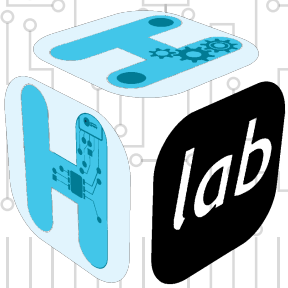
H2LAB-STIG-EMDEV
08-2026

Systèmes embarqués : Développement sécurisé de firmware bare metal en C



STIG H²Lab

Informations

Créée en 2020, H2Lab est une association loi 1901 dont l'objectif est de concevoir, développer et diffuser des solutions open-hardware et open-source dédiées à l'expérimentation autour des systèmes embarqués.
Ses principaux axes de travail sont :

- Conception de plateformes matérielles pour l'analyse hardware et software
- Développement d'alternatives ouvertes aux outils propriétaires, souvent opaques
- Publication de guides techniques et de recommandations pour promouvoir les bonnes pratiques en matière de sécurité, de confidentialité et de durabilité dans l'écosystème des objets connectés et des systèmes embarqués

Soutien aux chercheurs, enseignants et étudiants via la mise à disposition d'outils, de contenus pédagogiques et de formations.
Partenariats académiques pour encourager la mutualisation des ressources et des savoirs.

Historique des révisions

Version	**Date**	**Auteur**	**Modifications**
1.0	Août 2026	H2Lab	Version initiale du guide

Table des matières

1
Glossaire et acronymes

1.1 Glossaire

ABI. Contrat binaire : appel de fonction, registres, pile, alignement et format des objets entre unités compilées.

Barrière mémoire. Primitive qui contraint l'ordre observable de lectures et écritures entre processeurs, périphériques ou compilateur.

Comportement indéfini. Cas pour lequel ISO C ne fixe aucune exigence.

Bootloader. Logiciel de démarrage qui prépare et sélectionne une autre image firmware.

BootROM. Code de démarrage gravé dans un SoC ou un microcontrôleur par son fabricant.

Cache cohérent. Ensemble de caches dont le matériel maintient une vue compatible des lignes partagées.

Coprocesseur. Unité de calcul ou de contrôle distincte du coeur généraliste, par exemple une FPU, un DSP ou un accélérateur cryptographique.

DMA. Transfert de données réalisé par un périphérique sans intervention du coeur pour chaque mot.

Endianness. Ordre de représentation des octets d'un entier en mémoire.

Interconnect. Réseau matériel qui transporte et arbitre les transactions entre coeurs, mémoires et périphériques.

Mailbox. Périphérique ou zone de registres utilisée pour transmettre un message entre domaines.

Memory-mapped I/O. Accès aux registres d'un périphérique par des adresses de la carte mémoire.

noRTE. Absence démontrée d'erreur d'exécution dans un périmètre et sous des hypothèses documentées.

Ownership. Propriété temporaire d'une ressource, notamment d'un buffer partagé entre producteur et consommateur.

Race condition. Résultat dépendant d'un entrelacement non synchronisé.

Ring buffer. Buffer circulaire dont les positions de lecture et d'écriture progressent modulo une capacité fixée.

Scratchpad. Mémoire rapide gérée explicitement par le logiciel, sans politique de cache nécessairement automatique.

Sémaphore. Objet de synchronisation dont le compteur limite le nombre d'accès simultanés à une ressource.

Surface d'attaque. Interfaces physiques, logiques et organisationnelles exploitables par un attaquant [3, 35].

TOCTOU. Défaut "time of check to time of use".

Wear-leveling. Répartition des écritures afin de limiter l'usure différentielle d'une mémoire non volatile.

W ⊕ X. Politique qui interdit qu'une même région soit inscriptible et exécutable.

XIP. Exécution directe du code depuis une mémoire non volatile (*Execute In Place*).

1.2 Liste des acronymes

ABI. Application Binary Interface.

ACSL. ANSI/ISO C Specification Language.

ANSSI. Agence nationale de la sécurité des systèmes d'information.

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. Évaluation de valeurs par analyse statique.

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. Générateur pseudo-aléatoire à compteur de 32 bits.

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.

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 et périmètre

2.1 Objectif du guide

Ce guide fournit une méthodologie complète pour implémenter du code embarqué pour microcontrôleur en C permettant d'assurer une qualimétrie suffisante dans des contextes critiques. Ainsi, le but de ce document est d'assurer les éléments essentiels suivants :

- La portabilité et la testabilité du code embarqué et de ses modules
- la pérennité et la maintenabilité du code, permettant de réduire le coût de maintenance et d'évolution associé à la vie du projet
- la preuve d'absence de run time error (noRTE) sur l'ensemble du code
- la livraison, de manière efficace et sûre, les informations essentielles du logiciel, incluant sa SBOM (Software Bill of Materials), ses informations de production ou encore sa licence
- les performances en termes d'empreinte et de consommation
- la résilience aux attaques de type side channel, fault injection et supply chain

Ce guide se concentre exclusivement sur l'étape de développement et de production logicielle. Il intègre donc la problématique de gestion de configuration, de production et de livraison du logiciel mais ne traite pas de la problématique de gestion de projet ou d'intégration à un système complet incluant matériel et logiciel.

Ce guide se concentre sur le langage C dans un contexte embarqué contraint car correspond à la très grande majorité de l'état de l'art industriel. Ce guide n'a pas pour but de décrire comment architecturer un système d'exploitation embarqué ou produire un SDK, mais se concentre tout particulièrement sur le bon usage du langage C et les cas typique de mauvaise implémentation rendant complexes ou infaisables les objectifs cités ci-dessus.

2.2 Public visé

Le guide s'adresse :

- aux équipes de développement logiciel embarqué
- aux responsables de lot et aux architectes logiciels
- aux auditeurs techniques et pairs de relecture des développements

2.3 Périmètre technique et hypothèses

Le périmètre couvre les systèmes embarqués contraints de type microcontrôleur avec un ou plusieurs niveau(x) d'exécution. Le guide est applicable tant aux développements purement bare-metal (de type *super-loop* par exemple) qu'aux développement de type micro-noyaux embarqués ou applications critiques associées.

Le guide prend comme hypothèse que le logiciel cible est pensé dans une optique de réutilisation, afin d'assurer une portabilité et une maintenabilité maximale. Néanmoins, il s'applique tout autant à une implémentation dédiée à un besoin particulier.

Il faut noter que ce guide a pour but de fournir les clefs pour des implémentations critiques, nécessitant des démonstrations formelles, des qualification de niveau moyen à élevé et des conformités techniques strictes.

A retenir

Le guide suit un fil directeur unique : chaque propriété importante doit être exprimée par un contrat, isolée dans la bonne couche, vérifiée par analyse ou test, puis reliée à l'artefact effectivement livré.

3
Le langage C : rappels et évolutions

Ce chapitre est un rappel sur le langage C, ses évolutions et les notions essentielles à connaître pour assurer la portabilité et la maintenabilité du code embarqué.

3.1 Historique et standardisations du C

Le C naît au début des années 1970 chez Bell Labs, dans le contexte de la réécriture d'Unix sur PDP-11. La description de Kernighan et Ritchie, communément appelée K&R C, a longtemps joué le rôle de référence de fait [27]. Elle décrit un langage très proche de la machine, adapté aux systèmes contraints, mais dont les implémentations pouvaient diverger sur la taille des types, les conventions d'appel, le préprocesseur ou les bibliothèques. Le code K&R reste fréquent dans les vieux BSP (Board Support Package) et pilotes, il ne constitue cependant pas une base acceptable pour un développement nouveau, car les déclarations de fonctions à l'ancienne ne portent pas de prototype et laissent le compilateur appliquer des promotions implicites potentiellement incompatibles avec l'ABI (Application Binary Interface).

Le premier socle normatif est ANSI X3.159-1989, rapidement repris par ISO/IEC 9899 :1990. Les appellations C89 et C90 désignent, pour le développeur, le même langage de base [2, 22]. Cette normalisation fixe notamment les prototypes, les en-têtes de la bibliothèque standard, les qualificatifs `const` et `volatile`, ainsi que la distinction entre comportement défini, non spécifié, dépendant de l'implémentation et indéfini.

Pour l'embarqué, le point essentiel est que la norme ne fixe ni la taille de `int`, ni l'endianness, ni la représentation des pointeurs, ni l'ABI. Ces propriétés appartiennent à la cible et au compilateur, elles doivent être exprimées dans la couche architecture et non supposées par le code applicatif.

C99 constitue l'évolution la plus visible pour les firmwares portables [23]. Les en-têtes `<stdint.h>` et `<inttypes.h>` donnent accès aux types de largeur exacte, minimale et maximale, ainsi qu'aux macros de formatage correspondantes. Ils évitent de confondre une taille de protocole avec `int` ou `long`, dont la largeur varie entre ARM, RISC-V et PowerPC. L'en-tête `<stdbool.h>`

rend les intentions booléennes explicites. Les initialiseurs désignés facilitent la construction sûre de tables de vecteurs, de registres ou de structures versionnées ; `inline` permet d'exprimer une primitive courte sans imposer un appel de fonction.

```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 – Initialisation portable des registres UART

Il est important de noter que si les booléens ISO fournis par `<stdbool.h>` fournissent une aide à l'implémentation, ils ne garantissent pas la portabilité de l'ABI. Un booléen ISO peut être représenté par un octet, un mot ou un double mot selon la cible et le compilateur.
De plus, la conversion vers un type booléen produit une valeur logique zéro ou un, mais la représentation en mémoire et l'ABI restent propres à la cible. Un code qui transporte des états de sécurité dans un octet ou un mot doit donc définir explicitement son encodage et ne pas confondre valeur non nulle et état valide.

Les compilateurs récents peuvent proposer des extensions de vérification ou de durcissement des booléens, mais celles-ci restent propres à la chaîne utilisée et ne sont pas standardisées par l'ISO C. Il est donc recommandé de définir les états critiques dans des types et des contrats explicites, par exemple des énumérations dont les valeurs valides sont vérifiées, avec une distance de Hamming suffisante, selon le modèle de faute retenu, en cas de risques de faute avérée (EM ou glitch) dans le contexte du projet [20].

Certaines évolutions, comme le qualificatif `restrict`, sont risqués. Ce mot-clef est en effet une promesse d'absence d'aliasing faite au compilateur par le développeur : le violer produit un comportement indéfini et peut transformer une routine de copie correcte en code erroné après optimisation. De plus, `inline` ne garantit pas l'inlining et possède une sémantique de liaison subtile en C99 : une fonction `inline` publique nécessite une stratégie explicite de définition externe. Les tableaux de longueur variable, introduits par C99, peuvent consommer une pile bornée par une entrée non fiable ; ils sont à interdire dans le code de production critique. Enfin, les littéraux composés et les macros variadiques sont utiles, mais ne doivent pas masquer une durée de vie d'objet, une évaluation multiple ou un effet de bord.

C11 apporte les primitives décisives pour les systèmes concurrents [24]. `<stdatomic.h>` et le mot-clef `_Atomic` définissent les opérations atomiques et les ordres mémoire. Attention cependant, le mot clef `_Atomic` porte sur la variable associée et non sur l'opération en elle-même.
`_Static_assert` permet de refuser dès la compilation une configuration incompatible. Cet ajout est particulièrement utile en embarqué où un grand nombre d'information est calculable au moment de la compilation. Il permet ainsi de démontrer la conformité d'une configuration à l'ABI, aux contraintes de taille et d'alignement, ou encore à la disponibilité de certaines ressources.
`_Alignas` et `_Alignof` expriment des contraintes d'alignement importantes pour le DMA (Direct Memory Access), les registres et certaines ABI. L'ajout au standard ISO permet d'éviter l'usage des extentions (comme les extensions GNU), simplifiant la portabilité logicielle entre chaines de

compilations hétérogènes (GCC, ARM, IAR, etc.).

Enfin, les mots-clefs `_Thread_local` et `_Noreturn`, ainsi que les en-têtes `<threads.h>`, `<stdalign.h>`, `<stdnoreturn.h>` et `<uchar.h>`, complètent cette évolution. Dans un firmware bare metal, `<threads.h>` est souvent absent ou inadapté : sa disponibilité ne doit jamais être présumée. Il en va de même, sauf exception particulière, pour `<uchar.h>` qui est à éviter dans un firmware embarqué, car il ne fournit pas de garantie de portabilité sur l'ABI et la représentation des caractères.

L'ISO C11 fournit également la notion de `_Generic`, qui permet de sélectionner une expression en fonction de son type à la compilation. Efficace, cette notion doit néanmoins être utilisée avec parsimonie.

```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 – Signalisation atomique d'un événement

Une atomique C11 ne rend pas un protocole correct par magie. `volatile` ne remplace pas `_Atomic` ; un ordre `memory_order_relaxed` ne publie pas les données associées ; et une opération atomique n'est pas nécessairement lock-free. Avant un appel depuis une ISR (Interrupt Service Routine), vérifier la propriété `atomic_is_lock_free` sur la cible et étudier le code généré. Les primitives C11 ne remplacent pas non plus les barrières d'architecture et la maintenance de cache requises par un DMA ou un interconnect non cohérent.

C17 est une révision corrective de C11 : il ne modifie pas le modèle de programmation du firmware, mais clarifie et corrige le texte de la norme [25]. Son intérêt est donc principalement contractuel : une chaîne qualifiée C17 conserve les acquis de C11 sans imposer de migration fonctionnelle.

C23 modernise plusieurs aspects utiles à l'embarqué [26]. Il standardise notamment `nullptr`, les attributs normalisés tels que `[[nodiscard]]` et `[[maybe_unused]]`, `_BitInt`, l'inclusion binaire `#embed` et `<stdckdint.h>` pour l'arithmétique entière contrôlée.
Ces apports peuvent améliorer l'expressivité d'une interface, la vérification des retours d'erreur, l'intégration de ressources binaires ou la prévention des débordements. Leur emploi reste conditionné à la maturité du compilateur, de l'analyse statique et de la qualification associée.
Le choix d'une édition de C est donc une décision de configuration : le projet fixe son profil autorisé, interdit les extensions non justifiées et vérifie que chaque cible fournit effectivement les en-têtes et sémantiques employés.

3.2 Concept de la machine abstraite C

Un programme C est défini par une machine abstraite et non par une séquence d'instructions processeur. Le compilateur est libre de transformer l'implémentation selon le principe *as-if* : il

peut produire tout code dont le comportement observable est le même que celui du programme C ayant un comportement défini [24, 26]. Il peut ainsi propager une constante, supprimer un calcul dont le résultat n'est jamais utilisé, mettre une valeur en registre, fusionner des accès ou déplacer une écriture lorsque ces transformations ne changent pas l'observation autorisée par la norme.

Les entrées-sorties et les accès à un objet qualifié `volatile` font partie de ce comportement observable. Une lecture d'un registre MMIO (Memory-Mapped Input/Output) doit donc être déclarée `volatile` afin que le compilateur effectue l'accès demandé à chaque évaluation du code C. Sans ce qualificatif, il peut mémoriser la première valeur lue dans un registre, supprimer une lecture dont le résultat semble inutilisé ou sortir cette lecture d'une boucle. `volatile` est un contrat avec le compilateur, pas avec le matériel : il n'apporte ni atomicité, ni exclusion mutuelle, ni barrière mémoire pour les autres coeurs, ni maintenance de cache pour le DMA. Ces propriétés relèvent des atomiques C11, des instructions de barrière de l'architecture et du protocole d'accès au périphérique.

```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) {
    /* Chaque iteration relit le registre materiel. */
  }
}
```

Listing 3.3 – Différence entre état interne et registre matériel

Le mot-clef `static` ne rend pas un objet observable et ne le protège pas contre la concurrence. À portée de fichier, il donne une liaison interne : le nom n'est pas exporté et le compilateur sait qu'aucune autre unité de traduction conforme ne peut référencer directement `transmitted_bytes` ou `uart_status`.
À portée de bloc, il donne une durée de stockage statique : l'objet est unique et existe pendant toute l'exécution, mais son nom reste limité au bloc. Dans les deux cas, `static` ne signifie ni immuable, ni local au thread ; un objet `static` mutable partagé entre thread, ISR ou coeur nécessite une synchronisation explicite.

L'unité de traduction est le résultat du prétraitement d'un fichier source et de ses inclusions. Lors de sa compilation, l'optimiseur connaît les corps des fonctions et les objets à liaison interne de cette unité. Il peut donc supprimer une fonction `static` inutilisée, intégrer `uart_status` dans son appelant, conserver `transmitted_bytes` dans un registre entre deux points observables ou constater qu'une fonction `static const` est pure. À l'inverse, une fonction déclarée dans un autre objet de compilation est, sans LTO, une frontière d'information : le compilateur doit respecter son ABI et supposer les effets de bord déclarés ou possibles à travers son interface.

Cette limite n'est pas une barrière de sécurité. L'édition de liens, et éventuellement la LTO, peuvent ensuite analyser plusieurs unités de traduction et étendre les optimisations interprocédurales. Le développement embarqué doit donc démontrer la correction au niveau du C défini, puis vérifier le binaire pour les propriétés qui dépendent de la cible : accès MMIO de largeur imposée, barrières, séquence de démarrage, placement des sections et ABI. Se fier au désassemblage d'une version optimisée ne justifie jamais un programme comportant un comportement indéfini, une mise à jour de compilateur ou d'options peut alors légalement modifier le code généré.

3.3 les objets de compilation, le scoping et la notion de modèle de thread

La compilation séparée traite chaque fichier source, après expansion de ses en-têtes, comme une unité de traduction. Le compilateur produit un objet contenant du code, des données, des symboles définis et des symboles à résoudre. L'éditeur de liens assemble ensuite ces objets selon leur nom, leur liaison et leur ABI. Cette organisation permet de limiter la recompilation, mais elle ne garantit pas que deux déclarations externes décrivent le même type, le même alignement ou la même convention d'appel. Un en-tête public est donc un contrat : il est inclus par le propriétaire de l'implémentation et par tous ses appelants, puis contrôlé par la chaîne de compilation [24].

La portée définit où un nom est visible dans le texte source. Une variable locale a une portée de bloc ; une déclaration au niveau fichier a une portée de fichier. La liaison définit, elle, si le même nom peut désigner l'entité depuis une autre unité de traduction. Une fonction ou un objet déclaré sans `static` au niveau fichier a, en règle générale, une liaison externe. Le préfixe `static` donne une liaison interne à une fonction ou à un objet de fichier : son nom n'est pas exporté au lien. Il ne doit pas être confondu avec la durée de stockage statique, qui maintient un objet vivant pendant toute l'exécution, notamment pour une variable `static` locale.

Le code embarqué privilégie les fonctions `static` pour les détails d'un module et les fonctions publiques pour son interface. Cela réduit l'espace de noms global, empêche les appels directs à des primitives internes et donne à l'optimiseur une vision complète de leurs appelants possibles dans l'unité de traduction. Les objets globaux mutables à liaison externe sont à éviter : ils cachent une dépendance, empêchent de connaître leur propriétaire et compliquent les tests, la revue ainsi que la preuve d'absence d'interférence.

```
/* 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++;
```

```c
}

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

Listing 3.4 – Interface de module sans objet mutable exporté

`extern` est légitime pour déclarer une entité définie ailleurs ; les prototypes de fonctions dans un en-tête ont d'ailleurs une liaison externe par défaut. En revanche, un en-tête ne devrait presque jamais déclarer `extern` un objet mutable. Cette pratique fournit à tout appelant un accès en lecture et écriture sans protocole. Une fonction d'accès, une structure de contexte transmise explicitement ou un message vers le propriétaire expriment mieux l'autorité et les règles de concurrence. Lorsqu'un objet partagé est indispensable, une seule unité de traduction le définit, les autres le déclarent, et son invariant, son initialisation et sa synchronisation sont documentés.

Depuis C11, le modèle de threads définit des fils d'exécution abstraits, les objets atomiques et les relations *happens-before*. Deux threads qui accèdent au même objet non atomique, dont au moins un écrit, forment une course de données s'ils ne sont pas synchronisés, le comportement est alors indéfini. Un objet à durée de stockage statique est partagé par défaut entre les threads. `_Thread_local` donne au contraire une instance par thread, mais son support et son coût doivent être validés sur la cible. Les ISR, DMA et autres coeurs ne sont pas tous modélisés directement par `<threads.h>`, mais un firmware les traite néanmoins comme des contextes concurrents et définit les atomiques, verrous, barrières et protocoles matériels nécessaires.

3.4 Nombre cyclomatique et notion de couverture

Le nombre cyclomatique de McCabe mesure le nombre de chemins linéairement indépendants dans le graphe de contrôle d'une fonction. Pour un graphe connexe, il vaut $M = E - N + 2$, où E est le nombre d'arêtes et N le nombre de noeuds. Dans une fonction C structurée, une approximation pratique consiste à partir de 1 et à ajouter 1 pour chaque décision : `if`, `while`, `for`, `case`, condition ternaire et, selon la convention de l'outil, chaque opérateur booléen à court-circuit. La convention retenue doit être documentée, car elle modifie la valeur rapportée [30].

```c
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 – Complexité cyclomatique et décisions indépendantes

Cet exemple contient trois décisions si `frame == NULL || length < 2U` est considérée comme une seule décision, soit une complexité de 4. Si l'analyse compte les deux opérandes de `||` séparément, elle obtient 5. Cette valeur n'est pas une preuve de défaut ; elle indique le nombre minimal de cas de test à concevoir pour exercer des décisions indépendantes. Une valeur élevée signale surtout une fonction difficile à relire, modifier et tester. En code critique, découper une fonction par responsabilité est généralement préférable à accumuler les exceptions et les conditions imbriquées.

La couverture mesure ce qu'une campagne de test a effectivement exécuté. La couverture d'instructions confirme que chaque instruction a été exécutée au moins une fois. La couverture de branches confirme que chaque issue d'une décision a été exercée. La couverture de conditions vérifie chaque condition élémentaire ; MC/DC démontre en plus que chaque condition peut modifier indépendamment la décision. La couverture de chemins exhaustive est rapidement irréaliste dès que des boucles, des erreurs et des entrées variables sont présents. Pour une fonction critique, sélectionner des chemins bornés et justifier les chemins non atteignables est plus utile que revendiquer une couverture globale sans analyse des résultats.

Dans l'exemple précédent, le code, assez basique, s'autorise plusieurs points de sortie. Habituellement, pour des raisons de sécurité et de testabilité, chaque fonction ne doit comporter qu'un seul point de sortie.
Il est intéressant de constater que dans le cas particulier du traitement d'erreur dans une fonction, l'usage du goto peut être un facilitateur de la couverture et réduire la complexité cyclomatique. Il permet de plus d'accroitre la lisibilité tout en assurant le principe de point de sortie unique.

Un point de sortie unique concentre les actions qui doivent être exécutées quelle que soit l'issue de la fonction : restauration d'un invariant, libération de ressources, sortie d'une section critique, effacement d'un tampon sensible ou journalisation du résultat. Les cas de test vérifient alors un seul épilogue, au lieu de devoir prouver que chaque `return` intermédiaire a exécuté le bon sous-ensemble de nettoyages. Il ne faut pas transformer cette règle en dogme : un retour anticipé est acceptable pour une précondition sans effet de bord. En revanche, dès qu'une fonction acquiert plusieurs ressources, les sorties dispersées sont une source fréquente d'oubli de libération.

Le `goto err` est approprié lorsqu'il ne saute que vers des étiquettes de nettoyage situées à la fin de la fonction. Chaque étiquette libère une ressource déjà acquise, dans l'ordre inverse de son acquisition, puis rejoint l'épilogue commun. Ce schéma évite les conditions imbriquées et les duplications de code ; il ne doit pas servir à sauter vers le coeur de l'algorithme ni à contourner l'initialisation d'une 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 – Nettoyage centralisé avec goto err

Dans cet exemple, tous les chemins rejoignent un retour unique. Le libellé `err_release_buffer` documente précisément l'état atteint et garantit que la ressource est libérée une fois, y compris après une erreur de copie ou de démarrage DMA. Les tests doivent toutefois couvrir chaque saut et vérifier que les fonctions de libération sont sûres après les erreurs prévues par leur contrat.

Sans goto, une telle fonction, devant également assurer l'usage d'un seul point de sortie, ressemblerait à au code décrit dans l'exemple suivant, avec des conditions imbriquées et un code de libération dupliqué. La complexité cyclomatique est plus élevée, la lisibilité moindre et la couverture plus difficile à atteindre.
On passe alors, dans ce cas, d'un nombre cyclomatique de 4 à 6.

A retenir

La complexité cyclomatique guide la conception des tests, mais ne constitue pas une preuve de correction. Les chemins d'erreur, les ressources acquises et l'épilogue de nettoyage doivent être testés explicitement.

```
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      if (dma_buffer_copy(buffer, frame, length) == 0) {
8        if (dma_start_transfer(buffer) == 0) {
9          status = 0;
10       }
11     }
12     dma_buffer_release(buffer);
13   }
14   return status;
15 }
```

Listing 3.7 – Nettoyage centralisé sans goto

`gcov` collecte la couverture à partir d'un binaire instrumenté par GCC ; `gcovr` agrège ces données et produit des rapports exploitables en intégration continue. Ces résultats sont valides seulement pour le binaire, les options de compilation, la cible et les scénarios réellement exécutés. Pour une cible sans système de fichiers, les compteurs peuvent être récupérés par liaison série, semihosting ou mémoire réservée ; ce mécanisme de collecte ne doit pas modifier le comportement temps réel du test au point d'invalider ses résultats.

L'analyse statique est complémentaire et non une mesure de couverture. Des outils tels que CodeSonar recherchent des défauts de flux de données et des comportements dangereux ; SonarQube centralise notamment règles, alertes et indicateurs de qualité. Les alertes doivent être triées, justifiées ou corrigées, et non simplement comptées. Enfin, `gprof` est un outil de profilage : il estime le coût des fonctions et de leurs appels, mais ne prouve ni couverture ni absence d'erreur.

Les compteurs de couverture, l'analyse statique et le profilage répondent donc à trois questions distinctes : “qu'est-ce qui a été exécuté ?”, “quels défauts sont possibles ?” et “où le temps est-il consommé ?”.

3.5 portabilité en C

La portabilité C ne signifie pas que le même binaire fonctionne sur toutes les cibles. Elle signifie que le même code source, associé à une couche d'adaptation documentée, conserve la même sémantique lorsqu'il est recompilé. ISO C fixe les bornes minimales des types et les opérations du langage, mais laisse l'implémentation choisir la taille, l'alignement, la représentation en mémoire, l'endianness et l'ABI [24, 26]. Un projet portable distingue donc le code indépendant de la cible des contrats propres au processeur, au compilateur, au script de lien et aux périphériques.

Les types `char`, `short`, `int`, `long` et `long long` ne doivent pas porter une largeur de protocole, de registre ou de stockage persistante. Par exemple, `int` peut être sur 16, 32 ou 64 bits ; `long` vaut couramment 32 bits dans certaines ABI ARM et PowerPC, mais 64 bits dans une ABI RISC-V LP64. La signature d'une fonction, la disposition d'une structure et le calcul d'un débordement changent alors si ces types sont employés comme substituts de tailles précises. De même, `char` est toujours l'unité adressable du C, mais le signe associé n'est pas défini par la norme : il peut être signé ou non selon l'implémentation. Le code portable ne doit jamais supposer que `char` est signé ou non-signé.

Depuis C99, `<stdint.h>` fournit les types adaptés aux contrats numériques. `uint32_t` et `int32_t` expriment une largeur exacte de 32 bits et ne sont définis que si la cible peut les fournir sans bits de bourrage. `uint_least32_t` exprime au moins 32 bits ; `uint_fast32_t` privilégie la vitesse, non le format de stockage. Employer `size_t` pour une taille ou un indice d'objet, `ptrdiff_t` pour une différence de pointeurs et `uintptr_t` uniquement lorsqu'une adresse doit être conservée sous forme entière. Une conversion pointeur–entier ne rend pas une adresse valide sur une autre cible et ne doit jamais servir à contourner les règles d'aliasing. Le type `size_t` est le seul entier garanti pour contenir la taille d'un objet, il ne doit pas être utilisé pour des valeurs négatives.

L'en-tête `<inttypes.h>` fournit les macros `PRIu32`, `PRIxPTR` et assimilées pour formater ces types sans faire d'hypothèse sur `printf`. Les formats `%d`, `%ld` ou `%x` appliqués à un type de largeur exacte peuvent provoquer un comportement indéfini lorsque l'argument variadique ne correspond pas au format attendu. Dans un firmware, les mêmes règles s'appliquent aux journaux binaires et aux interfaces de diagnostic, même lorsqu'elles n'utilisent pas la bibliothèque standard.

Il est important de noter que l'usage des fonction de la famille `<stdio.h>` est globalement à proscrire dans un firmware embarqué. Ces fonctions sont en effet très consommatrices de ressources et ne sont pas adaptées à un usage embarqué. Il est préférable d'utiliser des fonctions de formatage plus légères, ou d'implémenter des fonctions de formatage plus simples, spécifiques à l'usage du firmware embarqué.
De plus, les fonctions a arguments variadiques sont à proscrire dans un firmware embarqué, car elles ne permettent pas de vérifier la correspondance entre le format et les arguments passés aisément. Des compilateurs comme GCC permettent d'ajouter des attributs de vérification de format, mais cela ne garantit pas la sécurité de l'usage de ces fonctions.
Enfin les fonctions variadiques sont problématiques à démontrer en terme de noRTE.

```
1 #include <limits.h>
2 #include <stdint.h>
3
```

```c
_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 – Décodage portable d'une valeur 32 bits au format réseau

Cet exemple construit la valeur à partir d'octets du protocole ; il ne dépend ni de l'endianness du processeur ni de l'alignement de l'entrée. À l'inverse, convertir directement un pointeur `uint8_t *` en `uint32_t *` puis le déréférencer peut violer l'alignement et l'aliasing strict, et aller jusqu'à provoquer une exception du processeur.

Les assertions de compilation constituent le dernier niveau du contrat de portage. Elles doivent vérifier la largeur et l'alignement exigés par une interface, tandis que le BSP vérifie au démarrage les propriétés dynamiques : carte mémoire, présence d'un périphérique ou cohérence de cache. Les accès MMIO restent confinés dans des primitives de largeur explicite et dans des types définis par le constructeur du composant. Cette séparation permet de partager le code métier entre ARM, PowerPC et RISC-V sans masquer les écarts réels de leurs ABI et de leur matériel.

4
Undefined Behavior et Run Time Error

4.1 Rappels et définitions

Le comportement indéfini, ou *undefined behavior* (UB), désigne un cas pour lequel ISO C n'impose aucune exigence à l'implémentation [24]. Dès que le programme l'atteint, le compilateur peut produire tout comportement, y compris un code apparemment correct pour une version donnée puis erroné après optimisation. Un dépassement d'entier signé, un accès hors limites, une division par zéro, un décalage invalide, le déréférencement d'un pointeur invalide ou une course de données sont des UB courants dans un firmware. Certains UB sont détectés par le processeur et peuvent provoquer une exception, comme la division par zéro.

Une *run-time error* (RTE, Run-Time Error) est une erreur susceptible de se produire à l'exécution et que l'analyse ou le test cherche à exclure dans le périmètre du projet. Une conversion non signée est définie par ISO C, mais peut provoquer une perte d'information ou une violation de contrat applicatif ; elle devient alors une RTE à traiter. Les RTE et les UB sont donc deux catégories distinctes, même si une RTE peut conduire à un UB si le contrat n'est pas contrôlé.

De plus, les comportements dépendants de l'implémentation ne sont pas nécessairement des erreurs, mais ils doivent être recensés. Le signe de `char`, la taille des types fondamentaux, l'endianness, l'alignement, la représentation des pointeurs et la convention d'appel dépendent de la cible, du compilateur ou de l'ABI. Un projet embarqué les fige dans sa configuration, les vérifie par assertions et les isole dans le BSP. Une option d'optimisation, une version de compilateur ou une extension GNU ne doit jamais être considérée comme une propriété de l'ISO C.

A retenir

Un comportement indéfini n'est jamais rendu acceptable par un binaire qui semble fonctionner. Le code C doit rester défini ; le désassemblage, l'ELF et la carte de lien servent ensuite à vérifier les propriétés propres à la cible.

4.2 Le modèle mémoire et la portabilité

Le modèle mémoire C définit les objets, leur durée de vie, leur représentation et les règles d'accès ; il ne décrit pas la carte mémoire physique du microcontrôleur. Une adresse C ordinaire ne désigne un objet que si l'objet existe, est correctement aligné et est accédé avec un type compatible. Le modèle de threads C11 ajoute les opérations atomiques et les relations d'ordre entre threads. Il ne remplace ni les barrières de l'ISA, ni la cohérence de cache, ni les règles du DMA.
Il est critique de bien comprendre que les compilateurs, tout langage confondu, ne traitent la protection mémoire que du point de vue de l'exécutable, sur la base du modèle de thread associé au langage lorsqu'il existe. Ils ne **peuvent pas** garantir la cohérence de cache, la visibilité d'une écriture vers un périphérique comme le DMA, la variation de valeur d'un registre matériel entre deux lectures, etc.

Un portage entre ARM, PowerPC et RISC-V doit donc séparer trois niveaux. Le code C exprime l'algorithme et ses préconditions ; la couche architecture traduit les atomiques et les barrières en instructions adaptées ; le BSP décrit les attributs de mémoire, le cache, le DMA et le MMIO. `volatile` impose un accès observable au compilateur, mais ne publie pas une donnée entre coeurs et ne rend pas une mise à jour atomique. L'employer pour un registre matériel n'autorise pas son usage comme verrou ou comme substitut à `_Atomic`.

La portabilité exige aussi de ne pas déduire les propriétés du binaire à partir du seul C source. Une lecture non alignée peut réussir sur une cible, être divisée en plusieurs accès sur une autre, ou lever une exception. De même, la visibilité d'une écriture vers une zone DMA dépend des attributs cache et d'une maintenance explicite. Les contrats de l'interface indiquent donc le contexte d'appel, l'alignement, la propriété des tampons et les barrières nécessaires avant toute optimisation locale.

A retenir

`volatile` rend un accès observable par le compilateur, mais ne fournit ni atomicité, ni verrou, ni cohérence de cache. Les accès MMIO et DMA exigent le protocole matériel, les barrières et la maintenance de cache appropriés.

4.3 La mémoire pour le C

Le modèle mémoire C est fondé sur les objets et leur type. Une zone de mémoire n'est pas une suite indifférenciée d'octets : son accès doit respecter la durée de vie de l'objet, son alignement et les règles de type compatible, souvent appelées *strict aliasing*. Accéder à un objet `uint32_t` par un pointeur `uint16_t *`, hors cas prévus par la norme, peut donc être un UB même si le processeur accepte physiquement la lecture. Le compilateur est alors autorisé à supposer que les deux pointeurs ne désignent pas le même objet.

La conversion entre un pointeur vers octets et un pointeur vers entier est une erreur fréquente dans les parseurs et pilotes. Elle peut cumuler trois défauts :

- alignement insuffisant

- aliasing invalide
- endianness implicite

Lire les octets un à un, ou copier dans un objet correctement aligné puis convertir explicitement l'endianness, conserve une sémantique définie. Les fonctions de copie doivent elles-mêmes respecter la validité et la non-superposition de leurs zones, sauf contrat de type `memmove`.

```
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 – Lecture sûre d'un entier 32 bits non aligné

L'exemple suppose explicitement un encodage similaire entre la donnée et l'endianess du coeur processeur sur lequel s'exécute l'algorithme, ce qui peut être faux si la donnée vient de l'extérieur. Sur une données propre à l'exécutable, cet algorithme est sûr : il ne déréférence jamais un pointeur invalide, ne lit pas hors limites et ne viole pas l'aliasing strict.
De plus, il est aussi portable : il évite le déréférencement potentiellement mal aligné de `(const uint32_t *)input`. Les analyseurs RTE et EVA peuvent vérifier les bornes, les décalages et la validité des pointeurs si les préconditions de longueur et de validité sont exprimées dans les appels ou les contrats ACSL.

4.4 Notion d'encodage de la donnée

Toute donnée externe est associée à un encodage, et non une représentation C native. Un protocole doit définir largeur, ordre des octets, ordre des bits si nécessaire, valeurs admises, alignement et longueur. Les conversions `htons`, `ntohs`, `htonl` et `ntohl` transforment entre l'ordre hôte et l'ordre réseau big-endian. Elles doivent être appliquées aux frontières du système, jamais dispersées dans le code métier. Cela rend les structures internes indépendantes de l'endianness de l'ARM, du PowerPC ou du RISC-V visé.

Les compilateurs modernes proposent souvent `__builtin_bswap16` et `__builtin_bswap32`. Lorsque la cible possède une instruction dédiée, le compilateur la sélectionne généralement ; sinon il optimise les masques et décalages. Le test du builtin doit rester isolé dans une couche de compatibilité. Un repli C pur est nécessaire pour les compilateurs qui ne le fournissent pas et doit employer des entiers non signés afin d'éviter les décalages ou dépassements signés. L'implémentation de Camelot-OS Shield suit ce principe pour ses conversions POSIX [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))

```

```
12
13 uint32_t htonl(uint32_t x)
14 {
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 }
```

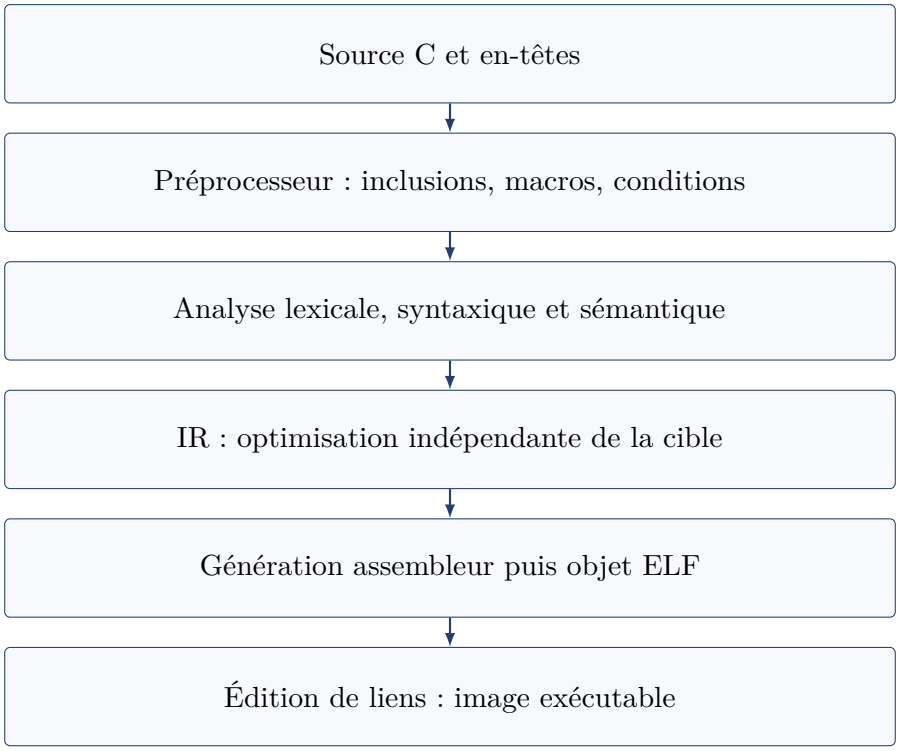
Listing 4.2 – Conversion 32 bits hôte vers réseau avec repli portable

L'endianess d'une plateforme est normalement reconnaissable dans du code C portable. Bien que l'ISO C ne définisse pas l'endianness, les macros du compilateur ou la configuration validée du BSP permettent de la déterminer. L'implémentation de Camelot-OS Shield utilise la macro `PLATFORM_LITTLE_ENDIAN`, qui est définie dans le BSP et vérifiée par des assertions de compilation.
Gcc et Clang définissent `__BYTE_ORDER__` et `__ORDER_LITTLE_ENDIAN__` pour identifier l'endianness. Le code applicatif ne doit pas tenter de la déduire à l'exécution.
Dans cet exemple, l'endianess est basé sur ces macros.
Le builtin ne change pas la sémantique : il ne fait qu'offrir une implémentation efficace du même échange d'octets. Pour un format 16 bits, appliquer le même contrat avec `__builtin_bswap16` et un repli de deux octets.

5
Du C au binaire

5.1 Rappel sur les compilateurs

Le compilateur ne traduit pas le C ligne à ligne. Il construit une représentation du programme, y applique les transformations autorisées par le langage, puis produit un objet conforme à l'ABI de la cible. Dans un firmware, la configuration de cette chaîne fait donc partie du produit : version du compilateur, drapeaux, en-têtes, script de lien, bibliothèques et options d'optimisation doivent être archivés avec le binaire. Une même source peut engendrer des comportements différents si l'ABI, les extensions activées ou les propriétés documentées du processeur changent.



Le préprocesseur normalise d'abord l'unité de traduction : il substitue les macros, résout les inclusions et élimine les branches conditionnelles. Il ne connaît ni types ni portée des identifiants ; une macro ne constitue donc pas une fonction sûre. Le front-end analyse ensuite les tokens, les déclarations, les types et les contraintes du langage. Il produit une représentation intermédiaire

(IR), sur laquelle le compilateur propage les constantes, élimine le code mort, simplifie les branches et réordonne les opérations lorsque le principe *as-if* le permet. Les erreurs de type, les UB et les accès `volatile` déterminent les limites de ces transformations.
Sur la base de la notion de *as-if*, le compilateur peut réorganiser ou supprimer des opérations lorsque leurs effets observables sont préservés. Il peut notamment éliminer un appel, une branche ou une variable locale dont aucun effet observable ne subsiste, y compris dans une fonction `static` connue dans l'unité de traduction.
Il est important de bien comprendre ce principe, car il est à l'origine de l'apparition des marqueurs *optimized-out* et des marqueurs *anonymous* dans les debogueurs, rendant l'analyse plus complexe.

Le générateur de code abaisse l'IR vers l'ISA visée et respecte son ABI. À l'entrée d'une fonction ordinaire, le prologue réserve la pile, sauvegarde les registres que l'ABI impose de préserver et établit, si besoin, un pointeur de frame. L'épilogue restaure cet état puis retourne vers l'appelant. Les détails varient entre AAPCS, RISC-V psABI et EABI PowerPC, mais l'appelant et l'appelé doivent toujours partager la même convention, que l'on appelle la *convention d'appel*.
Par défaut, toutes les fonctions C sont compilées avec un prologue et un épilogue.

```
/* Exemple indicatif de prologue et d'epilogue Cortex-M. */
example:
  push {r4, lr}
  sub  sp, sp, #16
  /* Corps de fonction. */
  add  sp, sp, #16
  pop  {r4, pc}
```

Une fonction qualifiée `naked` supprime tout ou partie de ce prologue et de cet épilogue. Elle est réservée au code d'amorçage, aux vecteurs ou aux changements de contexte dont l'assembleur est intégralement maîtrisé. Son corps ne doit pas contenir de C ordinaire : le compilateur peut utiliser la pile ou des registres sans que le cadre habituel existe. Cette extension est spécifique à la chaîne de compilation et doit être isolée dans la couche architecture.
Ces fonctions contiennent en général des instructions assembleur inline, qui ne sont pas portables et ne doivent pas être utilisées dans le code métier.

L'inlining remplace un appel par le corps de la fonction. Il peut diminuer le coût d'appel et donner davantage d'informations à l'optimiseur, mais augmente parfois la taille du binaire et la pression sur les registres. Le mot-clef C `inline` ne constitue pas une obligation d'inlining. Les attributs de type `always_inline` ou `forceinline` sont des extensions à employer uniquement après mesure du binaire et du temps d'exécution ; ils ne doivent pas compenser une interface ou une décomposition de module inadéquate.
De manière générale, le compilateur décide de l'inlining en fonction de la taille de la fonction, du nombre d'appels et des options d'optimisation. Ainsi, les options `-O0` et `-O1` favorisent la lisibilité du code généré, tandis que `-O2` et `-O3` favorisent l'inlining et la vitesse d'exécution. L'option `-Os` favorise quant à elle la taille du binaire, ce qui peut être critique dans un firmware embarqué.
Le compilateur, en fonction des optimisation activées peut également dérouler des boucles, vectoriser des opérations et réordonner des instructions.

```
static inline uint32_t set_bit(uint32_t value, uint32_t bit)
```

```
{
  return value | (UINT32_C(1) << bit);
}
```

Listing 5.1 – Primitive C candidate à l'inlining, sans obligation d'optimisation

Les attributs `pure` et `const` sont des promesses faites au compilateur. Dans les terminologies GCC et Clang, une fonction `pure` ne produit pas d'effet observable et son résultat peut dépendre de mémoire lisible ; une fonction `const` est encore plus restrictive. Les déclarer à tort permet de supprimer ou de fusionner un appel qui devait être exécuté. Une fonction lisant du MMIO, un compteur, une horloge, un état partagé ou un `volatile` ne doit jamais recevoir ces attributs.

Au lien, un symbole fort défini une fois fournit l'implémentation normale. Un symbole faible (`weak`) peut être remplacé par une définition forte, ce qui facilite une adaptation de BSP mais masque aussi les erreurs de configuration. Limiter les symboles faibles à des points d'extension explicitement documentés et vérifier la carte de lien empêche qu'une implémentation par défaut soit livrée par inadvertance. Les alias de symboles sont soumis à la même prudence : ils modifient la résolution au lien, pas les règles d'aliasing des pointeurs du langage C.

Enfin, l'optimiseur suppose que deux pointeurs de types incompatibles ne désignent pas le même objet et que le contrat `restrict`, lorsqu'il est présent, est respecté. Cette hypothèse d'aliasing autorise notamment le déplacement ou la vectorisation de lectures et d'écritures. Elle ne doit être invalidée ni par un cast de pointeur, ni par une zone MMIO, ni par un tampon partagé avec DMA. Le code C doit d'abord rester défini : le désassemblage et l'analyse du lien servent ensuite à vérifier les propriétés dépendantes de la cible, jamais à justifier un programme source incorrect.

5.2 Optimisation de l'empreinte mémoire

L'empreinte mémoire se mesure sur l'image réellement livrée, non sur la somme des fichiers objets. La carte de lien (`fichier .map`) indique la taille et l'adresse de chaque section, mais aussi les symboles qui les conservent en vie. Elle est donc l'artefact à comparer entre versions pour justifier une hausse de Flash, RAM ou pile. Une optimisation utile respecte le budget mémoire sans dégrader les contraintes de temps réel, la lisibilité, la capacité de diagnostic ou les mécanismes de protection.
Il est intéressant de noter que, en embarqué, le compilateur est apte à fournir la consommation maximale de pile pour chaque fonction, ainsi que la consommation totale de pile pour l'ensemble du programme. Ces informations sont très utiles pour dimensionner correctement la pile et éviter les dépassements de pile, qui sont des erreurs critiques dans un firmware embarqué.
Les options `-fstack-usage` et `-fstack-clash-protection` permettent de générer des informations sur l'utilisation de la pile et de détecter les dépassements de pile à l'exécution.

Les options `-ffunction-sections` et `-fdata-sections` placent chaque fonction et chaque objet de données dans une section distincte. Associées à `-Wl,-gc-sections`, elles permettent à l'éditeur de liens d'éliminer les sections non atteignables depuis les points d'entrée retenus. Cette technique réduit souvent fortement un firmware modulaire, mais impose de marquer explicitement comme utilisés les tables de vecteurs, callbacks indirects, sections d'initialisation et objets référencés uniquement par le script de lien. Vérifier la carte de lien après chaque activation : une

élimination silencieuse d'un handler est un défaut fonctionnel, pas une optimisation. On utilise alors l'attribut `__attribute__((used))` pour marquer les symboles qui doivent être conservés. Cet usage est aussi nécessaire pour les points d'entrée lorsqu'ils ne sont pas nommés sous la forme d'un symbole standard de point d'entrée du C comme `main` ou `_start`.

La LTO (*Link-Time Optimization*) transmet une représentation intermédiaire au lieu d'objets machine définitifs. Le compilateur peut alors supprimer des fonctions inter-modules, propager des constantes et spécialiser des appels au niveau du programme. Le gain peut être significatif, mais les frontières d'unités de traduction changent et les diagnostics, l'inlining ainsi que les symboles de débogage évoluent. Une cible critique active LTO dans une configuration reproductible, vérifie le binaire, la carte de lien et les résultats de test, puis requalifie cette configuration comme toute autre modification de compilateur.

La réduction d'empreinte passe d'abord par le modèle de données : types de largeur nécessaire, tables constantes en `.rodata`, buffers dimensionnés et absence de duplication. Éviter de désactiver une protection de pile ou les informations de débogage archivées pour gagner quelques octets. Les options de compression ou d'élimination sont acceptables seulement si la chaîne de livraison conserve l'ELF, les symboles et une preuve que l'image distribuée correspond aux sources contrôlées.

5.3 Le script de lien

Le script de lien transforme la carte mémoire matérielle en politique de placement. Il déclare les régions, fixe le point d'entrée et associe les sections aux adresses virtuelles ou physiques employées au démarrage. Il impose aussi les alignements requis par l'ABI, les caches, le DMA ou la MPU, et publie des symboles consommés par le code d'amorçage. Ce fichier est une source critique : il est versionné, relu et vérifié avec la carte de lien, au même titre que le C et l'assembleur.
Ce fichier permet aussi d'associer des sections à des secteurs physique dans la mémoire non-volatile. En effet, en général, les mémoires flash, présentes dans les microcontrôleurs, sont découpées en secteurs physiques. Ces secteurs sont les plus petits éléments que l'on peut effacer dans la mémoire flash. Il est donc important de bien dimensionner les sections de code et de données dans le script de lien afin d'éviter d'effacer des secteurs contenant des données critiques lors d'une mise à jour du firmware.

Dans une configuration XIP (*Execute In Place*), typique sur les microcontrôleurs sans abstraction mémoire, `.text` et `.rodata` résident et s'exécutent en NVM. `.data` possède une image initiale en NVM, mais son adresse d'exécution est en RAM afin d'être modifiable. `.bss` est uniquement réservée et mise à zéro en RAM. Les symboles `_sidata`, `_sdata`, `_edata`, `_sbss` et `_ebss` évitent de coder des adresses magiques dans le startup, et permette au code de démarrage d'assurer la recopie ou la zéroification de ces zones sur la base d'adresses spécifiées dans le script de lien.

```
1 ENTRY(Reset_Handler)
2
```

```
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 – Initialisation XIP de la RAM depuis les symboles du script de lien

Cette routine s'exécute avant l'usage d'objets C à durée de stockage statique dont l'initialisation exige RAM ou zéro. Le code de démarrage vérifie également la pile, la table des vecteurs et les régions de protection avant d'activer les interruptions. Après chaque changement de script, vérifier les limites des régions, l'alignement, les symboles d'entrée et les attributs lecture/écriture/exécution dans l'ELF final.

5.4 A propos des binaires

Une section est une organisation logique de l'ELF produite par le compilateur et le linker : `.text` contient en général le code, `.rodata` les constantes, `.data` les données initialisées et `.bss`

les données initialisées à zéro. Chaque section reçoit des attributs, notamment lecture, écriture, exécution, alignement et adresse. Un segment est une vue de chargement regroupant des sections ayant des attributs et un mode de chargement communs ; il intéresse surtout le chargeur et le débogueur. Les noms exacts, les sections d'initialisation et la politique de placement relèvent du script de lien et de l'ABI de la cible.

ELF est le format de travail principal d'un exécutable, au côté d'autres formats comme Mach-O (propre à macOS) ou PE (dans le monde Windows). Il conserve en-têtes, sections, segments, relocations, table des symboles et éventuellement informations de débogage DWARF. `readelf`, `objdump`, `nm` et la carte de lien permettent de vérifier le point d'entrée, les adresses, les symboles exportés et l'absence de section inscriptible et exécutable. Un débogueur tel que GDB exploite les symboles et DWARF pour associer adresses, fonctions, variables et lignes source ; les optimiser peut toutefois supprimer ou déplacer ces éléments visibles.

IHEX représente des enregistrements textuels d'adresse et de données, adaptés au transfert vers des programmateurs. Un binaire brut est une suite d'octets sans en-tête, sans symbole ni adresse intrinsèque : l'outil de flash doit connaître son adresse de chargement. Aucun de ces deux formats ne remplace l'ELF dans l'archive de production. Le projet livre le format attendu par la ROM ou le programmateur, mais archive aussi l'ELF, la carte de lien, les informations de version et leurs empreintes.
La conversion d'ELF vers IHEX ou binaire brut est triviale, mais elle ne doit pas être faite par le compilateur : elle est effectuée par un outil de post-traitement, souvent `objcopy`, qui est lui-même une fourniture de la chaîne de cross-compilation. Le firmware livré doit être vérifiable par comparaison d'empreintes avec le binaire archivé.
La conversion d'un ELF vers un binaire brut, prêt à être flashé, se fait avec le même outil. Les binaire bruts ne contiennent pas d'information d'adresse de base, à l'inverse du format IHEX. Il est donc nécessaire de connaître l'adresse de base du binaire brut pour le flasher correctement.

A retenir

L'ELF, la carte de lien et la configuration de construction sont des artefacts de référence. Le binaire brut ou l'IHEX destiné au programmateur ne doit jamais être le seul élément archivé.

6
Concepts de firmware bare metal

6.1 Rappel sur la notion de pile

La pile est une région mémoire utilisée pour les adresses de retour, les registres sauvegardés, les arguments temporaires et les variables locales. Le pointeur de pile doit rester aligné conformément à l'ABI à chaque point d'appel. Une *stack frame* est la portion de pile associée à un appel ; elle peut contenir un cadre de sauvegarde, des variables locales, des paramètres débordant les registres et un espace réservé aux appels descendants. L'optimisation peut supprimer une frame, fusionner des appels ou conserver une valeur dans un registre : la pile observée dans le désassemblage ne doit donc pas être déduite d'une règle C simplifiée.

Au reset, le processeur ne possède pas encore un environnement C complet. Sur un Cortex-M, le premier mot de la table de vecteurs fournit la valeur initiale pour la pile système (MSP) et le second fournit l'adresse de `Reset_Handler`. La pile doit déjà être placée dans une RAM accessible et correctement alignée. Le handler peut ensuite sélectionner la pile généraliste PSP (Process Stack Pointer) pour un contexte de thread, tandis que le MSP reste généralement réservé aux exceptions et au code privilégié.

Sur RISC-V, le reset vector et le code de démarrage doivent initialiser explicitement le registre `sp` ; l'ISA ne fournit pas une paire MSP/PSP équivalente au Cortex-M. Un système peut réserver une pile par hart, une pile d'exception et une pile par tâche, mais cette séparation relève du logiciel et du profil de privilège [39]. Le handler de trap doit sauvegarder les registres nécessaires avant d'appeler du C, puis restaurer `mepc` et les autres éléments de contexte avant `mret`.

Sur un PowerPC MPC5xxx, le reset handler initialise le registre de pile selon l'ABI PowerPC avant d'appeler une fonction C. Les exceptions utilisent un format de sauvegarde imposé par le profil Book E de PowerPC ; certains registres sont sauvegardés automatiquement ou dans un cadre spécifique, tandis que les autres doivent l'être par le handler. Une pile d'exception dédiée est préférable lorsque le contexte interrompu peut être corrompu ou lorsque la pile du thread n'est pas accessible dans le mode courant.

6.1.1 Construction et contrôle de la pile au démarrage

Le script de lien réserve une région de pile et publie ses limites. Le code de démarrage place le pointeur au sommet de cette région, en tenant compte du sens de croissance et de l'alignement de l'ABI. Une garde MPU ou une région non accessible peut détecter un dépassement, mais elle ne remplace pas le dimensionnement. Les usages maximaux doivent être estimés à partir de l'appel le plus profond, des handlers imbriqués, des chemins d'erreur et des éventuels contextes de coprocesseurs.

```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;

  /* L'ABI exige ici un alignement de 8 octets. */
  top &= ~(uintptr_t)7U;
  if ((top <= bottom) || ((top - bottom) < 256U)) {
    for (;;) {
    }
  }

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

Listing 6.1 – Initialisation contrôlée d'une pile avant le code C

Cet exemple montre le contrat, mais l'instruction d'écriture de `sp` est propre à l'architecture et doit être remplacée par la séquence prévue pour la cible. La borne minimale de 256 octets est illustrative : le projet doit la calculer selon ses appels et ses handlers. Le démarrage vérifie également que la région est dans une RAM utilisable et que son placement n'entre pas en collision avec les données, le tas, les buffers DMA ou les piles des autres coeurs.

A retenir

La pile est une ressource bornée et architecturale : son adresse, son alignement, sa taille, sa garde et ses collisions avec les autres régions doivent être vérifiés avant l'exécution du code C.

6.1.2 Sauvegarde de contexte et piles d'interruption

Lorsqu'une interruption survient, le matériel ou le handler sauvegarde le minimum nécessaire pour reprendre l'exécution interrompue. Le reste dépend du contrat de l'ISR. Une ISR qui modifie un registre callee-saved, un registre flottant ou un CSR doit le préserver, sauf si l'architecture définit une sauvegarde automatique ou si le contexte est explicitement transféré. L'ordre de sauvegarde et de restauration doit être symétrique et documenté ; une erreur sur la valeur restaurée de `pc`, du registre d'état ou de la pile peut retourner vers un niveau de privilège incorrect.

Le Cortex-M empile automatiquement un cadre matériel lors d'une exception, puis peut ajouter les registres flottants selon la configuration de la FPU. Le bit `EXC_RETURN` indique

notamment la pile utilisée pour reprendre le contexte. RISC-V laisse davantage de travail au pied d'interruption : le logiciel choisit la structure de cadre et doit préserver les registres requis par l'ABI et le niveau de privilège. PowerPC fournit un cadre d'exception lié au profil, mais le nombre de registres à sauver dépend du type d'exception et de l'implémentation.

Dans un système préemptif, chaque thread possède sa pile et son contexte sauvegardé, le changement de tâche doit aussi traiter les registres de coprocesseur possédés par le thread. Dans un système collaboratif, le contexte peut rester dans les registres jusqu'à un appel explicite, mais une interruption reste susceptible de l'interrompre. Dans tous les cas, les limites de pile sont surveillées par remplissage de motif, compteur de marge, garde MPU ou combinaison de ces mécanismes. La méthode de mesure et sa perturbation éventuelle du temps réel doivent être documentées.

A retenir

Une interruption doit sauvegarder et restaurer exactement le contexte qu'elle modifie. Les registres de contrôle, la FPU et la pile d'exception font partie du contrat de contexte.

6.2 Vecteur de reset et table des interruptions

Le *reset vector* est le mécanisme matériel qui fournit le premier point de contrôle après le reset. Le *reset handler* est le code exécuté à ce point ; il réalise habituellement l'initialisation très précoce avant d'appeler le point d'entrée métier. Une table de vecteurs associe ensuite une cause d'exception ou une interruption à une adresse de traitement. Elle doit être placée dans une région connue du matériel, alignée comme l'exige l'architecture et conservée par le script de lien, même si ses entrées ne sont pas référencées par un appel C ordinaire.
Les vecteurs d'interruption sont propres à chaque architecture. Ceux-ci doivent respecter les spécifications de la cible, notamment l'alignement, la taille et le format des entrées.

Sur un ARM Cortex-M, la table commence par la valeur initiale du Main Stack Pointer (pointeur de pile initial des architectures ARM), suivie de l'adresse du reset handler puis des exceptions et interruptions. Le matériel positionne la pile à l'adresse fournie et lit le second mot lors du reset. Sur ARM Cortex-M, l'architecture ARM spécifie les exceptions système strictement ordonnées et priorisée dans le tableau de vecteurs, suivies d'un nombre variable d'interruptions périphériques, dont le nombre et l'ordre sont définis par la cible [4, 5].

```c
#include <stdint.h>

/* symbole du script de lien */
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: pointeur de pile et adresse du reset handler */
  (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 – Table de vecteurs minimale pour un ARM Cortex-M

Le type exact des entrées, l'alignement et les attributs de section sont propres à la chaîne utilisée ; le script de lien doit placer `.isr_vector` à l'adresse attendue et utiliser `KEEP`. Sur les Cortex-M se configure via l'assignation du registe VTOR, ce qui donne la capacité de déplacer la table lors des passage de relai entre chaque firmware (bootloader, image métier, etc.).

```
#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;
  /* Le mode 1 demande le mode vectored; l'adresse doit etre alignee. */
  __asm__ volatile ("csrw mtvec, %0" : : "r"(address | 1U) : "memory");
}
```

Listing 6.3 – Configuration d'un handler de trap RISC-V

Le code assembleur de `riscv_trap_entry` doit préserver le contexte selon le contrat de la cible, lire `mcause`, `mepc` et `mtval` lorsque nécessaire, puis restaurer le contexte avec `mret`. La taille et l'alignement effectifs de la table dépendent de la microarchitecture et du nombre de causes prises en charge. L'instruction `csrw` est une extension d'architecture, non une opération ISO C : elle reste confinée à la couche architecture et est testée avec le désassemblage de l'image finale.

Sur PowerPC, il n'existe pas une table universelle commune à toutes les variantes. Les MPC5xxx de profil Book E utilisent en général un vecteur de reset à une adresse fixe et des registres IVPR/IVOR pour construire les adresses des exceptions et interruptions. Le vecteur matériel contient souvent une courte séquence d'assembleur qui sauvegarde l'état minimal puis branche vers un handler commun. D'autres variantes PowerPC utilisent des offsets ou des mécanismes de vecteurs différents ; le manuel du processeur prime donc sur tout exemple générique.

```
/* Le symbole est place par le script de lien a l'adresse de reset. */
__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)
{
  /* Initialiser l'etat machine, IVPR/IVOR et la pile avant le code C. */
  for (;;) {
  }
}
```

Listing 6.4 – Entrée de vecteur indicative pour un PowerPC Book E

Cet extrait illustre seulement la frontière entre le vecteur et le handler ; il ne constitue pas une séquence de démarrage complète. Le script de lien doit imposer l'adresse et la taille de `.reset_vector`, et le code d'initialisation doit configurer les registres de vecteurs conformément au SoC. Les handlers d'exception doivent aussi respecter le format de sauvegarde imposé par le processeur avant d'exécuter du C.

Une table de vecteurs est une donnée de démarrage, mais elle est aussi une interface avec le matériel et le script de lien. Les contrôles suivants font partie de la vérification du firmware :

- vérifier par assertions et par la carte de lien l'adresse, l'alignement et la taille de la table
- conserver la table et les handlers atteints uniquement par référence matérielle
- vérifier que la valeur initiale de la pile appartient à la région RAM autorisée et respecte l'alignement de l'ABI
- documenter, pour chaque handler, le contexte d'appel, les registres sauvegardés, l'état des interruptions et le comportement en cas d'événement inattendu
- tester les chemins de reset, d'exception, d'interruption et de retour depuis le bootloader sur le binaire réellement lié

La séquence de démarrage doit rester courte et déterministe. Elle ne doit pas utiliser un objet C global avant la copie de `.data` et la mise à zéro de `.bss`, ni activer un périphérique avant que son horloge, son adresse et ses règles d'accès soient établies. Le passage au point d'entée métier intervient seulement lorsque la pile, les sections mémoire, la table des vecteurs et les registres de contrôle nécessaires satisfont leurs contrats.

6.3 Registres, coprocesseurs et périphériques

Un registre est une zone de stockage de largeur et de sémantique définies par le processeur ou le périphérique. Il ne doit pas être confondu avec une variable C ordinaire : une lecture peut acquitter un événement, une écriture peut déclencher une action et certaines valeurs peuvent changer indépendamment du code exécuté. La documentation de la cible fixe pour chaque registre son adresse, sa largeur d'accès, son alignement, ses bits réservés, sa valeur après reset et les effets de lecture ou d'écriture. Ces propriétés font partie du contrat matériel du firmware.

Les registres généraux servent principalement à contenir des opérandes, des adresses et des résultats temporaires. Les registres spéciaux contrôlent l'état du processeur, la mémoire, les interruptions ou le flot d'exécution. Les registres de périphériques sont exposés dans la carte mémoire du SoC et pilotent une fonction externe : horloge, GPIO, UART, contrôleur DMA, timer ou contrôleur d'interruption. Enfin, un coprocesseur peut disposer de son propre jeu de registres et de bits d'activation. Une FPU traite les opérations flottantes ; une MPU contrôle

des régions mémoire ; un accélérateur cryptographique ou un DSP possède généralement une interface analogue, mais avec des contrats propres au composant.

6.3.1 Registres du processeur selon l'architecture

Sur un ARM Cortex-M, les registres généraux `R0` à `R12` sont utilisés par l'ABI pour les arguments, les résultats et les temporaires. `R13` est le Stack Pointer, `R14` le Link Register et `R15` le Program Counter. Le registre d'état `xPSR` regroupe notamment les indicateurs de calcul et l'état d'exception. `CONTROL` sélectionne le mode d'exécution et la pile utilisée, tandis que `PRIMASK`, `BASEPRI` et `FAULTMASK` contrôlent le masquage des exceptions. `VTOR` contient la base de la table des vecteurs lorsqu'il est implémenté. Les registres `MSP` et `PSP` permettent de distinguer la pile principale de la pile de processus.

Les extensions Cortex-M ajoutent des registres de contrôle tels que `CPACR` pour l'accès aux coprocesseurs et des registres MPU comme `RNR`, `RBAR` et `RLAR` sur les profils qui les fournissent. Les noms et la disponibilité varient selon le coeur : le code doit donc être conditionné par la configuration de la cible, et non seulement par la présence d'un en-tête CMSIS.

Sur RISC-V, les registres généraux `x0` à `x31` ont une largeur définie par l'ISA, avec `x0` toujours égal à zéro. Le registre `pc` n'est pas accessible comme un registre général ordinaire. Les CSR contrôlent l'état machine et les exceptions : `mstatus`, `mie`, `mip`, `mtvec`, `mepc`, `mcause` et `mtval` sont courants au niveau machine. Les modes privilégiés supplémentaires peuvent fournir `satp` pour la traduction d'adresses et des CSR spécifiques à l'implémentation. Les extensions `F` et `D` ajoutent des registres flottants et `fcsr` ; leur état doit être sauvegardé si le contexte d'exécution peut changer.

Sur un PowerPC MPC5xxx de profil Book E, les registres généraux `r0` à `r31` sont complétés par le registre de condition `CR`, le compteur de boucle `CTR`, le registre de retour `LR` et le registre d'exception `XER`. Le registre d'état machine `MSR` et les registres spéciaux `SPR` contrôlent notamment les exceptions, la mémoire et les mécanismes propres au coeur. Les registres `IVPR` et `IVOR` servent à établir les vecteurs sur les variantes qui les implémentent [21, 36]. Les MPC5xxx ne possèdent pas tous le même jeu de périphériques ou de coprocesseurs : le manuel du SoC et le fichier de configuration du BSP sont l'autorité pour la liste effective.

A retenir

Les noms et la disponibilité des registres dépendent de l'architecture et du SoC. Le code portable passe par un BSP validé et ne transforme pas une extension de processeur en garantie ISO C.

6.3.2 Initialisation des coprocesseurs et protections

Un coprocesseur ne doit pas être utilisé simplement parce que le compilateur sait produire son instruction. Le code de démarrage vérifie d'abord sa présence, son alimentation et son horloge, puis configure son accès, son état initial et les protections associées. Si l'équipement n'est pas disponible, le profil de compilation doit interdire les instructions correspondantes ou fournir un chemin de repli défini. L'état initial doit être déterministe : exceptions flottantes, registres de contrôle, état d'erreur, arrondis et éventuels tampons internes sont réinitialisés selon le contrat du composant. Des coprocesseurs typiques sont la FPU, la MPU et les accélérateurs vectoriels

comme SIMD ; leur disponibilité varie selon les architectures, les fabricants et les SoCs.

Sur un Cortex-M4/M7 doté d'une FPU, l'accès aux coprocesseurs est généralement autorisé par `CPACR` avant le premier usage d'une instruction flottante. L'activation doit être suivie de la synchronisation d'instructions requise par l'architecture. Dans un système préemptif, les registres flottants doivent également être intégrés à la sauvegarde et la restauration de contexte, ou leur utilisation doit être limitée à un contexte dont la propriété est explicitement contrôlée. Une ISR ne doit pas modifier implicitement l'état flottant d'un thread.

```c
#include <stdint.h>

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

static void fpu_enable(void)
{
  SCB_CPACR |= CPACR_CP10_CP11_FULL_ACCESS;
  /* Barriers maintain memory ordering and instruction visibility. */
  __asm__ volatile ("dsb" : : : "memory");
  __asm__ volatile ("isb" : : : "memory");
}
```

Listing 6.5 – Activation contrôlée de la FPU sur Cortex-M

Cet extrait est volontairement réduit : dans un produit, l'adresse de `CPACR` et les masques proviennent de la description validée du coeur, non d'une constante recopiée. L'opération de lecture-modification-écriture doit aussi être sûre lorsque le registre est modifié par un autre contexte. Une MPU ne s'active qu'après la définition de régions non chevauchantes, de leurs droits d'accès, de leur alignement et de la région par défaut. La pile, les tables de vecteurs et les registres MMIO doivent recevoir des attributs compatibles avec leurs usages ; une protection activée trop tôt peut empêcher le code de démarrage d'accéder à ses propres données. Pour une présentation dédiée aux objectifs de sécurité, aux scénarios d'attaque et aux critères de validation de la protection mémoire physique sur ARM, PowerPC et RISC-V, voir le guide H2Lab [18].

Sur RISC-V, l'activation de la FPU dépend de l'extension et du champ d'état flottant de `mstatus` ; une implémentation peut également fournir une unité vectorielle avec un état beaucoup plus volumineux. Le BSP doit définir les instructions disponibles, les CSR concernés et la politique de sauvegarde. Sur PowerPC, la présence d'une FPU, d'une unité SPE ou d'un contrôleur de protection dépend de la variante. Le bit d'accès dans `MSR`, les registres de contrôle et la séquence de synchronisation doivent être initialisés avec les instructions et les contraintes de la cible.

A retenir

Un coprocesseur doit être activé, initialisé et intégré au contrat de contexte avant son premier usage. La FPU, la MPU et les accélérateurs ne sont pas des options invisibles pour le scheduler ou les handlers.

6.3.3 Memory map et accès aux périphériques

La *memory map* décrit la correspondance entre les adresses et les ressources physiques : mémoire non volatile, SRAM, zones réservées au boot, mémoire de périphérique, DMA, registres du coeur et éventuelles zones partagées entre processeurs. Une adresse ne suffit pas à définir un

accès : il faut aussi connaître le type de mémoire, la largeur autorisée, l'alignement, les attributs de cache, la politique de sécurité et les barrières requises. Les régions réservées et les trous d'adressage ne doivent jamais être sondés pour détecter la présence d'un périphérique.

Les accès MMIO sont confinés dans le BSP et utilisent des types volatils de largeur explicite. Une structure de registres peut améliorer la lisibilité, mais elle ne doit pas supposer que le compilateur insère le padding attendu : les offsets sont vérifiés par des assertions et comparés au manuel du composant. Les champs réservés restent présents dans la structure ou sont traités par des offsets documentés. Un registre marqué *write-one-to-clear*, *read-to-clear*, *write-only* ou *read-only* ne peut pas être manipulé comme une variable ordinaire.

A retenir

Une adresse MMIO n'est pas une variable C. Sa largeur, son alignement, ses effets de lecture/écriture, ses bits réservés et ses attributs de cache doivent être définis par le manuel du périphérique et encapsulés dans le BSP.

On retrouve dans les implémentations deux modèles de manipulation des périphériques MMIO :

- assignation d'une structure à une adresse MMIO de périphérique. Cet usage se retrouve par exemple dans la HAL de STMicroelectronics
- manipulation des registres sur la base de primitives d'accès (`ioread()` et `iowrite()`), comme dans le noyau Linux. Ce modèle est plus sûr, car il permet d'ajouter des vérifications de largeur, d'alignement et de barrière mémoire et assure une bonne abstraction pour les drivers de périphériques

```c
#include <stdint.h>

struct uart_registers {
  volatile uint32_t control;
  volatile const uint32_t status;
  volatile uint32_t interrupt_clear;
  /* suite des registres */
};

#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 – Accès MMIO de largeur et de sémantique explicites, abstractions manquantes

```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. */
```

```
7     __asm__ volatile ("dmb" : : : "memory");
8     return res;
9   }
10
11  static inline void iowrite32(uintptr_t addr, uint32_t value)
12  {
13    *(volatile uint32_t *)addr = value;
14    /* Write barrier to guarantee access visibility. */
15    __asm__ volatile ("dmb" : : : "memory");
16  }
17
18  /* driver implementation, arch independent */
19  #define UART0 ((struct uart_registers *)0x40000000U)
20  #define UART0_CR_OFFSET 0x00U
21  #define UART_STATUS_TX_READY (1U << 7)
22
23  static void uart_clear_interrupt(uint32_t mask)
24  {
25      uint32_t val = ioread32((uintptr_t)UART0 + UART0_CR_OFFSET);
26      val |= mask;
27      iowrite32((uintptr_t)UART0 + UART0_CR_OFFSET, val);
28  }
```

Listing 6.7 – Accès MMIO par primitive d'accès registre

La qualification `volatile` impose au compilateur de conserver les accès observables ; elle ne garantit ni atomicité, ni ordre entre coeurs, ni cohérence avec un DMA. Une écriture vers un périphérique peut exiger une barrière de type `DMB` (Data Memory Barrier), une synchronisation de type `DSB` (Data Synchronization Barrier), une invalidation de cache ou une séquence de lecture de retour. Ces opérations sont définies dans la couche architecture et ne sont ajoutées qu'après lecture du manuel du périphérique. Une boucle d'attente doit par ailleurs avoir une condition de sortie ou une justification de blocage permanent ; sinon une panne d'horloge ou de périphérique peut provoquer un dépassement du délai de démarrage.
Il est important de noter que l'emploi de boucles d'attente active sur un changement de valeur dans un registre ne permet pas de garantir une couverture par analyse symbolique. En effet, le prouveur considère qu'une modification extérieur est non garantie, et considère la suite de la fonction comme non atteignable (violation de la garantie de terminaison). On utilise alors des compteurs de timeout, permettant de détecter tout problème de traitement des écritures ou lectures de registre sans jamais risquer un blocage définitif.

La lecture-modification-écriture est un risque fréquent. Elle est inadaptée aux registres dont certains bits sont réservés, acquittent une interruption à la lecture ou sont modifiés par le matériel. Pour un champ contrôlé par logiciel, le BSP conserve la valeur et écrit un masque complet lorsque le fabricant l'autorise. Pour un registre partagé avec une ISR ou un autre coeur, il faut employer le protocole matériel prévu : opération atomique, verrou, masques d'interruptions ou registre SET/CLR. Un simple objet `volatile` ne constitue pas un verrou.

Enfin, l'initialisation d'un périphérique suit une séquence contractuelle : activer son horloge, retirer son reset, configurer ses I/O et son mode, programmer ses limites, effacer les événements potentiellement liés à un usage antérieur, puis, au besoin, installer son handler et autoriser son ou ses interruption(s) une fois le controleeur d'interruption central convenablement configuré.

6.4 La hiérarchie mémoire

De manière générale, y compris sur les microcontrôleurs, la mémoire n'a pas un coût d'accès uniforme [17]. Le processeur accède d'abord aux registres, puis aux niveaux de cache, à la SRAM ou à la DRAM, et enfin aux mémoires non volatiles et aux périphériques. Chaque niveau possède une latence, une largeur de transfert, une persistance et des attributs de cohérence différents. Une optimisation correcte ne consiste donc pas seulement à rapprocher une donnée du coeur : elle doit aussi préserver les règles d'accès du périphérique, du DMA et des autres coeurs.

Les registres généraux sont les plus proches du coeur mais ne constituent pas une mémoire générale. Les caches contiennent des copies de lignes de mémoire et peuvent rendre une écriture invisible à un périphérique tant qu'elle n'a pas été nettoyée vers le niveau suivant. La SRAM interne offre généralement une latence déterministe et est adaptée aux piles, aux tampons courts et aux données temps réel. Une mémoire externe ou une Flash peut introduire des temps d'attente, des contraintes d'alignement et des opérations d'effacement incompatibles avec un chemin critique.

Un cache peut être séparé en cache d'instructions et cache de données, ou être unifié. La plupart du temps, le cache de premier niveau (cache L1) est spécialisé, séparant cache instruction et cache de données.
Le cache d'instructions accélère les lectures de code ; le cache de données accélère les lectures et écritures des objets. Lorsqu'une zone de code est écrite puis exécutée, le firmware doit rendre les nouvelles instructions visibles au cache d'instructions. Lorsqu'un tampon est rempli par le CPU puis consommé par un DMA, il doit nettoyer les lignes concernées avant de lancer le transfert. Lorsqu'un DMA remplit un tampon lu par le CPU, le firmware doit invalider les copies éventuellement présentes avant la lecture. Les opérations exactes dépendent de la cohérence matérielle et sont définies par le BSP ; une simple qualification `volatile` ne réalise aucune maintenance de cache.

Les mémoires *scratchpad* (mémoires tampons gérées explicitement) sont des SRAM explicitement gérées par le logiciel. Elles n'ont pas nécessairement de cache et offrent donc une latence plus prévisible, au prix d'une gestion manuelle du placement et de la capacité. Les *Tightly Coupled Memories* (TCM) sont reliées directement au coeur. Sur ARM Cortex-M, l'ITCM est destinée aux instructions et la DTCM aux données ; elles peuvent fournir un accès déterministe pour un handler, une pile ou un tampon critique. Elles ne sont toutefois pas toujours visibles par un contrôleur DMA ou par un autre coeur. Le script de lien, les attributs MPU et la documentation du SoC doivent donc confirmer qui peut accéder à chaque région.

```
/* Le script de lien associe .fast_data a la SRAM ou a la DTCM. */
__attribute__((section(".fast_data"), aligned(32)))
static uint8_t dma_staging_buffer[1024U];

/* Le code critique est place en ITCM lorsque la cible le permet. */
__attribute__((section(".fast_code")))
static void control_loop_step(void)
{
  /* Traitement a duree bornee et sans acces a une memoire externe. */
}
```

Listing 6.8 – Placement de données critiques dans une mémoire dédiée

Le placement ne suffit pas à établir une propriété temps réel. Il faut vérifier la taille des sections, leur adresse de chargement et d'exécution, l'initialisation des données, l'accès du DMA et la présence éventuelle de conflits avec la pile ou les vecteurs. Une section rapide mais saturée peut provoquer une erreur d'édition de liens ; une section non initialisée peut contenir une valeur résiduelle au reset.

6.4.1 Synchronisation et barrières mémoire

Une barrière mémoire ne transporte pas une donnée et ne rend pas une opération atomique. Elle impose seulement un ordre ou une visibilité à un acteur donné. Il faut donc distinguer au moins trois garanties :

- l'ordre observé par le compilateur
- l'ordre observé par le coeur
- l'ordre observé par les périphériques ou l'interconnect.

Une barrière C11 telle que `atomic_thread_fence` concerne le modèle mémoire du langage. Une instruction de barrière de l'ISA concerne le processeur ; une opération de maintenance de cache concerne la visibilité physique d'une ligne. Ces mécanismes peuvent être nécessaires simultanément.

Sur ARM, `DMB` ordonne les accès mémoire observables par le domaine indiqué, `DSB` attend l'achèvement des accès avant de poursuivre et `ISB` vide le flux d'instructions après une modification de contexte ou de contrôle. Sur RISC-V, `fence` ordonne les catégories de lectures et d'écritures précisées, tandis que les instructions atomiques et leurs bits d'ordre fournissent les garanties associées à une opération atomique. Sur PowerPC, les instructions de synchronisation et d'ordre mémoire dépendent du profil et du SoC, le BSP doit fournir une primitive nommée selon sa sémantique plutôt que d'exposer des instructions dans le code métier.

```
/**
 * @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");
}
```

```
25
26 /**
27  * @brief Processor instruction synchronisation barrier
28  *
29  * This barrier will discard all further prefetched instruction
30  * This is needed after enabling an isr, context task switch, etc.
31  */
32 inline __attribute__((always_inline)) void arch_inst_sync_barrier(void) {
33     asm volatile ("isb 0xf" ::: "memory");
34 }
```

Listing 6.9 – Abstraction des barrières mémoire pour ARM Cortex-M

Une séquence producteur-consommateur illustre la distinction. Le producteur écrit d'abord les champs d'un descripteur, exécute la barrière requise et publie ensuite l'état prêt. Le consommateur lit l'état, acquiert la visibilité du descripteur puis lit les champs. Une barrière placée après la publication est trop tardive. De même, une barrière ne corrige pas une course sur une variable non atomique et ne remplace pas un registre matériel de type SET/CLR.

A retenir

Une barrière ordonne des accès, mais ne transporte pas une donnée et ne rend pas une opération atomique. Le choix entre barrière C11, instruction ISA et maintenance de cache dépend du domaine de partage.

Les instructions exclusives, telles que les primitives ARM basées sur `LDREX` et `STREX`, ou les opérations atomiques de RISC-V, permettent de construire un verrou ou une mise à jour conditionnelle lorsque le matériel garantit leur portée. Elles peuvent échouer et doivent alors être réessayées selon une borne documentée. Elles ne sont pas nécessairement disponibles pour une région MMIO, une mémoire non partagée ou un interconnect particulier. Avant de les utiliser entre coeurs, vérifier la cohérence des caches, la granularité de réservation, les domaines de partage et la progression garantie par le fabricant.

```
1  #include <stdint.h>
2
3  struct arm_semaphore {
4    volatile uint32_t count;
5  };
6
7  /* Return 0 on success, -1 when the semaphore is unavailable. */
8  static int semaphore_try_wait(struct arm_semaphore *semaphore)
9  {
10   uint32_t observed;
11   uint32_t updated;
12   uint32_t status;
13
14   do {
15     __asm__ volatile (
16       "ldrex %0, [%2]\n"
17       : "=r"(observed)
18       : "0"(observed), "r"(&semaphore->count)
19       : "memory");
20     if (observed == 0U) {
21       __asm__ volatile ("clrex" : : : "memory");
22       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"
      : "=r"(observed)
      : "0"(observed), "r"(&semaphore->count)
      : "memory");
    if (observed == UINT32_MAX) {
      __asm__ volatile ("clrex" : : : "memory");
      return -1;
    }
    __asm__ volatile (
      "strex %0, %2, [%1]\n"
      : "=r"(status)
      : "r"(&semaphore->count), "r"(observed + 1U)
      : "memory");
  } while (status != 0U);

  return 0;
}
```

Listing 6.10 – Sémaphore binaire ARM avec LDREX et STREX

Cette implémentation utilise une réservation exclusive sur une variable située en mémoire partagée et naturellement alignée. `LDREX` lit la valeur et établit la réservation ; `STREX` écrit seulement si cette réservation est encore valide, et renvoie alors zéro. Une interruption, un autre coeur ou une autre écriture peut invalider la réservation : la boucle doit donc traiter explicitement l'échec de `STREX`. `CLREX` est nécessaire lorsque l'acquisition échoue sans écriture.

Le compteur doit être initialisé à la capacité du sémaphore avant l'activation des contextes concurrents. En pratique, `semaphore_post` doit aussi vérifier le débordement et le protocole doit définir qui possède le droit de libérer le sémaphore. Les barrières `DMB` ne remplacent pas la réservation exclusive : elles ordonnent les accès autour de la publication du verrou, mais ne rendent pas une opération ordinaire atomique. Cette primitive ne convient ni aux registres MMIO ni à une mémoire dont la cohérence n'est pas garantie par l'interconnect ; le BSP doit alors fournir une primitive adaptée au domaine de partage.

Habituellement, sur les SoCs n'assurant pas de cohérence globale, par exemple lorsqu'ils

hébergent des cœurs hétérogènes, la synchronisation entre domaines s'effectue au moyen de périphériques dédiés comme des FIFO matérielles ou des mailboxes, décrits dans le chapitre suivant.
Ces périphériques se chargent alors d'assurer les mécanismes de verrouillage entre accesseurs.

A retenir

Une primitive atomique ne suffit pas à définir un protocole. Il faut également préciser l'alignement, le domaine de partage, la cohérence de cache, la borne de progression et le comportement lorsque la réservation ou la file échoue.

6.5 Notion de firmware et de bootloader

Un firmware est l'ensemble du logiciel persistant destiné à piloter une cible matérielle déterminée. Il comprend notamment le code d'initialisation, les pilotes, les services applicatifs et les données constantes nécessaires à son exécution.
Dans un système bare-metal, le firmware s'exécute sans système d'exploitation qui fournirait une abstraction du processeur ou des périphériques. Il doit donc prendre en charge les premières opérations de démarrage, la configuration de la mémoire et la distribution des interruptions. Le terme firmware décrit une fonction et un mode de livraison, et non un format binaire particulier : l'image peut être représentée par un ELF archivé, un fichier binaire brut ou des enregistrements IHEX.

Le bootloader est un composant de démarrage dont la responsabilité est de préparer l'exécution d'une autre image. Il peut être intégré à l'image applicative, placé dans une région réservée de la mémoire non volatile ou être fourni par le fabricant. Ses responsabilités doivent être bornées et documentées : déterminer la source de l'image, contrôler ses limites et son format, préparer les ressources indispensables. Il transfère ensuite le contrôle à un point d'entrée conforme au contrat de l'image. La vérification cryptographique des images et le secure boot sont volontairement hors du périmètre de ce guide afin d'en limiter la complexité, mais la problématique de boot sécurisé reste à étudier en regard du contexte du produit.

6.5.1 BootROM, bootloader et image applicative

Après une mise sous tension ou une remise à zéro, l'initialisation du processeur est déterminée par le fabricant du SoC. Le processeur principal n'est pas forcément le premier à être initialisé. C'est typiquement le cas dans les SoC comportant un HSE (*Hardware Security Engine*) ou tout processeur spécialisé en charge de l'initialisation de la plateforme. Selon les fabricants et les modèles de SoC, plusieurs firmwares peuvent être chaînés avant l'exécution de l'image applicative. On retrouve alors des firmwares intégrés au SoC par le fabricant, comme les BootROM.
Lors de l'implémentation d'un firmware, il est important de comprendre la logique de boot d'une plateforme, et de connaître les étapes précédant l'exécution de l'image applicative.
Il peut ainsi être nécessaire de réinitialiser certains périphériques, de reconfigurer les horloges, de compléter l'initialisation de la mémoire ou de vérifier l'intégrité d'une image avant de pouvoir l'exécuter.

Une chaîne de démarrage typique comporte les étapes suivantes :

1. le reset met le processeur et les périphériques dans un état défini
2. la BootROM ou le vecteur de reset sélectionne et localise l'image suivante
3. le bootloader configure son contexte minimal, vérifie les bornes de l'image et initialise éventuellement la mémoire nécessaire à sa lecture
4. l'image applicative installe sa table de vecteurs, initialise ou reconfigure la mémoire, les horloges, les protections et les périphériques requis
5. le code de démarrage appelle le point d'entrée métier

A retenir

Chaque transition de boot doit définir l'image, la pile, le mode processeur, les interruptions, les registres réservés et les conditions de retour. Une adresse de saut seule ne constitue pas un contrat de transfert.

Chaque transition est une interface entre deux composants. Elle fixe au minimum l'adresse de chargement, l'adresse d'exécution, l'état des interruptions, le mode processeur, le pointeur de pile, les registres réservés et la convention d'appel. Avant le saut, un bootloader doit désactiver ou remettre dans un état documenté les interruptions qu'il a activées, arrêter ses périphériques et transmettre un contexte explicitement défini. Il ne suffit pas de brancher le pointeur d'instruction sur une adresse de fonction. Une image peut exiger la présence d'une table de vecteurs, une pile alignée ou des registres de contrôle déjà configurés. Habituellement, le code de démarrage d'une image applicative s'assure d'une remise en état connu de l'ensemble des ressources qu'il utilise, même si le bootloader a déjà fait une partie de ce travail, afin de garantir l'absence de dépendance trop forte vis-à-vis d'un bootloader particulier.

7

Intégration dans des architectures multicoeurs

Les architectures multicoeurs sont de plus en plus présentes, y compris dans les microcontrôleurs. C'est par exemple le cas des SoCs (System on Chip) des Raspberry Pi Pico2, fournissant deux coeurs SMP (Symmetric Multi-Processing), dans les SoCs type MPC5xxx, fournissant deux coeurs activable en SMP ou sous forme de *step core* pour assurer une redondance d'exécution.

Les schémas suivants représentent trois organisations courantes. Ils sont fonctionnels plutôt que physiques : les adresses, les protocoles de cohérence et les latences exactes dépendent du SoC. Dans chaque schéma, les caches sont associés au coeur ou au cluster qui les possède, tandis que la SRAM et la Flash représentent les ressources visibles par le système.

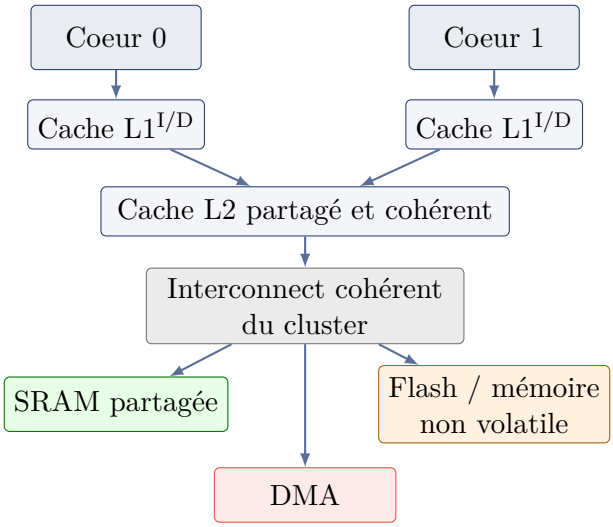


FIGURE 7.1 – Architecture SMP homogène à deux coeurs dans un cluster cohérent.

Dans une architecture SMP homogène, les coeurs exécutent le même jeu d'instructions et partagent généralement un domaine de cohérence. Chaque coeur peut posséder un cache L1 privé, tandis qu'un cache L2 et l'interconnect arbitrent les accès à la SRAM, à la Flash et au DMA. La cohérence ne supprime pas les protocoles : elle rend visibles les lignes de cache selon les règles du matériel, mais les atomiques, les barrières et la propriété des buffers restent nécessaires.

Plusieurs clusters homogènes peuvent chacun disposer d'un cache local et d'un interconnect interne. L'interconnect système relie ensuite les clusters, les mémoires et les périphériques. La

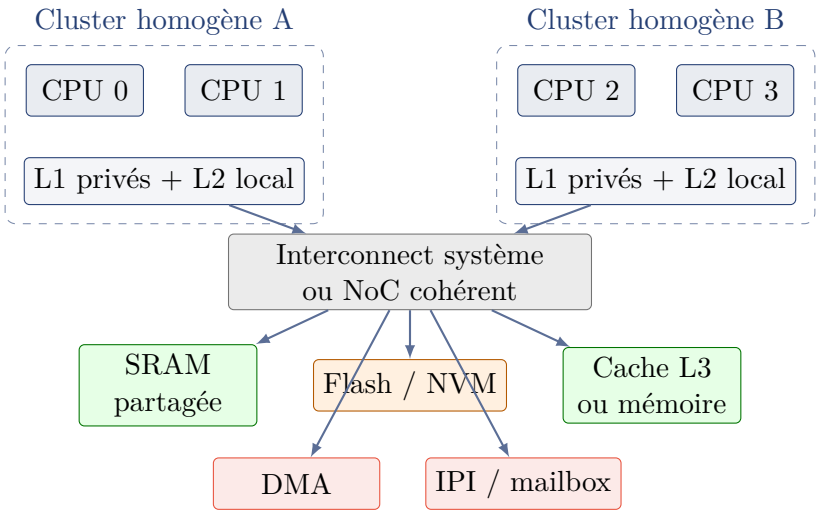


FIGURE 7.2 – Architecture homogène multi-clusters avec cohérence à l'échelle du système.

cohérence peut être globale, limitée à certaines régions ou absente entre clusters : le firmware doit alors déclarer le domaine de partage et effectuer les opérations de cache et de barrière nécessaires avant un transfert. Une mailbox ou un IPI peut signaler un événement, mais ne remplace pas la publication ordonnée des données.

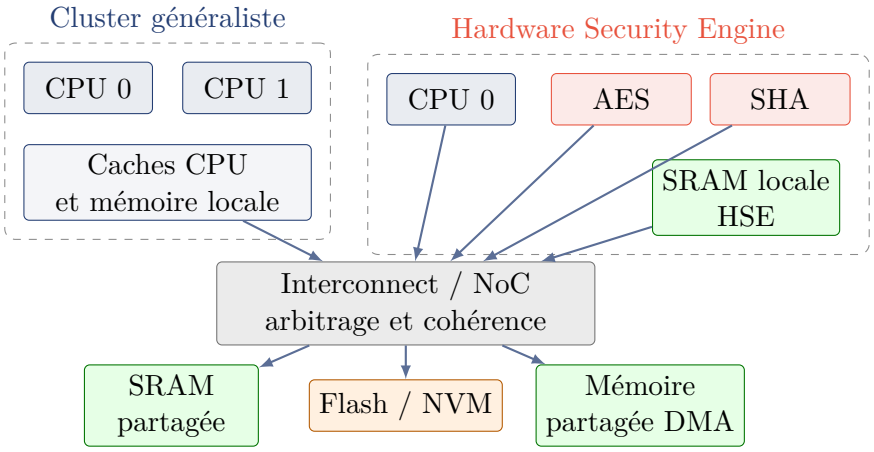


FIGURE 7.3 – Architecture hétérogène combinant coeurs généralistes et accélérateurs.

Dans une architecture hétérogène, le cluster généraliste, le coeur d'un HSE, etc. ont en généralement des jeux d'instructions, des caches et des domaines de mémoire différents. L'interconnect assure le transport et l'arbitrage, mais ne transforme pas ces domaines en une mémoire uniforme. Les buffers échangés doivent être décrits par une interface binaire fixe, avec adresse, taille, alignement, format, propriétaire et état. Une maintenance de cache peut être requise avant et après l'accès d'un accélérateur. La priorité de bus doit aussi être vérifiée contre les exigences temps réel du DMA et des coeurs généralistes.

7.1 Problématique de concurrence

La concurrence apparaît dès que plusieurs contextes peuvent observer ou modifier un même état : coeurs d'un processeur SMP, threads, ISR, contrôleurs DMA ou coeurs hétérogènes. Elle

ne dépend donc pas de la présence d'un système d'exploitation. Un firmware bare-metal doit recenser les producteurs et consommateurs de chaque objet, son propriétaire, sa durée de vie, son domaine de partage et le protocole qui garantit son accès. Une qualification `volatile` rend certains accès observables par le compilateur, mais ne règle ni la course de données, ni l'atomicité, ni la cohérence des caches.

Une *race condition* survient lorsque le résultat dépend de l'entrelacement de deux séquences non coordonnées. Lorsqu'au moins une séquence écrit un objet non atomique et qu'aucune relation d'ordre ne les synchronise, il s'agit d'une course de données et le comportement est indéfini au regard du modèle mémoire C11. Un registre matériel, une zone DMA ou une mémoire partagée entre coeurs doit en plus respecter les garanties du SoC, qui ne sont pas toutes exprimées par le langage C [31].

Un cas classique est le compteur partagé suivant : deux coeurs lisent la valeur 4, calculent chacun 5 et écrivent chacun 5. Une mise à jour a alors été perdue. Une interruption peut produire le même défaut sur un seul coeur entre la lecture et l'écriture d'une variable. Les accès composés comme `counter++`, un test suivi d'une action et une lecture-modification-écriture ne sont atomiques que si leur contrat matériel ou logiciel le garantit.

Le défaut TOCTOU (*time of check to time of use*) est une forme particulière de course : le programme vérifie une propriété, puis l'état change avant son utilisation. Dans un firmware, cela concerne par exemple la vérification d'un descripteur DMA avant son lancement, la lecture d'un état de périphérique avant l'effacement d'une alerte, ou la validation d'une adresse de mémoire partagée avant un accès ultérieur. La solution est de maintenir le verrou ou la propriété pendant toute l'opération, de revalider l'état juste avant l'usage, ou d'utiliser une primitive qui combine test et transition d'état [7, 33].

Ces défauts sont exploités dans plusieurs familles d'attaques : double-fetch de données contrôlées par un périphérique ou un autre domaine de privilège, modification d'un descripteur entre sa validation et son emploi, libération concurrente d'un buffer, ou déclenchement répété d'une interruption pendant une séquence supposée indivisible. Le point commun n'est pas le type de processeur, mais l'absence de propriété explicite sur la donnée et la fenêtre temporelle. Une revue de sécurité recherche donc les contrôles séparés de leur utilisation, les callbacks réentrants, les compteurs partagés et les chemins d'erreur qui libèrent deux fois une ressource [16, 32].

La concurrence thread/ISR existe également en mono-coeur. L'ISR interrompt le thread entre deux instructions et peut modifier un état que le thread utilise ensuite. Pour une courte section critique, le firmware peut masquer les interruptions au niveau approprié, à condition de borner sa durée et de ne pas bloquer une interruption temps réel critique. Pour un état partagé avec plusieurs coeurs ou un DMA, le masquage local ne suffit pas : il faut une primitive atomique, un verrou inter-coeur, une file ou un protocole de propriété.

```c
#include <stdint.h>

static volatile uint32_t pending_events;

void timer_isr(void)
{
  pending_events++;
```

```
}

uint32_t take_events(void)
{
  uint32_t events;

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

Listing 7.1 – Compteur partagé entre un thread et une ISR

Cet exemple ne constitue pas une primitive universelle : il est valable seulement si la lecture, l'effacement et la restauration du masque sont atomiques vis-à-vis de l'ISR sur cette cible. Entre coeurs, on remplace cette protection locale par une opération atomique ou une file partagée. Chaque solution documente également sa progression : un spinlock sans borne peut bloquer définitivement un coeur si le propriétaire est arrêté, préempté ou victime d'une faute.

7.2 Échanges entre les cœurs

L'affectation d'un événement à un coeur est une décision d'architecture. Une interruption peut être dirigée vers un coeur fixe, distribuée entre coeurs d'un même cluster, ou reçue par un contrôleur qui réveille le coeur propriétaire d'une file.
Le BSP décrit les registres d'affinité, le coeur qui configure le contrôleur, les interruptions réservées au démarrage et le comportement lorsqu'un coeur est arrêté. Une interruption reçue par le mauvais coeur doit être acquittée selon le protocole du contrôleur, sans modifier un état dont ce coeur n'est pas propriétaire.

Pour deux coeurs homogènes partageant une mémoire cohérente, un spinlock, une opération atomique ou un IPI peuvent suffire. Le spinlock protège une courte section critique, alors que l'IPI signale qu'un événement ou une donnée est disponible. L'IPI ne transporte pas à elle seule la donnée : le producteur publie d'abord les données avec la barrière requise, puis déclenche l'IPI, et le consommateur acquiert la visibilité avant de lire la structure. L'acquittement et la perte éventuelle d'un IPI doivent être explicitement traités. Un IPI ne doit pas être utilisé comme un compteur sans mécanisme associé.

Dans cet exemple, le coeur émetteur écrit d'abord la charge utile dans `DATA0` et `DATA1`, puis publie l'événement dans `DOORBELL`. Le registre `STATUS` peut indiquer l'état vide, plein, occupé ou acquitté selon le contrat du périphérique. Le coeur consommateur lit les données après avoir observé la notification, puis écrit l'acquittement attendu. Les registres sont représentés comme des points d'accès MMIO : leur ordre, leurs effets de lecture et d'écriture, ainsi que la possibilité de coalescer plusieurs notifications, sont définis par le SoC.

Une mailbox (boîte aux lettres matérielle) fournit généralement des registres ou une petite file pour transmettre un mot, une adresse ou un événement entre domaines. Une FIFO matérielle transporte plusieurs éléments et peut gérer le remplissage, le vidage et la pression amont. Ces deux mécanismes réduisent le partage direct de structures, mais leurs registres ont leurs propres

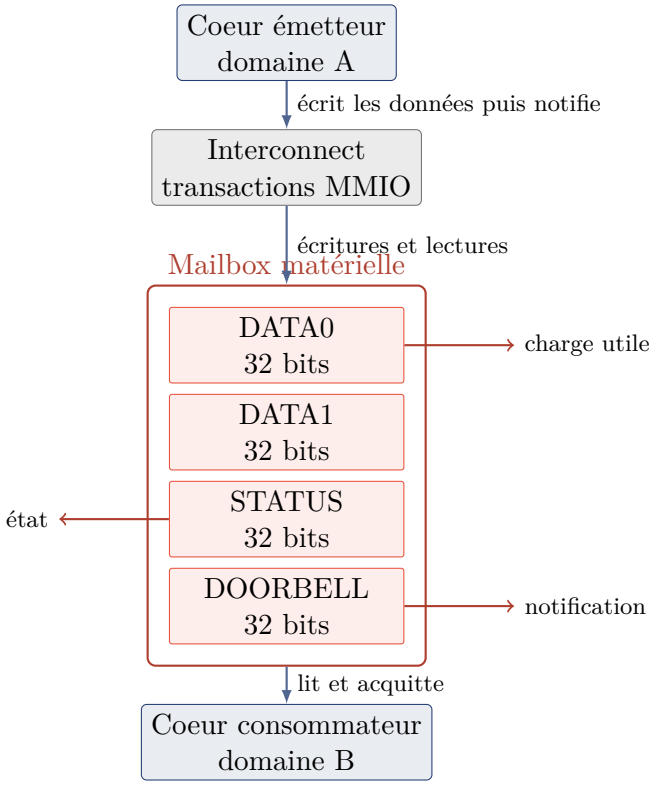


FIGURE 7.4 – Mailbox matérielle à quatre registres 32 bits entre deux domaines.

règles : écriture atomique, acquittement, dépassement, reset et visibilité des données référencées. Une adresse placée dans une mailbox doit rester valide et accessible au consommateur pendant toute la durée du message.

```
struct mailbox_registers {
  volatile uint32_t data0;
  volatile uint32_t data1;
  volatile uint32_t status;
  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;
}

```

```c
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;
    dispatch_message(command, argument);
  }
}
```

Listing 7.2 – Publication d'un message par mailbox

La primitive `arch_data_mem_barrier` est fournie par le BSP et traduit la barrière requise par l'ISA et le domaine de partage. Elle garantit ici que les écritures dans `DATA0` et `DATA1` sont observables avant l'écriture de `DOORBELL`. Cette dernière constitue la notification matérielle : le consommateur peut la recevoir comme une interruption de périphérique ou la détecter en lisant `STATUS`. Ce modèle permet une communication entre coeurs hétérogènes. Il permet également de traverser les domaines d'horloge.
Le protocole doit empêcher le producteur d'écraser le message avant sa consommation. Dans cet exemple, le producteur refuse l'envoi lorsque `STATUS` n'est pas vide et le consommateur libère la mailbox après avoir copié les deux données. Les valeurs de statut, les effets de lecture et la sémantique d'acquittement doivent être définis par le manuel du périphérique ; un registre `STATUS` de type write-one-to-clear ne se manipule pas comme une simple variable.

La synchronisation de coeurs d'un même cluster bénéficie généralement d'un domaine de cohérence commun, d'une latence faible et d'instructions atomiques directement applicables à la SRAM partagée.
À l'inverse, entre clusters homogènes, la cohérence peut être partielle ou limitée à certaines régions. Il faut alors préciser le domaine de partage, nettoyer ou invalider les caches si nécessaire et employer l'interconnect ou les registres de communication prévus. Entre coeurs hétérogènes, les ISA, les largeurs de mots, les ABIs et les modèles de mémoire peuvent différer ; une mailbox, une FIFO ou une zone mémoire décrite par un format binaire fixe est généralement préférable à un pointeur ou une structure C partagée directement.

7.3 Concurrence d'exécution

La concurrence d'exécution ne concerne pas seulement les variables partagées. Les coeurs, les DMA, les accélérateurs et les périphériques se disputent aussi les interconnexions et les banques mémoire. L'interconnect transporte les transactions, applique les domaines de visibilité et peut fournir une cohérence de cache. Il ne garantit pas automatiquement que deux écritures concurrentes sont traitées dans l'ordre attendu par le logiciel ; cet ordre relève du protocole et des barrières.

L'arbitreur de bus choisit quelle requête est servie lorsque plusieurs maîtres demandent une ressource. Selon le SoC, l'arbitrage peut être fixe, tournant, pondéré par priorité ou sensible à la qualité de service. Un coeur qui détient un verrou peut être ralenti par une transaction DMA prioritaire ; à l'inverse, un accès répétitif à une mémoire partagée peut affamer un périphérique temps réel. Le dimensionnement doit donc considérer la latence maximale, la bande passante, les

bursts, les conflits de banques et les files internes de l'interconnect.

Les coeurs homogènes exécutent souvent le même code et partagent un espace mémoire, mais ils peuvent encore avoir des caches privés, des buffers d'écriture et des latences différentes. Les coeurs hétérogènes ajoutent des contraintes de conversion de format, de largeur et de propriété des buffers. Un DSP ou un GPU peut ne voir une écriture CPU qu'après une maintenance de cache et une barrière spécifique. Un pointeur C transmis tel quel est alors insuffisant : le message doit préciser l'adresse, la taille, l'alignement, le format et la durée de propriété du buffer.

Un protocole de communication robuste définit au minimum l'état du message, son producteur, son consommateur, les transitions autorisées, la condition de saturation, la reprise après reset et les délais acceptables. Les compteurs et séquences sont de largeur fixée ; les valeurs réservées et les erreurs de transport sont distinguées des erreurs du traitement métier. Le protocole est testé sous charge, avec retards, réordonnancements, pertes d'événements et reset d'un seul domaine.

Enfin, la priorité d'un bus ne doit pas devenir un mécanisme implicite de sécurité. Un périphérique qui peut écrire une zone partagée doit être limité par la MPU, la SMMU ou les attributs du contrôleur DMA lorsque ces mécanismes existent. La séparation des domaines, la validation des longueurs et la propriété des buffers restent nécessaires même lorsque l'interconnect est cohérent.

8

Concevoir un firmware embarqué sécurisé

8.1 Rappels sur la sécurité embarquée

La sécurité d'un firmware embarqué ne se réduit pas à la confidentialité d'une clé ou à la présence d'une primitive cryptographique. Elle vise la disponibilité, l'intégrité, l'authenticité et la maîtrise des accès aux fonctions et aux données, depuis la conception jusqu'au retrait du produit. L'analyse de risque décrit les actifs, les acteurs, les chemins d'attaque, les conséquences et les mesures de réduction du risque. Elle doit couvrir le matériel, le logiciel, les outils de développement, la production, la maintenance et les interfaces physiques [3, 35].

Un microcontrôleur expose une surface d'attaque particulière : broches de debug, interfaces de récupération, bus internes, DMA, bootloader, mises à jour, protocoles radio ou filaires et données conservées en Flash. Un attaquant peut exploiter une entrée mal validée, un défaut de concurrence, une configuration de privilèges, une faute physique, une fuite temporelle ou une dépendance logicielle compromise. Les mesures de sécurité doivent donc être reliées à une menace identifiée et testées sur le binaire et la configuration réellement livrés.

La sécurité physique concerne notamment l'accès aux broches, la lecture ou le remplacement de la mémoire, l'injection de fautes, l'observation de la consommation et la résistance mécanique du boîtier. La sécurité logique concerne l'isolation des domaines, la validation des entrées, les droits de mémoire, la robustesse des protocoles, la journalisation et la récupération après erreur. La sécurité organisationnelle concerne les comptes, les secrets de signature, les revues, la traçabilité des versions, la séparation des rôles et la réponse aux vulnérabilités. Une protection logique ne compense pas une clé de production accessible à toute l'équipe, et une procédure organisationnelle ne corrige pas un DMA non restreint.

La sécurité par conception fixe les invariants de sécurité avant l'implémentation : un périphérique ne peut écrire que dans ses buffers, une interface refuse les longueurs hors contrat, un privilège est accordé seulement à la fonction qui en a besoin et une erreur conduit à un état défini. La sécurité par défaut choisit le profil le plus restrictif lorsqu'une option n'est pas explicitement configurée. La sécurité par l'usage ajoute des contrôles d'exploitation : procédure de mise à jour,

gestion des secrets, surveillance des événements, limitation des accès et réponse aux incidents. Ces trois principes doivent apparaître dans les exigences, les interfaces, les tests et les preuves.

8.1.1 Actifs, frontières et objectifs de sécurité

Pour chaque actif, le projet documente son propriétaire, sa représentation, sa durée de vie, les domaines qui peuvent le lire ou l'écrire et le comportement attendu en cas de faute. Les clés, identifiants, compteurs anti-rejeu, images firmware, données de calibration et journaux ne possèdent pas les mêmes exigences. Une clé peut exiger une zone non lisible par le code applicatif, un compteur de séquence exige surtout une mise à jour monotone et résistante aux interruptions, un journal exige une reprise cohérente après reset.

Les frontières de sécurité sont placées aux interfaces réellement traversées : appel d'une ISR, descripteur DMA, mailbox inter-coeurs, commande utilisateur, fonction de bootloader et primitive de pilote. Chaque frontière possède un format, des bornes, une politique d'erreur et une preuve de propriété. Les pointeurs ne sont pas transmis entre domaines comme s'ils avaient la même signification ; une interface inter-domaine transporte une adresse validée ou un identifiant de buffer selon le contrat du matériel.

A retenir

Un actif n'est pas défini par son seul nom : sa propriété, sa durée de vie, ses lecteurs, ses écrivains, son format et sa réaction aux fautes doivent être documentés.

A retenir

Une menace doit être reliée à un actif, une frontière, une mesure de protection et un moyen de vérification. Une mesure isolée, sans propriétaire ni test, ne constitue pas un objectif de sécurité démontré.

8.2 Processus de développement sécurisé

Le développement sécurisé est un processus de configuration et de preuve, et non une étape ajoutée avant la livraison. Une version de firmware est associée à un commit, une configuration de compilation, un compilateur, un script de lien, les sources de ses dépendances, les résultats d'analyse et les artefacts de production. Toute modification du code, du BSP, du compilateur, des options, de la carte mémoire ou des outils de génération peut modifier le périmètre de qualification [34, 9].

La supply chain comprend les bibliothèques, générateurs de code, outils de test, images de coprocesseurs, firmwares de périphériques, fichiers de configuration et matériels de production. Le projet conserve la provenance, la version, l'intégrité, la licence et le statut de support de chaque composant. Les dépendances non utilisées sont supprimées, celles qui restent sont surveillées et leur impact est analysé.

Une SBOM décrit les composants présents dans le produit, leurs versions, leurs relations et, lorsque cela est possible, leurs identifiants et empreintes. Elle doit être générée à partir de la configuration effectivement construite, archivée avec l'ELF et reliée à l'image livrée. Une SBOM ne prouve pas l'absence de vulnérabilité : elle rend les composants identifiables et permet de déclencher une analyse lorsqu'une vulnérabilité ou une nouvelle exigence apparaît [14, 34].
On y associe de plus en plus une CBOM, permettant d'expliciter les algorithmes cryptographiques, leur usage, le positionnement et le cycle de vies des assets cryptographiques associés.

La revue par les pairs vérifie les contrats, les invariants, les erreurs et les effets de bord. Elle doit examiner le code C, les listings assembleur, les scripts de lien, les configurations MPU/cache, les règles de compilation et les tests. Le TDD peut guider l'implémentation d'une primitive pure ou d'un protocole : un test décrit d'abord le comportement attendu, puis l'implémentation est écrite et enfin refactorisée sous contrôle de la campagne. Pour le code dépendant du matériel, les tests unitaires sont complétés par des tests sur cible, des tests d'intégration et des injections de fautes contrôlées.

8.2.1 Auditabilité et noRTE

L'auditabilité exige que chaque résultat soit reproductible et explicable. Le projet conserve les versions d'outils, les options, les journaux de CI, les rapports d'analyse statique, les couvertures, les preuves, la carte de lien et l'empreinte du binaire. Une alerte supprimée est remplacée par une justification référencée, jamais par une désactivation silencieuse de l'outil.

A retenir

L'auditabilité exige de pouvoir reproduire un résultat et d'expliquer chaque hypothèse, alerte, exclusion et artefact utilisé pour le produire.

La propriété noRTE est définie sur un périmètre et des hypothèses : bornes des entrées, mémoire valide, alignement, absence d'interruption non modélisée, contrat du périphérique et comportement du matériel. Elle couvre notamment débordements, accès hors limites, divisions invalides, conversions dangereuses, dépassements de pile, erreurs de protocole et chemins de non-terminaison. Une preuve noRTE ne prouve pas automatiquement la sécurité fonctionnelle ni l'absence de défaut matériel, elle établit les propriétés annoncées sous les hypothèses contrôlées.

A retenir

Une preuve noRTE est toujours bornée par son périmètre et ses hypothèses. Elle ne remplace ni les tests sur cible, ni les tests de protocole, ni la validation de la configuration matérielle.

8.2.2 Tests et couverture

Les tests unitaires vérifient les cas nominaux, les limites et chaque erreur accessible d'une fonction. Les tests d'intégration vérifient les contrats entre modules. Les tests sur cible vérifient

le code réellement lié, les timings, les attributs mémoire et les comportements d'exception.

La couverture d'instructions mesure les instructions exécutées ; la couverture de branches mesure les résultats de décisions ; la couverture de conditions examine les conditions élémentaires, MC/DC démontre que chaque condition peut influencer indépendamment la décision. La couverture de chemin est utile sur des chemins bornés, mais devient rapidement combinatoire avec des boucles et des interruptions. Une couverture élevée ne compense ni des tests mal spécifiés ni du code mort non identifié. Les exclusions sont justifiées, versionnées et réexaminées.

A retenir

La SBOM identifie les composants, tandis que les tests, analyses et preuves établissent les propriétés du firmware. Aucun de ces artefacts ne remplace les autres.

A retenir

La couverture mesure ce qui a été exécuté, pas ce qui est correct. Les cas nominaux, limites, erreurs, interruptions et chemins de reprise doivent être sélectionnés à partir des contrats.

8.3 Le bon usage de la documentation

La documentation automatique est un instrument de cohérence, pas une architecture. Doxygen peut extraire les interfaces C, les groupes de modules, les préconditions et les relations d'appel. Sphinx peut assembler des documents structurés, des décisions d'architecture, des procédures et des rapports. Les versions des générateurs, leurs options et leurs entrées doivent être figées pour que la documentation soit reproductible.

Une interface documente son rôle, ses entrées, ses sorties, les erreurs, la propriété des buffers, l'alignement, le contexte d'appel, les interruptions, la concurrence, la durée d'exécution et les effets de bord. Une fonction critique documente aussi les hypothèses de preuve, les registres touchés, les barrières, les ressources acquises et le comportement en cas de faute. Les exemples doivent préciser s'ils sont normatifs, indicatifs ou dépendants d'un SoC particulier.

La documentation d'architecture décrit les concepts et leurs limites : domaines de privilège, carte mémoire, flux de démarrage, ownership des buffers, modèle de cache, chemins d'erreur, stratégie de mise à jour et dépendances externes. Doxygen ne peut pas déduire la sémantique d'un registre, la propriété d'une donnée DMA ou la raison pour laquelle une interruption est masquée. Ces informations doivent être écrites dans un document d'architecture et reliées aux exigences et aux tests.

Une documentation utile distingue les garanties de l'ISO C, celles du compilateur, celles de l'ABI et celles du matériel. Elle signale explicitement les extensions, les profils non portables et les hypothèses qui doivent être vérifiées dans le BSP.

A retenir

Une documentation d'interface doit exposer les données, les erreurs, la concurrence, les effets de bord et les hypothèses de cible. Doxygen ou Sphinx ne peuvent pas déduire ces contrats à la place de l'architecte.

8.4 La modularité logicielle

Un CSC est un composant logiciel identifié dans une architecture ; une CSCI est un élément de configuration regroupant le logiciel et ses artefacts contrôlés ; un ICD décrit l'interface entre éléments. Ces notions empêchent de traiter le dépôt comme un bloc indifférencié : chaque module possède un propriétaire, un contrat, une configuration et une stratégie de test.

Une bonne interface minimise les données partagées, exprime la propriété des ressources et sépare les erreurs de validation des erreurs d'exécution. Elle ne retourne pas une valeur dont le même nombre signifie à la fois une donnée valide et une erreur. Les buffers ont une taille explicite et les opérations indiquent si les zones peuvent se recouvrir. Les appels d'ISR et les callbacks documentent leur contexte et leurs interdictions de blocage.

Les interfaces standardisées apportent une interopérabilité, mais pas une preuve automatique de sûreté. PSA peut structurer l'accès à des services cryptographiques ; AUTOSAR formalise des couches et des contrats automobiles ; ARINC 615 décrit des échanges de chargement dans certains environnements aéronautiques. Le projet vérifie la version, le profil retenu, les extensions et les écarts entre l'interface standard et le matériel réellement disponible.

La spécification d'une interface suit au minimum ces étapes :

- définir les actifs et les frontières
- choisir les types et formats
- fixer les préconditions et postconditions
- décrire ownership et concurrence
- définir les erreurs et les timeouts
- établir les tests et les preuves

Toute interface externe est traitée comme hostile jusqu'à validation de son format et de ses limites.

Une interface POSIX peut être utile sur une couche d'adaptation, mais elle ne doit pas être confondue avec le modèle mémoire typé du firmware. Des handles opaques, des retours à double sémantique, des tailles implicites ou des conversions de pointeurs peuvent rendre les contrats difficiles à prouver. Le code critique expose donc des wrappers à types explicites et transforme les erreurs POSIX en résultats distincts et vérifiables.
Il est important de souligner que certaines interfaces POSIX se prêtent mal à une preuve formelle directe, car leurs contrats utilisent des handles opaques, des conventions implicites ou des retours à plusieurs significations. Par exemple, `read()` retourne soit une taille, soit une valeur d'erreur ; un wrapper typé peut séparer ces deux résultats avant d'exposer l'interface au code critique.

Lors de l'implémentation modulaire d'un firmware, une décomposition typique des CSC est la suivante :

- support de l'architecture : ABI, exceptions, atomiques, barrières et contexte
- support du SoC et BSP (pilotes) : horloges, gpio, DMA, etc.
- primitives utilitaires *zero lib* : `memcpy`, `memset`, `strnlen`, conversions d'octets et vérifications de bornes
- primitives de pilotes : accès MMIO, files, verrous, timeouts et acquittements
- services et application de plus haut niveau : protocoles, stockage, journalisation et politique d'erreur

Les implémentations de `memcpy`, `memcmp`, des conversions d'octets et des primitives de pilotes de Sentry Kernel et Shield peuvent servir de références de conception [10, 11]. Elles doivent être adaptées au profil de la cible, puis soumises à l'analyse et aux tests du projet : réutiliser un code ne transfère ni ses hypothèses ni sa 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 – Interface typée de copie mémoire

Cette interface rend visibles les tailles et les erreurs, contrairement à une primitive qui supposerait implicitement que le buffer est valide ou qui confondrait une longueur retournée avec un code d'erreur. Le contrat complet précise si les zones peuvent se recouvrir et si une longueur nulle autorise des pointeurs nuls.

A retenir

Une interface sûre rend explicites les types, les tailles, la propriété des ressources, les erreurs et les règles de concurrence. Elle réduit la surface de preuve et la surface d'attaque.

8.5 La sécurité par conception

La sécurité par conception applique la minimisation de la surface d'attaque, la séparation des privilèges et la séparation des responsabilités. Un module ne reçoit que les registres, buffers et commandes nécessaires à sa fonction. Les entrées sont validées au plus près de la frontière ; les secrets ne traversent pas une interface qui n'en a pas besoin. Les chemins d'erreur détruisent ou verrouillent les ressources sensibles dans un état connu.

Une super-loop bare-metal est simple à analyser mais partage souvent un espace de privilège, une pile et des objets globaux. Une faute dans un pilote peut alors affecter toute l'application. Un firmware collaboratif introduit plusieurs tâches qui cèdent volontairement le processeur : il doit contrôler la réentrance, les appels qui ne cèdent pas et les sections critiques, mais permet un cloisonnement minimaliste via la MPU. Un micro-noyau préemptif sépare mieux les domaines et les piles, mais ajoute des transitions, des contextes, des files et des chemins de planification à vérifier.

Dans un système préemptif, chaque thread possède un contexte sauvegardé et une pile bornée. Le changement de contexte doit préserver les registres callee-saved, l'état d'interruption, les registres de contrôle et, si nécessaire, les registres de FPU, SIMD ou DSP. Le choix entre sauvegarde systématique et sauvegarde paresseuse doit être documenté, car un état flottant oublié peut divulguer ou corrompre les données d'un autre thread. Le contexte d'interruption possède ses propres règles : une ISR ne doit pas appeler une primitive bloquante ou modifier l'état d'un thread sans protocole.

A retenir

La simplicité d'une super-loop facilite l'analyse, mais ne fournit pas automatiquement l'isolation. Un système préemptif ou multicoeur ajoute des mécanismes de protection qui doivent eux-mêmes être spécifiés et vérifiés.

8.6 A propos de l'ordonnancement

Un ordonnanceur collaboratif ne change de tâche qu'à des points explicites. Il est prévisible et peu coûteux, mais une tâche qui ne rend jamais la main bloque les autres. Un ordonnanceur préemptif peut interrompre une tâche selon une horloge, une priorité ou un événement ; il améliore l'isolation temporelle mais exige une sauvegarde de contexte et une synchronisation rigoureuse.

Une priorité est utile seulement si le protocole traite l'inversion de priorité, la famine et les ressources partagées. Un verrou détenu par une tâche basse priorité peut retarder une tâche critique ; l'héritage ou le plafond de priorité peuvent réduire ce phénomène lorsque leur coût est acceptable. Les sections critiques restent bornées et ne réalisent pas d'opération lente ou non déterministe.

Dans les systèmes temps réel, pour les tâches périodiques, RMA attribue des priorités fixes selon les périodes, EDF choisit la tâche dont l'échéance est la plus proche ; TDM réserve des fenêtres temporelles. FIFO et round-robin distribuent le temps sans garantir une échéance temps réel. Le choix doit être relié aux bornes d'exécution, aux interruptions, aux accès mémoire et aux temps de blocage, pas seulement à la politique nominale.

En SMP temps réel, la charge peut être répartie, mais les coûts de migration, de cache, d'IPI, de verrou et d'accès à l'interconnect s'ajoutent aux temps d'exécution. Une analyse vérifie les affinités, les ressources non migrables, les files par coeur, la contention mémoire et le comportement lorsqu'un coeur est arrêté. Un ordonnanceur correct ne rend pas automatiquement les

pilotes réentrants.

8.7 Effet de bords et interférences

Un effet de bord est une modification observable en dehors du résultat local d'une fonction : écriture MMIO, modification d'un buffer, publication d'un événement, acquittement d'interruption ou changement de cache. Les interfaces documentent ces effets afin que l'appelant puisse établir l'ordre, la propriété et la restauration des invariants.

Les interférences se produisent entre modules par les globales et les callbacks, entre threads par les objets partagés, entre coeurs par les caches et l'interconnect, entre périphériques par les horloges et les bus, et entre coprocesseurs par les registres de contexte et les buffers. Une analyse par composition recense les ressources communes, les ordres d'accès, les priorités et les chemins de faute. Elle vérifie aussi qu'un reset partiel ne laisse pas un périphérique dans un état que le logiciel suivant interpréterait comme valide.

Les contrôles pratiques comprennent ownership explicite, types opaques, copies aux frontières, barrières documentées, compteurs de timeout, masques de droits, absence de globales mutables exportés et tests sous concurrence. Une optimisation ne doit pas supprimer un accès matériel ou une barrière nécessaire au motif qu'il semble inutile au modèle C.

8.8 Preuve, auditabilité et traçabilité

La preuve noRTE commence par un modèle : types bornés, préconditions, invariants de boucle, postconditions et hypothèses sur les appels externes. Frama-C et ses plugins peuvent alors vérifier des bornes, des pointeurs, des divisions, des décalages et des propriétés fonctionnelles [12, 6]. Le résultat est un ensemble d'obligations de preuve, pas une affirmation globale indépendante de la configuration.

La preuve par composition établit d'abord les propriétés d'un module, puis vérifie que les contrats des interfaces relient les modules sans invalider leurs hypothèses. Pour un buffer DMA, elle combine par exemple la preuve des bornes du parseur, la propriété d'alignement du BSP, la maintenance de cache et le protocole de propriété. Pour une ISR, elle ajoute les invariants interrompus, les registres préservés et la borne de temps de la section critique. Les interférences non modélisées restent des risques résiduels explicitement déclarés.

```
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 */
```

```
int copy_checked(unsigned char *destination,
                 unsigned long destination_size,
                 const unsigned char *source,
                 unsigned long length);
```

Listing 8.2 – Exemple ACSL de preuve de bornes

Pour une longueur nulle, le contrat doit être formulé selon la syntaxe et le modèle retenus par l'outil afin d'éviter de créer une plage invalide. Les fonctions réelles ajoutent les préconditions de non-recouvrement ou choisissent explicitement la sémantique de `memmove`. La preuve de `memcpy` et `memcmp` dans Sentry Kernel constitue un exemple de démarche noRTE réutilisable, mais elle doit être rejouée avec la configuration du firmware.

8.9 Preuve noRTE de firmware embarqué

ACSL permet d'exprimer les préconditions, postconditions, effets de bord, invariants et comportements en erreur d'une fonction C. La logique de Hoare résume le contrat par une triple : {P} C {Q}, où la précondition P doit garantir qu'après l'exécution de C, la postcondition Q est vraie. Pour un parseur, P décrit la validité et la longueur du buffer ; Q décrit le résultat et l'état des curseurs ; les obligations intermédiaires empêchent les accès hors limites et les débordements.

Le prouveur travaille souvent sur un sur-ensemble des exécutions réelles. Un chemin considéré possible par l'analyse peut être impossible sur le matériel, mais il doit néanmoins être prouvé ou explicitement délimité par une hypothèse vérifiée. À l'inverse, une couverture de preuve complète ne démontre que les propriétés écrites et ne remplace pas les tests de protocole, de performance et de configuration.

La preuve d'une fonction critique suit une séquence reproductible : réduire les types et les états, écrire les contrats, établir les invariants, lancer l'analyse, résoudre les obligations, examiner les hypothèses restantes, puis rejouer la preuve dans la configuration CI. Les warnings, axiomes, fonctions laissées en `axiomatic` et appels non modélisés sont des éléments du dossier de preuve.

Pour une copie mémoire, la preuve établit les bornes, la validité des pointeurs, l'absence de recouvrement si elle est requise, les valeurs écrites et la terminaison. Pour un parseur X.509, elle ajoute les longueurs encodées, les profondeurs de structures, les conversions d'entiers et les erreurs de certificat. La publication SSTIC sur un parseur X.509 sans RTE montre l'intérêt de borner les formats et de séparer le décodage de la décision métier [28]. La publication sur une pile USB illustre la composition d'une preuve sur une grande pile de protocoles, avec états, buffers, callbacks et erreurs de transport [29].

La propriété W $\oplus$ X d'une MPU se prouve à deux niveaux. Au niveau logiciel, chaque configuration de région est vérifiée : une région possède des droits cohérents, les bornes sont alignées et les transitions d'état sont autorisées. Au niveau binaire, le script de lien et l'ELF sont inspectés pour confirmer que les sections exécutables ne sont pas inscriptibles et que les données ne sont pas exécutables. L'exemple fourni par Sentry Kernel dans le profil MPU Cortex-M peut guider cette démarche, mais les attributs effectifs du SoC doivent être vérifiés localement.

Le dossier final relie exigences, contrats, code, preuves, tests, couverture et artefacts binaires. Il indique les propriétés démontrées, les hypothèses matérielles, les chemins exclus, les limitations des outils et les risques résiduels. C'est cette traçabilité, plus que le seul pourcentage de preuve, qui rend la démarche auditable.

A retenir

Une preuve utile est reproductible et bornée par ses hypothèses. Le dossier d'audit doit relier exigences, contrats, code, outils, tests, preuves et binaire livré.

8.10 Fonctions critiques et autoprotections

La CFI (Control-Flow Integrity, intégrité du flot de contrôle) limite les transitions de contrôle aux cibles prévues par le graphe d'appel [1]. Elle réduit les possibilités d'attaques ROP et JOP, mais son efficacité dépend des appels indirects, des tables de vecteurs, des callbacks et des exceptions réellement déclarés [40, 8]. Les protections du compilateur sont vérifiées sur le binaire, car une table conservée par le script de lien ou une transition d'assembleur peut sortir du modèle automatique.

Un générateur pseudo-aléatoire comme PGC32 (générateur pseudo-aléatoire à compteur de 32 bits) peut produire des valeurs de canary, des identifiants de session ou des masques non secrets à partir d'une graine fournie par un TRNG matériel au démarrage. La graine est validée, protégée contre les valeurs triviales et ne doit pas être confondue avec une source cryptographique complète. Les compteurs et l'état du générateur sont réinitialisés et sauvegardés selon un contrat qui empêche la réutilisation dangereuse.

La protection de pile combine une région MPU non exécutable et non inscriptible par les domaines qui ne la possèdent pas, une garde contre le dépassement, une mesure de marge et éventuellement un canary. Le principe W $\oplus$ X interdit qu'une même région soit inscriptible et exécutable ; les exceptions nécessaires au démarrage sont temporaires, documentées et vérifiées dans l'ELF. Un canary détecte une écriture linéaire ayant atteint une valeur sentinelle, mais ne détecte pas toutes les corruptions et ne remplace pas une borne de pile [15].

GCC fournit des options de protection de pile et des mécanismes de durcissement des branches dont la disponibilité dépend de la version et de la cible. Une violation doit conduire à un trap ou à une voie de récupération définie, sans réutiliser un contexte corrompu. Les options activées, leurs limites et l'impact sur le temps réel sont archivés avec la configuration qualifiée.

```
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)
```

```
9  {
10     int res;
11     /* update SSP value with given seed */
12     __stack_chk_guard = seed;
13
14     /* main has stack protection active */
15     res = main();
16     /* inform kernel that the thread is finished */
17     __sys_exit(res);
18 }
```

Listing 8.3 – Activation du SSP au démarrage d'un thread

9

Les fonctions nécessaires à un firmware embarqué sécurisé

9.1 Les journaux d'événements

A retenir

Un journal embarqué doit être borné, vérifiable après une coupure et indépendant de la représentation native des structures C. Le CRC détecte les corruptions accidentelles, mais n'authentifie pas un attaquant.

Un journal embarqué enregistre des événements dans un format binaire fixe et borné. La largeur des champs, l'endianness, la version, le calcul du CRC et la taille de la charge utile sont spécifiés indépendamment de la représentation native des structures C. Un enregistrement contient généralement un type, une longueur, un numéro de séquence, un temps monotone, une charge utile et une intégrité. Les événements inconnus sont ignorés selon leur longueur afin de préserver la compatibilité entre versions du firmware.

Un temps monotone mesure l'ordre et la durée depuis un reset ou un point de référence, il ne constitue pas une date civile. Le numéro de séquence distingue deux événements produits au même tick et permet de détecter pertes, répétitions et reprises après reset. Une horloge temps réel peut compléter l'enregistrement, mais ne remplace pas ces compteurs pour reconstruire une chronologie.

```
#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;
};

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 – Format binaire borné d'un événement journalisé

Le CRC détecte les erreurs accidentelles de stockage et de transport ; il ne fournit pas une authentification contre un attaquant capable de réécrire la mémoire. Lorsque le journal doit résister à une falsification, un MAC ou une signature est ajoutée selon le modèle de menace. Les clés ne sont pas stockées dans chaque entrée et les erreurs d'authentification sont distinguées des corruptions accidentelles.

En embarqué, la gestion des journaux est souvent contrainte à l'usage d'un ring buffer. Un ring buffer réserve un nombre fixe d'enregistrements ou de secteurs. Le producteur écrit une entrée, calcule son intégrité, puis publie un marqueur de validité en dernier. Le consommateur ignore une entrée incomplète et recherche la dernière séquence valide. En Flash, l'effacement par secteur et le nombre limité de cycles imposent un wear-leveling : une génération monotone répartit les écritures entre secteurs. Une coupure pendant un effacement ou une programmation doit laisser au moins une copie récupérable.

```
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 – Publication atomique d'une entrée de ring buffer

Cet exemple illustre un protocole et non une implémentation Flash universelle. Le contrôleur peut imposer des écritures par mot, des transitions de bits dans un seul sens et des délais de

lecture après programmation. La récupération vérifie le marqueur, les bornes, le CRC et la progression de séquence avant de rendre l'entrée visible.

9.2 Audit et analyse

L'audit embarqué combine les journaux, les compteurs de faute et une sauvegarde d'état post-mortem. Lors d'une exception, d'un watchdog ou d'une assertion critique, le firmware capture la cause, le compteur de reset, le contexte processeur, l'adresse d'instruction, l'état des registres de contrôle, les limites de pile, la version et l'empreinte de l'image. Les buffers contenant des secrets sont exclus, chiffrés ou effacés selon le modèle de menace.

La sauvegarde est bornée et réentrante. Un handler de faute n'appelle ni allocateur, ni périphérique non fiable, ni fonction dépendant d'une pile déjà corrompue. Il écrit un en-tête avec une longueur et une version, copie les registres disponibles, calcule une intégrité, puis publie un marqueur de validité en dernier. Si la capture échoue, un code minimal de cause et un compteur de reset restent conservés.

```
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 – Cadre minimal de sauvegarde post-mortem

Au redémarrage, le code valide le magic, la version, la taille, le CRC et le marqueur avant de remettre le dump à l'analyse. Il distingue un dump nouveau d'un dump déjà consommé et em-

pêche une boucle de redémarrage de réécrire indéfiniment la même zone. Les outils post-mortem associent l'adresse de faute à l'ELF et aux symboles de la version exacte, un simple numéro de version textuel ne suffit pas.

A retenir

Une capture post-mortem doit être courte, réentrante et validée avant analyse. Elle ne doit ni dépendre d'un allocateur, ni attendre un périphérique non fiable, ni réutiliser un contexte corrompu.

9.3 Processus de mise à jour

A retenir

La sécurité d'un firmware est une propriété de chaîne : conception, compilation, démarrage, exécution, journalisation, mise à jour et analyse post-mortem doivent conserver les mêmes contrats et les mêmes preuves.

Une mise à jour de firmware est une transition contrôlée entre deux images, et non une simple copie de fichier. Le produit définit la version, le format, les tailles, les régions Flash, les métadonnées, les états de progression et la reprise après perte d'alimentation. Le guide H2Lab consacré à une stratégie de mise à jour à bascule complète ces exigences de déploiement et de reprise [19].

Une architecture à deux images réserve une image active et une image candidate, avec des métadonnées séparées. Le bootloader écrit la candidate dans une zone inactive, vérifie ses limites et son intégrité, puis publie l'état *candidate-ready*. Après un redémarrage, il sélectionne la candidate et exige un acquittement de bon démarrage. Si l'image ne démarre pas, déclenche un watchdog ou ne confirme pas son fonctionnement dans le délai prévu, le bootloader revient à l'image précédente.

La stratégie de mise à jour vérifie au minimum :

- la compatibilité matérielle et la version minimale du bootloader ;
- la taille, l'adresse de chargement, l'alignement et les régions réservées ;
- l'intégrité de l'image et, lorsque le modèle de menace l'exige, son authenticité ;
- la monotonie de version et la protection contre le retour à une image interdite ;
- la cohérence des métadonnées après coupure pendant l'effacement ou la programmation ;
- la conservation de l'ELF, de la carte de lien, de la SBOM et de l'empreinte livrée.

Une mise à jour différentielle réduit le volume transféré mais ajoute un moteur de reconstruction et de nouvelles preuves de bornes. Elle n'est retenue que si la reconstruction est déterministe, vérifiable et repriseable. Les secrets de transport, les clés d'authentification et les journaux de mise à jour suivent un cycle de vie distinct ; un journal ne doit pas devenir un moyen de modifier silencieusement la politique de démarrage.

Après installation, le firmware vérifie la version active, l'état des périphériques, les partitions et les migrations de données persistantes. Une migration doit être transactionnelle ou réversible. Les tests couvrent l'effacement interrompu, la programmation partielle, la corruption des métadonnées, le watchdog, la perte de communication et le retour arrière. La livraison associe finalement l'image à sa configuration de construction, ses preuves et son empreinte afin qu'un appareil terrain puisse être relié à un artefact contrôlé.

A retenir

Une mise à jour robuste conserve une image de repli, publie l'image candidate seulement après vérification et revient à l'image précédente si le démarrage n'est pas confirmé.

10
Conclusion

Ce guide a présenté une démarche complète pour développer et livrer un firmware embarqué en C dans un contexte contraint ou critique. Le fil directeur est le contrat : contrat du langage, de l'ABI, du processeur, du SoC, des périphériques, des interfaces et des outils. La portabilité ne consiste pas à nier les différences entre ARM, RISC-V et PowerPC, mais à les isoler dans des couches documentées et à vérifier les propriétés dépendantes de la cible.

Le démarrage, la mémoire, les registres, les caches, les interruptions et les architectures multicoeurs ont été reliés à une même exigence : chaque accès partagé doit avoir un propriétaire, un format, une borne, un ordre et un comportement d'erreur défini. Les atomiques, barrières, mailbox, DMA, MPU et ordonnanceurs sont des moyens de réaliser ces contrats ; aucun ne remplace à lui seul l'analyse du matériel et du code généré.

La sécurité repose ensuite sur une chaîne traçable. Les menaces et les actifs sont identifiés, les composants sont recensés dans la SBOM, les interfaces sont relues, les tests couvrent les décisions et les erreurs, et les preuves noRTE explicitent leurs hypothèses. Les mécanismes d'autoprotection, les journaux, les sauvegardes post-mortem et les mises à jour réversibles prolongent cette démarche jusqu'au produit livré et maintenu sur le terrain.

L'intérêt pratique de ce guide est de fournir une grille de lecture utilisable à plusieurs niveaux : le développeur peut vérifier une fonction et ses bornes, l'équipe de plateforme peut contrôler le BSP, la carte mémoire et les barrières, l'architecte peut examiner les frontières et les interférences, l'auditeur peut relier exigences, code, tests, preuves et binaire. Cette continuité transforme la qualité et la sécurité en propriétés vérifiables plutôt qu'en intentions isolées.

Une qualification sérieuse reste nécessaire pour chaque produit. Les exemples du document sont des modèles de raisonnement et des points de départ : ils doivent être adaptés au SoC, au compilateur, au matériel, au niveau d'assurance et au modèle de menace retenus. La méthode est néanmoins stable : réduire les hypothèses, expliciter les contrats, mesurer les comportements, prouver ce qui peut l'être, tester le reste et conserver les artefacts qui permettent de refaire l'analyse.

A

Checklist des bonnes pratiques

Cette checklist transforme les recommandations du guide en points de contrôle vérifiables. Elle ne remplace ni l'analyse de risque ni les exigences propres au produit. Pour chaque ligne, le projet renseigne l'état, le responsable, la preuve associée et, si nécessaire, la justification d'une non-applicabilité.

Domaine	**Point de contrôle**	**État / preuve**
Langage C	Le profil C, les extensions autorisées et les options du compilateur sont versionnés.	☐
Langage C	Les types, conversions, tailles et formats sont adaptés à l'ABI et aux interfaces.	☐
Langage C	Les comportements indéfinis et les conversions susceptibles de perdre de l'information sont analysés.	☐
Langage C	Les fonctions critiques ont des préconditions, postconditions et contrats d'erreur explicites.	☐
Complexité	La complexité cyclomatique et la convention de comptage sont documentées.	☐
Complexité	Les décisions, chemins d'erreur et nettoyages sont couverts par des tests.	☐

Table A.1 – Checklist du langage, des contrats et de la testabilité.

Domaine	Point de contrôle	État / preuve
Mémoire	La carte mémoire, les régions réservées, les attributs de cache et les alignements sont documentés.	☐
Mémoire	Les sections ELF, la pile, le tas, les buffers DMA et les vecteurs sont placés par le script de lien contrôlé.	☐
Mémoire	Les accès MMIO utilisent une largeur explicite et respectent les effets de lecture/écriture du périphérique.	☐
Mémoire	Les opérations de cache et les barrières sont définies pour chaque domaine de partage.	☐
Mémoire	Les régions MPU/SMMU appliquent les droits attendus et la politique W xor X.	☐
Démarrage	La BootROM, le bootloader, le reset vector et le reset handler sont documentés.	☐
Démarrage	La pile initiale, les tables de vecteurs et les registres de contrôle sont vérifiés avant le code C.	☐
Démarrage	Les transitions entre images définissent l'état des interruptions, des périphériques et du contexte.	☐

Table A.2 – Checklist mémoire, démarrage et protections matérielles.

Domaine	Point de contrôle	État / preuve
Concurrence	Chaque objet partagé possède un propriétaire, un format et un protocole d'accès.	☐
Concurrence	Les courses, TOCTOU, double-fetch et callbacks réentrants sont analysés.	☐
Concurrence	Les atomiques, verrous, IPI, mailboxes et FIFO ont une borne de progression.	☐
Concurrence	Les ISR ne bloquent pas et les registres de contexte sont préservés.	☐
Multicoeur	Les domaines de cohérence, affinités d'interruptions et accès inter-clusters sont documentés.	☐
Multicoeur	Les buffers échangés entre coeurs homogènes ou hétérogènes ont une propriété et une durée de vie explicites.	☐
Ordonnancement	Les priorités, affinités, temps de blocage et risques d'inversion sont analysés.	☐

Table A.3 – Checklist concurrence, multicoeur et ordonnancement.

Domaine	Point de contrôle	État / preuve
Sécurité	Les actifs, menaces, frontières et objectifs de sécurité sont identifiés.	☐
Sécurité	Les privilèges, responsabilités et surfaces d'attaque sont minimisés.	☐
Supply chain	Les composants, versions, licences, provenances et vulnérabilités sont recensés dans la SBOM.	☐
Interfaces	Les formats, tailles, erreurs, timeouts, ownership et effets de bord sont documentés.	☐
Autoprotection	CFI, canaries, protections de pile, MPU et mécanismes de trap sont vérifiés sur le binaire.	☐
Tests	Les tests unitaires, intégration, sur cible, fautes et couverture sont archivés.	☐
Preuve	Les contrats ACSL, hypothèses, obligations, exclusions et résultats noRTE sont reproductibles.	☐

TABLE A.4 – Checklist sécurité, supply chain, tests et preuves.

Domaine	Point de contrôle	État / preuve
Journaux	Le format binaire, la version, les longueurs, les séquences et le temps monotone sont spécifiés.	☐
Journaux	Le CRC, le marqueur de commit, le ring buffer et le wear-leveling résistent aux coupures.	☐
Post-mortem	Le dump de faute est borné, réentrant, intègre et associé à l'ELF exact.	☐
Mise à jour	L'image candidate est vérifiée avant activation et une image de repli reste disponible.	☐
Mise à jour	Les métadonnées, migrations, watchdogs, anti-rollback et retours arrière sont testés.	☐
Livraison	ELF, carte de lien, SBOM, preuves, empreintes et image programmée sont archivés ensemble.	☐

TABLE A.5 – Checklist journalisation, diagnostic et mise à jour.

Bibliographie

[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.

Licence