life.augmented

Concepts:
multi-context

Table of Contents

1 Context and Scope 2
2 References 2
3 Subject of study 2
3.1 Definitions 2
3.2 Multiple contexts of execution 3
3.3 Multi-context Examples 5
4 Concepts to be implemented 6
4.1 Execution context 6
4.1.1 Execution context basics 6
4.1.2 Execution context and execution modules 8
4.2 Execution groups 9
4.3 Execution domain 9
4.3.1 Concept 9
4.3.2 Mutualization possibility 11
4.3.3 Visibility 11
4.4 Concepts' possible implementation 12

1 Context and Scope

The purpose of this document is to expose concepts related to the “multi-context” feature, typically required for TrustZone-enabled cores and multi-cores systems on chip (SoC).

2 References

Document	Link
[1] ThreadX modules	https://docs.microsoft.com/en-us/azure/rtos/threadx-modules/chapter1
[2] SMP	https://developer.arm.com/documentation/den0024/a/Multi-core-processors/Multi-processing-systems/Symmetric-multi-processing
[3] AMP	https://developer.arm.com/documentation/den0024/a/Multi-core-processors/Multi-processing-systems/Asymmetric-multi-processing

3 Subject of study

3.1 Definitions

Application project

An **Application Project** is targeting an application product embedding STM32 devices, such as a connected thermostat or an autonomous lawnmower. In this context, one important part of the product is the printed circuit board – or only “board” – which holds one STM32 device; this is the physical support of the product. The other important part of such a product is the embedded software which is executed one the STM32 device(s).

SW Project:

A SW Project (mainly composed of source code) is meant to be compiled into a binary file via a SW build chain, and this binary file will then be programmed into an STM32 device to be executed on it.

Context of execution

A **context of execution** (see 4.1) is a concept to identify the “cradle” allowing to run a binary program on a given hardware platform. A context of execution has a 1-to-1 association to an HW execution unit (see below).
If we support virtual machines or interpreters (python, javascript…) then the virtual machine is associated to one execution contexts. The programs interpreted by the machine are considered as data sections in memory.

Binary program

This **binary program** can be:

- A bare-metal binary
- A monolithic OS binary and all the services running on top of it: still one execution context.
- An OS that can install modules. Each module can be a binary on its own but still from an execution perspective as the OS is required for this service to run we consider it as a single binary in term of execution context.
- A virtual machine or interpreter

SoC & Core

A **SoC** (system on chip) can be a microcontroller (MCU) or a microprocessor (MP).
A SoC can have one or several processors.
A processor can have one or several cores (core = CPU+FPU+…).
For example, Cortex-A7 can have two cores.
When a processor has a single core, we will talk about a core in this study (Cortex-M has only one core for now).

So, in this study, we will talk about multi-core only (even when multi-SoC): we will distinguish between symmetric or asymmetric.
Example: STM32H755 is 1 SoC with 2 asymmetric processors/cores (CM4 and CM7).

TrustZone is an ARM technology offering a system-wide approach to security with hardware-enforced isolation built into the CPU (Cortex-M33/55 at the moment in the ST MCU portfolio).
Example: Cortex-M33 has one core defining a Trusted and Non-Trusted world.

HW execution unit

To be more generic, in this study, we will consider that an **HW execution unit** is either a core or a TrustZone world within a core. This is the hardware unit/entity that can run a binary program. It has a 1-to1 association with an execution context at a given point in time (but several execution contexts can use a same HW execution unit sequentially). A hardware execution unit corresponds to a core and security mode. A synonym is: hardware execution context (MP wording).
Note: a DMA is not a hardware execution unit as it cannot run a binary program. It is piloted by a hardware execution unit.

Execution domain

An execution domain is a container aggregating all the platform resources (internal and external memories, peripherals) that an execution context can use “exclusively”. For shared resources (available for several execution contexts) we introduce the concept of shared domain (see 4.3).

3.2 Multiple contexts of execution

We study the support of **multiple contexts of execution**.
An execution context can be instantiated by:

- a core during a quantum of time
- a TrustZone world within a core (during a quantum of time)

Multi-context covers:

- contexts running sequentially (one binary then another one takes the hand)
- contexts running concurrently (binaries running at the same time on separated cores)
- contexts running in pseudo-parallel mode (interleaved execution of binaries by a same core via virtualization, ex: TrustZone; or ping-pong between binaries)

This study is about implementing

- this concept in the composition and configuration tool,
- and the related concepts for system partitioning between execution contexts (memory partitioning, IP partitioning and dependencies, capability enablement…will be studied in additional documents or next versions of this document).

Simple Application

Nevertheless, please keep in mind that for most of the applications the end-user will use a default setup with only one execution context running a single SW project. In this case all these concepts (execution context, execution domain) shall be transparent for the end-user.

Partitioning

1) In a multi-context application project (multi-core and/or TrustZone) we need to do some **resources partitioning**: this is the action of assigning a resource to one execution context or another. Or we can also let the resource as a shared one (accessible by all execution contexts). To memorize the assignment of a resource to an execution context we introduce the concept of **execution domain**.

2) An **initializer context** is the execution context which oversees the initialization (but not the driving) of a shared resource when required.

3) The **memory mapping** is the ability to define the organization of the memory resources (RAM, ROM, including external memories), the properties of the memory areas and their partitioning between the different execution contexts. For TrustZone solutions (ARMv8-M architecture), the memory compartmentation will be enforced by the MPC (memory protection controller: this is GTZC for our L5 product for instance). The 4GB address space of the STM32 is considered (so peripherals' addresses too).

4) When it comes to peripherals partitioning, some important definitions are related to TrustZone (ARMv8-M architecture). When the TrustZone security is active, a peripheral can be either Securable or TrustZone aware type as defined below:
   - Securable: a peripheral is protected by an AHB/APB firewall gate that is controlled from TZSC controller to define security properties.
   - TrustZone-aware[1]: a peripheral connected directly to AHB or APB bus and implementing a specific TrustZone behavior such as a subset of registers being secure. TrustZone-aware AHB masters always drive HNONSEC signal according to their security mode (as CM33 core and DMA).

   So, when we discuss the peripherals partitioning:
   - Some partitioning will be implemented by the PPC (peripheral protection controller: this is GTZC in our L5 architecture for instance) and the granularity is the peripheral.
   - Some partitioning will be implemented by the peripherals (TZ-aware) and here we can have a finer grain granularity. For instance:
     - An RTC alarm can be secure and another non-secure
     - A DMA channel can be secure and another one non secure
     - A GPIO pin can be secure and another one non secure
     - A RCC clock source can be secure and another one non-secure[2]

[1] Please note that this is a ST concept, this does not exist in ARM's specs
[2] Nevertheless, today the working view is that as soon as one source is secure the entire clock tree management must be handled by a secure context

Segregation

Segregation: the action or state of setting someone or something apart from others.

The context segregation is the possibility to have clear frontiers between the resources and settings used by different execution contexts. Therefore, this opens the possibility to extract from an application project all the artefacts related to one or several execution contexts based on their security attribute or hardware execution unit.

Examples:

- Extract all the elements related to Cortex-M4 for an STM32H745-based application project
- Extract all the elements related to the secure projects for an STM32L562-based application project.

3.3 Multi-context Examples

If we consider a typical application leveraging the TrustZone capability, then we have an application project using 2 execution contexts:

- 1 execution context associated to the secure world and running the secure SW project
- 1 execution context associated to the non-secure world and running the non-secure SW project

If we consider a typical application leveraging a dual-core SoC, then we have an application project using 2 execution contexts:

- 1 execution context associated to the first core
- 1 execution context associated to the second core

If we consider a typical application leveraging a firmware update capability like the one provided by X-CUBE-SBSFU, then we have an application project with 2 execution contexts:

- 1 execution context associated to the bootloader
- 1 execution context associated to the user application

In this situation, please note that the 2 contexts do not run in parallel but sequentially. Therefore, most of the peripherals should be in a shared execution domain. Only the memory regions they use exclusively must be assigned to different execution domains.

4 Concepts to be implemented

Disclaimer: we think it is good to split in different concepts the elements we need to handle. This allows a clear split of roles and better thinking. Nevertheless, all these concepts will not necessarily be shown to the end-user, the tool may very well abstract them for the sake of simplicity (example: the difference between execution context and execution domain is not needed for the end-user).

4.1 Execution context

The execution context is the “cradle” or the “shell” to host a binary produced by 1-N software project(s): we can have a binary within a binary or a library in the binary for instance.

- A binary is by default a bare-metal binary.
- If we are using an OS: the binary becomes the OS and all its applications.

If we deal with a MP (microprocessor), this processor can have several cores.
Then three situations can happen:

- Handle the MP as a group of cores: an OS like Linux manages all the cores[3]. In this situation, the processor is associated to only one execution context by default.
- Handle the MP as a “set of cores” (super MCU), each core being managed independently[4] (bare-metal or single-core OS): in this situation each core is associated to an execution context by default.
- Handle the MP in mixed “mode”/heterogenous architecture:
    - Some cores are grouped as a cluster so 1 execution context for all
    - Some other cores are exposed so 1 execution per core
    - Example for STM32MP1:
        - One cortex-A cluster with at least one core
        - One Cortex-M cluster

4.1.1 Execution context basics

In a nutshell, the **execution context** associates a **SW project** to an **HW execution unit** (today: a core or a TZ world within a core).
A very same core can have several contexts of execution.
For instance:

- Cortex-M4 can have 1 context for a bootloader and 1 context for an application
- Cortex-M33 can have 1 context for the Secure world and 1 context for the Non-Secure world.

[3] Symmetric multiprocessing (SMP) is a computing architecture in which two or more cores are attached to a single memory and operating system (OS) instance. SMP combines multiple cores to complete a process with the help of a host OS, which manages cores allocation, execution and management. See [2].
[4] See [3] An Asymmetric Multi-processing (AMP) system enables you to statically assign individual roles to a core within a cluster so that, in effect, you have separate cores, each performing separate jobs within each cluster.

So, a pseudo data structure for an execution context could be:

```
{
   Id: 0,                 // identifier within the application project
   Label: “myThermostatApp”
   SoC: “STM32L552TZxQ”,  // SoC identifier within the hardware project
   Core: “Cortex-M33[0][5]”, // Core identifier within the hardware project
   CoreMode: “Secure”,     // TZ-World identifier within the hardware project
   SWProject: 0,          // SW project identifier within the application project
   XDomain: 0             // execution domain identifier within the application
project
}
```

We may also have application projects dedicated to libraries.
In this case, an execution context can be partial when it is used to create a library (independent from the hardware). In this case, only the core type is known. We may have an associated hardware where only a genuine core is defined (generic empty board with a generic cortex processor type).
So, a pseudo data structure for an execution context like this could be:

```
{
   Id: 0,                 // identifier within the application project
   Label: “myCryptoLibrary”
   SoC: “none”,  // no specific SoC targeted
   Core: “Cortex-M33”, // Generic Core identifier (generic hardware)
   CoreMode: “Secure”,     // TZ-World identifier within the hardware project
   SWProject: 0,          // SW project identifier within the application project
   XDomain: 0             // execution domain identifier within the application
project
}
```

[5] [0] indicates which core we use in the SoC, it is [0] by default for single-core SoC.

4.1.2 Execution context and execution modules

An OS can install modules (separate binary element that can be loaded dynamically or not and can be installed during the lifecycle of the product). Each module can be a binary on its own but still from an execution perspective as the OS is required for this service to run we consider it as a single binary in term of execution context.
This concept is supported by various OSes, for instance ThreadX does so, as you can see in [1].

It may be interesting to reflect it in the tool, we would:

- Keep only one execution context
- But this execution context could have one or several associated **execution modules**

So, the execution context is a kind of parent entity having one or several (or no) execution modules.
At the end of the day, an execution module is associated to a **SW project**.
These execution modules are siblings and cannot work alone, they need a module support infrastructure provided by the parent (the software project associated to the execution context).

An execution module can use the memory regions defined in the execution domain associated to its execution context. An execution module can also use all peripherals defined in the execution domain associated to the execution context. It is good practice to keep the OS running in privileged mode and the modules running in non-privileged mode with all interrupts handled by the OS[6] (with callbacks in the module if needed). Nevertheless, at least ThreadX (and maybe other OSes) allows a module to run in privilege mode so we must not forbid this possibility (so the module may redefine the VTOR and reshuffle the privileged/non-privileged settings).

[6] See the concept of device driver: https://docs.microsoft.com/en-us/azure/rtos/threadx/chapter5

4.2 Execution groups

We must be able to determine if some execution contexts can run concurrently or not. This is useful to determine if there can be concurrency to use some resources for instance. This piece of information is fundamental for the tool to implement checkers to let the user know when a very same resource is used at the same time by two different SW projects (so the user can indicate he explicitly wants it and manages it at user application level or solve the concurrency issue in the resources assignment).
To do so, we introduce the concept of **execution group**.
A group will aggregate all the execution contexts:

- Either running in parallel mode (separate cores)
- Or running in pseudo-parallel mode (separate TZ worlds or ping-pong within a core)

So, a pseudo data structure for an execution group could be:

```
{
   Id: 0,
   Label: “myThermostatAppGroup”
   XContexts: [0, 1, 2]
}
```

If two execution contexts do not belong to the same group then it is fully allowed to reuse resources, the checkers will not complain at all.

4.3 Execution domain

4.3.1 Concept

We must be able to identify the resources available for an execution context. All the resources a binary can use are aggregated in an element we call an **execution domain**.
An execution domain is referenced by a software project, and it associates a set of resources with an execution context. A particular case is the shared domain as it contains all the shared resources but no execution context. All software projects (so execution contexts) have access to the resources of the shared domain (without a need for an explicit reference to it).

Note: we may use an exclusivity rule saying that a resource not assigned to an execution domain remains accessible for anybody within the system. Nevertheless, for the sake of context segregation I think it is good to have the shared domain with an explicit list of resources we can share.

Note: a resource assigned to the shared execution domain can be used by several execution contexts but none of them can claim the exclusive ownership of this resource. A resource not assigned to any execution domain can be claimed for exclusive ownership by an execution context.

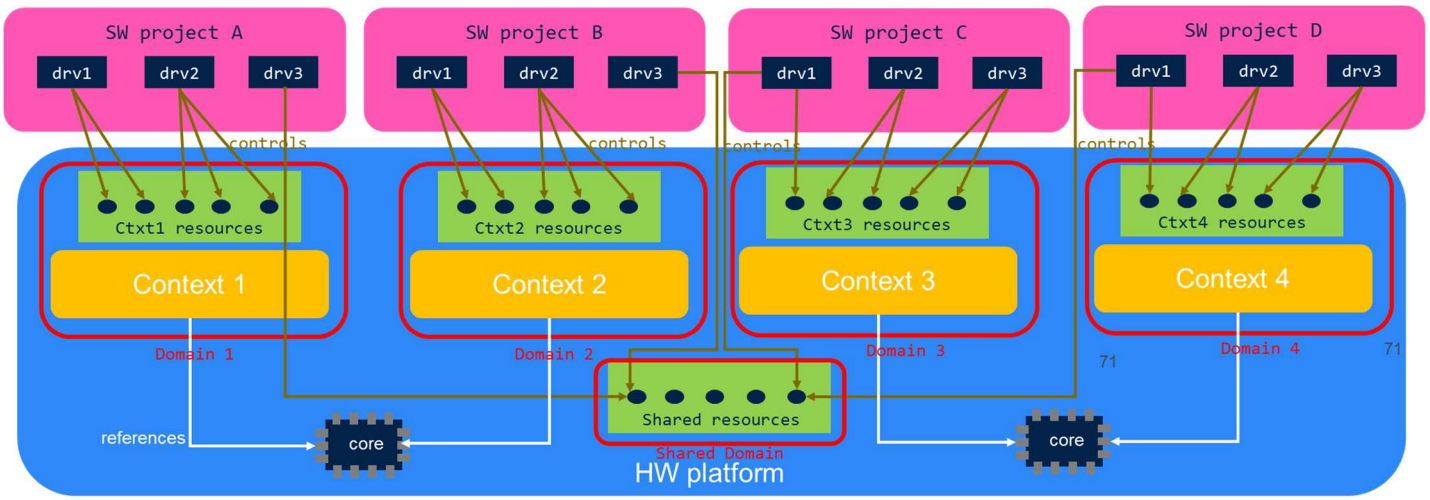

Figure 1: overview of contexts, domains, and software projects

So, a pseudo data structure for an execution domain could be:

```
{
   Id: 0,
   Label: “myThermostatAppDomain”
   XContext: 0,
   Resources: [res0, res1, res2],
   SecurityAttribute: S // to be confirmed
}
```

The execution domain is only a container aggregating resources, it holds references towards the resources themselves (and their settings). This is key to be able to easily switch a resource from one domain to another without losing its associated settings.

4.3.2 Mutualization possibility

Besides an optimization of the usage of the execution domains can be envisioned.
The idea would be that a very same execution domain could be re-used by different execution contexts which are not in the same group. Typically, a bootloader and an application may “share” the same execution domain (can be the shared execution domain by the way) rather than duplicating it if they use the same peripherals. But the two execution contexts may use different memory areas.
To handle this case, we may tune the execution context like this:

```
{
   Id: 0,                  // identifier within the application project
   Label: “myThermostatApp”
   SoC: “STM32L552TZxQ”,   // SoC identifier within the hardware project
   Core: “Cortex-M33[0]”, // Core identifier within the hardware project
   TZWorld: “Secure”,      // TZ-World identifier within the hardware project
   SWProject: 0,           // SW project identifier within the application project
   XDomain: 0              // execution domain identifier within the application
project
   MemoryResources : [res3] // refines the memory resources from the execution
domain (only the memory area differs between the 2 execution contexts, the
peripherals are the same)
}
```

And tune the execution domain like this:

```
{
   Id: 0,
   Label: “myThermostatAppDomain”
   XContexts: [0,1][7], // several execution contexts can be addressed
   Resources: [res0, res1, res2, res3]
}
```

4.3.3 Visibility

Please note that this concept of execution domain is totally transparent for the end-user. The UI does not show it to the user who deals with execution contexts only. The execution domain is an internal container used by the tool to aggregate the data and perform conflicts checking.

[7] These are execution context identifiers within the application project

4.4 Concepts' possible implementation

The concepts described previously belong to the **application project** entity. An application project associates a set of software projects to a given hardware platform. So, at application project level we can visualize the execution contexts and software projects and we can use the execution domains to partition the system.

Basically, at application project level we should offer a global view of the system:

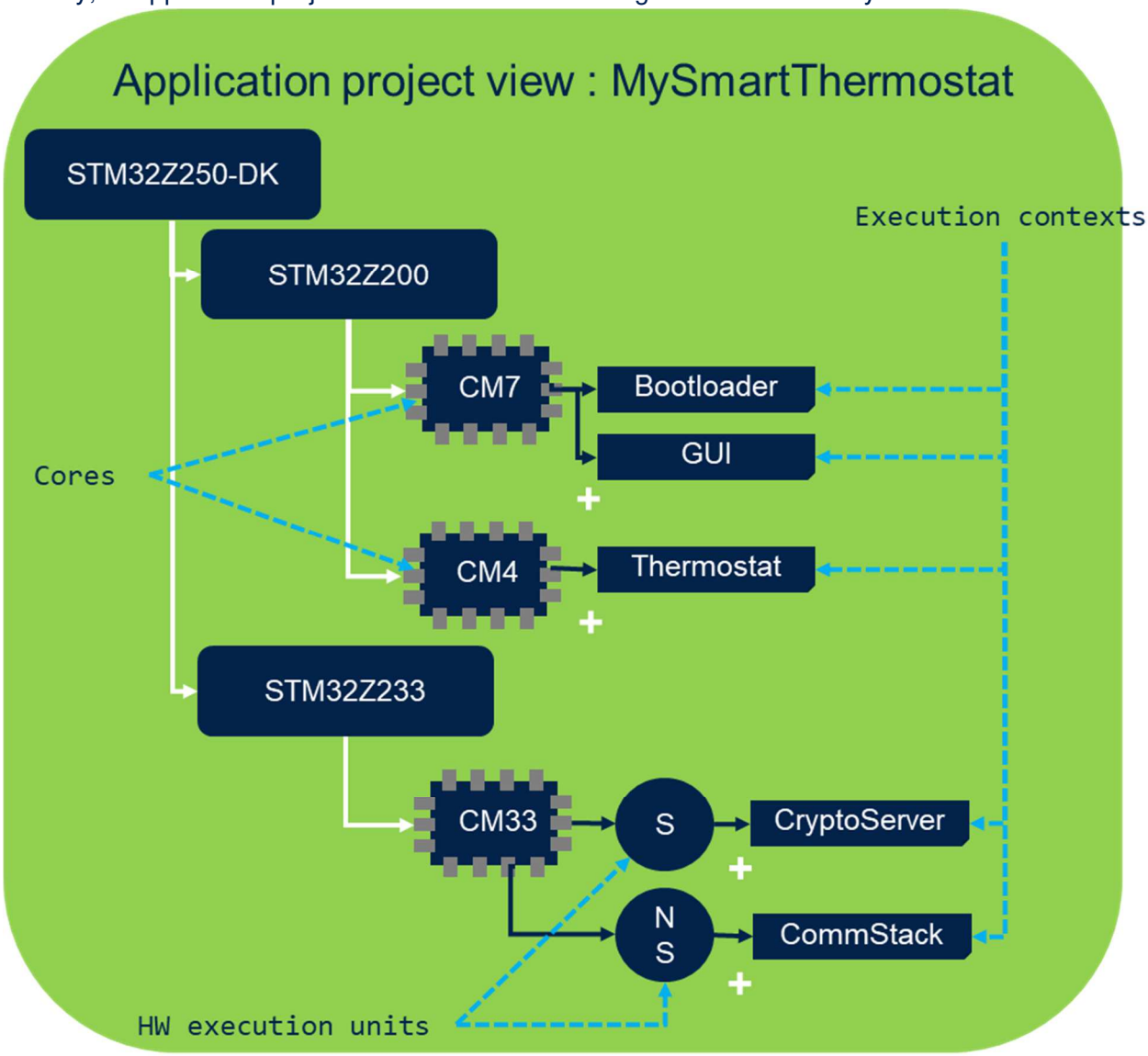

Figure 2 : application project view

In this (fantasy) example, we can see a HW platform with 1 Z250-DK board.
This contains 2 SoCs:

- STM32Z200 with 2 asymmetric processors (1 core each)
  - CM7 has 2 contexts
  - CM4 has 1 context
- STM32Z233 with 1 CM33 processor (1 core)
  - S world has 1 context
  - NS world has 1 context

Note: in the figure we show the cores and execution units when they do not match 1-to-1 but in the UI the execution unit can be a property of the execution context only (indicating nothing specific when directly linked to a core and indicating a security attribute when the TrustZone world matters).

The execution groups may appear as well (but not the execution domains as this is used by the tool only).
Each execution context is the entry point to access the associated software project view: this can be the software composer view or the IDE view of the project.

The application project view is also the entry point to do the resources partitioning: assign the resources to the various execution domains.

The resources consist of all the resources of the system, in particular:

- the peripherals
- the memory regions: these regions and their properties must be defined first thanks to a memory configurator

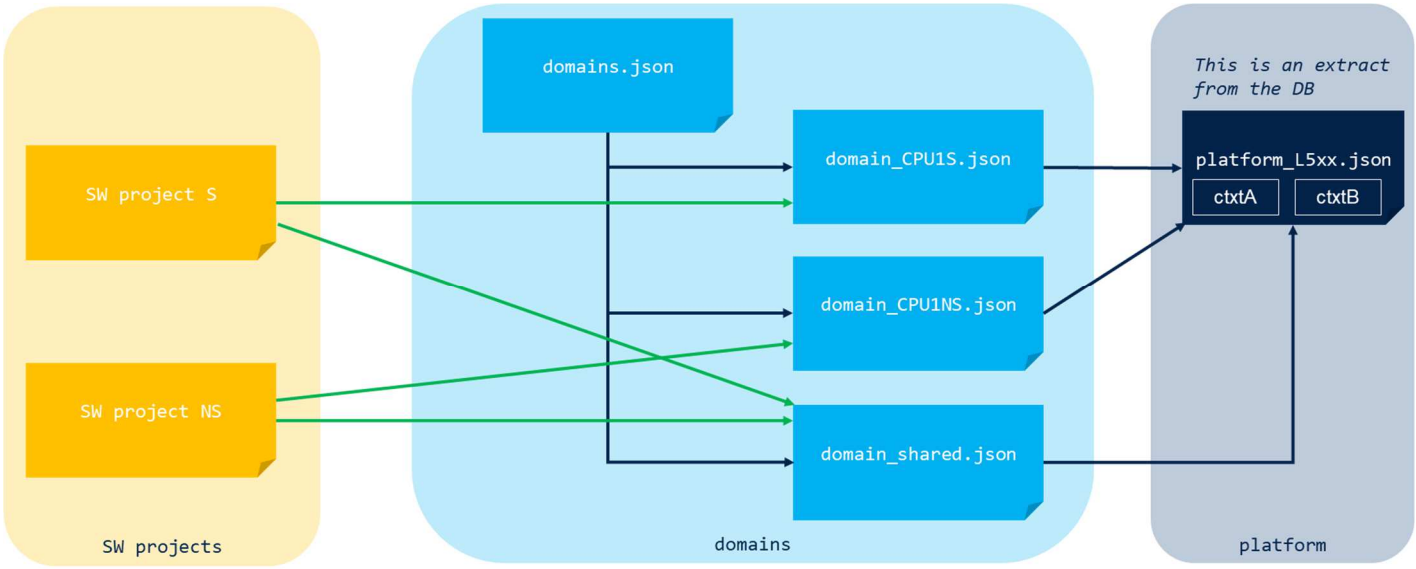

Figure 3 : resources assignments

The example above shows how an application project can be made of 2 software projects for an L5-based platform:

- one project running in the Secure context
- one project running in the Non-Secure context

The “platform_L5xx.json” file represents the description of the hardware platform. This file may not exist but the data it contains (execution contexts, SoC resources) must be available.
Typically, we should find here at least SoC and board information like for instance:

- the internal and external memories
- the default memory regions: address, size, type, security
- the peripherals: base address, size, type, security
- pins

The “domains.json” file represents the list of execution domains used in this application project. This file may not exist but the data it contains (list of domains) must be present, one section per execution domain, providing references to the resources.
Hence, when configuring the software components of a software project we know in which sections we can access the resources available/authorized for this project.
When two execution domains reference two execution contexts belonging to the same execution group, then the tool knows that these contexts must not have any resource in common.