BEYOND BLINKY

OBJECT-ORIENTED C++ PROGRAMMING

Fun, Fast, and Fearless Embedded Development



Nguyen Hoan Hoang

BEYOND BLINKY

OBJECT-ORIENTED C++ PROGRAMMING

Fun, Fast, and Fearless Embedded Development

Nguyen Hoan Hoang

Beyond Blinky: Object-Oriented C++ Programming

Copyright © 2025 by Nguyen Hoan Hoang. All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law. For permission requests, write to the publisher, addressed "Attention: Permissions Coordinator," at the address below.

Publisher: I-SYST inc.

Legal Disclaimer: The information presented in this book is for educational and informational purposes only. The author and publisher have made every effort to ensure the accuracy and completeness of the information. However, they assume no responsibility for errors, omissions, or any consequences resulting from the use of this information. The reader is responsible for their own use of the information and for any damage or injury that may result from it. The use of this information is at the sole discretion and risk of the reader. All product names, logos, and brands are the property of their respective owners.

First Edition

ISBN: 978-1-0699335-1-5 Print
ISBN: 978-1-0699335-0-8 eBook
ISBN: 978-1-0699335-2-2 Digital

TABLE OF CONTENTS

Preface.....9

PART 1 The Living Architecture 13

Chapter 1 Mythbusting of C++ in Embedded Systems.....15

1.0 Myth 1: "Object-Oriented Programming is bad for embedded".....16

1.1 Myth 2: "C++ makes embedded systems slow and bloated." 20

1.2 Myth 3: "C++ requires dynamic memory, which is unsafe." 21

1.3 Myth 4: "C++ abstractions for portability create endless configuration." 22

1.4 Conclusion: The Tool vs. The Blueprint 22

Chapter 2 Object Modeling — A Living Architecture.....25

The Core Distinction 25

The Orchard Awakens 26

2.0 The Feeling of Good Architecture.....27

2.1 The Land: What not to Objectify 28

2.2 The Roots: The Interface 28

2.3 The Trees: The Device Object.....29

2.4 What the World Sees: The Encapsulation 30

2.5 Different Fruit, Same Roots: The Inheritance.....31

2.6 One Tool, Many Forms: The Polymorphism.....32

2.7 The Pear Arrives: The Scalability.....33

2.8 The Orchard Keeper: Deciding What Becomes an Object.....34

2.9 Walking the Orchard: The Mental Model.....35

2.10 Differences That Matter (and Those That Don't).....36

2.11 Three Tiny Questions (Your On-the-Spot Checklist) 37

2.12 Anti-Patterns to Avoid (Gently but Firmly).....37

Chapter 3 Development Environment.....43

3.0 The Development Process 43

3.1 The Firmware Development Cycle 44
3.2 Requirements for a Professional Workbench 45
3.3 Choosing the Right Workbench: An Evaluation 46
3.4 Workbench Installation 50
3.5 Validating Your Workbench: The First Spark 52
3.6 Building Eclipse Projects Without the GUI 54

Chapter 4 Wield the Tool to Seed the Land.....57

4.0 Open Blinky Project: The First Spark 58
4.1 I/O Pins: The Land 65
4.1.1 GPIO: The Universal Principles 66
4.1.2 Abstraction: A Common Language 66
4.1.3 Configuration: The Blueprint and The Hand-Made 67
4.1.4 The Key to Portability: No Scripts, No BSPs 68
4.1.5 IOPin Control: The Core Functions 69
4.2 Core Clocks: The Orchard's Hum 71
4.3 Timers: The Rhythm.....74
4.4 The CFifo: The Basket.....77

Chapter 5 The Device Interface: The Roots 83

5.0 The Root Interface: A Design Philosophy 84
5.0.1 Communication: The Voice of the Device 84
5.0.2 A Byte's Journey.....85
5.0.3 The Interface, in Human Terms.....85
5.0.4 Why Not "Just Use the HAL"? 86
5.0.5 What Changes vs What Stays.....86
5.0.6 Misconceptions & Clarity Check.....86
5.0.7 The Orchard Test (Quick Self-Quiz).....86
5.1 The DeviceIntrf: A Common Pathway.....87
5.2 UART: The Human Translator 89
5.2.1 UART Benchmark: A Measured Voice.....90
5.2.2 UART MCU vs PC : Same Voice, Different "Dialects" 92
5.3 I²C: Two Wires, Many Branches.....94

5.4 SPI: A Direct, High-Speed Channel.....95
5.5 The Magic of Polymorphism: One Tree, Any Root 96
5.6 Protocol Encapsulation: SLIP Over Any Root.....96
5.7 Beyond the Wires: Swapping Protocols Over Air and Wire 97

Chapter 6 The Device: Fruits of the Orchard.....101

Part I — Mind Map: From Orchard to Objects (no code yet) 103
6.0 Stand in the Orchard (before you touch a keyboard).....103
6.1 Seasons, Roots, Bark, Sap (the four nouns you'll say all year) 103
6.2 Trunk, Branches, Fruits (how OOD becomes touchable) 104
6.3 Read this slowly: what exactly do branches and fruits buy you?.....105
6.3.1 Inheritance vs Polymorphism (pin it to the metaphor).....106
6.4 The Orchard Map (where fruits actually come from) 107
6.5 The Seven Beats (your ritual before any implementation).....108
6.6 Thought Experiments (quick "feel checks" before Part II).....108
Part II — From Map to Code: Growing Real Fruits (runnable pieces) 110
6.7 The Device Class: The Trunk 110
6.8 Inherited Devices: The Main Branches.....111
6.9 A Deeper Look: The Sensor Family Hierarchy.....113
6.10 The Init() Function: The Grafting Pattern.....114
6.11 A Minimal Runnable Example: The First Fruit.....116
6.12 Advanced C++ In Action: The Composite Branch 118
6.12.1 Visualizing the Composite Branch: The Multi-Talented Specialist .118
6.12.2 The Hidden Danger: The Diamond Problem 119
6.12.3 The Solution: Virtual Inheritance.....120
6.13 The Wider Orchard in Practice 120
6.14 Final Checks & Common Pitfalls.....122
6.15 Stretch: Where to Grow Next.....123

Chapter 7 The Wireless Interconnect 127

7.0 A Beginner-Friendly Guide to BLE 127
7.1 Managing The BLE Stack — Treat It As "The Land" 128
7.2 A Simple BLE Advertiser.....129

7.3 BleIntrf: The Wireless Root....131
7.4 Proof in Practice: PRBS over a Wireless Root....133
7.5 The Ultimate Test: Switching Roots at Runtime....134
7.6 Chapter 7 Wrap-Up: The Hive is Alive....135

PART 2 The Art of Cloning — Porting....137

Chapter 8 Startup & Vectors (ARM)....139

8.0 Startup....139
8.1 The Deliverable....140
8.2 Cortex-M Profile Rules....141
8.3 Implementation Template....141
8.4 Extraction workflow....144
8.5 Linker & VTOR Requirements....145
8.6 Validation....147
8.7 Troubleshooting....149
8.8 Done Checklist....149
8.9 What's Next....150

Chapter 9 SystemInit & Clock Tree....151

9.0 The Clock Configuration Blueprint....151
9.1 The Implementation....153
9.2 Verification....154
9.3 Validation....154
9.4 Recommended File Skeleton....155

Chapter 10 GPIO Porting: Configuration & Control....159

Part I — The Configuration Layer....160
10.0 Purpose....160
10.1 Deliverable....160
10.2 API Semantics....160
10.3 Implementation Pattern....161
10.4 Interrupt Handling Models....162
10.5 Validation....162

10.6 Troubleshooting 163
10.7 Done Checklist 163
Part II — The Control Layer 164
10.8 Purpose 164
10.9 Deliverable 164
10.10 Implementation Requirements 164
10.11 Validation 166
10.12 Troubleshooting 166
10.13 Done Checklist 166
10.14 Chapter Summary & Next Steps 167
Chapter 11 Console UART Bring-Up 169
11.0 Purpose 169
11.1 Deliverables 169
11.2 The Configuration Data (UARTCfg_t) 170
11.3 Implementation Summary: A Porter's Guide 171
11.4 Validation Test 1: Console Retargeting 173
11.5 Validation Test 2: PRBS Throughput Benchmark 174
11.6 Troubleshooting 175
11.7 Done Checklist & Next Steps 176
Chapter 12 Timers (LF + HF) 177
12.0 Purpose 177
12.1 Deliverables 177
12.2 Architectural Concept: Unified LF/HF Timer Indexing 178
12.3 The Configuration Data 179
12.4 Implementation Summary: A Porter's Guide 179
12.5 Validation Checklist 180
12.6 Troubleshooting 181
12.7 Done Checklist & Next Steps 182
Chapter 13 Bring-Up Validation Suite (Blinky + Pulse-Train)... 183

13.0 Purpose 183
13.1 Deliverables 184
13.2 Blinky: The First Spark 184
13.3 Minimal Implementation 184
13.4 Pass Criteria 185
13.5 Pulse-Train: The Full System Validation 185
13.6 Configuration and API 186
13.7 Reference Implementation 187
13.8 Pass Criteria (Oscilloscope/Logic Analyzer) 187
13.9 Troubleshooting 188
13.10 Done Checklist & Next Steps 188

Epilogue 191

Appendix A 193

MCU-Centric vs Board-Centric Architecture 193

A.1 Two Approaches To Hardware Abstraction 193
A.2 The Scaling Mathematics 193
A.3 A Concrete Comparison 195
A.4 The Pin Change Test 197
A.5 Connection To Analog Thinking 197

Appendix B Developer Recipes 201

Recipe 1: Adding a New Device 201
Recipe 2: Switching Between I²C and SPI 202
Recipe 3: Logging with UART 202
Recipe 4: Using LED With PWM 202

Appendix C Benchmarks & Methodology 205

MCU Transmitter Application 205
PC Receiver and Verifier 206
Interpreting the Results 207

Appendix D Manual Workbench Setup 209

D.1 Install Cross-Compilers 209
D.2 Install Debug Tools 211
D.3 Installing Eclipse Embedded CDT 215
D.4 Clone the IOsonata & Dependencies 218

Appendix E Create New Project 221

E.1 Create a New Project 221
E.2 Source Code referencing 224
E.3 The Project Settings 228
E.4 Bring Your Creation to Life 234

Appendix F Driver Pattern Cards 235

F.1 The Root Interface (DeviceIntrf) Pattern 235
F.2 The Device Object Pattern 236

Appendix G SystemInit Across Vendors—Production Implementation 237

G.1 Renesas RE01 Implementation 238
G.2 Microchip SAM4L Implementation 246
G.3 STMicroelectronics STM32L4xx Implementation 261
G.4 Side-by-Side Comparison 271
G.5 What This Code Proves 272

Appendix H C++ Refresher 277

H.1 Classes and Objects: Type and Instance 277
H.2 Inheritance: Interface Reuse 278
H.3 Polymorphism: Interface-Based Dispatch 279
H.4 Advanced Relationships 281

PREFACE

How to read this book

If you opened this book looking for code snippets to copy and paste, you are missing the point. This book is here to change how you see firmware architecture.

Chapter 2 is not an example—it is the lens. The orchard metaphor isn't mere decoration. It reveals what procedural thinking hides. Skim it for techniques, and you'll think you only learned C++ tricks. Let the metaphor do its work, and you'll begin to see firmware problems in a completely new way.

Take a quick test after reading Chapter 2, ask yourself, "Is a temperature sensor on I^2C or SPI?" If your instinct is "I^2C," the shift hasn't landed yet. The question itself should feel wrong—like being asked what language your thoughts are in. Communication is a relationship you inject, not an attribute of the device's identity.

You have been warned. Read Chapter 2 slowly.

What This Book Teaches

Procedural code feels "close to the metal"—until the third board variant. This book gives you a disciplined, object-oriented approach to embedded design without sacrificing performance or control.

Shaped during the development of IOsonata, an open-source C++ HAL, this approach delivers drivers that switch buses with a single line, avoid complex build scripts, and match the performance of bare-metal C. The orchard model names the pieces—pins, buses, devices, data flow—so you can see the relationships clearly. The metaphor captures the intent; the implementation stays literal C++.

Who This is For

- Builders ready to grow past blinking LEDs.
- Engineers who need patterns that scale across products.
- Teams seeking portable foundations without the bloat of complex BSPs.

C++ ground rules

We assume you know the basics (e.g., variables, pointers, loops). We focus on encapsulation, interfaces, polymorphism, static memory, and tight, predictable ISRs. We avoid the STL, exceptions, and heap allocation in hot paths.

What You Will Build

- Drivers that work on any bus—I²C, SPI, or wireless—without code changes.
- Architectures that handle many board variants with pin maps, not porting projects.
- Firmware that is unit-testable on your PC before you ever touch hardware.
- Systems that move to new silicon in days, not months.

What's Next

The goal is a new way of seeing: firmware as an ecosystem of stable relationships. Refactors shrink to one-line changes. Drivers become portable. Hardware specifics stay contained. So when the next board spin lands on your desk, you update pin maps—not your entire firmware—and the build is ready in minutes, days.

To Blinky and beyond—let's make your IO sing! 🎶

PART 1
THE LIVING ARCHITECTURE

The first seven chapters of this book are a guided walk through the digital orchard. We begin with foundational concepts of the living architecture and move toward practical implementation. By the end of this section, you will have built a complete, hardware-agnostic system and grasped the principles needed to write firmware that is fun, fast, and fearless in the face of complexity.

CHAPTER 1
MYTHBUSTING OF C++ IN EMBEDDED SYSTEMS

The Silent Crisis in the Lab

If you are reading this, you have likely felt the strain. Ten years ago, "embedded development" often meant writing a control loop for a washing machine or a thermostat. The constraints were tight, the features were static, and C was the undisputed king. You could hold the entire system state in your head.

Today, that same thermostat talks to the cloud, runs a Bluetooth stack, updates its display over SPI, manages power states for a battery, and handles over-the-air (OTA) updates—often on the same microcontroller. The hardware has evolved, but our firmware architecture has not. We are still trying to build skyscrapers using the same tools we used to build sheds.

The result is "**Super-Loop Spaghetti**": codebases that are fragile, tightly coupled to specific hardware, and terrified of change. When marketing asks for a new sensor, engineering groans because they know it means rewriting the `main.c` file yet again.

There is a better way. Modern software engineering has solved the problems of complexity and scale, primarily through Object-Oriented Design (OOD). Yet, in the embedded world, these tools are often viewed with suspicion. We are told they are too heavy, too slow, or "not close enough to the metal."

This chapter is here to clear the air. Before we can build a new architecture, we must first dismantle the fear that prevents us from using the right tools. We need to look at C++ not as a monster to be tamed, but as the only logical evolution for managing complexity.

1.0 MYTH 1: "OBJECT-ORIENTED PROGRAMMING IS BAD FOR EMBEDDED"

Ask many embedded C developers about object-oriented programming and you'll hear something like this:

> "We avoid OOP and stick to C. Objects and classes are bad for embedded."

It sounds straightforward. But if you look at how real C firmware evolves, you see something different. As soon as a project grows to include:

- More than one board or SKU
- Several sensors on different buses
- Drivers reused across products

The simple "everything wired by hand" style starts to strain. To keep duplication and code size in check, many C codebases drift toward the same patterns:

- A `struct` to hold state
- Function-pointer tables to describe operations, and
- Arrays of "devices" that can all be driven in the same loop

The code still says `C`, but the structure starts to look very familiar.

Example : Minimalist C on a single board

Suppose a board has four sensors on one I²C bus:

- BMI270 (IMU: accelerometer + gyro)
- ICM456x (another IMU)
- BMM350 (magnetometer)
- MS8607 (temperature + pressure + humidity)

A straightforward orchestration layer might look like this:

```
void Motion_InitAll(void)
{
    bmi270_init(&g_Imu1, &I2C1, addr1);
    icm456x_init(&g_Imu2, &I2C1, addr2);
    bmm350_init(&g_Mag,  &I2C1, addr3);
    ms8607_init(&g_Tph,  &I2C1, addr4);
}

void Motion_UpdateAll(void)
{
    bmi270_update(&g_Imu1);
    icm456x_update(&g_Imu2);
    bmm350_update(&g_Mag);
    ms8607_update(&g_Tph);
}

void Motion_SleepAll(void)
{
    bmi270_sleep(&g_Imu1);
    icm456x_sleep(&g_Imu2);
    bmm350_sleep(&g_Mag);
    ms8607_sleep(&g_Tph);
}
```

For a single fixed board, this works well: there is no indirection, the "All" functions are tiny, and the optimizer can inline most of it away.

When the product line evolves

Now imagine what usually comes next:

- A “Pro” variant adds a second IMU
- A cheaper variant drops the magnetometer
- A later board moves one IMU to SPI
- Another SKU swaps the environmental sensor

You still want simple high-level calls: initialize all sensors present on this board, update them in a loop, put them to sleep, and switch models without rewriting the motion pipeline. You *can* keep editing `Motion_InitAll` and friends with `#ifdef` blocks, but this quickly becomes repetitive and fragile.

The C Solution: Struct and Function Pointers

Most C codebases solve this by creating a generic sensor handle:

```
typedef struct {
    void *ctx;
    bool (*init)(void *ctx);
    bool (*enable)(void *ctx);
    void (*disable)(void *ctx);
    bool (*update)(void *ctx);
} SensorHandle_t;
```

Each concrete driver provides small adapter functions:

```
static bool bmi270_update(void *ctx) {
    return driver_bmi270_update((bmi270_t *)ctx);
}

static bool icm456x_update(void *ctx) {
    return driver_icm456x_update((icm456x_t *)ctx);
}
```

A board can now describe its sensors as data:

```
SensorHandle_t g_Sensors[] = {
    { &g_Imu1, bmi270_init, bmi270_enable, bmi270_disable,
bmi270_update },
    { &g_Imu2, icm456x_init, icm456x_enable, icm456x_disable,
icm456x_update },
```

```
    // …
};
```

And the high-level code becomes generic:

```
void Motion_UpdateAll(void)
{
    for (size_t i = 0; i < NUM_SENSORS; ++i)
    {
        g_Sensors[i].update(g_Sensors[i].ctx);
    }
}
```

From a C perspective, this is just careful use of `struct` and function pointers to avoid duplication. The list of sensors is centralized, the update loop is agnostic, and swapping a sensor is merely a data change.

Compare this to the C++ version

The same idea in C++ looks like this:

```
class Sensor {
public:
    virtual ~Sensor() = default;
    virtual bool Init(const void *cfg, DeviceIntrf *intrf) =
0;
    virtual bool Enable() = 0;
    virtual void Disable() = 0;
    virtual bool Update() = 0;
};
```

Concrete sensors inherit and override these functions. The orchestration array and loop become:

```
class Bmi270AccelGyro : public Sensor {
public:
    bool Init(const void *cfg, DeviceIntrf *intrf) override;
    bool Enable() override;
    void Disable() override;
    bool Update() override;
};

class Icm456xAccelGyro : public Sensor {
public:
    bool Init(const void *cfg, DeviceIntrf *intrf) override;
```

```
    bool Enable() override;
    void Disable() override;
    bool Update() override;
};

Sensor *g_Sensors[] = { &g_Imu1, &g_Imu2 /* … */ };

void Motion_UpdateAll() {
    for (auto s : g_Sensors) {
        s->Update();   // virtual call
    }
}
```

Under the hood, this becomes:

- a **vtable** (a function-pointer table), and
- a **vptr** (a per-object pointer to that table)

In other words, the same machinery the C version built manually.

What this implies for the myth

This comparison doesn't prove that *every* use of OOP is good for embedded, nor does it claim that all C codebases secretly want to be C++. But it does suggest something simpler:

Once you want reusable drivers, interchangeable sensors, and "do this to all devices" logic, the structures that naturally appear in a C codebase start to mirror what is usually called **object-oriented design**. The gap between "good C with function tables" and "careful C++ with virtual functions" is smaller—architecturally and mechanically—than the myth suggests.

1.1 MYTH 2: "C++ MAKES EMBEDDED SYSTEMS SLOW AND BLOATED."

Reality: Performance issues are caused by **architectural choices**, not by the `class` keyword. Overhead typically arises from dynamic allocation,

hidden copies, and interrupt service routine (ISR) bloat—not from encapsulation or carefully placed polymorphism.

Evidence A: The **`digitalWrite()`** fallacy**.** Many engineers' first "C++ performance" impression comes from Arduino's `digitalWrite()` API. On some boards (e.g., Arduino Due), a `digitalWrite()` call can take microseconds, while writing the port register directly takes tens of nanoseconds—roughly **100 × faster**. That gap reflects an ease-of-use wrapper, not an inherent C++ tax.

Evidence B — UART throughput benchmarks. With disciplined C++ (e.g., static buffers, lean ISR, stable seams), the performance narrative flips. In our UART PRBS tests, IOsonata's C++ drivers meet or exceed both Zephyr (C) and vendor nrfx (C). On some hardware, IOsonata sustained 2 MBaud where others did not. The 2 MBaud is achieved using custom baudrate calculation formula documented in Nordic DevZone (see IOsonata `uart_nrfx.cpp`). The nRF52832 UART hardware supports arbitrary baud rates beyond the predefined values listed in product specifications.

Board	IOsonata (C++ OOD)	Zephyr (C)	nrfx (C)
nRF54L15 DK	102.2 KB/s	82.9 KB/s	87.0 KB/s
nRF52832 DK	203 KB/s @ 2 MBaud	Not supported due to hardcoded config.	183.7 KB/s

Results may vary depending on the host PC.

1.2 MYTH 3: "C++ REQUIRES DYNAMIC MEMORY, WHICH IS UNSAFE."

Reality: C++ permits dynamic memory. It does not require it. Professional embedded designs should rely on **static allocation** for critical paths.

Evidence: Static by design. Heap fragmentation and leaks are legitimate concerns, but they're language-agnostic. Zephyr, written in C, exposes heap APIs like `k_malloc()` when flexibility is needed. By

contrast, IOsonata is designed to avoid dynamic memory entirely in data paths: drivers and communication buffers are statically allocated, keeping memory footprints fixed and timing predictable—while still leveraging C++'s encapsulation.

1.3 MYTH 4: "C++ ABSTRACTIONS FOR PORTABILITY CREATE ENDLESS CONFIGURATION."

Reality: Portability does not require heavy layering. You can achieve it with clean seams and capability-shaped interfaces.

Evidence: Seams vs. layers. Frameworks like Zephyr use `Devicetree` (`.dts`) and `Kconfig` for flexibility—powerful but complex. Our approach is different:

- Drivers target abstract interface
- Board specifics stay at the edge
- No external scripts, JSON files, or BSPs are required for core configuration.
- Adding a new board often requires a single pin-map header. Swapping $I^2C \leftrightarrow SPI$ is often a one-line change at initialization.

Portability comes from disciplined interfaces—not exhaustive configuration.

1.4 CONCLUSION: THE TOOL VS. THE BLUEPRINT

We have proven that C++ is not the enemy. With the right discipline—static allocation, virtual functions, and careful inheritance—it is lean, fast, and remarkably efficient. But a sharp saw does not build a house on its own.

Knowing that C++ is safe for embedded is different from knowing how to design with it. If you simply take C++ syntax and apply it within the

same old procedural mindset, you won't get better firmware; you'll just get "**C with Classes**"—a slightly fancier version of the same spaghetti.

To truly escape the trap of legacy firmware, we need more than a language change. We need a new **mental model**. We need to stop thinking of our code as a list of instructions to be executed and start thinking of it as an ecosystem to be tended.

We need to move from the factory floor to the garden.

Turn the page. It’s time to enter the **Orchard**.

CHAPTER 2
OBJECT MODELING — A LIVING ARCHITECTURE

This chapter has one job: to change how you see firmware.

What follows is not an analogy meant to help you remember object-oriented concepts. This chapter is a cognitive tool, meant to shift you from **digital** to **analog** thinking about firmware.

Read this chapter for perception, not patterns. The metaphor must be experienced in sequence—return after Chapter 6 if you want to analyze structure.

THE CORE DISTINCTION

Digital thinking asks: What are the states? What are the steps? What are the branches?

- "This sensor is on I^2C."
- "Configure the peripheral, enable it, then poll the data register."
- "If Bus A is busy, use Bus B."

Analog thinking asks: What are the relationships? What are the flows? What are the lifecycles?

- "This sensor speaks through an injectable communication relationship."
- "This device has seasons: it wakes, it works, it sleeps."
- "Data flows from soil, through roots, to fruit."

By the end of this chapter, firmware will look different—not because you learned new syntax, but because you learned to see what was always there.

THE ORCHARD AWAKENS

The orchard wakes before you do. The air is cool, still holding the memory of night. You step out, and the dew-kissed grass yields beneath your boots. The soil lies heavy and cool. Beneath your feet, roots stir, invisible, drinking deep in the dark. Trunks rise, steady and unhurried.

As the sun crests the horizon, branches stretch, and you can almost feel the sap rising. Soon, fruit will swell, first small, then heavy with potential. Each part of this living system has its own rhythm. The soil holds. The roots drink. The trees grow. The fruit ripens.

Good firmware should feel the same way: a living system, not a rigid machine.

This chapter is the first walk-through of the design orchard. We'll learn to ***see*** the land, ***feel*** the roots, and ***understand*** the life of the trees and their fruit. Before planting a single seed of logic, we must picture the rows, understand the soil, and envision the harvest. The growing will come naturally once we know the land.

2.0 THE FEELING OF GOOD ARCHITECTURE

Standing here, you notice: no one is shouting instructions. The orchard just... works.

This tranquility is what truly great firmware architecture feels like: solid ground, clear lifecycles, and predictable outcomes.

Bad design feels different. The ground shifts beneath you, and every time you reach for fruit, you must learn a new dance. Instead of enjoying the harvest, you spend your days fighting weeds and patching fences.

Most of us were taught to think in procedural steps: do this, then that. Like laying bricks in a straight row. It works, until one brick slips and the entire wall wobbles.

But an orchard does not live in steps. It lives in cycles and flows. This ecosystem is the mental model we will use for our architecture.

That shift—**from digital to analog thinking**—is what object-oriented design is really about.

Callout—Analog vs. Digital Thinking

- **Digital thinking:** "*Is this pin high or low?*" "*This feature on or off?*" It's about the immediate state.
- **Analog thinking:** "*What family does this belong to?*" "*What cycle does it follow?*" "*What remains the same if I swap one thing for another?*" It's about relationships and patterns.

A**nalog thinking** organizes the landscape so that digital details have a sane, predictable home.

Reflection: The Shift

Procedural thinking stacks bricks. Analog thinking grows orchards. One is rigid; the other is alive.

2.1 THE LAND: WHAT NOT TO OBJECTIFY

The land feels steady. It supports everything but is never the star. Scoop up a handful of soil; it doesn't change with every breeze, nor does it ask for applause. It just... is.

In firmware, some parts are like this too: simple, low-level resources such as GPIO pins, fixed memory buffers, or bit masks. These are T**he Land**—the foundation of the microcontroller.

We don't try to make the Land into something it is not. If it has no independent life, no family, no dynamic behavior, it must remain simple. We don't objectify the dirt. Doing so adds weight and confusion where speed and clarity are paramount. The Land is trusted, not modeled.

Reflection: The Ground Rule
Don't design life where there is none. Soil is soil. Keep it simple, lean, and fast.

2.2 THE ROOTS: THE INTERFACE

Field Guide: The Roots
- **Metaphor:** Roots drawing nutrients from the soil.
- **Engineering Concept:** An abstract interface (`DeviceIntrf`) that decouples high-level drivers from the physical bus (I^2C, SPI, UART).
- **Implementation:** See Chapter 5 for the `DeviceIntrf` class definition and examples.

The Roots never show off. They don't bear fruit, and they don't ask for praise. But beneath the surface, they are alive—drinking deep, binding tree to land, carrying lifeblood through hidden channels.

In design, the Roots are your interfaces—the hidden connections where everything meets. They don't hand you fruit directly, but they make fruit possible.

If roots rot, the tree falls. If interfaces tangle, the system chokes. The vital parts of the system are often invisible, but they are the most alive.

Reflection: The Invisible Work
Roots connect quietly. Interfaces in design should do the same—essential, but never the star.

2.3 THE TREES: THE DEVICE OBJECT

Key Concept: The Device Lifecycle
- **Metaphor**: A "Tree" has seasons (e.g, sprout, grow, rest, bear fruit).
- **Technical Principle**: A "Device Object" has a formal lifecycle. It must be initialized, enabled before use, and can be disabled to save power or reset to recover from an error. Good design respects this lifecycle.

Look up. The trees stand tall, patient, and alive. They didn't appear fully grown. Each began as a seed—fragile and small. Then came roots, then trunk, then branches. Storms bent them but did not break them. Winter put them to rest. Spring brings them back to life. Year after year, the cycle repeats: sprout, grow, rest, bear fruit. That is the **rhythm of life**.

In firmware, devices are like trees. They don't simply "turn on." They have lifecycles. They must be prepared, brought to life, used, put to sleep, and awakened.

Think of it this way:

- A sensor doesn't live forever "on." It needs to be woken, measured, and rested.
- A display isn't always drawing. It turns on, displays, then goes quiet again.

Treating devices like trees changes how you design. Instead of flipping switches, you start to see lifecycles. And when you respect lifecycles, things stop breaking. Your firmware feels less like forcing electricity through wires and more like tending something alive.

> **Reflection: The Life Test**
> A true object has a life to live. If it breathes in cycles, it deserves to be designed as alive.

2.4 WHAT THE WORLD SEES: THE ENCAPSULATION

The purpose of the **Fruit** is the **Juice**. The Juice is sweet, tangible, and nourishing. When someone asks for an apple, you don't hand them soil, roots, and bark. You hand them the Fruit.

That's how design should be. What the world sees is the outcome, not the machinery inside.

- The **Juice** of a temperature sensor is the temperature in °C, ready to use—not the raw voltages and calibration registers.
- The **Juice** of a display is pixels drawn at given coordinates, not frame-buffer gymnastics.

Encapsulation hides the dirt and tangled roots and simply offers you its **Juice**. The beauty is that the inside is still there—rich, messy, alive—but it's not what you're asked to wrestle with. You get the part that matters most.

> **Reflection: The Orchard's Gift**
> Encapsulation is generosity. The **Fruit** hides the mud so you can enjoy the **Juice**.

2.5 DIFFERENT FRUIT, SAME ROOTS: THE INHERITANCE

Walk a little farther, and you see order in the orchard. While the Fruits differ, the underlying patterns remain the same. This view is **inheritance**.

- **Variety**: You see rows of apple trees, orange trees, and kiwi trees. Each produces a different **Fruit**, with its own color, scent, and **Juice**.
- **Commonality**: And yet, they are all still *trees*. They share a fundamental identity: roots in the soil, a trunk that rises, and branches that stretch for the sun.
- **The Principle**: Each type of tree (class) inherits the deeper essence of "tree-ness" but expresses it in its own unique way. In our code, this means a `TempSensor` and a `PressureSensor` might both inherit from a common `Sensor` class, sharing its core behaviors while producing different kind of "**Juice**."

In design, **inheritance** works the same way.

- You start with a shared pattern (the “tree”).
- You grow variations (apples, oranges, kiwi).
- Each child shares a structure but produces its own **Juice**.

It’s family resemblance, in code and in life. And just like in orchards, inheritance brings both harmony and variety. You know what to expect —a trunk, branches, leaves—but the outcome is always unique.

Reflection: The Family Rule
Inheritance is about identity: what things *are*. Different fruits, same “treeness.”

2.6 ONE TOOL, MANY FORMS: THE POLYMORPHISM

Now comes the harvest. You carry your basket into the kitchen. On the counter sits a single machine: **The Processor**. It has just one button: **PROCESS.**

- You drop in an apple, and the machine slices the apple.
- You drop in an orange, and the machine peels the orange.
- You drop in a kiwi, and the machine scoops the kiwi.

Same machine. Same button. Different results.

That is **polymorphism**: one common action, many specialized forms of that action, all without changing the caller.

As Bjarne Stroustrup, the creator of C++, put it:

> "The key concept in OOP is not the class, it is the object and the way objects interact through interfaces—polymorphism is the essence of object-oriented programming." [2]

But after a while, the fruit processor starts to grumble. "Really? Apples, oranges, kiwis? Make up your mind already—I'm getting dizzy over here!"

And yet—it still does it. Same button. Different outcomes. That's the magic.

Eventually, you sigh. "Fine. I'll build an apple processor just for apples. An orange processor just for oranges. A kiwi processor just for kiwis."

That's inheritance again—dedicated children, each locked to one kind of fruit. Predictable, but now your kitchen is crowded with machines.

Inheritance and polymorphism are easy to confuse because they often go hand in hand. But their roles are different. Inheritance is about **identity**. Polymorphism is about **behavior**. One builds families. The other enables transformation. The art is knowing when to put up with the processor's complaints and when to build a new one.

> **Reflection: The Subtle Distinction**
> Inheritance is about *what things are*. Polymorphism is about *what things do*. One builds families. The other enables transformation.

2.7 THE PEAR ARRIVES: THE SCALABILITY

One spring, you plant something different at the edge of the orchard: a row of pears.

The apples glance over and whisper, "Who's that fuzzy newcomer?" The oranges gossip, "Strange fellow, but seems harmless enough." The kiwi roll their eyes. "Here we go again."

And yet, the orchard holds. The soil doesn't crumble. The Roots make room. The trees grow as before, and the fruit keeps coming.

Later, at harvest, the basket accepts everything. It simply holds apples, oranges, kiwis, and now pears, all side by side. The Processor groans a little when you toss in a pear, but it still does the job.

That is scalability. A good design welcomes the new without breaking the old. Adding a **Fruit** should feel like adding a row to the orchard, not tearing down the whole farm.

> **Reflection: The Hospitality of Design**
> Scalability isn't predicting the future—it's preparing to welcome it gracefully.

2.8 THE ORCHARD KEEPER: DECIDING WHAT BECOMES AN OBJECT

Every orchard has a keeper. He walks the rows at dawn, inspecting. He knows the difference between soil, roots, and trees, and he never confuses them.

- He doesn't prune the dirt.
- He doesn't plant roots in neat rows.
- He only tends to what truly grows.

That's the keeper's wisdom: not everything deserves the same care.

In design, the same rule applies. Not every part of your system needs to be an "object." Before you create one, pause and ask:

- **Does it have a life of its own?** (A predictable cycle: init, enable, disable, reset?)
- **Does it belong to a family?** (Could there be different versions, all responding to the same command?)

If yes, then it belongs in your orchard as a true object, ready to grow, adapt, and bear fruit.

If not, leave it as Land, or keep it as a simple structure. Let it stay lean and fast. The orchard keeper doesn't waste time trimming dirt, and you don't need to over-design what isn't alive.

> **Reflection: The Balance of Design**
> Wisdom is knowing what to prune, what to water, and what to leave alone. Not everything deserves to be an object.

2.9 WALKING THE ORCHARD: THE MENTAL MODEL

You've walked the orchard with me: felt the soil, traced the roots, stood under the trees, tasted the fruit, carried the basket, argued with the Processor, and even welcomed the pear.

Now it's your turn. Close your eyes and picture it:

- The **Land** beneath your feet, steady and simple.
- The **Roots** spreading unseen, carrying lifeblood.
- The **Trees**, each with its own cycle of life.
- The **Fruit**, from which you get the **Juice**.
- The **Family Rows**, repeating patterns with their own flavors.
- The **Blender**, one action, many results.
- The **Pear**, welcomed without breaking the orchard.
- The **Keeper**, tending only what grows.

If you can see it, you can design with it. The details of code will come later—the syntax, the classes, the headers. For now, envision firmware not as brittle wires and switches, but as a living orchard, with rhythms and relationships.

Carry this picture with you. Because when the late-night puzzles come, when you're buried in defines and configs, it won't be rules or checklists that save you. It will be this: the orchard in your mind, reminding you to design as if things are alive.

That's A**nalog thinking**. And once you see it, you can't unsee it.

> **Reflection: The Mental Shift**
> Designing with objects is not about code—it's about sight. If you remember the orchard, you'll remember the way.

2.10 DIFFERENCES THAT MATTER (AND THOSE THAT DON'T)

Not all differences are equal. Some change the **Fruit** itself, while others just change the flavor of its **Juice**.

The farmer would ask: *"Is this a new species, or just a different flavor?"*

- If the difference changes how you handle the object, it may deserve its own **Branch** (a new class definition).
- If it just changes the output data or a setting, it's a "flavor" (configuration data for an existing **Fruit** type).

Pick up a pear and an apple: different shapes, different handling. These are different **Fruits** from different **Branches**.

Pick up two apples—one green, one red: same handling, different flavor. These are the same kind of **Fruit**, but they produce different **Juice**.

Examples you'll see later:

- Temperature Sensor vs. Display → different **Branches** (they have different identities and lifecycles).
- Temperature Sensor Reading at 10 Hz vs. Reading at 50 Hz → flavor of **Juice** (same **Fruit** type, different setting).
- Metric vs. Imperial, Red vs. Green, Low vs. High brightness → "flavors," not species.

Reflection: The Species Test
Don't multiply species when a flavor will do.

2.11 THREE TINY QUESTIONS (YOUR ON-THE-SPOT CHECKLIST)

When you feel the itch to "make a class," pause and ask yourself three tiny questions:

- **What's the family trait?**
  Name what stays the same for all siblings (e.g., lifecycle, key verbs, guarantees).

- **What's the honest difference?**
  Only give children the differences that truly change how they're handled.

- **Can I pass a basket around?**
  If callers can treat every **Fruit** from every **Branch** as the same and still win, you've earned polymorphism.

Tape these above your desk. They'll prevent 90% of accidental bloat.

Reflection: The Quick Pause
Three questions, five seconds, countless hours saved.

2.12 ANTI-PATTERNS TO AVOID (GENTLY BUT FIRMLY)

Every orchard has weeds—things that choke growth if left untended. In object design, these are the weeds to pull early:

- **The God Objects**: One giant class that "does everything" and explains nothing. This class is a sign of poor cohesion and a violation of the single-responsibility principle. A well-designed system is composed of smaller, focused objects that each have a single, clear purpose. Split monolithic objects by capability.

- **Invasive Vines (Code Generation Macros):** This anti-pattern is particularly dangerous. It creates a "mini-language" inside a macro (e.g., `X-Macros` or custom `DEFINE_DEVICE(...)` macros)

which the preprocessor expands into structs, enums, or entire function bodies. While it reduces boilerplate, it creates "magic" that hides logic, breaks modern IDE tooling, and creates brittle error messages. Most importantly, it is a **crude workaround** that bypasses the safety of the C++ type system. Our orchard is grown with visible, type-safe C++ constructs, not hidden macros.

- **Brittle Family Trees**: Inheritance depth is not a problem if each level represents a clear, logical specialization. A `TphBme680` class inheriting from `TempSensor`, `PressSensor`, and `HumiSensor` —which all derive from `Sensor` and `Device`—is a powerful and correct design. Each step is a clear "is-a" relationship. The anti-pattern is creating a new "species" for what should be a "flavor." For instance, do not create a `TempInKSensor` class that derives from `TempSensor` just to get the temperature in Kelvin instead of Celsius. A sensor's unit of measurement is a data conversion, not a fundamental change in its identity. That logic belongs in a conversion method (`ReadTemperatureInK()`) within the `TempSensor` class, not in a new, redundant branch on the family tree.

- **Objectifying Dirt**: Turning simple, stateless helpers for low-level resources like I/O pins into full-blown classes adds unnecessary overhead and complexity. Dirt is dirt. Keep it lean and represented by simple functions or structures. If it doesn't have a lifecycle or complex state, it's not a living "tree."

- **Flavor Explosion**: Creating a new class or "type" for every minor variation in configuration is a common mistake. If the only difference is a setting, such as a sensor's sampling rate or a display's color, that variability belongs in the configuration data passed during `Init()`, not in new species.

Reflection: The Pruning Rule
A healthy orchard is pruned, not overgrown.

Closing Thought

The orchard is both a metaphor and a map. Land, roots, trees, fruit, families, blenders, baskets, kiwis, keepers—all of it helps us rewire how we think. But reflections and guardrails matter just as much. They prevent a lush orchard from turning into a tangled forest.

Design as if things are alive but remember: not everything deserves to grow.

Checkpoint: Did the Shift Happen?

Before moving to Chapter 3, test whether the perceptual shift has happened. Answer the following four questions honestly. Remember, your intuition matters more than the "correct" answer.

Question 1: Board Thinking

You need to support a new board with the same nRF52832 chip but different pin assignments. What answer feels most natural?

- **Digital answer:** "I need to create a new board variant, fork the BSP, update the device tree."
- **Analog answer:** "I need to change the pin map data. The board is a variation, not a new species."

Question 2: Driver Thinking

Your BME280 temperature sensor currently uses I²C. Marketing wants a variant that uses SPI to save PCB space. What approach feels most natural?

- **Digital answer:** "I need to write a new BME280_SPI driver or add conditional code to handle both buses."
- **Analog answer:** "I configure it to speak over SPI instead of I²C. The sensor's identity is unchanged; only its communication path differs."

Question 3: Pin Change Thinking

The hardware team moved UART RX from pin 8 to pin 11 due to a routing conflict. What answer feels most natural?

- **Digital answer:** "I need to rebuild the firmware, possibly regenerate code from a configuration tool."
- **Analog answer:** "I need to change two numbers in the pin map. This is data, not code."

Question 4: Bus Swap Thinking

Your pressure sensor needs to switch from I²C to SPI because the I²C bus is getting too noisy. What answer feels most natural?

- **Digital answer:** "I need to rewrite the driver's communication layer or port it to a different HAL."
- **Analog answer:** "The driver talks to a bus interface. I inject a different root (bus); the driver doesn't change."

Scoring Guide

4 analog answers: The shift has happened. Proceed with confidence.

3 analog answers: You're close. Reread the sections in this chapter on boards-as-data and on roots/soil (bus interfaces).

2 or fewer analog answers: The shift hasn't fully landed yet. This is not a failure—perceptual transformations take time. Return to the beginning of this chapter tomorrow and read it again, not to extract information, but to let the metaphor reshape how you see the relationships between components.

Why this matters: The code in the following chapters will feel awkward and over-abstracted if you're still thinking digitally. Once the analog shift happens, the same code will feel inevitable and elegant.

The Orchard is waiting. Take your time.

References for further reading

[1] Grady Booch, Object-Oriented Analysis and Design with Applications

[2] Bjarne Stroustrup, The C++ Programming Language

[3] Gamma, Helm, Johnson, Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software

CHAPTER 3
DEVELOPMENT ENVIRONMENT

An engineer's vision is the seed of great firmware, but that vision thrives best in the well-prepared soil of a professional development environment. This setting means moving beyond a simple text editor to a full workbench where compilers, debuggers, and hardware work together reliably.

This chapter builds that soil. We will assemble a professional, vendor-neutral firmware workbench, walk through the entire installation process, and compile a real project. The result is an environment built for stability, collaboration, and the reliable implementation of great ideas as firmware.

3.0 THE DEVELOPMENT PROCESS

Developing for embedded systems is fundamentally different from PC software development. The process is dictated by the nature of the hardware target, which falls into two main categories: microprocessors and microcontrollers.

Microprocessors (MPUs) vs. Microcontrollers (MCUs)

Understanding the difference between an MPU and an MCU is the first critical step.

- **Microprocessor (MPU):** A MPU is the CPU in a device like a Raspberry Pi. It requires external RAM, storage, and peripherals to function. MPUs are powerful enough to run a full operating system like Linux, where you can compile and run code directly on the device.
- **Microcontroller (MCU):** A MCU is a self-contained "computer on a chip" with an integrated CPU, RAM, flash memory, and peripherals. MCUs are designed to run a single, dedicated program and have limited resources, making it impossible to compile code on the chip itself.

3.1 THE FIRMWARE DEVELOPMENT CYCLE

The crucial takeaway is that for MCUs, all development work—writing, compiling, and debugging—is **cross-compiled** on a host PC. The final binary is the only thing transferred to the target hardware.

This separation between the development machine and the target hardware defines the standard, iterative five-step firmware development cycle:

1. **Edit:** Write C/C++ source code on your PC.
2. **Build:** Use a cross-compiler to translate the source code into machine code that the target MCU can understand.
3. **Flash:** Write the compiled program into the MCU's non-volatile flash memory using a debug probe connected via an interface like SWD or JTAG.
4. **Debug:** Use the same probe to control the MCU, step through code line-by-line, and inspect the state of registers and memory.
5. **Repeat:** Refine the code based on testing and repeat the cycle until the firmware is stable and complete.

3.2 REQUIREMENTS FOR A PROFESSIONAL WORKBENCH

Before choosing a specific tool, it's crucial to define what a professional embedded development environment must provide. A simple editor is not enough. Firmware development requires tight collaboration among software, the compiler, and the physical hardware.

A professional workbench is the bridge between these domains and must provide the following key capabilities:

- **Robust Project Management for Code Reuse**
  Effective code reuse requires a project management system that can compile files located anywhere on the filesystem, not just within the project's own folder. The environment must support **linking to shared source code from arbitrary and external locations** and manage those links with portable, relative paths. This relationship allows multiple projects to share a single copy of a library or SDK without duplicating code, which is essential for collaborative, large-scale firmware development.

- **Integrated Build System**
  The environment must seamlessly integrate with cross-compilers. It should provide one-click build and clean operations, automatically parse compiler errors, and map them back to the source code. This process should be managed by the IDE, not by fragile, hand-written scripts that need constant maintenance.

- **Hardware-Aware Debugging**
  Debugging embedded systems goes beyond inspecting variables. The environment needs deep, native integration with debug probes (like J-Link or CMSIS-DAP via OpenOCD). Crucially, it must provide **hardware-level views**, such as live peripheral registers, memory maps, and CPU core status. This allows a developer to see if the hardware is configured correctly, a task impossible with a standard software debugger.

- **Integrated Toolchain Management**
  A developer may use different versions of the ARM GCC and RISC-V GCC toolchains for various projects. The workbench

should automatically discover and manage the paths to these tools. Relying on system-wide PATH variables or manually editing JSON files for every project is inefficient and error-prone.

- **Vendor and Platform Neutrality**
  The ideal workbench is not tied to a single microcontroller vendor (like ST or NXP) or a single operating system. Projects should be shareable and build consistently on Windows, macOS, and Linux to support team collaboration and developer choice.

Any tool that fails to meet these fundamental requirements will inevitably create friction, waste time, and add unnecessary complexity to the development process.

3.3 CHOOSING THE RIGHT WORKBENCH: AN EVALUATION

Now that we have a clear set of requirements, we can evaluate the common tools used for firmware development. The goal is to find the environment that best fulfills all our needs out of the box.

Text Editors (VSCode, Sublime Text)

These are excellent code editors, but they are not integrated development environments. They require separate installation and management of toolchains (compilers, debuggers). While ecosystems like PlatformIO or vendor-specific extensions like nRF Connect for VS Code provide a more structured experience, they can create their own form of vendor or framework lock-in. Critically, even with these extensions, their custom project management systems often lack the robust, native, GUI-driven ability to link shared source code from arbitrary external locations using portable relative paths—a core requirement (see Section 3.2) for the reusable architecture presented in this book. Fragmentation across different VSCode ecosystems (e.g., PlatformIO vs. nRF Connect for the same nRF52 hardware) can also be challenging. With a text editor-based setup, you are often responsible

for integrating the toolchain components yourself or relying on specific, sometimes incompatible, extension frameworks.

Considerations for VSCode: While a highly capable editor, achieving the seamless, cross-platform linking of external source code required by our architecture involves manual configuration or potentially fragile scripting.

Vendor IDEs (STM32CubeIDE, MCUXpresso, Microchip Studio, Renesas e² studio, TI Code Composer Studio)

Silicon vendors provide these IDEs and come as a single installation that includes the full toolchain. Many, like STM32CubeIDE, NXP's MCUXpresso, Renesas e² studio, and TI's Code Composer Studio, are based on Eclipse and are cross-platform. Others, like Microchip Studio, are proprietary environments that are often Windows-only. While typically free, their purpose is to create vendor lock-in, making it difficult to switch to another vendor's hardware.

Modern C++ IDEs (CLion)

Developed by JetBrains, CLion is a powerful, cross-platform C++ IDE with a modern user interface and excellent code analysis capabilities. Its project model is built entirely around CMake, offering a script-centric build approach. However, like VSCode ecosystems, it lacks an integrated, GUI-driven feature for easily linking external source code directories to portable paths; this must be managed manually by editing CMakeLists.txt files. Furthermore, CLion typically lacks built-in, native views for live peripheral registers, which are essential for hardware-aware debugging (see Section 3.2), though third-party plugins may offer some capabilities. Its commercial subscription cost also places it in a different category from the free, open-source workbench recommended here.

Considerations for CLion: A superb C++ IDE, but its script-first workflow for managing dependencies and lack of native peripheral register views differ from the integrated, hardware-aware approach.

Proprietary 3rd-Party IDEs (IAR Embedded Workbench, Keil μVision)

This category includes independent, commercial toolchains that also **bundle the compiler and IDE in a single installer**. They are powerful tools with highly optimized compilers. Their primary drawbacks are high **cost**, restrictive licenses, and limited **platform neutrality**, as they are **often Windows-only**.

Eclipse Embedded CDT: The Professional's Choice

When evaluated against our list, Eclipse Embedded CDT is the tool that meets all requirements without compromise. It requires **separate toolchain installation**, which provides the **flexibility** to use any version of GCC you need for any project.

- It has a **robust project management system** with an intuitive, GUI-driven feature for linking and sharing code.
- It offers a fully integrated managed build system for ARM and RISC-V GCC.
- Its debugger provides the essential **native hardware-aware views** for peripherals and registers.
- It is vendor-neutral, free, and truly cross-platform.

Verdict: For these reasons, Eclipse Embedded CDT is our recommended workbench. It provides the solid foundation needed for professional, collaborative firmware development without the compromises of other tools.

Eclipse myths (quick view)

- **Myth: "Eclipse projects don't travel well."**
  Reality: When set up with relative paths and linked resources, Eclipse projects are highly portable. Clone the repo, import the project, and it just works—whether you're on Windows, macOS,

or Linux. Shared `.project` and `.cproject` files make teamwork natural.

- **Myth: "My team uses CMake, so we can't use Eclipse."**
  Reality: Eclipse Embedded CDT has excellent, first-class support for CMake-based projects. You can directly import a project containing a `CMakeLists.txt` file, and the IDE will automatically configure the build, provide full code completion, and integrate with the debugger. This makes it a powerful and flexible workbench even for teams already standardized on CMake, offering a robust graphical front-end for a command-line-friendly build system.

- **Myth: "Eclipse can't build from the CLI."**
  Reality: Eclipse CDT builds headless just fine. Use the `eclipsec` (Windows) or eclipse `-nosplash` (Linux/macOS) launcher to import and build without ever touching the GUI

  Prefer Makefiles? Export them once and call *`make`* directly. The build process yields the same result whether initiated from the GUI or the command line.

Eclipse Embedded CDT is not perfect, but it is professional. It's the workbench that doesn't get in your way—stocked with what you need and nothing more.

Note — About Build-System Layers
Many popular build frameworks, such as CMake, ultimately generate a Makefile or Ninja file before invoking the compiler. Each extra translation step adds setup time and potential confusion without changing the final binary. For embedded work, Eclipse Embedded CDT's Managed Build already performs this generation internally, producing the same reproducible results with fewer moving parts.
Less indirection means faster iteration, clearer diagnostics, and fewer surprises.
Prefer Managed Build for simplicity—or use External builder if your team standardizes on Make/CMake. Either way, the IOsonata architecture stays tool-agnostic.

3.4 WORKBENCH INSTALLATION

Setting up a professional environment involves installing the necessary compilers, debug tools, and an IDE, and then obtaining the project source code. There are two primary ways to accomplish this: an automated script for convenience or a manual step-by-step process for greater control and understanding.

Both paths lead to the same result: a fully functional workbench ready for development. By the end of this chapter, you'll have a complete workbench—compiler, debugger, and the core IOsonata library—ready to build professional-grade firmware.

The **manual path** lets you install each component individually, providing fine-grained control and a clear understanding of the complete setup. The detailed, step-by-step instructions for manual installation are in **Appendix D**.

For the quickest setup, IOsonata provides an installer script for each platform (macOS, Linux, Windows) that automates the entire process. Running this single command will:

- Download and install the latest **Arm** and **RISC-V GCC** toolchains**,** the translators for your microcontroller.
- Fetch and configure **OpenOCD** the open-source debugger glue.
- Install **Eclipse Embedded CDT**, pre-stocked with the right plugins.
- **Clone IOsonata itself** and pull in all the required vendor SDKs under (**…**/`IOsonata` and **…**/`external`).

A Note on Debugger Support: OpenOCD vs. pyOCD
The automated installer includes **OpenOCD**, a versatile, open-source debugger that works with a wide range of probes, including CMSIS-DAP. It does **not** install **pyOCD**, another popular open-source tool that also supports CMSIS-DAP probes. While both tools are excellent, support for specific microcontrollers can vary between them; one might support a target device that the other does not. Therefore, if you find your particular MCU is not supported by the included version of OpenOCD, install pyOCD. The manual installation guide in **Appendix D** provides the simple `pip` command to do so.

Run the script that matches your operating system:

macOS

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/
IOsonata/IOsonata/master/Installer/install_iocdevtools_macos.sh)"
```

Linux

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/
IOsonata/IOsonata/master/Installer/install_iocdevtools_linux.sh)"
```

Windows (PowerShell)

Important: You must run **PowerShell** as an Administrator. Right-click the PowerShell icon in your Start Menu and select "**Run as administrator**".

```
powershell -ExecutionPolicy Bypass -Command "iex ((New-Object
System.Net.WebClient).DownloadString('https://
raw.githubusercontent.com/IOsonata/IOsonata/master/Installer/
install_iocdevtools_win.ps1'))"
```

When the script finishes, your workspace is prepared without touching a single config file.

Understanding Your Project Layout

When you're done, your workspace will have a clean, layered structure. This layout keeps your code neatly separated from the third-party libraries.

```
/IOcomposer    - Development root directory
 |-- external  - Contains downloaded SDKs from silicon vendors
 |   |-- nRF5_SDK        - Latest Nordic SDK
 |   |-- nRF5_SDK_Mesh   - Latest Nordic SDK for Mesh
 |   |-- BSEC      - Bosch Sensortec Environmental Cluster Software
 |   |-- Others as needed
 |-- IOsonata  - IOsonata library
 |   |-- include   - Generic include common to all platforms
 |   |-- src       - Generic source common to all platforms
 |   |-- ARM       - ARM series based MCU port
 |   |-- RISV      - RISC series based MCU port
 |   |-- Others as needed
```

With the project code and its dependencies now organized, the core setup is complete.

3.5 VALIDATING YOUR WORKBENCH: THE FIRST SPARK

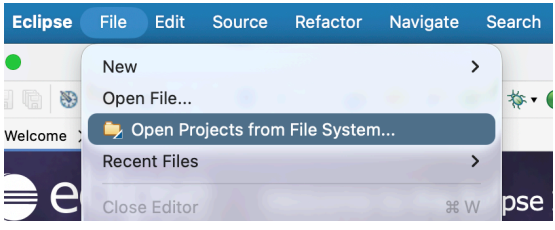
Whether you used the one-command installer or the manual DIY path, this final step is crucial. It verifies that your entire toolchain is working correctly by compiling the IOsonata library.

> **Note on Hardware**
> Throughout this book, we will use the Nordic nRF52832 microcontroller as our primary reference for all examples. However, IOsonata is designed to be portable and supports a variety of MCUs. If you are using a different supported chip, simply substitute the paths and hardware-specific details with those appropriate for your own board. The architectural principles remain the same.

Let's walk through the process in Eclipse:

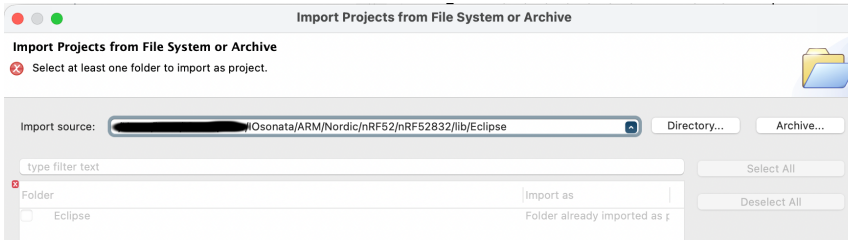
1. If the Welcome page is visible, hide it by clicking the "Hide" button in the top right corner to reveal the C/C++ perspective.

2. Go to **File → Open Projects from File System....**



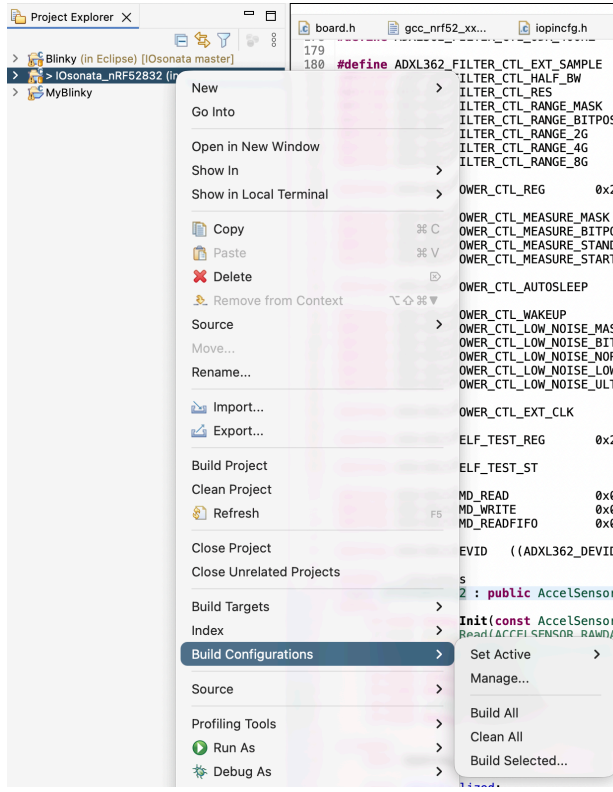
3. In the "Import Projects..." dialog, click the **Directory...** button. Navigate to the library project for your target hardware. For our reference nRF52832, select the folder:

```
IOsonata/ARM/Nordic/nRF52/nRF52832/lib/Eclipse
```



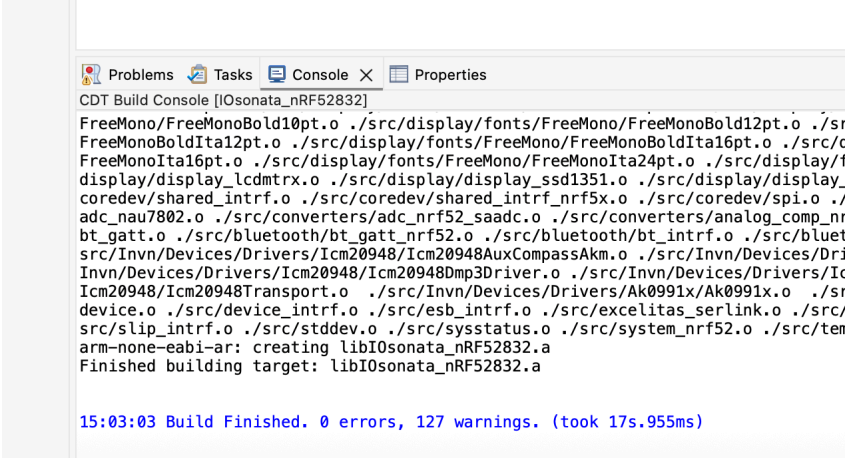
4. Click **Finish** to import the project.

5. From the **Project Explorer** view, right-click on the project (e.g., "IOsonata_nRF52832") and select **Build Configuration → Build All.**



6. The build progress and results will appear in the **Console** view at the bottom of the screen. If the console shows a successful build without errors, it means every part of your toolchain is working

in harmony. Your professional workbench is now fully configured and ready for development.



3.6 BUILDING ECLIPSE PROJECTS WITHOUT THE GUI

Sometimes you need to build a project from a script or command line. Eclipse CDT supports this through two headless paths:

- **Headless Eclipse Build**
  Use the Eclipse launcher to run a build directly from your terminal. This is perfect for CI/CD servers. Note that the path points to the library project we just imported.

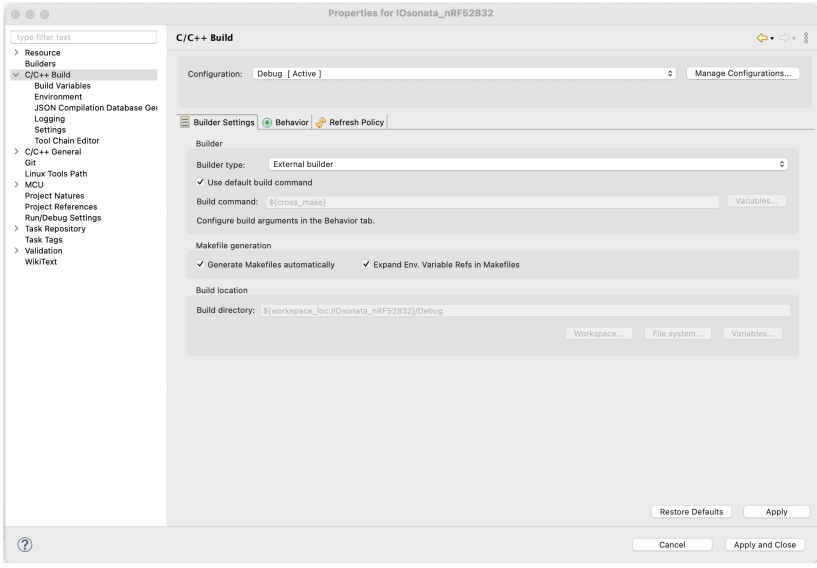
  ```
  eclipse -nosplash \
  -application org.eclipse.cdt.managedbuilder.core.headlessbuild \
  -data /path/to/workspace \
  -import /abs/path/to/IOsonata/ARM/Nordic/nRF52/nRF52832/lib/Eclipse \
  -cleanBuild IOsonata_nRF52832/Debug
  ```

- **Export Makefiles**
  You can have Eclipse generate standard Makefiles, allowing you to use the `make` command directly.

  1. Right-click your project in the Project Explorer and select **Properties**.

2. Go to **C/C++ Build** and ensure the two options below are selected for all build configurations

   - Generate Makefiles automatically
   - Use default build command



3. Build the project once inside Eclipse to generate the Makefiles in your build configuration directory (e.g., `/Debug`).

4. In a terminal, navigate to the Eclipse project directory (e.g., `…/lib/Eclipse`):

   ```
   make all
   ```

5. or, for a specific config (e.g., `Debug` or `Release`):

   ```
   make -C Debug
   ```

Whether you choose the headless launcher or plain `make`, Eclipse projects are ready for both interactive development and automated builds in the field.

Recap: Your Professional Workbench is Ready

In this chapter, you've accomplished the essential groundwork of a professional firmware developer. You've moved beyond a simple editor, choosing a proper workbench and stocking it with the core tools of the trade: compilers, debuggers, and the IOsonata source code. Most importantly, you validated the entire setup by successfully compiling the core library, proving that your new environment is ready for the work ahead.

Your professional workbench is operational—no placeholders, no guesses—just build, flash, debug.

CHAPTER 4
WIELD THE TOOL TO SEED THE LAND

You remember the moment you stepped back from your bench: everything installed, everything in its place. The IDE opened clean. The toolchain compiled without a fuss. The debug probe blinked to life the moment you connected it.

A workbench like that feels powerful—full of potential. But potential doesn't grow an orchard. It's quiet now. No fans. No LEDs. No signal. Just the whisper of code waiting to come alive.

This chapter is the first real movement. The beginning of a pattern you will repeat thousands of times: build, flash, debug, run. It's not glamorous, but it's sacred.

We will start with the humblest example in embedded history: Blinky. It does almost nothing—and that's exactly the point. It gets out of the way.

If the LED blinks, everything is working. Let there be light in the orchard.

A Note on the Hardware Used in This Book

For consistency, all hands-on examples from this chapter to the end of the book are based on a single microcontroller: the Nordic nRF52832 (ARM Cortex-M4F). We will use the I-SYST BLYST-NANO-DK as the primary development board for examples, as well as the official Nordic Development Kit (DK).

While we focus on this specific hardware, the IOsonata framework and the design principles you'll learn are highly portable. If you are using a different board or microcontroller, the overall process of creating and configuring projects will be the same. The only things that will change are the specific values you enter during the build configuration, such as the target processor details, pin definitions in your `board.h` *file, and the linker script.*
The goal is to provide a consistent, reliable learning path that you can adapt to any hardware you choose for your own projects.

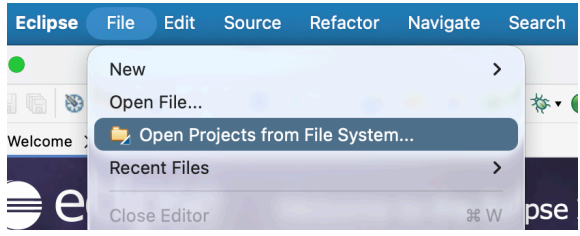
4.0 OPEN BLINKY PROJECT: THE FIRST SPARK

In this section, you will walk through the entire development cycle using a pre-built Blinky example. The goal isn't to write code, but to learn the mechanics of your new workbench.

Prerequisite: Core Library Build
Before you begin, please ensure you've completed the steps in Section 3.5: Validating Your Workbench. That process compiles the core IOsonata library (creating a `.a` *file). The Blinky example project depends on this library and it will fail to build if the library is missing.*

Step 1—Open the Blinky Project

With your IDE open, the first step is to bring the Blinky example into your workspace. You navigate through the menus: **File → Open Projects from File System...**.



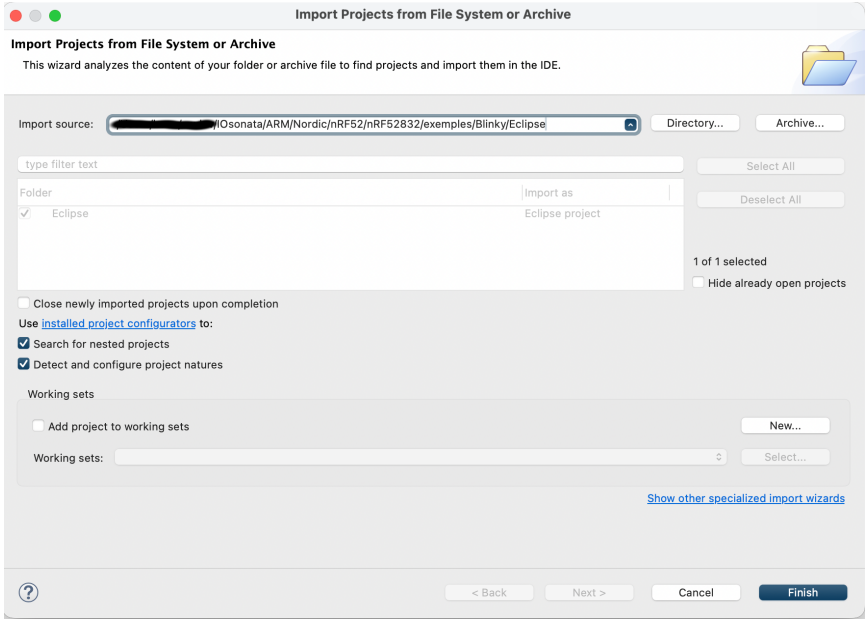
A dialog box appears, asking for a location.

Side Note: Path Conventions

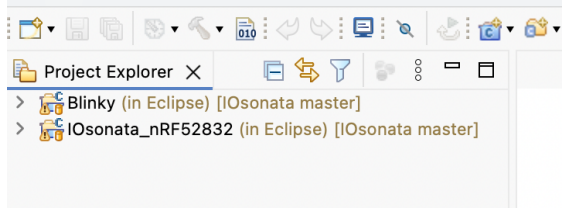
The IOsonata repo uses the folder name exemples (French spelling). Type it exactly as shown. In Eclipse dialogs, you can use forward slashes on any OS; Eclipse normalizes them. Case matters on Linux.

Click the **Directory...** and point it to the Blinky example folder:

`…/IOsonata/ARM/Nordic/nRF52/nRF52832/exemples/Blinky/Eclipse`

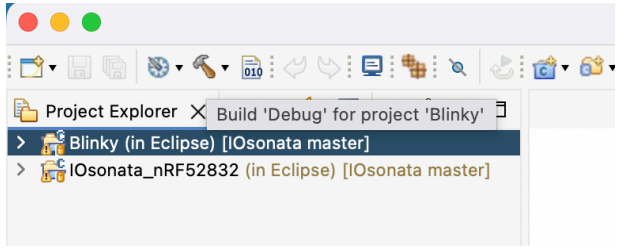


After you click "**Finish,**" the project appears in the Project Explorer on the left, ready to use.



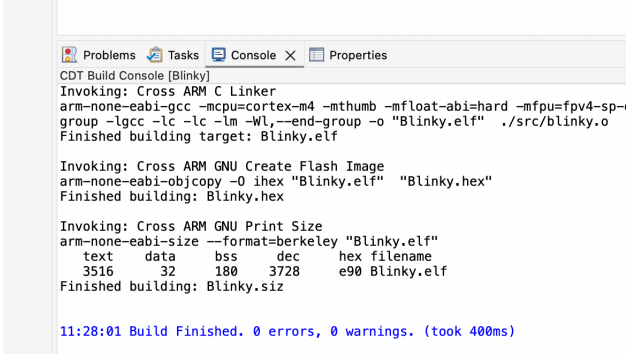
Step 2—Build the Firmware

In the **Project Explorer**, select the Blinky project to make it active. Now, find the **hammer icon** in the toolbar—this action is your compiler, the tool that will transform human-readable code into the machine language the chip understands.



You click it. The Console view at the bottom of the screen springs to life, scrolling with the messages of a successful build and ending with a satisfying confirmation:

```
Finished building target: Blinky.elf
```



With a successful build, your firmware is complete and available as an `.elf` file on your computer. But code is just a whisper until it meets the metal. It's time to bring your physical hardware onto the workbench.

Connect your development board via USB. Go ahead and plug it into your computer now. You should see a power LED light up, a small sign that it's awake and ready.

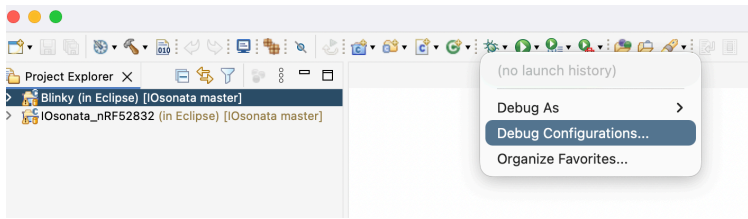
Now that both your digital code and your physical hardware are ready, we need to forge the link between them.

Step 3—Configure the Debugger

Before flashing the code, the IDE must learn the language your specific hardware understands. Think of this like tuning an instrument before a performance, ensuring the connection between your workbench and the board is perfect.

First, identify your debug probe. Are you using a board with a built-in J-Link (e.g., Nordic DK), or a CMSIS-DAP compatible probe (e.g., IDAP-Link)? Once you know, follow the appropriate guide below.

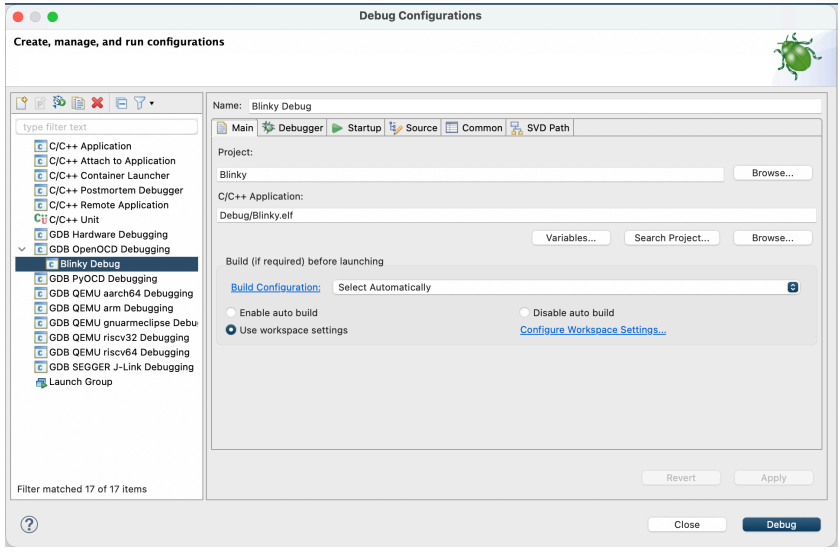
To begin, select the Blinky project in the Project Explorer. Find the **little bug icon** in the toolbar, click the small arrow beside it, and select **Debug Configurations.**



Option A—For CMSIS-DAP Probes with OpenOCD

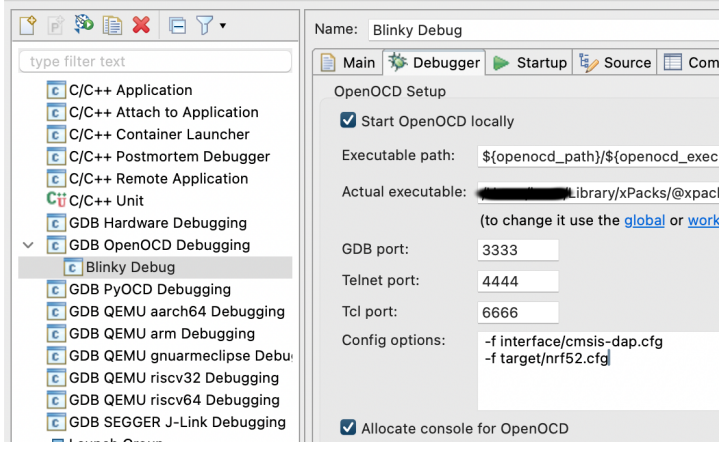
OpenOCD is a powerful, open-source debugger that works with many probes. It requires a configuration script to know about your board.

In the Debug Configurations window, double-click "**GDB OpenOCD Debugging**" to create a new configuration. Eclipse will automatically create a new configuration named "Blinky Debug."



Now switch to the "Debugger" tab. This spot tells OpenOCD how to talk to your hardware. In the "Config options" field, enter two parameters, each on its own line:

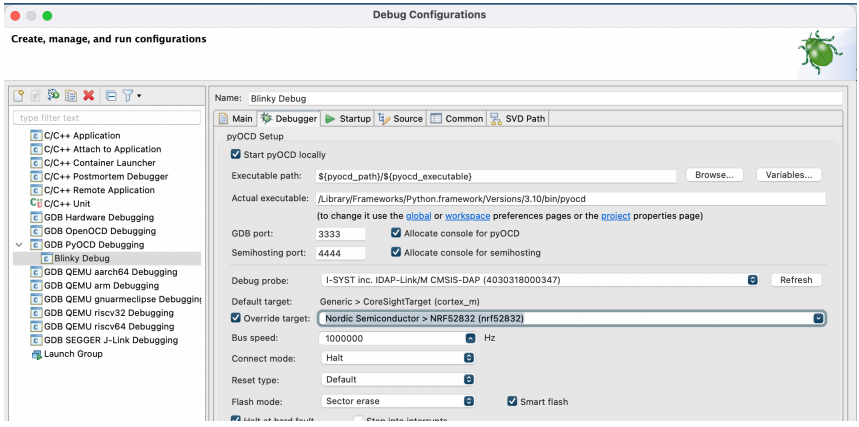
-f interface/cmsis-dap.cfg (This tells OpenOCD to use the standard CMSIS-DAP interface.)
-f target/nrf52.cfg (This selects the Nordic nRF52 as the target microcontroller.)



Option B: For CMSIS-DAP Probes with PyOCD

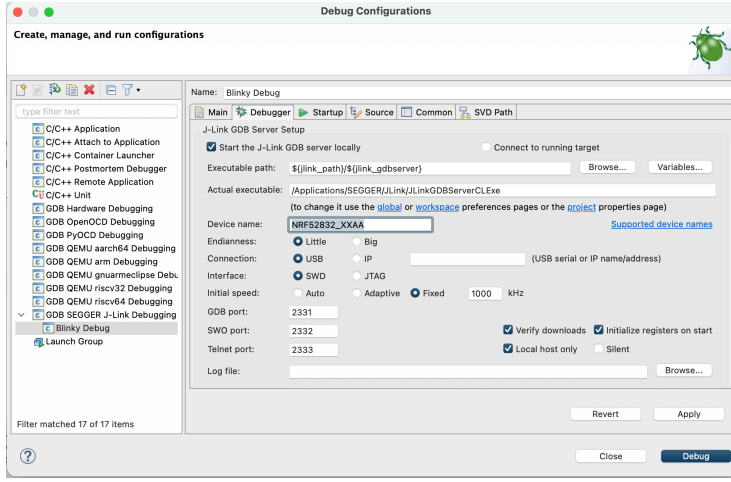
PyOCD is notable for its ability to automatically detect connected debug probes. In the Debug Configurations window, double-click "**GDB PyOCD Debugging**" to create the "Blinky Debug" configuration.

In the "**Debugger**" tab, select your specific probe from the "Debug probe" dropdown list. Then, to ensure you are targeting the correct chip, check the "**Override target**" box and select your specific MCU (e.g., nRF52832).



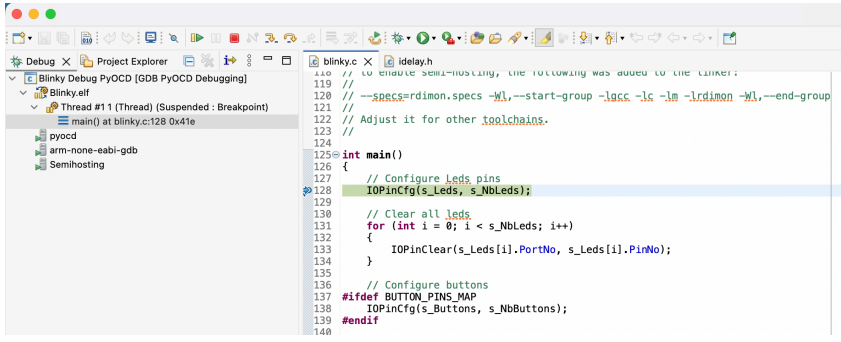
Option C: For J-Link Probes (e.g., Nordic DKs)

If your board uses a J-Link, use the native SEGGER integration. In the Debug Configurations window, double-click "GDB SEGGER J-Link Debugging" to create the "Blinky Debug" configuration. Verify the .elf file path in the "Main" tab. Then, in the "Debugger" tab, type or select your specific microcontroller under "Device name" (e.g., nRF52832_XXAA).



Step 4—Flash & Start Debugging

With your configuration set, click the **Debug** button. Eclipse immediately begins flashing the firmware to your board, switches to the Debug Perspective, and pauses execution right at the start of `main()`.



This is it. You're inside the chip.

Step 5—See It Blink

The program is paused. To let it run, click the **Resume** icon in the toolbar. Now, look at the board. There. A pulse. The LED blinks. It's subtle, but it's everything.

Pitfalls

- **Wrong MCU selected** → flash fails or won't halt at main
- **Board locked** → erase required
- **Wrong LED pin map** → check schematic

One Action Now

The code is already paused when the debug session begins. Instead of clicking **Resume**, use the **Step Over** icon to step through your code line by line. Watch the variables, the stack, the flow. You're no longer outside the system. You're inside, steering it.

4.1 I/O PINS: THE LAND

There is no tree without soil. And in firmware, there is no LED, no UART, no I²C, no ADC—none of it without a pin. A single I/O pin is the soil that gives life to everything above it. When the soil is healthy and simple, the whole orchard thrives.

That blinking light on your board isn't magic. It's an electron, an idea made physical, happening on a single pin. In the project you just grafted, one line brought that pin to life. But where does a pin fit in our orchard architecture?

That pin, and all the general-purpose I/O it represents, is **The Land**.

As we briefly saw, The Land is the foundational layer of our ecosystem, made of plain I/O helpers. It's the foundation upon which everything else is built. In this section, we'll look at `IOPins`, like the C++ expression of the Land. You'll see why this layer is intentionally kept lean, fast, and simple. Our rule is straightforward: **keep the dirt simple** so everything that grows from it stays strong.

Mantra: Pins are the soil; everything else grows from it.

4.1.1 GPIO: THE UNIVERSAL PRINCIPLES

Before we can raise a structure, we respect the ground. While every microcontroller family has its own dialect, a pin's core jobs are always the same:

- **Speak**—drive a signal out (tell the world what to do).
- **Listen**—sense a signal in (hear what the world is saying).

That "signal" can take many forms:

- A simple **on/off** state to control a LED.
- A precisely timed **series of pulses**, as used I²C or UART.
- A continuous **analog voltage** from a sensor, measured by the ADC.

Everything else, pull-ups/downs, drive strength, push-pull vs. open-drain, is seasoning. Important helpers that fine-tune electrical behavior, but the main dish is always speaking or listening.

The challenge is dialect: each MCU vendor exposes different low-level spells for the same fundamental. To build a portable system, we need a common language.

In an orchard, the soil gives life to the tree. In a microcontroller, **GPIO gives life to the entire system**. Without pins, there is no Universal Asynchronous Receiver/Transmitter (UART) to transmit data, no I²C to talk to sensors, and no ADC to taste the world. GPIO pins are truly **The Land**—the life-giving layer from which all other capabilities grow.

4.1.2 ABSTRACTION: A COMMON LANGUAGE

Instead of learning every vendor's dialect, use a single, portable vocabulary. You describe a pin's intent once, and each MCU backend does the right thing.

```
/// I/O pin configuration data from iopincfg.h
typedef struct __iopin_cfg {
  int             PortNo; //!< Port number
  int             PinNo;  //!< Pin number
  int             PinOp;  //!< Pin function select index from
0, MCU dependent
  IOPINDIR        PinDir; //!< Pin direction
  IOPINRES        Res;    //!< Pin resistor setting
  IOPINTYPE       Type;   //!< I/O type
} IOPinCfg_t;
```

PinOp note. *Some MCUs select explicit alternate functions (AF0…AFn). Others (e.g., nRF52) don't need an index for plain GPIO.* `PinOp` *stays vendor-agnostic. Each backend maps it.*

4.1.3 CONFIGURATION: THE BLUEPRINT AND THE HAND-MADE

`IOPinCfg()`—The Manufacturing Plan
Create an array of `IOPinCfg_t` that acts as a blueprint. One call applies the plan.

```
// Example pin configuration for a board's LEDs
const IOPinCfg_t g_Leds[] = {
    {LED1_PORT, LED1_PIN, LED1_PINOP, IOPINDIR_OUTPUT, \
     IOPINRES_NONE, IOPINTYPE_NORMAL}, // LED 1
    {LED2_PORT, LED2_PIN, LED2_PINOP, IOPINDIR_OUTPUT, \
     IOPINRES_NONE, IOPINTYPE_NORMAL}, // LED 2
};
const int g_NbLeds = sizeof(g_Leds) / sizeof(IOPinCfg_t);

// Configure all pins in the array at once
IOPinCfg(g_Leds, g_NbLeds);
```

`IOPinConfig()`—The hand-crafted one-off
Configure a **single** pin directly when you truly need bespoke control.

```
IOPinConfig(LED1_PORT, LED1_PIN, LED1_PINOP, IOPINDIR_OUTPUT,
IOPINRES_NONE, IOPINTYPE_NORMAL); // LED 1

IOPinConfig(LED2_PORT, LED2_PIN, LED2_PINOP, IOPINDIR_OUTPUT,
IOPINRES_NONE, IOPINTYPE_NORMAL); // LED 2
```

4.1.4 THE KEY TO PORTABILITY: NO SCRIPTS, NO BSPS

Portability comes from a clean separation between the **logical blueprint** (`IOPinCfg_t` array) and the **physical map** (`board.h`). To move from one MCU family to another, you **don't** change your logic, you just provide a new `board.h` that translates logical names (e.g., `LED1_PORT`) to the new silicon.

No external build scripts, JSON, XML/YAML, Devicetree, Kconfig, or code-gen, only standard C headers your compiler already understands.

Same logical LEDs, two boards — only `board.h` changes:

- Nordic nRF Series board.h
  On many Nordic chips, ports are simple zero-indexed numbers.

```
#define LED1_PORT 0
#define LED1_PIN  17UL
#define LED1_PINOP 0

#define LED2_PORT 0
#define LED2_PIN  18UL
#define LED2_PINOP 0
```

- STM32 Series board.h
  On STM32 devices, ports are often defined by constants like `IOPORTB`

```
#define LED1_PORT IOPORTB
#define LED1_PIN  17UL
#define LED1_PINOP IOPINOP_GPIO

#define LED2_PORT IOPORTB
#define LED2_PIN  18UL
#define LED2_PINOP IOPINOP_GPIO
```

Bottom line. Application code stays untouched. Porting is faster, the learning curve is lighter, and the project remains self-contained and compiler-friendly.

4.1.5 IOPIN CONTROL: THE CORE FUNCTIONS

Once the Land is prepared, you interact with it via a **tiny, fast, portable** set of verbs, defined in each MCU's I/O control header:

```
// Basic Pin Operations, from iopinctrl.h
void IOPinSet(int PortNo, int PinNo);      // Set pin high
void IOPinClear(int PortNo, int PinNo);    // Set pin low
void IOPinToggle(int PortNo, int PinNo);   // Invert pin state
int IOPinRead(int PortNo, int PinNo);      // Read pin state
(returns 0 or 1)
```

These functions are implemented as **static inline**, providing zero-cost abstraction. The compiler replaces a function call with the function's body, eliminating the call/return overhead. You get the readability and portability of a clean API with the raw performance of hand-coded register writes. The abstraction has no runtime penalty.

Abstraction in action — same verb, different dialects:

nRF52:

```
/**
 * @briefSet pin to high (1 logic)
 *
 * @Param        PortNo  : Port number
 * @ParamPinNo   : Pin number
 */
static inline __attribute__((always_inline))
void IOPinSet(int PortNo, int PinNo) {
#if GPIO_COUNT > 1
    NRF_GPIO_Type *reg = nRFGpioGetReg(PortNo);

    reg->OUTSET = (1 << PinNo);
#elif defined(NRF91_SERIES) || defined(NRF53_SERIES)
    if (PortNo & 0x80)
    {
        NRF_P0_NS->OUTSET = (1 << PinNo);
    }
    else
    {
        NRF_P0_S->OUTSET = (1 << PinNo);
    }
#else
    NRF_GPIO->OUTSET = (1 << PinNo);
#endif
}
```

STM32:

```
/**
 * @briefSet pin to high (1 logic)
 *
 * @Param        PortNo  : Port number
 * @ParamPinNo   : Pin number
 */
static inline __attribute__((always_inline))
void IOPinSet(int PortNo, int PinNo) {
    GPIO_TypeDef *reg = (GPIO_TypeDef *)(GPIOA_BASE + (PortNo
<< 10));

    reg->BSRR = (1 << PinNo) & 0xFFFF;
}
```

You keep the clean verb, while the MCU keeps raw speed.

Gotchas (quick sanity check)

- **Active-low LEDs:** "On" can mean writing **0** (common on some DKs).
- **Floating inputs:** Add pull-up/down or you'll read ghosts.
- **Open-drain lines:** Need an external resistor; don't drive push-pull into a shared line.

Close the Land

Keep the dirt small and honest. Declare **where** the pin is, **what** it does, and **how** it behaves electrically. Nothing more. Rhythm belongs to **timers**, sentences to **interfaces** (UART/I²C/SPI), taste to the **ADC**.

Mantra: *No soil, no orchard. No pin, no system.*

Sidebar — Abstraction without configuration bloat
One header (`board.h`) and one blueprint (`IOPinCfg_t[]`). No

EPILOGUE

The journey that began with a single blinking LED has now come full circle. You have walked the rows of a digital orchard, learning to see firmware not as a rigid machine, but as a living ecosystem of patterns and relationships. You have moved beyond the digital switch and embraced an analog mindset, recognizing the seasons that govern a device's lifecycle. You have seen how soil (pins), roots (interfaces), trunk (devices) and fruit (instances and behavior) all fit together in real code.

This book presented the implementation principles of IOsonata, and you have witnessed them in action. The architectural blueprints are now yours. The goal was not merely to learn a framework, but to gain an analog thinking mindset.

The principles of this orchard now travel with you. The workbench is prepared, the tools are sharp, and you are ready.

Your Engineering Takeaways

Having completed this journey, you are now equipped with the principles and practices to:

- **Design and implement** a hardware-agnostic firmware architecture using modern C++ OOD patterns.

- **Write portable device drivers** that are decoupled from their underlying physical bus (I²C, SPI, BLE, etc…).
- **Swap physical interfaces** with a single line of code, dramatically improving scalability and testability.
- **Create testable and maintainable firmware** by separating concerns and defining clear interface between components.
- **Confidently port** the entire architecture to a new ARM Cortex-M microcontroller using a systematic, step-by-step process.

Now it's time to grow your own orchard.

APPENDIX A

MCU-CENTRIC VS BOARD-CENTRIC ARCHITECTURE

This appendix explains the fundamental architectural difference between IOsonata and most embedded frameworks. It forms the technical foundation beneath the "boards as data" concept introduced in Chapter 2.

A.1 TWO APPROACHES TO HARDWARE ABSTRACTION

When designing a portable firmware framework, there are two ways to think about supporting hardware:

- **Board-centric:** The framework's unit of support is the board. Each physical board (e.g., Arduino Uno, Nordic nRF52 DK, STM32 Nucleo) requires its own definition within the framework.
- **MCU-centric:** The framework's unit of support is the MCU. Once a microcontroller (e.g., nRF52832, STM32L476) is supported, every board using that chip is automatically supported.

The difference isn't just philosophical—it has real scaling consequences.

A.2 THE SCALING MATHEMATICS

Board-Centric: O(N) Scaling

In board-centric framework, such as Zephyr, Arduino, or Mbed:

- N boards require N board definitions.

- Each board needs its own folder with configuration files (e.g., DTS, defconfig, Kconfig).
- The framework must explicitly know about each board to support it.
- Adding a new board typically means contributing a board definition back to the framework, or maintaining a fork.

The Zephyr Example: Zephyr contains hundreds of board definition folders live under `boards/`. Each one was manually created and must be maintained as the framework evolves.

The scaling problem:

- If your company produces 12 product variants using the same MCU but different pin assignments, you need 12 board definitions.
- If you produce 50 variants, you need 50 definitions.

The maintenance burden scales linearly with board count: **O(N)**.

MCU-Centric: O(1) Framework Scaling

In IOsonata's MCU-centric architecture:

- One MCU implementation supports any number of boards.
- Board differences are pure data (pin maps in board.h).
- The framework does not "know" boards exist at all—only MCUs.
- Adding a new board means creating a single header file with pin definitions.

The IOsonata Example: Supporting the nRF52832 once means supporting every nRF52832-based board that exists or will ever exist—without the framework knowing about any specific board.

The scaling property:

- Whether your company produces one variant or 1,000 variants, the framework code doesn't change.
- You still create pin map headers (data), but IOsonata itself remains unchanged: O(1) framework maintenance per MCU family.

A.3 A CONCRETE COMPARISON

Consider a real scenario: you're designing a family of IoT sensors. Same nRF52832 chip, but different sensors, different pin assignments, different power supplies across 12 product SKUs.

Board-Centric Approach (Zephyr / Arduino / Mbed)

Initial effort:

- Create 12 board definition folders.
- Write 12 DTS files describing hardware topology.
- Write 12 defconfig files with build options.
- Possibly 12 Kconfig files.
- Submit all 12 upstream or maintain a fork.

Maintenance burden:

- Framework updates may break your board definitions.
- Each board definition must be re-tested and updated.
- Pin changes require modifying DTS and rebuilding the framework.

- Adding SKU #13 means creating another complete board definition.

What the framework sees:

```
boards/
├── mycompany_sensor_v1/
├── mycompany_sensor_v2/
├── mycompany_sensor_v3/
...
├── mycompany_sensor_v12/
```

MCU-Centric Approach (IOsonata)

Initial effort:

- Support nRF52832 once (if not already supported).
- Create 12 board.h header files with pin definitions.

Maintenance burden:

- Framework updates never touch board files.
- Pin changes? Edit a number in one header.
- No framework rebuild required—pin maps are compile-time data.
- Adding SKU #13 means creating one more header file.

What the framework sees:

- Nothing. The framework knows nRF52832 exists.
- Your boards are invisible to it. They're just pin configuration data the application uses.

A.4 THE PIN CHANGE TEST

You feel the difference when hardware changes—as it always does.

Scenario: The hardware team moves UART TX from pin 6 to pin 9 due to a routing conflict.

Board-centric:

- Edit the board's DTS file.
- Regenerate configuration.
- Rebuild the framework.
- Flash and test.

MCU-centric:

- Edit board.h: change UART_TX_PIN 6 to UART_TX_PIN 9.
- Rebuild your app. Done.

A board-centric architecture treats pin assignments as framework structure. An MCU-centric approach treats them as data your application consumes.

A.5 CONNECTION TO ANALOG THINKING

This architectural choice flows directly from the analog thinking in Chapter 2.

- Digital thinking asks: "What boards exist?" It sees each board as a distinct entity that must be cataloged and defined. A new board is a new thing the framework must learn about.

- Analog thinking asks: "What varies, and what stays constant?" It sees boards as variations of an MCU—different expressions of the same underlying silicon. The MCU is the species; boards are individuals within that species.

In the orchard metaphor: Board-centric frameworks catalog every tree. MCU-centric frameworks understand the species and let trees grow wherever they're planted.

When Board-Centric Makes Sense

Board-centric architectures aren't "wrong"—they optimize for different goals.

Advantages:

- Discoverability: a board list shows all supported boards
- Validation: framework maintainers test board definitions
- Turnkey experience: select a known board, everything "just works"
- Community contributions: board definitions are shareable artifacts

Best for:

- Hobbyist / maker ecosystems (Arduino)
- Educational platforms
- Standardized development kits
- Projects where board variety is low

When MCU-Centric Makes Sense

Advantages:

- Zero framework coupling to boards

- O(1) scaling for product families
- No upstream-contribution friction
- Pin changes don't touch framework code
- Works with boards that haven't been designed yet

Best for:

- Product families with many variants
- Custom / proprietary boards
- Rapid hardware iteration
- Companies avoiding lock-in
- Situations where board definitions can't be upstreamed (IP concerns)

The Unbounded Board Support Property

The most profound difference: in a board-centric framework, a board is supported only when someone adds a definition. In an MCU-centric framework, supporting the MCU means supporting an unbounded set of boards—including boards that don't exist yet.

When IOsonata added nRF54L15 support, it simultaneously supported:

- Nordic's DKs
- Your company's custom board
- Your next unbuilt prototype
- Every future nRF54L15 board—manufactured or imagined

This is the architectural consequence of treating boards as data rather than framework entities.

Summary

	Board-centric	MCU-centric
Unit of support	The board	The MCU family
Scaling	O(N) – one definition per board	~O(1) – framework per MCU family
Pin changes	Edit framework config, regenerate, rebuild	Edit a header, recompile the app
New board	Create a new board definition	Create a pin-map header
What framework knows	Every supported board	Only MCU families; boards are just data
Best for	Turnkey dev kits, education, low board variety	Product families, custom boards, rapid iteration

IOsonata chose the MCU-centric path because it aligns with analog thinking: see the MCU as the constant, boards as variations. The result is an architecture that scales gracefully as your product line grows, without coupling your hardware decisions to framework maintenance.

APPENDIX C
BENCHMARKS & METHODOLOGY

This appendix details the methodology behind the performance benchmarks cited in this book, particularly the UART throughput tests introduced in Chapter 1. The goal of these benchmarks is to provide standardized, repeatable measurements of the IOsonata framework's data transfer performance compared to other common firmware stacks.

UART Throughput Benchmark

The primary benchmark measures the maximum sustained data throughput over a UART interface using a Pseudo-Random Binary Sequence (PRBS) test. This test validates the entire data path, from the application-level driver interface down to the low-level interrupt handling and hardware layers.

Test Procedure

The benchmark consists of two components operating in tandem: a dedicated microcontroller transmitter application and a PC-based receiver and verifier.

MCU TRANSMITTER APPLICATION

The microcontroller is flashed with a test application that continuously generates a PRBS8 data pattern. It transmits it over the UART at the interface's maximum rate. To ensure an accurate measurement, no other logging or console output is active during the test. Three versions of this transmitter were compared:

- **IOsonata version** (`uart_prbs_tx.cpp`): Implements the test using the portable `UART` class and its `Tx()` method, demonstrating the framework's abstract device interface.
- **Zephyr RTOS version** (`main.c`): An event-driven implementation using Zephyr's device model, Devicetree for configuration, and an asynchronous callback that triggers the next transmission upon completion of the previous one.
- **Nordic `nrfx` HAL version** (`UartPrbsTxnrfx.c`): A bare-metal implementation written directly against the vendor's hardware abstraction layer, using a callback and a volatile flag to manage the flow of data.

PC RECEIVER AND VERIFIER

A host PC connected to the MCU's UART port runs a Python script (`uartprbs_rx.py`). This script performs two functions simultaneously:

- It receives the incoming serial data stream.
- It generates the exact same PRBS8 sequence locally, comparing every received byte against the expected value in real-time to detect errors.

The script periodically prints the measured throughput in bytes per second and the total count of any mismatched bytes, referred to as "drops".

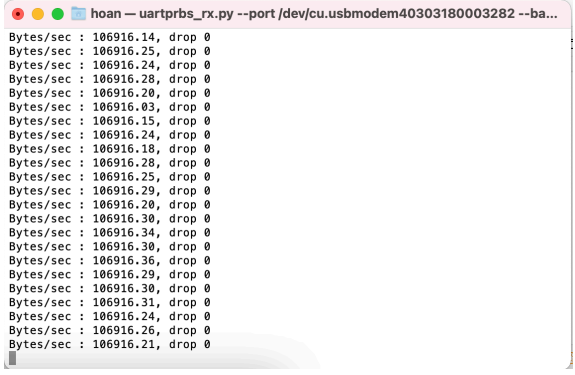
Test Environment

- Hardware
  - Nordic nRF52832 DK
  - Nordic nRF54L15 DK

- Toolchain
  - Compiler: `arm-none-eabi-gcc`
  - Optimization: -O2 with Link-Time Optimization (LTO) enabled
- Test Parameters
  - **Baud Rates**: 115200, 1 Mbaud, and 2 Mbaud
  - **Workloads**: Tests were conducted using interrupt-driven, FIFO-buffered transfers to measure performance under realistic, high-load conditions.

INTERPRETING THE RESULTS

The output from the PC-side validation script provides two key metrics for evaluating performance and reliability:



- **Throughput (Bytes/sec)**: This is the measured data rate. The theoretical maximum payload rate for a standard 8-N-1 UART configuration is the baud rate divided by 10 (1 start bit + 8 data bits + 1 stop bit). For example, at 1,000,000 baud, the theoretical maximum is 100,000 bytes/s. A high-performing driver should achieve a rate very close to this limit.
- **Drop Count**: This is the total number of bytes that did not match the expected PRBS sequence. For a data path to be considered robust and reliable, this value **must be 0**. Any drops indicate data

loss, typically caused by buffer overruns, slow interrupt handling, or hardware limitations.

The full source code for the test applications, Python validation scripts, and project configurations are all available in the IOsonata code repository for transparency and reproducibility.

IOsonata:

https://github.com/IOsonata/IOsonata/blob/master/exemples/uart/uart_prbs_tx.cpp

nrfx HAL:

https://github.com/IOsonata/IOsonata/blob/master/ARM/Nordic/exemples/UartPrbsTxnrfx.c

Zephyr:

https://github.com/IOsonata/IOsonata/blob/master/ARM/Nordic/nRF52/nRF52832/exemples/UartPrbsTxTest_NCS/src/main.c

Python receiver:

https://github.com/IOsonata/IOsonata/blob/master/Python/uartprbs_rx.py

APPENDIX D
MANUAL WORKBENCH SETUP

This appendix provides manual installation steps for building the development environment introduced in Chapter 3, Section 3.4. Choose this path if you prefer fine-grain control over the installation process or want to understand how each component fits into the workbench.

D.1 INSTALL CROSS-COMPILERS

Depending on the target architecture, install one or both of the primary open-source toolchains:

- **Arm GNU Toolchain**
  This toolchain targets AArch32 bare-metal microcontrollers (like ARM Cortex-M) and provides the arm-none-eabi-gcc executable.

  Download Location: `https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads`

- **xPack GNU RISC-V Embedded GCC**
  This toolchain targets RISC-V bare-metal microcontrollers and provides the `riscv-none-elf-gcc` executable.

  Download Location: https://github.com/xpack-dev-tools/riscv-none-elf-gcc-xpack/releases

Recommendation

The examples in this book use Nordic Semiconductor's nRF series MCUs, which are based on the ARM architecture. **Install the Arm GNU Toolchain to follow along.** Installing the RISC-V toolchain is optional, but doing so will give you a more versatile workbench for future projects.

Callout: How Eclipse Finds Your Tools

The **main way** to connect these compilers to the IDE is to set the path directly in **Eclipse's `Global Toolchains` settings**. We will cover this when we configure the workbench in a later step.

Follow the instructions for your operating system.

Windows

1. Download and run the installer:
   • **Arm GNU Toolchain for Windows (.exe)** – from Arm's site above.
   • **xPack RISC-V GCC (.zip)** – extract it manually.

2. Add each toolchain's `bin` folder to your System PATH.
   Example:
   `C:\Program Files\Arm GNU Toolchain\bin`

3. Verify in PowerShell:
   ```
   arm-none-eabi-gcc --version
   riscv-none-elf-gcc --version
   ```

macOS

1. Download the `.tar.xz` archives from the links above.

2. Extract them into `/opt` for a permanent location:

   ```
   sudo mkdir -p /opt
   sudo tar -xvf gcc-arm-*-mac-x86_64-arm-none-eabi.tar.xz -C /
   opt
   sudo tar -xvf xpack-riscv-none-elf-gcc-*.tar.gz -C /opt
   ```

3. Add both to your PATH in `~/.zshrc` (or `~/.bashrc`):

   ```
   export PATH=/opt/gcc-arm-VERSION/bin:$PATH
   export PATH=/opt/riscv-none-elf-gcc/bin:$PATH
   ```

4. Verify:

```
arm-none-eabi-gcc --version
riscv-none-elf-gcc --version
```

Linux (Ubuntu/Debian example)

1. Download the Linux archives from Arm and xPack.

2. Extract to `/opt`:

```
sudo mkdir -p /opt
sudo tar -xvf gcc-arm-*-x86_64-arm-none-eabi.tar.xz -C /opt
sudo tar -xvf xpack-riscv-none-elf-gcc-*.tar.gz -C /opt
```

3. Add to PATH in `~/.bashrc` or `~/.zshrc`:

```
export PATH=/opt/gcc-arm-VERSION/bin:$PATH
export PATH=/opt/riscv-none-elf-gcc/bin:$PATH
```

4. Verify:

```
arm-none-eabi-gcc --version
riscv-none-elf-gcc --version
```

D.2 INSTALL DEBUG TOOLS

Compiling produces a binary, but you still need a way to load that binary into the microcontroller and observe its behavior while it runs. That's the role of a **debugger** and its companion **host software**. A debugger connects to special pins on the chip—**Serial Wire Debug (SWD)** for ARM Cortex-M or **JTAG** for RISC-V. Through this connection, the debugger can:

- **Flash the MCU:** Write your program into non-volatile flash memory so it runs after reset or power-up.
- **Control execution:** Halt, resume, or reset the CPU.
- **Step through code:** Execute one instruction at a time.
- **Inspect state:** View registers, memory, and variables live.

Required Components

Category	Examples	Purpose
Hardware Probes	J-Link, IDAP-Link, ST-Link	A physical bridge between your PC and the MCU for flashing and debugging
Host Software	SEGGER J-Link Pack, OpenOCD, pyOCD	Enables communication between your IDE and the probe

Windows

SEGGER J-Link Software Pack

- Download: `https://www.segger.com/downloads/jlink/`
- Install: Run `JLink_Windows_<version>.exe` installer.
- Verify:
  `JLinkExe`

OpenOCD

- Download: `https://github.com/xpack-dev-tools/openocd-xpack/releases`
- Verify:
  `openocd --version`

pyOCD (ARM only)

- Install:
  `pip install pyocd`
- Verify:
  `pyocd --version`

macOS

SEGGER J-Link Software Pack

- Download: `https://www.segger.com/downloads/jlink/`

- Install: Run `JLink_MacOSX_<version>.pkg`.
- Verify:
  `JLinkExe`

OpenOCD

- Install:
  `brew install open-ocd`
- Verify:
  `openocd --version`

pyOCD (ARM only)

- Install:
  `pip3 install pyocd`
- Verify:
  `pyocd --version`

Linux (Ubuntu/Debian)

SEGGER J-Link Software Pack

- Download: `https://www.segger.com/downloads/jlink/`
- Install:
  `sudo dpkg -i JLink_Linux*.deb`
- Verify:
  `JLinkExe`

OpenOCD

- Install:
  `sudo apt install openocd`
- Verify:
  `openocd --version`

pyOCD (ARM only)

- Install:
  `pip3 install pyocd`

- Verify:
  `pyocd --version`

Callout: Choosing a Debug Probe

Not all probes speak the same language. Matching your probe to your chip and workflow saves hours of frustration.

Probe	Works With	Interface	Key Strengths	Typical Use
SEGGER J-Link	ARM Cortex-M, many RISC-V	SWD / JTAG	Fast, professional-grade, broad IDE support, extremely stable	Best choice for enterprise and production speed
IDAP-Link	ARM Cortex-M	SWD / JTAG	CMSIS-DAP probe with built-in level shifters (1.8 V–3.6 V), UART bridge, and stand-alone programming via **IDAPnRFProg**(Windows/ macOS/Linux). Supports parallel programming of Nordic nRF devices for production.	Ideal for labs, test benches, and cost-sensitive production workflows
CMSIS-DAP	ARM Cortex-M only	SWD	Open standard; often built into NXP and Microchip dev boards	Excellent for education and open-source projects
ST-Link	STM32 (ARM Cortex-M)	SWD / JTAG	Integrated into most ST Nucleo and Discovery boards	Perfect if you mainly target ST devices

Rule of Thumb:

- For ARM Cortex-M, **J-Link** and **IDAP-Link** are vendor agnostic.
- For RISC-V, start with an **OpenOCD-compatible** probe.
- For production programming and multi-device flash, **IDAP-Link + IDAPnRFProg** is the clear winner in value and efficiency.

D.3 INSTALLING ECLIPSE EMBEDDED CDT

With your compilers and debuggers ready, it's time to assemble the workbench where you'll tend the orchard. This is the **Eclipse Embedded CDT IDE**, a ready-made distribution of Eclipse specifically configured for embedded developers.

What You're Getting

The Embedded CDT IDE bundle contains the essential components for professional firmware development.

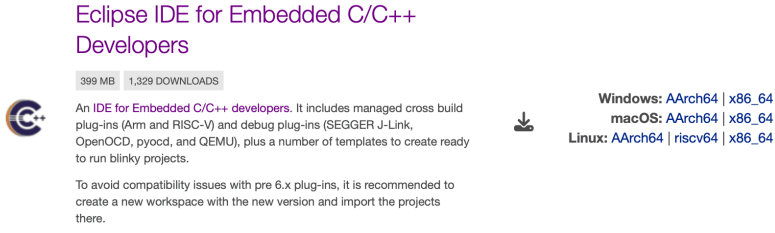
Component	Purpose
Eclipse Platform	The underlying workbench and project manager.
CDT (C/C++ Tools)	Editing, indexing, and build integration.
Embedded CDT Plug-ins	ARM & RISC-V project wizards, debug-probe integration, and peripheral viewers.
Built-in Launch Configs	Pre-made run/debug templates for J-Link, OpenOCD, and pyOCD.

Installation Instructions

First, go to the official Eclipse packages download page in your browser:

`https://www.eclipse.org/downloads/packages/`

Scroll down to find **Eclipse IDE for Embedded C/C++ Developers** and download the appropriate package for your system. Then, follow the instructions for your operating system below.



Windows (AArch64 | x86_64)

1. Download the `.zip` archive.
2. **Extract** the file to a permanent folder, for example: `C:\Program Files\Eclipse.` No installer is required.
3. Run `eclipse.exe` and select a workspace location when prompted.

macOS (AArch64 | x86_64)

1. Download the `.dmg` file.
2. **Install** by mounting the DMG and dragging the `Eclipse.app` into your `/Applications` folder.
3. **Run** the application and choose a workspace location.

Linux (AArch64 | riscv64 | x86_64)

1. Download the `.tar.gz` archive.
2. **Install** by extracting the archive to a system location like `/opt`

```
sudo mkdir -p /opt/eclipse
sudo tar -xvf eclipse-*.tar.gz -C /opt/eclipse
```

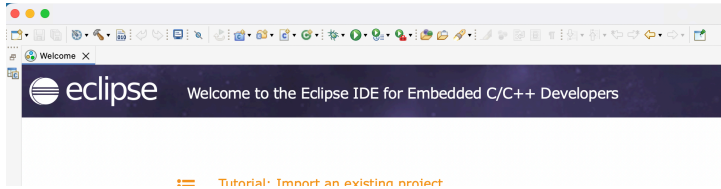
3. **Run** the executable from `/opt/eclipse/eclipse` and select your workspace.

Post-Installation Checks

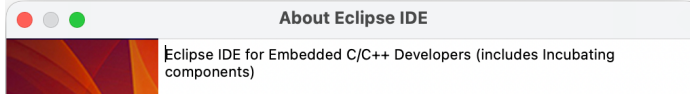
There are two quick ways to confirm you've installed the correct package:

1. Check the Welcome Page When the IDE first opens, the Welcome page should greet you with the title "**Welcome to the Eclipse**

IDE for Embedded C/C++ Developers". This is your most immediate confirmation.

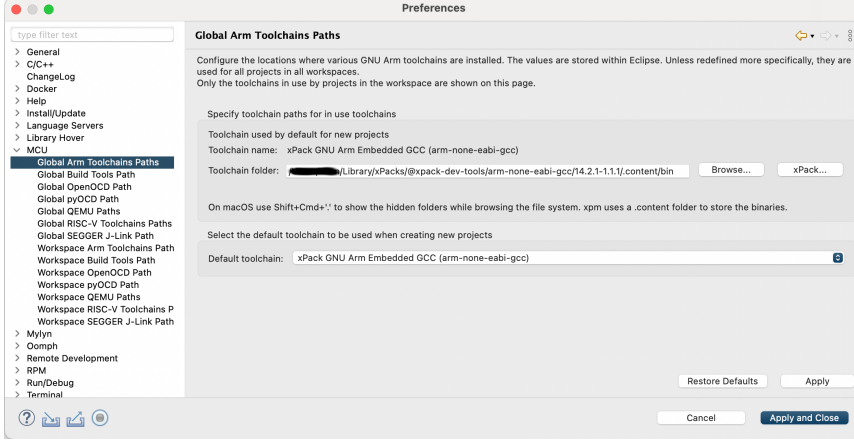


2. Check the "**About**" Dialog. If the Welcome page is closed, go to **Help → About Eclipse**. On macOS it is **Eclipse →About Eclipse**. As long as the dialog box that opens shows the "**Eclipse IDE for Embedded C/C++ Developers**" branding, your installation is correct and complete.



First Run Tips

- **Set Compiler Paths**: Go to Preferences → MCU → Global Toolchains and set the path to the "`…/bin`" folder of your installed toolchain for both ARM & RISC-V and OpenOCD.



- **Disable Unneeded Plug-ins**: If startup feels slow, go to Help → Eclipse Marketplace → Installed and uncheck features you don't use.

D.4 CLONE THE IOSONATA & DEPENDENCIES

With your core tools in place, it's time to bring the project into your workshop. This step involves setting up the IOsonata source code and the third-party vendor libraries it depends on.

Option A: Automated Download Script (Recommended)

For convenience, IOsonata provides a script that automatically creates the necessary folders and clones all required SDKs and libraries. Run the script for your operating system:

- **macOS**

```
/bin/zsh -c "$(curl -fsSL https://raw.githubusercontent.com/
IOsonata/IOsonata/master/Installer/
clone_iosonata_sdk_macos.zsh)"
```

- **Linux**

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/
IOsonata/IOsonata/master/Installer/
clone_iosonata_sdk_linux.sh)"
```

- **Windows (PowerShell)**
  Important: You may need to run PowerShell as an Administrator.

```
powershell -ExecutionPolicy Bypass -Command "iex ((New-Object
System.Net.WebClient).DownloadString('https://
raw.githubusercontent.com/IOsonata/IOsonata/master/Installer/
clone_iosonata_sdk_win.ps1'))"
```

Option B: Manual Cloning

If you prefer to perform each step yourself, follow the instructions below.

1. **Create Your Workspace:** Start by creating a single root folder for all your development work (e.g., `IOcomposer`).

2. **Make an `external` Folder:** Inside your new root folder, create a subfolder named `external`. This is where all third-party libraries will live.

3. **Clone the SDKs:** Populate your `external` folder. The easiest way to do so is using `git.` Open a terminal or command prompt, navigate to your `external` folder, and clone the following repositories:

   - **nRF5_SDK** – Nordic Bluetooth Low Energy SDK
     `git clone https://github.com/IOsonata/nRF5_SDK.git`
   - **nRF5_SDK_Mesh** – Nordic Bluetooth Mesh SDK
     `git clone https://github.com/IOsonata/nRF5_SDK_Mesh.git`
   - **nrfx** – Nordic MCU HAL
     `git clone https://github.com/NordicSemiconductor/nrfx.git`
   - **sdk-nrf-bm** – Nordic nRFConnect Baremetal
     `git clone https://github.com/nrfconnect/sdk-nrf-bm.git`
   - **sdk-nrfxlib** – Nordic nRFConnect libraries
     `git clone https://github.com/nrfconnect/sdk-nrfxlib.git`
   - **BSEC** – Bosch environmental sensor library
     `git clone https://github.com/boschsensortec/Bosch-BSEC2-Library.git`

   After cloning, the Bosch library will be in a folder named `Bosch-BSEC2-Library`. **Rename this folder to `BSEC`**. This process stocks your "Lego shelf" with all the specific pieces IOsonata needs to communicate with the hardware used in this book.

4. **Get IOsonata:** Finally, navigate back to your root folder and clone the IOsonata repository itself.

   ```
   git clone https://github.com/IOsonata/IOsonata.git
   ```

APPENDIX E
CREATE NEW PROJECT

This appendix shows how to create your own Eclipse project that uses IOsonata directly, instead of starting from a preconfigured example.

The goal is to:

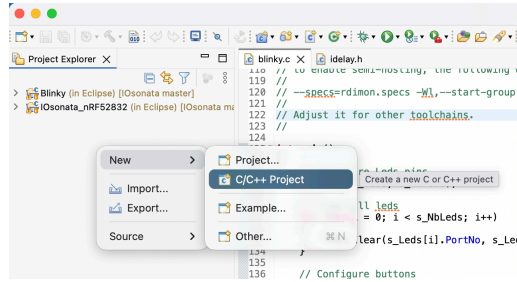
- Create a new Eclipse C++ project (MyBlinky).
- Add a board.h that defines your board's pin map.
- Link in the generic `blinky.c` example from IOsonata.
- Configure include paths, libraries, and linker flags.
- Build and run the project under the debugger.

The steps assume:

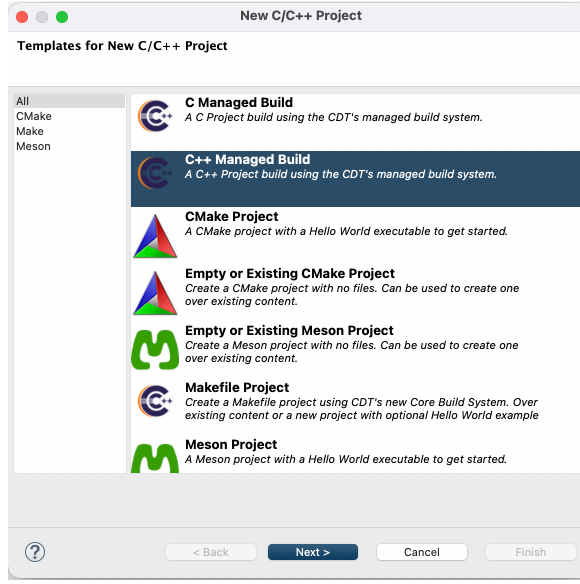
- Eclipse IDE for Embedded C/C++ Developers is installed and configured.
- The IOsonata library project for your MCU (for example, `IOsonata_nRF52832`) is already building cleanly as described in Chapter 3.
- You have a supported board and debug probe connected.

E.1 CREATE A NEW PROJECT

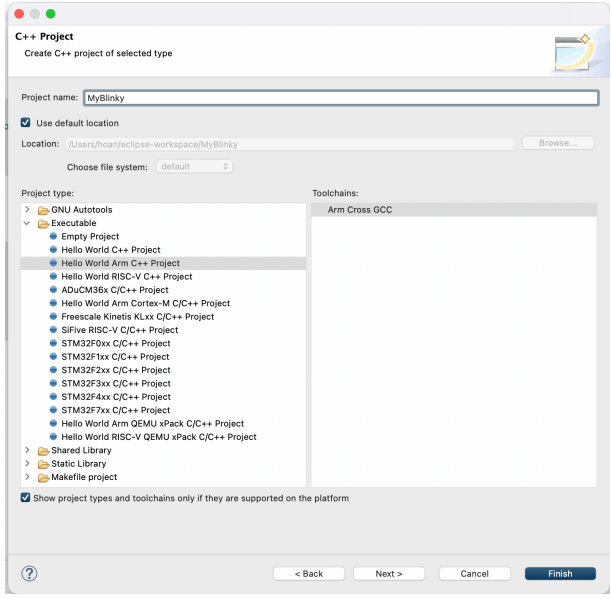
1. In the **Project Explorer** pane, right-click and select **New → C/C++ Project**.



2. The C/C++ Project wizard will appear. On the first page, select **C++ Managed Build** and click **Next**.



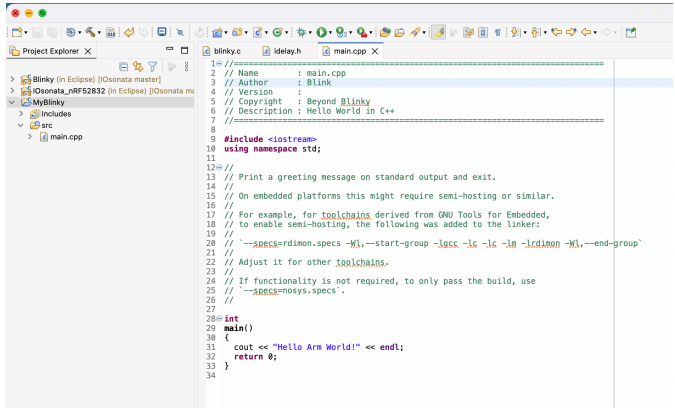
3. Now for the crucial step: under the **Executable** project type, select the **Hello World Arm C++ Project** template. Give your project a name, such as `MyBlinky`, and click **Next**.



4. The next few pages ask for basic information. Enter your name in the Author field and click **Next**. The following page can usually be left with its default settings, so click **Next** again.

5. The final page confirms the cross-compiler toolchain that will be used for the project. Verify that it matches the one you configured in **Chapter 3**.

6. Click **Finish**. Your new project, `MyBlinky`, will appear in the Project Explorer containing a default main.cpp file—a hello world canvas ready to compile.



E.2 SOURCE CODE REFERENCING

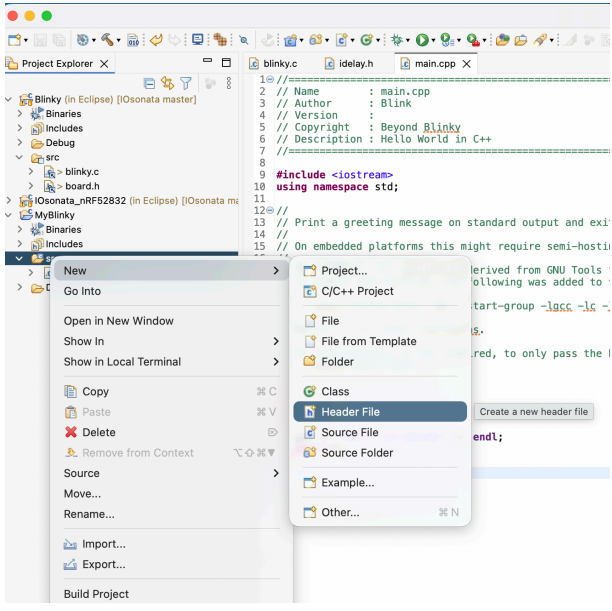
The "Hello World" template gave us a solid project structure, but its default `main.cpp` isn't what we need. The IOsonata framework already provides a generic Blinky example that is portable across different microcontrollers. Our goal is to connect the new project to that generic code while providing our own board-specific details.

A. Create Your `board.h` Pin Map

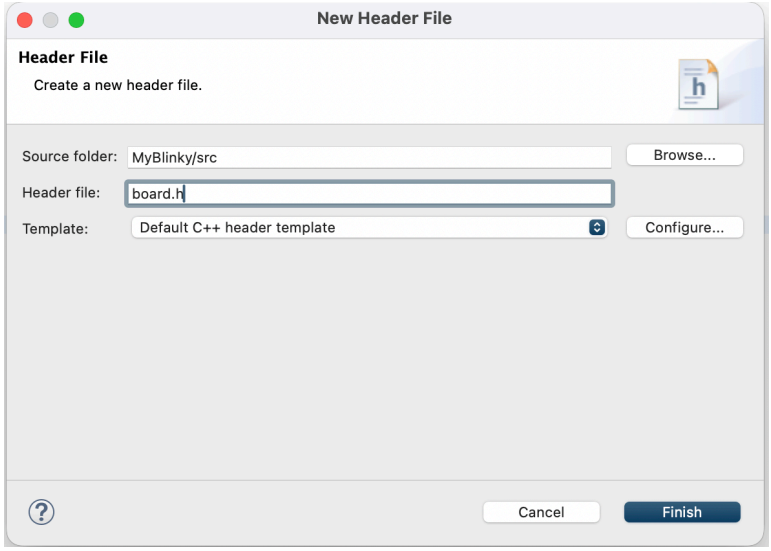
We need a place to define the unique pinout for your specific board. This keeps all the hardware-specific details in one place.

1. Expand the `MyBlinky` project in the **Project Explorer** and select the `src` folder.

2. Right-click on `src` and select **New → Header File**.



3. In the "Header file" field, type `board.h` and click **Finish**.



The new `board.h` file will be added to your project and opened. Paste the following pin definitions into the file. These are for the BLYST-NANO-DK board. Replace them with the appropriate definitions for your own hardware.

```
// Predefine pin maps for the BLYST Nano (nRF52832) based
boards
#include "blystnano_boards.h"

#define BUTTON_PINS_MAP.         BLYST_NANO_DK_BUT_PINS_CFG
#define BUT1_SENSE               BLYST_NANO_DK_BUT1_SENSE
#define BUT2_SENSE               BLYST_NANO_DK_BUT2_SENSE

#define BUT1_INT                 0
#define BUT1_INT_PRIO.           6
#define BUT2_INT.                1
#define BUT2_INT_PRIO.           6

#define LED_PINS_MAP             BLYST_NANO_DK_LED_PINS_CFG
#define PULSE_TRAIN_PINS_MAP     BLYSTNANO_PULSE_TRAIN_PINS
```

B. Remove the Template's `main.cpp`

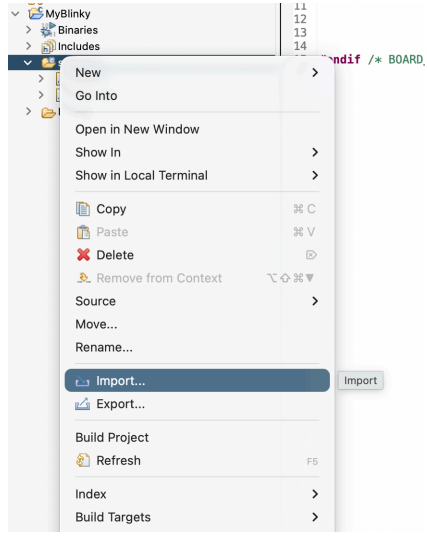
Now, we'll remove the default main file that the template created.

1. In the `src` folder, find the `main.cpp` file.
2. Right-click on it and select **`Delete`**. Confirm when prompted.

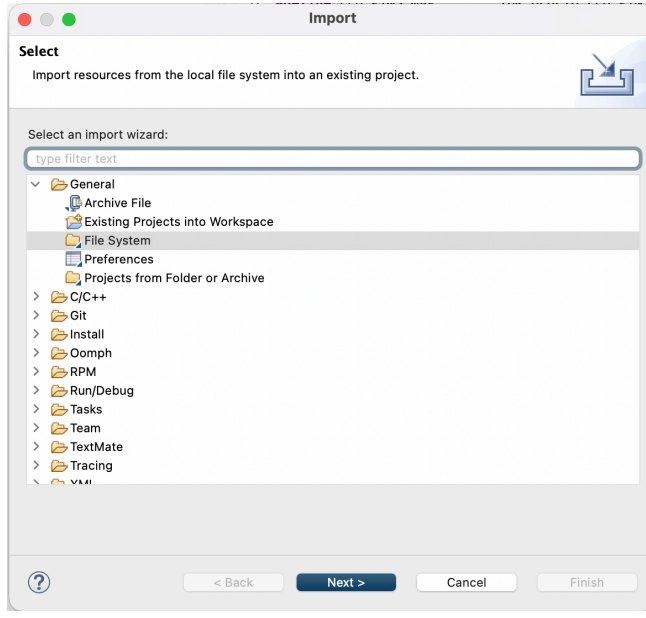
C. Link the Generic `blinky.c` Example

Finally, we will bring in the generic Blinky source code. We won't copy the file. Instead, we'll create a link to it. This is a beautiful feature of Eclipse that lets your project reference a file outside its own directory, ensuring you're always using the latest version from the library. This ability to manage external files as virtual links is a powerful project management feature within Eclipse. Unlike a standard filesystem link, which can easily break when a project is moved to another computer, these IDE-managed links are highly portable, making it easy to share code across a team.

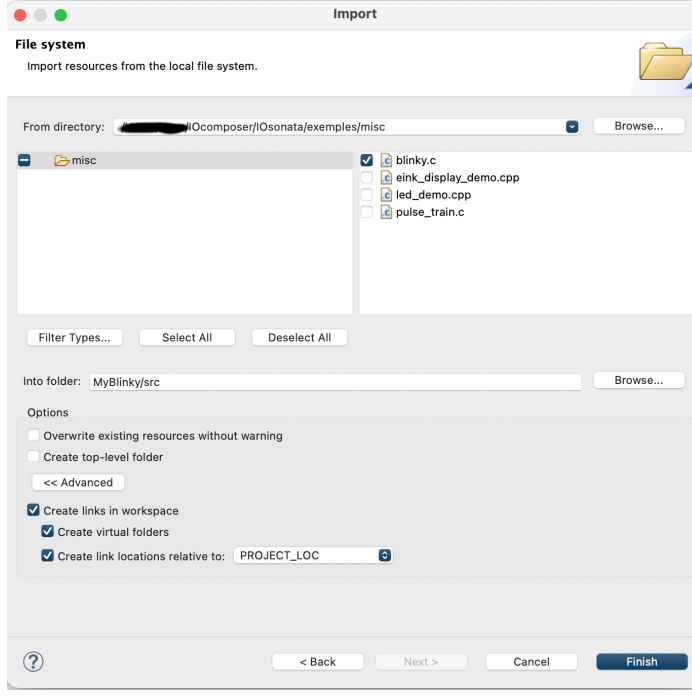
1. Right-click on the `src` folder again and select **`Import`**....



2. In the Import dialog, expand the **General** folder and select **File System**, then click **Next**.



3. Click the **Browse...** button and navigate to the folder containing the generic examples, located at …/`IOsonata/exemples/misc`.



4. In the left pane of the dialog, you will see a list of files. Place a checkmark next to `blinky.c`.

5. Click the "**Advanced >>**" button at the bottom to reveal more options.

6. Check the box that says "**Create links in workspace**".

7. Click **Finish**.

You will now see `blinky.c` inside your src folder. Notice that its icon has a small arrow, indicating it is a link to the original file, not a copy. This generic `blinky.c` file is designed to automatically include the `board.h` from your project, allowing it to adapt to your specific hardware.

At this point, you can click the hammer icon to compile your new project. If you do, you will likely encounter a flood of errors. Don't be alarmed—this is expected. Because we started with a nearly blank canvas, we haven't yet told the compiler where to find the IOsonata library headers, nor have we provided the linker with a "map" (the **linker script**) to correctly place your code into the microcontroller's memory.

Our next step is to configure these essential paths and dependencies. This is the final part of the grafting process, where we truly connect your project to the rootstock.

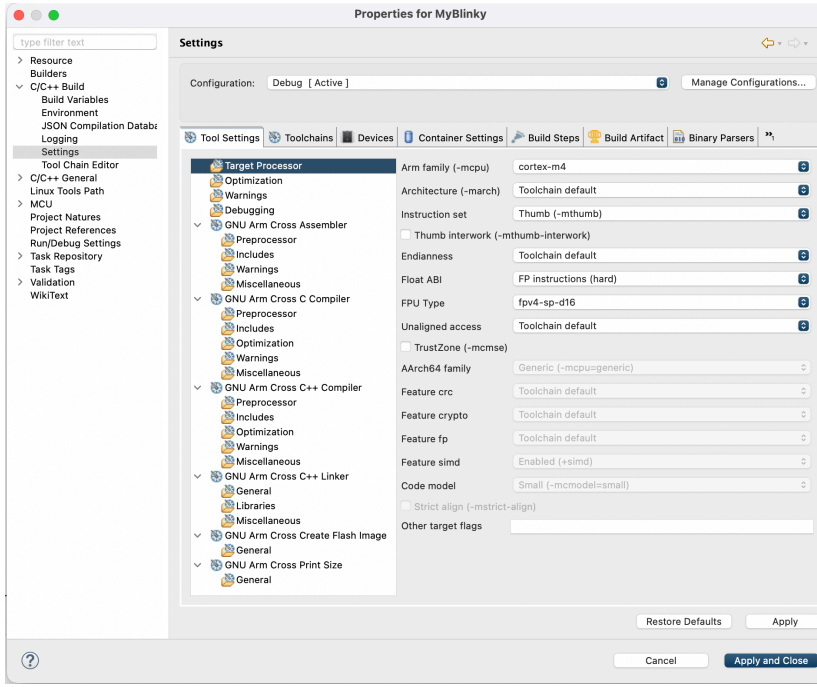
E.3 THE PROJECT SETTINGS

This is the most detailed part of the process, where we configure the project's properties to work with the IOsonata framework.

Right-click your `MyBlinky` project in the Project Explorer and select **Properties** from the bottom of the menu. This action opens the central control panel for your project. In the left pane, expand the **C/C++ Build** section and click **Settings**.

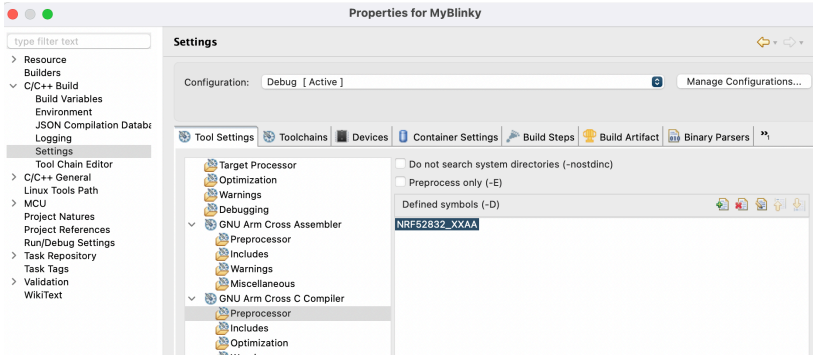
1. **Configure the Target Processor**
   First, we'll tell the compiler about our specific MCU. In the **Tool Settings** tab, select **Target Processor**. Configure the following settings:

   - Arm family: `Cortex-M4`
   - Float ABI: FP Instructions (hard)
   - FPU Type: `fpv4-sp-d16`



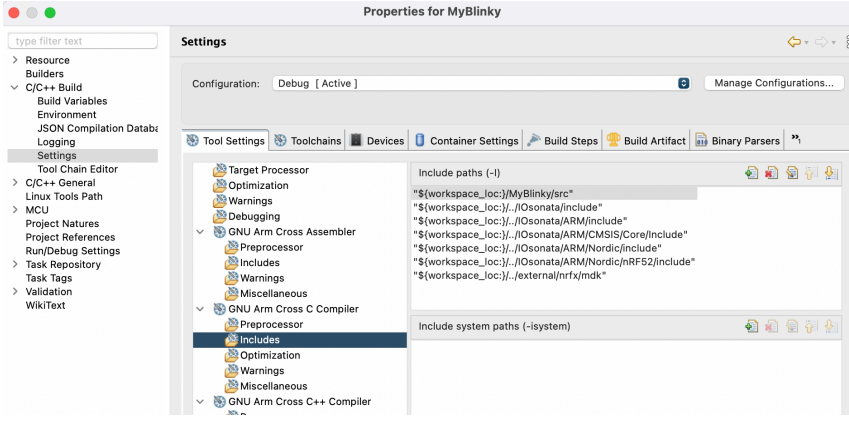
2. **Configure C Compiler Settings**
   Now, we'll set up the paths for the C compiler (for the `blinky.c` file). Expand **GNU Arm Cross C Compiler**.

   - **Preprocessor** subsection. Under "Defined symbols (-D)," add `NRF52832_XXAA`.



- Go to the **Includes** subsection. Under "Include paths (-I)," add the following paths:

```
“${workspace_loc:}/MyBlinky/src”
“${workspace_loc:}/../IOsonata/include”
“${workspace_loc:}/../IOsonata/ARM/include"
"${workspace_loc:}/../IOsonata/ARM/CMSIS/Core/Include"
“${workspace_loc:}/../IOsonata/ARM/Nordic/include"
“${workspace_loc:}/../IOsonata/ARM/Nordic/nRF52/include"
“${workspace_loc:}/../external/nrfx/mdk”
```



- Go to the **Optimization** subsection. For "Language standard," select **ISO C17 (-std=c17)**.

3. **Configure C++ Compiler Settings**
   Apply the same settings to the C++ compiler. Expand **GNU Arm Cross C++ Compiler**.

   - In the **Preprocessor** and **Includes** subsections, add the same entries as above.

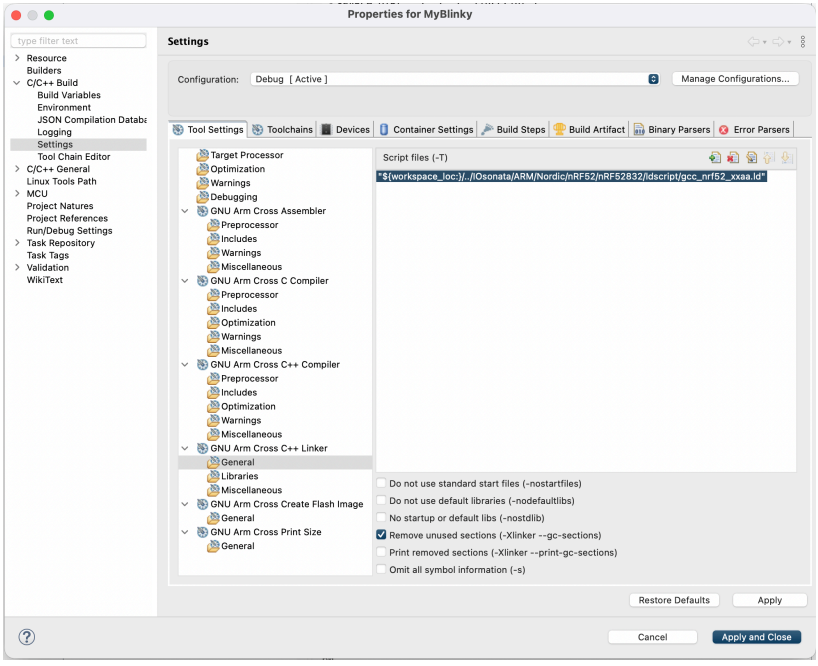
- Go to the **Optimization** subsection and configure the following:
  - Language standard: Select **`GNU ISO C++23 (-std=c++23)`**.
  - Ensure the "Do not use exceptions" box is checked.
  - Ensure the "Do not use RTTI" box is also checked.

  These settings ensure your code is modern, efficient, and avoids unnecessary overhead.

4. **Configure Linker Settings**
   We've told the compiler what to do. Now, we need to instruct the **linker**. The linker takes all compiled pieces and "links" them into the final `.elf` executable. We must provide it with a memory map and the path to the `IOsonata` library.

   In the left pane, select **Cross ARM C++ Linker**.

   General Settings



First, select the **General** subsection. In the "Script file (-T)" field, provide the path to the linker script. This file is the "map" that tells the linker where to place every piece of code and data.

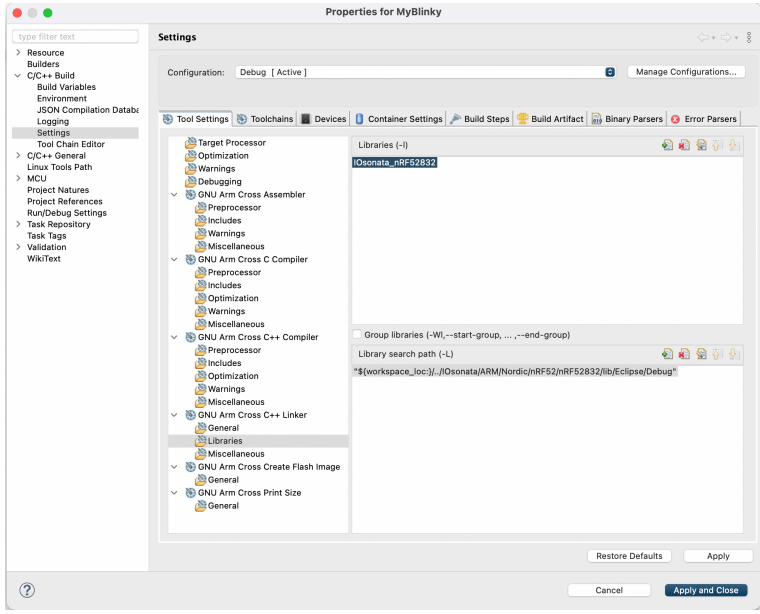
```
"${workspace_loc:}/../IOsonata/ARM/Nordic/nRF52/nRF52832/
ldscript/gcc_nrf52_xxaa.ld}"
```

Library Settings

Next, select the **Libraries** subsection. Here, you'll tell the linker to use the `IOsonata` library that you built in Chapter 3.

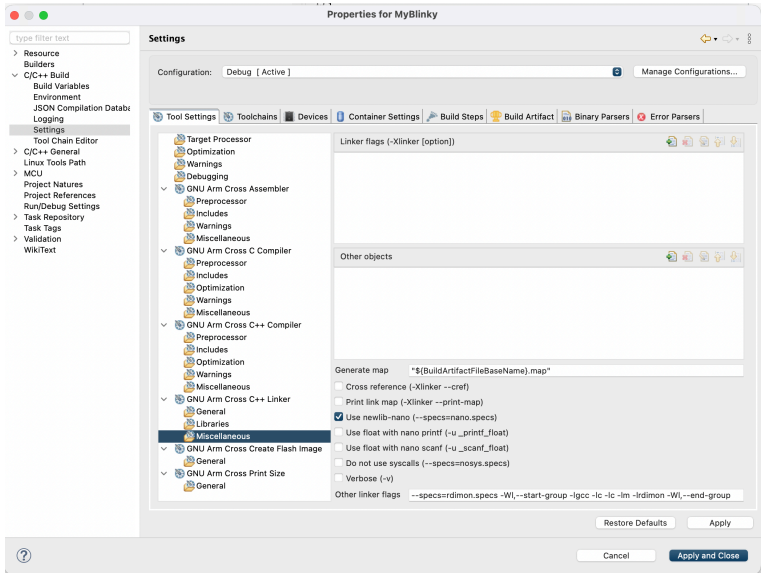
1. In the "**Libraries (-l)**" field, click the 'Add' icon (+) and enter `IOsonata_nRF52832`.

2. In the "Library search path (-L)" field, click the 'Add' icon (+) and enter the path to the folder containing that library file:
   `"${workspace_loc:}/../IOsonata/ARM/Nordic/nRF52/nRF52832/lib/Eclipse/Debug"`



3. Now, select the **Miscellaneous** subsection and check the box for "**Use newlib-nano (-specs=nano.specs)**". This is a critical step that tells the linker to use a size-optimized

version of the standard C library, significantly reducing your firmware's final size.



5. **Configure the Release Build**
   Finally, we'll set up the **Release** configuration to create a lean, standalone program. A key difference from the Debug build is how functions like `printf` are handled. By default, the Debug build uses a library called `rdimon` to route `printf` output to your Eclipse console. This library is great for debugging, but it requires a debug probe to run the firmware. For a Release build, we must remove this dependency.

   1. At the top of the Properties window, use the "Configuration:" dropdown to switch from "**Debug**" to "**Release**".

   2. Now, you must repeat parts A, B, C, and D for the Release configuration, applying the exact same settings you did for the Debug build.

3. The one crucial change is for the linker flags. Navigate to **Cross ARM C++ Linker → Miscellaneous**.

4. In the "**Other flags**" field, replace the default text with the following to remove the debug monitor dependency:
   `-Wl,--start-group -lgcc -lc -lm -Wl,--end-group`

5. Click **Apply and Close**.

With this change, your Release build will no longer depend on the debug monitor and can run without a probe attached.

E.4 BRING YOUR CREATION TO LIFE

With the project fully configured, it's time for the final step.

This time, when you click the **hammer icon**, the build will complete without any errors. Follow the same steps you learned in Section 4.0 to start a debug session for your `MyBlinky` project.

Once the debugger pauses at `main()`, click Resume. The LED on your board will start blinking.

APPENDIX F
DRIVER PATTERN CARDS

This appendix summarizes the core design patterns used throughout the book for quick reference.

F.1 THE ROOT INTERFACE (`DEVICEINTRF`) PATTERN

- **Use When:** A driver must exchange data without coupling to any specific physical bus (e.g., I²C, SPI, UART, BLE).
- **Goal:** Write the driver logic once. Then, graft it into any compatible bus implementation with a single line of code during initialization.
- **Solution:** Define an abstract base class (`DeviceIntrf`) that specifies a minimal, universal interface for byte-oriented data transfer (e.g., `Enable`, `Disable`, `Rate`, `Read`, `Write`, `StartTx`, `TxData`, `StopTx`, `StartRx`, `RxData`, `StopRx`). Implement concrete classes (e.g., `UART`, `I2C`, `SPI`, `BtIntrf`) that inherit from `DeviceIntrf` and fulfill the interface using specific hardware peripherals.
- **Result:** Drivers interact only with a `DeviceIntrf*` pointer. Swapping buses, creating mock interfaces for testing, or porting the driver to new hardware only requires changing the concrete object passed during initialization. The driver code itself remains unchanged.

F.2 THE DEVICE OBJECT PATTERN

- **Use When:** Whenever modeling a hardware peripheral or logical component that has a distinct lifecycle, configuration, and communicates via a standard interface (e.g., sensors, displays, memory chips, power management ICs).
- **Goal:** Create self-contained, predictable, reusable, and testable components that encapsulate hardware-specific details and are decoupled from the underlying communication bus.
- **Solution:** Create an abstract base class (`Device`) defining a universal interface, including lifecycle management methods (`Enable`, `Disable`, `Reset` - often pure virtual) and potentially shared communication helpers (`Read`/`Write` wrappers using `DeviceIntrf*`). Derive specialized abstract branches (e.g., `Sensor`, `Display`) inheriting virtually from `Device` to add domain-specific roles and behaviors. Finally, create concrete device classes (e.g., `TphBme280`, `LcdHX8357`) that inherit from the appropriate branch, implement all pure virtual functions, and contain the hardware-specific logic. Use dependency injection via an `Init()` method to provide the object with its configuration, `DeviceIntrf*` (the "Soil"), and optional `Timer*` (the "Rhythm.")
- **Result:** Uniform behavior and lifecycle management across all device drivers, easy swapping of underlying buses (polymorphism via `DeviceIntrf*`), simplified testing, painless porting, and a coherent power/recovery strategy.

APPENDIX H
C++ REFRESHER

This appendix exists solely to refresh language features used in the object-oriented designs introduced in Chapters 5 and 6. It is not a C++ tutorial.

H.1 CLASSES AND OBJECTS: TYPE AND INSTANCE

A **class** defines a type: a set of data (member variables) and functions (member functions) that operate on that data.

An **object** is an instance of that type stored in memory.

```
// Class definition
class Counter {
public:
    void Init(int startValue) {
        value = startValue;
    }

    void Increment() {
        value++;
    }

    int GetValue() {
        return value;
    }

private:
    int value;
}; // Note the required semicolon
```

Creating objects:

```
Counter counterA;
Counter counterB;

counterA.Init(0);
```

```
counterB.Init(100);
counterA.Increment(); // counterA now holds 1
```

Key points:

- `public` members form the **interface** visible to users of the type.
- `private` members are internal implementation details.
- Each object has its **own copy** of all member variables. `counterA.value` and `counterB.value` occupy separate memory locations.

This maps directly to device instances in the framework: each device object holds its own state.

H.2 INHERITANCE: INTERFACE REUSE

Inheritance allows one class to reuse the interface and behavior of another.

```
class Vehicle {
public:
    void StartEngine() {
        // implementation
    }
};

class Car : public Vehicle {
public:
    void HonkHorn() {
        // implementation
    }
};
```

Usage:

```
Car myCar;
myCar.StartEngine(); // inherited
myCar.HonkHorn();    // Car-specific
```

Key points:

- Inheritance expresses an **"is-a" relationship** (a Car is a Vehicle).
- Inheritance alone does **not** provide runtime polymorphism.
- Runtime substitution requires **virtual functions**, covered next.

In Chapters 5–6, inheritance is used to build interface hierarchies, not deep class trees.

H.3 POLYMORPHISM: INTERFACE-BASED DISPATCH

Polymorphism allows code to operate on an **interface** rather than a concrete implementation.

This is the foundation of hardware portability in the architecture.

Abstract Classes and Pure Virtual Functions

An **abstract class** defines an interface. It cannot be instantiated directly.

```
class Drawable {
public:
    virtual void draw() = 0;  // pure virtual
    virtual ~Drawable() {}
};
```

Concrete implementations:

```
class Circle : public Drawable {
public:
    void draw() override {
        // draw circle
    }
};

class Square : public Drawable {
public:
    void draw() override {
```

```
        // draw square
    }
};
```

Runtime dispatch through the interface:

```
Circle myCircle;
Square mySquare;

Drawable* item = &myCircle;
item->draw();   // calls Circle::draw()

item = &mySquare;
item->draw();   // calls Square::draw()
```

Key points:

- The call target is selected **at runtime**.
- The calling code does not change when implementations change.
- This is virtual dispatch.

Relation to Chapters 5–6

In this book, polymorphism appears in two equivalent forms:

1. **C++ virtual functions** (base class interfaces)
2. **C-style function tables** (`DevIntrf_t`) embedded in device objects

Both achieve the same architectural goal. Application code depends on an interface, not on hardware-specific implementation details.

Cost and Benefit in Embedded Systems

Virtual dispatch introduces a small, predictable overhead:

- One table of function pointers per class (or per device interface)
- One pointer reference per call site

In the IOsonata architecture:

- C++ uses compiler-generated virtual dispatch
- C and C++ both use explicit function tables (`DevIntrf_t`)

In both cases, the overhead is negligible compared to peripheral state, FIFOs, and buffers, and it enables:

- Hardware substitution without application rewrite
- Clear separation between interface and implementation
- Deterministic, testable driver behavior

H.4 ADVANCED RELATIONSHIPS

A. Composition (**Has-A** Relationship)

Composition builds objects from other objects.

```
class Engine { /* ... */ };
class Wheel  { /* ... */ };

class Car {
private:
    Engine engine;
    Wheel wheels[4];
};
```

Key points:

- Composition delegates responsibility to specialized components.
- This is the **primary** construction pattern used in Chapters 5–6.

B. Virtual Inheritance (Solving the Diamond Problem)

When multiple inheritance paths share a common base, duplicate base instances can occur.

```
class PoweredDevice { /* ... */ };

class Printer : virtual public PoweredDevice { /* ... */ };
class Scanner : virtual public PoweredDevice { /* ... */ };

class MultifunctionDevice : public Printer, public Scanner {
    // single PoweredDevice instance
};
```

Key points:

- `virtual` inheritance ensures only **one shared base instance**.
- Required when building interface hierarchies with multiple inheritance.

Embedded example:

This technique is used by `Sensor : virtual public Device` to safely compose complex sensor drivers without duplicating the underlying `Device` state.

About the author

Nguyen Hoan Hoang is a hardware and software engineer and the creator of IOsonata, a modern C++ HAL used in medical and industrial products. He has designed certified Bluetooth modules and development kits, taking products from schematics and RF layout all the way to validation and production. A graduate of École Polytechnique de Montréal (B.Eng., Electrical Engineering), he focuses on architectures that stay fast, reusable, and inexpensive to maintain over a product's lifetime. **Beyond Blinky** captures the design patterns behind IOsonata and his "**analog-thinking**" approach to firmware: model flows, separate policy from mechanism, wire everything at a clear composition root, and verify with repeatable criteria.

About this book

Hit the "custom board wall" after your Arduino prototype worked perfectly? Drowning in vendor HAL spaghetti that breaks every time marketing asks for a new board variant?

Most firmware starts with a blinking LED but grows into a tangle of copy-paste code locked to specific hardware. The problem isn't the code—it's the mental model.

This book teaches you to see firmware through **The Orchard**: a living metaphor where hardware is soil, interfaces are roots, and devices are trees bearing fruit. Shifting from procedural code to this object-oriented view changes how you work:

- **Twelve board variants** become twelve pin maps—not twelve porting projects.
- **Swap I²C for SPI**—or route traffic over BLE—without rewriting drivers. Your code doesn't care which bus you picked.
- **Unit-test firmware** on your desktop, yet ship with bare-metal C performance.

Whether you're moving from prototype to production or breaking free from vendor lock-in across product families, this book lays out an architecture that scales.

It distills proven patterns from years of developing and refining **IOsonata**, the open-source C++ framework for MCU-centric design, and includes complete porting guides for Nordic nRF, STM32, and Renesas platforms.

Master these blueprints and gain more than a framework: drivers that stay portable, refactors that shrink to a **single line**, and hardware details that stay contained.

Welcome to the Orchard. Now it's time to grow your own.

To Blinky and beyond—let's make your IO sing!