same bootstrap from a clean slate — it lacks the mechanism that makes Stage 1 GCC possible. Building a Clang-based cross-compiler therefore builds on top of the GCC toolchain produced in the earlier build chapters. The three-stage bootstrap was already done, and Clang leverages its results.

The Foundational Challenge

The bootstrap problem is the foundational challenge of this entire book not because it is the hardest technical problem you will encounter — the actual build steps, once understood, are systematic and repeatable — but because it is the conceptual key that unlocks everything else.

Every decision in the toolchain build follows inevitably from the structure of the three-stage bootstrap. When something goes wrong – and in toolchain work, something always goes wrong – understanding which stage dependency was violated is what allows you to reason your way to a fix rather than searching blindly for a solution.

Developers who understand the bootstrap problem do not need to memorise build procedures. They can derive them. That capacity for independent reasoning is what this book is ultimately trying to build, and the bootstrap problem is where it starts.

How the Dependency Cycle Changes with Clang and LLVM

When the toolchain discussion shifts from GCC to the LLVM and Clang ecosystem, the nature of the bootstrap problem changes substantially. The three-stage dance required by GCC simplifies considerably – not because the dependency between the compiler runtime and the C library disappears entirely, but because Clang's architecture eliminates the most painful part of the GCC bootstrap: the need for a crippled Stage 1 compiler.

Understanding why requires looking at the fundamental architectural difference between how GCC and Clang approach cross-compilation.

Clang is a Universal Cross-Compiler by Design

The root cause of GCC's three-stage bootstrap is that a standard GCC binary is bound to a single target architecture at the time it is compiled. A GCC binary configured for x86-64 cannot emit ARM code. To produce ARM binaries you must build a completely separate GCC binary from scratch, and that process triggers the Stage 1 dependency problem described earlier in this chapter.

Clang takes a fundamentally different approach. Every Clang binary is a native cross-compiler out of the box. A single Clang executable running on your x86-64 host already knows how to compile and emit machine code for x86-64, ARM, AArch64, RISC-V, MIPS, and other architectures simultaneously – all backends are compiled into the same binary. To target a different architecture you do not rebuild the compiler; you simply tell it which target to use at invocation time.

This single architectural decision eliminates Stage 1 of the GCC bootstrap entirely. There is no need for a crippled preliminary compiler because the

host's existing Clang can already target ARM directly. The compiler exists; it just needs the target's runtime support and C library to build against.

compiler-rt Replaces libgcc

The second source of GCC's circular dependency is `libgcc` – the low-level compiler support library that provides software implementations of operations the target hardware may not support natively. Because `libgcc` is deeply embedded within GCC's own build lifecycle, you cannot easily separate the compiler binary from these target-specific runtime routines. They are built together, which is precisely what creates the deadlock.

LLVM breaks this coupling by separating its equivalent functionality into a completely standalone project called `compiler-rt`. Because the host Clang can already target ARM without being rebuilt, `compiler-rt` can be compiled directly for the target architecture in a single straightforward pass – no special preliminary compiler required, no circular dependency to navigate.

The result is that the complex three-stage GCC bootstrap collapses into a simpler linear progression:

```
Host Clang
│
├── 1. Compile compiler-rt for the target
│
├── 2. Compile the target C library (against compiler-rt)
│
└── 3. Full Clang cross-compilation environment ready
```

One practical nuance is worth flagging here. Even though `compiler-rt` is architecturally independent of the C library, its source files require a small number of standard C headers – things like `stddef.h` and `features.h` – simply to compile. This means that in a pure from-scratch LLVM build with no GCC present on the host, a fully clean Step 1 is not always achievable without either supplying those headers in isolation first or using build flags that instruct `compiler-rt` to operate in a freestanding mode. The practical resolution of this is covered in full when we get to the Clang toolchain build – the point here is that the linear progression, while substantially simpler than the GCC three-stage bootstrap, is not entirely without its own ordering subtleties.

The ordering still matters – `compiler-rt` must exist before the C library, and the C library must exist before fully linked target binaries can be produced.

The dependencies have not vanished; they have simply been straightened from a circle that required staged navigation into a line that can be walked in one direction, with one small caveat at the start of that line.

A Note on glibc and musl

It is worth being precise about the C library side of this picture. The claim that Clang can compile any C library for a target in a clean single pass is true today for recent versions of both glibc and the Linux kernel — both can now be compiled with Clang — but this was not always the case. For most of their history both projects were deeply coupled to GCC-specific language extensions and compiler behaviours, and Clang compatibility was an afterthought that took years of work to achieve properly.

musl tells a cleaner story here. Its codebase was written with standards compliance as a primary goal, which means it has always been more naturally compatible with Clang than glibc. For a Clang-based cross-compilation workflow, musl is the more natural C library pairing and the one with the smoother build experience. The Clang toolchain build chapters reflect this — when we construct the Clang toolchain, musl is used rather than glibc, and the reasons are discussed in full at that point.

Why the GCC Bootstrap Still Matters

If Clang handles cross-compilation so much more cleanly, a reasonable question is why the industry continues to invest heavily in mastering the GCC bootstrap — and why this book works through it in depth before reaching the Clang toolchain build.

There are two concrete reasons.

First, the embedded Linux ecosystem remains substantially built around GCC. The vast majority of production Buildroot configurations, Yocto layers, vendor BSPs, and industrial sysroots use GCC as the reference toolchain. glibc itself, despite gaining Clang compatibility, is still developed and tested primarily against GCC. Clang compatibility in these environments is improving steadily but is not yet the default assumption in most production embedded workflows.

Second, and more importantly for the reader of this book: mastering the GCC bootstrap teaches you exactly where the seams are between the

compiler, the low-level architecture support routines, and the C library. That understanding transfers directly to Clang. A developer who can reason through a GCC bootstrap failure — who understands why Stage 1 exists, why the sysroot must be populated before Stage 3, and why the C library ABI must match across all components — will find the Clang cross-compilation setup straightforward by comparison. The concepts are the same; the execution is simply less painful.

The GCC bootstrap is where the hard understanding is earned. The Clang toolchain build chapters are where that understanding pays dividends.

Comparing the Two Bootstrap Approaches

To make the contrast concrete, the table below summarises the key differences between the GCC and Clang bootstrap processes as they apply to the builds in this book:

	GCC	Clang / LLVM
Cross-target capability	One binary per target, built at configure time	All targets in one binary, selected at invocation
Compiler runtime	`libgcc`, tightly coupled to GCC build lifecycle	`compiler-rt`, standalone and independently buildable
Bootstrap stages	Three — minimal compiler, C library, full compiler	Effectively two — compiler-rt, then C library
Stage 1 requirement	Mandatory — crippled compiler needed to break the cycle	Eliminated — host Clang already targets ARM
Sysroot catch	Must be populated between Stage 1 and Stage 3	Minor header dependency at compiler-rt build time
C library pairing	glibc — the natural and reference pairing	musl — the cleaner and more natural pairing

	GCC	Clang / LLVM
Industry adoption	Reference standard for embedded Linux sysroots	Growing adoption, not yet the embedded default
Learning value	Teaches the fundamental seams of compiler construction	Rewards the understanding built by the GCC bootstrap

The two approaches are not in competition — they are complementary. GCC builds the foundation of understanding; Clang demonstrates what that foundation makes possible when the toolchain ecosystem evolves to remove historical constraints. The Clang toolchain build chapters are where that payoff arrives.

The Target Boards

Throughout this book you will build toolchains for two physical target boards. Each was chosen deliberately — not for novelty, but because together they cover the two dominant ARM architectures a professional embedded developer encounters in practice, and each one introduces a specific dimension of cross-compilation complexity that the other does not. A third board — the BeagleBone Black — appears in Volume 2 and is introduced briefly here so the reader understands the complete hardware landscape across both volumes.

TinkerBoard S R2.0

The TinkerBoard S R2.0 is the primary workhorse of this volume. It is present from the first toolchain build in the first practical sections of this book through to the containerisation exercises in the latter part, and every core technique is demonstrated on it before being extended to the second board.

Hardware Specifications

The TinkerBoard S R2.0 is built around the **Rockchip RK3288** system-on-chip, a quad-core ARM Cortex-A17 processor running at up to 1.8GHz. The Cortex-A17

implements the **ARMv7-A** architecture — 32-bit ARM with hardware floating-point support via a VFPv4 FPU and NEON SIMD extensions. The board ships with 2GB of LPDDR3 RAM and 16GB of onboard eMMC storage, and runs a mainstream ARM Linux distribution built against glibc.

SoC	Rockchip RK3288
CPU	Quad-core ARM Cortex-A17 @ 1.8GHz
Architecture	ARMv7-A, 32-bit
FPU	VFPv4 + NEON
RAM	2GB LPDDR3
Storage	16GB eMMC
OS	Linux (Debian / Ubuntu derivatives)

Why This Board

The RK3288 Cortex-A17 is an excellent teaching platform because ARMv7-A demands the most from a toolchain builder of any architecture covered in this volume. The choice between hard-float and soft-float matters here in a way it does not on AArch64. The EABI must be specified explicitly. The FPU variant must be identified and matched. Every decision discussed in the CPU architecture section of this chapter has a concrete, non-trivial answer for this board, which means building a toolchain for it teaches the full reasoning process rather than a simplified subset of it.

It is also worth noting that the ARMv7-A toolchain built for the TinkerBoard is intentionally constructed as a generic toolchain — not locked to the Cortex-A17 at build time. CPU-specific optimisations are applied at compile time rather than baked into the toolchain binary itself. This is a deliberate design decision that will be explained and motivated fully when the build is performed in Part 2, and it has a practical consequence that matters across both volumes: the same toolchain binary can target the TinkerBoard and the BeagleBone Black without being rebuilt.

When It Appears

The TinkerBoard S R2.0 is the target for all toolchain builds in the Crosstool-NG chapters and the manual GCC build chapters. It is the primary ARMv7-A

target in the Clang/LLVM cross-compilation chapters and the Docker-based toolchain containerisation chapters. It carries over into Volume 2 as one of the deployment targets.

No hardware yet? If you do not have a TinkerBoard S R2.0 or a compatible ARMv7-A board available, Chapter 3 of this part walks through setting up a Docker-based ARMv7-A environment on the host that supports all the binary testing exercises in this volume. You can follow every chapter using that environment and move to real hardware when ready.

Raspberry Pi 4

The Raspberry Pi 4 is the second target board, introduced when the focus shifts to building a toolchain for a 64-bit ARM processor — once the foundational toolchain concepts have been established on the TinkerBoard.

Hardware Specifications

The Raspberry Pi 4 is built around the **Broadcom BCM2711** system-on-chip, a quad-core ARM Cortex-A72 processor running at up to 1.8GHz. The Cortex-A72 implements the **ARMv8-A** architecture — a 64-bit architecture that is fundamentally different from the ARMv7-A of the TinkerBoard, not an extension of it. The board is available in multiple RAM configurations; the 4GB variant is used throughout this volume. It runs a mainstream 64-bit Linux distribution built against glibc.

SoC	Broadcom BCM2711
CPU	Quad-core ARM Cortex-A72 @ 1.8GHz
Architecture	ARMv8-A, 64-bit
FPU	Mandatory hardware FPU + NEON
RAM	4GB LPDDR4
Storage	microSD / USB
OS	Linux (Raspberry Pi OS 64-bit / Ubuntu)

Why This Board

The Raspberry Pi 4 introduces the ARMv8-A architecture and the AArch64 execution state. As established in the CPU architecture section, AArch64 is not an incremental upgrade from ARMv7-A — it is a different instruction set, a different register file, and a different ABI. Building a cross-compiler for it therefore requires repeating the toolchain construction process with a completely different target triple, different default flags, and different ABI assumptions.

This repetition is intentional and instructive. By the time you build the AArch64 toolchain in Part 2 you will have already built an ARMv7-A toolchain from scratch. The contrast between the two builds makes the significance of each configuration choice tangible in a way that studying either board in isolation would not.

Unlike the generic ARMv7-A toolchain built for the TinkerBoard, the Raspberry Pi 4 toolchain is built CPU-specific — locked to the Cortex-A72 microarchitecture at toolchain build time. This contrast between the generic and CPU-specific approaches is itself a teaching point, and the tradeoffs between them are discussed in full when the builds are performed.

When It Appears

The Raspberry Pi 4 is introduced as the second build target in the manual GCC build chapters and carries through the Clang/LLVM and Docker containerisation chapters alongside the TinkerBoard. It does not appear in Volume 2, where the BeagleBone Black takes the ARMv7-A deployment role.

No hardware yet? If you do not have a Raspberry Pi 4 or a compatible ARMv8-A board available, Chapter 3 of this part walks through setting up a Docker-based AArch64 environment on the host that supports all the binary testing exercises in this volume.

BeagleBone Black

The BeagleBone Black is built around the **Texas Instruments AM335x** SoC, a single-core ARM Cortex-A8 processor running at 1GHz. It implements the same **ARMv7-A** architecture as the TinkerBoard, which means the generic

ARMv7-A toolchain built in this volume targets it without modification — the only difference is the CPU-specific compile flags applied at build time. Its distinguishing characteristic for Volume 2 is its resource constraints: 512MB of RAM and 4GB of eMMC storage. These constraints make discussions of deployment image size, binary footprint, and runtime overhead concrete and necessary rather than academic. Deploying to a board that genuinely feels the difference between a 50MB and a 200MB image produces a different quality of understanding than deploying to a board where storage is effectively unlimited.

SoC	Texas Instruments AM335x
CPU	Single-core ARM Cortex-A8 @ 1GHz
Architecture	ARMv7-A, 32-bit
FPU	VFPv3 + NEON
RAM	512MB DDR3
Storage	4GB eMMC
OS	Linux (Debian derivatives)

The Hardware at a Glance

Board	Architecture	Volume 1	Volume 2
TinkerBoard S R2.0	ARMv7-A, 32-bit	Parts 1–4	Deployment target
Raspberry Pi 4	ARMv8-A, 64-bit	Parts 2–4	Not used
BeagleBone Black	ARMv7-A, 32-bit	Not used	Primary deployment demo

A general note on hardware flexibility applies across all three boards covered in this book. The TinkerBoard S R2.0 can be substituted with any board built around an ARMv7-A processor — a Cortex-A7, Cortex-A9, or Cortex-A15 based board will follow the same toolchain construction process with only

the CPU-specific compile flags adjusted. Similarly, the Raspberry Pi 4 can be substituted with any ARMv8-A board – any Cortex-A53, Cortex-A55, or Cortex-A72 based single-board computer running a 64-bit Linux distribution will work with the AArch64 toolchain built in Part 2. The architectures, ABIs, and toolchain construction principles are what matter – the specific board is simply the concrete surface on which those principles are demonstrated. Where a specific board introduces a genuinely board-specific consideration, that will be called out explicitly in the text.