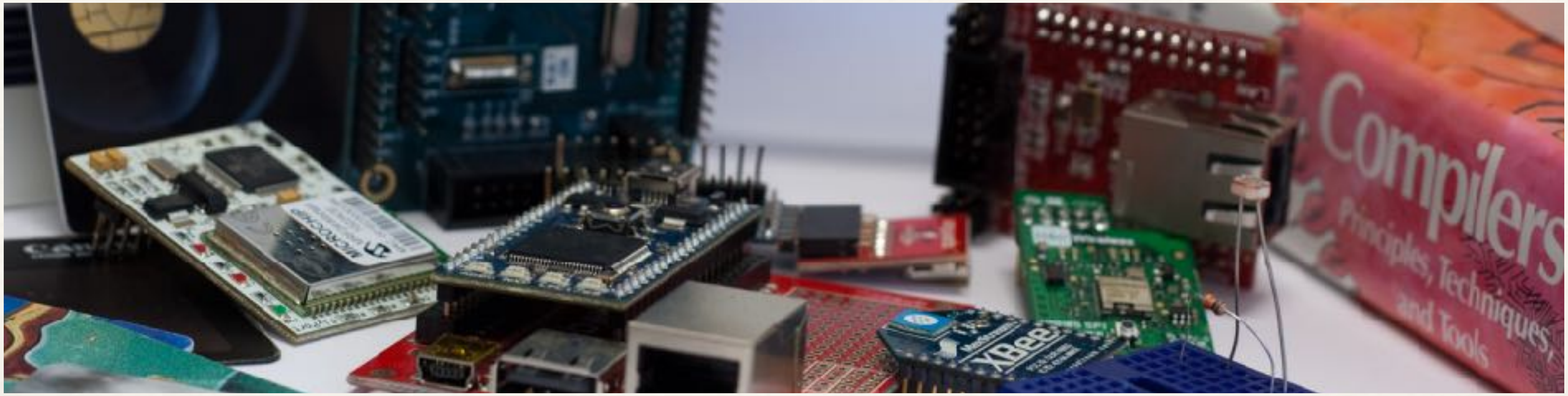


*RIOT over PIP: embedding secure online deployment*

# Safe post-issuance software provisioning

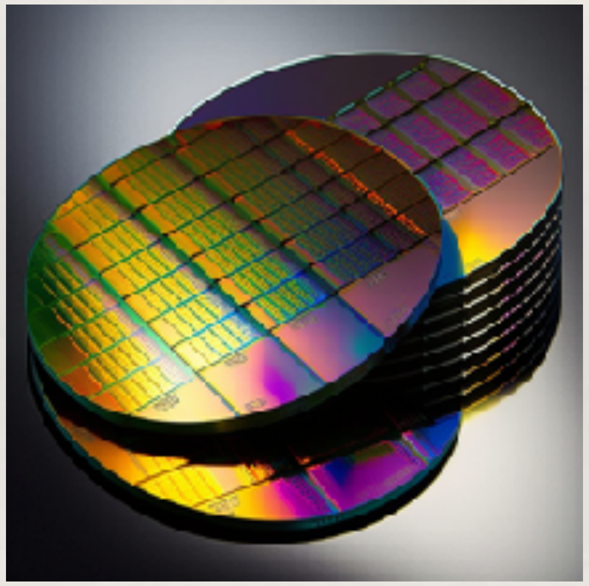
Damien Amara
Gilles Grimaud

## Summary

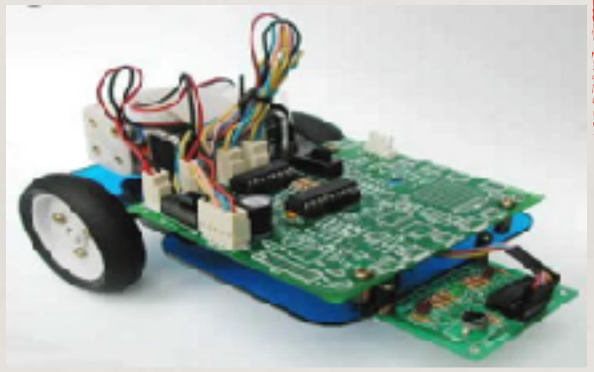
# Deployment "post issuance"

*Offline deployment*

1. Silicon Production
by the Chipmaker



2. Device production
by the Manufacturer



3. Device distribution
by the Retailer

4. Device usage
by the Consumer

*Online deployment*

Timeline

## Online software provisioning for RIOT-OS

Why manufacturers look for Post-Issuance?

- Allows Execution of Single-Shot Software

Why retailers look for Post-Issuance?

- Enables Software Personalization
- Opens Support for Third-Party Software

Why consumers look for Post-Issuance?

- Enables Software Provisioning

*"In an industrial mass-production setting, post-issuance deployment primarily allows for a basic system image to be written to ROM, while still leaving the door open for future extensions, amendments, and partnerships once millions of units are already deployed all around the world."*

## How to support Post-Issuance in RIOT-OS?

Introducing the Execute-In-Place FileSystem (XIPFS)

The **xipfs** module delivers:

A streamlined file system for seamless software provisioning across the entire IoT lifecycle.

## How to support Post-Issuance in RIOT-OS?

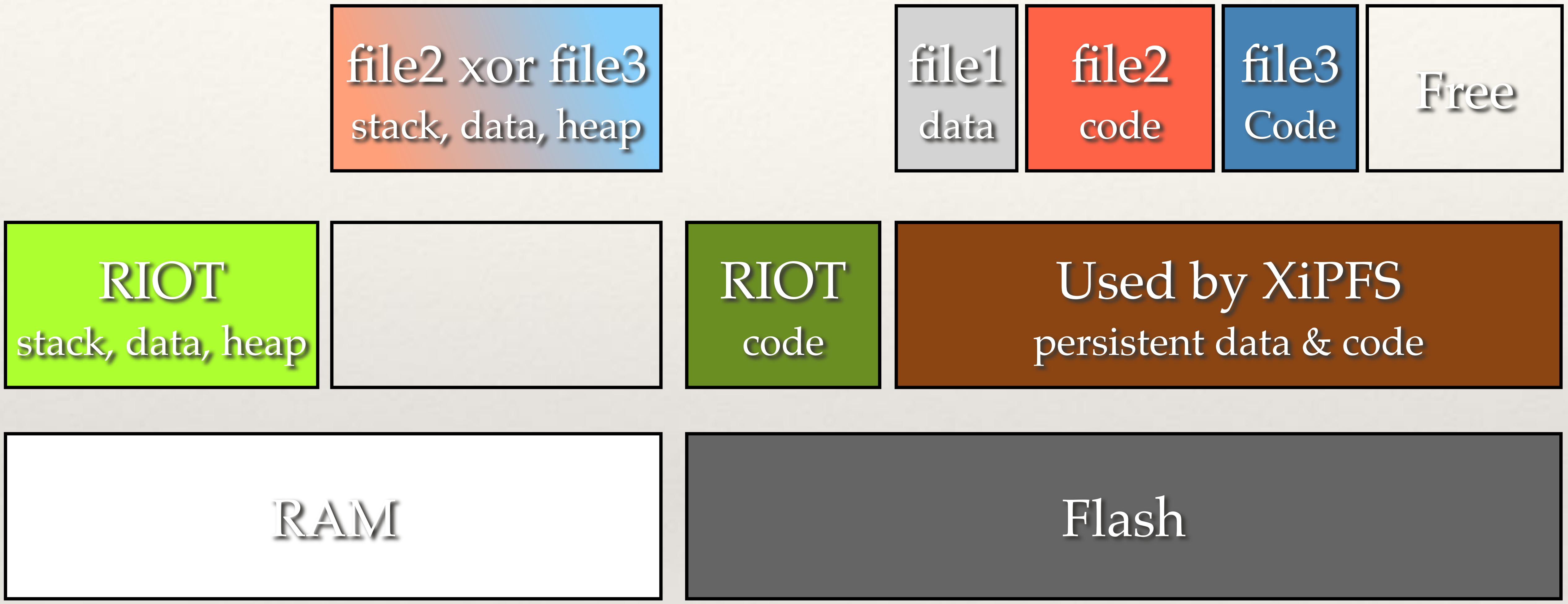
Introducing the Execute-In-Pl System (XIPFS)

The **xipfs** module de

A streamlined fil n for seamless software provisioning ac he entire IoT lifecycle.

Demonstration !

## The design of an eXecute-In-Place File System



The devil is in the details :

- position independent code compilation
- global offset table online relocation
- syscall function table
- filesystem fragmentation
- … but also …

## Security issue

## Security issue

Demonstration !

# PIP-MPU design principles

1. Manage Memory Partitions using Memory Protection Unit
   Dedicated address spaces where code can execute and perform read/write operations. (Details in the following slide)

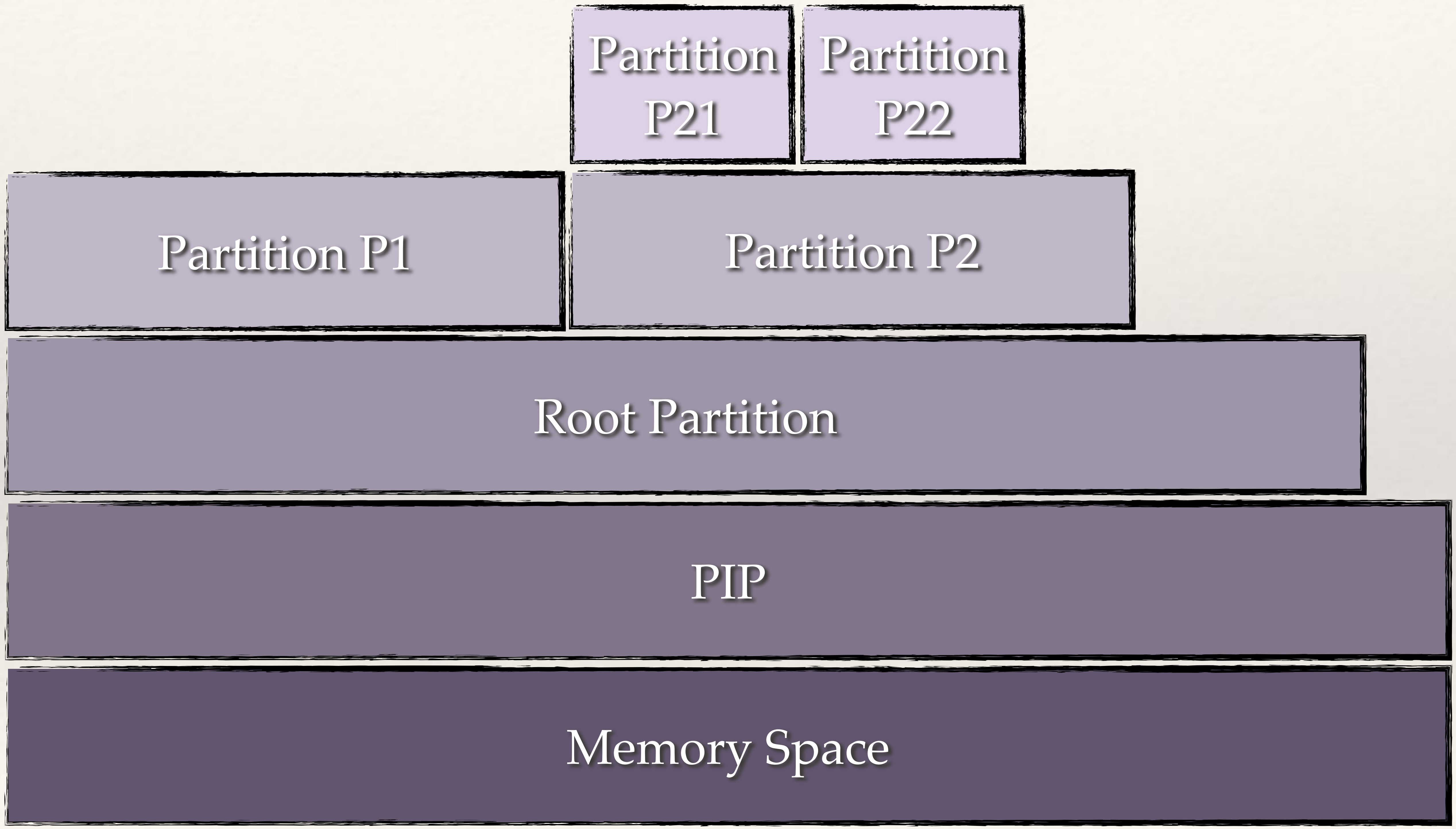
2. Provide Critical System Primitives
   PIP-MPU is designed to perform critical system tasks:
   - **Memory Partitioning**: Create and delete memory partitions;
   - **Execution Flow Transfer:** on demand switch between partitions;
   - **Hardware Interrupt Handling:** Redirected to the root partition;
   - **Hardware Register Access:** under PIP-MPU supervision
     (addressing DMA security issue).

3. Prove Reliability of Features
   Implemented in Gallina and formally proved using the Coq proof assistant.

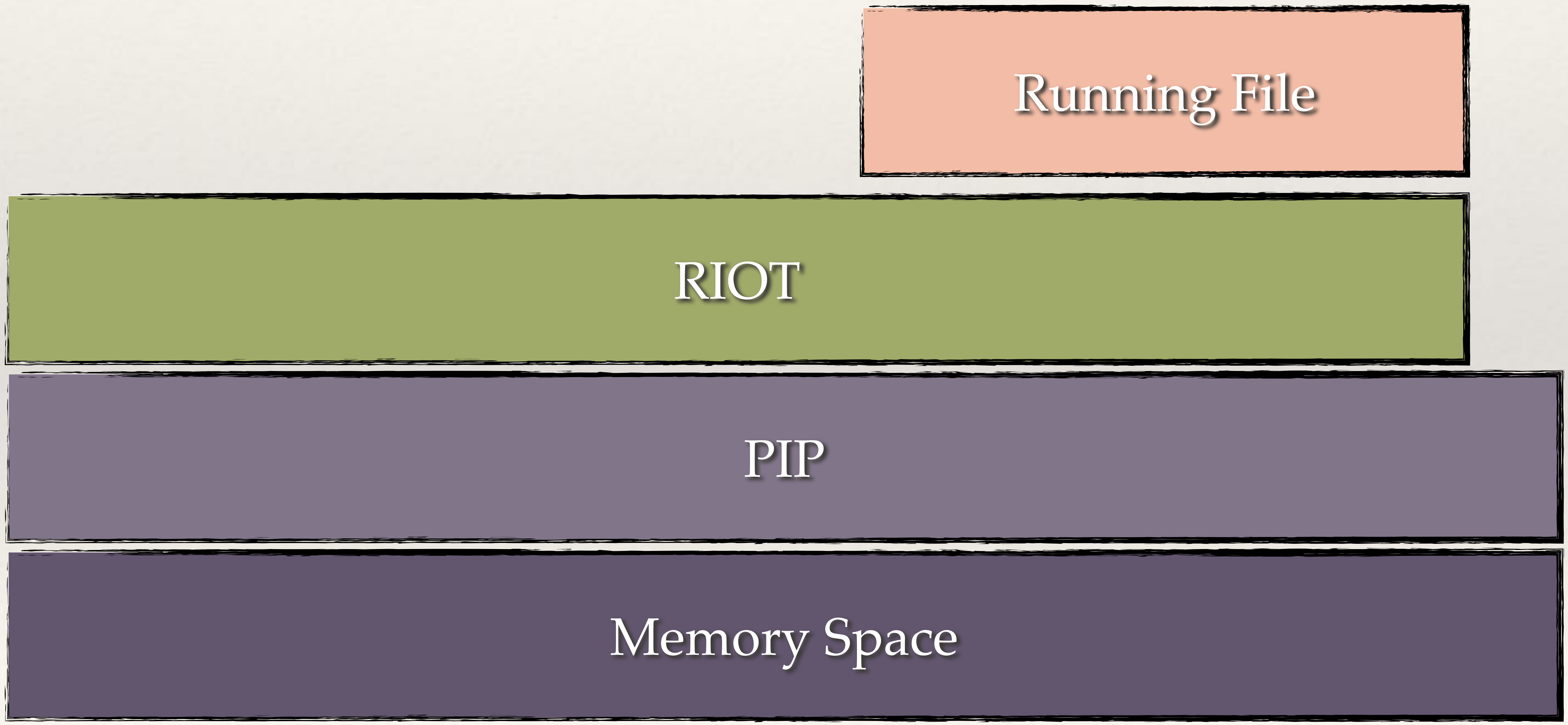
# Typical PIP-MPU hardware

- IoT microcontroller for mass-market production
- Our prototypes run on DWM1001
  - RAM: 64Kb
  - Flash: 512Kb
  - ARM Cortex M3

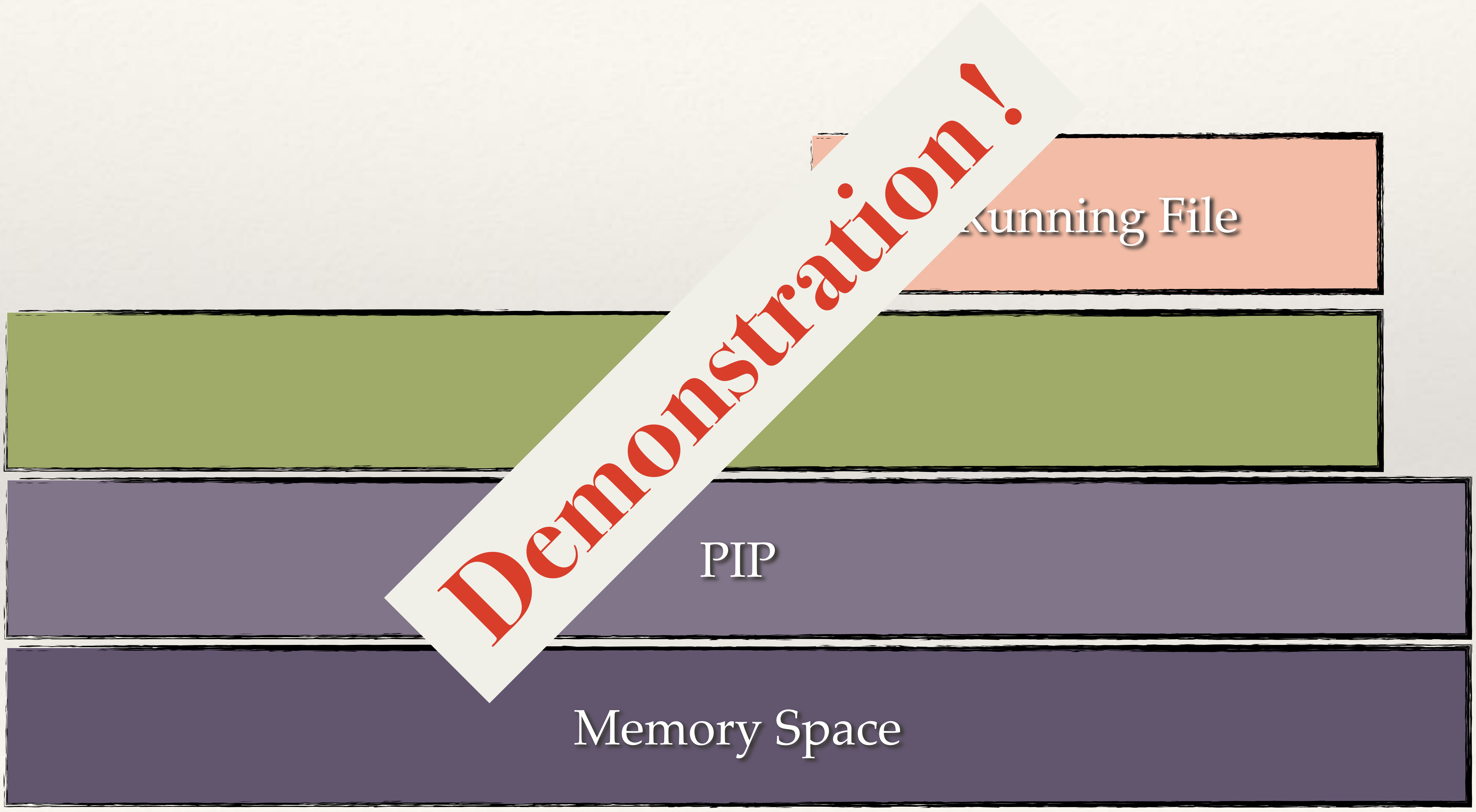
## Memory Partitioning



## Porting RIOT over PIP-MPU



# Porting RIOT over PIP-MPU



## Conclusion

Safe and Secure Post-Issuance Software Deployment Is Possible:

**xipfs** requires 12 bytes of RAM and 1058 bytes of Flash

**PIP-MPU** requires 3028 bytes of RAM and 28988 of Flash

Memory partitioning through MPU slows down code execution by 5 to 10% (further investigation ongoing).

Sources available @

https://pip.univ-lille.fr
https://github.com/2xs
https://github.com/2xs/pipcore-mpu

and as soon as possible
a pull request of the **xipfs** module in riot-os :)

Questions ?

## Why Does PIP-MPU Impact ARM CPU Efficiency?

Where Are ARM Cycles Lost?

The loss of CPU cycles is proportional to the time spent executing within memory partitions, specifically due to interruption handling and system call handling via PIP-MPU. In detail:

1. Interruption Handling with PIP-MPU:
   PIP-MPU serves as the interrupt handler, much like a standard OS. Minimal cycles are lost here ;
2. State Saving of Interrupted Partition:
   We optimize by saving only the general-purpose registers and defer saving floating-point unit registers (*Lazy FPU context switching*) unless another partition requires them. No cycles are lost when the FPU is not used ;
3. MPU Configuration Restoration for Root Partition:
   This is the main source of cycle overhead, costing hundreds of cycles. PIP-MPU must emulate an "idealized" MPU configuration (a kind of virtual MPU), which incurs this cost;
4. Context Restoration to Virtual Interrupt Handler:
   Here, additional cycles are lost, consistent with any hypervisor—approximately one CPU state load (~32 cycles).

## Why Run RIOT-OS on PIP-MPU Instead of Embedding MPU Management in `xipfs`?

**[Pro] Robust MPU Management:**

PIP-MPU uses Gallina to implement MPU management, minimizing risks like stack smashing and buffer overflows.

**[Pro] Formally Verified:**

PIP-MPU comes with a Coq proof. When you request a memory partition, the isolation of the partition is proven to be guaranteed.

**[Pro] Reduced Error Probability:**

Implementing MPU management manually can be complex and error-prone. it's proven that PIP-MPU avoids this risk.

**[Cons] Security Limitations:**

While PIP-MPU ensures robust partitioning, it cannot extend this guarantee to code executed in place via **xipfs**. RIOT-OS serves as an intermediary layer between PIP-MPU and the child partition, and although it's not formally verified, it's part of the trusted computing base for the child partition. Because RIOT-OS, the intermediary layer, is susceptible to security vulnerabilities like buffer overflows, it compromises the trusted computing base of the child partition, thereby potentially undermining the child partition's own security…

# Why Run RIOT-OS Over PIP-MPU Instead of Making PIP-MPU a Module in RIOT-OS?

**Proof Integrity:**
Proving PIP-MPU's isolation guarantees requires it to operate independently of any external modules. This is the last of the three proofs done in PIP-MPU :

1. Horizontal isolation ; 2. Vertical sharing ; *3. PIP-MPU isolation*

**Immutable Code:**
PIP-MPU's code must not be modifiable. This integrity is achieved because everything within PIP-MPU is formally verified. When embedding PIP-MPU in RIOT-OS, this part of the proof is lost.

**Data Structure Modification Constraints:**
Any changes to PIP-MPU data structures must adhere to isolation constraints. Embedding PIP-MPU in RIOT-OS would require proving the isolation of PIP-MPU for each memory access in RIOT-OS, however RIOT-OS is not written in Gallina.

**Hardware Configuration Security:**
External hardware configurations, like DMA settings, should not compromise MPU settings by PIP-MPU. By running PIP-MPU as a lower-level layer, hardware drivers have to request changes through PIP-MPU, which only allows secure hardware configurations, such as DMA settings.

# Writing a kernel in Galina-c?

**Why Galina-C?**

**Directly maps to C:** What you see in galina-c is what you get in C. No runtime required.

**Security Guarantees:** MPU Configuration: Execution in galina-c configures the MPU such that only partition-specific memory is accessible.

**How It Works**

**Proof on galina-c:** Formal verification is conducted on the galina-c code.

**DX tool:** A galina-based tool called "dx" converts galina-c to C. The tool itself can be formally verified.

**Proof Continuity**

**CompCert:** Allow to transform the proved C code into machine code while maintaining the proof integrity.

**Security Guarantees**

**MPU Configuration:** Execution in galina-c configures the MPU such that only partition-specific memory is accessible.

## The Devil's in the Details

**Position Independent Code Compilation:**

Utilization of the -fPIC, -msingle-pic-base, -mpic-register=r10, flag in GCC.

Less significant for ARM architectures, which are inherently designed for Position Independent Code.

**Global Offset Table (GOT) and Dynamic Relocation:**

a dedicated CRT0 initializes RAM, based on the address provided by `xipfs` to store data.

**System Call Function Table:**

To allow dynamically loaded code to invoke RIOT functions:
a function table is passed as a parameter from RIOT to the "CRT0" of the called program.

**Filesystem Fragmentation and Contiguity:**

`xipfs` allocates a minimum of 4K pages (flash memory page size).

Executables are stored contiguously in memory.

Deletion of a file triggers memory defragmentation to ensure contiguous loading.