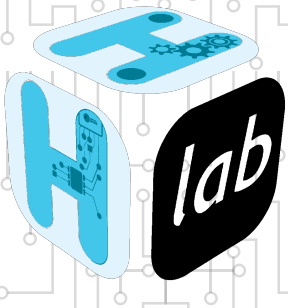
H2LAB-STIG-MPU-001
05-2026

Embedded Systems: Guide to Physical Memory Protection



STIG H²Lab

Information

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

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

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

Contents

1
Introduction and perimeter

1.1 Objective of the guide

This guide focuses on two profiles:

- embedded cybersecurity engineers who must evaluate, justify and audit a memory protection strategy;
- Bare-metal developers and firmware experts (with or without RTOS) who design and implement the MPU configuration as close as possible to the hardware.

It is written to meet an operational need: to move from equipment without memory protection to a system where the MPU is "used as security control, with defensible logic in technical review, cyber audit and product validation. The objective is not only to describe registries, but to link physical mechanisms to concrete threats, software architecture choices and verification criteria.

The guide covers:

- the generic MPU principles applied to microcontrollers;
- the significant variations between ARM Cortex-M, PowerPC 32 bit MPCxx and RISC-V with PMP;
- the construction of a cybersecurity-oriented memory protection policy; Countermeasures against attacks of memory corruption, unauthorized execution and elevation of privilege;
- technical validation of the MPU configuration (tests, reviews, acceptance criteria).

The guide does not cover, or only in interface:

- full operational safety in the normative sense (ISO 26262, IEC 61508, etc.), excluding points of contact with cybersecurity;
- advanced physical security (invasive attacks, advanced material fault injection, complex channel side);
- the comprehensive details of each manufacturer reference (complete manuals per variant), which remain to be processed in the official documentation of the material target.

In summary, this document provides a technical framework for designing, implementing and verifying a robust use of the MPU, explaining what it allows to protect, what it does not protect, and how to integrate it into a coherent on-board cyber strategy.

2

History and evolution of embedded memory protection

2.1 Historical background

For a long time, much of the 8/16 bit microcontrollers were designed with a monolithic execution model: a single software image, a flat memory space, and little or no hardware isolation between critical and non-critical functions. This model responded to cost and performance constraints, but left a major blind spot: local memory corruption could quickly become a global compromise of the system.

The gradual introduction of memory protection mechanisms into embedded cores was driven first by robustness and safe operation, then by cybersecurity. On the ARM side, the arrival of the MPU on the generations Cortex-M3/M4 and Cortex-M7 made possible pragmatic regional isolation on systems without complete MMU [2, 4, 26, 27]. On the automotive side, the PowerPC e200 families (e.g. NXP MPC57xx) have structured memory partitioning and access control approaches adapted to real-time hard constraints, with a strong proximity between safety and security [19, 20].

From ARMv8-M, the combination of MPU and security domain separation (TrustZone-M) was an important step: memory protection is no longer just a defense against accidental software errors, but a central mechanism for reducing attack surface between trusted worlds [5, 3]. This logic is found in recent families such as STM32U5 or NXP S32K3, where the architecture explicitly aims to integrate the product [28, 21] cybersecurity requirements.

In parallel, the ecosystem converged on similar concepts on other embedded ISAs. The physical memory protection (PMP) of RISC-V illustrates this convergence: regional granularity, privilege access policies, and anchoring in a global hardening strategy [24, 25]. This trajectory confirms a key point for cyber engineering: embedded memory protection has become a basic control, at the same level as secure boot, privilege management and debug access control.

2.2 MPU vs MMU

The MPU/MMU distinction is crucial to frame what can be required of a cybersecurity microcontroller.

A MMU (Memory Management Unit) typically provides translation of virtual-to-physical addresses, fine pagination and a strong isolation between processes, at the cost of a higher hardware and software complexity (page tables, TLB, virtual memory management). It is suitable for rich systems (Linux, hypervisors), but rarely relevant for most real time MCUs with limited resources.

A Memory Protection Unit, on the other hand, does not implement virtual memory. It applies access policies to areas of physical address: reading, writing, execution, level of privilege, and sometimes memory attributes. This model is simpler, deterministic and compatible with the latency constraints of bare-metal/RTOS [2, 5, 6].

In cybersecurity, this implies:

- excellent ability to contain faults and many attacks of memory corruption *si* partitioning is rigorous;
- a strong dependence on configuration quality (regional size, priorities, default policy);
- the absence of certain properties offered natively by an MMU (virtual address, isolation fine process per page).

In practice, for platforms such as STM32F4/H7, SAM E70, S32K3 and MPC57xx, the MPU is not a "degraded version of the MMU: it is a different mechanism, optimized for the critical embark, which must be designed as a barrier for containment and privilege control within a defence architecture in depth [26, 27, 17, 21, 19].

3

Architecture calls: what an MPU actually protects

3.1 Fundamental Mechanisms

3.1.1 Memory regions

The central concept of any MPU is *region*: a segment of the physical address space associated with access properties. Each region is defined by a base address, size and set of attributes. The number of configurable regions simultaneously is a fixed hardware constraint: 8 or 16 for most Cortex-M [2, 5], typically 16 to 24 for PowerPC e200 cores [19], and 4 to 64 for RISC-V PMP [24].

Alignment constraints vary according to architecture: in ARMv7-M, each region must be aligned with a multiple of its own size (power constraint of two), which requires compromises in the cutting of memory space [2, 4]. ARMv8-M removes this constraint by allowing regions aligned with 32 bytes, offering significantly finer granularity [5, 6]. PowerPC and RISC-V implementations have their own minimum granularity and alignment rules to check in the documentation of each variant [19, 24].

An unconfigured region (or in an implementation without a default background region) causes a fault on any access. This *deny-by-default* behavior is the safest mode: only explicitly allowed memory is accessible. Its effective activation depends on the background policy adopted (see section 3.2).

3.1.2 Access permissions: reading, writing, execution

Each region has a triplet of permissions:

- **Read (R)**: permission to read by the processor or a bus master.
- **Writing**: permission to modify the content of the region.
- **Execution (X) or *Execute Never* (XN)**: permission or prohibition to recover instructions from this region. The XN bit (or its equivalent) is one of the most structuring protections: it prohibits a data zone from being used as a code zone, blocking an entire class of operations.

In practice, a robust policy applies the principle of less privilege on these three dimensions: the code zones (Flash) are marked R+X but not W, the data zones (RAM, stack) are marked R+W but XN, and the mapped memory devices (MMIO) only receive the strictly necessary accesses. Any deviation from this principle constitutes an attack surface [2, 5].

3.1.3 Privacy levels

The MPU operates in direct interaction with the processor's execution levels. On Cortex-M, two levels coexist: *privileged* (Privileged) and *nonprivileged* (Unprivileged). The access attributes of a region can be differentiated according to this level, allowing for example an area to be accessible in read-write by privileged code (RTOS kernel, drivers) but inaccessible or read-only for user code [2, 6].

This separation is fundamental to cybersecurity: it materially translates the boundary between areas of trust. On ARMv8-M with TrustZone-M, a third dimension is added — secure world / unsecured world — allowing partitioning cryptographic assets or supply secrets beyond what the MPU alone can offer [5, 3].

On PowerPC e200 cores, supervisor/user separation (modes `PR` de `MSR`) plays a similar role, with memory protection inputs (IVOR, `SPROT` and attributes `UM`) defining the rights by mode [19].

3.1.4 Regional recovery and prioritization

When multiple regions overlap over the same address, the UPM applies a deterministic priority rule. In ARMv7-M and ARMv8-M, the highest number region prevails [2]. This rule allows you to define a wide region with default permissions, then superimpose narrower and more restrictive regions for sensitive areas (e.g., protect a section of RAM by first marking all R+W RAM and then superimposing an XN+RO region on the vector table).

This mechanism is powerful but a source of error: unintentional overlap can silently cancel a security restriction. Verifying overlaps is part of mandatory configuration reviews (see chapter 12). However, it is not advisable to use this type of methodology in order to avoid any uncontrolled side effects. It is preferable to cut the regions in a non-covery way, even if several regions are used to cover a critical area.

3.1.5 Exceptions of memory error

Any violation of the MPU policy triggers a processor exception. On Cortex-M, this is the *MemManage Fault* (a dedicated vector in the NVIC table), which allows the exception handler to analyze the cause via the registers `CFSR` (*Configurable Fault Status Register*) and `MMFAR` (*MemManage Fault Address Register*) [2, 26, 27].

From the cybersecurity point of view, this mechanism is as much a detection tool as a barrier. A properly implemented fault manager must:

- record the context of the violation (faultful address, criminal investigation, method of execution);
- decide on a proportionate reaction: task reset, switch to degraded mode, secure system shutdown (*fail-safe*);
- never allow the silent prosecution of the execution after a violation, unless such conduct is explicitly justified and isolated.

An incomplete or absent fault manager turns a security barrier into a simple non-response detection mechanism, which is insufficient in an active threat model [5, 6].

On PowerPC e200, memory protection violations generate exceptions *Data Storage* or *Instruction Storage* (IVOR2/IVOR3), with dedicated status registers (`ESR`, `DEAR`) allowing a similar analysis [19].

3.2 Intrinsic limitations

The MPU is a memory access control mechanism, not a logical consistency control. It is essential to clearly define what it actually protects.

3.2.1 What the MPU does not protect

Intra-region attacks. The MPU does not distinguish legitimate access from malicious access *within the same region*. A buffer overflow remaining in the same RAM area will not be detected. The granularity of protection is that of the regions: the wider the regions, the rougher the protection.

Applicative logic. The MPU does not validate the behavior of the code allowed to run. A legitimate code can produce side effects (information leak, incorrect use of an API) without ever triggering an MPU violation. It does not replace validation of entries or logical integrity checks.

Uncontrolled DMA accesses. The processor MPU only applies to accesses initiated by the core. DMA transfers operate as independent bus masters: without bus-level protection (XRDC, SMPU, system MPU depending on vendor terminology), a poorly configured or compromised DMA can reach sensitive areas without triggering the MPU [19, 21]. This is discussed in detail in Chapter 11.

Physical attacks and auxiliary channels. The MPU does not protect against the injection of material errors, auxiliary channel attacks (consumption analysis, EM) or direct access to memory by unlocked debug interfaces.

The code outside the protection model. If the boot charger (*bootloader*) or boot code (*startup*) runs before the MPU is activated, it is not subject to any restrictions. The time window before activation is an attack surface to be treated separately (secure boot, early locking).

3.2.2 Conditions necessary for real efficiency

The MPU is only effective if several conditions are simultaneously met:

1. **Activation effective.** The MPU must be explicitly activated (bit `ENABLE` in `MPU_CTRL` on Cortex-M). It is disabled by default at the ignition [2]. One of the most common errors is forgetting this initialization step.

2. **Restrictive background policy.** In the absence of a corresponding region, the default behavior must be the total ban (*background region* disabled or configured as *no access*). A permissive background policy cancels the safe value of the entire configuration.

3. **Coherence of configuration throughout the life cycle.** The MPU can be dynamically reconfigured to run. Any reconfiguration must be carried out in a privileged context and validated. An uncontrolled reconfiguration (e.g. via an attack vector that has obtained the execution of privileged code) can dismantle the entire security policy.

4. **Operational fault manager.** Without *MemManage Fault* manager implemented and tested, MPU violations trigger unspecified behavior or a *HardFault* of withdrawal, whose reaction may be unsafe.

5. **Synchronization with other protection mechanisms.** The MPU must be integrated into a broader strategy including secure boot, debug interface locking and RTOS privilege management. Isolated, it is a necessary but insufficient control.

In summary, the MPU is an effective structural barrier against the spread of memory corruption and against a broad category of corruption attacks of execution flows, provided its configuration is rigorous, complete and maintained throughout the entire life cycle of the product [2, 5, 19, 24].

4

Threat model for MCU embedded systems

4.1 Assets, attackers, and vectors

4.1.1 Actives to be protected

The construction of a coherent threat model begins with the identification of assets whose compromise results in a significant security impact. For an MCU embedded system, five categories of memory assets are distinguished:

Executable code (firmware). The binary image stored in Flash or ROM is the fundamental asset: its partial modification or replacement allows an attacker to inject arbitrary behavior. Code writing protection, combined with secure boot, forms the first line of defense [2, 5].

Configuration data and secrets. Cryptographic keys, calibration parameters, provision IDs, certificates: these data are often stored in Flash or protected RAM. Unauthorized reading allows identity usurpation or product cloning; their modification leads to incorrect behaviour or deactivation of safety mechanisms.

Execution control structures. The table of interrupt vectors, the execution stack (handler and thread), the internal structures of the RTOS (task control blocks, scheduling lists, mutex), function pointers: these elements direct the execution flow. Their corruption is the main objective of *control-flow hijacking* [29, 11].

Memory of devices (MMIO). The device configuration registers, accessible via fixed addresses in the physical memory space, allow to control the hardware behavior: activation of the DMA, reconfiguration of clocks, access to debug interfaces, modification of Flash protections. Their unrestricted accessibility is a major pivot vector in attacks on industrial systems.

Update and Start Region. Bootloader and OTA mechanisms are high-value assets: an attacker able to alter their behaviour gets a persistence of restarting and full access to the system. The Triton/TRISIS [12] and Stuxnet [14] incidents illustrate how the compromise of the lower layers of an on-board control system can remain undetected for months.

4.1.2 Attacker profiles

The safety literature of industrial and on-board systems generally distinguishes three attacker profiles, characterized by their access capabilities and technical capabilities [15, 13].

Remote attacker (network or bus). It has a connection to exposed communication interfaces: Industrial Ethernet, CAN, LIN, BLE, Modbus/TCP, etc. It does not have physical access to the equipment and its input vector necessarily passes through the message processing code received. Miller and Valasek's work on the Jeep Cherokee (2015) [18] illustrates this profile: the initial access via the Uconnect cellular interface allowed to reach the internal CAN bus and then to control control units (ECU) without any physical contact with the vehicle. This taxonomy has been formalized for on-board car systems by Checkoway *et al.* [9], whose systematic analysis of attack surfaces remains a reference for the threat modelling of connected on-board systems.

attack with limited physical access. It can connect equipment to exposed or semi-exposed physical interfaces: JTAG/SWD connector, UART maintenance port, CAN beam interface, USB port. Its capabilities include direct reading of memory if debug protections are not activated, step-by-step debugging and possibly injection of minor errors. Keen Security Lab's search for Tesla [16] shows how the combination of limited physical access and inadequately locked interfaces can provide a pivot for more sophisticated remote attacks.

Co-located attacker (compromised code). He has already obtained the execution of unprivileged code on the MCU — by exploiting a software vulnerability, injection of firmware or compromise of the supply chain. Its objective is to climb to the privileged context, read assets in other memory regions or modify control structures. This is the profile against which the MPU offers the most direct protection, as Clements *et al.* [11, 10]: on representative bare-metal systems, the lack of privilege control systematically leads to complete arbitrary execution after successful memory corruption.

4.1.3 Memory attack sensors

Exploitation of software vulnerabilities

Memory corruption vulnerabilities remain the most documented attack family on embedded systems. Szekeres *et al.* [29] have formalized this class in their systematization: stack buffer overflows (*stack buffer overflow*), heap zone exceedances (*heap overflow*), invalid pointer dereferences and post-release uses (*use-after-free*). On the MCU bare-metal without standard dynamic allocator, the first two forms largely dominate.

The particularity of MCU flat physical address space architectures aggravates the impact of these vulnerabilities: successful corruption on the stack can directly overwrite the return address and redirect execution to an arbitrary area, without any physical barrier intervening in the absence of MPU. On implementations without stack insulation, only one off-limit write is enough to get total control of the execution stream [29].

Unlocked debug interfaces

The open maintenance of JTAG/SWD interfaces is a complete compromise vector, independent of any software vulnerability. An attacker with physical access to these interfaces can read the entire Flash and RAM, modify the execution in real time and extract the supply secrets. Many industrial equipments remain deployed with minimum reading protections (*read-out protection*)

or disabled [13]. The search at Keen Security Lab [16] illustrates how this vector can be combined with software to obtain persistent access to the system.

Even when *read-out protection* is enabled, it should not be considered an absolute mechanism. On several STM32 families, public works have shown practical circumventions of the intermediate level RDP (often *Level 1*) via error injection (glitch voltage/clock), with partial or total recovery of Flash content according to silicon revision and activated countermeasures [1, 22]. On the other hand, the strictest levels (e.g. RDP2 on certain ranges) greatly increase the cost of attack but at the cost of significant operational impacts (debogage and maintenance). In practice, therefore, the RDP should be treated as a physical hardening brick, to be combined with secure boot, update authentication and strict life cycle management of debug [26, 27, 23].

Update mechanisms

OTA mechanisms and production updating procedures are high impact vectors. An unauthenticated update or one whose verification chain is bypassable allows the firmware to be replaced. The Stuxnet [14] attack demonstrated the feasibility of a targeted change in the firmware of a Siemens S7 automaton for discreet industrial sabotage, using a combination of vulnerabilities in engineering tools and update protocols. The Triton/TRISIS [12] incident applied a similar logic to Schneider Electric Triconex security controllers, injecting malicious code into security modules to neutralize critical industrial process protection functions.

Pivots via DMA and peripherals

In systems with DMA or several bus masters, a compromised or misconfigured device can access RAM directly without going through the processor core, completely bypassing the processor MPU. This vector is discussed in detail in Chapter 11.

4.1.4 Privilege escalation: from entry vector to memory compromise

The typical operating chain on a memoryless MCU follows a predictable sequence:

1. **Initial access**: operating an exposed interface (network, bus, debug) to reach a vulnerable parsing code.
2. **Memory corruption**: triggering a buffer overflow or an off-limit write, allowing to reach a control structure (return address, function pointer, vector table input).
3. **Redirecting execution stream**: the corrupted structure directs execution to a payload controlled by the attacker (injected in RAM or ROP/JOP string in legitimate code) [29].
4. **Arbitrary execution in privileged context**: on an MCU without MPU, the injected code inherits the privilege level of the interrupted context — often fully privileged on bare-metal systems — giving full access to system resources.

The MPU acts as a two-level barrier in this chain:

- between steps 2 and 3, restricting areas accessible in writing from the current context (protection of the table of vectors, stack, RTOS structures);
- between steps 3 and 4, via the XN bit that prevents execution from data areas, blocking direct injection of shellcode into RAM.

The study by Clements *et al.* [11] on a corpus of bare-metal embedded binaries confirms that virtually all systems deployed without configured MPUs offer a trivial attack surface once initial access is obtained, and that privilege partitioning significantly reduces the reachable perimeter from a compromised task.

4.2 Security objectives

4.2.1 Formulation of memory protection objectives

From the previous threat model, a set of security objectives that can be directly addressed by the MPU configuration is derived. These objectives are formulated in a verifiable way: it should be possible to test or demonstrate them (see Chapter 14).

OS-1 — Confiscation of memory corruption. Memory corruption occurring in the context of a given software component must not be able to modify memory belonging to another component or kernel/RTOS. *MPU mechanism concerned : partitioning RAM regions by domain, write permissions restricted by execution context.*

OS-2 — Integrity of execution flow. No data area (RAM, stack, heap, receiving buffers) shall be used as a source of instructions executed. *MPU mechanism concerned : systematic XN attribute on all data regions, execution authorization limited to Flash code areas.*

OS-3 — Protection of critical control structures. The interrupt vector table, RTOS control structures and areas containing function pointers must not be editable from a non-privileged context or from an isolated application task. *MPU mechanism concerned : writing permissions reserved for the privileged context, regional recoveries restricting write access to sensitive areas.*

OS-4 — Restrictions on MMIO access. Access to the configuration records of sensitive devices (DMA controller, debug interfaces, Flash configuration, security records) should be limited to the preferred low-level code. *MPU mechanism concerned : MMIO regions with privileged access only, access ban from application tasks.*

OS-5 — Detection and Controlled Reaction. Any attempt to violate previous objectives must trigger a material exception whose treatment is deterministic, traceable and oriented towards a safe state. *MPU mechanism concerned : MemManage Fault manager (or equivalent) implemented, tested and integrated into the response strategy.*

4.2.2 Correspondence with Threat Model

Table 4.1 matches identified attacker profiles, associated vectors, and security objectives addressed by the MPU.

4.2.3 Residual device not covered by the MPU

The objectives OS-1 to OS-5 cover threats that the MPU can address directly. Three risk categories remain outside this scope and require additional controls (see Chapter 15):

- **Debug and Update Interfaces** : secure JTAG/SWD interfaces, debug access deactivation or authentication, and cryptographic verification of update images are the responsibility of

Profile attacker	Vector attack	Impact without MPU	OS address
Remote (network/bus)	Vulnerable Parsing (stack overflow)	Full arbitrary execution	OS-1, OS-2, OS-3
Remote (network/bus)	Structure corruption RTOS	Denial of service, elevation	OS-3
Physical access	Unlocked debug interface	Total reading/writing	Out of perimeter MPU
Physical Access	Unauthenticated Update	Malignant Persistence	Out of Field MPU
Co-located	Climbing from Unprivileged Task	Access to Secrets, Core Pivot	OS-1, OS-3, OS-4
Uncontrolled DMA system/DMA	Access	Contouring MPU processor	OS-4 + bus protection

Table 4.1: Correspondence between attacker profiles, vectors and MPU security objectives

secure boot and life cycle management [5, 3]. These controls are complementary to the MPU but not substitutable.

- **Advanced Physical Attacks** : Injecting material errors (voltage or clock glitching), auxiliary channel attacks and semi-invasive attacks exceed the MPU's protective model [13]. They are covered by additional protections documented in CC EAL and sectoral standards [15].
- **Logic errors in the allowed code** : a code correctly placed in an authorized region can produce adverse effects (information leak, incorrect use of an API, default control logic) without ever triggering MPU violation. Functional validation and static analysis remain essential [29].

5
Memory attack surface mapping

5.1 Critical areas

The mapping of memory attack surfaces consists of linking each address area to three operational questions: *which can access it*, *with which rights* and *which impact in case of compromise*. The aim is not to draw up an exhaustive list of addresses, but to produce a usable view for the MPU configuration, negative tests and security review [9, 11, 15].

The priority areas to be classified are:

- **Executable code** (bootloader, kernel, application): risk of write alteration and execution outside planned perimeter.
- **Critical data** (keys, security configuration, control state): risk of disclosure and falsification.
- **Pile and Dynamic Zones**: Risk of corruption of control streams (returns, function pointers, RTOS structures).
- **Table of vectors and handlers** : risk of usurpation of interruptions and pivot to privileged code.
- **MMIO and DMA** controllers: risk of circumvention of application logic and indirect access to memory.
- **Debug and Update Interfaces**: Risk of overall reading/writing of firmware off nominal path.

5.1.1 Case 1: sequential bare-metal (single thread)

In a bare-metal architecture without competing application tasks, software functions run in sequence in the same privilege context. This reduces the complexity of scheduling, but focuses the risk: memory corruption in a function can contaminate the following functions, as they share the same overall RAM and the same execution stack.

Figure 5.1 highlights two particularly sensitive areas in bare-metal:

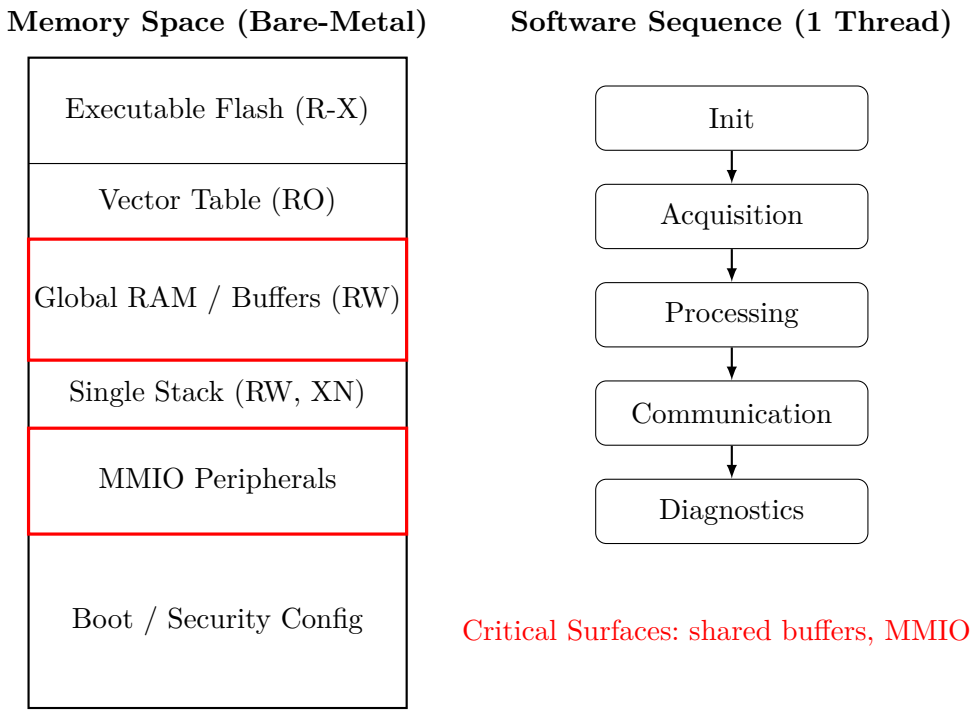


Figure 5.1: Memory attack surface mapping in sequential bare-metal mode

- **shared RAM buffers** between functions (acquisition, communication, diagnosis), which become vectors of transversal propagation;
- **the MMIO** zone, where uncontrolled writing can reconfigure critical devices (clock, debug, DMA).

For this profile, mapping must explicitly identify which functions handle these areas, even in the absence of a multitache. This allows you to cut the MPU policy into minimum regions (code, stack, sensitive data, MMIO) and avoid a *flat mapping* that is too permissive.

5.1.2 Case 2: partitioned mode with applications (RTOS, multi-domains)

In partitioned mode, the attack surface is wider (several tasks, more context transitions, more interfaces), but it becomes more manageable if each application domain has its own memory space and explicit rights. The basic principle is to transform a local compromise into a local incident, without systematic escalation to the kernel or other applications [10, 5].

Figures 5.2 and 5.3 illustrate the key control points:

- **Insulation App A / App B** in RAM (no direct cross access);
- **Prior call gateway** (SVC/syscall) as the only route to kernel services;
- **reconfiguration MPU to context change**: kernel profile during service, then profile of the planned task;
- **MMIO whitelist** with separation between authorized and prohibited registers for non-privileged tasks;
- **minimum shared area** for inter-task exchanges, explicitly bounded and audited.

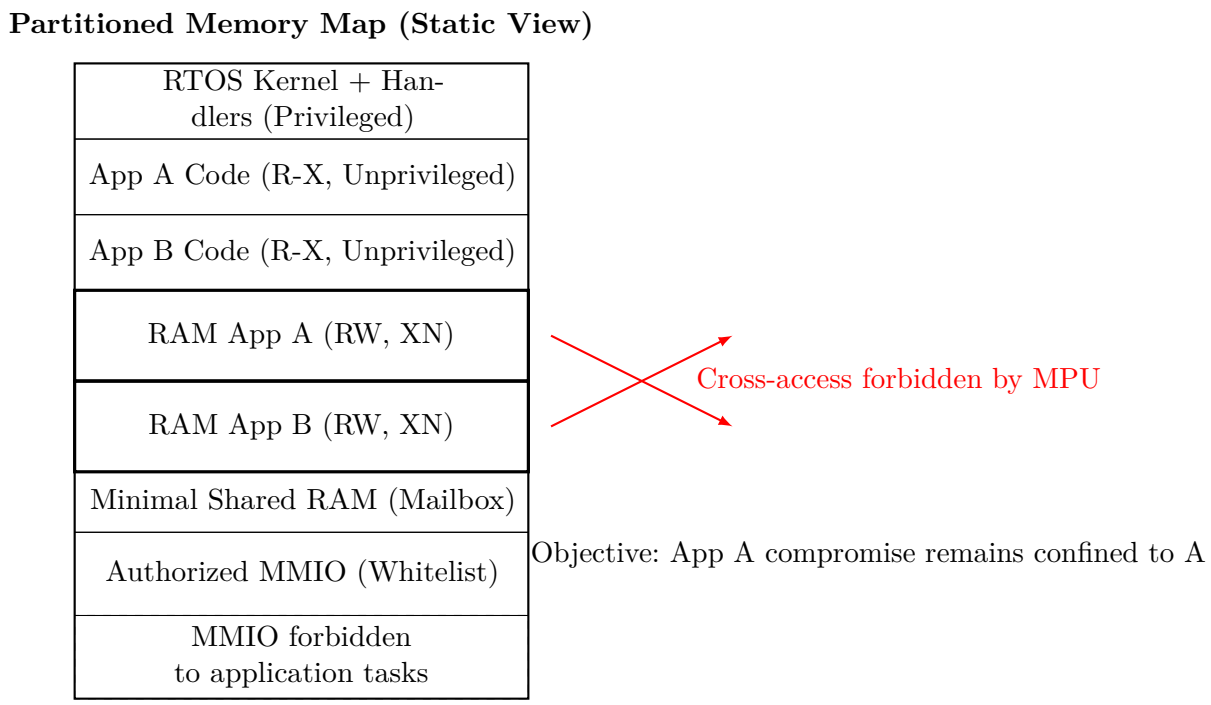


Figure 5.2: Memory mapping in partitioned mode type RTOS

This mapping must be maintained at the level of requirement and implementation (MPU configuration file, script link, code reviews), otherwise the theoretical partitioning quickly drifts as the product evolves.

5.2 Risk priority

Once areas are identified, prioritization is used to decide where to focus hardening and validation efforts. A simple and reproducible method is to note each area according to three axes:

- **Impact**: Work severity and safety if the area is compromised (from 1 to 5).
- **Exposure**: easy to reach from real input vectors (network, bus, debug, update) (1 to 5).
- **Exploitability**: technical effort to transform access into a useful compromise (from 1 to 5).

An initial score $S = I \times E \times X$ can be used to establish a treatment order, and then adjusted by contextual factors (maturity of countermeasures, detection capability, field exposure). The objective is not absolute mathematical precision, but decision-making coherence between teams.

In practice, a priority area must trigger three minimum actions:

1. **containment measure** in the MPU configuration (right reduction, region separation, XN);
2. **dedicated negative test** (no read/write/execution access);
3. **trace of justification** in the security documentation (requirement, proof of test, decision to accept residual risk).

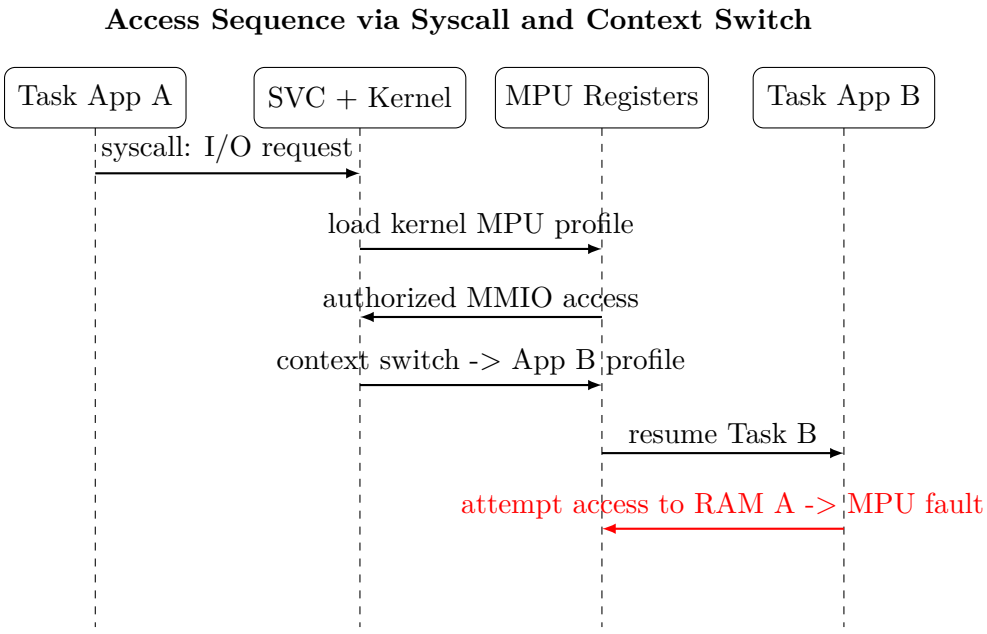


Figure 5.3: Simplified sequence of syscalls and MPU reconfiguration to context change

Memory area	**Impact**	**Exposition**	**Exploit.**	**Score**
Table of vectors / handlers	5	3	4	60
Inter-domain shared RAM	4	4	4	64
MMIO (DMA, debug, flash ctrl)	5	3	5	75
Non-critical application code	3	3	3	27
Non-sensitive log area	1	4	2	8

Table 5.1: Example of prioritisation of memory attack surfaces

This prioritization logic aligns mapping with the following chapters: MPU configuration by architecture, isolation strategy, countermeasures, validation and non-regression security.

6
MPU on ARM Cortex-M: operation and variants

6.1 generic logic Cortex-M

On Cortex-M, memory protection is based on the *Protected Memory System Architecture* (PMSA), i.e. a region-based model of physical addresses, without virtual translation, applied on each instruction or data access [2, 5]. The MPU is therefore a hardware filter between the CPU pipeline and the memory bus:

1. the core calculates a target address (fetch, load/store);
2. the MPU logic determines the corresponding region (or the absence of a region);
3. the rights of the region (R/W/X, privilege, memory attributes) are compared to the current context;
4. in case of violation, an exception *MemManage Fault* is generated.

This model is deterministic and adapted to the real time constraints of the MCU. The robustness of protection depends directly on regional partitioning, the default policy and the quality of fault handlers [4, 6].

In the software ecosystem, CMSIS-Core presents a unified view of MPU registries via the `MPU_Type` structure and utility functions (e.g. `ARM_MPU_SetRegion`, `ARM_MPU_Enable`) [7, 8]. The following register cards contain the typical registers used in practice.

6.1.1 Example implementation assembly ARMv7-M (PMSAv7)

The following example illustrates a minimum initialization in ARMv7-M: read-only executable Flash region, non-executable read/write SRAM region, and then MPU activation with synchronization barriers.

```
.syntax unified
.thumb

```

ARMv7-M PMSA: Typical MPU Registers (CMSIS)

Offset	Nom	Main Role
0x00	TYPE	MPU capabilities (number of regions).
0x04	CTRL	Enable + privileged default policy.
0x08	RNR	Select region N to program.
0x0C	RBAR	Region base address (aligned).
0x10	RASR	Size + AP + XN + attributes + subregions.
0x14	RBAR_A1	RBAR alias for fast loading.
0x18	RBAR_A2	RBAR alias for fast loading.
0x1C	RASR_A1	RASR alias for fast loading.
0x20	RASR_A2	RASR alias for fast loading.

View aligned with `MPU_Type` CMSIS-Core (Cortex-M3/M4/M7).

Figure 6.1: typical MPU records in ARMv7-M according to CMSIS-Core modelling

```
/* Registres MPU ARMv7-M */
.equ MPU_TYPE, 0xE000ED90
.equ MPU_CTRL, 0xE000ED94
.equ MPU_RNR,  0xE000ED98
.equ MPU_RBAR, 0xE000ED9C
.equ MPU_RASR, 0xE000EDA0

/* Exemples d'attributs RASR */
.equ RASR_ENABLE,     (1 << 0)
.equ RASR_SIZE_1MB,   (19 << 1)       /* log2(1MB)-1 */
.equ RASR_SIZE_128KB,(16 << 1)       /* log2(128KB)-1 */
.equ RASR_AP_RO_ALL, (6 << 24)
.equ RASR_AP_RW_ALL, (3 << 24)
.equ RASR_XN,         (1 << 28)

mpu_init_v7m:
  /* 1) Desactiver MPU */
  ldr r0, =MPU_CTRL
  movs r1, #0
  str  r1, [r0]
  dsb
  isb

  /* 2) Region 0: Flash @0x08000000, RO, executable */
  ldr r0, =MPU_RNR
  movs r1, #0
```

ARMv8-M PMSA: Typical MPU Registers (CMSIS)

Offset	Nom	Main Role
0x00	TYPE	MPU capabilities (available regions).
0x04	CTRL	Enable and default policy.
0x08	RNR	Select region N.
0x0C	RBAR	Base + AP/XN/shareability (PMSAv8).
0x10	RLAR	Limit + AttrIndx + enable.
0x14	RBAR_A1	RBAR alias for multi-region loading.
0x18	RLAR_A1	RLAR alias (limit/attribute/index).
0x1C	RBAR_A2	RBAR alias for multi-region loading.
0x20	RLAR_A2	RLAR alias (limit/attribute/index).
0x24	RBAR_A3	RBAR alias for multi-region loading.
0x28	RLAR_A3	RLAR alias (limit/attribute/index).
0x30	MAIR0	Memory attributes index 0..3.
0x34	MAIR1	Memory attributes index 4..7.

View aligned with `MPU_Type` CMSIS-Core (Cortex-M23/M33/M55).

Figure 6.2: MPU records typical in ARMv8-M (PMSAv8) according to CMSIS-Core modelling

```
    str  r1, [r0]

    ldr r0, =MPU_RBAR
    ldr r1, =0x08000000
    str  r1, [r0]

    ldr r0, =MPU_RASR
    ldr r1, =(RASR_ENABLE | RASR_SIZE_1MB | RASR_AP_RO_ALL)
    str  r1, [r0]

    /* 3) Region 1: SRAM @0x20000000, RW, XN */
    ldr r0, =MPU_RNR
    movs r1, #1
    str  r1, [r0]

    ldr r0, =MPU_RBAR
    ldr r1, =0x20000000
    str  r1, [r0]

    ldr r0, =MPU_RASR
    ldr r1, =(RASR_ENABLE | RASR_SIZE_128KB | RASR_AP_RW_ALL | RASR_XN)
    str  r1, [r0]

    /* 4) Activer MPU + default map privilegiee */
    ldr r0, =MPU_CTRL
    movs r1, #5                  /* ENABLE=1, PRIVDEFENA=1 */
```

```
    str  r1, [r0]
    dsb
    isb
    bx   lr
```

This example is deliberately minimal: in production, MMIO hardening, vector table protection, a handler/thread strategy, and systematic negative tests must be added [2, 26, 27].

6.1.2 Example of implementation ARMv8-M assembler (PMSAv8)

The example below shows the typical principle: programming `MAIR0`, creating a Flash region (index attribute 0) and a SRAM XN region (index attribute 1), and then activation MPU.

```
.syntax unified
.thumb

/* Registres MPU ARMv8-M (PMSAv8) */
.equ MPU_CTRL,  0xE000ED94
.equ MPU_RNR,   0xE000ED98
.equ MPU_RBAR,  0xE000ED9C
.equ MPU_RLAR,  0xE000EDA0
.equ MPU_MAIR0, 0xE000EDC0

/* RBAR bits simplifies */
.equ RBAR_AP_RO_ALL, (2 << 1)
.equ RBAR_AP_RW_ALL, (1 << 1)
.equ RBAR_XN,        (1 << 0)

/* RLAR bits simplifies */
.equ RLAR_EN,        (1 << 0)
.equ RLAR_ATTR0,     (0 << 1)
.equ RLAR_ATTR1,     (1 << 1)

mpu_init_v8m:
  /* 1) Desactiver MPU */
  ldr r0, =MPU_CTRL
  movs r1, #0
  str  r1, [r0]
  dsb
  isb

  /* 2) MAIR0: Attr0=0xFF (Normal WBWA), Attr1=0x44 (Normal non-cache
   ) */
  ldr r0, =MPU_MAIR0
  ldr r1, =0x000044FF
  str  r1, [r0]

  /* 3) Region 0: Flash 0x08000000..0x080FFFFF, RO, executable, Attr0
    */
  ldr r0, =MPU_RNR
  movs r1, #0
```

```
    str  r1, [r0]

    ldr r0, =MPU_RBAR
    ldr r1, =(0x08000000 | RBAR_AP_RO_ALL)
    str  r1, [r0]

    ldr r0, =MPU_RLAR
    ldr r1, =(0x080FFFFF | RLAR_ATTR0 | RLAR_EN)
    str  r1, [r0]

    /* 4) Region 1: SRAM 0x20000000..0x2001FFFF, RW, XN, Attr1 */
    ldr r0, =MPU_RNR
    movs r1, #1
    str  r1, [r0]

    ldr r0, =MPU_RBAR
    ldr r1, =(0x20000000 | RBAR_AP_RW_ALL | RBAR_XN)
    str  r1, [r0]

    ldr r0, =MPU_RLAR
    ldr r1, =(0x2001FFFF | RLAR_ATTR1 | RLAR_EN)
    str  r1, [r0]

    /* 5) Activer MPU */
    ldr r0, =MPU_CTRL
    movs r1, #5                 /* ENABLE=1, PRIVDEFENA=1 */
    str  r1, [r0]
    dsb
    isb
    bx   lr
```

The key point of migration v7 to v8 is not mechanically translating the encodings `RASR`. The memory attributes (MAIR), the high terminal (RLAR) and the preferred/syscall strategy must be explicitly rebuilt to maintain equivalent security properties [5, 8].

6.2 Variations by architecture

The main differences between ARMv7-M (PMSAv7) and ARMv8-M (PMSAv8) directly impact the portability of MPU policies.

In ARMv8-M, the presence of TrustZone-M adds an additional partitioning dimension: depending on the SoC core and integration, the configuration can distinguish secure and unsecured domains, with specific rules for access to shared resources [5, 3, 6, 23].

Aspect	ARMv7-M	ARMv8-M
Definition of region	Base + size (`RBAR`/`RASR`)	Base + limit (`RBAR`/`RLAR`)
Granularity	High power alignment constraint of two	more flexible model (explicit limit)
Memory Attributes	Encoding in `RASR` (TEX/C/B/S)	Attribute Index via `MAIR0/MAIR1`
Sub-regions	Available (deactivation by sub-regions)	Not used as in v7, preferred limit logic
Security TrustZone-M	No	Yes (Secure/Non-secure according to implementation)
CMSIS API	`ARM_MPU_*`v7	`ARM_MPU_*`v8 (RBAR/RLAR/MAIR)

Table 6.1: Structuring differences between PMSAv7 and PMSAv8 for MPU configuration

7

MPU on PowerPC MPC57xx: operation and variants

7.1 e200 core MPU: principle and registers

On MPC57xx, the first layer of memory protection is the MPU *instantiated in each e200* core. In concrete terms, each core has its own protection input (programmed via MAS/TLB registers) and applies access rights to its fetish instructions and load/store [19, 20].

This point is structuring: on a multicore SoC, the core MPU policy must be configured for **each core** at startup. A valid configuration on core 0 does not automatically protect the core 1.

The operational principle is as follows:

1. the core resolves an MPU/TLB input corresponding to the accessed address;
2. provides area size, memory attributes and permissions (R/W/X, privilege);
3. in case of violation, the heart triggers an exception *Data Storage* or *Instruction Storage*.

This heart MPU covers the accesses initiated by the e200 heart. Other bus masters (DMA, bus master devices) require additional protection on the SMPU system side.

7.1.1 Example implementation assembler of the heart MPU

The example below shows a minimum programming of a static protection input (TLB1) for a Flash area in play+ execution. The exact bit encodings (size, WIMGE attributes, permissions) must be adapted to the targeted e200 core and product policy [19].

```
/* Exemple simplifie e200 core MPU/TLB - syntaxe GNU as */

.equ MAS0_TLBSEL1_ESEL0,  0x10000000
.equ MAS1_VALID_1MB,      0x80000500   /* exemple: V=1, TSIZE=1MB */
.equ MAS2_EPN_FLASH,      0x00400000   /* + attributs WIMGE si
   necessaire */
```

Register	Name	Main role
SPR 624	MAS0	Selects the TLB/MPU entry to program (bank, index, command).
SPR 625	MAS1	Entry validity, region size, context identifier.
SPR 626	MAS2	Base virtual/effective address and memory attributes (WIMGE).
SPR 627	MAS3	Physical address and access rights (R/W/X supervisor and user).
SPR 944	MAS6	Additional context/protection-ID selection depending on variant.
SPR 945	MAS7	High-address bits for variants with extended address space.
SPR 1015	MMUCSR0	Core MMU/MPU control, global invalidation of entries.
SPR 48	PID0	Process/context identifier used for entry matching.
SPR 688	TLB1CFG	TLB1 capacity (number of available static entries).

Exact names and behavior depend on e200 core (z2/z4/z7) and MPC57xx variant.

Figure 7.1: E200 core records used to program the MPU (MPC57xx)

```
.equ MAS3_RPN_FLASH_RX,   0x0040003D   /* exemple: SR/SX autorises,
    SW interdit */

mpu_core_init_mpc57xx:
  /* 1) Selection entree TLB1[0] */
  lis   r3, MAS0_TLBSEL1_ESEL0@h
  ori   r3, r3, MAS0_TLBSEL1_ESEL0@l
  mtspr MAS0, r3

  /* 2) Definir taille/validite et base effective */
  lis   r3, MAS1_VALID_1MB@h
  ori   r3, r3, MAS1_VALID_1MB@l
  mtspr MAS1, r3

  lis   r3, MAS2_EPN_FLASH@h
  ori   r3, r3, MAS2_EPN_FLASH@l
  mtspr MAS2, r3

  /* 3) Definir adresse physique + droits */
  lis   r3, MAS3_RPN_FLASH_RX@h
  ori   r3, r3, MAS3_RPN_FLASH_RX@l
  mtspr MAS3, r3

  /* 4) Ecrire l'entree dans le TLB */
  tlbwe
  msync
  isync
  blr
```

7.2 SMPU system: generic logic MPC57xx

On NXP MPC57xx families, memory protection is also based on a SMPU (System Memory Protection Unit) mechanism associated with a multi-master bus logic. The model remains regional (base/limit + rights), but the security analysis must take into account both the e200 core and other masters (DMA, accelerators, bus master devices) [19, 20].

The typical access verification sequence is as follows:

1. a master issues instruction or data access;
2. the SMPU selects the corresponding region;
3. the rights associated with the domain/master are compared;
4. in case of refusal, an exception (or an event of violation) is lifted with address and status.

From the cyber point of view, the major interest in a purely CPU-centric MPU is the ability to govern several access initiators. This reduces the risk of indirect bypasses, particularly via DMA, provided that each master's profiles are explicitly configured [19, 21].

Offset	**Name**	**Main role**
0x000	CESR	Global enable, violation status, and IRQ.
0x010	EAR0	Captured faulting address (port 0).
0x014	EDR0	Violation details (rights, master, access type).
0x400	RGD0_WORD0	Region 0 start address.
0x404	RGD0_WORD1	Region 0 end address or mask (depending on variant).
0x408	RGD0_WORD2	Read/write/execute permissions per domain.
0x40C	RGD0_WORD3	Validity, lock, and region control attributes.
0x800	RGD1_WORD0	Region 1 start (same scheme as region 0).
0x804	RGD1_WORD1	Region 1 end or mask.
0x900	RGDAAC0	Alternate access control by master/bus.

Offsets are relative to the SMPU block; exact naming depends on MPC57xx subfamily.

Figure 7.2: Mapping typical SMPU records on MPC57xx

7.2.1 Example of implementation SMPU assembler (MPC57xx)

The example below illustrates a minimum initialization of two regions: Flash in read+ run and SRAM in read/write non-executable, then global activation of protection. The exact offsets and permission bits vary according to subfamily; the schema remains representative of the programming logic [19].

```
/* Exemple simplifie e200/MPC57xx - syntaxe GNU as */

.equ SMPU_BASE,         0xFC010000
.equ SMPU_CESR,         (SMPU_BASE + 0x000)
.equ SMPU_RGD0_W0,      (SMPU_BASE + 0x400)
```

```
.equ SMPU_RGD0_W1,      (SMPU_BASE + 0x404)
.equ SMPU_RGD0_W2,      (SMPU_BASE + 0x408)
.equ SMPU_RGD0_W3,      (SMPU_BASE + 0x40C)
.equ SMPU_RGD1_W0,      (SMPU_BASE + 0x800)
.equ SMPU_RGD1_W1,      (SMPU_BASE + 0x804)
.equ SMPU_RGD1_W2,      (SMPU_BASE + 0x808)
.equ SMPU_RGD1_W3,      (SMPU_BASE + 0x80C)

/* Constantes simplifiees de droits */
.equ CESR_ENABLE,       0x00000001
.equ RGD_VALID,         0x00000001
.equ RGD_PERM_RX,       0x0000002A   /* domaine: lecture+execution */
.equ RGD_PERM_RW,       0x00000033   /* domaine: lecture+ecriture */
.equ RGD_XN,            0x00000100   /* execute-never */

smpu_init_mpc57xx:
  /* 1) Desactiver SMPU pendant programmation */
  lis   r3, SMPU_CESR@h
  ori   r3, r3, SMPU_CESR@l
  li    r4, 0
  stw   r4, 0(r3)
  msync
  isync

  /* 2) Region 0: Flash 0x00400000..0x004FFFFF, RX */
  lis   r3, SMPU_RGD0_W0@h
  ori   r3, r3, SMPU_RGD0_W0@l
  lis   r4, 0x0040
  stw   r4, 0(r3)

  lis   r3, SMPU_RGD0_W1@h
  ori   r3, r3, SMPU_RGD0_W1@l
  lis   r4, 0x004F
  ori   r4, r4, 0xFFFF
  stw   r4, 0(r3)

  lis   r3, SMPU_RGD0_W2@h
  ori   r3, r3, SMPU_RGD0_W2@l
  li    r4, RGD_PERM_RX
  stw   r4, 0(r3)

  lis   r3, SMPU_RGD0_W3@h
  ori   r3, r3, SMPU_RGD0_W3@l
  li    r4, RGD_VALID
  stw   r4, 0(r3)

  /* 3) Region 1: SRAM 0x40000000..0x4001FFFF, RW + XN */
  lis   r3, SMPU_RGD1_W0@h
  ori   r3, r3, SMPU_RGD1_W0@l
  lis   r4, 0x4000
  stw   r4, 0(r3)

```

```
58    lis   r3, SMPU_RGD1_W1@h
59    ori   r3, r3, SMPU_RGD1_W1@l
60    lis   r4, 0x4001
61    ori   r4, r4, 0xFFFF
62    stw   r4, 0(r3)
63
64    lis   r3, SMPU_RGD1_W2@h
65    ori   r3, r3, SMPU_RGD1_W2@l
66    li    r4, (RGD_PERM_RW | RGD_XN)
67    stw   r4, 0(r3)
68
69    lis   r3, SMPU_RGD1_W3@h
70    ori   r3, r3, SMPU_RGD1_W3@l
71    li    r4, RGD_VALID
72    stw   r4, 0(r3)
73
74    /* 4) Reactiver SMPU */
75    lis   r3, SMPU_CESR@h
76    ori   r3, r3, SMPU_CESR@l
77    li    r4, CESR_ENABLE
78    stw   r4, 0(r3)
79    msync
80    isync
81    blr
```

7.3 Variants by subfamily

The MPC57xx family covers different profiles (body/chassis, powertrain, gateway) and integration gaps that directly impact memory policy:

- number of core MPU inputs (static TGBs) and effective granularity per e200 core;
- support of extended MAS registers (e.g. MAS7/MAS6) according to heart and addressable space;
- number of regions and granularity actually available;
- number of ports/masters supervised; *semantic precise region words (W0..W3) and lock bits;*
- *behavior of exceptions (Data/Instruction storage) and associated diagnostic records.*

In practice, migration between variants MPC574x and MPC577x should not be limited to copying offsets. The core MPU coverage (code/data areas by execution context) and system SMPU coverage (bus masters, DMA, peripherals), as well as involuntary recovery anti-patterns [19, 20], should be revalidated separately.

8
PMP on RISC-V: operation and variants

8.1 generic RISC-V (PMP) logic

On RISC-V, the mechanism equivalent to the MPU is the PMP (Physical Memory Protection). The PMP is programmed via dedicated CSRs (`pmpcfgX`, `pmpaddrX`) and applies R/W/X rights over physical intervals, with TOR, NA4 or NAPOT matching modes according to the chosen format [24, 25].

Operational principles are close to ARM/PowerPC:

1. a PMP input defines a physical area;
2. a straight fixed configuration field and address mode;
3. priority is given to the entry of smaller index that matches;
4. a violation triggers an exception of type instruction/load/store access fault.

However, the PMP has two structural features for safety:

- the bit `L` (lock) can freeze an entry until the reset;
- depending on implementation, Machine (M-mode) accesses may or may not be subject to certain PMP rules, which requires clarification of the privilege model in the defence strategy.

8.1.1 Example of implementation assembler RISC-V

The following example shows a minimum PMP configuration with two TOR inputs: Flash region execute/read-only and SRAM region read/write no-exec. The exact encoding depends on the architecture (RV32/RV64) and the number of entries implemented on the target heart [24, 25].

```
/* Exemple simplifie RISC-V PMP - syntaxe GNU as */

.equ PMP_R,      0x01
```

CSR	Name	Main role
0x300	mstatus	Privilege context; interaction with trap delegation.
0x305	mtvec	Exception vector (including PMP faults).
0x340	mscratch	Minimal context backup in machine-mode handler.
0x342	mcause	Fault cause (load/store/instruction access fault).
0x343	mtval	Faulting address in case of PMP violation.
0x3A0	pmpcfg0	8 PMP configuration fields (R/W/X/A/L).
0x3A1	pmpcfg1	Additional PMP entries (if implemented).
0x3B0	pmpaddr0	Address associated with PMP entry 0.
0x3B1	pmpaddr1	Address associated with PMP entry 1.
0x3BF	pmpaddr15	Last typical entry (count varies by core).

Available PMP CSRs depend on the implementation (E31, U54, etc.).

Figure 8.1: Mapping typical CSRs related to PMP in RISC-V

```
.equ PMP_W,       0x02
.equ PMP_X,       0x04
.equ PMP_A_TOR,   0x08
.equ PMP_L,       0x80

/* pmpcfg0 octet0 -> entree 0 ; octet1 -> entree 1 */

mpu_init_riscv_pmp:
  /* 1) Nettoyage config PMP */
  li      t0, 0
  csrw    pmpcfg0, t0

  /* 2) Entree 0 TOR: [0x00000000, 0x080FFFFF] RX */
  li      t0, (0x080FFFFF >> 2)
  csrw    pmpaddr0, t0

  li      t1, (PMP_R | PMP_X | PMP_A_TOR)

  /* 3) Entree 1 TOR: (0x080FFFFF, 0x2001FFFF] RW, no-exec */
  li      t0, (0x2001FFFF >> 2)
  csrw    pmpaddr1, t0

  li      t2, (PMP_R | PMP_W | PMP_A_TOR)

  /* 4) Composer pmpcfg0: cfg1 dans l'octet 1, cfg0 dans l'octet 0 */
  slli    t2, t2, 8
  or      t3, t2, t1
  csrw    pmpcfg0, t3

  fence
  ret
```

8.2 Variances according to implementations

PMP portability is less immediate than it seems, as the specification allows important implementation choices:

- number of entries (4, 8, 16, 64);
- presence/absence of S/U mode and delegations of exceptions;
- precise behavior of M-mode access to PMP;
- support of Smepmp (strengthening extensions on recent targets).

So we need to explicitly document the *security contract* of the platform: which modes run the application, which areas are locked, and which handlers treat fault access with which fail-safe policy.

9

Comparative security between architectures

9.1 Corrections of concepts

The table 9.1 aligns memory protection concepts useful for security between ARM Cortex-M, PowerPC MPC57xx and RISC-V PMP.

Concept	**ARM Cortex-M**	**PowerPC MPC57xx**	**RISC-V PMP**
Protection unit	MPU (PMSAv7/v8)	SMPU / System	PMP MPU (CSRs)
Definition region	Base+size (v7) or base+limit (v8)	Start/end + fee per region	TOR/NA4/NAPOT via pmpaddr/pmpcfg
Priority Recovery	Highest Region Number	Rules by Region/Port Based on Implementation	Smallest PMP Index Matching
Data execution	Bit XN / Execute-Never	Attribute controlled by region	Absence of X on PMP input
Privilege modes	Priv/Unpriv (+ Secure/NS in v8-M TZ)	Supervisor/User (MSR[PR])	M/S/U according to heart
Diagnosis fault	MemManage + CFSR/MMFAR	Exceptions storage + ESR/DEAR	mcause/mtval access fault
Governance DMA	Often external to MPU CPU	Plus naturally integrated multi-master	Depends on the local bus/IOMMU subsystem

Table 9.1: Comparison of memory protection concepts according to architecture

9.2 Impact on portability

The portability of a memory security policy is not a register-to-record translation exercise; It's a conservation of security properties.

In practice, the following invariants should be kept to a minimum during migration:

1. **Code in RX only**: no data area should remain executable.
2. **Unmodifiable critical data** from application domains.
3. **MMIO sensitive reserved** to the privileged context.
4. **Deterministic fault management** with logging and fail-safe reaction.
5. **DMA controls consistent** with CPU policy.

The most common deviations observed during migration are:

- naïve transposition of ARMv7 region sizes to TOR/NAPOT into PMP;
- forgot the PMP index priority (rule masking effects);
- erroneous assumption that the CPU protection automatically covers all bus masters;
- does not take into account lock/default behaviors according to architecture.

A robust migration therefore requires double checking: static proof of coverage of critical areas and dynamic negative tests on each target architecture [5, 19, 24, 11].

10
MPU as countermeasure to memory attacks

This chapter consolidates the elements of the threat model (chapter 4) and security objectives OS-1 at OS-5 (section 4.2) to explain, attack by attack, the actual contribution of the MPU. The approach is operational: for each scenario, the operating chain, the expected impact without protection, and then the bulwark brought by a properly implemented MPU policy are described.

10.1 Guidelines for MPU Countermeasure

The MPU is effective in cybersecurity only if the memory policy simultaneously adheres to four engineering principles:

1. **Deny-by-default**: Any area not explicitly described remains inaccessible.
2. **Less privilege**: each component only obtains the strictly necessary rights (R/W/X and level of privilege).
3. **Code/data separation**: code is executable and non-modifiable; the data is non-executable.
4. **Closing by Trust Domain**: kernel, drivers, application, diagnosis and update are isolated as much as material granularity allows.

These principles directly structure the objectives OS-1 (confinement), OS-2 (integrity of the execution stream), OS-3 (protection of critical structures), OS-4 (restriction MMIO) and OS-5 (detection/response) defined above.

10.2 Attack 1: memory corruption and control-flow diversion

10.2.1 Threat scenario

The typical attack string is: unreliable input (network, bus, local interface), corruption of a buffer, alteration of a control structure (return address, function pointer, table entry), then

redirect execution stream [29]. On MCU without insulation, this chain quickly leads to a global compromise, as shown by the analysis of bare-metal binaries by Clements *et al.* [11, 10].

From a concrete point of view, the remote compromise of ECU illustrated by Checkoway *et al.* and Miller/Valasek shows that an external input vector can be converted into internal control of critical functions when memory and privilege boundaries are insufficient [9, 18].

10.2.2 MPU mitigation

The MPU operates at two levels in the operating chain:

- **Corruption (OS-1, OS-3).** RAM is segmented into regions by domain (task pile, application data, kernel/RTOS structures, interrupt vectors). An out-of-domain writing triggers a fault instead of silent corruption.
- **Impact reduction (OS-5).** The fault manager treats the event in a deterministic manner (journalization + transition to safe status), which limits the spread of the incident.

In practice, the *strict breakdown + blocking errors* couple turns a potentially systemic compromise into localized and observable fault.

10.3 Attack 2: Unauthorized code execution

10.3.1 Threat scenario

After memory corruption, the attacker seeks either to execute a payload injected in RAM (shellcode) or to chain gadgets already present in the legitimate binary (ROP/JOP) [29]. Without strict code/data separation, RAM becomes a trivial execution surface.

In documented industrial attacks (e.g. Stuxnet, Triton/TRISIS), the operational objective is to have unauthorized behaviour performed on lower layers of the system, with persistence and discretion [14, 12]. Although these campaigns are not limited to the MPU, they illustrate the impact of uncontrolled execution on a control system.

10.3.2 MPU mitigation

The central contribution of the MPU is the closure of executable surfaces:

- **XN systematic on data (OS-2).** Cells, piles, I/O buffers and shared areas are marked non-executable.
- **Code in RX only (OS-2).** Flash/code regions remain executable but unmodifiable in runtime.
- **Reducing usable material for ROP/JOP.** Minimizing executable regions and avoiding permissive mappings increase the technical prerequisites of the attacker and reduce exploitable chains.

The MPU does not eliminate all forms of ROP/JOP, but removes the direct RAM code injection path and contributes significantly to the increase in attack cost.

10.4 Attack 3: privilege escalation

10.4.1 Threat scenario

A code already executed in non-privileged context attempts to access core resources: RTOS scheduling structures, vector table, sensitive MMIO registers (DMA, Flash controls, debug), or memory reconfiguration mechanisms. Without strict privilege separation, local compromise becomes a global takeover.

Work on inboard partitioning shows that the user/supervisor boundary is the determining factor between local incident and kernel escalation [11, 10].

10.4.2 MPU mitigation

The MPU provides the climbing rampart through three complementary mechanisms:

- **Prior/non-privileged separation (OS-3, OS-4).** Critical kernel and MMIO structures are accessible only from the privileged context.
- **Controlled Access Passover.** Application tasks go through explicit system calls (SVC/syscall); the parameters are validated on the kernel side before any sensitive operation.
- **Protection of politics itself.** MPU and reconfiguration registers remain under privileged control, limiting the ability to disable the barrier from a compromised context.

This strategy materializes the border of trust: a compromised application code can fail locally without obtaining access to the system's sovereign assets.

10.5 Operational synthesis: threat, objective, MPU rampart

The table 10.1 synthesizes the correspondence between threats, objectives and expected effects of the MPU countermeasure.

Scenario attack	OS target	Restart MPU main
Memory corruption (stack/heap/pointers)	OS-1, OS-3, OS-5	RAM fragmentation by domain, restricted writing critical areas, hardware fault and fail-safe processing.
Unauthorized execution (RAM shellcode, ROP/JOP prerequisites)	OS-2, OS-5	XN on all data, RX code only, minimization of executable regions and deterministic fault management.
Elevation of privilege from compromised task	OS-3, OS-4, OS-5	Strict privilege separation, sensitive MMIOs reserved for the kernel, access via controlled syscall, MPU configuration protection.

Table 10.1: Contribution of the MPU as a bulwark against memory attack scenarios

10.6 Conditions of validity of countermeasure

In order for this bulwark to be effective in a real product, five deployment conditions remain non-negotiable:

1. early and verified activation of the MPU on startup;
2. restrictive default policy on unmapped areas;
3. lack of ambiguous or permissive regional recovery;
4. fault manager tested in negative campaign;
5. consistency with non-MPU protections (secure boot, debug locking, DMA governance).

In summary, the MPU is not a universal protection, but a primary structural barrier against memory attack chains: it reduces exploitability, limits the impact of local compromises, and provides an indispensable detection/response mechanism in a deep defence architecture.

11
MPU against DMA attacks and compromised devices

11.1 Specific risks DMA

The DMA introduces a fundamental model change: memory transfers are no longer only initiated by the CPU core, but also by autonomous bus masters. A processor MPU policy can therefore be perfectly correct while remaining bypassable by a poorly configured DMA.

The main risk scenarios are:

- **reading sensitive areas** (keys, kernel state, inter-domain buffers) via a too wide DMA window;
- **unauthorised writing** in control structures (vectors, descriptors, TCB RTOS);
- **pivot attack** from an exposed device (network, radio, storage) to system RAM;
- **applicative logic bypass** by modifying buffers after software validation.

This risk is particularly critical on multi-master platforms (automobile/industrial) where the bus interconnects CPU, DMA, accelerators and communication controllers [19, 21].

11.2 Defensive measures

A robust strategy combines MPU CPU configuration and bus governance:

1. **Minimum DMA windows.** Each DMA channel is limited to the strict memory sub-assembly required.
2. **Buffer segmentation.** Separate DMA buffers, application buffers and critical data into distinct regions.
3. **Multi-master system protection.** Use XRDC/SMPU/IOMMU according to architecture to impose rights per master.
4. **MMIO whitelist.** Restrict the reconfiguration of DMA controllers to the preferred code.

5. **Time Controls.** Disable off-use DMA channels and invalidate descriptors after transfer.

The MPU remains a useful bulwark, but its role vis-à-vis the DMA is indirect: it protects the CPU path and reduces the possibilities of climbing after compromise, while the inter-master containment must be treated at the level of the bus subsystem.

12
Frequent and anti-pattern configuration errors

12.1 Design defects

The most common design defects observed are:

- **Regions too wide.** Only one RAM RWX region covers several areas of trust and cancels partitioning.
- **Uncontrolled recoveries.** A more permissive priority area masks an area that is supposed to be restrictive.
- **permissive background policy.** The *background map* implicitly allows unplanned access.
- **Incoherent attributes.** Data marked executable, code marked written, MMIO exposed in non privileged.
- **Absence of regional allocation model.** Regions are chosen by opportunity, without explicit active matrix/domain/right.

To avoid these anti-patterns, the configuration must be derived from traceable requirements (OS-1 at OS-5), then verified by cross-inspection (code, linker script, security documentation).

12.2 Operational defects

Even with a good initial design, protection often drifts in the integration phase:

- **Incomplete faulthandlers.** Absence of journalization, insufficient diagnosis, non-deterministic restart.
- **Uncontrolled dynamic reconfiguration.** Change of regions without validation or memory barriers.
- **Bypass in maintenance.** Temporary activation of broad rights retained in production.

- **Non-regression absent.** An application evolution modifies the memory card without updating negative tests.
- **Multicore inconsistency.** Policy applied to one heart but not to others.

The key operating criterion is simple: any policy violation must be detected, correlated with a cause, and treated according to a validated fail-safe reaction.

13
Integration into a secure development cycle

13.1 Requirements and traceability

MPU security must be governed by an explicit traceability chain:

1. **Security requirement**: formulation of the objective (e.g. OS-2, no execution in RAM).
2. **Architecture decision**: definition of partitioning and privileges.
3. **Implementation**: log configuration, linker script, wrappers syscalls.
4. **Prove**: positive/negative test, code review, proof of integration.
5. **Criteria of acceptance**: compliance decision or reasoned derogation.

A typical traceability artifact combines each memory region with: protected asset, authorized domain, R/W/X rights, privilege level, justification, associated test and validation result.

13.2 Industrialisation

The industrialization of the MPU policy is based on four practices:

- **Systematic reviews.** Security review dedicated to each memory card evolution.
- **Automated validation.** Integration of MPU IC/CD test campaigns (emulation, bench, HIL depending on availability).
- **Readable configuration diff.** Generate reports of differences in regions and permissions between versions.
- **Governance release.** Release block in case of failure of negative tests or drift of rights.

This discipline reduces silent regressions, which are the primary cause of loss of efficiency in operating memory partitioning mechanisms.

14
Validation and robustness tests

14.1 Safety tests

Validation must combine functional tests and attack oriented abuse tests:

- **Positive Tests.** Verify that each domain has access to only authorized areas.
- **Tests negative read/write/execution.** Force prohibited accesses to trigger expected errors.
- **Context Transition Tests.** Check MPU reconfiguration for each task/mode change.
- **MMIO/DMA Tests.** Try unauthorized access to DMA devices and channels.
- **Sturdy Campaigns.** Fault injection software and controlled disturbances to validate fail-safe behavior.

Each negative test must check three simultaneous results: correct hardware fault, usable logging, and a response consistent with the safety/cybersecurity policy.

14.1.1 Example of full compliance tests

The table 14.1 offers a standard test suite ensuring full coverage of the MPU implementation's compliance with the objectives OS-1 at OS-5. Each test must be performed at least in the qualifying environment (bank/HIL), and replayed in non-regression at each memory mapping evolution.

In addition, it is recommended that each test include a unique identifier, a precondition for execution, an oracle of verdicts, and archived evidence (raw log, timed trace, dump of fault records) for audit.

14.2 Acceptance criteria

The recommended minimum acceptance criteria are:

1. **Cover critical assets.** 100% of critical classified assets are associated with an explicit MPU rule.
2. **Prohibition coverage.** 100% of security matrix bans have an automated negative test.
3. **Reproducibility.** Stable results on several buildings and several physical targets.
4. **Reaction time.** Fault behaviour limited and compatible with system constraints.
5. **Absence of known bypass.** No documented path allows non-political access without alert.

Residual deviations must be addressed by formal derogation with impact assessment, action plan and closing date.

Test Entitled	Description	OS Associate
Interdomain Confiscation in Writing	From an unprivileged application task, try to write in the RAM of another domain (or in a kernel area). Expected result: MPU fault, no target memory change, logging the wrong address.	OS-1
Inter-domain confining in read	From an application domain, try to read a sensitive area of another domain (key, RTOS status, critical configuration). Expected result: denied access and traceability of the event.	OS-1
Executing forbidden from RAM (XN)	Injecting an instruction stub into stack/heap/buffer and then forcing a jump to this area. Expected result: execution exception, no valid RAM fetish.	OS-2
Zone integrity code (Flash in RX)	Try an application script in a code region marked RX. Expected result: write rejected, content unchanged (hash/CRC identical).	OS-2
Protection of the vector table	In non-privileged context, try to modify a vector table entry. Expected result: write refused and vector unchanged after verification.	OS-3
Protection of critical RTOS structures	Simulate alteration of TCB/scheduling lists from unauthorized task. Expected result: material refusal and absence of corruption of core structures.	OS-3
Restriction of sensitive MMIO	Since a non-privileged task, try access to sensitive MMIO registers (DMA, flash ctrl, debug). Expected result: no access; Only licensed core services succeed.	OS-4
Validation of syscall gateways	Run legitimate and misformed system calls (out-of-area pointers, invalid sizes). Expected result: strict acceptance of valid cases, rejection of invalid cases without extending rights.	OS-3, OS-4
Deterministic reaction to MPU faults	Cause controlled R/W/X violations and verify the complete sequence (context capture, error code, fail-safe action, restart if required).	OS-5
Non-regression of MPU configuration	Compare two firmware versions: any difference in regions/permissions must be detected, justified and covered by updated tests.	OS-1 at OS-5

Table 14.1: Example of continued tests covering the compliance of the MPU implementation with the objectives OS-1 at OS-5

15
MPU limits and complementary defences

15.1 Structural limitations

Even properly configured, the MPU retains intrinsic limits:

- **Regional Granularity.** Intra-regional attacks remain possible.
- **No logical validation.** An authorized code may remain functionally hazardous.
- **Contouring by external masters.** DMA and bus masters require dedicated protections.
- **Except physical perimeter.** Glitching, side-channels and invasive attacks are not covered.
- **Pre-activation window.** The code executed before MPU activation remains unforced.

The MPU must therefore be considered a necessary but not sufficient control of a defence in depth.

15.2 Additional measures

Priority supplements are:

- **Secure boot and authenticated update string.**
- **Debug interface locking** (JTAG/SWD/UART maintenance).
- **Protection of secrets** (secure storage, key management, runtime bypass).
- **Supervision runtime** (watchdog, integrity checks, fault telemetry).
- **Software duration** (static analyses, reviews, coding rules and targeted buzzing).

The real effectiveness depends on the coherent orchestration of these controls, not on a single mechanism.

16
Migration and portability of MPU policies

16.1 Translation Methodology

A robust migration does not translate registers, it retains security properties. The recommended method is:

1. **Training invariants** (RX code, XN data, privileged MMIO, etc.).
2. **Project assets** on the memory card of the arrival target.
3. **Map the mechanisms** (PMSAv7/v8, SMPU, PMP TOR/NAPOT) without implicit hypothesis.
4. **Check priorities/resolutions** for architecture-specific recoveries.
5. **Replay the negative test suite** and compare expected safety results.

The proof of migration must demonstrate safety equivalence, not just functional validity.

16.2 Frequent traps

The most common portability failures are:

- direct transposition of ARMv7 sizes/alignments to PMP TOR/NAPOT;
- forgets different priority rules according to architecture;
- confusion between CPU protection and multi-master system protection;
- does not take into account the privilege modes actually used in production;
- no locking (bit LPMP or equivalent mechanism) when required.

A mature migration strategy includes a property match matrix (objective, source mechanism, target mechanism, proof test).

17
Technical annexes

17.1 Reference resources

The appendices serve as an operational kit for the architecture, development, validation and audit teams.

17.1.1 Recommended Matrixes and Checklists

Working documents to be maintained in controlled configuration:

- active matrix ×domain ×rights R/W/X;
- table of regions (base, size/limit, attributes, justification, owner);
- checklist of MPU review (design, implementation, testing, operation);
- negative test frame per objective OS-1 at OS-5;
- safety non-regression plan and release blocking criteria.

17.2 Panorama of solutions supporting MPU

The table 17.1 presents a panorama of representative solutions (bare metal, RTOS and partition kernels) that can implement an MPU policy aligned with the objectives OS-1 at OS-5.

Type	**Solution**	**Editor**	**Open source**	**ARM Cortex-M**	**PowerPC MPC57xx**	**RISC-V PMP**	**Security targets covered**
Bare metal	CMSIS-Core + policy MPU application	Arm Ltd. (CMSIS) + product team	Yes (CMSIS)	Yes	No	No	OS-2, OS-3, OS-4, OS-5; OS-1part (depending on application partitioning)
Bare metal	S32 SDK / drivers MPU-SMPU	NXP	Partial (mixed)	Yes (S32K3)	Yes (MPC57xx)	No	OS-1 a OS-5 (with SMPU configuration + fault management)
RTOS	FreeRTOS-MPU	Amazon / FreeRTOS community	Yes (MIT)	Yes	No	Yes	OS-1 to OS-5 on the CPU side; OS-4 complete if MMIO is filtered via privileged services
RTOS	Zephyr (Userspace + Memory Domains)	Linux Foundation	Yes (Apache-2.0)	Yes	Partial	Yes	OS-1 to OS-5 (depending on userspace, memory domains and handler activation)
RTOS	AUTOSAR Classic OS (e.g. RTA-OS, MICROSAR OS)	ETAS / Vector (according to stack)	No	Yes	Yes	No	OS-1, OS-3, OS-4, OS-5; OS-2 according to integration XN/W^X and boot
Partitioning core	Tock OS (capsules + isolated processes)	Tock Project	Yes (Apache-2.0/MIT)	Yes	No	Yes	OS-1 to OS-5 on MPU/PMP-compatible targets
Partitioning core	seL4 (microkernel)	seL4 Foundation / UNSW	Yes (BSD)	Yes	No	Yes	OS-1 to OS-5 (with capability policy and configured partitioning)
Partitioning core	PikeOS	SYSGO	No	Yes	Yes	Partial	OS-1 a OS-5 (according to certified profile, BSP and active partitioning)

Table 17.1: Panorama of solutions supporting MPU on the three architectures and coverage of security objectives

Note: the indicated coverage corresponds to a conformite potential, conditioned by the actual configuration (partitioning, rights, handlers, MMIO/DMA restrictions) and by the BSP maturite level on the target.

17.2.1 Glossary and acronyms

API. *Application Programming Interface*: Software interface exposed to an external component.

ARM. Family of processor architectures widely used in embedded MCU and SoC.

CAN. *Controller Area Network*: series communication bus widely used in automotive and industrial.

CC. *Common Criteria*: framework for the security assessment of computer products.

CI/CD. *Continuous Integration / Continuous Delivery*: automation of integration, testing and delivery.

CFSR. *Configurable Fault Status Register*: Cortex-M register for the diagnosis of configurable errors.

CMSIS. *Cortex Microcontroller Software Standard Interface*: standard ARM layer for Cortex-M cores.

CPU. *Central Processing Unit*: central processing unit running the machine code.

CSR. *Control and Status Register*: control/state registry (notably in RISC-V).

DMA. *Direct Memory Access*: memory transfer without direct CPU intervention.

ECU. *Electronic Control Unit*: electronic control unit, particularly in automobiles.

EAL. *Evaluation Assurance Level*: level of insurance in Common Criteria.

ESR. *Exception Syndrome Register*: Emergency status register (PowerPC e200).

Flash. Non-volatile memory usually containing executable firmware.

GNU. *GNU's Not Unix*: a free project whose tools (e.g. GNU as assembler) are used in embedded systems.

HardFault. Exception Cortex-M critical fault not recovered by dedicated handlers.

HIL. *Hardware-in-the-Loop*: test method integrating real hardware and simulation.

IOMMU. *Input/Output Memory Management Unit*: protection/translation unit for I/O devices.

ISA. *Instruction Set Architecture*: instruction set architecture.

IVOR. *Interrupt Vector Offset Register* : exceptional vector registry on PowerPC e200.

JOP. *Jump-Oriented Programming*: a jumping gadgets chaining technique.

JTAG. *Joint Test Action Group*: standard hardware test and debug interface.

LIN. *Local Interconnect Network*: low cost car series bus.

MAIR. *Memory Attribute Indirection Register*: ARMv8-M indexing register of memory attributes.

MCU. *Microcontroller Unit*: microcontroller.

MMFAR. *MemManage Fault Address Register*: Cortex-M registry containing the MemManage faulty address.

MMIO. *Memory-Mapped I/O*: registers of devices addressed as memory.

MMU. *Memory Management Unit*: memory management unit with translation of addresses.

MPU. *Memory Protection Unit*: memory protection unit by region.

MSR. *Machine State Register*: status register machine on PowerPC.

NA4. *Naturally Alignment 4-byte region*: PMP mode for 4 bytes aligned region.

NAPOT. *Naturally Aligned Power-Of-Two*: PMP mode for aligned region of power size of two.

NVIC. *Nested Vectored Interrupt Controller*: Cortex-M interruption controller.

OS. Security objective used in this guide (OS-1 at OS-5).

OTA. *Over-The-Air*: remote software update.

PMP. *Physical Memory Protection*: RISC-V memory protection mechanism.

PMSA. *Protected Memory System Architecture*: Cortex-M MPU model.

PMSAv7/PMSAv8. Model PMSA versions for ARMv7-M and ARMv8-M generations.

RAM. *Random Access Memory*: volatile working memory.

RBAR/RASR/RLAR. ARM records describing MPU regions (base, attributes, limit).

RDP. *Readout Protection*: a mechanism to protect against on-board memory reading.

RO. *Read Only*: read-only permission.

ROM. *Read-Only Memory*: non-volatile memory read-only, often used for boot code.

ROP. *Return-Oriented Programming*: return chain operation technique.

RTOS. *Real-Time Operating System*: real-time operating system.

RW. *Read Write*: read and write permission.

RX. *Read Execute*: read and run permission.

SMPU. *System Memory Protection Unit*: multi-master system memory protection.

Smepmp. Extension RISC-V PMP reinforcement for preferred modes.

SoC. *System on Chip*: circuit incorporating multiple subsystems on a chip.

SVC. *Supervisor Call*: system call exception on ARM Cortex-M.

SWD. *Serial Wire Debug*: two-wire ARM debug interface.

TCB. *Task Control Block*: control structure of a RTOS task.

TCP. *Transmission Control Protocol*: IP-oriented transport protocol.

TLB. *Translation Lookaside Buffer*: caches translations/protections.

TOR. *Top Of Range*: PMP address mode based on top terminal.

TrustZone-M. ARMv8-M partitioning extension Secure/Non-Secure.

UART. *Universal Asynchronous Receiver-Transmitter*: asynchronous serial interface.

USB. *Universal Serial Bus*: standard serial communication and power interface.

WBWA. *Write-Back, Write-Allocate*: memory cache policy.

XN. *Execute Never*: attribute prohibiting the execution of instructions.

XRDC. *eXtended Resource Domain Controller*: resource insulation controller/multi-master memory.

Bibliography

[1] Anvil Secure. Glitching STM32 read out protection with voltage fault injection. Technical report, Anvil Secure, 2024.

[2] Arm Ltd. *ARMv7-M Architecture Reference Manual*, 2010. Revision E.b.

[3] Arm Ltd. Arm trustzone for cortex-m. Technical report, Arm Ltd., 2017. Technical overview white paper.

[4] Arm Ltd. *Cortex-M7 Processor Technical Reference Manual*, 2020. Latest revision.

[5] Arm Ltd. *Armv8-M Architecture Reference Manual*, 2021. Latest revision.

[6] Arm Ltd. *Cortex-M33 Processor Technical Reference Manual*, 2021. Latest revision.

[7] Arm Ltd. *CMSIS-Core (Cortex-M) Documentation*, 2024.

[8] Arm Ltd. *CMSIS-Core MPU Functions and Register Abstraction*, 2024. See CMSIS groups group__mpu__functions and group__mpu8__functions.

[9] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX Security Symposium*. USENIX Association, 2011.

[10] Abraham A. Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. ACES: Automatic compartments for embedded systems. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, 2018.

[11] Abraham A. Clements, Naif Saleh Almakhdhub, Khaled S. Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2017.

[12] Dragos Inc. TRISIS: Targeting safety instrumented systems. Technical report, Dragos Inc., 2017. Threat intelligence report, https://dragos.com/blog/trisis/.

[13] ENISA. ENISA threat landscape 2023. Technical report, European Union Agency for Cybersecurity, 2023.

[14] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32.Stuxnet dossier. Technical report, Symantec Corporation, 2011. Version 1.4, https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.

[15] International Electrotechnical Commission. IEC 62443 series: Security for industrial automation and control systems. International standard series, 2018. Multi-part standard series.

[16] Keen Security Lab. Experimental security research of Tesla autopilot. Technical report, Tencent Keen Security Lab, 2016. Publicly released research report, https://keenlab.tencent.com/en/.

[17] Microchip Technology Inc. *SAM E70/S70/V70/V71 Family Data Sheet*, 2023. Latest revision.

[18] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. In *DEF CON 23*, 2015. White paper, http://illmatics.com/Remote%20Car%20Hacking.pdf.

[19] NXP Semiconductors. *MPC5748G Reference Manual*, 2020. Latest revision.

[20] NXP Semiconductors. *MPC5744P Data Sheet*, 2021. Latest revision.

[21] NXP Semiconductors. *S32K3xx Reference Manual*, 2024. Latest revision.

[22] Timo Obermaier and Stefan Tatschner. Shedding too much light on a microcontroller's firmware protection. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. USENIX Association, 2017.

[23] Sandro Pinto and Nuno Santos. Demystifying Arm TrustZone: A comprehensive survey. *ACM Computing Surveys*, 51(6), 2019.

[24] RISC-V International. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*, 2024. Ratified specification.

[25] SiFive Inc. *SiFive E31 Core Complex Manual*, 2019. Includes PMP programming model.

[26] STMicroelectronics. *RM0090 Reference Manual: STM32F405/407, STM32F415/417, STM32F42x and STM32F43x Advanced Arm-based 32-bit MCUs*, 2023. Latest revision.

[27] STMicroelectronics. *RM0433 Reference Manual: STM32H743/753 and STM32H750 Value Line Advanced Arm-based 32-bit MCUs*, 2024. Latest revision.

[28] STMicroelectronics. *RM0456 Reference Manual: STM32U575/585 Arm-based 32-bit MCUs*, 2024. Latest revision.

[29] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2013.