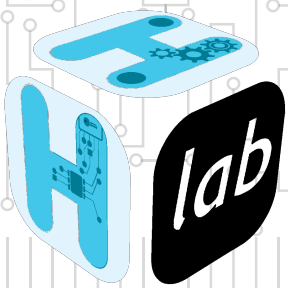
H2LAB-STIG-EMDEV
08-2026

Embedded systems: Secure bare-metal firmware development in C



STIG H²Lab

Information

Created in 2020, H2Lab is a non-profit organization dedicated to designing, developing, and sharing open-hardware and open-source solutions for experimentation around embedded systems. Its main areas of work are:

- Design of hardware platforms for hardware and software analysis
- Development of open alternatives to proprietary, often opaque tools
- Publication of technical guides and recommendations to promote best practices in security, privacy, and sustainability across the ecosystem of connected devices and embedded systems

Support for researchers, teachers, and students through the provision of tools, educational content, and training.
Academic partnerships to encourage sharing of resources and knowledge.

Revision history

Version	Date	Author	Changes
1.0	August 2026	H2Lab	Initial version of the guide

Contents

1
Glossary and acronyms

1.1 Glossary

ABI. Binary contract: function call, registers, stack, alignment and object format between compiled units.

Memory barrier. Primitive that constrains the observable order of reads and writes across processors, devices, or the compiler.

Undefined behavior. Case for which ISO C imposes no requirement.

Bootloader. Boot software that prepares and selects another firmware image.

BootROM. Boot code burned into a SoC or microcontroller by its manufacturer.

Coherent cache. Set of caches for which hardware maintains a compatible view of shared lines.

Coprocessor. Compute or control unit distinct from the general-purpose core, for example an FPU, a DSP, or a cryptographic accelerator.

DMA. Data transfer performed by a device without core intervention for each word.

Endianness. Order of representation of the bytes of an integer in memory.

Interconnect. Hardware network that carries and arbitrates transactions between cores, memories, and peripherals.

Mailbox. Device or register area used to pass a message between domains.

Memory-mapped I/O. Access to a device's registers through addresses in the memory map.

noRTE. Demonstrated absence of run-time error within a scope and under documented assumptions.

Ownership. Temporary property of a resource, in particular a buffer shared between producer and consumer.

Race condition. Result that depends on an unsynchronized interleaving.

Ring buffer. Circular buffer whose read and write positions progress modulo a fixed capacity.

Scratchpad. Fast memory explicitly managed by software, without a necessarily automatic cache policy.

Semaphore. Synchronization object whose counter limits the number of simultaneous accesses to a resource.

Attack surface. Physical, logical, and organizational interfaces exploitable by an attacker [3, 35].

TOCTOU. "Time of check to time of use" defect.

Wear-leveling. Distribution of writes to limit the differential wear of a non-volatile memory.

W ⊕ X. Policy that forbids the same region from being both writable and executable.

XIP. Direct execution of code from a non-volatile memory (*Execute In Place*).

1.2 List of acronyms

ABI. Application Binary Interface.

ACSL. ANSI/ISO C Specification Language.

ANSSI. Agence nationale de la sécurité des systèmes d'information (French national cybersecurity agency).

AAPCS. ARM Architecture Procedure Call Standard.

AES. Advanced Encryption Standard.

ANSI. American National Standards Institute.

ARINC. Aeronautical Radio, Incorporated.

AUTOSAR. AUTomotive Open System ARchitecture.

BSP. Board Support Package.

CBOM. Cryptography Bill of Materials.

CFI. Control-Flow Integrity.

CMSIS. Cortex Microcontroller Software Interface Standard.

CPU. Central Processing Unit.

CRC. Cyclic Redundancy Check.

CSR. Control and Status Register.

DMA. Direct Memory Access.

DMB. Data Memory Barrier.

DRAM. Dynamic Random-Access Memory.

DSB. Data Synchronization Barrier.

DSP. Digital Signal Processor.

DWARF. Debugging With Attributed Record Formats.

EDF. Earliest Deadline First.

ELF. Executable and Linkable Format.

EVA. Value analysis by static analysis.

FIFO. First In, First Out.

FPU. Floating-Point Unit.

GPIO. General-Purpose Input/Output.

GPU. Graphics Processing Unit.

HAL. Hardware Abstraction Layer.

HSE. Hardware Security Engine.

IPI. Inter-Processor Interrupt.

ISA. Instruction Set Architecture.

ISB. Instruction Synchronization Barrier.

ISR. Interrupt Service Routine.

JOP. Jump-Oriented Programming.

LTO. Link-Time Optimization.

MC/DC. Modified Condition/Decision Coverage.

MMIO. Memory-Mapped Input/Output.

MPU. Memory Protection Unit.

NoC. Network on Chip.

NVM. Non-Volatile Memory.

PGC32. 32-bit counter-based pseudo-random generator.

POSIX. Portable Operating System Interface.

PSA. Platform Security Architecture.

RMA. Rate Monotonic Analysis.

ROP. Return-Oriented Programming.

RTE. Run-Time Error.

SBOM. Software Bill of Materials.

SIMD. Single Instruction, Multiple Data.

SMMU. System Memory Management Unit.

SMP. Symmetric Multi-Processing.

SoC. System on Chip.

SPE. Signal Processing Engine.

SSP. Stack Smashing Protector.

SSTIC. Symposium sur la sécurité des technologies de l'information et des communications (French security research conference).

TDD. Test-Driven Development.

TDM. Time-Division Multiplexing.

TCM. Tightly Coupled Memory.

TRNG. True Random Number Generator.

UART. Universal Asynchronous Receiver-Transmitter.

UB. Undefined Behavior.

USB. Universal Serial Bus.

XIP. Execute In Place.

2
Introduction and scope

2.1 Purpose of the guide

This guide provides a complete methodology for implementing embedded microcontroller code in C so as to ensure sufficient quality metrics in critical contexts. The purpose of this document is therefore to ensure the following essential elements:

- Portability and testability of the embedded code and its modules
- Longevity and maintainability of the code, allowing the maintenance and evolution cost associated with the project's lifetime to be reduced
- Proof of absence of run-time errors (noRTE) across the whole codebase
- Efficient and safe delivery of the essential information about the software, including its SBOM (Software Bill of Materials), its production information, or its license
- Performance in terms of footprint and consumption
- Resilience against side-channel, fault-injection, and supply-chain attacks

This guide focuses exclusively on the software development and production stage. It therefore covers configuration management, production, and delivery of the software, but does not address project management or integration into a complete system including hardware and software.

This guide focuses on the C language in a constrained embedded context, as this corresponds to the vast majority of industrial state of the art. This guide does not aim to describe how to architect an embedded operating system or produce an SDK, but focuses specifically on proper use of the C language and the typical cases of poor implementation that make the objectives listed above complex or infeasible.

2.2 Intended audience

The guide is intended for:

- Embedded software development teams

- Work-package leads and software architects
- Technical auditors and peer reviewers of the developments

2.3 Technical scope and assumptions

The scope covers constrained microcontroller-class embedded systems with one or more execution levels. The guide applies both to purely bare-metal developments (of the *super-loop* kind, for example) and to embedded micro-kernel developments or the associated critical applications.

The guide assumes that the target software is designed with reuse in mind, in order to ensure maximum portability and maintainability. Nevertheless, it applies just as well to an implementation dedicated to a specific need.

Note that this guide aims to provide the keys for critical implementations, requiring formal proofs, medium- to high-level qualification, and strict technical compliance.

Key takeaway

The guide follows a single thread: every important property must be expressed by a contract, isolated in the right layer, verified through analysis or testing, and then linked to the artifact actually delivered.

3

The C language: reminders and evolutions

This chapter is a reminder about the C language, its evolutions, and the notions essential to ensuring the portability and maintainability of embedded code.

3.1 History and standardization of C

C was born in the early 1970s at Bell Labs, in the context of rewriting Unix on the PDP-11. Kernighan and Ritchie's description, commonly called K&R C, long served as the de facto reference [27]. It describes a language very close to the machine, well suited to constrained systems, but whose implementations could diverge on the size of types, calling conventions, the preprocessor, or the libraries. K&R code remains frequent in old BSPs (Board Support Packages) and drivers; it is not, however, an acceptable basis for new development, because old-style function declarations carry no prototype and let the compiler apply implicit promotions that may be incompatible with the ABI (Application Binary Interface).

The first normative baseline is ANSI X3.159-1989, quickly adopted as ISO/IEC 9899:1990. The names C89 and C90 refer, for the developer, to the same base language [2, 22]. This standardization notably fixed prototypes, standard library headers, the `const` and `volatile` qualifiers, as well as the distinction between defined, unspecified, implementation-defined, and undefined behavior.

For embedded systems, the essential point is that the standard fixes neither the size of `int`, nor the endianness, nor the representation of pointers, nor the ABI. These properties belong to the target and the compiler; they must be expressed in the architecture layer and never assumed by application code.

C99 is the most visible evolution for portable firmware [23]. The `<stdint.h>` and `<inttypes.h>` headers give access to exact-width, minimum-width, and maximum-width types, as well as the corresponding formatting macros. They avoid confusing a protocol size with `int` or `long`, whose width varies between ARM, RISC-V, and PowerPC. The `<stdbool.h>` header makes boolean intent explicit. Designated initializers help build vector tables, registers, or versioned structures

safely; `inline` allows a short primitive to be expressed without forcing a function call.

```c
#include <stdint.h>

struct uart_registers {
  volatile uint32_t control;
  volatile uint32_t status;
};

static const struct uart_registers reset_values = {
  .control = 0U,
  .status = 0U,
};
```

Listing 3.1: Portable initialization of UART registers

It is important to note that although the ISO booleans provided by `<stdbool.h>` help the implementation, they do not guarantee ABI portability. An ISO boolean may be represented by a byte, a word, or a double word depending on the target and the compiler.
Furthermore, conversion to a boolean type produces a logical value of zero or one, but the in-memory representation and the ABI remain specific to the target. Code that carries security states in a byte or a word must therefore explicitly define its encoding and must not confuse a non-zero value with a valid state.

Recent compilers may offer extensions for checking or hardening booleans, but these remain specific to the toolchain used and are not standardized by ISO C. It is therefore recommended to define critical states with explicit types and contracts, for example enumerations whose valid values are checked with a sufficient Hamming distance, according to the fault model adopted, in case of a documented risk of physical fault (EM or glitch) in the project's context [20].

Some evolutions, such as the `restrict` qualifier, are risky. This keyword is indeed a promise of absence of aliasing made to the compiler by the developer: violating it produces undefined behavior and can turn a correct copy routine into incorrect code after optimization. Furthermore, `inline` does not guarantee inlining and has a subtle linkage semantics in C99: a public `inline` function requires an explicit strategy for its external definition. Variable-length arrays, introduced by C99, can consume a stack amount bounded by an untrusted input; they must be forbidden in critical production code. Finally, compound literals and variadic macros are useful, but must not hide an object's lifetime, multiple evaluation, or a side effect.

C11 brings the decisive primitives for concurrent systems [24]. `<stdatomic.h>` and the `_Atomic` keyword define atomic operations and memory orders. Note, however, that the `_Atomic` keyword applies to the associated variable, not to the operation itself.
`_Static_assert` allows an incompatible configuration to be rejected as early as compile time. This addition is particularly useful in embedded systems, where a large amount of information can be computed at compile time. It thus makes it possible to demonstrate the conformance of a configuration to the ABI, to size and alignment constraints, or to the availability of certain resources.
`_Alignas` and `_Alignof` express alignment constraints important for DMA (Direct Memory Access), registers, and certain ABIs. Their addition to the ISO standard avoids the use of extensions (such as GNU extensions), simplifying software portability across heterogeneous toolchains (GCC, ARM, IAR, etc.).

Finally, the `_Thread_local` and `_Noreturn` keywords, together with the `<threads.h>`,

`<stdalign.h>`, `<stdnoreturn.h>`, and `<uchar.h>` headers, complete this evolution. In a bare-metal firmware, `<threads.h>` is often absent or unsuited: its availability must never be assumed. The same holds, barring a specific exception, for `<uchar.h>`, which should be avoided in embedded firmware, as it provides no portability guarantee regarding the ABI and character representation.

ISO C11 also provides the notion of `_Generic`, which allows an expression to be selected according to its type at compile time. Effective as it is, this notion must nevertheless be used sparingly.

```c
#include <stdatomic.h>

_Static_assert(sizeof(uint32_t) == 4U, "uint32_t must be 32 bits");
static atomic_uint_least32_t pending_events;

void signal_event(uint32_t event)
{
  (void)atomic_fetch_or_explicit(&pending_events, event, memory_order_release);
}
```

Listing 3.2: Atomic signaling of an event

A C11 atomic does not magically make a protocol correct. `volatile` does not replace `_Atomic`; a `memory_order_relaxed` order does not publish the associated data; and an atomic operation is not necessarily lock-free. Before a call from an ISR (Interrupt Service Routine), check the `atomic_is_lock_free` property on the target and study the generated code. C11 primitives also do not replace the architecture barriers and cache maintenance required by a DMA or a non-coherent interconnect.

C17 is a corrective revision of C11: it does not modify the firmware programming model, but clarifies and corrects the text of the standard [25]. Its interest is therefore mainly contractual: a toolchain qualified for C17 keeps the gains of C11 without imposing a functional migration.

C23 modernizes several aspects useful to embedded systems [26]. It notably standardizes `nullptr`, standardized attributes such as `[[nodiscard]]` and `[[maybe_unused]]`, `_BitInt`, binary inclusion via `#embed`, and `<stdckdint.h>` for checked integer arithmetic.
These additions can improve the expressiveness of an interface, the checking of error returns, the inclusion of binary resources, or overflow prevention. Their use remains conditioned on the maturity of the compiler, of static analysis, and of the associated qualification.
Choosing a C edition is therefore a configuration decision: the project fixes its allowed profile, forbids unjustified extensions, and verifies that every target actually provides the headers and semantics used.

3.2 The C abstract machine concept

A C program is defined by an abstract machine, not by a sequence of processor instructions. The compiler is free to transform the implementation according to the *as-if* principle: it may produce any code whose observable behavior is the same as that of the C program with defined behavior [24, 26]. It can thus propagate a constant, remove a computation whose result is never used, keep a value in a register, merge accesses, or move a write, as long as these transformations do not change the observation authorized by the standard.

Input/output and accesses to an object qualified `volatile` are part of this observable behavior. A read of an MMIO (Memory-Mapped Input/Output) register must therefore be declared `volatile` so that the compiler performs the requested access on every evaluation of the C code. Without this qualifier, it may cache the first value read from a register, remove a read whose result seems unused, or hoist that read out of a loop. `volatile` is a contract with the compiler, not with the hardware: it provides neither atomicity, nor mutual exclusion, nor a memory barrier for other cores, nor cache maintenance for the DMA. These properties are the responsibility of C11 atomics, of the architecture's barrier instructions, and of the device access protocol.

```c
#include <stdint.h>

#define UART_STATUS_ADDRESS ((uintptr_t)0x40001004U)

static uint32_t transmitted_bytes;

static uint32_t uart_status(void)
{
  return *(volatile const uint32_t *)UART_STATUS_ADDRESS;
}

void uart_account_transfer(uint32_t length)
{
  transmitted_bytes += length;
  while ((uart_status() & 1U) == 0U) {
    /* Each iteration re-reads the hardware register. */
  }
}
```

Listing 3.3: Difference between internal state and a hardware register

The `static` keyword does not make an object observable and does not protect it against concurrency. At file scope, it gives internal linkage: the name is not exported and the compiler knows that no other conforming translation unit can directly reference `transmitted_bytes` or `uart_status`.
At block scope, it gives static storage duration: the object is unique and exists for the entire execution, but its name remains limited to the block. In both cases, `static` means neither immutable nor thread-local; a mutable `static` object shared between a thread, an ISR, or a core requires explicit synchronization.

The translation unit is the result of preprocessing a source file and its inclusions. When compiling it, the optimizer knows the bodies of the functions and the internally linked objects of that unit. It can therefore remove an unused `static` function, inline `uart_status` into its caller, keep `transmitted_bytes` in a register between two observable points, or notice that a `static const` function is pure. Conversely, a function declared in another compilation object is, without LTO, an information boundary: the compiler must respect its ABI and assume the side effects declared or possible through its interface.

This limit is not a security barrier. Linking, and possibly LTO, can then analyze several translation units and extend interprocedural optimizations. Embedded development must therefore demonstrate correctness at the level of well-defined C, then check the binary for the properties that depend on the target: MMIO access of the required width, barriers, boot sequence, section placement, and ABI. Relying on the disassembly of an optimized build never justifies a

program that contains undefined behavior; a compiler or option update may then legally change the generated code.

3.3 Compilation units, scoping, and the notion of a thread model

Separate compilation treats each source file, after expanding its headers, as a translation unit. The compiler produces an object containing code, data, defined symbols, and symbols to be resolved. The linker then assembles these objects according to their name, their linkage, and their ABI. This organization limits recompilation, but does not guarantee that two external declarations describe the same type, the same alignment, or the same calling convention. A public header is therefore a contract: it is included by the owner of the implementation and by all its callers, then checked by the toolchain [24].

Scope defines where a name is visible in the source text. A local variable has block scope; a file-level declaration has file scope. Linkage, on the other hand, defines whether the same name can refer to the entity from another translation unit. A function or object declared without `static` at file scope generally has external linkage. The `static` prefix gives internal linkage to a file-scope function or object: its name is not exported at link time. It must not be confused with static storage duration, which keeps an object alive for the entire execution, in particular for a local `static` variable.

Embedded code favors `static` functions for a module's internal details and public functions for its interface. This reduces the global namespace, prevents direct calls to internal primitives, and gives the optimizer full visibility of their possible callers within the translation unit. Mutable global objects with external linkage should be avoided: they hide a dependency, prevent knowing their owner, and complicate testing, review, and the proof of absence of interference.

```c
/* telemetry.h */
struct telemetry;
void telemetry_record(struct telemetry *context, uint32_t event);

/* telemetry.c */
struct telemetry {
  uint32_t event_count;
};

static void telemetry_increment(struct telemetry *context)
{
  context->event_count++;
}

void telemetry_record(struct telemetry *context, uint32_t event)
{
  (void)event;
  telemetry_increment(context);
}
```

Listing 3.4: Module interface without an exported mutable object

`extern` is legitimate for declaring an entity defined elsewhere; function prototypes in a header indeed have external linkage by default. On the other hand, a header should almost never declare a mutable object `extern`. This practice gives any caller read and write access without a protocol. An accessor function, a context structure passed explicitly, or a message to the owner better express authority and the concurrency rules. When a shared object is indispensable, a single translation unit defines it, the others declare it, and its invariant, its initialization, and its synchronization are documented.

Since C11, the thread model defines abstract threads of execution, atomic objects, and *happens-before* relations. Two threads that access the same non-atomic object, of which at least one writes, form a data race if they are not synchronized, and the behavior is then undefined. An object with static storage duration is shared by default between threads. `_Thread_local`, on the contrary, gives one instance per thread, but its support and its cost must be validated on the target. ISRs, DMA, and other cores are not all modeled directly by `<threads.h>`, but firmware nonetheless treats them as concurrent contexts and defines the necessary atomics, locks, barriers, and hardware protocols.

3.4 Cyclomatic number and the notion of coverage

McCabe's cyclomatic number measures the number of linearly independent paths in the control graph of a function. For a connected graph, it equals $M = E - N + 2$, where E is the number of edges and N the number of nodes. In a structured C function, a practical approximation is to start from 1 and add 1 for each decision: `if`, `while`, `for`, `case`, a ternary condition and, depending on the tool's convention, each short-circuit boolean operator. The convention adopted must be documented, as it changes the reported value [30].

```
int validate_frame(const uint8_t *frame, size_t length)
{
  if (frame == NULL || length < 2U) {
    return -1;
  }
  if (frame[0] != 0xA5U) {
    return -2;
  }
  return (frame[1] <= 32U) ? 0 : -3;
}
```

Listing 3.5: Cyclomatic complexity and independent decisions

This example contains three decisions if `frame == NULL || length < 2U` is considered a single decision, giving a complexity of 4. If the analysis counts both operands of `||` separately, it obtains 5. This value is not proof of a defect; it indicates the minimum number of test cases to design in order to exercise independent decisions. A high value mainly signals a function that is difficult to read, modify, and test. In critical code, splitting a function by responsibility is generally preferable to piling up exceptions and nested conditions.

Coverage measures what a test campaign has actually executed. Statement coverage confirms that every statement was executed at least once. Branch coverage confirms that every outcome

of a decision was exercised. Condition coverage checks each elementary condition; MC/DC further demonstrates that each condition can independently change the decision. Exhaustive path coverage quickly becomes unrealistic as soon as loops, errors, and variable inputs are present. For a critical function, selecting bounded paths and justifying unreachable paths is more useful than claiming overall coverage without analysis of the results.

In the previous example, the code, fairly basic, allows itself several exit points. Usually, for security and testability reasons, each function should have only a single exit point.
It is interesting to note that in the particular case of error handling within a function, the use of `goto` can facilitate coverage and reduce cyclomatic complexity. It also increases readability while ensuring the single-exit-point principle.

A single exit point concentrates the actions that must be executed regardless of the function's outcome: restoring an invariant, releasing resources, exiting a critical section, wiping a sensitive buffer, or logging the result. Test cases then verify a single epilogue, instead of having to prove that each intermediate `return` executed the right subset of cleanups. This rule must not be turned into a dogma: an early return is acceptable for a precondition without a side effect. However, as soon as a function acquires several resources, scattered exits are a frequent source of forgotten releases.

`goto err` is appropriate when it only jumps to cleanup labels located at the end of the function. Each label releases an already-acquired resource, in the reverse order of its acquisition, then joins the common epilogue. This pattern avoids nested conditions and code duplication; it must not be used to jump into the heart of the algorithm nor to bypass the initialization of a variable.

```
1  int transmit_frame(const uint8_t *frame, size_t length)
2  {
3    int status = -1;
4    struct dma_buffer *buffer = dma_buffer_acquire(length);
5
6    if (buffer == NULL) {
7      goto err;
8    }
9    if (dma_buffer_copy(buffer, frame, length) != 0) {
10     goto err_release_buffer;
11   }
12   if (dma_start_transfer(buffer) != 0) {
13     goto err_release_buffer;
14   }
15
16   status = 0;
17
18 err_release_buffer:
19   dma_buffer_release(buffer);
20 err:
21   return status;
22 }
```

Listing 3.6: Centralized cleanup with goto err

In this example, all paths join a single return. The `err_release_buffer` label precisely documents the state reached and guarantees that the resource is released once, including after a copy or DMA-start error. Tests must nevertheless cover each jump and verify that the release functions are safe after the errors provided for by their contract.

Without `goto`, such a function, which must also ensure a single exit point, would look like

the code described in the following example, with nested conditions and duplicated release code. The cyclomatic complexity is higher, readability lower, and coverage harder to achieve.
In that case, the cyclomatic number goes from 4 to 6.

Key takeaway

Cyclomatic complexity guides test design, but does not constitute a proof of correctness. Error paths, acquired resources, and the cleanup epilogue must be tested explicitly.

```c
int transmit_frame(const uint8_t *frame, size_t length)
{
  int status = -1;
  struct dma_buffer *buffer = dma_buffer_acquire(length);

  if (buffer != NULL) {
    if (dma_buffer_copy(buffer, frame, length) == 0) {
      if (dma_start_transfer(buffer) == 0) {
        status = 0;
      }
    }
    dma_buffer_release(buffer);
  }
  return status;
}
```

Listing 3.7: Centralized cleanup without goto

`gcov` collects coverage from a binary instrumented by GCC; `gcovr` aggregates this data and produces reports usable in continuous integration. These results are valid only for the binary, the compilation options, the target, and the scenarios actually executed. For a target without a filesystem, the counters can be retrieved via serial link, semihosting, or reserved memory; this collection mechanism must not alter the test's real-time behavior to the point of invalidating its results.

Static analysis is complementary and not a coverage measurement. Tools such as CodeSonar look for data-flow defects and dangerous behaviors; SonarQube notably centralizes rules, alerts, and quality indicators. Alerts must be triaged, justified, or fixed, not merely counted. Finally, `gprof` is a profiling tool: it estimates the cost of functions and their calls, but proves neither coverage nor absence of error. Coverage counters, static analysis, and profiling therefore answer three distinct questions: "what was executed?", "what defects are possible?", and "where is time spent?".

3.5 Portability in C

C portability does not mean that the same binary runs on every target. It means that the same source code, combined with a documented adaptation layer, keeps the same semantics when recompiled. ISO C fixes the minimum bounds of the types and the language's operations, but leaves the implementation to choose the size, alignment, in-memory representation, endianness, and ABI [24, 26]. A portable project therefore separates target-independent code from the contracts specific to the processor, the compiler, the linker script, and the peripherals.

The types `char`, `short`, `int`, `long`, and `long long` must not carry a persistent protocol, register, or storage width. For instance, `int` can be 16, 32, or 64 bits wide; `long` is commonly

32 bits in some ARM and PowerPC ABIs, but 64 bits in a RISC-V LP64 ABI. A function's signature, a structure's layout, and an overflow computation then change if these types are used as substitutes for precise sizes. Likewise, `char` is always the addressable unit of C, but its associated signedness is not defined by the standard: it may be signed or unsigned depending on the implementation. Portable code must never assume that `char` is signed or unsigned.

Since C99, `<stdint.h>` provides the types suited to numeric contracts. `uint32_t` and `int32_t` express an exact width of 32 bits and are defined only if the target can provide them without padding bits. `uint_least32_t` expresses at least 32 bits; `uint_fast32_t` favors speed, not storage format. Use `size_t` for an object's size or index, `ptrdiff_t` for a pointer difference, and `uintptr_t` only when an address must be kept as an integer. A pointer-to-integer conversion does not make an address valid on another target and must never be used to bypass aliasing rules. The `size_t` type is the only integer guaranteed to hold the size of an object; it must not be used for negative values.

The `<inttypes.h>` header provides the `PRIu32`, `PRIxPTR`, and similar macros to format these types without assuming anything about `printf`. The `%d`, `%ld`, or `%x` formats applied to an exact-width type can cause undefined behavior when the variadic argument does not match the expected format. In firmware, the same rules apply to binary logs and diagnostic interfaces, even when they do not use the standard library.

It is important to note that the use of the `<stdio.h>` family of functions is generally to be avoided in embedded firmware. These functions are indeed very resource-hungry and not suited to embedded use. It is preferable to use lighter formatting functions, or to implement simpler formatting functions specific to the embedded firmware's use case.
Furthermore, variadic-argument functions are to be avoided in embedded firmware, as they do not allow easy verification of the correspondence between the format and the arguments passed. Compilers such as GCC allow format-checking attributes to be added, but this does not guarantee the safe use of these functions.
Finally, variadic functions are problematic to prove in terms of noRTE.

```c
#include <limits.h>
#include <stdint.h>

_Static_assert(CHAR_BIT == 8, "The protocol uses eight-bit bytes");

static uint32_t read_be32(const uint8_t input[4])
{
  return ((uint32_t)input[0] << 24) |
         ((uint32_t)input[1] << 16) |
         ((uint32_t)input[2] << 8) |
         (uint32_t)input[3];
}
```

Listing 3.8: Portable decoding of a 32-bit value in network format

This example builds the value from protocol bytes; it depends neither on the processor's endianness nor on the alignment of the input. Conversely, directly converting a `uint8_t *` pointer to a `uint32_t *` and then dereferencing it can violate alignment and strict aliasing, and can even trigger a processor exception.

Compile-time assertions form the last level of the porting contract. They must verify the width and alignment required by an interface, while the BSP checks dynamic properties at startup: memory map, presence of a device, or cache coherence. MMIO accesses remain confined to primitives of explicit width and to types defined by the component's manufacturer. This

separation allows business code to be shared between ARM, PowerPC, and RISC-V without hiding the actual differences in their ABIs and their hardware.

4

Undefined Behavior and Run Time Error

4.1 Reminders and definitions

Undefined behavior (UB) refers to a case for which ISO C imposes no requirement on the implementation [24]. As soon as the program reaches it, the compiler may produce any behavior, including code that appears correct for a given version and then becomes incorrect after optimization. A signed integer overflow, an out-of-bounds access, a division by zero, an invalid shift, dereferencing an invalid pointer, or a data race are common UBs in firmware. Some UBs are detected by the processor and can trigger an exception, such as division by zero.

A *run-time error* (RTE, Run-Time Error) is an error that may occur at execution time and that analysis or testing seeks to exclude within the project's scope. An unsigned conversion is defined by ISO C, but can cause a loss of information or a violation of an application contract; it then becomes an RTE to be handled. RTEs and UBs are therefore two distinct categories, even though an RTE can lead to a UB if the contract is not controlled.

Furthermore, implementation-defined behaviors are not necessarily errors, but they must be inventoried. The signedness of `char`, the size of fundamental types, endianness, alignment, pointer representation, and the calling convention depend on the target, the compiler, or the ABI. An embedded project fixes them in its configuration, checks them with assertions, and isolates them in the BSP. An optimization option, a compiler version, or a GNU extension must never be considered an ISO C property.

Key takeaway

Undefined behavior is never made acceptable by a binary that appears to work. C code must remain well-defined; disassembly, the ELF, and the link map are then used to verify the properties specific to the target.

4.2 The memory model and portability

The C memory model defines objects, their lifetime, their representation, and the access rules; it does not describe the microcontroller's physical memory map. An ordinary C address only designates an object if the object exists, is correctly aligned, and is accessed with a compatible type. The C11 thread model adds atomic operations and ordering relations between threads. It replaces neither ISA barriers, nor cache coherence, nor DMA rules.
It is critical to understand well that compilers, regardless of the language, only handle memory protection from the point of view of the executable, based on the thread model associated with the language when one exists. They **cannot** guarantee cache coherence, the visibility of a write toward a device such as DMA, the variation in value of a hardware register between two reads, etc.

A port between ARM, PowerPC, and RISC-V must therefore separate three levels. The C code expresses the algorithm and its preconditions; the architecture layer translates atomics and barriers into suitable instructions; the BSP describes memory attributes, cache, DMA, and MMIO. `volatile` imposes an observable access on the compiler, but does not publish data between cores and does not make an update atomic. Using it for a hardware register does not authorize its use as a lock or as a substitute for `_Atomic`.

Portability also requires not deducing binary properties from the C source alone. An unaligned read may succeed on one target, be split into several accesses on another, or raise an exception. Likewise, the visibility of a write toward a DMA area depends on the cache attributes and on explicit maintenance. Interface contracts therefore indicate the calling context, the alignment, the ownership of buffers, and the barriers required before any local optimization.

Key takeaway

`volatile` makes an access observable by the compiler, but provides neither atomicity, nor a lock, nor cache coherence. MMIO and DMA accesses require the appropriate hardware protocol, barriers, and cache maintenance.

4.3 Memory for C

The C memory model is founded on objects and their type. A memory area is not an undifferentiated sequence of bytes: accessing it must respect the object's lifetime, its alignment, and the compatible-type rules, often called *strict aliasing*. Accessing a `uint32_t` object through a `uint16_t *` pointer, outside cases provided for by the standard, can therefore be UB even if the processor physically accepts the read. The compiler is then allowed to assume that the two pointers do not designate the same object.

Conversion between a pointer to bytes and a pointer to an integer is a frequent mistake in parsers and drivers. It can accumulate three defects:

- insufficient alignment
- invalid aliasing
- implicit endianness

Reading the bytes one by one, or copying into a correctly aligned object and then explicitly converting the endianness, keeps a well-defined semantics. Copy functions must themselves respect the validity and the non-overlap of their areas, unless the contract is of the `memmove` kind.

```
static uint32_t load_u32_unaligned(const uint8_t input[4])
{
  return ((uint32_t)input[0]) |
         ((uint32_t)input[1] << 8) |
         ((uint32_t)input[2] << 16) |
         ((uint32_t)input[3] << 24);
}
```

Listing 4.1: Safe read of an unaligned 32-bit integer

The example explicitly assumes a similar encoding between the data and the endianness of the processor core on which the algorithm runs, which may be false if the data comes from the outside. On data internal to the executable, this algorithm is safe: it never dereferences an invalid pointer, does not read out of bounds, and does not violate strict aliasing.
Furthermore, it is also portable: it avoids the potentially misaligned dereference of `(const uint32_t *)input`. RTE and EVA analyzers can verify the bounds, the shifts, and the validity of the pointers if the length and validity preconditions are expressed in the calls or in the ACSL contracts.

4.4 Notion of data encoding

Any external data is associated with an encoding, not a native C representation. A protocol must define width, byte order, bit order if necessary, allowed values, alignment, and length. The `htons`, `ntohs`, `htonl`, and `ntohl` conversions transform between host order and big-endian network order. They must be applied at the system's boundaries, never scattered throughout business code. This makes internal structures independent of the endianness of the targeted ARM, PowerPC, or RISC-V.

Modern compilers often provide `__builtin_bswap16` and `__builtin_bswap32`. When the target has a dedicated instruction, the compiler generally selects it; otherwise it optimizes the masks and shifts. Testing for the builtin must remain isolated in a compatibility layer. A pure-C fallback is required for compilers that do not provide it and must use unsigned integers to avoid signed shifts or overflows. The Camelot-OS Shield implementation follows this principle for its POSIX conversions [11].

```
#include <arpa/inet.h>

#define BSWAP32(__bsx) \
    ((((__bsx) & 0xff000000u) >> 24) | \
    (((__bsx) & 0x00ff0000u) >> 8)   | \
    (((__bsx) & 0x0000ff00u) << 8)   | \
    (((__bsx) & 0x000000ffu) << 24))

#define BSWAP16(__bs) \
    ((((__bs) & 0xff00u) >> 8) | (((__bs) & 0x00ff) << 8))


uint32_t htonl(uint32_t x)
{
```

```
15 #if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
16 #if defined(__has_builtin) && __has_builtin(__builtin_bswap32)
17     return __builtin_bswap32(x);
18 #else
19     return BSWAP32(x);
20 #endif/*! has builtin */
21 #else
22     return x;
23 #endif
24 }
```

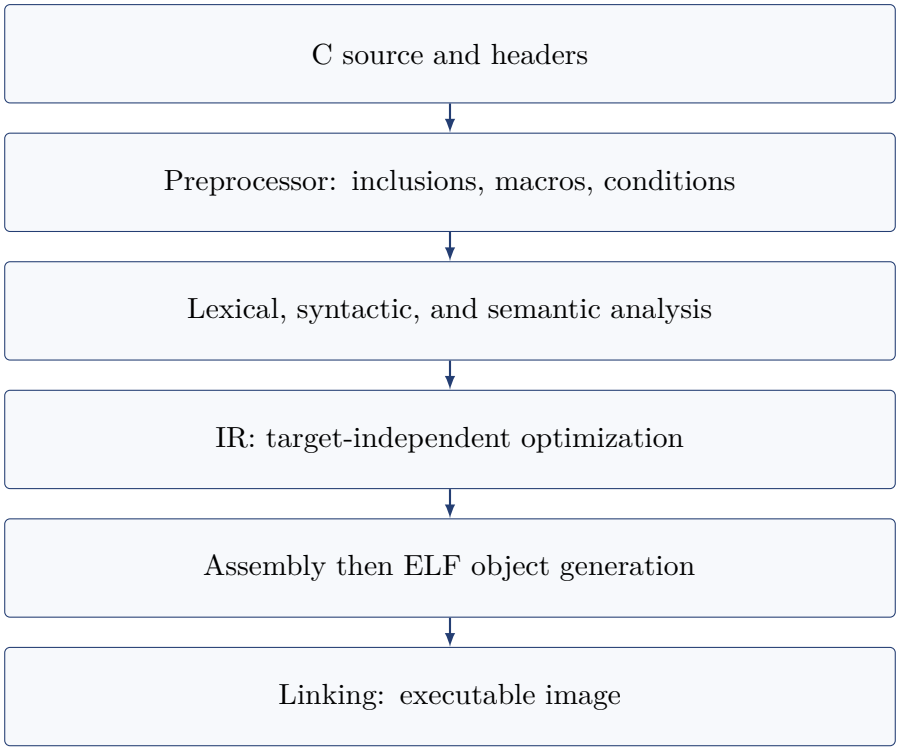
Listing 4.2: Host-to-network 32-bit conversion with a portable fallback

The endianness of a platform is normally recognizable in portable C code. Although ISO C does not define endianness, compiler macros or the validated BSP configuration allow it to be determined. The Camelot-OS Shield implementation uses the `PLATFORM_LITTLE_ENDIAN` macro, which is defined in the BSP and checked by compile-time assertions.
GCC and Clang define `__BYTE_ORDER__` and `__ORDER_LITTLE_ENDIAN__` to identify endianness. Application code must not try to deduce it at runtime.
In this example, endianness is based on these macros.
The builtin does not change the semantics: it merely offers an efficient implementation of the same byte swap. For a 16-bit format, apply the same contract with `__builtin_bswap16` and a two-byte fallback.

5
From C to binary

5.1 Reminder about compilers

The compiler does not translate C line by line. It builds a representation of the program, applies to it the transformations authorized by the language, and then produces an object conforming to the target's ABI. In firmware, the configuration of this toolchain is therefore part of the product: compiler version, flags, headers, linker script, libraries, and optimization options must be archived together with the binary. The same source can produce different behaviors if the ABI, the enabled extensions, or the documented properties of the processor change.



The preprocessor first normalizes the translation unit: it substitutes macros, resolves inclusions, and eliminates conditional branches. It knows neither the types nor the scope of identifiers; a macro therefore does not constitute a safe function. The front-end then analyzes tokens, declarations, types, and the language's constraints. It produces an intermediate representation

(IR), on which the compiler propagates constants, eliminates dead code, simplifies branches, and reorders operations when the *as-if* principle allows it. Type errors, UBs, and `volatile` accesses determine the limits of these transformations.
Based on the notion of *as-if*, the compiler can reorganize or remove operations as long as their observable effects are preserved. It can in particular eliminate a call, a branch, or a local variable with no remaining observable effect, including in a `static` function known within the translation unit.
It is important to understand this principle well, as it is the origin of the *optimized-out* and *anonymous* markers appearing in debuggers, which makes analysis more complex.

The code generator lowers the IR to the targeted ISA and respects its ABI. On entry to an ordinary function, the prologue reserves the stack, saves the registers that the ABI requires to be preserved, and establishes a frame pointer if needed. The epilogue restores this state and then returns to the caller. The details vary between AAPCS, the RISC-V psABI, and PowerPC EABI, but the caller and the callee must always share the same convention, called the *calling convention*.
By default, all C functions are compiled with a prologue and an epilogue.

```
/* Indicative example of a Cortex-M prologue and epilogue. */
example:
  push {r4, lr}
  sub  sp, sp, #16
  /* Function body. */
  add  sp, sp, #16
  pop  {r4, pc}
```

A function qualified `naked` removes all or part of this prologue and epilogue. It is reserved for boot code, vectors, or context switches whose assembly is entirely under control. Its body must not contain ordinary C: the compiler may use the stack or registers without the usual frame existing. This extension is specific to the toolchain and must be isolated in the architecture layer. These functions generally contain inline assembly instructions, which are not portable and must not be used in business code.

Inlining replaces a call with the function's body. It can lower the call cost and give the optimizer more information, but sometimes increases binary size and register pressure. The C `inline` keyword does not constitute an obligation to inline. Attributes such as `always_inline` or `forceinline` are extensions to be used only after measuring the binary and the execution time; they must not compensate for an inadequate interface or module decomposition.
In general, the compiler decides on inlining based on the size of the function, the number of calls, and the optimization options. Thus, the `-O0` and `-O1` options favor the readability of the generated code, while `-O2` and `-O3` favor inlining and execution speed. The `-Os` option favors binary size, which can be critical in embedded firmware.
The compiler, depending on the optimizations enabled, can also unroll loops, vectorize operations, and reorder instructions.

```
static inline uint32_t set_bit(uint32_t value, uint32_t bit)
{
  return value | (UINT32_C(1) << bit);
```

```
}
```

Listing 5.1: C primitive candidate for inlining, with no optimization obligation

The `pure` and `const` attributes are promises made to the compiler. In GCC and Clang terminology, a `pure` function produces no observable effect and its result may depend on readable memory; a `const` function is even more restrictive. Declaring them wrongly allows removing or merging a call that had to be executed. A function reading MMIO, a counter, a clock, shared state, or a `volatile` object must never receive these attributes.

At link time, a strong symbol defined once provides the normal implementation. A weak symbol (`weak`) can be replaced by a strong definition, which facilitates a BSP adaptation but also hides configuration errors. Limiting weak symbols to explicitly documented extension points and checking the link map prevents a default implementation from being shipped inadvertently. Symbol aliases are subject to the same caution: they change resolution at link time, not the C language's pointer aliasing rules.

Finally, the optimizer assumes that two pointers of incompatible types do not designate the same object, and that the `restrict` contract, when present, is respected. This aliasing assumption notably authorizes moving or vectorizing reads and writes. It must not be invalidated by a pointer cast, an MMIO area, or a buffer shared with DMA. C code must first remain well-defined: disassembly and link analysis then serve to verify target-dependent properties, never to justify an incorrect source program.

5.2 Optimizing memory footprint

Memory footprint is measured on the actually delivered image, not on the sum of the object files. The link map (`.map file`) indicates the size and address of each section, as well as the symbols that keep them alive. It is therefore the artifact to compare between versions to justify an increase in Flash, RAM, or stack. A useful optimization respects the memory budget without degrading real-time constraints, readability, diagnostic capability, or protection mechanisms.
It is worth noting that, in embedded systems, the compiler can provide the maximum stack consumption for each function, as well as the total stack consumption for the whole program. This information is very useful for correctly sizing the stack and avoiding stack overflows, which are critical errors in embedded firmware.
The `-fstack-usage` and `-fstack-clash-protection` options generate information about stack usage and detect stack overflows at runtime.

The `-ffunction-sections` and `-fdata-sections` options place each function and each data object in a separate section. Combined with `-Wl,-gc-sections`, they allow the linker to eliminate sections unreachable from the retained entry points. This technique often substantially reduces a modular firmware, but requires explicitly marking as used the vector tables, indirect callbacks, initialization sections, and objects referenced only by the linker script. Check the link map after every activation: a silent elimination of a handler is a functional defect, not an optimization. The `__attribute__((used))` attribute is then used to mark the symbols that must be kept. This use is also required for entry points when they are not named according to a standard C entry-point symbol such as `main` or `_start`.

LTO (*Link-Time Optimization*) passes an intermediate representation instead of final machine objects. The compiler can then remove cross-module functions, propagate constants, and specialize calls at the program level. The gain can be significant, but translation-unit boundaries change, and diagnostics, inlining, and debugging symbols evolve as well. A critical target enables LTO in a reproducible configuration, checks the binary, the link map, and the test results, then requalifies this configuration like any other compiler change.

Footprint reduction starts with the data model: types of the required width, constant tables in `.rodata`, properly sized buffers, and no duplication. Avoid disabling stack protection or the archived debugging information to save a few bytes. Compression or elimination options are acceptable only if the delivery chain keeps the ELF, the symbols, and proof that the distributed image matches the controlled sources.

5.3 The linker script

The linker script turns the hardware memory map into a placement policy. It declares the regions, fixes the entry point, and associates sections with the virtual or physical addresses used at startup. It also imposes the alignments required by the ABI, the caches, the DMA, or the MPU, and publishes symbols consumed by the boot code. This file is a critical source: it is versioned, reviewed, and checked together with the link map, on the same footing as C and assembly.
This file also allows sections to be associated with physical sectors in non-volatile memory. Indeed, in general, flash memories present in microcontrollers are divided into physical sectors. These sectors are the smallest elements that can be erased in flash memory. It is therefore important to properly size the code and data sections in the linker script in order to avoid erasing sectors containing critical data during a firmware update.

In an XIP (*Execute In Place*) configuration, typical of microcontrollers without memory abstraction, `.text` and `.rodata` reside and execute in NVM. `.data` has an initial image in NVM, but its execution address is in RAM so that it can be modified. `.bss` is only reserved and zeroed in RAM. The `_sidata`, `_sdata`, `_edata`, `_sbss`, and `_ebss` symbols avoid hard-coding magic addresses in the startup code, and allow the boot code to perform the copy or zeroing of these areas based on addresses specified in the linker script.

```
ENTRY(Reset_Handler)

MEMORY {
  FLASH (rx)  : ORIGIN = 0x08000000, LENGTH = 512K
  RAM   (rwx) : ORIGIN = 0x20000000, LENGTH = 128K
}

SECTIONS {
  .text : {
    KEEP(*(.isr_vector))
    *(.text*)
    *(.rodata*)
  } > FLASH
  .data : AT(LOADADDR(.text) + SIZEOF(.text)) {
```

```
    _sdata = .;
    *(.data*)
    _edata = .;
  } > RAM
  _sidata = LOADADDR(.data);
  .bss (NOLOAD) : {
    _sbss = .;
    *(.bss*)
    *(COMMON)
    _ebss = .;
  } > RAM
}
```

```
extern uint8_t _sidata, _sdata, _edata, _sbss, _ebss;

static void crt_initialize_memory(void)
{
  uint8_t *source = &_sidata;
  uint8_t *destination;

  for (destination = &_sdata; destination < &_edata;) {
    *destination++ = *source++;
  }
  for (destination = &_sbss; destination < &_ebss;) {
    *destination++ = 0U;
  }
}
```

Listing 5.2: XIP initialization of RAM from linker-script symbols

This routine runs before any use of C objects with static storage duration whose initialization requires RAM or zeroing. The boot code also checks the stack, the vector table, and the protection regions before enabling interrupts. After every script change, check the region boundaries, the alignment, the entry symbols, and the read/write/execute attributes in the final ELF.

5.4 About binaries

A section is a logical organization of the ELF produced by the compiler and the linker: `.text` generally contains the code, `.rodata` the constants, `.data` the initialized data, and `.bss` the zero-initialized data. Each section receives attributes, notably read, write, execute, alignment, and address. A segment is a loading view grouping sections with common attributes and a common loading mode; it mainly interests the loader and the debugger. The exact names, the initialization sections, and the placement policy are the responsibility of the linker script and the target's ABI.

ELF is the main working format of an executable, alongside other formats such as Mach-O (specific to macOS) or PE (in the Windows world). It keeps headers, sections, segments, relocations, a symbol table, and possibly DWARF debugging information. `readelf`, `objdump`, `nm`, and the link map allow the entry point, the addresses, the exported symbols, and the absence of a writable-and-executable section to be verified. A debugger such as GDB uses the symbols

and DWARF to associate addresses, functions, variables, and source lines; optimizing them can nevertheless remove or move these visible elements.

IHEX represents textual records of address and data, suited to transfer toward programmers. A raw binary is a sequence of bytes with no header, no symbol, and no intrinsic address: the flashing tool must know its loading address. Neither of these two formats replaces the ELF in the production archive. The project delivers the format expected by the ROM or the programmer, but also archives the ELF, the link map, the version information, and their fingerprints.
Converting ELF to IHEX or raw binary is trivial, but must not be done by the compiler: it is performed by a post-processing tool, often `objcopy`, which is itself provided by the cross-compilation toolchain. The delivered firmware must be verifiable by comparing fingerprints against the archived binary.
Converting an ELF to a raw binary ready to be flashed is done with the same tool. Raw binaries do not contain base-address information, unlike the IHEX format. It is therefore necessary to know the base address of the raw binary in order to flash it correctly.

Key takeaway

The ELF, the link map, and the build configuration are reference artifacts. The raw binary or IHEX intended for the programmer must never be the only element archived.

6
Bare-metal firmware concepts

6.1 Reminder about the notion of a stack

The stack is a memory region used for return addresses, saved registers, temporary arguments, and local variables. The stack pointer must remain aligned according to the ABI at every call point. A *stack frame* is the portion of the stack associated with a call; it may contain a save frame, local variables, parameters overflowing the registers, and space reserved for downstream calls. Optimization can remove a frame, merge calls, or keep a value in a register: the stack observed in the disassembly must therefore not be deduced from a simplified C rule.

At reset, the processor does not yet have a complete C environment. On a Cortex-M, the first word of the vector table provides the initial value for the system stack (MSP) and the second provides the address of `Reset_Handler`. The stack must already be placed in accessible, correctly aligned RAM. The handler can then select the general-purpose PSP (Process Stack Pointer) for a thread context, while the MSP generally remains reserved for exceptions and privileged code.

On RISC-V, the reset vector and the boot code must explicitly initialize the `sp` register; the ISA does not provide an MSP/PSP pair equivalent to the Cortex-M. A system may reserve a per-hart stack, an exception stack, and a per-task stack, but this separation is a matter of software and privilege profile [39]. The trap handler must save the necessary registers before calling into C, then restore `mepc` and the other context elements before `mret`.

On a PowerPC MPC5xxx, the reset handler initializes the stack register according to the PowerPC ABI before calling a C function. Exceptions use a save format imposed by PowerPC's Book E profile; some registers are saved automatically or in a specific frame, while the others must be saved by the handler. A dedicated exception stack is preferable when the interrupted context may be corrupted or when the thread's stack is not accessible in the current mode.

6.1.1 Building and checking the stack at startup

The linker script reserves a stack region and publishes its limits. The boot code places the pointer at the top of this region, taking into account the growth direction and the ABI's alignment.

An MPU guard or an inaccessible region can detect an overflow, but it does not replace proper sizing. Maximum usage must be estimated from the deepest call, nested handlers, error paths, and possible coprocessor contexts.

```c
#include <stdint.h>

extern uint8_t _stack_bottom;
extern uint8_t _stack_top;

static void stack_initialize(void) __attribute__((naked))
{
  uintptr_t top = (uintptr_t)&_stack_top;
  uintptr_t bottom = (uintptr_t)&_stack_bottom;

  /* The ABI requires 8-byte alignment here. */
  top &= ~(uintptr_t)7U;
  if ((top <= bottom) || ((top - bottom) < 256U)) {
    for (;;) {
    }
  }

  __asm__ volatile ("mov sp, %0" : : "r"(top) : "memory");
}
```

Listing 6.1: Controlled initialization of a stack before C code

This example shows the contract, but the instruction writing `sp` is specific to the architecture and must be replaced by the sequence expected for the target. The minimum bound of 256 bytes is illustrative: the project must compute it based on its calls and its handlers. Startup also checks that the region is in usable RAM and that its placement does not collide with the data, the heap, DMA buffers, or the stacks of other cores.

Key takeaway

The stack is a bounded, architectural resource: its address, its alignment, its size, its guard, and its collisions with other regions must be verified before C code executes.

6.1.2 Context saving and interrupt stacks

When an interrupt occurs, the hardware or the handler saves the minimum necessary to resume the interrupted execution. The rest depends on the ISR's contract. An ISR that modifies a callee-saved register, a floating-point register, or a CSR must preserve it, unless the architecture defines automatic saving or the context is explicitly transferred. The order of saving and restoring must be symmetric and documented; an error in the restored value of `pc`, the status register, or the stack can return to an incorrect privilege level.

The Cortex-M automatically stacks a hardware frame during an exception, and can then add floating-point registers depending on the FPU configuration. The `EXC_RETURN` bit notably indicates the stack used to resume the context. RISC-V leaves more work to the interrupt trap footer: the software chooses the frame structure and must preserve the registers required by the ABI and the privilege level. PowerPC provides an exception frame tied to the profile, but the number of registers to be saved depends on the exception type and the implementation.

In a preemptive system, each thread has its own stack and saved context; the task switch must also handle the coprocessor registers owned by the thread. In a cooperative system, the context can remain in the registers until an explicit call, but an interrupt can still interrupt it. In all cases, stack limits are monitored via pattern filling, margin counters, MPU guards, or a combination of these mechanisms. The measurement method and its potential disturbance of real-time behavior must be documented.

Key takeaway

An interrupt must save and restore exactly the context it modifies. Control registers, the FPU, and the exception stack are part of the context contract.

6.2 Reset vector and interrupt table

The *reset vector* is the hardware mechanism that provides the first control point after reset. The *reset handler* is the code executed at that point; it usually performs very early initialization before calling the business entry point. A vector table then associates an exception cause or an interrupt with a handling address. It must be placed in a region known to the hardware, aligned as the architecture requires, and kept by the linker script, even if its entries are not referenced by an ordinary C call.
Interrupt vectors are specific to each architecture. They must comply with the target's specifications, in particular alignment, size, and entry format.

On an ARM Cortex-M, the table begins with the initial value of the Main Stack Pointer (the initial stack pointer of ARM architectures), followed by the address of the reset handler and then the exceptions and interrupts. The hardware places the stack at the supplied address and reads the second word at reset. On ARM Cortex-M, the ARM architecture specifies the system exceptions strictly ordered and prioritized in the vector table, followed by a variable number of peripheral interrupts, whose number and order are defined by the target [4, 5].

```c
#include <stdint.h>

/* linker-script symbol */
extern uintptr_t _stack_end;

void Reset_Handler(void);
void Default_Handler(void);
void SysTick_Handler(void);

__attribute__((section(".isr_vector"), used, aligned(128)))
const uintptr_t vector_table[] = {
  /* Standard CMSIS layout: stack pointer and reset-handler address */
  (uintptr_t)&_stack_end,
  (uintptr_t)Reset_Handler,
  (uintptr_t)Default_Handler, /* NMI */
  (uintptr_t)Default_Handler, /* HardFault */
  (uintptr_t)Default_Handler, /* MemManage */
  (uintptr_t)Default_Handler, /* BusFault */
  (uintptr_t)Default_Handler, /* UsageFault */
  0U, 0U, 0U, 0U,
  (uintptr_t)SysTick_Handler,
```

```
};
```

Listing 6.2: Minimal vector table for an ARM Cortex-M

The exact type of the entries, the alignment, and the section attributes are specific to the toolchain used; the linker script must place `.isr_vector` at the expected address and use `KEEP`. On Cortex-M this is configured via the VTOR register assignment, which allows the table to be relocated during the handover between each firmware (bootloader, business image, etc.).

```c
#include <stdint.h>

extern void riscv_trap_entry(void);

static inline void riscv_install_trap_vector(void)
{
  uintptr_t address = (uintptr_t)riscv_trap_entry;
  /* Mode 1 requests vectored mode; the address must be aligned. */
  __asm__ volatile ("csrw mtvec, %0" : : "r"(address | 1U) : "memory");
}
```

Listing 6.3: Configuring a RISC-V trap handler

The assembly code of `riscv_trap_entry` must preserve the context according to the target's contract, read `mcause`, `mepc`, and `mtval` when necessary, then restore the context with `mret`. The effective size and alignment of the table depend on the microarchitecture and the number of causes supported. The `csrw` instruction is an architecture extension, not an ISO C operation: it remains confined to the architecture layer and is tested with the disassembly of the final image.

On PowerPC, there is no single table common to all variants. Book E profile MPC5xxx devices generally use a reset vector at a fixed address and IVPR/IVOR registers to build the addresses of exceptions and interrupts. The hardware vector often contains a short assembly sequence that saves the minimal state and then branches to a common handler. Other PowerPC variants use different offsets or vector mechanisms; the processor's manual therefore takes precedence over any generic example.

```c
/* The symbol is placed by the linker script at the reset address. */
__attribute__((section(".reset_vector"), used, aligned(16)))
void powerpc_reset_vector(void);

__asm__(
  ".section .reset_vector,\"ax\"\n"
  ".global powerpc_reset_vector\n"
  "powerpc_reset_vector:\n"
  "  b powerpc_reset_handler\n"
);

void powerpc_reset_handler(void)
{
  /* Initialize the machine state, IVPR/IVOR, and the stack before C code. */
  for (;;) {
  }
}
```

Listing 6.4: Indicative vector entry for a PowerPC Book E

This excerpt illustrates only the boundary between the vector and the handler; it does not constitute a complete boot sequence. The linker script must impose the address and size of

`.reset_vector`, and the initialization code must configure the vector registers according to the SoC. Exception handlers must also respect the save format imposed by the processor before executing C code.

A vector table is boot data, but it is also an interface with the hardware and the linker script. The following checks are part of firmware verification:

- verify by assertions and by the link map the address, alignment, and size of the table
- keep the table and the reached handlers reachable only by hardware reference
- verify that the initial stack value belongs to the allowed RAM region and respects the ABI's alignment
- document, for each handler, the calling context, the saved registers, the state of interrupts, and the behavior in case of an unexpected event
- test the reset, exception, interrupt, and return-from-bootloader paths on the actually linked binary

The boot sequence must remain short and deterministic. It must not use a global C object before copying `.data` and zeroing `.bss`, nor enable a device before its clock, its address, and its access rules are established. The handover to the business entry point occurs only once the stack, the memory sections, the vector table, and the necessary control registers satisfy their contracts.

6.3 Registers, coprocessors, and peripherals

A register is a storage area of defined width and semantics specific to the processor or the peripheral. It must not be confused with an ordinary C variable: a read can acknowledge an event, a write can trigger an action, and some values can change independently of the code executed. The target's documentation fixes, for each register, its address, its access width, its alignment, its reserved bits, its value after reset, and the effects of reading or writing. These properties are part of the firmware's hardware contract.

General-purpose registers are mainly used to hold operands, addresses, and temporary results. Special registers control the state of the processor, memory, interrupts, or the flow of execution. Peripheral registers are exposed in the SoC's memory map and drive an external function: clock, GPIO, UART, DMA controller, timer, or interrupt controller. Finally, a coprocessor may have its own register set and enable bits. An FPU handles floating-point operations; an MPU controls memory regions; a cryptographic accelerator or a DSP generally has a similar interface, but with contracts specific to the component.

6.3.1 Processor registers by architecture

On an ARM Cortex-M, general-purpose registers `R0` through `R12` are used by the ABI for arguments, results, and temporaries. `R13` is the Stack Pointer, `R14` the Link Register, and `R15` the Program Counter. The `xPSR` status register groups notably the computation flags and the exception state. `CONTROL` selects the execution mode and the stack used, while `PRIMASK`, `BASEPRI`, and `FAULTMASK` control exception masking. `VTOR` contains the base of the vector table when implemented. The `MSP` and `PSP` registers allow the main stack to be distinguished from the process stack.

Cortex-M extensions add control registers such as `CPACR` for coprocessor access and MPU registers like `RNR`, `RBAR`, and `RLAR` on the profiles that provide them. The names and availability vary by core: the code must therefore be conditioned by the target's configuration, not merely by the presence of a CMSIS header.

On RISC-V, the general-purpose registers `x0` through `x31` have a width defined by the ISA, with `x0` always equal to zero. The `pc` register is not accessible as an ordinary general-purpose register. CSRs control the machine state and exceptions: `mstatus`, `mie`, `mip`, `mtvec`, `mepc`, `mcause`, and `mtval` are common at the machine level. Additional privileged modes may provide `satp` for address translation and implementation-specific CSRs. The `F` and `D` extensions add floating-point registers and `fcsr`; their state must be saved if the execution context can change.

On a Book E profile PowerPC MPC5xxx, the general-purpose registers `r0` through `r31` are complemented by the condition register `CR`, the loop counter `CTR`, the link register `LR`, and the exception register `XER`. The machine state register `MSR` and the special-purpose registers `SPR` control notably exceptions, memory, and mechanisms specific to the core. The `IVPR` and `IVOR` registers are used to build the vectors on the variants that implement them [21, 36]. MPC5xxx devices do not all have the same set of peripherals or coprocessors: the SoC manual and the BSP configuration file are the authority for the actual list.

Key takeaway

Register names and availability depend on the architecture and the SoC. Portable code goes through a validated BSP and does not turn a processor extension into an ISO C guarantee.

6.3.2 Initializing coprocessors and protections

A coprocessor must not be used simply because the compiler knows how to produce its instruction. The boot code first checks its presence, its power supply, and its clock, then configures its access, its initial state, and the associated protections. If the equipment is not available, the compilation profile must forbid the corresponding instructions or provide a defined fallback path. The initial state must be deterministic: floating-point exceptions, control registers, error state, rounding, and any internal buffers are reset according to the component's contract. Typical coprocessors are the FPU, the MPU, and vector accelerators such as SIMD; their availability varies by architecture, manufacturer, and SoC.

On a Cortex-M4/M7 equipped with an FPU, access to the coprocessors is generally authorized by `CPACR` before the first use of a floating-point instruction. The activation must be followed by the instruction synchronization required by the architecture. In a preemptive system, floating-point registers must also be integrated into context save and restore, or their use must be limited to a context whose ownership is explicitly controlled. An ISR must not implicitly modify a thread's floating-point state.

```
#include <stdint.h>

#define SCB_CPACR (*(volatile uint32_t *)0xE000ED88U)
#define CPACR_CP10_CP11_FULL_ACCESS (0xFU << 20)

static void fpu_enable(void)
{
```


```
8     SCB_CPACR |= CPACR_CP10_CP11_FULL_ACCESS;
9     /* Barriers maintain memory ordering and instruction visibility. */
10    __asm__ volatile ("dsb" : : : "memory");
11    __asm__ volatile ("isb" : : : "memory");
12 }
```

Listing 6.5: Controlled activation of the FPU on Cortex-M

This excerpt is deliberately reduced: in a product, the address of `CPACR` and the masks come from the core's validated description, not from a copied constant. The read-modify-write operation must also be safe when the register is modified by another context. An MPU is only activated once non-overlapping regions, their access rights, their alignment, and the default region have been defined. The stack, the vector tables, and MMIO registers must receive attributes compatible with their uses; a protection enabled too early can prevent the boot code from accessing its own data. For a presentation dedicated to security objectives, attack scenarios, and validation criteria for physical memory protection on ARM, PowerPC, and RISC-V, see the H2Lab guide [18].

On RISC-V, FPU activation depends on the extension and on the floating-point state field of `mstatus`; an implementation may also provide a vector unit with a much larger state. The BSP must define the available instructions, the CSRs concerned, and the saving policy. On PowerPC, the presence of an FPU, an SPE unit, or a protection controller depends on the variant. The access bit in `MSR`, the control registers, and the synchronization sequence must be initialized with the instructions and constraints of the target.

Key takeaway

A coprocessor must be enabled, initialized, and integrated into the context contract before its first use. The FPU, the MPU, and accelerators are not options invisible to the scheduler or the handlers.

6.3.3 Memory map and peripheral access

The *memory map* describes the correspondence between addresses and physical resources: non-volatile memory, SRAM, boot-reserved areas, peripheral memory, DMA, core registers, and any areas shared between processors. An address alone does not define an access: one also needs to know the memory type, the allowed width, the alignment, the cache attributes, the security policy, and the barriers required. Reserved regions and address gaps must never be probed to detect the presence of a peripheral.

MMIO accesses are confined to the BSP and use volatile types of explicit width. A register structure can improve readability, but must not assume that the compiler inserts the expected padding: the offsets are checked by assertions and compared against the component's manual. Reserved fields remain present in the structure or are handled through documented offsets. A register marked *write-one-to-clear*, *read-to-clear*, *write-only*, or *read-only* cannot be manipulated as an ordinary variable.

Key takeaway

An MMIO address is not a C variable. Its width, its alignment, its read/write effects, its reserved bits, and its cache attributes must be defined by the peripheral's manual and encapsulated in the BSP.

Two models for handling MMIO peripherals are found in implementations:

- assigning a structure to an MMIO peripheral address. This usage is found for example in STMicroelectronics' HAL
- manipulating registers via access primitives (`ioread()` and `iowrite()`), as in the Linux kernel. This model is safer, as it allows width, alignment, and memory-barrier checks to be added and ensures good abstraction for peripheral drivers

```c
#include <stdint.h>

struct uart_registers {
  volatile uint32_t control;
  volatile const uint32_t status;
  volatile uint32_t interrupt_clear;
  /* remaining registers */
};

#define UART0 ((struct uart_registers *)0x40000000U)
#define UART_STATUS_TX_READY (1U << 7)

static void uart_clear_interrupt(uint32_t mask)
{
  /* direct register assignment with data synchronisation barrier */
  UART0->interrupt_clear = mask;
  /* Write barrier to guarantee access visibility. */
  __asm__ volatile ("dmb" : : : "memory");
}
```

Listing 6.6: MMIO access with explicit width and semantics, missing abstraction

```c
#include <stdint.h>
/* generic register manipulation header, arch specific */
static inline uint32_t ioread32(uintptr_t addr)
{
  uint32_t res = *(volatile uint32_t *)addr;
  /* Read barrier to guarantee access visibility. */
  __asm__ volatile ("dmb" : : : "memory");
  return res;
}

static inline void iowrite32(uintptr_t addr, uint32_t value)
{
  *(volatile uint32_t *)addr = value;
  /* Write barrier to guarantee access visibility. */
  __asm__ volatile ("dmb" : : : "memory");
}

/* driver implementation, arch independent */
#define UART0 ((struct uart_registers *)0x40000000U)
#define UART0_CR_OFFSET 0x00U
```


```c
#define UART_STATUS_TX_READY (1U << 7)

static void uart_clear_interrupt(uint32_t mask)
{
    uint32_t val = ioread32((uintptr_t)UART0 + UART0_CR_OFFSET);
    val |= mask;
    iowrite32((uintptr_t)UART0 + UART0_CR_OFFSET, val);
}
```

Listing 6.7: MMIO access through register access primitives

The `volatile` qualifier requires the compiler to keep accesses observable; it guarantees neither atomicity, nor ordering across cores, nor coherence with a DMA. A write to a peripheral may require a `DMB` barrier (Data Memory Barrier), a `DSB` synchronization (Data Synchronization Barrier), a cache invalidation, or a readback sequence. These operations are defined in the architecture layer and are added only after reading the peripheral's manual. A busy wait loop must furthermore have an exit condition or a justification for permanent blocking; otherwise a clock or peripheral failure can cause the boot deadline to be exceeded.
It is important to note that using active wait loops on a value change in a register does not guarantee coverage through symbolic analysis. Indeed, the prover considers that an external modification is not guaranteed, and considers the rest of the function unreachable (violation of the termination guarantee). Timeout counters are then used, allowing any problem in processing register writes or reads to be detected without ever risking a definitive lockup.

Read-modify-write is a frequent risk. It is unsuited to registers with reserved bits, that acknowledge an interrupt on read, or that are modified by the hardware. For a software-controlled field, the BSP keeps the value and writes a full mask when the manufacturer allows it. For a register shared with an ISR or another core, the hardware protocol provided must be used: an atomic operation, a lock, interrupt masks, or a SET/CLR register. A plain `volatile` object does not constitute a lock.

Finally, initializing a peripheral follows a contractual sequence: enable its clock, remove its reset, configure its I/O and its mode, program its limits, clear events potentially left over from a previous use, then, if needed, install its handler and enable its interrupt(s) once the central interrupt controller is properly configured.

6.4 The memory hierarchy

In general, including in microcontrollers, memory does not have a uniform access cost [17]. The processor accesses registers first, then cache levels, SRAM or DRAM, and finally non-volatile memories and peripherals. Each level has a different latency, transfer width, persistence, and coherence attributes. A correct optimization therefore does not merely bring a piece of data closer to the core: it must also preserve the access rules of the peripheral, the DMA, and the other cores.

General-purpose registers are closest to the core but do not constitute general memory. Caches hold copies of memory lines and can make a write invisible to a peripheral until it has been flushed to the next level. Internal SRAM generally offers deterministic latency and is suited to stacks, short buffers, and real-time data. External memory or Flash can introduce wait states,

alignment constraints, and erase operations incompatible with a critical path.

A cache can be split into an instruction cache and a data cache, or be unified. Most of the time, the first-level cache (L1 cache) is specialized, separating instruction cache and data cache. The instruction cache speeds up code reads; the data cache speeds up reads and writes of objects. When a code area is written and then executed, the firmware must make the new instructions visible to the instruction cache. When a buffer is filled by the CPU and then consumed by a DMA, it must flush the relevant lines before starting the transfer. When a DMA fills a buffer read by the CPU, the firmware must invalidate any copies present before reading it. The exact operations depend on hardware coherence and are defined by the BSP; a plain `volatile` qualifier performs no cache maintenance at all.

Scratchpad memories (explicitly managed buffer memories) are SRAMs explicitly managed by software. They do not necessarily have a cache and therefore offer more predictable latency, at the cost of manual management of placement and capacity. *Tightly Coupled Memories* (TCM) are connected directly to the core. On ARM Cortex-M, the ITCM is intended for instructions and the DTCM for data; they can provide deterministic access for a handler, a stack, or a critical buffer. They are not, however, always visible to a DMA controller or to another core. The linker script, the MPU attributes, and the SoC documentation must therefore confirm who can access each region.

```
/* The linker script maps .fast_data to SRAM or to the DTCM. */
__attribute__((section(".fast_data"), aligned(32)))
static uint8_t dma_staging_buffer[1024U];

/* Critical code is placed in ITCM when the target allows it. */
__attribute__((section(".fast_code")))
static void control_loop_step(void)
{
  /* Bounded-duration processing with no access to external memory. */
}
```

Listing 6.8: Placing critical data in a dedicated memory

Placement alone does not establish a real-time property. One must verify the size of the sections, their load and execution addresses, the initialization of the data, DMA access, and the possible presence of conflicts with the stack or the vectors. A fast but saturated section can cause a linking error; an uninitialized section may hold a residual value at reset.

6.4.1 Synchronization and memory barriers

A memory barrier does not carry data and does not make an operation atomic. It only imposes an order or a visibility on a given actor. At least three guarantees must therefore be distinguished:

- the order observed by the compiler
- the order observed by the core
- the order observed by peripherals or the interconnect.

A C11 barrier such as `atomic_thread_fence` concerns the language's memory model. An ISA barrier instruction concerns the processor; a cache-maintenance operation concerns the physical visibility of a line. These mechanisms may be needed simultaneously.

On ARM, `DMB` orders the memory accesses observable by the indicated domain, `DSB` waits for the completion of accesses before continuing, and `ISB` flushes the instruction stream after a context or control change. On RISC-V, `fence` orders the specified categories of reads and writes, while atomic instructions and their ordering bits provide the guarantees associated with an atomic operation. On PowerPC, synchronization and memory-ordering instructions depend on the profile and the SoC; the BSP must provide a primitive named according to its semantics rather than exposing instructions in business code.

```
/**
 * @brief Processor memory barrier
 *
 * This barrier prevents processor to re-order load/store after this barrier
 */
inline __attribute__((always_inline)) void arch_data_mem_barrier(void) {
    asm volatile ("dmb 0xf" ::: "memory");
}


/*
 * Make sure that any explicit data memory transfer specified before is done
    before the
 * next instruction (harder than previous case, but slower).
 */

/**
 * @brief Processor memory synchronisation barrier
 *
 * This barrier wait for all previous memory access to be done before continue
 * execution.
 */
inline __attribute__((always_inline)) void arch_data_sync_barrier(void) {
    asm volatile ("dsb 0xf" ::: "memory");
}

/**
 * @brief Processor instruction synchronisation barrier
 *
 * This barrier will discard all further prefetched instruction
 * This is needed after enabling an isr, context task switch, etc.
 */
inline __attribute__((always_inline)) void arch_inst_sync_barrier(void) {
    asm volatile ("isb 0xf" ::: "memory");
}
```

Listing 6.9: Abstraction of memory barriers for ARM Cortex-M

A producer-consumer sequence illustrates the distinction. The producer first writes the fields of a descriptor, executes the required barrier, and then publishes the ready state. The consumer reads the state, acquires visibility of the descriptor, and then reads the fields. A barrier placed after the publication is too late. Likewise, a barrier does not fix a race on a non-atomic variable and does not replace a hardware SET/CLR register.

Key takeaway

A barrier orders accesses, but does not carry data and does not make an operation atomic. The choice between a C11 barrier, an ISA instruction, and cache maintenance depends on the sharing domain.

Exclusive instructions, such as the ARM primitives based on `LDREX` and `STREX`, or RISC-V's atomic operations, allow a lock or a conditional update to be built when the hardware guarantees their scope. They can fail and must then be retried according to a documented bound. They are not necessarily available for an MMIO region, non-shared memory, or a particular interconnect. Before using them across cores, check cache coherence, reservation granularity, sharing domains, and the progress guaranteed by the manufacturer.

```
#include <stdint.h>

struct arm_semaphore {
  volatile uint32_t count;
};

/* Return 0 on success, -1 when the semaphore is unavailable. */
static int semaphore_try_wait(struct arm_semaphore *semaphore)
{
  uint32_t observed;
  uint32_t updated;
  uint32_t status;

  do {
    __asm__ volatile (
      "ldrex %0, [%2]\n"
      : "=r"(observed)
      : "0"(observed), "r"(&semaphore->count)
      : "memory");
    if (observed == 0U) {
      __asm__ volatile ("clrex" : : : "memory");
      return -1;
    }
    updated = observed - 1U;
    __asm__ volatile (
      "strex %0, %2, [%1]\n"
      : "=r"(status)
      : "r"(&semaphore->count), "r"(updated)
      : "memory");
  } while (status != 0U);

  __asm__ volatile ("dmb" : : : "memory");
  return 0;
}

static int semaphore_post(struct arm_semaphore *semaphore)
{
  uint32_t observed;
  uint32_t status;

  __asm__ volatile ("dmb" : : : "memory");
  do {
    __asm__ volatile (
      "ldrex %0, [%2]\n"
```

```
45        : "=r"(observed)
46        : "0"(observed), "r"(&semaphore->count)
47        : "memory");
48      if (observed == UINT32_MAX) {
49        __asm__ volatile ("clrex" : : : "memory");
50        return -1;
51      }
52      __asm__ volatile (
53        "strex %0, %2, [%1]\n"
54        : "=r"(status)
55        : "r"(&semaphore->count), "r"(observed + 1U)
56        : "memory");
57    } while (status != 0U);
58
59    return 0;
60  }
```

Listing 6.10: ARM binary semaphore with LDREX and STREX

This implementation uses an exclusive reservation on a variable located in shared, naturally aligned memory. `LDREX` reads the value and establishes the reservation; `STREX` writes only if this reservation is still valid, and then returns zero. An interrupt, another core, or another write can invalidate the reservation: the loop must therefore explicitly handle the failure of `STREX`. `CLREX` is necessary when the acquisition fails without a write.

The counter must be initialized to the semaphore's capacity before concurrent contexts are activated. In practice, `semaphore_post` must also check for overflow, and the protocol must define who has the right to release the semaphore. `DMB` barriers do not replace the exclusive reservation: they order accesses around the publication of the lock, but do not make an ordinary operation atomic. This primitive is suited neither to MMIO registers nor to memory whose coherence is not guaranteed by the interconnect; the BSP must then provide a primitive suited to the sharing domain.

Typically, on SoCs that do not ensure global coherence, for example when they host heterogeneous cores, synchronization between domains is performed using dedicated peripherals such as hardware FIFOs or mailboxes, described in the next chapter.
These peripherals then take charge of ensuring locking mechanisms between accessors.

Key takeaway

An atomic primitive alone does not define a protocol. It is also necessary to specify the alignment, the sharing domain, cache coherence, the progress bound, and the behavior when the reservation or the queue fails.

6.5 Notion of firmware and bootloader

A firmware is the whole of the persistent software intended to drive a given hardware target. It notably includes the initialization code, the drivers, the application services, and the constant data required for its execution.
In a bare-metal system, the firmware runs without an operating system that would provide

an abstraction of the processor or the peripherals. It must therefore handle the earliest boot operations, memory configuration, and interrupt distribution. The term firmware describes a function and a delivery mode, not a particular binary format: the image may be represented by an archived ELF, a raw binary file, or IHEX records.

The bootloader is a boot component whose responsibility is to prepare the execution of another image. It may be integrated into the application image, placed in a reserved region of non-volatile memory, or provided by the manufacturer. Its responsibilities must be bounded and documented: determine the source of the image, check its bounds and its format, prepare the indispensable resources. It then transfers control to an entry point conforming to the image's contract. Cryptographic verification of images and secure boot are deliberately outside the scope of this guide in order to limit its complexity, but the secure-boot issue must still be examined in light of the product's context.

6.5.1 BootROM, bootloader, and application image

After power-up or a reset, the processor's initialization is determined by the SoC manufacturer. The main processor is not necessarily the first to be initialized. This is typically the case in SoCs featuring an HSE (*Hardware Security Engine*) or any specialized processor in charge of platform initialization. Depending on the manufacturer and the SoC model, several firmwares may be chained before the application image executes. This is where firmwares integrated into the SoC by the manufacturer, such as BootROMs, come in.
When implementing firmware, it is important to understand a platform's boot logic, and to know the steps preceding execution of the application image.
It may thus be necessary to reset certain peripherals, reconfigure clocks, complete memory initialization, or verify the integrity of an image before it can be executed.

A typical boot chain has the following steps:

1. reset puts the processor and peripherals into a defined state
2. the BootROM or the reset vector selects and locates the next image
3. the bootloader configures its minimal context, checks the image's bounds, and possibly initializes the memory required to read it
4. the application image installs its vector table, initializes or reconfigures memory, clocks, protections, and the required peripherals
5. the boot code calls the business entry point

Key takeaway

Every boot transition must define the image, the stack, the processor mode, the interrupts, the reserved registers, and the return conditions. A jump address alone does not constitute a transfer contract.

Each transition is an interface between two components. It fixes at a minimum the load address, the execution address, the state of interrupts, the processor mode, the stack pointer, the reserved registers, and the calling convention. Before the jump, a bootloader must disable, or put back into a documented state, the interrupts it has enabled, stop its peripherals, and hand over

an explicitly defined context. It is not enough to point the instruction pointer at a function's address. An image may require the presence of a vector table, an aligned stack, or control registers already configured. Usually, the boot code of an application image ensures that all the resources it uses are put back into a known state, even if the bootloader has already performed part of this work, in order to guarantee the absence of too strong a dependency on a particular bootloader.

7

Integration into multi-core architectures

Multi-core architectures are increasingly present, including in microcontrollers. This is, for example, the case of the SoCs (System on Chip) of the Raspberry Pi Pico2, providing two SMP (Symmetric Multi-Processing) cores, and of MPC5xxx-type SoCs, which provide two cores that can be enabled in SMP or as a *step core* to ensure execution redundancy.

The following diagrams represent three common organizations. They are functional rather than physical: the addresses, coherence protocols, and exact latencies depend on the SoC. In each diagram, caches are associated with the core or cluster that owns them, while SRAM and Flash represent the resources visible to the system.

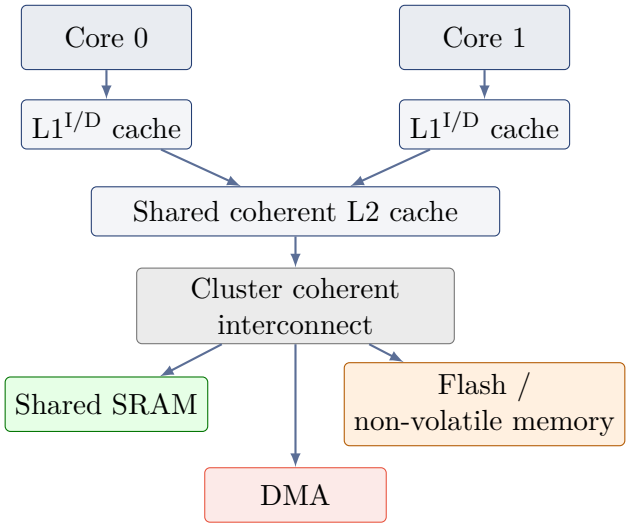


Figure 7.1: Homogeneous two-core SMP architecture within a coherent cluster.

In a homogeneous SMP architecture, the cores execute the same instruction set and generally share a coherence domain. Each core may have a private L1 cache, while an L2 cache and the interconnect arbitrate access to SRAM, Flash, and DMA. Coherence does not remove the protocols: it makes cache lines visible according to the hardware's rules, but atomics, barriers, and buffer ownership remain necessary.

Several homogeneous clusters may each have a local cache and an internal interconnect. The system interconnect then links the clusters, the memories, and the peripherals. Coherence may

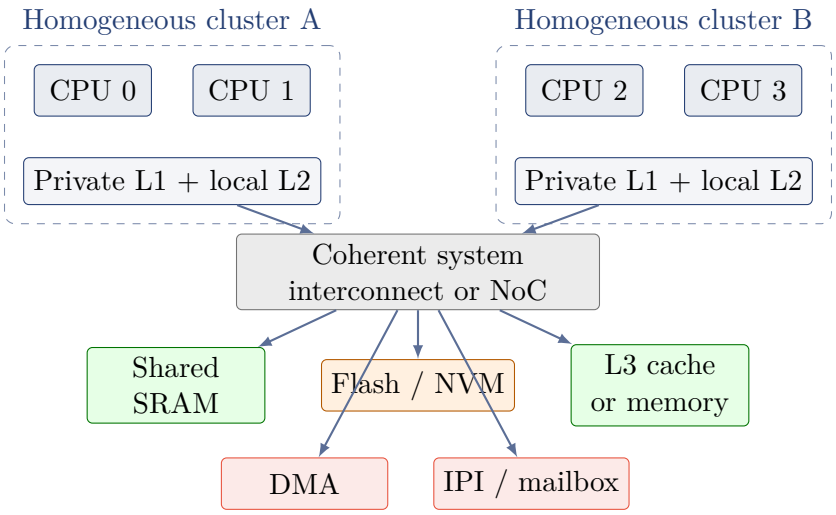


Figure 7.2: Homogeneous multi-cluster architecture with system-wide coherence.

be global, limited to certain regions, or absent between clusters: the firmware must then declare the sharing domain and perform the necessary cache and barrier operations before a transfer. A mailbox or an IPI can signal an event, but does not replace the ordered publication of data.

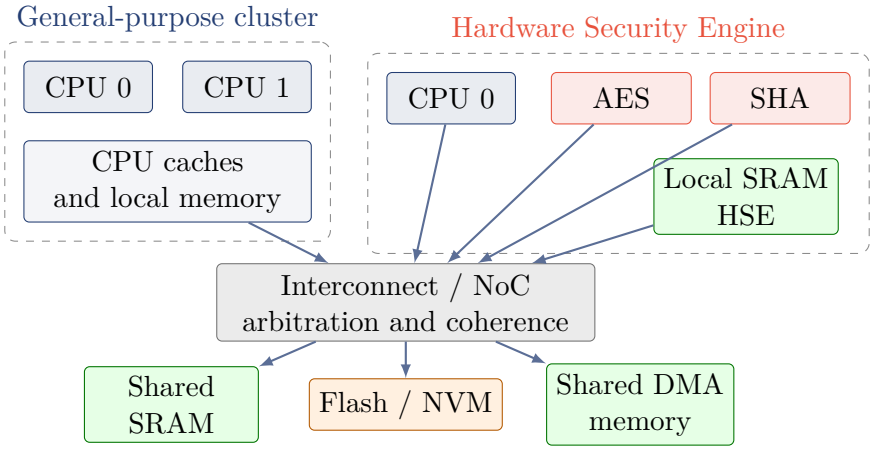


Figure 7.3: Heterogeneous architecture combining general-purpose cores and accelerators.

In a heterogeneous architecture, the general-purpose cluster, an HSE core, etc. generally have different instruction sets, caches, and memory domains. The interconnect provides transport and arbitration, but does not turn these domains into uniform memory. Exchanged buffers must be described by a fixed binary interface, with address, size, alignment, format, owner, and state. Cache maintenance may be required before and after an accelerator's access. Bus priority must also be checked against the real-time requirements of the DMA and the general-purpose cores.

7.1 Concurrency issues

Concurrency appears as soon as several contexts can observe or modify the same state: cores of an SMP processor, threads, ISRs, DMA controllers, or heterogeneous cores. It therefore does not depend on the presence of an operating system. A bare-metal firmware must inventory the producers and consumers of each object, its owner, its lifetime, its sharing domain, and the

protocol that guarantees its access. A `volatile` qualification makes certain accesses observable by the compiler, but resolves neither the data race, nor atomicity, nor cache coherence.

A *race condition* occurs when the result depends on the interleaving of two uncoordinated sequences. When at least one sequence writes a non-atomic object and no ordering relation synchronizes them, this is a data race and the behavior is undefined with respect to the C11 memory model. A hardware register, a DMA area, or memory shared between cores must additionally respect the SoC's guarantees, which are not all expressed by the C language [31].

A classic case is the following shared counter: two cores read the value 4, each compute 5, and each write 5. One update has then been lost. An interrupt can produce the same defect on a single core between the read and the write of a variable. Compound accesses such as `counter++`, a test followed by an action, and a read-modify-write are atomic only if their hardware or software contract guarantees it.

The TOCTOU defect (*time of check to time of use*) is a particular form of race: the program checks a property, then the state changes before it is used. In firmware, this concerns for example checking a DMA descriptor before launching it, reading a peripheral state before clearing an alert, or validating a shared-memory address before a later access. The solution is to hold the lock or the property for the entire operation, revalidate the state right before use, or use a primitive that combines the test and the state transition [7, 33].

These defects are exploited in several families of attacks: double-fetch of data controlled by a peripheral or another privilege domain, modification of a descriptor between its validation and its use, concurrent release of a buffer, or repeated triggering of an interrupt during a sequence assumed to be indivisible. The common point is not the type of processor, but the absence of an explicit property on the data and the time window. A security review therefore looks for checks separated from their use, reentrant callbacks, shared counters, and error paths that release a resource twice [16, 32].

Thread/ISR concurrency also exists on a single core. The ISR interrupts the thread between two instructions and can modify a state that the thread then uses. For a short critical section, the firmware can mask interrupts at the appropriate level, provided its duration is bounded and it does not block a critical real-time interrupt. For a state shared with several cores or a DMA, local masking is not enough: an atomic primitive, an inter-core lock, a queue, or an ownership protocol is required.

```
#include <stdint.h>

static volatile uint32_t pending_events;

void timer_isr(void)
{
  pending_events++;
}

uint32_t take_events(void)
{
  uint32_t events;

```

```
14    /* The architecture layer disables and restores the local interrupt mask. */
15    arch_irq_state_t state = arch_irq_save();
16    events = pending_events;
17    pending_events = 0U;
18    arch_irq_restore(state);
19    return events;
20  }
```

Listing 7.1: Counter shared between a thread and an ISR

This example is not a universal primitive: it is valid only if reading, clearing, and restoring the mask are atomic with respect to the ISR on this target. Between cores, this local protection is replaced with an atomic operation or a shared queue. Each solution also documents its progress: an unbounded spinlock can permanently block a core if the owner is stopped, preempted, or the victim of a fault.

7.2 Exchanges between cores

Assigning an event to a core is an architectural decision. An interrupt may be directed to a fixed core, distributed among the cores of the same cluster, or received by a controller that wakes the core that owns a queue.
The BSP describes the affinity registers, the core that configures the controller, the interrupts reserved for startup, and the behavior when a core is stopped. An interrupt received by the wrong core must be acknowledged according to the controller's protocol, without modifying a state that this core does not own.

For two homogeneous cores sharing coherent memory, a spinlock, an atomic operation, or an IPI may be sufficient. The spinlock protects a short critical section, while the IPI signals that an event or data is available. The IPI does not by itself carry the data: the producer first publishes the data with the required barrier, then triggers the IPI, and the consumer acquires visibility before reading the structure. Acknowledgment and the possible loss of an IPI must be explicitly handled. An IPI must not be used as a counter without an associated mechanism.

In this example, the sending core first writes the payload into `DATA0` and `DATA1`, then publishes the event in `DOORBELL`. The `STATUS` register may indicate the empty, full, busy, or acknowledged state depending on the peripheral's contract. The consuming core reads the data after observing the notification, then writes the expected acknowledgment. The registers are represented as MMIO access points: their order, their read and write effects, and the possibility of coalescing several notifications are defined by the SoC.

A mailbox (hardware letterbox) generally provides registers or a small queue to transmit a word, an address, or an event between domains. A hardware FIFO carries several elements and can manage filling, draining, and upstream pressure. These two mechanisms reduce direct sharing of structures, but their registers have their own rules: atomic writing, acknowledgment, overflow, reset, and visibility of referenced data. An address placed in a mailbox must remain valid and accessible to the consumer for the entire duration of the message.

```
1  struct mailbox_registers {
2    volatile uint32_t data0;
3    volatile uint32_t data1;
4    volatile uint32_t status;
```

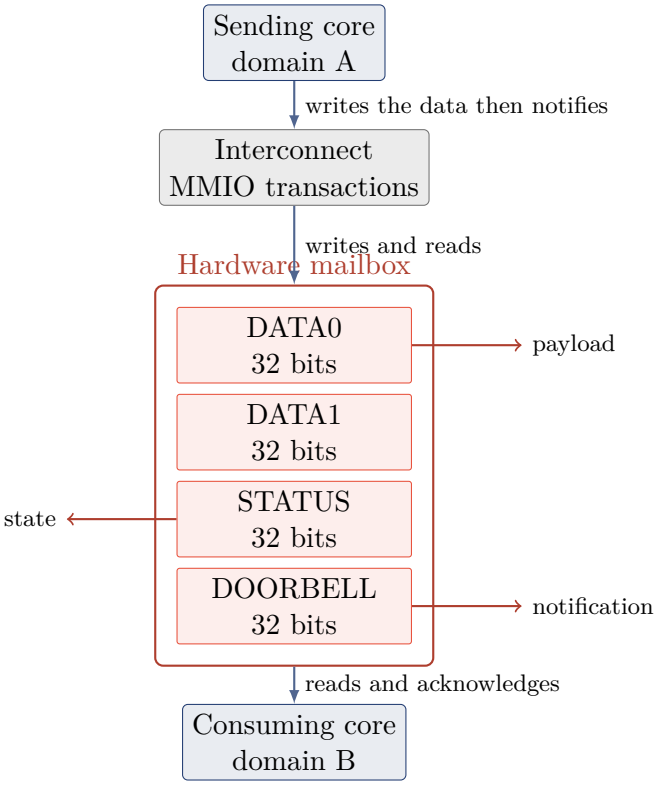


Figure 7.4: Hardware mailbox with four 32-bit registers between two domains.

```
  volatile uint32_t doorbell;
};

#define MAILBOX0 ((struct mailbox_registers *)0x40010000U)
#define MAILBOX_STATUS_EMPTY 0U
#define MAILBOX_STATUS_READY 1U
#define MAILBOX_DOORBELL_READY 1U

static int core0_send_message(uint32_t command, uint32_t argument)
{
  if (MAILBOX0->status != MAILBOX_STATUS_EMPTY) {
    return -1;
  }

  MAILBOX0->data0 = command;
  MAILBOX0->data1 = argument;
  arch_data_mem_barrier();
  /* The doorbell write can generate an interrupt toward the consumer */
  MAILBOX0->doorbell = MAILBOX_DOORBELL_READY;
  return 0;
}

static void core1_poll_mailbox(void)
{
  if (MAILBOX0->status == MAILBOX_STATUS_READY) {
    uint32_t command = MAILBOX0->data0;
    uint32_t argument = MAILBOX0->data1;

    arch_data_mem_barrier();
    MAILBOX0->status = MAILBOX_STATUS_EMPTY;
```

```
35      dispatch_message(command, argument);
36    }
37  }
```

Listing 7.2: Publishing a message via a mailbox

The `arch_data_mem_barrier` primitive is provided by the BSP and translates the barrier required by the ISA and the sharing domain. It guarantees here that the writes to `DATA0` and `DATA1` are observable before the write to `DOORBELL`. The latter constitutes the hardware notification: the consumer may receive it as a peripheral interrupt or detect it by reading `STATUS`. This model allows communication between heterogeneous cores. It also allows crossing clock domains.
The protocol must prevent the producer from overwriting the message before it is consumed. In this example, the producer refuses to send when `STATUS` is not empty, and the consumer frees the mailbox after copying both data items. The status values, the read effects, and the acknowledgment semantics must be defined by the peripheral's manual; a `STATUS` register of the write-one-to-clear kind is not manipulated like a plain variable.

Synchronization of cores within the same cluster generally benefits from a common coherence domain, low latency, and atomic instructions directly applicable to the shared SRAM.
Conversely, between homogeneous clusters, coherence may be partial or limited to certain regions. It is then necessary to specify the sharing domain, flush or invalidate caches if necessary, and use the interconnect or the communication registers provided. Between heterogeneous cores, the ISAs, word widths, ABIs, and memory models may differ; a mailbox, a FIFO, or a memory area described by a fixed binary format is generally preferable to a directly shared C pointer or structure.

7.3 Execution concurrency

Execution concurrency does not only concern shared variables. Cores, DMAs, accelerators, and peripherals also compete for interconnects and memory banks. The interconnect carries transactions, applies visibility domains, and may provide cache coherence. It does not automatically guarantee that two concurrent writes are handled in the order expected by the software; this ordering is a matter of protocol and barriers.

The bus arbiter chooses which request is served when several masters request a resource. Depending on the SoC, arbitration may be fixed, round-robin, priority-weighted, or quality-of-service aware. A core holding a lock can be slowed down by a higher-priority DMA transaction; conversely, repeated access to shared memory can starve a real-time peripheral. Sizing must therefore consider maximum latency, bandwidth, bursts, bank conflicts, and the interconnect's internal queues.

Homogeneous cores often run the same code and share an address space, but they may still have private caches, write buffers, and different latencies. Heterogeneous cores add constraints of format conversion, width, and buffer ownership. A DSP or a GPU may only see a CPU write after cache maintenance and a specific barrier. A C pointer passed as-is is then insufficient: the message must specify the address, the size, the alignment, the format, and the ownership duration of the buffer.

A robust communication protocol defines at a minimum the message's state, its producer, its consumer, the allowed transitions, the saturation condition, recovery after reset, and acceptable delays. Counters and sequences have a fixed width; reserved values and transport errors are distinguished from business-processing errors. The protocol is tested under load, with delays, reordering, lost events, and a reset of a single domain.

Finally, bus priority must not become an implicit security mechanism. A peripheral that can write a shared area must be limited by the MPU, the SMMU, or the DMA controller's attributes when these mechanisms exist. Domain separation, length validation, and buffer ownership remain necessary even when the interconnect is coherent.

8

Designing secure embedded firmware

8.1 Reminders about embedded security

The security of embedded firmware is not reduced to the confidentiality of a key or the presence of a cryptographic primitive. It targets availability, integrity, authenticity, and control over access to functions and data, from design through the product's retirement. Risk analysis describes the assets, the actors, the attack paths, the consequences, and the risk-reduction measures. It must cover the hardware, the software, the development tools, production, maintenance, and the physical interfaces [3, 35].

A microcontroller exposes a particular attack surface: debug pins, recovery interfaces, internal buses, DMA, bootloader, updates, radio or wired protocols, and data kept in Flash. An attacker may exploit a poorly validated input, a concurrency defect, a privilege configuration, a physical fault, a timing leak, or a compromised software dependency. Security measures must therefore be linked to an identified threat and tested on the binary and configuration actually delivered.

Physical security concerns notably access to the pins, reading or replacing memory, fault injection, observation of consumption, and the mechanical resistance of the enclosure. Logical security concerns domain isolation, input validation, memory rights, protocol robustness, logging, and recovery after an error. Organizational security concerns accounts, signing secrets, reviews, version traceability, separation of roles, and vulnerability response. A logical protection does not compensate for a production key accessible to the whole team, and an organizational procedure does not fix an unrestricted DMA.

Security by design fixes the security invariants before implementation: a peripheral can only write into its buffers, an interface refuses lengths outside its contract, a privilege is granted only to the function that needs it, and an error leads to a defined state. Security by default chooses the most restrictive profile when an option is not explicitly configured. Security by usage adds operational controls: update procedure, secrets management, event monitoring, access restriction, and incident response. These three principles must appear in the requirements, the interfaces, the tests, and the proofs.

8.1.1 Assets, boundaries, and security objectives

For each asset, the project documents its owner, its representation, its lifetime, the domains that can read or write it, and the expected behavior in case of a fault. Keys, identifiers, anti-replay counters, firmware images, calibration data, and logs do not have the same requirements. A key may require an area unreadable by application code; a sequence counter mainly requires a monotonic update resistant to interruptions; a log requires a coherent recovery after reset.

Security boundaries are placed at the interfaces actually crossed: an ISR call, a DMA descriptor, an inter-core mailbox, a user command, a bootloader function, and a driver primitive. Each boundary has a format, bounds, an error policy, and a proof of ownership. Pointers are not passed between domains as if they had the same meaning; a cross-domain interface carries a validated address or a buffer identifier according to the hardware's contract.

Key takeaway

An asset is not defined by its name alone: its ownership, its lifetime, its readers, its writers, its format, and its reaction to faults must be documented.

Key takeaway

A threat must be linked to an asset, a boundary, a protective measure, and a means of verification. An isolated measure, with no owner and no test, does not constitute a demonstrated security objective.

8.2 Secure development process

Secure development is a configuration and proof process, not a step added before delivery. A firmware version is associated with a commit, a build configuration, a compiler, a linker script, the sources of its dependencies, analysis results, and the production artifacts. Any modification of the code, the BSP, the compiler, the options, the memory map, or the generation tools can change the qualification scope [34, 9].

The supply chain includes libraries, code generators, test tools, coprocessor images, peripheral firmwares, configuration files, and production equipment. The project keeps the provenance, version, integrity, license, and support status of each component. Unused dependencies are removed, and the ones that remain are monitored and their impact analyzed.

An SBOM describes the components present in the product, their versions, their relationships, and, when possible, their identifiers and fingerprints. It must be generated from the actually built configuration, archived with the ELF, and linked to the delivered image. An SBOM does not prove the absence of vulnerabilities: it makes the components identifiable and allows an analysis to be triggered when a vulnerability or a new requirement appears [14, 34].

Increasingly, a CBOM is also associated with it, making it possible to make the cryptographic algorithms, their usage, their positioning, and the lifecycle of the associated cryptographic assets explicit.

Peer review checks contracts, invariants, errors, and side effects. It must examine the C code, the assembly listings, the linker scripts, the MPU/cache configurations, the compilation rules, and the tests. TDD can guide the implementation of a pure primitive or a protocol: a test first describes the expected behavior, then the implementation is written and finally refactored under the campaign's control. For hardware-dependent code, unit tests are complemented by on-target tests, integration tests, and controlled fault injections.

8.2.1 Auditability and noRTE

Auditability requires that every result be reproducible and explainable. The project keeps tool versions, options, CI logs, static-analysis reports, coverage, proofs, the link map, and the binary's fingerprint. A suppressed alert is replaced by a referenced justification, never by silently disabling the tool.

Key takeaway

Auditability requires being able to reproduce a result and to explain every assumption, alert, exclusion, and artifact used to produce it.

The noRTE property is defined over a scope and assumptions: input bounds, valid memory, alignment, absence of an unmodeled interrupt, the peripheral's contract, and the hardware's behavior. It notably covers overflows, out-of-bounds accesses, invalid divisions, dangerous conversions, stack overflows, protocol errors, and non-termination paths. A noRTE proof does not automatically prove functional safety nor the absence of a hardware defect; it establishes the announced properties under the controlled assumptions.

Key takeaway

A noRTE proof is always bounded by its scope and its assumptions. It replaces neither on-target tests, nor protocol tests, nor validation of the hardware configuration.

8.2.2 Tests and coverage

Unit tests verify nominal cases, boundaries, and each reachable error of a function. Integration tests verify contracts between modules. On-target tests verify the actually linked code, the timings, the memory attributes, and the exception behaviors.

Statement coverage measures the instructions executed; branch coverage measures decision outcomes; condition coverage examines elementary conditions; MC/DC demonstrates that each condition can independently influence the decision. Path coverage is useful on bounded paths, but quickly becomes combinatorial with loops and interrupts. High coverage compensates neither poorly specified tests nor unidentified dead code. Exclusions are justified, versioned, and

re-examined.

Key takeaway

The SBOM identifies components, while tests, analyses, and proofs establish the firmware's properties. None of these artifacts replaces the others.

Key takeaway

Coverage measures what was executed, not what is correct. Nominal cases, boundaries, errors, interrupts, and recovery paths must be selected from the contracts.

8.3 Proper use of documentation

Automatic documentation is an instrument of consistency, not an architecture. Doxygen can extract C interfaces, module groups, preconditions, and call relations. Sphinx can assemble structured documents, architecture decisions, procedures, and reports. The versions of the generators, their options, and their inputs must be pinned so that the documentation is reproducible.

An interface documents its role, its inputs, its outputs, the errors, buffer ownership, the alignment, the calling context, interrupts, concurrency, execution time, and side effects. A critical function also documents its proof assumptions, the registers touched, the barriers, the resources acquired, and the behavior in case of a fault. Examples must specify whether they are normative, indicative, or dependent on a particular SoC.

Architecture documentation describes concepts and their limits: privilege domains, memory map, boot flow, buffer ownership, cache model, error paths, update strategy, and external dependencies. Doxygen cannot deduce the semantics of a register, the ownership of DMA data, or the reason why an interrupt is masked. This information must be written in an architecture document and linked to the requirements and the tests.

Useful documentation distinguishes the guarantees of ISO C, those of the compiler, those of the ABI, and those of the hardware. It explicitly flags extensions, non-portable profiles, and the assumptions that must be verified in the BSP.

Key takeaway

Interface documentation must expose the data, the errors, the concurrency, the side effects, and the target assumptions. Doxygen or Sphinx cannot deduce these contracts in place of the architect.

8.4 Software modularity

A CSC is a software component identified in an architecture; a CSCI is a configuration element grouping the software and its controlled artifacts; an ICD describes the interface between elements. These notions prevent treating the repository as an undifferentiated block: each module has an owner, a contract, a configuration, and a test strategy.

A good interface minimizes shared data, expresses resource ownership, and separates validation errors from execution errors. It does not return a value where the same number means both valid data and an error. Buffers have an explicit size, and operations indicate whether the areas may overlap. ISR calls and callbacks document their context and their prohibitions against blocking.

Standardized interfaces bring interoperability, but not an automatic proof of safety. PSA can structure access to cryptographic services; AUTOSAR formalizes layers and automotive contracts; ARINC 615 describes loading exchanges in certain aeronautical environments. The project verifies the version, the retained profile, the extensions, and the gaps between the standard interface and the hardware actually available.

Specifying an interface follows at least these steps:

- define the assets and the boundaries
- choose the types and formats
- fix the preconditions and postconditions
- describe ownership and concurrency
- define the errors and the timeouts
- establish the tests and the proofs

Every external interface is treated as hostile until its format and its limits have been validated.

A POSIX interface can be useful on an adaptation layer, but must not be confused with the firmware's typed memory model. Opaque handles, dual-meaning returns, implicit sizes, or pointer conversions can make the contracts difficult to prove. Critical code therefore exposes wrappers with explicit types and turns POSIX errors into distinct, verifiable results.
It is important to stress that some POSIX interfaces lend themselves poorly to a direct formal proof, because their contracts use opaque handles, implicit conventions, or returns with several meanings. For example, `read()` returns either a size or an error value; a typed wrapper can separate these two results before exposing the interface to critical code.

When modularly implementing firmware, a typical CSC decomposition is as follows:

- architecture support: ABI, exceptions, atomics, barriers, and context
- SoC and BSP support (drivers): clocks, GPIO, DMA, etc.

- *zero-lib* utility primitives: `memcpy`, `memset`, `strnlen`, byte conversions, and bounds checks
- driver primitives: MMIO access, queues, locks, timeouts, and acknowledgments
- higher-level services and application: protocols, storage, logging, and error policy

The implementations of `memcpy`, `memcmp`, byte conversions, and driver primitives of Sentry Kernel and Shield can serve as design references [10, 11]. They must be adapted to the target's profile, then subjected to the project's analysis and tests: reusing code transfers neither its assumptions nor its qualification.

```
enum copy_status {
  COPY_OK = 0,
  COPY_INVALID_ARGUMENT = 1,
  COPY_OVERLAP = 2
};

enum copy_status copy_bytes(uint8_t *destination,
                            size_t destination_size,
                            const uint8_t *source,
                            size_t length);
```

Listing 8.1: Typed memory-copy interface

This interface makes the sizes and the errors visible, unlike a primitive that would implicitly assume the buffer is valid or would confuse a returned length with an error code. The complete contract specifies whether the areas may overlap and whether a zero length allows null pointers.

Key takeaway

A safe interface makes types, sizes, resource ownership, errors, and concurrency rules explicit. It reduces both the proof surface and the attack surface.

8.5 Security by design

Security by design applies attack-surface minimization, privilege separation, and separation of responsibilities. A module receives only the registers, buffers, and commands necessary for its function. Inputs are validated as close as possible to the boundary; secrets do not cross an interface that does not need them. Error paths destroy or lock sensitive resources into a known state.

A bare-metal super-loop is simple to analyze but often shares a privilege space, a stack, and global objects. A fault in a driver can then affect the whole application. Cooperative firmware introduces several tasks that voluntarily yield the processor: it must control reentrancy, calls that do not yield, and critical sections, but allows minimal partitioning via the MPU. A preemptive micro-kernel better separates domains and stacks, but adds transitions, contexts, queues, and scheduling paths that must be verified.

In a preemptive system, each thread has a saved context and a bounded stack. Context switching must preserve the callee-saved registers, the interrupt state, the control registers, and,

if necessary, the FPU, SIMD, or DSP registers. The choice between systematic saving and lazy saving must be documented, as a forgotten floating-point state can leak or corrupt another thread's data. The interrupt context has its own rules: an ISR must not call a blocking primitive nor modify a thread's state without a protocol.

Key takeaway

The simplicity of a super-loop makes analysis easier, but does not automatically provide isolation. A preemptive or multi-core system adds protection mechanisms that must themselves be specified and verified.

8.6 About scheduling

A cooperative scheduler only switches tasks at explicit points. It is predictable and inexpensive, but a task that never yields blocks the others. A preemptive scheduler can interrupt a task according to a clock, a priority, or an event; it improves temporal isolation but requires context saving and rigorous synchronization.

A priority is useful only if the protocol handles priority inversion, starvation, and shared resources. A lock held by a low-priority task can delay a critical task; priority inheritance or priority ceiling can reduce this phenomenon when their cost is acceptable. Critical sections remain bounded and do not perform a slow or non-deterministic operation.

In real-time systems, for periodic tasks, RMA assigns fixed priorities based on periods, EDF chooses the task whose deadline is closest, and TDM reserves time windows. FIFO and round-robin distribute time without guaranteeing a real-time deadline. The choice must be linked to execution bounds, interrupts, memory accesses, and blocking times, not only to the nominal policy.

In real-time SMP, the load can be distributed, but migration, cache, IPI, lock, and interconnect-access costs add to execution times. An analysis checks affinities, non-migratable resources, per-core queues, memory contention, and the behavior when a core is stopped. A correct scheduler does not automatically make the drivers reentrant.

8.7 Side effects and interference

A side effect is a modification observable outside a function's local result: an MMIO write, a buffer modification, an event publication, an interrupt acknowledgment, or a cache change. Interfaces document these effects so that the caller can establish the order, the ownership, and the restoration of invariants.

Interference occurs between modules through globals and callbacks, between threads through shared objects, between cores through caches and the interconnect, between peripherals through

clocks and buses, and between coprocessors through context registers and buffers. A compositional analysis inventories the common resources, the access orders, the priorities, and the fault paths. It also checks that a partial reset does not leave a peripheral in a state that the following software would interpret as valid.

Practical controls include explicit ownership, opaque types, copies at the boundaries, documented barriers, timeout counters, rights masks, absence of exported mutable globals, and tests under concurrency. An optimization must not remove a hardware access or a barrier required just because it appears unnecessary to the C model.

8.8 Proof, auditability, and traceability

The noRTE proof starts with a model: bounded types, preconditions, loop invariants, postconditions, and assumptions about external calls. Frama-C and its plugins can then verify bounds, pointers, divisions, shifts, and functional properties [12, 6]. The result is a set of proof obligations, not a global claim independent of the configuration.

Compositional proof first establishes a module's properties, then verifies that the interfaces' contracts link the modules without invalidating their assumptions. For a DMA buffer, it combines, for example, the proof of the parser's bounds, the BSP's alignment property, cache maintenance, and the ownership protocol. For an ISR, it adds the interrupted invariants, the preserved registers, and the time bound of the critical section. Unmodeled interference remains an explicitly declared residual risk.

```
1  /*@ requires destination != \null;
2      requires source != \null;
3      requires length <= destination_size;
4      requires \separated(destination + (0 .. length -1), source + (0 .. length -
       1));
5      assigns destination[0 .. length - 1];
6      ensures \valid(destination + (0 .. length - 1));
7      ensures \valid_read(source + (0 .. length - 1));
8  */
9  int copy_checked(unsigned char *destination,
10                  unsigned long destination_size,
11                  const unsigned char *source,
12                  unsigned long length);
```

Listing 8.2: ACSL bounds-proof example

For a zero length, the contract must be formulated according to the syntax and the model adopted by the tool in order to avoid creating an invalid range. Real functions add non-overlap preconditions or explicitly choose `memmove` semantics. The proof of `memcpy` and `memcmp` in Sentry Kernel constitutes an example of a reusable noRTE approach, but it must be replayed with the firmware's configuration.

8.9 noRTE proof of embedded firmware

ACSL allows the preconditions, postconditions, side effects, invariants, and error behaviors of a C function to be expressed. Hoare logic summarizes the contract with a triple: {P} C {Q}, where the precondition P must guarantee that after the execution of C, the postcondition Q is true. For a parser, P describes the validity and length of the buffer; Q describes the result and the state of the cursors; the intermediate obligations prevent out-of-bounds accesses and overflows.

The prover often works on a superset of the actual executions. A path considered possible by the analysis may be impossible on the hardware, but it must nevertheless be proved or explicitly bounded by a verified assumption. Conversely, complete proof coverage only demonstrates the properties written and does not replace protocol, performance, and configuration tests.

Proving a critical function follows a reproducible sequence: reduce the types and states, write the contracts, establish the invariants, launch the analysis, resolve the obligations, examine the remaining assumptions, and then replay the proof in the CI configuration. Warnings, axioms, functions left `axiomatic`, and unmodeled calls are elements of the proof file.

For a memory copy, the proof establishes the bounds, the validity of the pointers, the absence of overlap if required, the values written, and termination. For an X.509 parser, it adds encoded lengths, structure depths, integer conversions, and certificate errors. The SSTIC publication on an RTE-free X.509 parser shows the value of bounding the formats and separating decoding from business decisions [28]. The publication on a USB stack illustrates the composition of a proof over a large protocol stack, with states, buffers, callbacks, and transport errors [29].

The W $\oplus$ X property of an MPU is proved at two levels. At the software level, each region configuration is verified: a region has consistent rights, the bounds are aligned, and state transitions are authorized. At the binary level, the linker script and the ELF are inspected to confirm that executable sections are not writable and that data is not executable. The example provided by Sentry Kernel in the Cortex-M MPU profile can guide this approach, but the SoC's actual attributes must be verified locally.

The final dossier links requirements, contracts, code, proofs, tests, coverage, and binary artifacts. It states the demonstrated properties, the hardware assumptions, the excluded paths, the tools' limitations, and the residual risks. It is this traceability, more than the sole proof percentage, that makes the approach auditable.

Key takeaway

A useful proof is reproducible and bounded by its assumptions. The audit dossier must link requirements, contracts, code, tools, tests, proofs, and the delivered binary.

8.10 Critical functions and self-protections

CFI (Control-Flow Integrity) restricts control transfers to the targets expected by the call graph [1]. It reduces the possibilities of ROP and JOP attacks, but its effectiveness depends on the indirect calls, vector tables, callbacks, and exceptions actually declared [40, 8]. Compiler protections are checked on the binary, since a table kept by the linker script or an assembly transition can fall outside the automatic model.

A pseudo-random generator such as PGC32 (32-bit counter-based pseudo-random generator) can produce canary values, session identifiers, or non-secret masks from a seed provided by a hardware TRNG at startup. The seed is validated, protected against trivial values, and must not be confused with a full cryptographic source. The generator's counters and state are reset and saved according to a contract that prevents dangerous reuse.

Stack protection combines a non-executable, non-writable MPU region for domains that do not own it, a guard against overflow, a margin measurement, and possibly a canary. The W $\oplus$ X principle forbids the same region from being both writable and executable; the exceptions needed at startup are temporary, documented, and verified in the ELF. A canary detects a linear write that reached a sentinel value, but does not detect every corruption and does not replace a stack bound [15].

GCC provides stack-protection options and branch-hardening mechanisms whose availability depends on the version and the target. A violation must lead to a trap or a defined recovery path, without reusing a corrupted context. The options enabled, their limits, and their impact on real-time behavior are archived with the qualified configuration.

```
void __stack_chk_fail(void)
{
    __sys_exit(ERR_STACK_CORRUPTION);
    while (1) {
    };
}
/* _start do not have stack protection, SSP need to be initialized first */
void __attribute__((no_stack_protector, used, noreturn)) _start(uint32_t seed)
{
    int res;
    /* update SSP value with given seed */
    __stack_chk_guard = seed;

    /* main has stack protection active */
    res = main();
    /* inform kernel that the thread is finished */
    __sys_exit(res);
}
```

Listing 8.3: Activating SSP at thread startup

9

Functions required for secure embedded firmware

9.1 Event logs

Key takeaway

An embedded log must be bounded, verifiable after a power loss, and independent of the native representation of C structures. The CRC detects accidental corruption, but does not authenticate an attacker.

An embedded log records events in a fixed, bounded binary format. Field widths, endianness, version, CRC computation, and payload size are specified independently of the native representation of C structures. A record generally contains a type, a length, a sequence number, a monotonic time, a payload, and an integrity value. Unknown events are skipped according to their length in order to preserve compatibility between firmware versions.

A monotonic time measures the order and the duration since a reset or a reference point; it does not constitute a civil date. The sequence number distinguishes two events produced at the same tick and allows losses, repetitions, and recoveries after reset to be detected. A real-time clock can complement the record, but does not replace these counters for reconstructing a timeline.

```c
#include <stdint.h>

#define EVENT_PAYLOAD_MAX 16U

struct event_record {
  uint16_t format_version;
  uint16_t type;
  uint32_t sequence;
  uint32_t monotonic_ticks;
  uint16_t payload_length;
  uint8_t payload[EVENT_PAYLOAD_MAX];
  uint32_t crc32;
};
```

```c

static int event_record_is_valid(const struct event_record *record)
{
  if (record->format_version != 1U ||
      record->payload_length > EVENT_PAYLOAD_MAX) {
    return 0;
  }
  return crc32_record(record) == record->crc32;
}
```

Listing 9.1: Bounded binary format of a logged event

The CRC detects accidental storage and transport errors; it does not provide authentication against an attacker capable of rewriting memory. When the log must resist tampering, a MAC or a signature is added according to the threat model. Keys are not stored in every entry, and authentication errors are distinguished from accidental corruption.

In embedded systems, log management is often constrained to using a ring buffer. A ring buffer reserves a fixed number of records or sectors. The producer writes an entry, computes its integrity, and then publishes a validity marker last. The consumer skips an incomplete entry and looks for the last valid sequence. In Flash, sector erasure and the limited number of cycles impose wear-leveling: a monotonic generation distributes writes across sectors. A power loss during an erase or a program operation must leave at least one recoverable copy.

```c
struct journal_slot {
  uint32_t sequence;
  uint32_t payload;
  uint32_t crc32;
  uint32_t committed;
};

#define JOURNAL_COMMITTED 0xA5C39E71U

static int journal_append(uint32_t sequence, uint32_t payload)
{
  struct journal_slot slot = {
    .sequence = sequence,
    .payload = payload,
    .crc32 = crc32_bytes(&sequence, sizeof(sequence)) ^ payload,
    .committed = 0U
  };

  if (flash_slot_is_used(next_slot) ||
      flash_program(next_slot, &slot, sizeof(slot)) != 0) {
    return -1;
  }
  return flash_program_word(&next_slot->committed, JOURNAL_COMMITTED);
}
```

Listing 9.2: Atomic publication of a ring-buffer entry

This example illustrates a protocol and not a universal Flash implementation. The controller may impose writes by word, bit transitions in only one direction, and read delays after programming. Recovery checks the marker, the bounds, the CRC, and the sequence progression before making the entry visible.

9.2 Audit and analysis

Embedded auditing combines logs, fault counters, and a post-mortem state save. During an exception, a watchdog, or a critical assertion, the firmware captures the cause, the reset counter, the processor context, the instruction address, the state of the control registers, the stack limits, the version, and the fingerprint of the image. Buffers containing secrets are excluded, encrypted, or wiped according to the threat model.

The save is bounded and reentrant. A fault handler calls neither an allocator, nor an unreliable peripheral, nor a function that depends on an already corrupted stack. It writes a header with a length and a version, copies the available registers, computes an integrity value, and then publishes a validity marker last. If the capture fails, a minimal cause code and a reset counter are still kept.

```
struct crash_dump {
  uint32_t magic;
  uint16_t version;
  uint16_t size;
  uint32_t reset_count;
  uint32_t fault_cause;
  uintptr_t instruction_address;
  uintptr_t stack_pointer;
  uint32_t registers[16];
  uint32_t crc32;
  uint32_t committed;
};

static void crash_dump_commit(const struct exception_frame *frame,
                              uint32_t cause)
{
  struct crash_dump dump = {0};

  dump.magic = 0x43524153U;
  dump.version = 1U;
  dump.size = sizeof(dump);
  dump.reset_count = read_reset_count();
  dump.fault_cause = cause;
  dump.instruction_address = frame->pc;
  dump.stack_pointer = frame->sp;
  copy_registers(dump.registers, frame);
  dump.crc32 = crc32_bytes(&dump, offsetof(struct crash_dump, crc32));
  backup_store(&dump, sizeof(dump));
  backup_commit_word(CRASH_DUMP_COMMITTED);
}
```

Listing 9.3: Minimal post-mortem save frame

On restart, the code validates the magic, the version, the size, the CRC, and the marker before handing the dump over for analysis. It distinguishes a new dump from one already consumed, and prevents a reboot loop from indefinitely rewriting the same area. Post-mortem tools associate the fault address with the ELF and the symbols of the exact version; a simple textual version number is not enough.

Key takeaway

A post-mortem capture must be short, reentrant, and validated before analysis. It must neither depend on an allocator, nor wait for an unreliable peripheral, nor reuse a corrupted context.

9.3 Update process

Key takeaway

The security of firmware is a chain property: design, compilation, boot, execution, logging, updating, and post-mortem analysis must keep the same contracts and the same proofs.

A firmware update is a controlled transition between two images, not a simple file copy. The product defines the version, the format, the sizes, the Flash regions, the metadata, the progress states, and recovery after power loss. The H2Lab guide dedicated to a flip-flop update strategy completes these deployment and recovery requirements [19].

A two-image architecture reserves an active image and a candidate image, with separate metadata. The bootloader writes the candidate into an inactive area, checks its bounds and its integrity, and then publishes the *candidate-ready* state. After a reboot, it selects the candidate and requires an acknowledgment of successful boot. If the image does not boot, triggers a watchdog, or does not confirm its operation within the expected delay, the bootloader reverts to the previous image.

The update strategy checks at least:

- hardware compatibility and the bootloader's minimum version;
- the size, the load address, the alignment, and the reserved regions;
- the integrity of the image and, when the threat model requires it, its authenticity;
- version monotonicity and protection against reverting to a forbidden image;
- the consistency of the metadata after a power loss during erasing or programming;
- keeping the ELF, the link map, the SBOM, and the delivered fingerprint.

A differential update reduces the volume transferred but adds a reconstruction engine and new bounds proofs. It is only adopted if the reconstruction is deterministic, verifiable, and resumable. Transport secrets, authentication keys, and update logs follow a distinct lifecycle; a log must not become a means of silently modifying the boot policy.

After installation, the firmware checks the active version, the state of the peripherals, the partitions, and persistent data migrations. A migration must be transactional or reversible. Tests cover interrupted erasure, partial programming, metadata corruption, the watchdog, loss

of communication, and rollback. Delivery finally links the image to its build configuration, its proofs, and its fingerprint so that a field device can be tied to a controlled artifact.

Key takeaway

A robust update keeps a fallback image, publishes the candidate image only after verification, and reverts to the previous image if boot is not confirmed.

10
Conclusion

This guide has presented a complete approach for developing and delivering embedded firmware in C in a constrained or critical context. The common thread is the contract: the contract of the language, of the ABI, of the processor, of the SoC, of the peripherals, of the interfaces, and of the tools. Portability does not consist in denying the differences between ARM, RISC-V, and PowerPC, but in isolating them in documented layers and verifying the target-dependent properties.

Boot, memory, registers, caches, interrupts, and multi-core architectures have all been linked to the same requirement: every shared access must have an owner, a format, a bound, an order, and a defined error behavior. Atomics, barriers, mailboxes, DMA, MPUs, and schedulers are means of fulfilling these contracts; none of them alone replaces the analysis of the hardware and the generated code.

Security then rests on a traceable chain. Threats and assets are identified, components are inventoried in the SBOM, interfaces are reviewed, tests cover the decisions and the errors, and noRTE proofs make their assumptions explicit. Self-protection mechanisms, logs, post-mortem saves, and reversible updates extend this approach all the way to the product delivered and maintained in the field.

The practical value of this guide is to provide a reading grid usable at several levels: the developer can check a function and its bounds, the platform team can check the BSP, the memory map, and the barriers, the architect can examine the boundaries and the interference, the auditor can link requirements, code, tests, proofs, and binary. This continuity turns quality and security into verifiable properties rather than isolated intentions.

Serious qualification remains necessary for each product. The document's examples are reasoning models and starting points: they must be adapted to the SoC, the compiler, the hardware, the assurance level, and the threat model adopted. The method is nonetheless stable: reduce the assumptions, make the contracts explicit, measure the behaviors, prove what can be proved, test the rest, and keep the artifacts that make it possible to redo the analysis.

A
Best-practices checklist

This checklist turns the guide's recommendations into verifiable checkpoints. It replaces neither the risk analysis nor the product's own requirements. For each line, the project records the status, the owner, the associated proof, and, if necessary, the justification for a non-applicability.

Domain	Checkpoint	Status / proof
C language	The C profile, allowed extensions, and compiler options are versioned.	☐
C language	Types, conversions, sizes, and formats are suited to the ABI and the interfaces.	☐
C language	Undefined behaviors and conversions likely to lose information are analyzed.	☐
C language	Critical functions have explicit preconditions, postconditions, and error contracts.	☐
Complexity	Cyclomatic complexity and the counting convention are documented.	☐
Complexity	Decisions, error paths, and cleanups are covered by tests.	☐

Table A.1: Checklist for language, contracts, and testability.

Domain	Checkpoint	Status / proof
Memory	The memory map, reserved regions, cache attributes, and alignments are documented.	☐
Memory	ELF sections, the stack, the heap, DMA buffers, and vectors are placed by the controlled linker script.	☐
Memory	MMIO accesses use an explicit width and respect the peripheral's read/write effects.	☐
Memory	Cache operations and barriers are defined for each sharing domain.	☐
Memory	MPU/SMMU regions apply the expected rights and the W xor X policy.	☐
Boot	The BootROM, the bootloader, the reset vector, and the reset handler are documented.	☐
Boot	The initial stack, the vector tables, and the control registers are checked before C code runs.	☐
Boot	Transitions between images define the state of interrupts, peripherals, and the context.	☐

Table A.2: Checklist for memory, boot, and hardware protections.

Domain	Checkpoint	Status / proof
Concurrency	Every shared object has an owner, a format, and an access protocol.	☐
Concurrency	Races, TOCTOU, double-fetch, and reentrant callbacks are analyzed.	☐
Concurrency	Atomics, locks, IPIs, mailboxes, and FIFOs have a progress bound.	☐
Concurrency	ISRs do not block and context registers are preserved.	☐
Multi-core	Coherence domains, interrupt affinities, and inter-cluster accesses are documented.	☐
Multi-core	Buffers exchanged between homogeneous or heterogeneous cores have explicit ownership and lifetime.	☐
Scheduling	Priorities, affinities, blocking times, and inversion risks are analyzed.	☐

Table A.3: Checklist for concurrency, multi-core, and scheduling.

Domain	Checkpoint	Status / proof
Security	Assets, threats, boundaries, and security objectives are identified.	☐
Security	Privileges, responsibilities, and attack surfaces are minimized.	☐
Supply chain	Components, versions, licenses, provenances, and vulnerabilities are inventoried in the SBOM.	☐
Interfaces	Formats, sizes, errors, timeouts, ownership, and side effects are documented.	☐
Self-protection	CFI, canaries, stack protections, MPU, and trap mechanisms are checked on the binary.	☐
Tests	Unit, integration, on-target, fault, and coverage tests are archived.	☐
Proof	ACSL contracts, assumptions, obligations, exclusions, and noRTE results are reproducible.	☐

Table A.4: Checklist for security, supply chain, tests, and proofs.

Domain	Checkpoint	Status / proof
Logs	The binary format, the version, lengths, sequences, and monotonic time are specified.	☐
Logs	The CRC, the commit marker, the ring buffer, and wear-leveling withstand power loss.	☐
Post-mortem	The fault dump is bounded, reentrant, consistent, and tied to the exact ELF.	☐
Update	The candidate image is verified before activation and a fallback image remains available.	☐
Update	Metadata, migrations, watchdogs, anti-rollback, and rollbacks are tested.	☐
Delivery	ELF, link map, SBOM, proofs, fingerprints, and the programmed image are archived together.	☐

Table A.5: Checklist for logging, diagnostics, and updates.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security*, 2005.

[2] ANSI. Ansi x3.159-1989: Programming language c, 1989.

[3] ANSSI. Ebios risk manager, 2024.

[4] Arm Limited. Armv7-m architecture reference manual, 2023.

[5] Arm Limited. Armv8-m architecture reference manual, 2023.

[6] Patrick Baudin et al. Acsl: Ansi/iso c specification language, version 1.22, 2024.

[7] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, 1996.

[8] Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *ACM Symposium on Information, Computer and Communications Security*, 2011.

[9] BSI. It-grundschutz compendium, 2023.

[10] Camelot-OS. Sentry kernel: verified embedded kernel primitives, 2026.

[11] Camelot-OS. Shield: implementation of posix byte-order conversion functions, 2026.

[12] CEA LIST. Frama-c platform for static analysis of c code, 2024.

[13] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *USENIX Security Symposium*, 2002.

[14] CISA. The minimum elements for a software bill of materials, 2021.

[15] Crispin Cowan, Calton Pu, Dave Maier, Heather Walpole, Paul Bakke, Steve Beatties, Aaron Grier, Perry Wagle, and Qian Zhou. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[16] Ang Cui, M. Kataria, et al. Analyzing the impact of double-fetch bugs in the linux kernel. In *USENIX Security Symposium*, 2017.

[17] Ulrich Drepper. What every programmer should know about memory. Technical report, Red Hat, Inc., 2007.

[18] H2Lab. Nouveau guide technique: protection mémoire physique, 2026.

[19] H2Lab. Nouveau guide technique: stratégie de mise à jour flipflop, 2026.

[20] Richard W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, 1950.

[21] IBM. Powerpc book e architecture: A specification for enhanced embedded powerpc architecture, 2005.

[22] ISO/IEC. Iso/iec 9899:1990: Programming languages — c, 1990.

[23] ISO/IEC. Iso/iec 9899:1999: Programming languages — c, 1999.

[24] ISO/IEC. Iso/iec 9899:2011: Programming languages — c, 2011.

[25] ISO/IEC. Iso/iec 9899:2018: Programming languages — c, 2018.

[26] ISO/IEC. Iso/iec 9899:2024: Programming languages — c, 2024.

[27] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, first edition, 1978.

[28] Guillaume Le Guernic et al. Journey to a rte-free x.509 parser. SSTIC presentation, 2019.

[29] Guillaume Le Guernic et al. From cves to proof: make your usb device stack great again. SSTIC presentation, 2021.

[30] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[31] MITRE. Cwe-362: Concurrent execution using shared resource with improper synchronization, 2024.

[32] MITRE. Cwe-365: Race condition in switch, 2024.

[33] MITRE. Cwe-367: Time-of-check time-of-use race condition, 2024.

[34] NIST. Secure software development framework (ssdf) version 1.1, 2022.

[35] NIST. The nist cybersecurity framework 2.0, 2024.

[36] NXP Semiconductors. Mpc5777c reference manual, 2020.

[37] Raspberry Pi Ltd. Rp2350 datasheet, 2024.

[38] RISC-V International. Risc-v instruction set manual, volume i: Unprivileged architecture, 2024.

[39] RISC-V International. Risc-v instruction set manual, volume ii: Privileged architecture, 2024.

[40] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls. In *ACM Conference on Computer and Communications Security*, 2007.

License