TF-RMM Live Activation

Design Discussion

Andre Przywara, Manish Badarkhe, Soby Mathew

12/06/2025

Live Firmware Activation overview

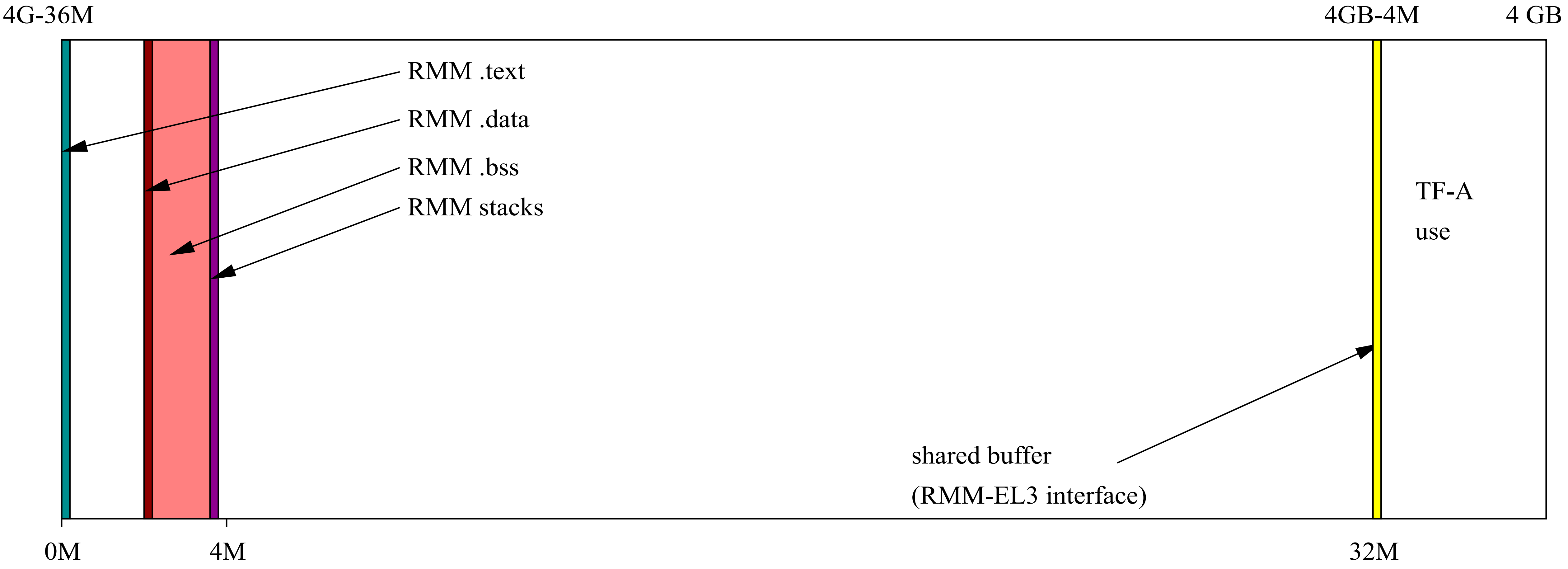
- An Arm spec (DEN0147) describing discovery and activation of updated firmware components
- Allows firmware updates without a reboot
- Describes an interface between an **LFA client** and an **LFA agent**
- LFA agent lives in EL3 runtime, so typically TF-A BL31
- Defines SMCCC compliant functions:
  - Detection: `LFA_VERSION, LFA_FEATURES`
  - Firmware discovery: `LFA_GET_INFO, LFA_GET_INVENTORY`
  - Firmware activation: `LFA_PRIME, LFA_ACTIVATE, LFA_CANCEL`
- Firmware activations might require CPU rendezvous
  - To prevent accidental calls into the to-be-updated firmware
  - To allow every CPU to (re-)initialise during the update process
- Spec is not concerned about the image updates itself, just the activation

TF-RMM live update

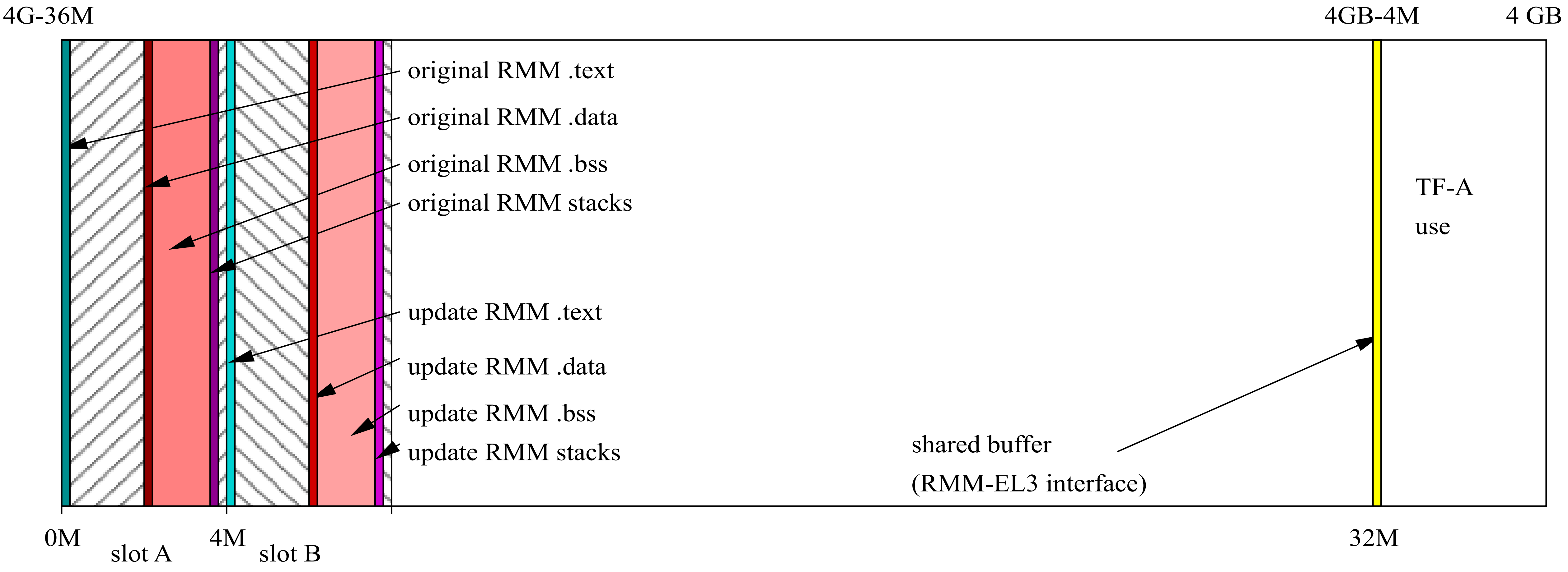
- Early example of a live-updatable component
- Looks like a nice target, since it does not carry much state
  - VCPU and guest state memory belongs to normal world
  - Pointers are passed in on every RMI call
  - Only a very few data structures to preserve: granule array, VMID array
- Mostly relays requests from normal world, no "life of its own"
- Constraints: (shared) data structures must stay compatible
- Expectation: Live updates only for small/minor revisions (fixes only?)

TF-RMM

Current TF-RMM carveout usage (on FVP)



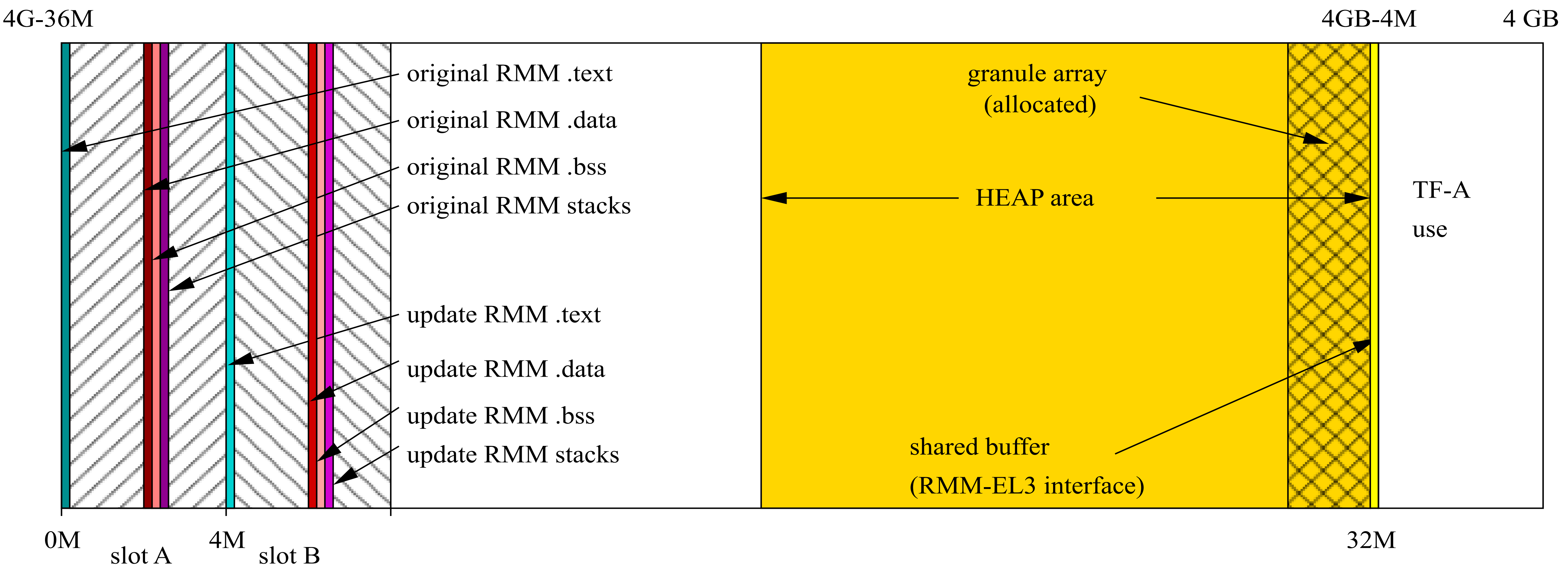
Naïve RMM update scenario



TF-RMM changes to allow live activation

- RMM gets (very limited) support for memory reservation from EL3
  - Only in the RMM setup phase
  - Before the MMU gets enabled, so mappings stay fixed
  - Only reservations, no freeing
- Allows for keeping global data persistent
  - So data can be shared quite naturally
- Allows for getting local memory for a particular core
  - To address NUMA requirements
- Allows for sizing memory regions
  - To scale with the number of cores
  - To scale with the amount of DRAM
- Per-CPU data structure as an anchor point
- Contains pointers (addresses) to global data structures for all persistent data
- Address of this per-CPU struct is passed between EL3 and RMM

Naïve RMM update scenario



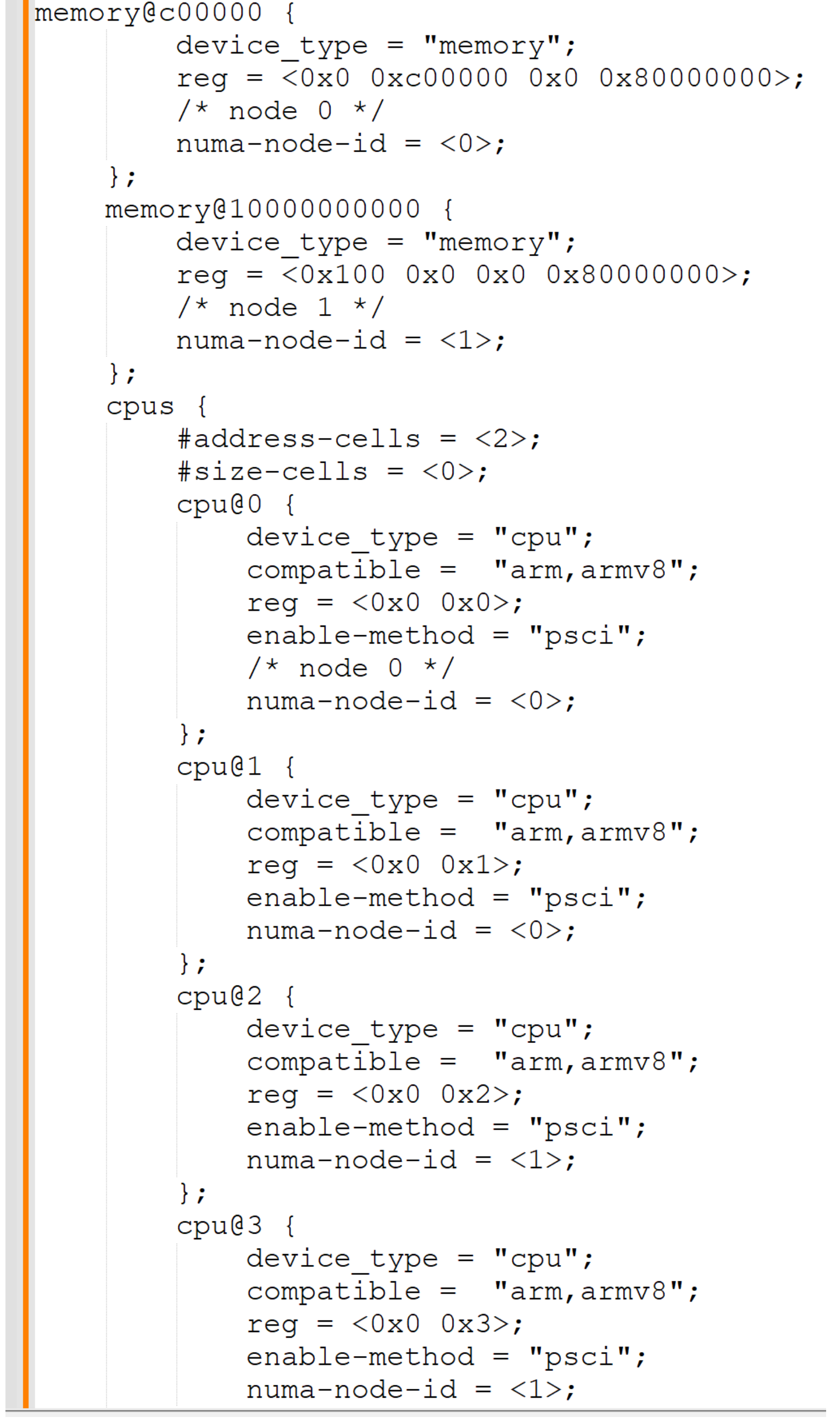
EL3-RMM communication changes : Rationale and details

RMM based allocation complexities

- Design choice for LFA in RMM –
    - Instead of migrating data from the old RMM instance to the new one during a live firmware update, the design proposes **reusing objects already allocated in the old RMM**.
    - This **requires a separate memory pool**, external to the RMM image itself, to store RMM state that must persist across the update.
- An alternative approach is describing this pool of memory via manifest and let RMM manage the memory which has some complexities.
- Bootstrapping complexity, particularly when Memory is needed before the C runtime is initialized.
    - A solution is to use a **temporary stack for the primary CPU** during early initialization.
- Allocating memory for secondary CPUs **before the MMU** is enabled
    - Mutual exclusion becomes an issue.
        - Using locks (e.g., like bakery-locks) in this early stage is **complicated.**
    - A cleaner approach: Have the **primary CPU allocate and partition** memory ahead of time for each secondary, and pass this pre-allocated memory via a structured hand-off.
        - Note that RMM would still need to cater for **hot spare cores/redundant cores** which are not used typically, but can be brought online when a regular core fails. Typically these are hidden as platform specific implementation in EL3.

Complexities Continued

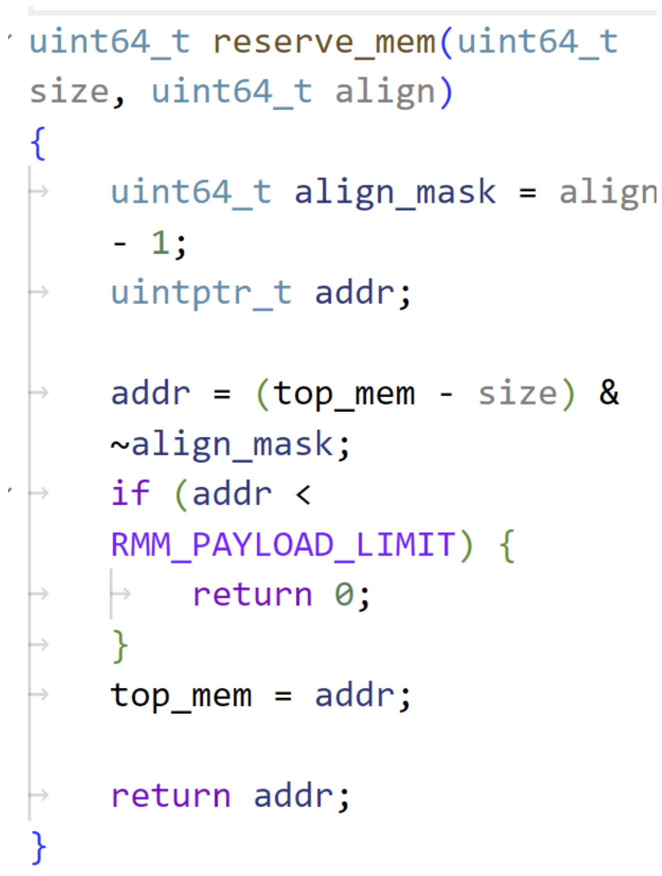
- NUMA or multi-chip needs special handling
  - EL3 would need to describe NUMA/Multi Chip topology to RMM with the NUMA node information via manifest.
    - NUMA node would describe Realm PAS carveout for use by RMM.
  - RMM should use this data to allocate memory from the appropriate node-local carveout.
    - The **primary CPU would need to preprocess and pass summarized topology/memory data** to the secondary CPUs in an assembly friendly structure/table. The secondaries would need to look up this table using MPIDR as a key.
- Need RMM to migrate allocation info for each pool to new RMM or embed the same within the data pool.
  - Chicken and egg situation if memory needs to be allocated prior to init of memory allocation data in new RMM.



```
memory@c00000 {
          device_type = "memory";
          reg = <0x0 0xc00000 0x0 0x80000000>;
          /* node 0 */
          numa-node-id = <0>;
      };
      memory@10000000000 {
          device_type = "memory";
          reg = <0x100 0x0 0x0 0x80000000>;
          /* node 1 */
          numa-node-id = <1>;
      };
      cpus {
          #address-cells = <2>;
          #size-cells = <0>;
          cpu@0 {
              device_type = "cpu";
              compatible =  "arm,armv8";
              reg = <0x0 0x0>;
              enable-method = "psci";
              /* node 0 */
              numa-node-id = <0>;
          };
          cpu@1 {
              device_type = "cpu";
              compatible =  "arm,armv8";
              reg = <0x0 0x1>;
              enable-method = "psci";
              numa-node-id = <0>;
          };
          cpu@2 {
              device_type = "cpu";
              compatible =  "arm,armv8";
              reg = <0x0 0x2>;
              enable-method = "psci";
              numa-node-id = <1>;
          };
          cpu@3 {
              device_type = "cpu";
              compatible =  "arm,armv8";
              reg = <0x0 0x3>;
              enable-method = "psci";
              numa-node-id = <1>;
```

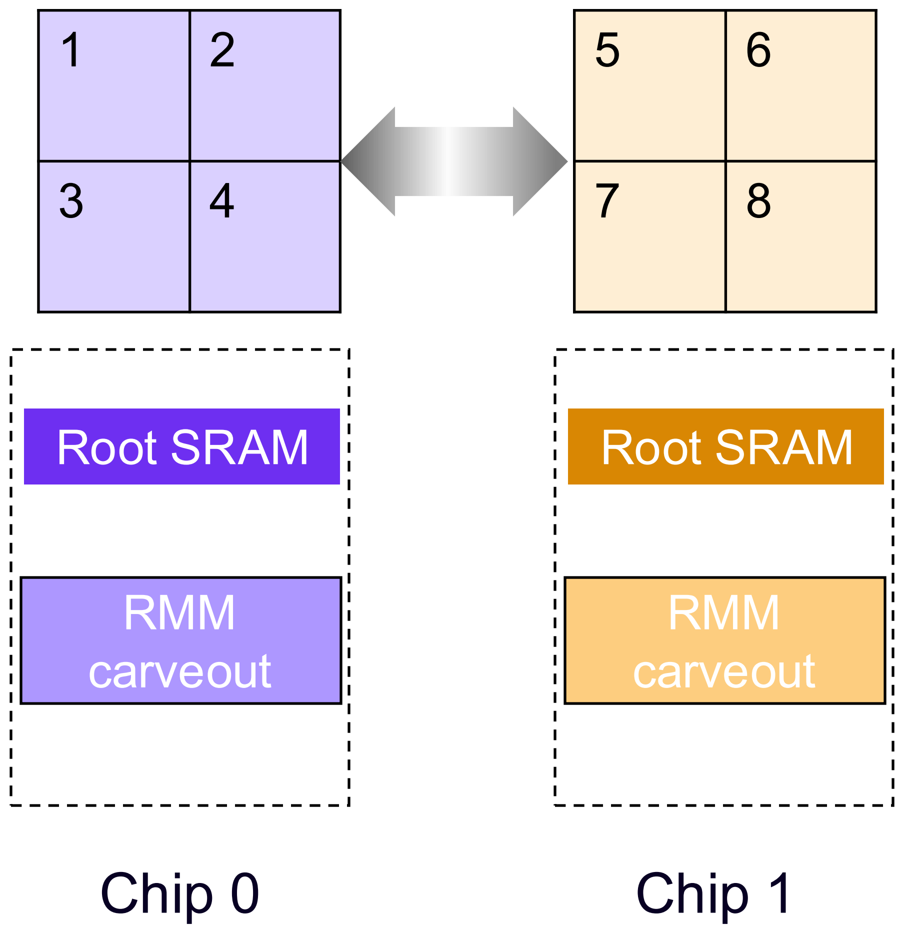
EL3 assisted mem reserve from Realm carveout

- Using an EL3 service to reserve memory in the Realm PAS carveout reduces design complexity and minimizes the need for platform-specific implementations in RMM.
- Hence the current proposal is to introduce an EL3 RMMD service to "reserve" memory from the Realm PAS carveout based on request from RMM.
  - There will be no free() support , hence overhead associated with traditional memory mgmt is absent.
  - EL3 will not map this memory in its own Stage 1 MMU nor try to access this memory.
  - Any misuse or incorrect handling of the allocation does not impact EL3 security.
- The proposal is that RMM allocates all required memory as part of boot
  - PCPU memory will be allocated as part of each individual CPU boot
  - Global memory will be allocated during primary CPU boot.
  - Failure in boot phase if the platform does not provision adequate memory.



```
uint64_t reserve_mem(uint64_t
size, uint64_t align)
{
    uint64_t align_mask = align
    - 1;
    uintptr_t addr;

    addr = (top_mem - size) &
    ~align_mask;
    if (addr <
    RMM_PAYLOAD_LIMIT) {
        return 0;
    }
    top_mem = addr;

    return addr;
}
```

Proposed NUMA handling for local (PCPU) RMM data



- EL3 creates Realm PAS carveout in the memory nodes.
- EL3 firmware, if it has support for NUMA, will locate its local data to corresponding memory nodes.
- When RMM makes a request to reserve memory for its PCPU data, EL3 reserves memory Realm carveout local to the CPU which made the request.
  - This can leverage NUMA framework in TF-A, eliminating the need for EL3 to perform explicit runtime topology lookups. As a result, the implementation will be simple.

RMM_RESERVE_MEM

Input values

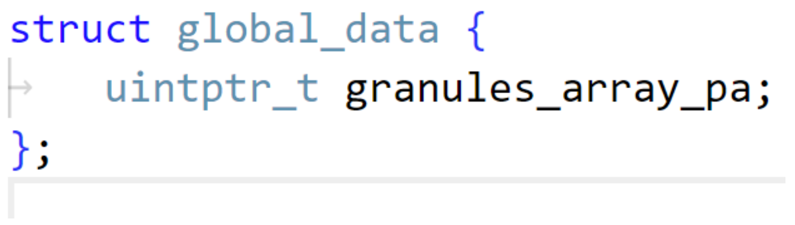
Name	Register	Bits	Type	Description
FID	x0	[63:0]	UInt64	Command FID
Size	x1	[63:0]	UInt64	Size in bytes
Alignment	x2	[63:56]	UInt8	Alignment requirement in power of 2. A value of 16 would return a 64 KB aligned base address
Flags	x2	[31:0]	UInt32	[0]: determine whether the allocation should allocate from a pool close to the calling CPU. [31:1]: reserved

Output values

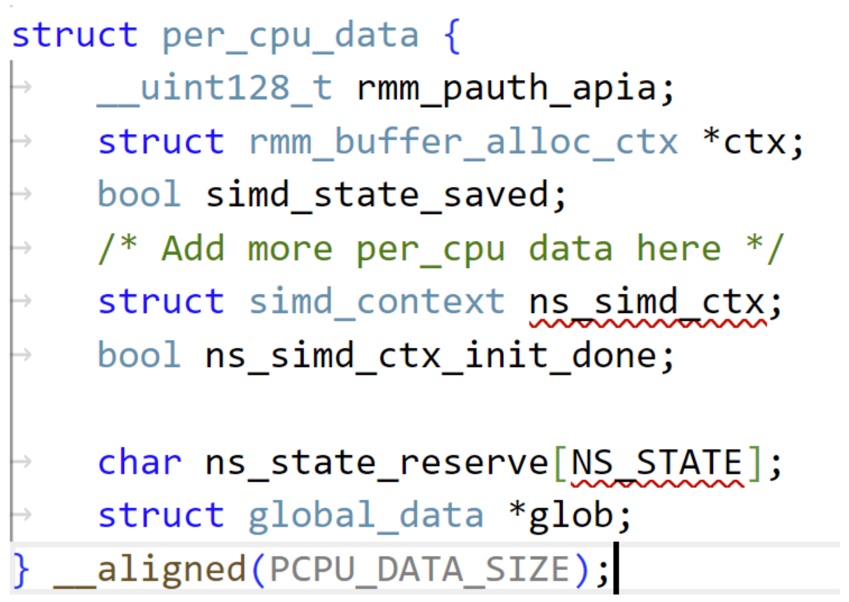
Name	Register	Bits	Type	Description
Result	x0	[63:0]	UInt64	Error code {E_RMM_INVAL, E_RMM_UNK, E_RMM_NOMEM, E_RMM_OK}
Address	x1	[63:0]	UInt64	PA of start of reserved mem

Enhance Boot protocol for RMM

- EL3 is expected to facilitate LFA of RMM.
- RMM will maintain a structure for global and PCPU allocations. This will need to be passed from old RMM to new RMM.
- Propose to add a `cookie` in boot interface
  - Cold boot uses [x0 – x3] , add x4 as cookie.
  - Warm boot uses [x0], add x1 as cookie.
  - On successful RMM boot, returns `cookie` in x1 back to EL3.
- EL3 keeps the `cookie` on per-CPU basis and passes it back to RMM for next boot – either next warmboot or LFA update boot.
  - Initial value of cookie is 0.
- RMM implementation detail:
  - RMM populates the global and PCPU allocations and passes address of per_cpu_data object as cookie back to EL3 when boot completes.
  - If the value of the cookie is non-zero, RMM will skip initialization and bootstrap from the provided per_cpu_data structure



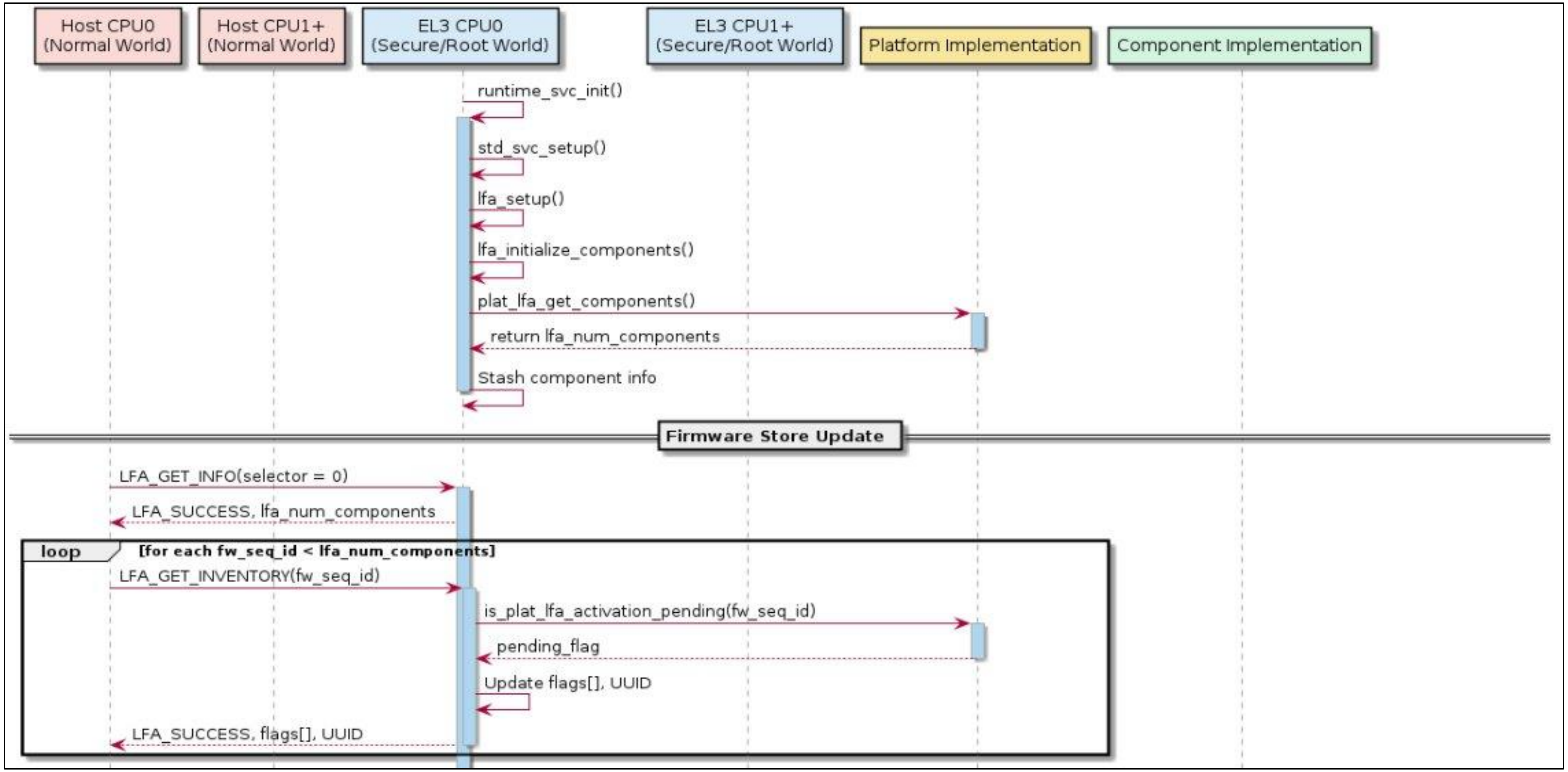
```c
struct global_data {
	uintptr_t granules_array_pa;
};
```



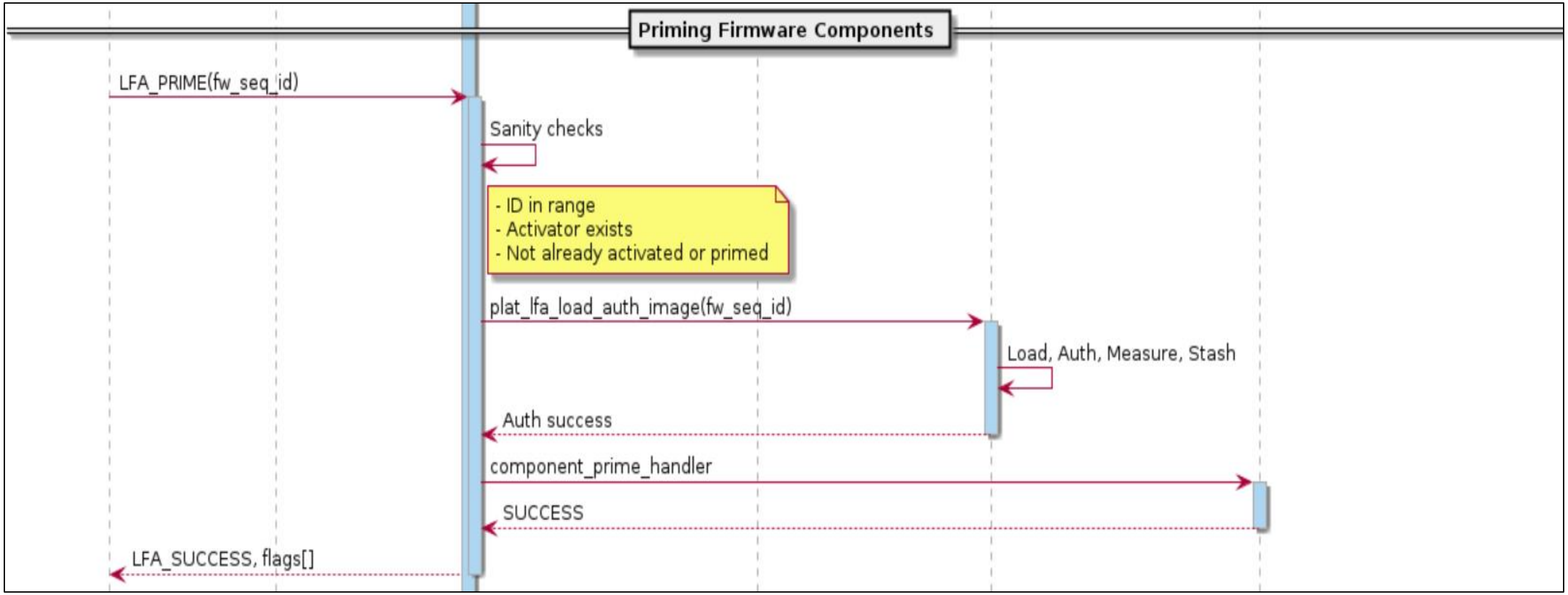
```c
struct per_cpu_data {
	__uint128_t rmm_pauth_apia;
	struct rmm_buffer_alloc_ctx *ctx;
	bool simd_state_saved;
	/* Add more per_cpu data here */
	struct simd_context ns_simd_ctx;
	bool ns_simd_ctx_init_done;

	char ns_state_reserve[NS_STATE];
	struct global_data *glob;
} __aligned(PCPU_DATA_SIZE);
```

LFA SMC Implementation in EL3

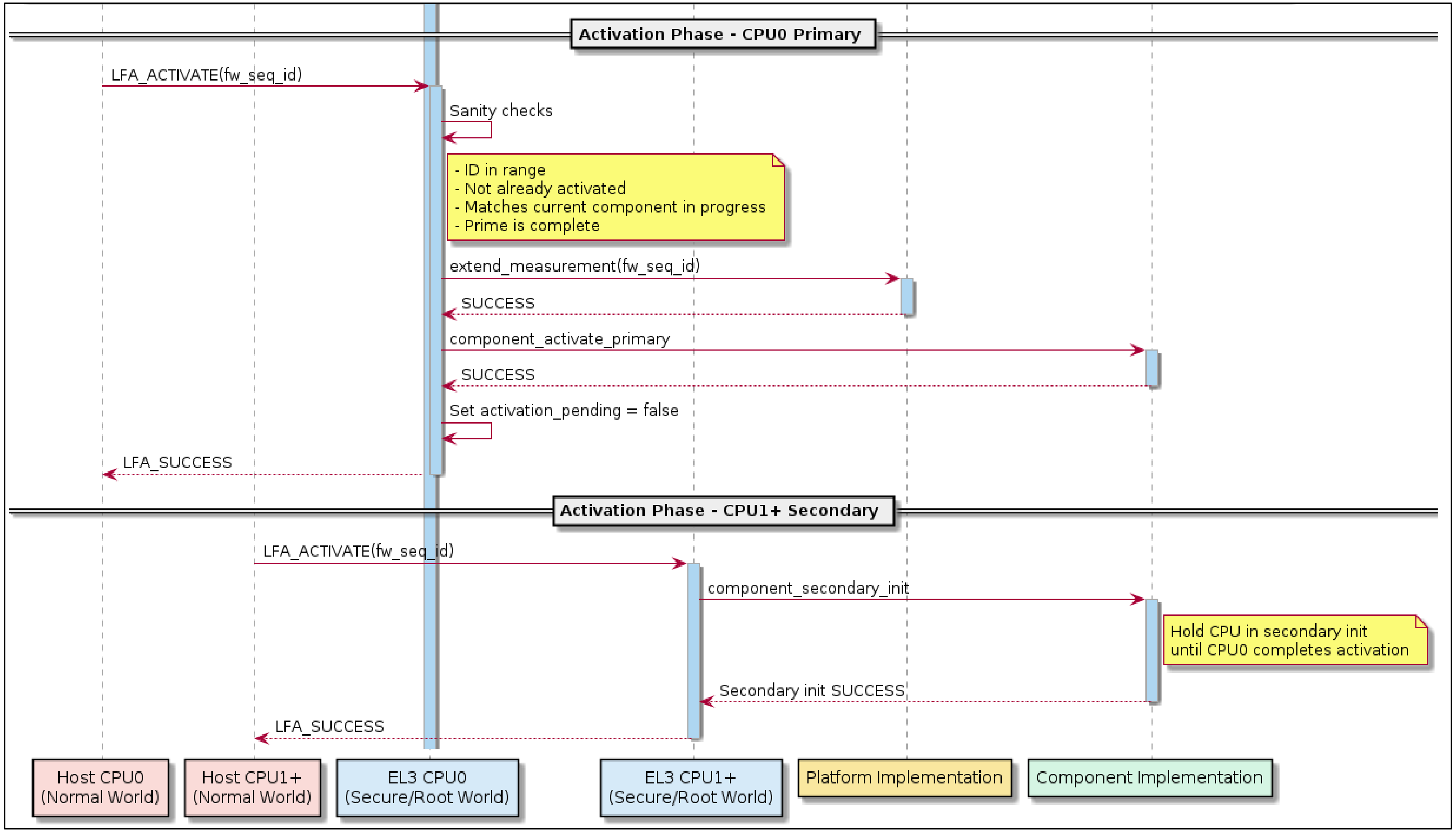
LFA Flow – Firmware Discovery


Host CPU0 (Normal World)
Host CPU1+ (Normal World)
EL3 CPU0 (Secure/Root World)
EL3 CPU1+ (Secure/Root World)
Platform Implementation
Component Implementation
runtime_svc_init()
std_svc_setup()
lfa_setup()
lfa_initialize_components()
plat_lfa_get_components()
return lfa_num_components
Stash component info
Firmware Store Update
LFA_GET_INFO(selector = 0)
LFA_SUCCESS, lfa_num_components
loop
[for each fw_seq_id < lfa_num_components]
LFA_GET_INVENTORY(fw_seq_id)
is_plat_lfa_activation_pending(fw_seq_id)
pending_flag
Update flags[], UUID
LFA_SUCCESS, flags[], UUID

LFA Flow – Priming Firmware Component



LFA Flow – Activating Firmware Component



Activation Phase - CPU0 Primary
LFA_ACTIVATE(fw_seq_id)
Sanity checks
- ID in range
- Not already activated
- Matches current component in progress
- Prime is complete
extend_measurement(fw_seq_id)
SUCCESS
component_activate_primary
SUCCESS
Set activation_pending = false
LFA_SUCCESS
Activation Phase - CPU1+ Secondary
LFA_ACTIVATE(fw_seq_id)
component_secondary_init
Hold CPU in secondary init
until CPU0 completes activation
Secondary init SUCCESS
LFA_SUCCESS
Host CPU0 (Normal World)
Host CPU1+ (Normal World)
EL3 CPU0 (Secure/Root World)
EL3 CPU1+ (Secure/Root World)
Platform Implementation
Component Implementation

Merci
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Thank You
감사합니다
ধন্যবাদ
Kiitos
شكرًا
ধন্যবাদ
תודה
ధన్యవాదములు
Köszönöm

Backup

LFA SMC Table (1/3)

SMC Name	Description	Inputs	Outputs	Return Codes
LFA_VERSION	Retrieve the LFA version	None	Major, Minor version	LFA_SUCCESS, LFA_NOT_SUPPORTED
LFA_FEATURES	Use to retrieve the existence of functions in the LFA ABI.	FID	None	LFA_SUCCESS, LFA_NOT_SUPPORTED
LFA_GET_INFO	Retrieves the number of components under the supervision of LFA	FID, lfa_info_selector=0	lfa_num_components	LFA_SUCCESS
LFA_GET_INVENTORY	To discover the Firmware component managed by LFA	FID, fw_seq_id	UUID_0, UUID_1, Flags - 1. Activation capable 2. Activation pending 3. May_reset_cpu 4. cpu_rendezvous_optional	LFA_SUCCESS, LFA_INVALID_PARAMETERS, LFA_WRONG_STATE.

LFA SMC Table (2/3)

SMC Name	Description	Inputs	Outputs	Return Codes
LFA_PRIME	To prepare platform for live activation of the given component	FID, fw_seq_id	Flags - 1. call_again	LFA_SUCCESS, LFA_AUTH_ERROR, LFA_NO_MEMORY, LFA_DEVICE_ERROR, LFA_WRONG_STATE, LFA_BUSY, LFA_PRIME_FAILED
LFA_CANCEL	To abort firmware activation process during prime or activate stages	FID, fw_seq_id		LFA_SUCCESS, LFA_BUSY, LFA_INVALID_PARAMETERS

LFA SMC Table (3/3)

SMC Name	Description	Inputs	Outputs	Return Codes
LFA_ACTIVATE	To request an immediate activation of the firmware component primed for activation.	FID, fw_seq_id, Flags - 1. skip_cpu_rendezvous entry_point_address context_id	Flags - 1. Call_again	LFA_SUCCESS, LFA_AUTH_ERROR, LFA_NO_MEMORY, LFA_DEVICE_ERROR, LFA_WRONG_STATE, LFA_BUSY, LFA_ACTIVATION_FAILED, LFA_INVALID_PARAMETERS LFA_INVALID_ADDRESS, LFA_COMPONENT_WRONG_STATE