Caliptra Subsystem

Hardware Specifications

V0.3

Contents

Caliptra Subsystem Architecture

Caliptra Subsystem Requirements

Definitions

RA: Recovery Agent

MCI: Manufacturer Control Interface

MCU: Manufacturer Control Unit

Caliptra-Passive-Mode (Legacy)

- SOC manages boot media peripherals and Caliptra is used as Root of trust for measurements.

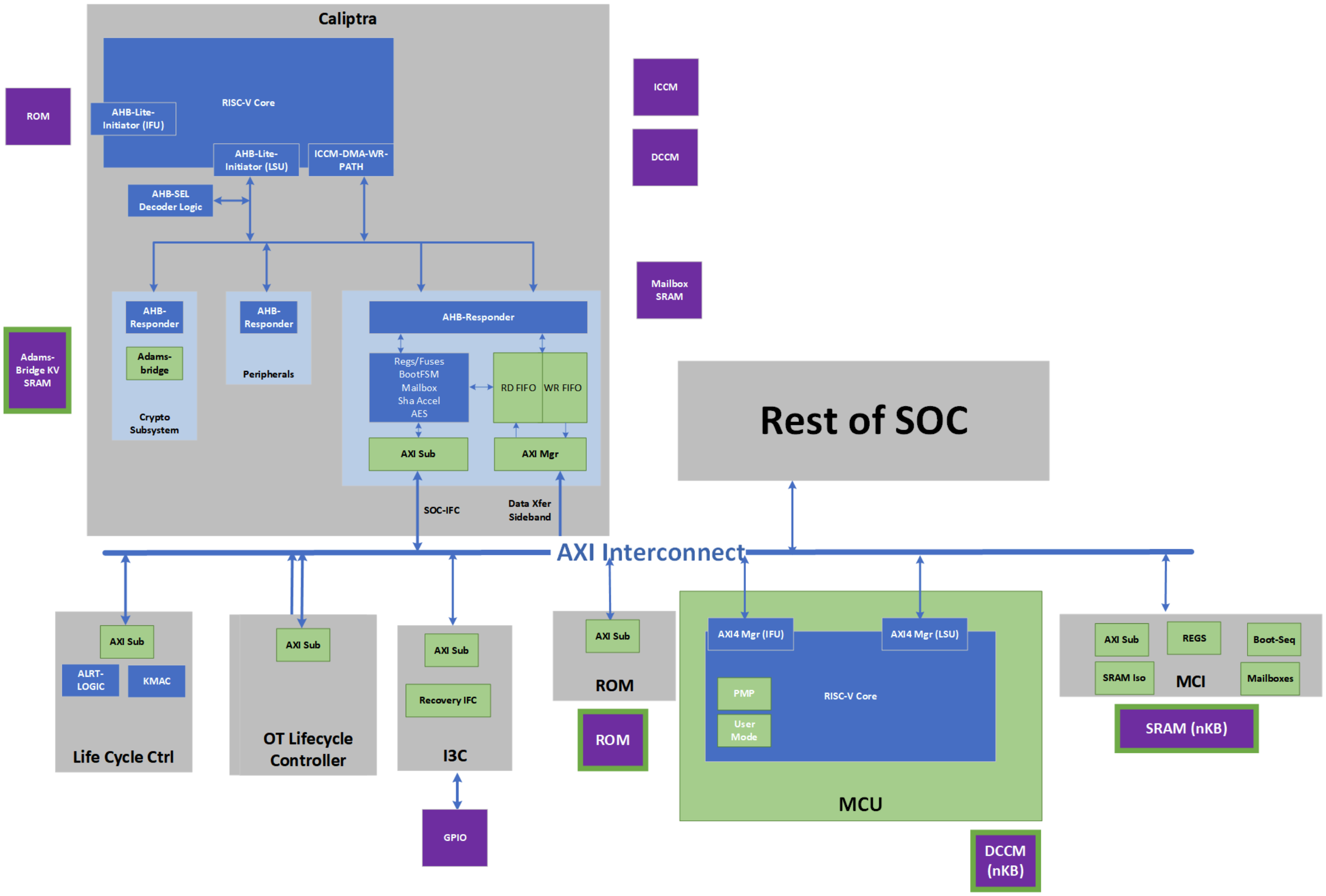
Caliptra-Active-Mode

- Caliptra owns the recovery interface (peripheral independent) and Caliptra is THE RoT of the SoC. Any SOC specific variables that needs to be stored and retrieved securely from NV storage is handled using Caliptra.

Caliptra Subsystem Architectural Requirements

1. MCU [Manufacturer Control Unit] runs Manufacturer specific FW authenticated, measured & loaded by Caliptra at boot time.
2. MCU will use Caliptra RT FW to auth/measure/load all of the FW belonging to the SOC.
3. MCU ROM/FW on MCU should be capable of performing SOC specific initialization.
4. MCU ROM/FW should be able to perform SoC management functions including performing reset control, SoC specific security policy enforcement, SOC initial configuration (eg. any GPIO programming, glitch detection circuits, reading/moving non-secret fuses etc.).
    - Note: Widgets that toggle the reset or other wires that set security permissions are SOC specific implementations.
5. Fuse controller for provisioning Caliptra fuses -> IFP (In-field programmable) fusing is performed by MCU RT; SW partition fuses in fuse controller are managed by MCU (ROM or RT); Caliptra HW is responsible for reading the secret fuses (Caliptra ROM, MCU ROM or any other SOC ROM or any RT FW should NOT have access to read the secret fuses in production).

6. Recovery stack must be implemented. Please refer to I3C recovery section for more details and references.
7. OCP Recovery registers implemented in I3C must follow the security filtering requirements specified in the recovery implementation spec (eg. MCU can ONLY access subset of the recovery registers as defined by the recovery implementation).
8. Supports silicon t0 boot to load and run required FW across chiplets.
9. OCP recovery stack is implemented in Caliptra for Caliptra-Active-Mode
10. MCU SRAM (or part of the SRAM that is mapped for Code/Data execution) should be readable/writeable ONLY by Caliptra until Caliptra gives permission to MCU to use it.
11. Caliptra can only load its FW, SOC manifest and MCU FW by reading the recovery interface registers.
12. Caliptra ROM should know the offset of the recovery interface at its boot time. SOC has a choice to program this interface using MCU ROM or Hard strapped register address at integration time (Hard strapping at SOC integration time is recommended)
13. Caliptra HW must know the offset of the registers where secrets (UDS, FE) and assets (PK hashes) exist at boot time. These should be hard strapped/configured at integration time.
14. Caliptra ROM must know the address to where UDS-seed needs to be written to; this address should be hard strapped/configured at integration time.
15. Any registers holding secrets/assets in fuse controller must follow the same rules as Caliptra 1.0/2.0 fuse register requirements (eg. clear on debug, clear on read once etc.)
16. MCU SRAM and ROM sizes should be configurable at SOC integration time.



Caliptra Subsystem HW Requirements

Caliptra 2.0 HW requirements (Subsystem Support)

1. Full AXI read/write channels (aka AXI manager) for subsystem (for MCU and Caliptra)
   a. For backward compatibility, AXI mgr. interface can be a no-connect and that configuration is validated.
2. HW logic/Programmable DMA
   a. Read MMIO space for variable length.
   b. Data returned from the above read can be exposed directly to the FW OR allow it to be written to a subsystem/SOC destination as programmed by ROM/FW.
   c. Programmable logic to allow for SOC directed writes (from FW or from the above route back) to be sent through the SHA accelerator.
   d. (Future open/stretch goal): If AES engine accelerator is integrated into Caliptra, then implement decryption before sending the writes back to the destination programmed by the ROM/FW.

        e. This widget should have accessibility in manufacturing and debug mode over JTAG (can be exposed using the same JTAG interface as Caliptra 1.0). Ensure through validation that no asset can be read using this widget in those modes.

3. Expand manuf flow register to include UDS programming request steps
4. SOC Key Release HW
    a. Separate SOC Key Vault must be implemented (it is a separate structure from the current Caliptra KV).
        i. In at least one configuration, the SOC KV must be implemented as an SRAM that is external and configurable by the SOC OR or an internal configurable SOC KV structure. If this is achievable within the Caliptra 2.0 milestone, only one of these would be the chosen implementation and configuration. This will be a design decision based on effort & schedule.
    b. If implemented as a SRAM, data written and read into the SOC KV SRAM is decrypted & encrypted so that SOC DFT or other side channels cannot exfilterate the data.
    c. Caliptra FW will indicate to the SOC KV Release HW to release the key by supplying the key handle to read from.
        i. Destination address to which the key must be written to is programmed by the Caliptra FW into AXI MGR DMA HW.
    d. SOC KV must have the same attributes as Caliptra internal KV. Additionally, it must also have an attribute of read-once and clear.
    e. This same implementation will be reused for OCP Lock flow too

Caliptra 2.0 HW requirements (Not subsystem related)

1. Ability to use two or more cryptos concurrently
2. Change APB -> AXI-Sub with the same capabilities (AXI-ID filtering replaces PAUSER based filtering, multiple targets for SHA acc, mailbox, fuses, registers etc. all)

    a. Future/Stretch Goal: Parity support on AXI-SUB & MGR

MCU HW requirements

1. MCU should not be in the FIPS boundary and must not have any crypto functions. MCU must use Caliptra for any security flows (eg. security policy authorization) or traditional RoT support (eg. SPDM).
2. MCU VeeR must support PMP & User mode.

3. MCU AXI is directly connected to the SOC fabric in a way that allows a MMIO based control of the SoC as required for SOC security, manageability and boot functionality.
4. MCU HW should add AXI-ID without any involvement of the ROM or FW running on the MCU; Implying any AXI access out of MCU (LSU side) should carry MCU USERID that HW populates.
5. MCU executes from a SRAM that is at subsystem level.
6. MCU uses instruction cache for speed up
7. It is required that all NV read/writes (eg. NV variables in flash) that require a RoT support to securely store/restore must go through Caliptra.
8. NV-storage peripheral shall be built in such a way that it will only accept transactions from MCU.
9. Support for MCU first fetch vector to direct towards MCU SRAM post reset

Subsystem Components HW requirements:

1. **Fabric**
    a. AXI Interconnect connects Caliptra, I3C, Fuse Controller, Life Cycle Controller, MCU and its memory components with the rest of the SOC. Because each SOC integration model is different, AXI interconnect is not required to be implemented by the subsystem but subsystem must validate using an AXI interconnect VIP to ensure all the components operate per the flows documented in this specification. For the VIP interconnect configuration, all subtractively decoded transactions are sent towards SoC. AXI interconnect & Subsystem is validated with the assumption that all of them are running on the same clock domain.
    b. To be documented: AXI-ID requirements
2. **MCU SRAM**
    a. Since it's used for instruction and data execution – therefore requires AXI Sub with USERID/AXI-ID filtering.
    b. Provide JTAG accessibility to allow for SRAM to be populated in a secured debug mode at power on time (debug policies will be same as Caliptra)
    c. MCU SRAM should have an aperture that can only be unlocked by Caliptra after it loads the image
    d. MCU SRAM has an aperture for MCU ROM to use part of the SRAM as a stack for its execution.
    e. MCU SRAM supports an aperture to be used as a MCU Mailbox.
3. **MCU ROM**
    a. MCU ROM also needs to have AXI Sub for any data access from MCU and thereby requires AXI-ID/USERID filtering.

4. **I3C**
    a. I3C on AXI interconnect with AXI Subordinate
    b. Spec 1.1.1 aligned, but only with optional features that are required per PCIe ECN # <>
    c. AXI Sub must be supported.
    d. UserID to MCU and Caliptra
    e. MCU access lock for I3C recovery and data (FIFO) registers until recovery flow is completed. In other words, MCU ROM must not impact the data flow into Recovery IFC registers.
    f. Stretch Goal: DMA data payload back to destination (Caliptra or MCU)
5. **Fuse Controller**
    a. AXI sub interface
    b. Secrets (separate USERID and access constraints) vs SW partition separation
    c. Registers implemented for "Secrets" should follow the same rules as Caliptra (no scan, clear on specific life cycle/security states)
6. **HW logic** to move secret fuses from Fuse controller to Caliptra.

Caliptra Subsystem Architectural Flows

Subsystem (Pre-FW Load) Boot Flow

Note: Any step done by MCU HW/ROM would have been performed by "SOC Manager" in Caliptra 1p0.

1. SoC (using its HW or MCU ROM) performs pre-steps like bringing up CRO or PLL, MBIST flows on SRAMs, SRAM Init etc.
2. SOC will assert MCU & Caliptra pwrgood and after 10 cycles MCU
3. MCU [ROM] or SOC Manager wrapper will bring up fuse controller and any other SOC specific infrastructure RTL modules (I3C, GPIO programming, Glitch detector programming etc.)
4. MCU ROM or SOC Manager wrapper will deassert Caliptra reset.
5. Caliptra HW will read the security centric (secret) fuses.
6. MCU [ROM] waits for ready_for_fuses to be asserted.
7. MCU ROM or SOC Manager will populate the remaining fuses of Caliptra and reads its own fuses (if any). Note that this step is gated behind the completion of security fuse writes to ensure the step has completed.
8. MCU ROM will write "fuse done" to Caliptra.
9. Caliptra will go through its boot flow of bringing up uC.

10. Caliptra ROM starts and executes various KATs flows.

Subsystem Boot Flow

If (Caliptra-Passive-Mode)

1. SOC Manager goes through legacy flows => loads Caliptra FW using legacy flows, Caliptra sets RT ready and SOC <-> Caliptra boot flow is done.

If (Caliptra-Active-Mode)

1. Caliptra ROM waits for SOC infrastructure readiness indication. If this indication is set, Caliptra will do the identity derviation flows. If it is not set, then this flow is run when the SOC infrastructure readiness indication is set.
2. Caliptra ROM will follow the recovery interface protocol to load its FW. Please see the specific section for next level specifics; At a high level, Caliptra ROM sets the device ready in the I3C controller and poll I3C for the payloads.
3. BMC or platform components will send the image (code or data) through OCP recovery flow protocol.
    a. Caliptra ROM should implement a recovery capability to allow for BMC to send ‘data’ instead of ‘code’ as a SOC specific configuration OR allow MCU ROM to send some data to be either integrity checked or authenticated (TBD). The data flow and code flow over recovery interface is the same from physical interface point of view and follows the recovery spec as implemented in Caliptra subsystem documentation (please see the recovery section).
    b. This need for data flow (from flash or BMC) is indicated by a SOC configuration bit to Caliptra ROM
    c. This ‘data’ flow is possible only before SOC infra readiness is set. This is specifically used for scenarios where PUF or other characterization data is coming off-chip (say a flash or BMC)
    d. This data must be hashed into PCR0
    e. To keep the scope limited, only one ‘data’ flow is allowed
4. If the data was required to be run (is indicated by a SOC configuration bit to Caliptra ROM), Caliptra ROM waits for SOC infrastructure readiness to be set. Once set, it will do the required key derivations.
5. Caliptra ROM will read the recovery interface registers (data payload registers) over AXI manager interface and write into Caliptra MB SRAM. The offset of the recovery interface registers are available through a config register that is set at SOC integration time or by MCU ROM.

    a. Note that an intelligent I3C peripheral could “stream” the image. This is a future enhancement.

6. Caliptra ROM will authenticate its image sitting in Caliptra MB SRAM
7. Caliptra ROM flow will be similar to Caliptra 1.0 flow with PQC FW Authentication.
8. Caliptra ROM will derive required keys similar to Caliptra 1.0 flow (while accounting for PQC)
9. Caliptra ROM will switch to RT image.
10. Caliptra RT FW will set the RECOVERY INTERFACE (IFC) to allow BMC’s Recovery Agent (RA) to send the next image (which MUST be SOC image manifest).
    a. BMC RA is required to know the different component of the images using the similar manifestation as MCTP PLDM components.
11. Caliptra RT FW will read the recovery interface registers over AXI manager interface and write the image to its mailbox.
12. Caliptra RT FW will authenticate SOC manifest using the keys available through Caliptra Image, authenticate the image, capture the measurement and capture the relevant information into DCCM.
13. Caliptra RT FW will set the RECOVERY INTERFACE (IFC) to allow BMC’s Recovery Agent (RA) to send the next image (which MUST be MCU image manifest).
    a. BMC RA is required to know the different component of the images using the similar manifestation as MCTP PLDM components.
14. Caliptra RT FW will read the recovery interface registers over AXI manager interface and write the image to MCU SRAM aperture (that is open to Caliptra only by HW construction).
    a. The address of the MCU SRAM is provided to Caliptra’s RT FW through SOC manifest.
    b. Note: From validation front, need to ensure the Caliptra ROM and MCU are aligned in endianness.
15. Caliptra RT FW will instruct Caliptra HW to read MCU SRAM and generate the hash (Caliptra HW will use the SHA accelerator and AXI mastering capabilities to do this)
    a. Open: Should we have a capability to do something like this for decryption too? (Key to be provided by MCU/SOC before running the decryption flow?)
16. Caliptra RT FW will use this hash and verify it against the hash in the SOC manifest.

17. Caliptra RT FW after verifying/authorizing the image and if it passes, it will set EXEC/GO bit into the register as specified in the previous command. This register write will also assert a Caliptra interface wire.
    a. MCU ROM will be polling the breadcrumb that the MCU SRAM has valid content and will jump to the MCU SRAM to execute from it.
        i. **NOTE:** This breadcrumb should be one of the status bits available on the MCU interface that is set by Caliptra GO bit wires.
    b. Until this step MCU SRAM aperture that holds the MCU RT FW and certain recovery interface registers are not accessible to MCU.
18. MCU RT FW will now set recovery flow is completed.
19. BMC will now do MCTP enumeration flow to MCU over I3C.
20. MCU RT FW is responsible for responding to all MCTP requests.
21. MCU RT FW will do the PLDM T5 flow, extract FW or configuration payload, use Caliptra to authenticate and deploy the rest of the images as described in run-time authentication flows.

Common Run-time Authentication Flows:

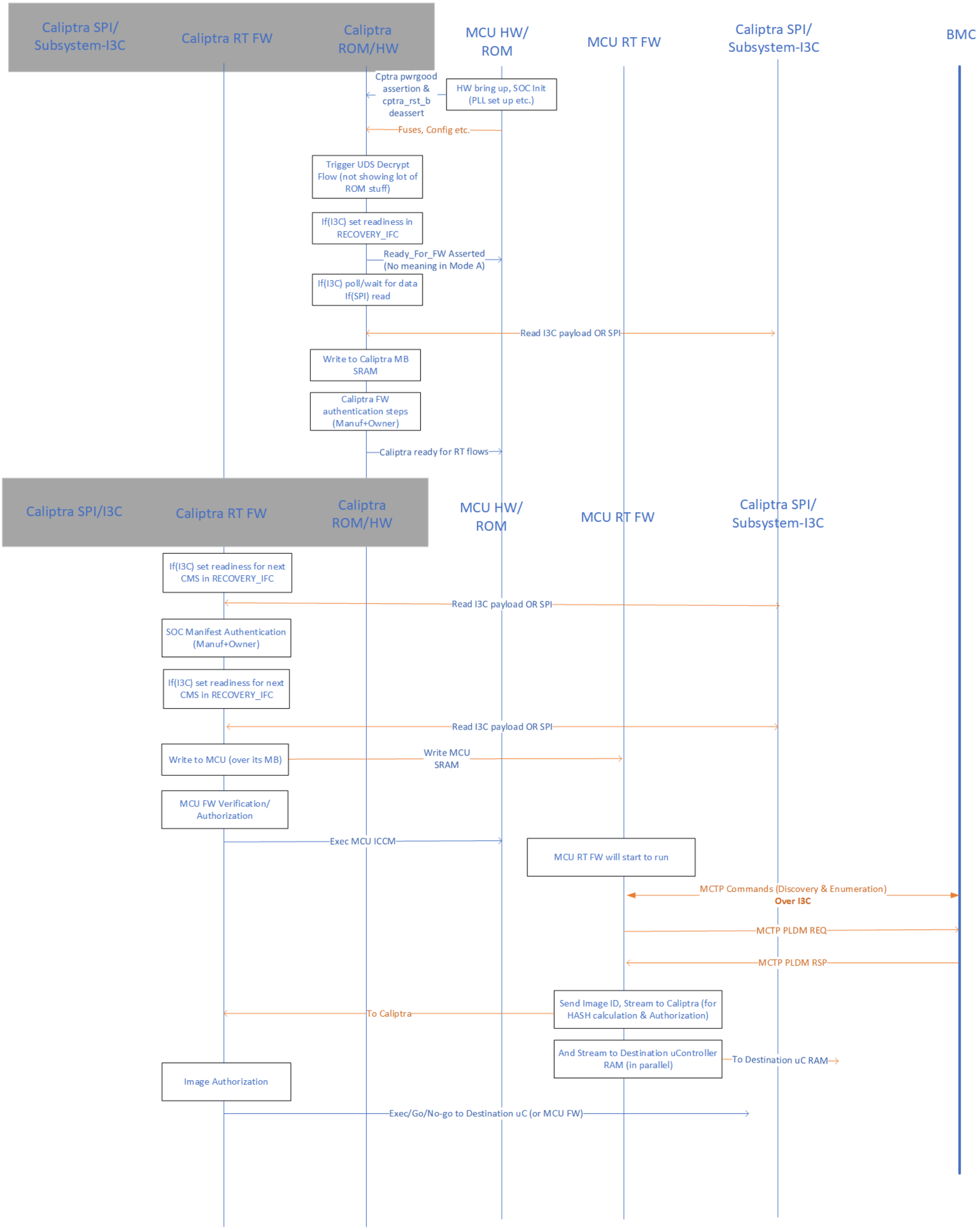
1. MCU RT FW will do PLDM T5 flow to obtain the FW images for downstream uControllers (or other SOC configuration)
2. MCU RT FW will send/stream the (FW or config) payload to Caliptra SHA Acc to perform hash measurements as the payload comes through the MCTP transport.
    a. MCU RT FW can either stage the entire image or write to the final destination as a part of the previous step depending on the SOC construction.
    b. Note: By SOC security/design construction, the FW/payload that is loaded must NOT be allowed to execute or be used until Caliptra authorizes that the FW/payload.
3. MCU RT FW will issue the imageID & GO-field bit (bit that Caliptra RT FW would set if the image authorization is successful) to Caliptra RT FW to start off the process of image authorization of the image that was hashed.
    a. Caliptra RT FW will obtain this hash from the internal SHA accelerator register that was used to hash in the previous step.
4. Caliptra RT FW after verifying/authorizing the image and if it passes, it will set EXEC/GO bit into the register as specified in the previous command. This register write will also assert a Caliptra interface wire.
5. MCU RT FW has an option of looking at the Mailbox command success or read the register or use the wire that the register will drive to allow the execution of the

FW. This wire allows SOCs to construct a hardened logic of allowing executions from ICCM/TCMs only after the wire is set.

   a. SOC construction outside of MCU SRAM is SOC specific construction and the spec here provides recommendations on using MCU and Caliptra to build such a logic.

6.

Caliptra-Active-Mode Flow Diagram


Caliptra SPI/ Subsystem-I3C
Caliptra RT FW
Caliptra ROM/HW
MCU HW/ ROM
MCU RT FW
Caliptra SPI/ Subsystem-I3C
BMC
Cptra pwrgood assertion & cptra_rst_b deassert
HW bring up, SOC Init (PLL set up etc.)
Fuses, Config etc.
Trigger UDS Decrypt Flow (not showing lot of ROM stuff)
If(I3C) set readiness in RECOVERY_IFC
Ready_For_FW Asserted (No meaning in Mode A)
If(I3C) poll/wait for data If(SPI) read
Read I3C payload OR SPI
Write to Caliptra MB SRAM
Caliptra FW authentication steps (Manuf+Owner)
Caliptra ready for RT flows
Caliptra SPI/I3C
Caliptra RT FW
Caliptra ROM/HW
MCU HW/ ROM
MCU RT FW
Caliptra SPI/ Subsystem-I3C
If(I3C) set readiness for next CMS in RECOVERY_IFC
Read I3C payload OR SPI
SOC Manifest Authentication (Manuf+Owner)
If(I3C) set readiness for next CMS in RECOVERY_IFC
Read I3C payload OR SPI
Write to MCU (over its MB)
Write MCU SRAM
MCU FW Verification/ Authorization
Exec MCU ICCM
MCU RT FW will start to run
MCTP Commands (Discovery & Enumeration) Over I3C
MCTP PLDM REQ
MCTP PLDM RSP
Send Image ID, Stream to Caliptra (for HASH calculation & Authorization)
To Caliptra
And Stream to Destination uController RAM (in parallel)
To Destination uC RAM
Image Authorization
Exec/Go/No-go to Destination uC (or MCU FW)

Hitless Updates

Caliptra Hitless Update

1. Payloads of all hitless update come over MCTP PLDM flow to the MCU similar to the boot time flows.
2. MCU provides SOC manifest to Caliptra and waits for authentication to be successful. If this wasn't provided Caliptra will use the latest SOC manifest available.
    a. If failed, MCU uses MCTP PLDM to report the same to the update agent using PLDM protocol
3. MCU provides the Caliptra FW using Caliptra Mailbox using the hitless update flows documented in the Caliptra specification

MCU Hitless Update

1. Payloads of all hitless update come over MCTP PLDM flow to the MCU similar to the boot time flows.
2. MCU provides SOC manifest to Caliptra and waits for authentication to be successful. If this wasn't provided Caliptra will use the latest SOC manifest available.
    a. If failed, MCU uses MCTP PLDM to report the same to the update agent using PLDM protocol
3. MCU stages the incoming FW payload in an SOC-defined staging SRAM and provides the MMIO address of the staging memory to Caliptra. It is better to keep this as a part of the authenticated SOC manifest (as a configuration) from a security perspective.
4. Caliptra RT FW will use the Caliptra DMA engine to issue the read and hash the image (note that the length of the image must be part of the SOC manifest)
5. Caliptra RT FW after verifying/authorizing the image and if it passes, waits for activate command to be issued from MCU. MCU will get this command over MCTP PLDM flow; At this point, Caliptra will **reset** EXEC/GO bit into the register as specified in the previous command. This register write will also deassert a Caliptra interface wire.
6. MCU HW logic will use this indication to initiate MCU uC reset flow
    a. MCU HW logic sends a reset-go-req to MCU uC (an interrupt)
    b. MCU HW logic waits for reset-go-ack from MCU uC (Note that this handshake exists to ensure uController is in an appropriate quiescent state to take the reset)
    c. MCU HW logic will assert the reset to the MCU uC

7. Caliptra RT FW will wait for reset assertion and then read the staged SRAM over AXI manager interface and write the image to the MCU SRAM aperture (that is open to Caliptra only by HW construction).
    a. The address of the MCU SRAM is provided to Caliptra's RT FW through SOC manifest.
    b. Note: From the validation front, need to ensure the Caliptra ROM and MCU are aligned in endianness.
    c. Note: True downtime of MCU is from when its reset is asserted; It is SOC implementation requirement that it handles (eg. through buffering) all transactions to MCU while it is going through a hitless update.
8. After the MCU SRAM is populated, Caliptra RT FW will set EXEC/GO bit into the register as specified in the previous command. This register write will also assert a Caliptra interface wire.
9. MCU HW logic will use this indication to deassert the MCU reset.
10. MCU ROM will look at the breadcrumb that the MCU SRAM has valid content and will start the execution from it directly.
    a. **NOTE:** This breadcrumb should be one of the status bits available on the MCU interface that is set by Caliptra GO bit wires.

SOC-FW Hitless Update

SOC may have other components that may need to be updated at run-time in a hitless/impactless manner.

The update flow will follow the same sequence as MCU Hitless update except they are executed by the MCU by using Caliptra as the RoT engine for doing all the required authentication/authorization flows.

Further SOCs may require the hitless update without impacting the workloads/VMs running on the host or the VMs using the devices. This essentially means that impactless update must happen without causing any timeouts to the in-flight transactions. While the treatment of those transactions are device dependent, Caliptra subsystem must provide a way to be able to authenticate and activate the FW in the shortest time possible.

Caliptra subsystem provides this architectural capability as follows:

1. MCU provides SOC manifest to Caliptra and waits for authentication to be successful. If this wasn't provided Caliptra will use the latest SOC manifest available.
    a. If failed, MCU uses MCTP PLDM to report the same to the update agent using PLDM protocol

2. MCU stages all the incoming FW payload in an SOC-defined staging memory and provides the MMIO address of the staging SRAM to Caliptra. It is better to keep this as a part of the authenticated SOC manifest (as a configuration) from security perspective.
3. Caliptra RT FW will use the Caliptra DMA engine to issue the read and hash the image (note that the length of the each image must be part of the SOC manifest)
4. Caliptra RT FW will verify & authorize the images. It will also compare the hash of the images against the “current” hash of each of the image.
5. MCU will send the ‘activate’ command to Caliptra (which is part of PLDM spec that MCU understands)
6. If MCU FW is updated/new, Caliptra will execute the MCU Hitless update flow.
7. Caliptra RT FW will then set the GO bits for all the SOC FWs that are updated (vs what was already running)
8. SOC specific logic & MCU RT FW will use this information to update the remaining FW using SOC specific architectural flows.
    a. Note: Since this work is mainly distribution of the FW to the destination uCs, SOC should be built to do this flow as fast as possible to meet workload non-interruption/impactless requirements.

Multi-Chiplet Flows (Subsystem-Active-Mode)

Generic FW Load Flows

Note: Additional control signals that MCU would control are SOC specific and are implemented through SOC widget(s).

1. Primary tile uses Caliptra-Active-Mode at its silicon boot time
2. Secondary tile’s MCU ROM will go through the same common boot flow as the primary tile (except the peripheral could be inter-chiplet link).
3. Secondary tile’s MCU ROM will wait for inter-chiplet link to be available for use (this would be an indication to MCU ROM)
4. Primary tile’s MCU RT FW will fetch the secondary tile’s FW using MCTP PLDM T5 flow and ‘stream’ using the same recovery interface protocol to the secondary tile(s).
5. Based on SOC integration, inter-chiplet could be an intelligent peripheral that can DMA or implement data payload registers for Caliptra to read.
    a. Note that the indication from Caliptra for “next-image” follows the same recovery interface protocol.
    b. Note that to load the remaining images of a secondary tile, SOC can choose to do recovery flow for rest of the remaining images. Depending on the SOC architecture and chiplets, MCU RT FW may coordinate the

SOC to boot in such a way that it “broadcasts” the same image to multiple chiplets that require the same image. This is a SOC optimized flow outside of Caliptra or Subsystem Context.

Debug Policy Authentication Flows

<To be filled later>

SOC Keys (Subsytem-Active-Mode)

Generic SOC Key Release Architecture (Subsytem-Active-Mode)

Note: Each SOC key requires very specific algorithms/flows to reach to the final key that is used for SOC specific reasons. For example, PCIe IDE, Memory (DRAM) encryption keys or OCP LOCK related keys are all derived differently and the hardware required for each of them will be specific to those keys.

The key release architecture described in this section only describes moving the key from SOC Key Vault structure (which is implemented as per the requirements mentioned in “Caliptra HW requirements” section) to a SOC destination. .

Architectural Flow:

1. Caliptra FW gets a SOC Key Derivation Manifest from MCU. This manifest is authorized by Caliptra FW similar to any other image. The difference versus other images is that the content of this manifest is stored within Caliptra DCCM. Caliptra SOC key derivation manifest contains
    a. key-tag (eg. IDE-KEY-1, IDE-KEY-2, OCP-LOCK etc.)
    b. Destination address to which the key should be written to
    c. Additional meta-data (width of the key, type of key [eg. AES], if it’s a write to the same address or increments by 4B etc.) - Open: Full Meta-data definition.
2. MCU issues mailbox command to Caliptra FW to derive and release a particular key by providing the associate key-tag-ID in the mailbox command.
3. When a key needs to be released to the SOC destination address (after all the key derivations are completed and the final key is present in the SOC KV).
    a. Caliptra FW will write the destination address, value to be programmed, number of writes, if the address needs to be incremented (note that these are all Caliptra HW DMA parameters already).
    b. Caliptra FW will also instruct SOC key release HW to read a key handle.
    c. Caliptra DMA HW & key release HW together will read the SOC KV, and do the writes to the destination address (please refer to Caliptra AXI Manager and DMA Assist Overview).

PCIe IDE KM Flow (Subsytem-Active-Mode)

1. MCU receives the DOE mailbox commands deposited into its mailbox.
2. MCU FW will read its mailbox, strips out the DOE MB objects, extract the SPDM command/payload, structures into Caliptra mailbox format as in “Command Code” (that denotes the initiator/requestor information that is translated to KEY-ID-TAG that denotes the PCIe EP(s) or I3C etc. or other initiator specific information) + “Data” (where data is the command received).
3. Caliptra FW implements the SPDM responder stack to do the SPDM flow per requestor.
4. When IDE_KEY_PROGRAMMING object command is sent to Caliptra by MCU by the requestor/initiator, Caliptra FW will decrypt the key by using the shared key that was derived for that requestor and the key is released to the final destination using the Caliptra internal DMA widget.

Memory/DRAM encryption keys (Subsytem-Active-Mode)

1. MCU will initiate a Caliptra mailbox command to generate the memory encryption key and the instance(s) to which that write needs to be done.
2. Caliptra RT FW will use the iTRNG/eTRNG to generate the values and write to the requested memory controller(s) using the Caliptra internal DMA widget.

OCP LOCK (Subsytem-Active-Mode)

<TB Documented later>

I3C Recovery Interface

The I3C recovery interface acts as a standalone I3C target device for recovery. It will have a unique address compared to any other I3C endpoint for the device. It will comply with I3C Basic v1.1.1 specification. It will support I3C read and write transfer operations. It must support Max read and write data transfer of 1-256B excluding the command code (1 Byte), length (2 Byte), and PEC (1 Byte), total 4 Byte I3C header. Therefore, max recovery data per transfer will be limited to 256-byte data.

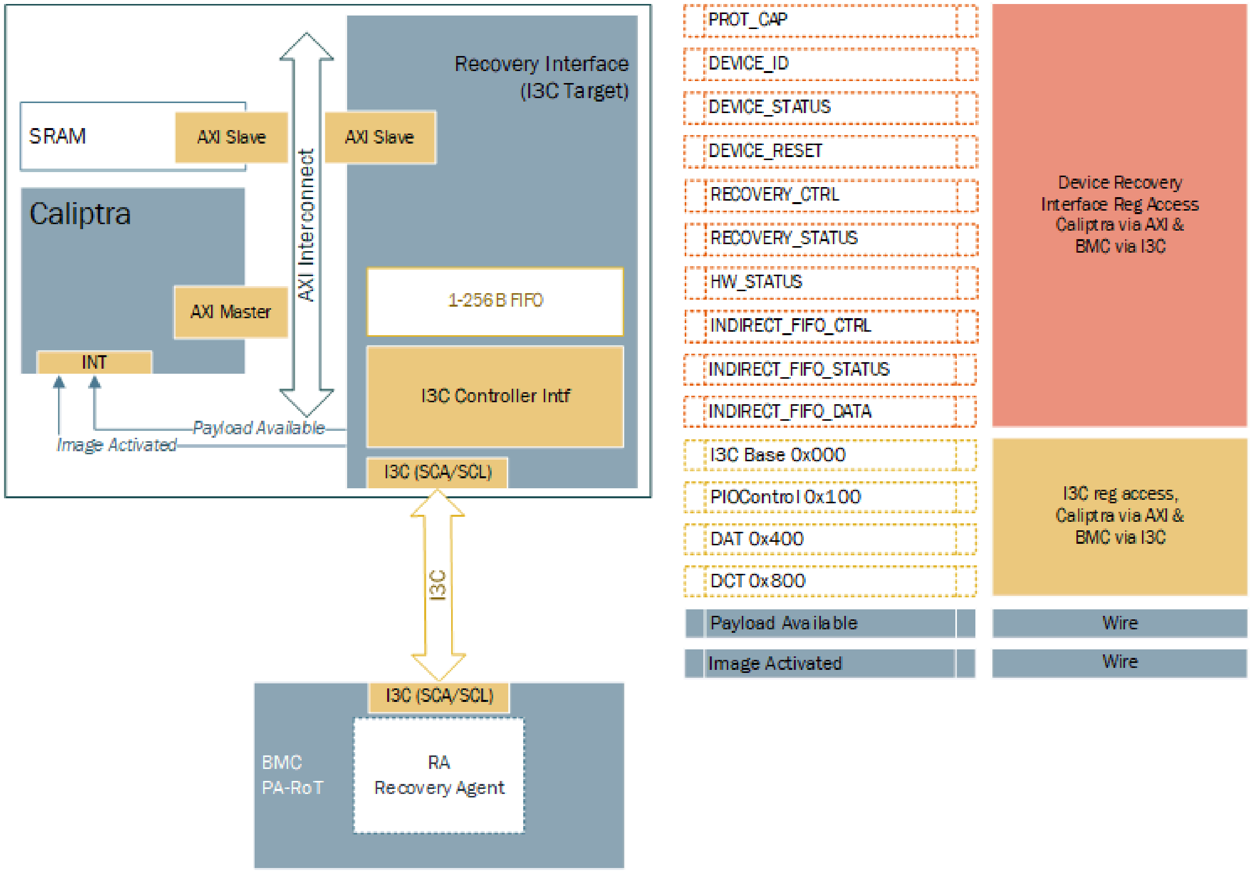
I3C recovery interface is responsible for the following list of actions:

1. Responding to command sent by Recovery Agent (RA)
2. Updating status registers based on interaction of AC-RoT and other devices
3. Asserting / Deasserting “payload_available” & “image_activated” signals

Recovery interface hardware specifications (links below).

1. OCP Recovery Document
2. [EXTERNAL] Flashless Boot using OCP, PCIe, and DMTF Standards - Google Docs

Recovery Interface hard coded logic [move to HW spec]



Hardware Registers [move to HW spec]

Hardware registers size is fixed to multiple of 4 bytes, so that firmware can read or write with word boundary. Address offset will be programmed outside of the I3C device. Register access size must be restricted to individual register space and burst access with higher size must not be allowed.

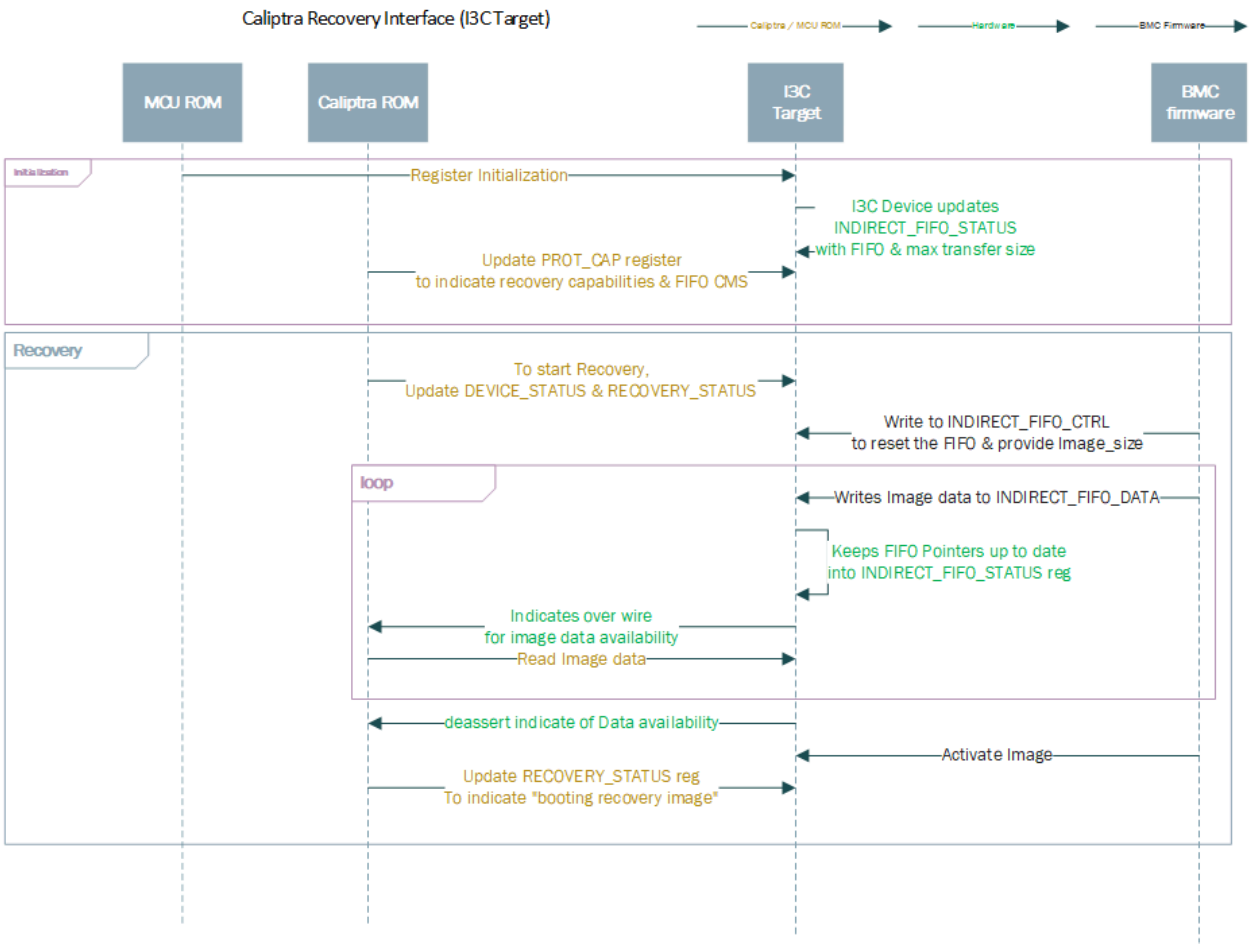
Register	Size (bytes)	Address	Valid Bytes
PROT_CAP	16	Offset + 0x00	15

DEVICE_ID	24	Offset + 0x10	24
DEVICE_STATUS	8	Offset + 0x28	7
DEVICE_RESET	4	Offset + 0x30	3
RECOVERY_CTRL	4	Offset + 0x34	3
RECOVERY_STATUS	4	Offset + 0x38	2
HW_STATUS	4	Offset + 0x3C	4
INDIRECT_FIFO_CTRL	8	Offset + 0x40	6
INDIRECT_FIFO_STATUS	24	Offset + 0x48	6
INDIRECT_FIFO_DATA	256	Offset + 0x6C	256

Note: Accessing the same address for INDIRECT_FIFO_DATA register will write or read the FIFO. It will not be available to access randomly as specified by the specification.

Recovery Interface Init Sequence

Caliptra ROM initializes PROT_CAP, DEVICE_ID, DEVICE_STATUS, RECOVERY_STATUS, HW_STATUS, INDIRECT_FIFO_STATUS (remove these two reg from ROM initialization) default values. **Note:** Any I3C initialization is done b/w MCU ROM, I3C target HW and I3C initiator. This is not part of this document.


Caliptra Recovery Interface (I3C Target)
Caliptra / MCU ROM
Hardware
BMC Firmware
MCU ROM
Caliptra ROM
I3C Target
BMC firmware
Initialization
Register Initialization
I3C Device updates INDIRECT_FIFO_STATUS with FIFO & max transfer size
Update PROT_CAP register to indicate recovery capabilities & FIFO CMS
Recovery
To start Recovery, Update DEVICE_STATUS & RECOVERY_STATUS
Write to INDIRECT_FIFO_CTRL to reset the FIFO & provide Image_size
loop
Writes Image data to INDIRECT_FIFO_DATA
Keeps FIFO Pointers up to date into INDIRECT_FIFO_STATUS reg
Indicates over wire for image data availability
Read Image data
deassert indicate of Data availability
Activate Image
Update RECOVERY_STATUS reg To indicate "booting recovery image"

Recovery Sequence

1. MCU Specific SoC init of I3C & Recovery interface.
    a. MCU ROM can set HW_STATUS register per recovery spec, at any time based on SOC specific conditions.
    b. MCU ROM will program DEVICE_ID register value based on associated fuse values.
    c. I3C device must update FIFO size (1-256 Byte), Max transfer size and type of region (tie this to 0x1) to INDIRECT_FIFO_STATUS register, which could be read by BMC firmware to understand the size of the FIFO & max transfer size.
2. Caliptra ROM will update PROT_CAP register, bit 11 to set to ‘1 *“Flashless boot (From RESET)”*. Caliptra ROM will set other register bits based on other recovery capabilities. PROT_CAP will also indicate support for FIFO CMS for I3C device by updating byte 10-11, bit 12 with 0x1 *“FIFO CMS Support”.*
3. To start recovery or boot, Caliptra ROM will write DEVICE_STATUS register to “RECOVERY_MODE” by writing byte 0, with data 0x3. Caliptra ROM will write DEVICE_STATUS register’s byte 2-3 to set the FSB parameter (0x12).

4. I3C Recovery HW will set byte 1 based on the DEVICE_STATUS register based on the rules defined for this register. This register status will assist BMC operation.
5. Caliptra ROM will write via DMA assist to RECOVERY_STATUS register with data of (byte 0, 0x1) and sets the recovery image index to 0x0 (Open: FW team to decide the index based on the PLDM component definition)
6. BMC will update INDIRECT_FIFO_CTRL with Component Memory Space (CMS) byte 0 with 0x0, Reset field byte 1 with 0x1 and Image size byte 2 to 5 field to size of the image.
7. BMC writes to INDIRECT_FIFO_DATA register. I3C device shall return a NACK response for any transfer that would cause the Head Pointer to advance to equal the Tail Pointer. BMC can implement flow control through NACK responses or by monitoring the FIFO space remaining via the Head and Tail Pointers.
8. The I3C device will keep head and tail pointers along with FIFO status up to date into INDIRECT_FIFO_STATUS register. I3C recovery interface HW wait for an update to INDIRECT_DATA_REG with 1-256B data from BMC.
9. If there is a write to INDIRECT_DATA_FIFO, I3C device will indicate data availability via side channel implemented as wire "payload_available" (for more details read here) to Caliptra. Caliptra HW will latch this wire into the register for Caliptra firmware to read.
10. Caliptra ROM arms DMA interface to read INDIRECT_FIFO_CTRL for the image size and programs DMA engine back to read the image data from INDIRECT_FIFO_DATA.
11. Steps 9 through 11 repeat until all the images are pushed over I3C and it matches the image size initialized into the INDIRECT_FIFO_CTRL register.
12. After the above steps, Caliptra ROM Firmware will wait for BMC to activate image indicated to Caliptra via side channel (for more details read here)
13. If the Image is activated, update RECOVERY STATUS to "Booting recovery image" by writing byte0, with data 0x2. If the image is authenticated, then the Caliptra RT FW will update the image index in RECOVERY_STATUS register (0x1 in byte 0, bits 7:4) and then set then update the byte 0, bit 3:0 to "Awaiting for recovery image" (0x1)
14. BMC will send the next image as requested in the image index, and Caliptra RT FW and I3C HW go through the same flow as above.
15. Open to discuss with FW team: Caliptra RT FW through SOC manifest can make a decision on booting all the FWs using the recovery flow.

BMC requirements

1. BMC should not send payload to recovery interface (/I3C target) device if RECOVERY_CTRL register has byte 2 indicating Image Activated. BMC must wait to clear the byte 2. (Recovery Interface is responsible for clearing this bye by writing 1).

2. BMC must send payload to I3C target device in chunks of 256 bytes (header (4B) + FW bytes(256B) as I3C target transfer) only unless it is the last write for the image. Before sending the payload, BMC must read FIFO empty status from INDIRECT_FIFO_STATUS register.
3. After last write for the image, BMC must activate the image after reading INDIRECT_FIFO_STATUS register, FIFO empty status.

Recovery Interface Wires [move to HW spec]

1. **Payload available**

   The Recovery Interface (I3C target) should receive a write transaction to INDIRECT_FIFO_DATA reg from BMC - 256B + 4B (Header), and wait for each I3C write to finish. Once I3C write transaction to INDIRECT_FIFO_DATA register is completed and PEC verification is successful, then the I3C target must assert "payload_available". DMA assist must wait for "payload_available" before reading. It must read 256B or last read with remaining data.

   The "payload_available" signal remains asserted until Recovery Interface receives Read from DMA over AXI for INDIRECT_FIFO_DATA.

2. **Image_activated**

   The I3C target will assert "image_activated" signal as soon as write to RECOVERY_CTRL register is received.

   ROM will clear this bit by writing to RECOVERY_CTRL register via DMA assist after the image is authenticated.

RI firmware requirements

Firmware must program DMA assist with correct image size (multiple of 4B) + FIXED Read + block size is 256B (burst / FIFO size). Firmware must wait for "image_activated" signal to assert before processing the image. Once the image is processed, firmware must initiate a write with data 1 via DMA to clear byte 2 of the RECOVERY_CTRL register. This will allow BMC to initiate subsequent image writes.

AXI Subordinate / I3C and AXI interactions

Received **transfer data** can be obtained by the driver via a read from XFER_DATA_PORT register. Received **data threshold** is indicated to BMC by the controller with TX_THLD_STAT interrupt if RX_THLD_STAT_EN is set. The RX threshold can be set via RX_BUF_THLD. In case of a read when **no RX data** is available, the controller shall raise an error on the frontend bus interface (AHB / AXI).

Caliptra AXI Manager & DMA assist Overview

SOC_IFC includes a hardware-assist block that is capable of initiating DMA transfers to the attached SoC AXI interconnect. The DMA transfers include several modes of operation, including raw transfer between SoC addresses, moving data into or out of the SOC_IFC mailbox, and directly driving data into the SHA acceleration engine. One additional operating mode allows the DMA engine to autonomously wait for data availability via the OCP Recovery interface (which will be slowly populated via an I3C or similar interface). When arming the engine, Caliptra firmware is expected to have information about available AXI addresses that are legal for DMA usage. The DMA can facilitate transfer requests with 64-bit addresses. DMA AXI Manager logic automatically breaks larger transfer sizes into multiple valid AXI burst transfers (max 4KiB size), and legally traverses 4KiB address boundaries. The DMA AXI Manager supports a maximum transfer size of 1MiB in total. FIXED AXI bursts are allowed, to facilitate a FIFO-like operation on either read or write channels.

Descriptor

The following table illustrates registers implemented for DMA control.

Register	Description
ID	Identifies DMA engine uniquely
Capabilities	Reports DMA configuration/capabilities
Control	0: GO 1: Flush (abort operation safely and purge FIFO. Zeroize all entries?) 15:2 RESERVED 17:16: Read Route 00: Read path disabled 01: AXI RD -> Mailbox 10: AXI RD -> AHB reads to consume 11: AXI RD -> AXI WR 19:18: RESERVED Route bits 20: Read addr fixed

	23:21 RESERVED 25:24 Write Route 00=Write path disabled 01=Mailbox -> AXI WR 10=AHB writes to fill -> AXI WR 11=AXI RD -> AXI WR 27:26 RESERVED Route bits 28: Write addr fixed 31:29 RESERVED
Status0	0: Busy (0 = ready to accept transfer request, 1 = operation in progress) 1: Error 15:2: RESERVED N:16: Control FSM State 31:N: RESERVED
Status1	31:0: Byte count remaining to destination
Source Addr L	31:0: Addr[31:0]. Must be aligned to AXI data width.
Source Addr H	N:0: Addr[AW-1:32] 31:N: RESERVED
Dest Addr L	31:0: Addr[31:0]. Must be aligned to AXI data width.
Dest Addr H	N:0: Addr[AW-1:32] 31:N: RESERVED
Byte Count	31:0: Byte count to send. Must be a multiple of AXI data width. Maximum allowed value is 0x10_0000 (decimal 1048576) which equates to a 1MiB transfer.
Block Size	11:0: Byte size of individual blocks to send as part of the total Byte Count. This register indicates what granularity of AXI transactions are issued at a time. When non-zero, this field instructs the DMA to wait for the input "WIRE" to pulse high before issuing each transaction. Total burst is done once "Byte_count/block_size" transactions have completed. When zero, DMA issues AXI transactions of maximum size without any stalls in between transactions. Must be a multiple of AXI data width. If block size is not aligned to the AXI data width, it will be rounded down. Value of 4096 or larger is unsupported – AXI native maximum size is 4096. 31:12: RESERVED
WR Data	WO. Data word push into WR FIFO from uC/AHB to send via AXI WR.
RD Data	RO. Data word output from RD FIFO after AXI RD.
Intr Block (xN…)	Status interrupts: - TXN done

	- RD FIFO Not Empty - RD FIFO Full - WR FIFO Not Full - WR FIFO Empty Error interrupts: - Command Error - AXI RD error - AXI WR error - Mailbox not locked error - RD FIFO Overflow - RD FIFO Underflow - WR FIFO Overflow - WR FIFO Underflow

Programming Flowchart

General Rules:

1. If either Read or Write route is configured to AXI RD -> AXI WR, both routes must be configured as AXI RD -> AXI WR.
2. Read Route and Write Route must not both be disabled.
3. If Read Route is enabled to any configuration other than AXI RD-> AXI WR, Write route must be disabled.
4. If Read Route is disabled, Write route must be enabled to a configuration that is not AXI RD -> AXI WR.
5. If Read Route is disabled, Read Fixed field is ignored.
6. If Write Route is disabled, Write Fixed field is ignored.
7. Addresses and Byte Counts must be aligned to AXI data width (1 DWORD).

Steps:

1. Write Byte Count
2. Write Block Size
3. Write Source Addr (value is ignored if data is from AHB)
4. Write Dest Addr (value is ignored if data is to AHB).
    a. To perform an accelerated SHA operation on incoming read data, firmware sets the Read/Write route to AXI RD-> AXI WR, and the destination address to the SoC address for the SHA Acceleration data in aperture.
5. Set Interrupt Enables (optional)
6. If Mailbox R/W: Acquire Mailbox Lock

7. If SHA Acc Op:
    a. First acquire Sha Accel Lock via AXI by using this flow (with the AHB-> AXI WR route) to initiate AXI manager action
    b. Initiate Sha Accel streaming operation via AXI by using this flow (with the AHB-> AXI WR route) to initiate AXI manager action
    c. Run this operation with the AXI RD -> AXI WR route to move data from SoC location into Sha Accelerator
8. Set Control Register
    a. Set Read/Write Routes
    b. Set Read/Write Fixed=0/1
    c. GO
    d. (All 3 may be single write or separate, GO must be last bit to set)
9. If AHB data: Wait for RD FIFO not empty or WR FIFO not full
    a. Push/Pop data (using Rd Data/Wr Data register offsets) until all requested bytes transferred
    b. If AHB Error – check status0 for Error, then check for "Command Error"
10. Wait for TXN Done Interrupt (or Poll Status0)
11. Read Status0, confirm Busy=0, Error=0

Diagram

Fuse Controller

Caliptra subsystem Fuse Controller is based on OpenTitan OTP (One-Time Programmable) Controller.

This module provides the device with one-time programming functionality, resulting in non-volatile programming that cannot be reversed (unlike flash). This functionality is delivered via an open-source Fuse Controller and a proprietary Fuse/OTP IP.

The fuse controller provides the following functionality:

a. Open –Source Abstraction Interface:
    a. Provides an open-source interface for software to communicate with the underlying Fuse macro.
    b. Enables hardware components (e.g., life cycle controllers and key managers) to interact with the Fuse macro.
b. Logical Security Protection:
    a. Implements integrity checks and content scrambling during OTP operations.

    b. Ensures software isolation when OTP contents are readable and programmable.

c. Proprietary OTP IP integration:

    a. Collaborates with the proprietary OTP IP to enhance functionality.

    b. Key features of the OTP IP include:

        i. Reliable, Non-Volatile Storage: Ensures data persistence.

        ii. Redundancy and Error-Correction Mechanisms: Enhances data integrity.

        iii. Physical Defensive Features:

            1. Side-Channel Attack (SCA) Resistance: Guards against information leakage.

            2. Fault Injection (FI) Resistance: Protects against deliberate attacks.

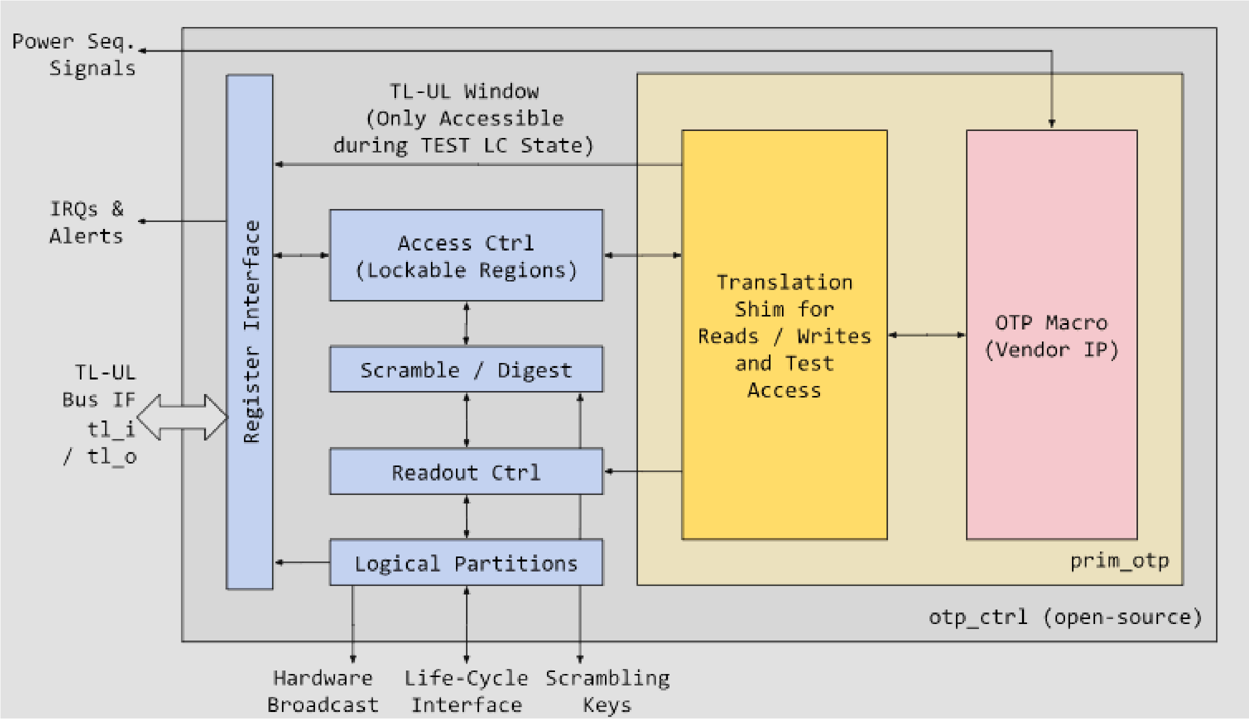
            3. Visual and Electrical Probing Resistance: Prevents unauthorized access.

Features

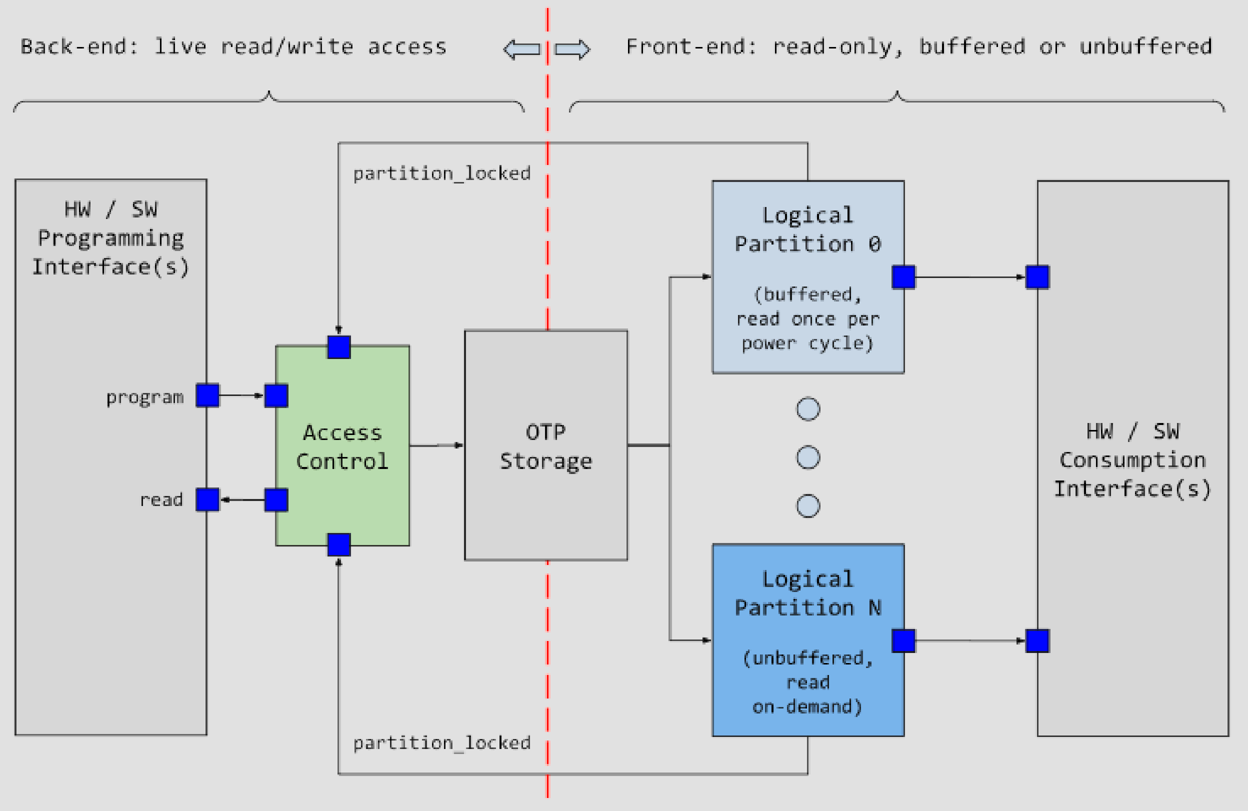
- Multiple logical partitions of the underlying OTP IP
    - Each partition is lockable, and integrity checked
    - Integrity digests are stored alongside each logical bank
- Periodic / persistent checks of OTP values
    - Periodic checks of shadowed content vs digests
    - Periodic checks of OTP stored content and shadowed content
    - Persistent checks for immediate errors
- Separate life cycle partition and interface to life cycle controller
    - Supports life cycle functions, but cannot be integrity locked
- Lightweight scrambling of secret OTP partition using a global netlist constant
- Lightweight ephemeral key derivation function for RAM scrambling mechanisms
- Lightweight key derivation function for FLASH scrambling mechanism

Overview

Below is a high-level block diagram depicting the OTP functionality split into an open-sourced and closed-source part.



Conceptually, OTP functionality can be split into “front-end” and “back-end.” The “front-end” contains logical partitions that feed the hardware and software consumer interfaces of the system. The “back-end” represents the programming interface used by hardware and software components to stage the upcoming values. The block diagram below illustrates this behavior model.


Back-end: live read/write access
Front-end: read-only, buffered or unbuffered
HW / SW Programming Interface(s)
program
read
partition_locked
Access Control
OTP Storage
Logical Partition 0
(buffered, read once per power cycle)
Logical Partition N
(unbuffered, read on-demand)
partition_locked
HW / SW Consumption Interface(s)

Logical Partitions

The OTP is logically separated into partitions that represent distinct functions. This means the isolation is virtual and maintained by the OTP controller instead of the underlying OTP IP.

Within each logical partition, there are specific enforceable properties

- Confidentiality via secret partitions
  - This controls whether a particular partition contains secret data.
  - If secret, a partition is not readable by software once locked, and is scrambled in storage.
- Read lockability
  - This controls whether a particular partition disables software readability for later stage software.
  - Some partitions can be locked statically (by computing and storing an associated digest in OTP), others can be read locked at runtime via CSRs.
- Write lockability
  - This controls whether a partition is locked and prevented from future updates.

    - A locked partition is stored alongside a digest to be used later for integrity verification.
- Integrity Verification
    - Once a partition is write-locked by calculating and writing a non-zero digest to it, it can undergo periodic verification (time-scale configurable by software). This verification takes two forms, partition integrity checks, and storage consistency checks.

Since the OTP is memory-like in nature (it only outputs a certain number of bits per address location), some of the logical partitions are buffered in registers for instantaneous and parallel access by hardware.

Partition Details

The Fuse Controller adapted for Caliptra contains five logical partitions.

Partition	Scrambled	ECC Integrity	Buffered	Write Lockable	Read Lockable	Description
VENDOR_TEST	No	No	No	Yes (SW Digest)	Yes (CSR)	Vendor smoke tests during manufacturing
CREATORSW_CFG	No	Yes	No	Yes (SW DIgest)	Yes (CSR)	Software config
SECRET0	Yes	Yes	Yes	Yes (HW Digest)	Yes (HW Digest)	Obfuscated UDS seed
SECRET1	Yes	Yes	Yes	Yes (HW Digest)	Yes (HW Digest)	Obfuscated field entropy
LIFE_CYCLE	No*	Yes	Yes	No	No	Life cycle state

Vendor Test Partition

The vendor test partition is used for OTP programming smoke checks during the manufacturing flow. This partition behaves like any other SW partition, with the exception that ECC uncorrectable errors will not lead to fatal errors/alerts as they do in all other partitions.

Partition	Security Property	Scrambled	ECC Integrity	Buffered	WR Lockable	Read Lockable	Fuse Name	Width	Description
VENDOR_TEST		No	no	No	Yes (SW Digest)	Yes (CSR)			For vendor smoke checks during manufacturing

Creator Software Configuration Partition

The creator software configuration partition is used as non-volatile storage for hashes, device ID, Soc stepping ID, FMC version number etc. The contents of this partition are consumed once as part of code execution or moved to another storage compartment elsewhere in the design. Software is responsible for calculating the integrity digest and programing it into the OTP.

The creator software configuration partition includes the following fuses.

Partition	Security Property	Scrambled	ECC Integrity	Buffered	WR Lockable	Read Lockable	Fuse Name	Width	Description
CREATOR_SW_CFG		No	No	No	Yes (SW Digest)	Yes (CSR)			For vendor smoke checks during manufacturing
	Integrity	No	Yes	No	Yes (SW Digest)	Yes (CSR)	FMC Key Manifest SVN	32	FMC Security Version Number
	Integrity	No	Yes	No	Yes (SW Digest)	Yes (CSR)	Runtime SVN	128	

	Integrity	No	Yes	No	Yes (SW Digest)	Yes (CSR)	Fuse_LMS_Verify	32	0: Verify Caliptra firmware images with ECDSA-1 only 1: Verify Caliptra firmware images with both ECDSA and LMS
	Integrity	No	Yes	No	Yes (SW Digest)	Yes (CSR)	Fuse_LMS_Revocation	32	Bits for revoking LMS public keys in the key manifest
	Integrity	No	Yes	No	Yes (SW Digest)	Yes (CSR)	Fuse_key_manifest_pk_hash[11:0]	384	Key manifest PK Hash Fuse
	Integrity	No	Yes	No	Yes (SW Digest)	Yes (CSR)	Fuse_key_manifest_pk_hash_mask	32	Key Manifest Mask Fuse
	Integrity	No	Yes	No	Yes (SW Digest)	Yes (CSR)	Fuse_owner_pk_hash[11:0]	384	Owner PK Hash Fuse
	Integrity	No	Yes	No	Yes (HW Digest)	No	Fuse_idevid_cert_attr[23:0]	768	Manufacturer IEE IDEVID Certificate Gen Attributes
	Integrity	No	Yes	No	Yes (HW Digest)	No	Fuse_idevid_manuf_hsm_id[3:0]	128	Manufacturer IDEVID Manufacturer's HSM identifier (this is used to find the certificate

									chain from the boot media)
	Integrity	No	Yes	No	Yes (HW Digest)	No	Fuse_soc_stepping_id	32	SoC stepping ID

Secret Partitions

Secret partitions contain data that is critical to security, such UDS seed and field entropy. These values are stored scrambled in OTP and are descrambled upon read. PRESENT cipher is used for scrambling and descrambling operations as it is a proven lightweight block cipher and lends well to iterative decomposition.

The contents of secret partitions are not readable by software once locked (other than the digest which must always be readable).

Secret partitions use a global netlist key for the scrambling operation as there is no other non-volatile storage to store a unique key.

Caliptra fuse controller includes two secret partitions.

Partition	Security Property	Scrambled	ECC Integrity	Buffered	WR Lockable	Read Lockable	Fuse Name	Width	Description
SECRET0	Confidentiality & Integrity	Yes	Yes	Yes	Yes (HW Digest)	Yes (HW Digest)	UDS Seed	384	Obfuscated UDS Seed
SECRET1	Confidentiality & Integrity	Yes	Yes	Yes	Yes (HW Digest)	Yes (HW Digest)	Field Entropy	256	Obfuscated Field Entropy

Life Cycle Partition

The life cycle partition is active throughout all stages and hence is the ONLY partition that cannot be locked. After the device finishes provisioning and goes into production, it must retain the ability to transition back to RMA in case of unexpected failures.

The life cycle state and counters must always be changeable to support this transition.

Partition	Security Property	Scrambled	ECC Integrity	Buffered	WR Lockable	Read Lockable	Fuse Name	Width	Description
LIFE_CYCLE		No	Yes	Yes	No	No			Life cycle state (Caliptra boot media integrated usage only)

Partition Checks

Integrity

The hardware integrity checker uses two methods to confirm the content of the volatile buffer registers after locking the relevant partitions:

All buffered partitions have extra ECC protection (8bit ECC for each 64bit block) that is continuously monitored.

The partition's digest is recalculated at variable intervals and matched with the digest stored with the partition.

This check does NOT aim to verify between the storage flops and the OTP, but to ensure that the buffer register contents are consistent with the computed digest. This verification checks if the storage flops have been affected by fault attacks. This check only applies to the HW_CFG* and SECRET* partitions. If an error is detected, the OTP controller will issue a fatal_check_error alert and reset all its hardware outputs to their defaults.

Storage Consistency

This verification makes sure the value in the buffer registers is the same as the value in the OTP. This process reads the OTP again at semi-random times and verifies that the value read matches the value stored. Note, since there are also integrity checks in parallel, some partitions do not need to check ALL read contents for consistency. If there is an integrity digest, only the digest needs to be read; otherwise, all values must be read.

This check applies to LIFE_CYCLE and SECRET* partitions. If a failure happens, the OTP controller will send out a fatal_check_error alert and reset all its hardware outputs to their defaults.

Note that checks on life cycle might fail if life cycle is updated, because life cycle is the only partition that can have live updates. The controller detects this condition based on the lc_check_byp_en_i signal from the life cycle controller and pauses background checks on this partition to avoid false positives.

Secret Partition Integrity Checks

The integrity digest is based on the scrambled form of the secret partitions, since that is how they are stored. To optimize the buffer registers usage, only the unscrambled form of the secret partitions is kept in buffer registers. Hardware re-scrambles the data before sending it to the digest to compute the digest.

Subsystem Fuse Manufacturing Flows

OT FUSE manufacturing flows

UDS Seed Programming

There are three ways of generating a UDS_SEED

1. Use the internal TRNG to directly generate a 384-bit random number.
2. Use an entity external to Caliptra such as an HSM or SOC-specific methodology to produce UDS-seed 384-bit random number that is pushed into the fuse controller (same as Caliptra 1.0).
3. Combine the internal TRNG output with a Manufacturing time provided value to produce a 384-bit output.

UDS Manufacturing – Mode A:

1. When SOC life cycle is in MANUFACTURING MODE, cp[2] is set to request for UDS seed programming flow.
2. Caliptra ROM will sample this bit on power up; when this bit is set and Caliptra ROM rechecks that the life cycle state is manufacturing mode, it reads the iTRNG for a 384-bit value.

3. Caliptra ROM writes the 384-bit value to the address available through a register UDS_SEED_OFFSET (HW/Integration requirement: Need to add this), using DMA HW assist macro available at ROM's disposal.

UDS Manufacturing – Mode B:

- SOC "injects" the UDS_SEED value directly through the fuse manufacturing interface. This is similar to Caliptra 1.0 flow where SOC owns this flow.

UDS Manufacturing – Mode C:

1. When SOC life cycle is in MANUFACTURING MODE, CPTRA_DBG_MANUF_SERVICE_REG[2] & CPTRA_DBG_MANUF_SERVICE_REG[2] are set to request for UDS seed programming flow.
2. Caliptra ROM will sample this bit on power up and also waits/consumes the FIPS compliant 384-bit entropy (the "message") provided at the manufacturing (TBD on the mechanism for this coming over JTAG to ROM);
3. When both the bits are set, ROM rechecks that the life cycle state is manufacturing mode, it reads the iTRNG for a 384-bit value (the "key").
4. Caliptra ROM performs an HMAC-SHA-384 using the message-key pair from steps 2 and 3 above appropriately padded for PRF usage per Caliptra 1.0.
5. Caliptra ROM writes the final 384-bit value to the address available through a register UDS_SEED_OFFSET (HW/Integration requirement: Need to add this), using DMA HW assist macro available at ROM's disposal.

Actual 'burning in' values in the OTP macro can be carried out by the same steps per fuse controller specification plus any SOC specific methodologies (eg. Fuse macro voltage elevation flows etc.).

The starting unprogrammed values of the SECRET partitions are assumed to be all 0's (all 1's or zeroized state). At the end of programming of the SECRET partition, the value "should" contain the unmodified UDS_SEED entry or be left in all 0's (all 1's) if the programming was unsuccessful unless a hardware error condition occurs.

Fuse manufacturing flow should be followed as defined by the spec once the secret partition is programmed. TBD: Check with Anjana if PK hash needs to be in its own partition.

In the rare event of a HW fault (electrical failure, bit flip) during the programming operation the programmed fuse value may be incorrect, and the integrity test of the SECRET0 partition will indicate an error which will be reflected when Caliptra ROM attempts to fetch the UDS_SEED. In that case, a locked non-zeroized state could permanently render the SoC device ‘bricked’.

NOTE. The LIFE_CYCLE partition of the device should be in MANUFACTURING before programming the SECRET partitions to ensure ROM code or firmware can take appropriate gating action if necessary.

FW load flow @ Manufacturing

There may be a need for subsystem FW to be loaded during manufacturing. Any FW that loads during Manufacturing flow must be authenticated before being used. This implies that the PK hashes required and as defined by architectural spec should already be programmed into the fuse controller. If there is no need for FW during manufacturing, then this can be skipped.

There are two modes supported for pushing firmware into the Caliptra subsystem:

a. A JTAG interface that is opened up in DEBUG and MANUFACTURING life cycle modes. Using this JTAG interface, the recovery interface registers can be populated.
b. I3C recovery interface which is already available as an alternate boot path.

Note: Both modes use the hardware recovery registers and follow the same OCP recovery spec definition.

Subsystem In-field Fuse Provisioning (IFP) Requirements & Flows

IFP Requirements

4. Any IFP fuse(s) that needs to be provisioned can come over in-band interface or out-of-band interface.
    a. In-band interface is SOC specific
    b. Out-of-band interface will use MCTP commands. TBD: Use PLDM or SOC specific MTCP commands
5. IFP payload is issued to MCU;
6. IFP payload must be signed and must be authenticated by Caliptra.
7. MCU will execute SOC specific fuse “burn” flow (which typically involves regulating an external/internal VRs or LDOs etc. and executing some checks)

IFP Flow

1. SOC gets the IFP payload packet in-band or out-of-band depending on the SOC specific architecture.

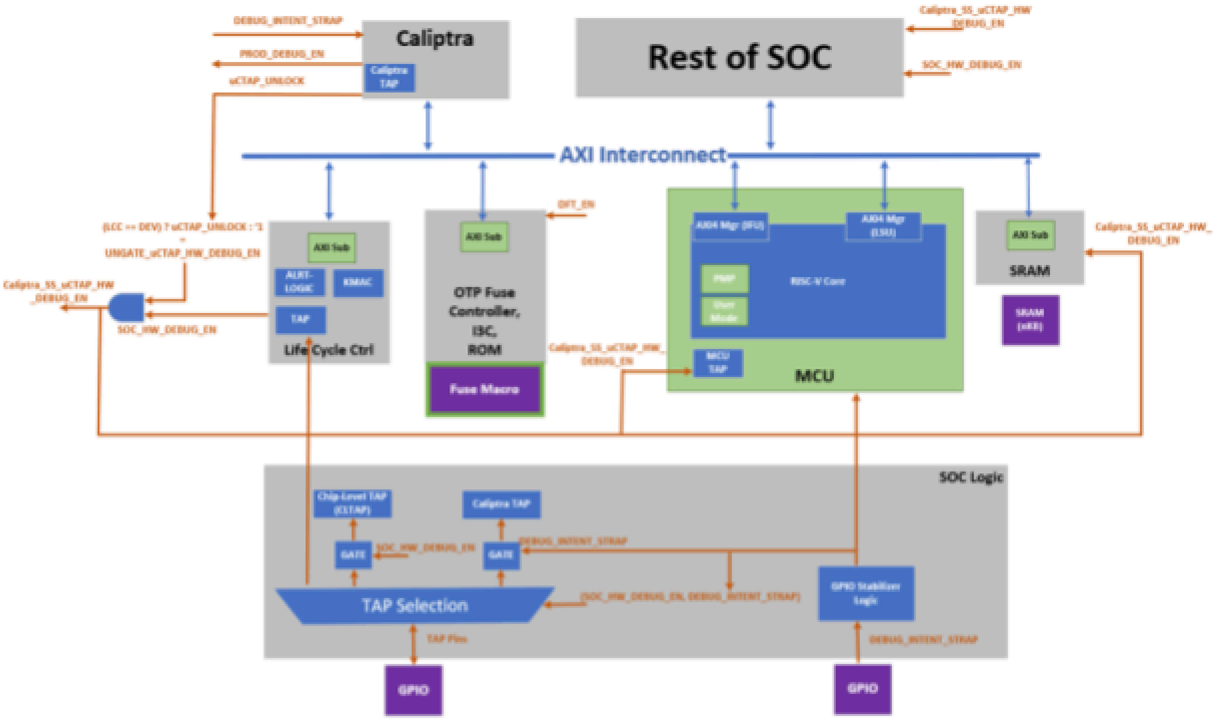
2. IFP payload is deposited to MCU mailbox
3. MCU will parse the payload and issues the payload for Caliptra to authenticate.
4. TBD by FW team: Caliptra will use the keys that came through as a part of SOC manifest to authenticate the IFP payload
5. If authentication passes, MCU will write the fuses to the fuse controller and execute SOC specific IFP burn flow.

Life Cycle Controller

It is an overview of the architecture of the Life-Cycle Controller (LCC) Module for its use in the Caliptra Subsystem. The LCC is responsible for managing the life-cycle states of the system, ensuring secure transitions between states, and enforcing security policies.

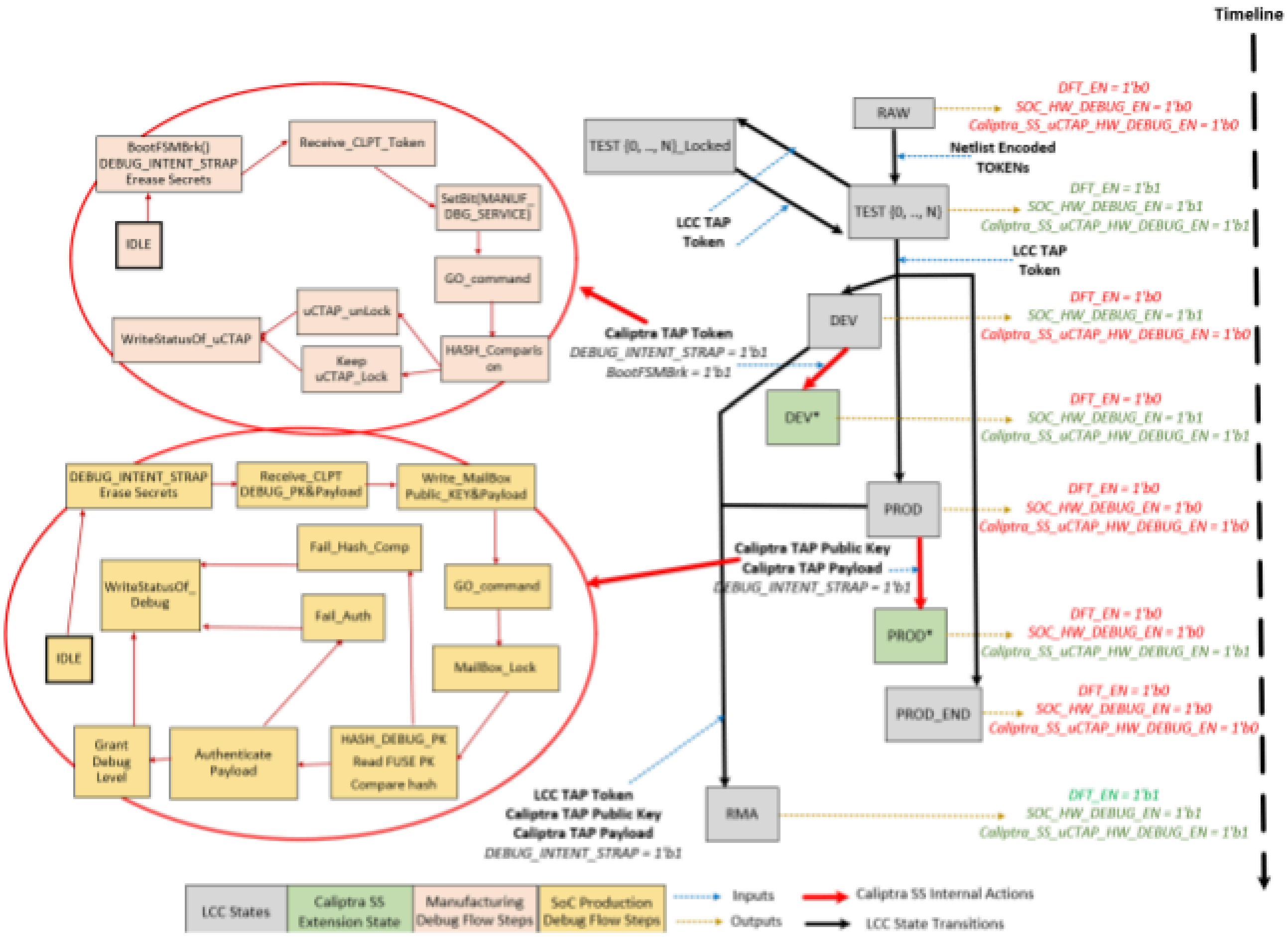
Caliptra Subsystem, SOC Debug Architecture

Figure below shows the Debug Architecture of the Caliptra Subsystem and some important high-level signals routed towards SOC. The table in *Key Components and Interfaces* section shows all the signals that are available to SOC (outside of Caliptra Subsystem usage).



SoC Debug Architecture of the Caliptra Subsystem with LCC; the red dashed circles highlight the newly added blocks and signals.

The figure below shows the LCC state transition and Caliptra Subsytem enhancement on LCC state transitions. It illustrates the life cycle flow of Caliptra Subsytem.



Caliptra Subsystem life cycle flow. This flow shows legal state transitions in life cycle controller by excluding its invalid states for simplicity.

Caliptra Subsystem LCC State Definitions

Name	Encoding	Description
RAW	OTP	This is the default state of the OTP. During this state, no functions other than transition to TEST_UNLOCKED0 are available. The token authorizing the transition from RAW to TEST_UNLOCKED0 is a value that is secret global to all devices. This is known as the RAW_UNLOCK token.

TEST_LOCKED{N}	OTP	TEST_LOCKED{N} states have identical functionality to RAW state and serve as a way for the Silicon Creator to protect devices in transit. It is not possible to provision OTP root secrets during this state. This is enforced by hardware and is implementation defined. To progress from a TEST_LOCKED state to another TEST_UNLOCKED state, a TEST_UNLOCK token is required.

TEST_UNLOCKED{N}	OTP	Transition from RAW state using OTP write. This state is used for **manufacturing and production testing.** During this state: ● uCTAPs (microcontroller TAPs) are enabled ● Debug functions are enabled ● DFT functions are enabled Note: during this state it is not possible to provision specific OTP root secrets. This will be enforced by hardware. It is expected that during TEST_UNLOCKED0 the TEST_UNLOCK and TEST_EXIT tokens will be provisioned into OTP. Once provisioned, these tokens are no longer readable by software.

PROD	OTP	Transition from TEST_UNLOCKED or TEST_LOCKED state via OTP writes. PROD is a mutually exclusive state to DEV and PROD_END. To enter this state, a TEST_EXIT token is required. This state is used both for provisioning and mission mode. During this state: ● uCTAPs (microcontroller TAPs) are disabled ● Debug functions are disabled ● DFT functions are disabled Caliptra Subsytem can grant SoC debug unlock flow if the conditions provided in “*SoC Debug Flow and Architecture for Production Mode”* section are satisfied. SoC debug unlock overwrites the signals and gives the following cases: ● uCTAPs (microcontroller TAPs) are enabled ● Debug functions are enabled based on the defined debug policy

		● DFT functions are disabled
PROD_END	OTP	This state is identical in functionality to PROD, except the device is never allowed to transition to RMA state. To enter this state, a TEST_EXIT token is required.
DEV	OTP	Transition from TEST_UNLOCKED state via OTP writes. This is a mutually exclusive state to PROD and PROD_END. To enter this state, a TEST_EXIT token is required. This state is used for developing provisioning and mission mode software. During this state ● uCTAPs (microcontroller TAPs) are enabled conditionally (uCTAP Unlock Token Routine section) ● Debug functions are enabled ● DFT functions are disabled

RMA	OTP	Transition from TEST_UNLOCKED / PROD / DEV via OTP write. It is not possible to reach this state from PROD_END. When transitioning from PROD or DEV, an RMA_UNLOCK token is required. When transitioning from TEST_UNLOCKED, no RMA_UNLOCK token is required. During this state ● uCTAPs (microcontroller TAPs) are enabled ● Debug functions are enabled ● DFT functions are enabled

SCRAP	OTP		Transition from any manufacturing state via OTP write. During SCRAP state the device is completely dead. All functions, including CPU execution are disabled. The only exception is the TAP of the life cycle controller which is always accessible so that the device state can be read out. No owner consent is required to transition to SCRAP. Note also, SCRAP is meant as an EOL manufacturing state. Transition to this state is always purposeful and persistent, it is NOT part of the device's native security countermeasure to transition to this state.
INVALID	OTP		Invalid is any combination of OTP values that do not fall in the categories above. It is the "default" state of life cycle when no other conditions match. Functionally, INVALID is identical to SCRAP in that no functions are allowed and no transitions are allowed. A user is not able to explicitly transition into INVALID (unlike SCRAP), instead, INVALID is

		meant to cover in-field corruptions, failures or active attacks.

DFT & DFD LC States

In addition to the decoding signals of the Life-Cycle Controller (LCC) proposed in the OpenTitan open-source silicon Root of Trust (RoT) project, we introduce a new signal: Caliptra_SS_uCTAP_HW_DEBUG_EN and renamed HW_DEBUG_EN as SOC_HW_DEBUG_EN. These signals are critical for managing the test and debug interfaces within the Caliptra Subsystem, as well as at the broader SoC level.

While this architecture document explains how the Caliptra Subsystem provides DFT and DFD mechanisms through the DFT_EN, SOC_HW_DEBUG_EN, and Caliptra_SS_uCTAP_HW_DEBUG_EN signals, it also offers greater flexibility by supporting various SoC debugging options through the broader SoC debug infrastructure. The architecture does not constrain the SoC's flexibility with the core security states of Caliptra and LCC states.

We establish a set of clear guidelines for how the Caliptra Subsystem transitions through the unprovisioned, manufacturing, and production phases, ensuring security. However, this architecture remains flexible, offering multiple levels of debugging options in production debug mode. Level 0 is designated for debugging the Caliptra Subsystem itself, while higher levels can be interpreted and customized by the SoC designer to implement additional debugging features. For instance, if the SoC designer wishes to include extra DFT and DFD capabilities, they can utilize one of the debug levels provided during production debug mode and expand its functionality accordingly. For more details, see "*SoC Debug Flow and Architecture for Production Mode"* Section and *"Masking Logic for Debugging Features in Production Debug Mode"* Section.

The DFT_EN signal is designed to control the scan capabilities of both the Caliptra Subsystem and the SoC. When DFT_EN is set to HIGH, it enables the scan chains and other Design for Testability (DFT) features across the system, allowing for thorough testing of the chip. This signal is already provided by the existing LCC module, ensuring compatibility with current test structures. However, the SoC integrator has the flexibility to assign additional functionality to one of the debugging options provided by the debug level signals during production debug mode. For example, at the SoC level, an integrator may choose to use one of these levels to enable broader SoC DFT features, allowing for system-level testing while maintaining Caliptra's protection. Additionally, SoC can override DFT_EN and set it to HIGH by using the infrastructure defined in "*Masking Logic for Debugging Features in Production Debug Mode"* Section.

The SOC_HW_DEBUG_EN signal is a new addition that governs the availability of the Chip-level TAP (CLTAP). CLTAP provides a hardware debug interface at the SoC level, and it is accessible when SOC_HW_DEBUG_EN is set to HIGH. For further details on how this signal integrates with the overall system, refer to the "*TAP Pin Muxing"* Section of this document.

In the manufacturing phase, the Caliptra Subsystem asserts SOC_HW_DEBUG_EN high, with the signal being controlled by the LCC. In PROD mode, this signal is low. However, the SoC integrator has the flexibility to enable CLTAP during production debug mode by incorporating additional logic, such as an OR gate, to override the SOC_HW_DEBUG_EN signal, like DFT_EN. This architecture provides a mask register that allows SoC to program/configure this overriding mechanism at integration time or using MCU ROM. This allows the SoC to maintain control over hardware debug access while preserving the intended security protections in production.

Finally, the Caliptra_SS_uCTAP_HW_DEBUG_EN signal is introduced to manage the microcontroller TAPs (uCTAPs) within the Caliptra subsystem. Although DFT_EN and SOC_HW_DEBUG_EN are directly controlled by LCC, Caliptra_SS_uCTAP_HW_DEBUG_EN is controlled by two conditions set by LCC and Caliptra. This document provides more details about these two conditions in uCTAP Unlock Token Routine. These TAPs are open before the development phase has been entered; and after that will be accessible during the LCC's DEV and PROD states, only if the token authentication is successful. The following table shows DFT_EN, SOC_HW_DEBUG_EN, and Caliptra_SS_uCTAP_HW_DEBUG_EN positions based on the LCC's states.

***Table**: LCC State and State Decoder Output Ports.*

LCC State\Decoder Output	**DFT_EN**	**SOC_HW_DEBUG_EN**	**Caliptra_SS_uCTAP_HW_DEBUG_EN**
RAW	Low	Low	Low

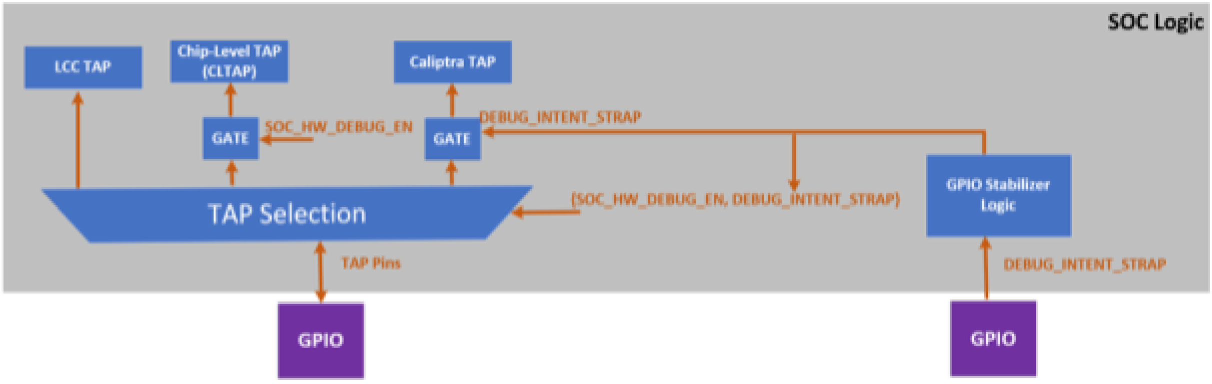
TEST_LOCKED	Low	Low	Low
TEST_UNLOCKED	High	High	High
DEV*	Low	High	TOKEN - CONDITIONED**
PROD*	TOKEN - CONDITIONED**	TOKEN - CONDITIONED**	TOKEN - CONDITIONED**
PROD_END	Low	Low	Low
RMA	High	High	High
SCRAP	Low	Low	Low
INVALID	Low	Low	Low
POST_TRANSITION	Low	Low	Low

*: Caliptra can enter debug mode and update these signals even though LCC is in DEV or PROD states. This case is explained in *“How does Caliptra enable uCTAP_UNLOCK?”* and *“SoC Debug Flow and Architecture for Production Mode”*.

**: Caliptra_SS_uCTAP_HW_DEBUG_EN can be high if Caliptra SS grants debug mode (either manufacturing or production). This case is explained in “How does Caliptra enable uCTAP_UNLOCK?” and “SoC Debug Flow and Architecture for Production Mode”. SOC_HW_DEBUG_EN and DEF_EN can be also set high to open CLTAP and enable DFT by SoC design support. However, this condition also needs to go through the flow described in “SoC Debug Flow and Architecture for Production Mode”.

TAP Pin Muxing

The LCC includes a TAP interface, which operates on its own dedicated clock and is used for injecting tokens into the LCC. Notably, the LCC TAP interface remains accessible in all life cycle states, providing a consistent entry point for test and debug operations. This TAP interface can be driven by either the TAP GPIO pins or internal chip-level wires, depending on the system's current configuration.



TAP pin muxing block diagram with a conceptual representation.

SOC logic incorporates the TAP pin muxing to provide the integration support and manage the connection between the TAP GPIO pins and the Chip-Level TAP (CLTAP). As illustrated in figure above, this muxing logic determines the source that drives the LCC TAP interface. The selection between these two sources is controlled by the SOC_HW_DEBUG_EN signal. When SOC_HW_DEBUG_EN is set to high, control is handed over to the CLTAP, allowing for chip-level debug access through the TAP GPIO pins. Conversely, when SOC_HW_DEBUG_EN is low, the TAP GPIO pins take control, enabling external access to the LCC TAP interface.

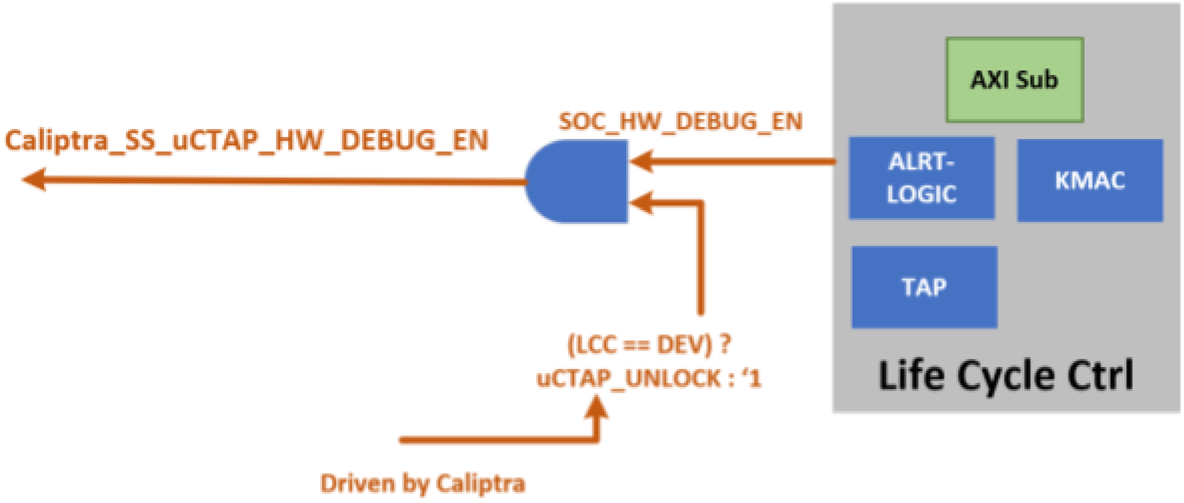
LCC State and State Decoder Output Ports Table outlines the specific LCC states that enable the SOC_HW_DEBUG_EN signal. These states include TEST_UNLOCK, DEV, PRDO, and RMA. In these states, the LCC allows internal chip-level debug access via CLTAP, thereby facilitating advanced debugging capabilities. This muxing approach ensures that the TAP interface is appropriately secured, and that access is granted only under specific conditions, aligning with the overall security and functional requirements of the Caliptra Subsystem.

TAP pin muxing also enables routing to Caliptra TAP. This selection happens when DEBUG_INTENT_STRAP is high. This selection is done through the GPIO and indicates that Caliptra will enter debug mode if the secret tokens are provided. Caliptra Subsystem has two debug modes: manufacturing debug and production debug. Entering these debug flows are explained in the following sections: *“How does Caliptra* *enable uCTAP_UNLOCK?", "SoC Debug Flow and Architecture for Production Mode"* and *"Masking Logic for Debugging Features in Production Debug Mode"*.

Note: The Caliptra TAP can run exact flow with AXI, targeting the mailbox and SOC provides that interface to platform.

uCTAP Unlock Token Routine for Manufacturing Phase

The figure given below illustrates the logic used to control the Caliptra_SS_uCTAP_HW_DEBUG_EN signal, which is a critical part of enabling the microcontroller TAP (uCTAP) debugging interface in the Caliptra Subsystem. This diagram shows an illustration of how uCTAP opens. However, the functionality of this logic will be implemented in MCI. The Caliptra_SS_uCTAP_HW_DEBUG_EN signal is governed by two primary inputs: SOC_HW_DEBUG_EN and a control signal driven by Caliptra. This signal opens uCTAP for MCU and Caliptra.



Domesticated LCC having additional logic to accommodate Caliptra

To assert Caliptra_SS_uCTAP_HW_DEBUG_EN high, the SOC_HW_DEBUG_EN signal must always be high, except SOC production debug mode case (see "SoC Debug Flow and Architecture for Production Mode" Section). This ensures that the uCTAP interface is only enabled when the system is in a secure and authorized state. According to the figure, SOC_HW_DEBUG_EN is derived from the LCC states. Specifically, SOC_HW_DEBUG_EN is high when the LCC is in the TEST_UNLOCKED, DEV, or RMA states. These are the only states where hardware debugging via the uCTAP interface is permitted. However, SOC_HW_DEBUG_EN can be set high by SoC during the production debug mode (see "*Masking Logic for Debugging Features in Production Debug Mode"*).

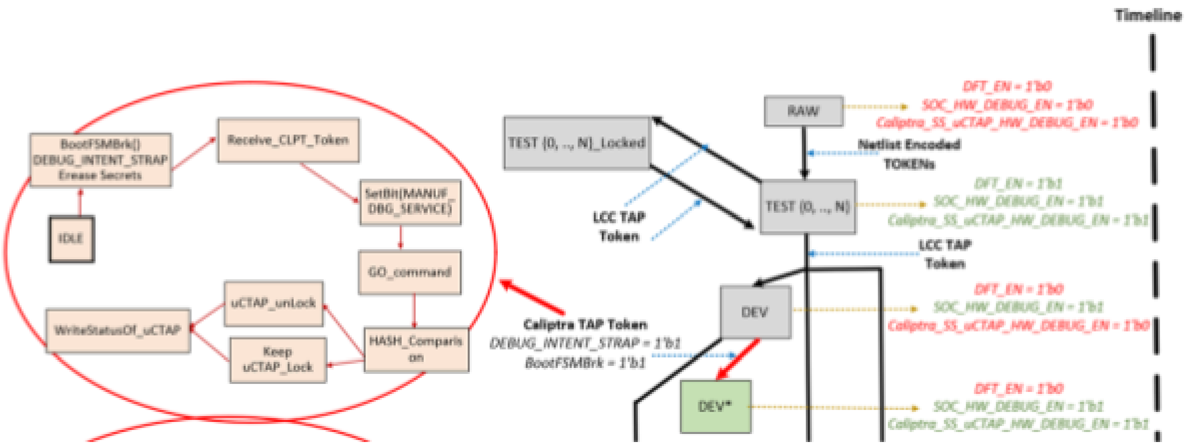
When the LCC is in the DEV state, an additional condition is required for Caliptra_SS_uCTAP_HW_DEBUG_EN to be asserted high. In this case, Caliptra must actively enable the signal, which is indicated by the uCTAP_UNLOCK control line. This allows for controlled and conditional access to the uCTAP interface during the development phase, ensuring that debugging is only allowed when it is explicitly permitted by the system's security policies.

The AND gate in the figure symbolizes this logic, showing that Caliptra_SS_uCTAP_HW_DEBUG_EN can only be high if both SOC_HW_DEBUG_EN is high, and the specific conditions driven by Caliptra are met. This design ensures that the uCTAP interface is securely controlled, limiting access to authorized lifecycle states and further protected by additional checks during the development phase.

Note: This architecture provides an extension on LCC state with SoC production debug logic. This logic is configured to provide an additional layer to unlock debugging in the production phase. This logic can override Caliptra_SS_uCTAP_HW_DEBUG_EN and sets this signal high (see "SoC Debug Flow and Architecture for Production Mode" Section and "Masking Logic for Debugging Features in Production Debug Mode").

How does Caliptra enable uCTAP_UNLOCK for manufacturing debug mode?

The following figure illustrates how Caliptra Subsystem enters the manufacturing debug mode. To enter this mode, LCC should be in DEV state. While being in manufacturing debug mode, LCC does not change its state from DEV to any other state. During the DEV state, Caliptra Subsystem can enable manufacturing debug flow by following steps:



The flow diagram on the right side shows the LCC states (grey boxes) and their transitions, while the flow diagram on the left illustrates Caliptra SS's enhancements to the LCC for the manufacturing phase. Specifically, the flow on the left depicts the routine for entering manufacturing debug mode.

Flow Explanation:

Assert BootFSMBrk: Temporarily halting the boot process to allow for secure operations like token injection and verification.

Assert DEBUG_INTENT_STRAP: If Caliptra samples this pin as high and sees that LCC is in DEV mode, it prepares itself for entering debug mode. Once the DEBUG_INTENT_STRAP is detected, Caliptra immediately wipes out its secret assets, including the Unique Device Secret (UDS) and Field entropy, ensuring that no sensitive data remains accessible. If this Pin is not high, Caliptra continues always in non-debug mode without taking any further action listed with the following states.

Inject Token via TAP/Register Write: Enabling the injection of a token through TAP and writing to a specific register. The injected token is combined with 256-bit nonce value that is updated for each boot session. That is used to prevent replay attacks.

Write to MANUF_DBG_SERVICE: The system intends to be in debug mode for manufacturing services.

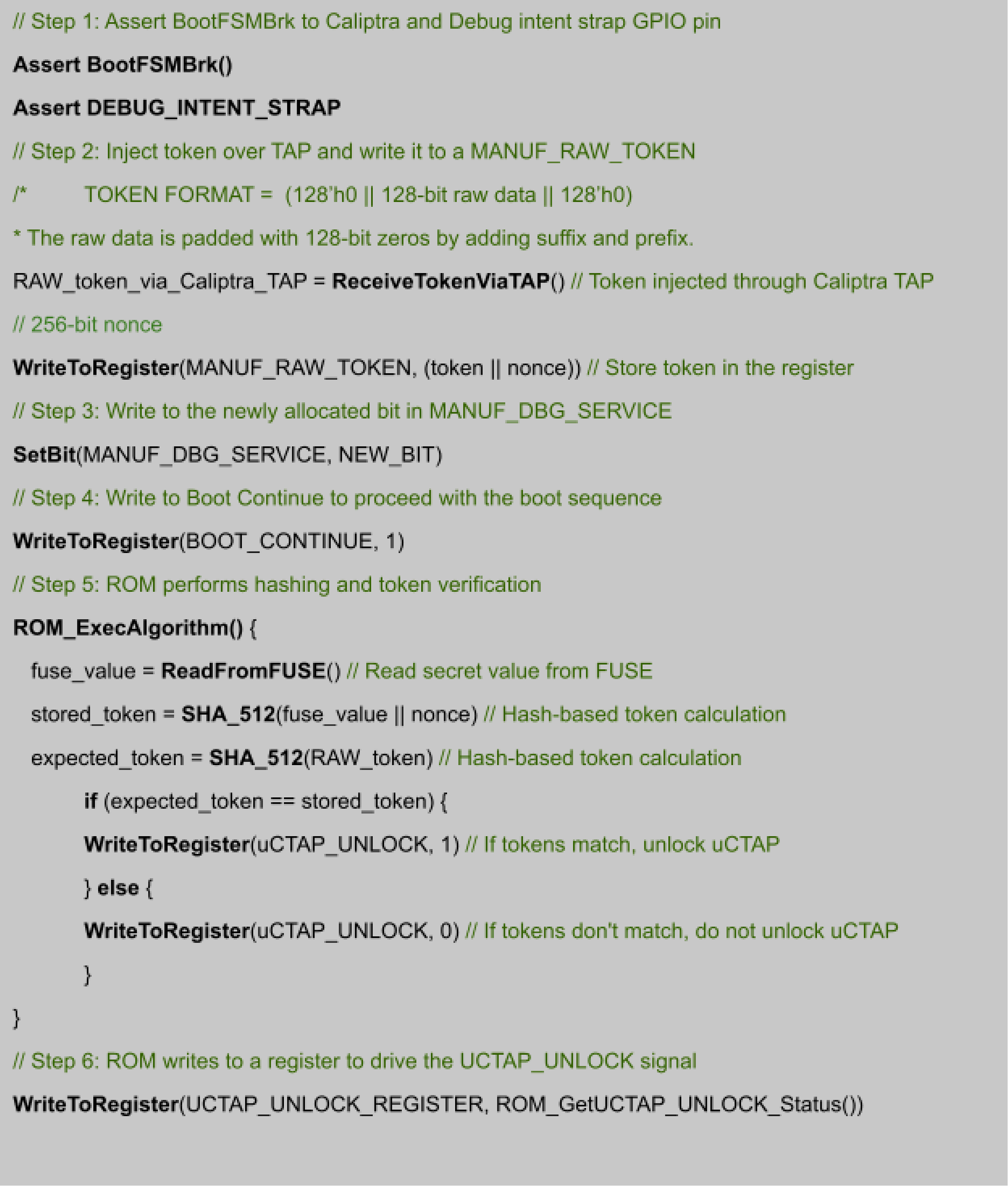
Write to Boot Continue: Resuming the boot process after halting it with BootFSMBrk.

ROM Executes Hash Hashing: Calculates the authentication value of the injected token using a hash-core which is a cryptographic authentication technique that uses a hash function and a secret key.

Token Comparison: A constant-time comparison between the authentication values of injected token and the FUSE content.

ROM Drives UCTAP_UNLOCK Signal: The ROM writes to a register that controls the UCTAP_UNLOCK signal based on the result of the token verification.

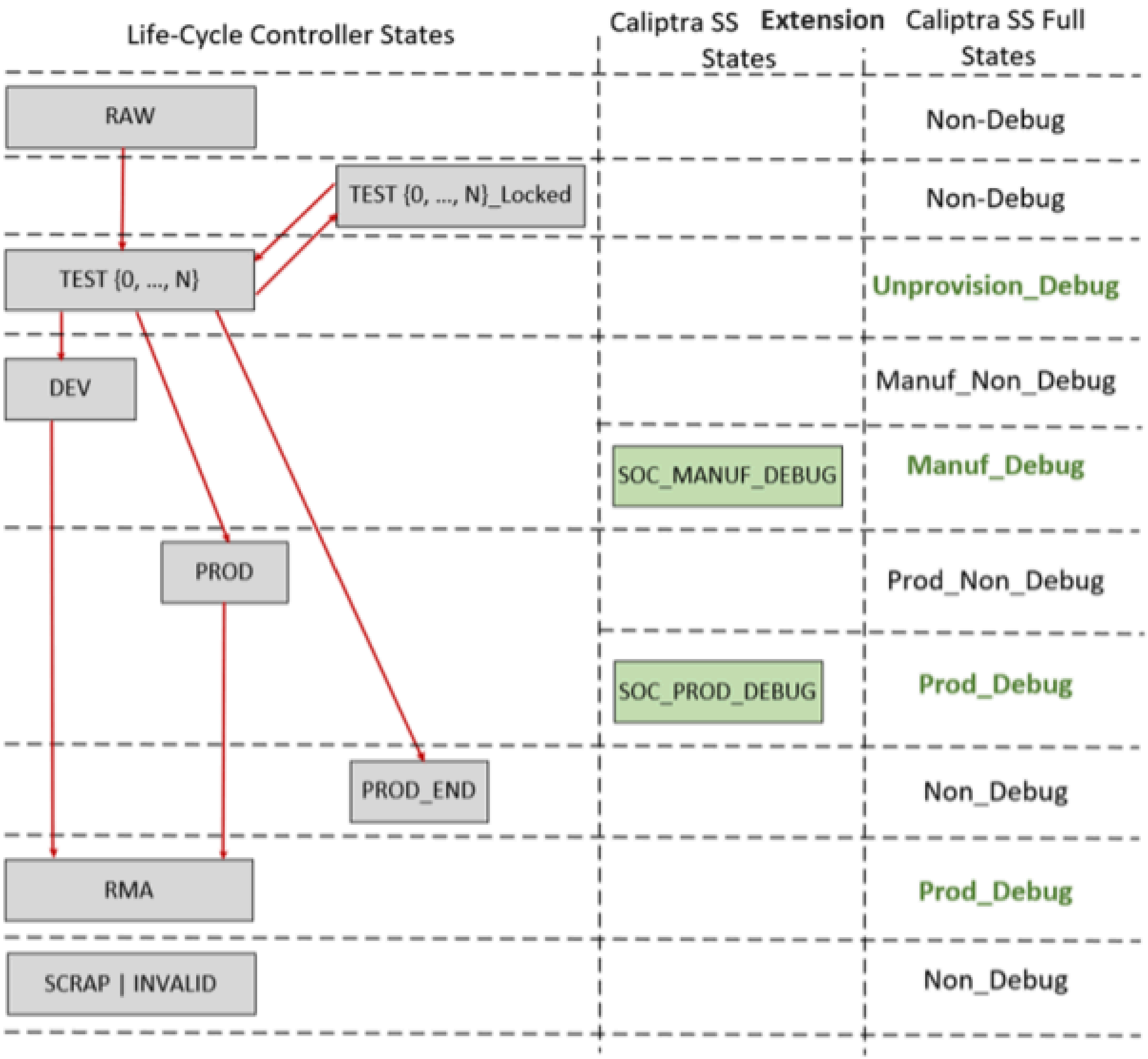
Note: The manufacturing debug flow relies on the hash comparison hardness rather than an authentication check provided with an asymmetric cryptography scheme. However, the following sections show that production debug flow relies on asymmetric cryptography scheme hardness. The reason behind this choice is that the manufacturing phase is the early phase of the delivery, and this phase is entered in an authorized entity's facility. Therefore, this architecture keeps the manufacturing phase simpler.



```
// Step 1: Assert BootFSMBrk to Caliptra and Debug intent strap GPIO pin
Assert BootFSMBrk()
Assert DEBUG_INTENT_STRAP
// Step 2: Inject token over TAP and write it to a MANUF_RAW_TOKEN
/*      TOKEN FORMAT =  (128'h0 || 128-bit raw data || 128'h0)
* The raw data is padded with 128-bit zeros by adding suffix and prefix.
RAW_token_via_Caliptra_TAP = ReceiveTokenViaTAP() // Token injected through Caliptra TAP
// 256-bit nonce
WriteToRegister(MANUF_RAW_TOKEN, (token || nonce)) // Store token in the register
// Step 3: Write to the newly allocated bit in MANUF_DBG_SERVICE
SetBit(MANUF_DBG_SERVICE, NEW_BIT)
// Step 4: Write to Boot Continue to proceed with the boot sequence
WriteToRegister(BOOT_CONTINUE, 1)
// Step 5: ROM performs hashing and token verification
ROM_ExecAlgorithm() {
  fuse_value = ReadFromFUSE() // Read secret value from FUSE
  stored_token = SHA_512(fuse_value || nonce) // Hash-based token calculation
  expected_token = SHA_512(RAW_token) // Hash-based token calculation
        if (expected_token == stored_token) {
        WriteToRegister(uCTAP_UNLOCK, 1) // If tokens match, unlock uCTAP
        } else {
        WriteToRegister(uCTAP_UNLOCK, 0) // If tokens don't match, do not unlock uCTAP
        }
}
// Step 6: ROM writes to a register to drive the UCTAP_UNLOCK signal
WriteToRegister(UCTAP_UNLOCK_REGISTER, ROM_GetUCTAP_UNLOCK_Status())
```

SoC Debug Flow and Architecture for Production Mode

The Caliptra Subsystem includes SoC debugger logic that supports Caliptra's production debug mode. This debugger logic extends the capabilities of the Lifecycle Controller (LCC) by providing a production debug mode architecture that the LCC does not inherently support, except in the RMA state. This architecture manages the initiation and handling of the production debug mode separately from the LCC's lifecycle states. The following figure shows how Caliptra Subsystem extends LCC and enables debug unlocking with the new state definitions.



Caliptra Subsystem flow extension on LCC with new states

The process of enabling production debug mode begins when the DEBUG_INTENT_STRAP pin is asserted high via the SoC's GPIO. This pin signals Caliptra to start the debug mode when the LCC is in the PROD state. Even though the DEBUG_INTENT_STRAP can be set high at any time, Caliptra evaluates the request only during two distinct phases: Pre-ROM execution and Post-ROM execution.

In addition to DEBUG_INTENT_STRAP pin, there is also SoC-based DEBUG intent strap configuration, which has two values: DEBUG_AUTH_PK_HASH_REG_BANK_OFFSET and NUM_OF_ DEBUG_AUTH_PK_HASHES. The value DEBUG_AUTH_PK_HASH_REG_BANK_OFFSET represents an address offset, while NUM_OF_ DEBUG_AUTH_PK_HASHES defines how many registers are available for reading. These two values establish the debug policy depth, allowing flexibility beyond the earlier limit of N number public keys for different debugging levels. When the subsystem powers up, Caliptra hardware latches and locks the DEBUG_AUTH_PK_HASH_REG_BANK_OFFSET and NUM_OF_ DEBUG_AUTH_PK_HASHES values, ensuring that these cannot be modified later through firmware or any run-time activity. NUM_OF_ DEBUG_AUTH_PK_HASHES is needed to prevent out-of-bound access.

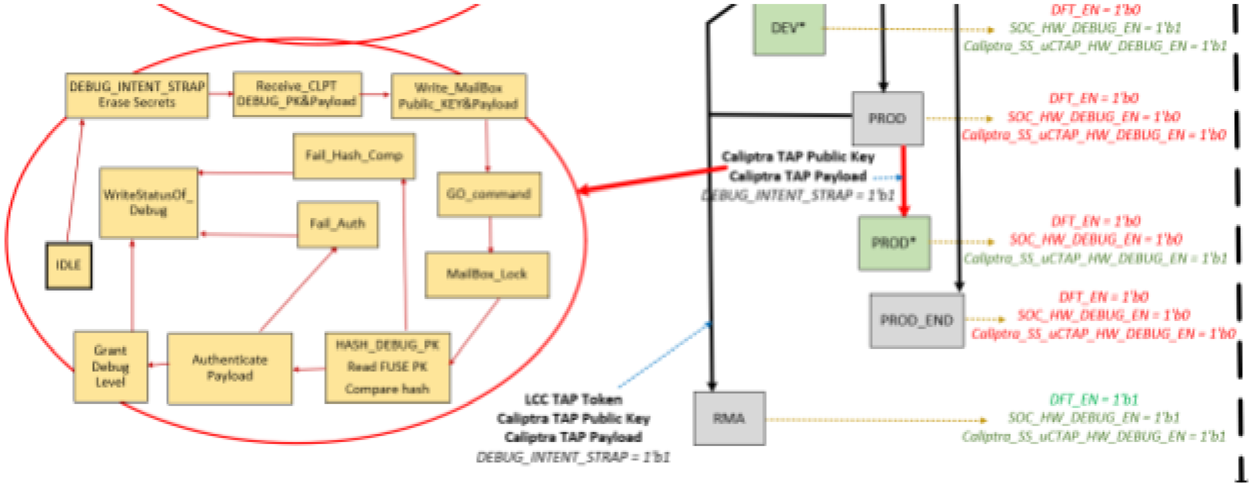
Once the DEBUG_INTENT_STRAP is detected, Caliptra immediately wipes out its secret assets, including the Unique Device Secret (UDS) and Field entropy, ensuring that no sensitive data remains accessible. After erasing the secret assets, Caliptra opens the TAP interface (in a safe mode to write to registers, not for active debug), which allows the debugger to interact with the system. The debugger verifies that the TAP interface is open by reading the TAP CSR (FIXME: to be documented once defined). The next step involves the debugger sending a Hybrid public key and a payload (employing both MLDSA and ECC cryptosystems) to Caliptra through the TAP interface. Caliptra receives these packages and writes them to its MailBox for further processing.

At this point, the debugger triggers the continuation of the boot sequence by setting the CPTRA_BOOTFSM_GO signal high through the TAP interface. This command signals Caliptra's BootFSM to proceed. If the debug mode request occurs during run-time (after ROM execution), the debugger sets the GO command through the MailBox instead. Upon receiving the GO command, Caliptra locks the data in the MailBox to ensure its integrity, while the debugger waits for Caliptra to evaluate the request.

When a public key corresponding to a specific debug level is provided—denoted by the number i—Caliptra reads the L+i(th) register after setting the R offset. This read operation retrieves the hash of the i(th) level PRE_DEBUG_PK from the appropriate register. Once Caliptra obtains this hash value, it compares it with the hash of the corresponding DEBUG_PK provided through the TAP interface. If the hash comparison is successful, Caliptra proceeds to authenticate the payload using the corresponding DEBUG_PKs.

If either the authentication or the hash comparison fails, Caliptra returns a failure status and updates the Reg_CLPT_to_MCU register in the MailBox to reflect the error. On the other hand, if both the hash comparison and the authentication are successful, Caliptra grants access to production debug mode by writing to the Reg_CLPT_to_MCU register, setting the PROD_DEBUG_EN bit. This action signals that the SoC is now in production debug mode and ready for further operations.

This flow establishes a secure and controlled process for entering Caliptra's production debug mode, ensuring that only authorized access is granted while maintaining the integrity and confidentiality of the system's sensitive assets. The more details about the flow sequence as illustrated with flow figure and explanation of each steps in the flow.



Steps:

1. **DEBUG_INTENT_STRAP Assertion**:
   - The process is initiated when the **DEBUG_INTENT_STRAP** pin, connected via the SoC's GPIO, is asserted high.
   - When this pin is high and the LCC is in the **PROD state**, Caliptra observes this activity. The **DEBUG_INTENT_STRAP** can be asserted at any time, but Caliptra handles it in two phases: **Pre-ROM Execution** and **Post-ROM Execution**.
2. **Erasing Secret Assets**:

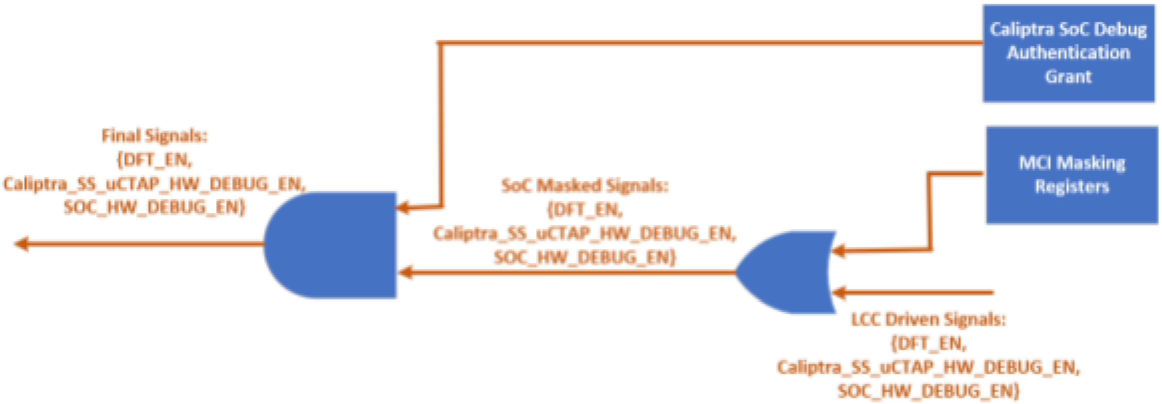
    - Caliptra **wipes secret assets** (Unique Device Secret (UDS) and Field entropy).
3. **Opening TAP Interface**:
    - Once secret assets are erased, the **Caliptra TAP** interface is opened, which the debugger can verify by reading the **TAP's CSR** (Control Status Register).
4. **Receiving Hybrid Public Key and Payload**:
    - The debugger sends a **Hybrid public key** and a **payload** (using a combination of MLDSA and ECC cryptosystems) to Caliptra via the TAP interface.
    - These packages are written to **Caliptra's MailBox.**
5. **Release**:
    - The debugger sets the **CPTRA_BOOTFSM_GO** high via the TAP interface, allowing the internal **BootFSM** to proceed.
    - If debug is requested during run-time (after the ROM has executed), the debugger must set a **GO command** via Caliptra's MailBox.
6. **MailBox Data Locking**:
    - After receiving the GO command, Caliptra **locks the MailBox** data and begins processing the debug request.
    - The debugger then waits for the **evaluation result**.
7. **Key Hash Verification**:
    - Caliptra points a location by using "DEBUG_AUTH_PK_HASH_REG_BANK_OFFSET", "DEBUG_AUTH_PK_HASHES" and "i" values to read the **Hybrid public key hash (PRE_DEBUG_PKs)** from the **FUSE** using the FUSE controller and writes it to the **Reg_CLPT_to_MCU** register.
    - Caliptra proceeds with the **hash comparison** between the **PRE_DEBUG_PKs** and the **DEBUG_PKs**.
8. **Payload Authentication**:
    - If the **hash comparison passes**, Caliptra reads the payload from the MailBox and **authenticates** it using the DEBUG_PKs.
    - If the **authentication** or **hash comparison fails**, Caliptra returns a failure status and reflects it in the **Reg_CLPT_to_MCU** register within the MailBox.
9. **Granting Production Debug Mode**:

    If authentication succeeds, Caliptra does not immediately grant full production debug mode. Instead, the system sets the appropriate **"level of debug" signal**, which corresponds to the type of debug access being requested. This signal is part of an **8-wide signal** that is mapped to the **payload encoding** received during the debug request. The payload encoding can either be **one-hot encoded** or a general **encoded format**, and this signal is passed to the **SoC** to allow it to make the final decision about the level of debug access that should be granted. In Caliptra's subsystem-specific implementation, the logic is configured to handle **one-hot encoding** for these 8 bits. The **level 0 bit** is routed to both **Caliptra** and the **MCU TAP interface**, allowing them to unlock based on this level of debug access. This granular approach ensures that the system can selectively unlock different levels of debugging capability, depending on the payload and the authorization level provided by the debugger.

Masking Logic for Debugging Features in Production Debug Mode

In the production debug mode, the SoC can enable certain debugging features—such as DFT_EN, SOC_HW_DEBUG_EN, and Caliptra_SS_uCTAP_HW_DEBUG_EN—using a masking mechanism implemented within the Manufacturer Control Interface (MCI). This masking infrastructure allows the SoC to selectively control debug features that are normally gated by Caliptra's internal signals. The masking logic functions as a set of registers, implemented in the MCI, that can be written by the SoC to override or enable specific debugging functionalities in production debug mode.

The masking registers in the MCI act as an OR gate with the existing debug signals. For instance, if DFT_EN is controlled by Caliptra, the SoC can assert the corresponding mask register to enable Design for Test (DFT) capabilities in the SoC, even if Caliptra has not explicitly enabled it. Similarly, the SOC_HW_DEBUG_EN and Caliptra_SS_uCTAP_HW_DEBUG_EN signals can be masked through the MCI registers, giving the SoC the flexibility to unlock TAP interfaces and provide the required debugging in production.



Caliptra subsystem's flexibility infrastructure to allow SoC to override debug signals

This mechanism is only authorized when both the LCC and Caliptra core are in the PROD state and operating in production debug mode. The masking logic ensures that these features are enabled securely, in line with the production debug flow described in the "SoC Debug Flow and Architecture for Production Mode" section. By leveraging the MCI’s masking infrastructure, the SoC integrator has greater flexibility to implement custom debugging options during production without compromising the security framework established by Caliptra.

Authorization Content for Debugging Policy

The Caliptra production debug mode architecture supports up to eight distinct categories of debug access, though it is not necessary to utilize all eight. Each category is associated with a unique set of cryptographic keys, offering flexibility and granularity in the debug process. The MCU sends the hash results of up to 8 PRE_DEBUG_PKs to Caliptra. These hash values are derived from the public keys corresponding to each debug category and are securely stored in the MCU’s FUSE.

Upon receiving the hash results, Caliptra selects the appropriate PRE_DEBUG_PK hash to compare against the stored DEBUG_PK, determining which debug category is being requested. This selection is made based on the specific debug operation being invoked. Caliptra then authenticates the request using a Hybrid cryptosystem that combines both ECC (Elliptic Curve Cryptography) and MLDSA (PQC Digital Signature Algorithm).

The private keys corresponding to these public keys are stored securely on an external authorization server, ensuring that they remain protected and isolated from the SoC. For each of the eight debug categories, the validity of the public keys is tracked via a valid/invalid bit stored in the FUSE. This bit serves as a key revocation mechanism, allowing keys to be invalidated if necessary, such as when a key is compromised or no longer needed. This revocation system enhances security by ensuring that only valid keys can be used for debug access, preventing unauthorized attempts to enter debug mode.

The Format of Payload is WIP (challenge, nonce, etc.).

Hardware Requirements Coming with LCC and Debug Flow

Caliptra Requirement List for SoC Debug

- MailBox command ID for payload and public key authentication
- 384-bit MailBox memory allocation for the hashed public key, named Reg_MCU_to_CLPT
- 8KB (WIP) MailBox memory allocation for the payload and public key coming from Caliptra TAP.
- 64-bit MailBox memory allocation for the authentication status register, named Reg_CLPT_to_MCU and a logic to reflect the status of authentication
- Wiring DEBUG_INTENT_STRAP to Caliptra debug mode logic to trigger debug preparation
- Accepting a GO command to start the authentication process and locking MailBox
- A functionality to communicate with MCU to ask hashed public key by using “DEBUG_AUTH_PK_HASH_REG_BANK_OFFSET”, “DEBUG_AUTH_PK_HASHES” and “i” values
- Having MLDSA and ECC infrastructure

Caliptra Requirement List for Manufacturing Debug:

- MailBox command ID for payload and public key authentication

- 256-bit memory allocation for TOKEN received via Caliptra TAP
- Capability to generate 256-bit Nonce
- Accepting a GO (connected to MANUF_DBG_SERVICE) command to start the authentication process and locking MailBox
- Having SHA_512 infrastructure
- A status register that can be accessible by Caliptra TAP and MCU

MCI Requirements:

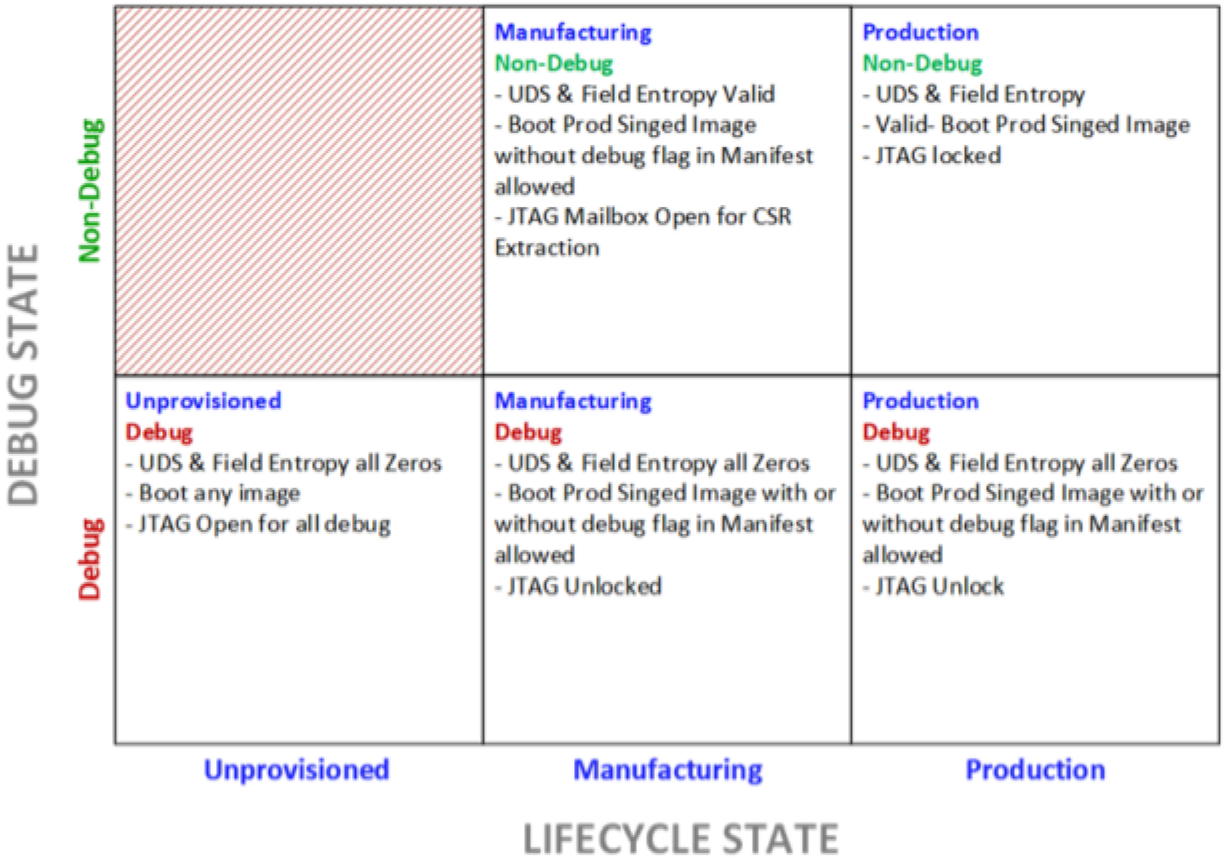
- Masking registers to override DFT_EN, SOC_HW_DEBUG_EN, and Caliptra_SS_uCTAP_HW_DEBUG_EN
- uCTAP Unlock logic
- Caliptra Subsystem's Boot/Reset Sequencer signals for LCC's power manager interface

FUSE Requirements:

- A public content FUSE for hashed public key
- MCU reading flow for hashed public key when Caliptra points "DEBUG_AUTH_PK_HASH_REG_BANK_OFFSET", "DEBUG_AUTH_PK_HASHES" and "i" pointers

LCC Interpretation for Caliptra "Core" Security States

Caliptra Core has five security states as given with the figure below (Copied from Caliptra core specification). Three of these states enable debugging with different features, while two of these states are in non-debug mode. Caliptra Subsystem implements gasket logic to translate the LCC states to Caliptra "Core" security states.



DEBUG STATE	Unprovisioned	Manufacturing	Production
Non-Debug	▨	Manufacturing Non-Debug - UDS & Field Entropy Valid - Boot Prod Singed Image without debug flag in Manifest allowed - JTAG Mailbox Open for CSR Extraction	Production Non-Debug - UDS & Field Entropy - Valid- Boot Prod Singed Image - JTAG locked
Debug	Unprovisioned Debug - UDS & Field Entropy all Zeros - Boot any image - JTAG Open for all debug	Manufacturing Debug - UDS & Field Entropy all Zeros - Boot Prod Singed Image with or without debug flag in Manifest allowed - JTAG Unlocked	Production Debug - UDS & Field Entropy all Zeros - Boot Prod Singed Image with or without debug flag in Manifest allowed - JTAG Unlock

LIFECYCLE STATE

Caliptra "Core" Security States

This translation is essential for ensuring that the system behaves correctly according to its current lifecycle stage, whether it is in a development phase, production, or end-of-life states.

The LCC provides specific decoding signals—DFT_EN, SOC_HW_DEBUG_EN, and Caliptra_SS_uCTAP_HW_DEBUG_EN—that accommodates the debug capabilities and security states of the Caliptra Core. Depending on the values of these signals, the Caliptra Core transitions between different security states as follows:

Non-Debug Mode: This state is enforced when the LCC is in RAW, TEST_LOCKED, SCRAP, INVALID, or POST_TRANSITION states. In these states, the DFT_EN, SOC_HW_DEBUG_EN, and Caliptra_SS_uCTAP_HW_DEBUG_EN signals are all low, ensuring that no debug functionality is available. Caliptra remains in a secure, non-debug state, with no access to debugging interfaces.

Unprovisioned Debug Mode: When the LCC is in the TEST_UNLOCKED state, the DFT_EN, SOC_HW_DEBUG_EN, and Caliptra_SS_uCTAP_HW_DEBUG_EN signals are all set high, enabling debug functionality. In this mode, the Caliptra Core allows extensive debugging capabilities, which is typically used during early development and bring-up phases.

Manufacturing Non-Debug Mode: This state occurs when the LCC is in the DEV state, with SOC_HW_DEBUG_EN high and Caliptra_SS_uCTAP_HW_DEBUG_EN low. In this state, the secrets have been programmed into the system, and the Caliptra can generate CSR (Certificate Signing Request) upon request. However, it remains in a secure, non-debug mode to prevent reading secrets through the debugging interfaces.

Manufacturing Debug Mode: Also occurring in the DEV state, this mode is enabled when both SOC_HW_DEBUG_EN and Caliptra_SS_uCTAP_HW_DEBUG_EN are high. Here, the Caliptra Core provides debugging capabilities while maintaining security measures suitable for manufacturing environments.

Production Non-Debug Mode: This state is active when the LCC is in the PROD or PROD_END states, with all debug signals (DFT_EN, SOC_HW_DEBUG_EN, and Caliptra_SS_uCTAP_HW_DEBUG_EN) set to low. The Caliptra Core operates in a secure mode with no debug access, suitable for fully deployed production environments.

Production Debug Mode : This state is active when the LCC is in the PROD state, with debug DFT_EN, SOC_HW_DEBUG_EN set to low, and Caliptra_SS_uCTAP_HW_DEBUG_EN is high. Caliptra Core provides debugging capabilities while maintaining security measures suitable for manufacturing environments.

Production Debug Mode in RMA: In the RMA state, all debug signals are set high, allowing full debugging access. This state is typically used for end-of-life scenarios where detailed inspection of the system's operation is required. *However, the LCC is not standalone enough to put Caliptra into production debug mode. Steps described in SoC Debug Flow and Architecture for Production Mode Section should be followed.*

The table below summarizes the relationship between the LCC state, the decoder output signals, and the resulting Caliptra “Core” security state:

***Table**: LCC state translation to Caliptra "Core" security states*

LCC State\Decoder Output	DFT_EN	SOC_HW_DEBUG_EN	Caliptra_SS_uCTAP_HW_DEBUG_EN	Caliptra “Core” Security States
RAW	Low	Low	Low	Non-Debug
TEST_LOCKED	Low	Low	Low	Non-Debug
TEST_UNLOCKED	High	High	High	Unprov Debug
DEV	Low	High	Low	Manuf Non-Debug
DEV*	Low	High	High	Manuf Debug
PROD	Low	Low	Low	Prod Non-Debug
PROD*	Low**	High**	High	Prod Debug
PROD_END	Low	Low	Low	Prod Non-Debug
RMA	High	High	High	Prod Debug
SCRAP	Low	Low	Low	Non-Debug
INVALID	Low	Low	Low	Non-Debug
POST_TRANSITION	Low	Low	Low	Prod Non-Debug

Note: In RAW, TEST_LOCKED, SCRAP and INVALID states, Caliptra “Core” is not brought out of reset.

*: These states are Caliptra SS’s extension to LCC. Although the LCC is in either DEV or PROD states, Caliptra core can grant debug mode through the logics explained in

“How does Caliptra enable uCTAP_UNLOCK?” and *“SoC Debug Flow and Architecture for Production Mode”*.

**: SOC_HW_DEBUG_EN and DFT_EN can be overridden by SoC support in PROD state.

SOC LCC Interface usage & requirements

The interaction between the SoC and the LCC within the Caliptra Subsystem is pivotal for maintaining the security and functionality of the overall system. This section outlines the specific usage and requirements for interfacing with the LCC from the SoC perspective.

SOC_HW_DEBUG_EN Utilization: The SOC_HW_DEBUG_EN signal plays a crucial role in controlling access to the Chip-Level TAP (CLTAP). When SOC_HW_DEBUG_EN is asserted high, it enables the exposure of registers to the TAP interface that are deemed secure for access during debugging. This is particularly important when CLTAP is open, as it allows authorized debugging operations while ensuring that sensitive or secure information remains protected.

uController TAPs Debug Access: For security reasons, SoCs must restrict access to their uCTAPs. These TAPs should only be opened for debugging when the Caliptra_SS_uCTAP_HW_DEBUG_EN signal, driven by the Caliptra, is set high. This ensures that debug access is only permitted under controlled conditions, typically during development or when specific secure transitions are in place.

LCC Outputs for Security/Debug Purposes: Beyond the primary debug controls, all other LCC outputs, as listed in the signal table below, are available for use by the SoC for any security or debugging purposes. These outputs provide the SoC with additional control signals that can be leveraged to enhance system security, monitor debug operations, or implement custom security features.

Integration with Clock/Voltage Monitoring Logic: If the SoC includes clock or voltage monitoring logic, it is recommended that these components be connected to the LCC's alert handling interface. By integrating with the LCC’s alert system, the SoC can ensure that any detected anomalies, such as voltage fluctuations or clock inconsistencies, trigger appropriate secure transitions as defined in [1]. This connection enhances the overall security posture by enabling the LCC to respond dynamically to potential threats or system irregularities.

These requirements ensure that the SoC and LCC work in tandem to maintain a secure environment, particularly during debugging and system monitoring. By adhering to these guidelines, the SoC can leverage the LCC's capabilities to protect sensitive operations and enforce security policies across the system.

LCC Module: Summarized Theory of operation

The LCC is designed to handle the various life-cycle states of the device, from initial provisioning through manufacturing to production and eventual decommissioning (RMA). It provides mechanisms for state transitions, secure key management, and debug control, ensuring that the device operates securely throughout its lifecycle. For more information, please refer to the Life-cycle controller documentation here [1]. While the LCC is reused from OpenTitan, Caliptra Subsystem's security & debug architecture has some differences. Therefore, Subsystem implements the functions required to meet Caliptra specific requirements.

LCC has the same power up sequence as described in Life cycle controller documentation of OpenTitan open-source silicon Root of Trust (RoT) project [1]. After completing the power up sequence, the LCC enters the normal operation routine. During this phase, its output remains static unless specifically requested to change. The LCC accepts life-cycle transition change requests from its TAP interface (TAP Pin Muxing). There are two types of state transitions: (i) unconditional transitions and (ii) conditional transitions.

For unconditional transitions, the LCC advances the state by requesting an OTP update to the FUSE controller. Once the programming is confirmed, the life cycle controller reports a success to the requesting agent and waits for the device to reboot.

For conditional transitions, the LCC can also branch different states (RAW_UNLOCK, TEST_UNLOCK, TEST_EXIT, RMA_UNLOCK) based on the received token through the TAP interface and compared the received tokens against the ones stored in FUSE . This branching is called conditional transitions. For more information about the conditional states, please refer to OpenTitan open-source silicon Root of Trust (RoT) project [1].

Notes:

- Some tokens are hardcoded design constants (specifically for RAW to TEST_UNLOCK), while others are stored in FUSE.

- Conditional transitions will only be allowed if the FUSE partition holding the corresponding token has been provisioned and locked.

- For transition counter limits and token hashing mechanism, please refer to OpenTitan open-source silicon Root of Trust (RoT) project [1].

- The LCC enters the post transition handling routine after completing conditional and unconditional transitions. During this routine, the LCC disables all of its decoded outputs and puts the system in an inert state.

Key Components and Interfaces

The LCC interfaces with various system components and controls multiple security-related signals.

***Table**: LCC RTL configurations*

Parameter	Default	Description
AlertAsyncOn	'h3	Enable asynchronous alerts.
IdcodeValue	'h1	ID code for the LCC JTAG TAP.
RndCnstLcKeymgrDivInvalid	'h0	Diversification value for invalid life cycle states.
RndCnstLcKeymgrDivTestUnlocked	'h1	Diversification value for TEST_UNLOCKED states.
RndCnstLcKeymgrDivDev	'h2	Diversification value for DEV state.
RndCnstLcKeymgrDivProduction	'h3	Diversification value for PROD/PROD_END states.

RndCnstLcKeymgrDivRma	'h4	Diversification value for RMA state.

The following table illustrates LCC Module’s inter-module signal. The first three column lists the port names, their direction and their struct type format, while the last column lists the sections that point to the sub-section where the functionality is described.

Table: *LCC Module's inter-module signals*

Port	Direction	Type	Description
jtag_i	input	jtag_pkg::jtag_req_t	Jtag interface connection
jtag_o	output	jtag_pkg::jtag_rsp_t	Jtag interface connection
tl_o	output	tlul_pkg::tl_d2h_t	TileLink-UL bus protocol but Caliptra has TileLink-UL to AXI converter to carry the transactions through AXI.

tl_i	input	tlul_pkg::tl_h2d_t	TileLink-UL bus protocol but Caliptra has TileLink-UL to AXI converter to carry the transactions through AXI.
alert_rx_i	input	prim_alert_pkg::alert_rx_t	Alert Handler Interface
alert_tx_o	output	prim_alert_pkg::alert_tx_t	Alert Handler Interface
esc_scrap_state0_tx_i	input	prim_esc_pkg::esc_tx_t	Alert Handler Interface
esc_scrap_state0_rx_o	output	prim_esc_pkg::esc_rx_t	Alert Handler Interface
esc_scrap_state1_tx_i	input	prim_esc_pkg::esc_tx_t	Alert Handler Interface
esc_scrap_state1_rx_o	output	prim_esc_pkg::esc_rx_t	Alert Handler Interface
pwr_lc_i	input	pwrmgr::pwr_lc_req_t	Boot/Reset Sequencer Interface

pwr_lc_o	output	pwrmgr::pwr_lc_rsp_t	Boot/Reset Sequencer Interface
lc_otp_program_o	output	otp_ctrl_pkg::lc_otp_program_req_t	FUSE Controller Interface
lc_otp_program_i	input	otp_ctrl_pkg::lc_otp_program_rsp_t	FUSE Controller Interface
kmac_data_o	output	kmac_pkg::app_req_t	(Internal) KMAC Interface
kmac_data_i	input	kmac_pkg::app_rsp_t	(Internal) KMAC Interface
otp_lc_data_i	input	otp_ctrl_pkg::otp_lc_data_t	FUSE Controller Interface
lc_keymgr_div_o	output	lc_keymgr_div_t	NOT REQUIRED
lc_flash_rma_seed_o	output	lc_flash_rma_seed_t	NOT REQUIRED
otp_device_id_i	input	otp_device_id_t	Device ID
otp_manuf_state_i	input	otp_manuf_state_t	FUSE Controller Interface
lc_otp_vendor_test_o	output	otp_ctrl_pkg::lc_otp_vendor_test_req_t	FUSE Controller Interface

lc_otp_vendor_test_i	input	otp_ctrl_pkg::lc_otp_vendor_test_rsp_t	FUSE Controller Interface
lc_dft_en_o	output	lc_tx_t	Decoder Interface
lc_nvm_debug_en_o	output	lc_tx_t	NOT REQUIRED
lc_hw_debug_en_o	output	lc_tx_t	Decoder Interface
lc_cpu_en_o	output	lc_tx_t	Decoder Interface
lc_creator_seed_sw_rw_en_o	output	lc_tx_t	Decoder Interface
lc_owner_seed_sw_rw_en_o	output	lc_tx_t	Decoder Interface
lc_iso_part_sw_rd_en_o	output	lc_tx_t	NOT REQUIRED
lc_iso_part_sw_wr_en_o	output	lc_tx_t	NOT REQUIRED
lc_seed_hw_rd_en_o	output	lc_tx_t	Directed to SoC
lc_keymgr_en_o	output	lc_tx_t	NOT REQUIRED
lc_escalate_en_o	output	lc_tx_t	Directed to SoC
lc_check_byp_en_o	output	lc_tx_t	Directed to SoC

lc_clk_byp_req_o	output	lc_tx_t	Clock Manager Interface
lc_clk_byp_ack_i	output	lc_tx_t	Clock Manager Interface
lc_flash_rma_req_o	output	lc_tx_t	NOT REQUIRED
scan_rst_ni	input	-	Directed to SoC. Resets scan chains for DFT purposes; active-low input.
scanmode_i	input	prim_mubi_pkg::mubi4_t	Directed to SoC. Controls the chip's scan mode, allowing for test operations via scan chains.
strap_en_override_o	output	-	Directed to SoC. Overrides hardware strap configurations, typically used during testing.

hw_rev_o	output	lc_hw_rev_t	Directed to SoC. Outputs the hardware revision, important for version control and debugging.

For security alter types and the security countermeasure, we refer OpenTitan open-source silicon Root of Trust (RoT) project [1].

Boot/Reset Sequencer Interface

The LCC Module communicates with Caliptra Subsystem's Boot/Reset Sequencer (implemented within MCI) through the pwr_lc_i and pwr_lc_o signals. The LCC Module initializes itself based on the indications carried by these signals. For further details about the functionality and the meaning of these signals, please refer to the OpenTitan open-source silicon Root of Trust (RoT) project [1].

FUSE Controller Interface

The FUSE controller interface is driven by Caliptra Subsystem's FUSE controller Module, providing an abstraction interface that the LCC can use to interact with a proprietary FUSE block. Data transitions between the LCC and the FUSE controller are carried through the "lc_otp_program_o" and "lc_otp_program_i" signals. The "otp_lc_data_t" is typically a data structure or type that carries information about the life cycle state and related data from the FUSE memory. It includes fields that represent the current life cycle state, transition counters, and any tokens or keys necessary for life cycle transitions. Additionally, "lc_otp_vendor_test_o" and "lc_otp_vendor_test_i" are used as vendor test control signals that can be accessed from/to the life cycle TAP interface.

(Internal) KMAC Interface

This architecture creates a wrapper around LCC and instantiates a KMAC module along with the LCC (see Figure 1). KMAC handles the hashing of token operations. Since the KMAC and life cycle controllers are in different clock domains, the KMAC interface signals are synchronized to the life cycle clock inside the life cycle controller [1].

Clock Manager Interface

The LCC has a connection with Caliptra Subsystem's clock manager. Since LCC starts the state transition before the clock becomes stable, LCC has an option to bypass the wait time of getting the stable clock. This request goes through "lc_clk_byp_req_o" and "lc_clk_byp_ack_i".

Alert Handler Interface

LCC is sensitive to alerts that are generated by peripherals such as voltage glitch detection or other fault injection sensors. Therefore, an alert receiver module [1] is connected to LCC's alert_rx_i and alert_tx_o ports (see Figure 1). Based on the received alert and also LCC state transition, LCC has a right to escalate the alerts. To that end, LCC is connected to two escalation receiver modules [1] for state0 and state1 escalation ports. Both alert receiver and escalation receiver modules are driven by the peripheral connections and these connections are out of Caliptra Subsystem's scope.

LCC Bibliography

[1] https://opentitan.org/book/hw/ip/lc_ctrl/doc/theory_of_operation.html

MCU Hardware

MCU (Manufacturer Core Unit) is a RISC-V core capable of loading firmware to the SoC. Mainly responsible for performing SoC specific initialization and configuration. MCU HW requirements are as documented in MCU HW requirements.

MCU Hardware Configuration & Capabilities

RISC-V is generated using following configuration for various capabilities,

- **BHT_ADDR_HI (8’h09), BHT_ADDR_LO (6’h02)**: These parameters define the high and low bits of the Branch History Table (BHT) address. The high address is set to 8'h09 and the low address is set to 6'h02.
- **BHT_ARRAY_DEPTH (15’h0100)**: This parameter defines the depth of the BHT array. The depth is set to 15'h0100, which means the BHT can store 256 entries.
- **BHT_GHR_HASH_1 (5’h00), BHT_GHR_SIZE (8’h08)**: These parameters define the size and hash of the Global History Register (GHR) in the BHT. The size is set to 8'h08, which means the GHR can store 8 bits. The hash is set to 5'h00.
- **BHT_SIZE (16’h0200)**: This parameter defines the size of the BHT. The size is set to 16'h0200, which means the BHT can store 512 entries.
- **BITMANIP_ZBA, BITMANIP_ZBB, BITMANIP_ZBC, BITMANIP_ZBE, BITMANIP_ZBF, BITMANIP_ZBP, BITMANIP_ZBR, BITMANIP_ZBS (5’h01, 5’h01, 5’h01, 5’h00, 5’h00, 5’h00, 5’h00, 5’h01)**: These parameters enable or disable various bit manipulation extensions. The ZBA, ZBB, ZBC, and ZBS extensions are enabled (5'h01), while the ZBE, ZBF, ZBP, and ZBR extensions are disabled (5'h00).
- **BTB_ADDR_HI (9’h009), BTB_ADDR_LO (6’h02)**: These parameters define the high and low bits of the Branch Target Buffer (BTB) address. The high address is set to 9'h009 and the low address is set to 6'h02.
- **BTB_ARRAY_DEPTH (13’h0100)**: This parameter defines the depth of the BTB array. The depth is set to 13'h0100, which means the BTB can store 256 entries.
- **BTB_BTAG_FOLD (5’h00), BTB_BTAG_SIZE (9’h005)**: These parameters define the size and folding of the BTB tag. The size is set to 9'h005, which means the BTB tag can store 5 bits. The folding is disabled (5'h00).
- **BTB_ENABLE (5’h01)**: This parameter enables or disables the BTB. The BTB is enabled (5'h01).
- **BTB_FOLD2_INDEX_HASH (5’h00)**: This parameter defines the hash for the second index in the BTB. The hash is set to 5'h00.
- **BTB_FULLYA (5’h00)**: This parameter enables or disables fully associative BTB. The fully associative BTB is disabled (5'h00).
- **BTB_INDEX1_HI (9’h009), BTB_INDEX1_LO (9’h002), BTB_INDEX2_HI (9’h011), BTB_INDEX2_LO (9’h00A), BTB_INDEX3_HI (9’h019), BTB_INDEX3_LO (9’h012)**: These parameters define the high and low bits of the three BTB indices. The high and low bits for the first index are 9'h009 and

9'h002, for the second index are 9'h011 and 9'h00A, and for the third index are 9'h019 and 9'h012.

- **BTB_SIZE (14’h0200)**: This parameter defines the size of the BTB. The size is set to 14'h0200, which means the BTB can store 512 entries.
- **BTB_TOFFSET_SIZE (9’h00C)**: This parameter defines the size of the BTB target offset. The size is set to 9'h00C, which means the BTB target offset can store 12 bits.
- **BUILD_AHB_LITE (5’h01), BUILD_AXI4 (4’h0), BUILD_AXI_NATIVE (5’h01)**: These parameters enable or disable various bus interfaces. The AHB Lite and AXI Native interfaces are enabled (5'h01), while the AXI4 interface is disabled (4'h0).
- **BUS_PRTY_DEFAULT (6’h03)**: This parameter defines the default bus priority. The default bus priority is set to 6'h03.
- **DATA_ACCESS_ADDR0 to DATA_ACCESS_ADDR7 (36’h000000000)**: These parameters define the addresses for data access. All addresses are set to 36'h000000000.
- **DATA_ACCESS_ENABLE0 to DATA_ACCESS_ENABLE7 (5’h00)**: These parameters enable or disable data access. All data accesses are disabled (5'h00).
- **DATA_ACCESS_MASK0 to DATA_ACCESS_MASK7 (36’h0FFFFFFFF)**: These parameters define the masks for data access. All masks are set to 36'h0FFFFFFFF.
- **DCCM_BANK_BITS (7’h02), DCCM_BITS (9’h011), DCCM_BYTE_WIDTH (7’h04), DCCM_DATA_WIDTH (10’h020), DCCM_ECC_WIDTH (7’h07), DCCM_ENABLE (5’h01), DCCM_FDATA_WIDTH (10’h027), DCCM_INDEX_BITS (8’h0D), DCCM_NUM_BANKS (9’h004), DCCM_REGION (8’h05), DCCM_SADR (36’h050000000), DCCM_SIZE (14’h0080), DCCM_WIDTH_BITS (6’h02)**: These parameters configure the Data Close Coupled Memory (DCCM).
- **DIV_BIT (7’h04), DIV_NEW (5’h01)**: These parameters configure the divider. The divider bit is set to 7'h04 and the new divider is enabled (5'h01).
- **DMA_BUF_DEPTH (7’h05), DMA_BUS_ID (9’h001), DMA_BUS_PRTY (6’h02), DMA_BUS_TAG (8’h01)**: These parameters configure the Direct Memory Access (DMA) controller. The buffer depth is set to 7'h05, the bus ID is set to 9'h001, the bus priority is set to 6'h02, and the bus tag is set to 8'h01.
- **FAST_INTERRUPT_REDIRECT (5’h01)**: This parameter enables or disables fast interrupt redirection. Fast interrupt redirection is enabled (5'h01).

- **ICACHE_2BANKS (5’h01), ICACHE_BANK_BITS (7’h01), ICACHE_BANK_HI (7’h03), ICACHE_BANK_LO (6’h03), ICACHE_BANK_WIDTH (8’h08), ICACHE_BANKS_WAY (7’h02), ICACHE_BEAT_ADDR_HI (8’h05), ICACHE_BEAT_BITS (8’h03), ICACHE_BYPASS_ENABLE (5’h01), ICACHE_DATA_DEPTH (18’h00200), ICACHE_DATA_INDEX_LO (7’h04), ICACHE_DATA_WIDTH (11’h040), ICACHE_ECC (5’h01), ICACHE_ENABLE (5’h00), ICACHE_FDATA_WIDTH (11’h047), ICACHE_INDEX_HI (9’h00C), ICACHE_LN_SZ (11’h040), ICACHE_NUM_BEATS (8’h08), ICACHE_NUM_BYPASS (8’h02), ICACHE_NUM_BYPASS_WIDTH (8’h02), ICACHE_NUM_WAYS (7’h02), ICACHE_ONLY (5’h00), ICACHE_SCND_LAST (8’h06), ICACHE_SIZE (13’h0010), ICACHE_STATUS_BITS (7’h01), ICACHE_TAG_BYPASS_ENABLE (5’h01), ICACHE_TAG_DEPTH (17’h00080), ICACHE_TAG_INDEX_LO (7’h06), ICACHE_TAG_LO (9’h00D), ICACHE_TAG_NUM_BYPASS (8’h02), ICACHE_TAG_NUM_BYPASS_WIDTH (8’h02), ICACHE_WAYPACK (5’h01)**: These parameters configure the Instruction Cache (ICache).
- **ICCM_BANK_BITS (7’h02), ICCM_BANK_HI (9’h003), ICCM_BANK_INDEX_LO (9’h004), ICCM_BITS (9’h011), ICCM_ENABLE (5’h01), ICCM_ICACHE (5’h00), ICCM_INDEX_BITS (8’h0D), ICCM_NUM_BANKS (9’h004), ICCM_ONLY (5’h01), ICCM_REGION (8’h04), ICCM_SADR (36’h040000000), ICCM_SIZE (14’h0080)**: These parameters configure the Instruction Close Coupled Memory (ICCM).
- **IFU_BUS_ID (5’h01), IFU_BUS_PRTY (6’h02), IFU_BUS_TAG (8’h03)**: These parameters configure the Instruction Fetch Unit (IFU) bus.
- **INST_ACCESS_ADDR0 to INST_ACCESS_ADDR7 (36’h000000000)**: These parameters define the addresses for instruction access. All addresses are set to 36'h000000000.
- **INST_ACCESS_ENABLE0 to INST_ACCESS_ENABLE7 (5’h00)**: These parameters enable or disable instruction access. All instruction accesses are disabled (5'h00).
- **INST_ACCESS_MASK0 to INST_ACCESS_MASK7 (36’h0FFFFFFFF)**: These parameters define the masks for instruction access. All masks are set to 36'h0FFFFFFFF, which means all bits are valid for instruction access.
- **LOAD_TO_USE_PLUS1 (5’h00)**: This parameter defines the load-to-use latency. The latency is set to 5'h00, which means there is no additional latency.
- **LSU2DMA (5’h00)**: This parameter enables or disables the Load Store Unit (LSU) to DMA interface. The interface is disabled (5'h00).

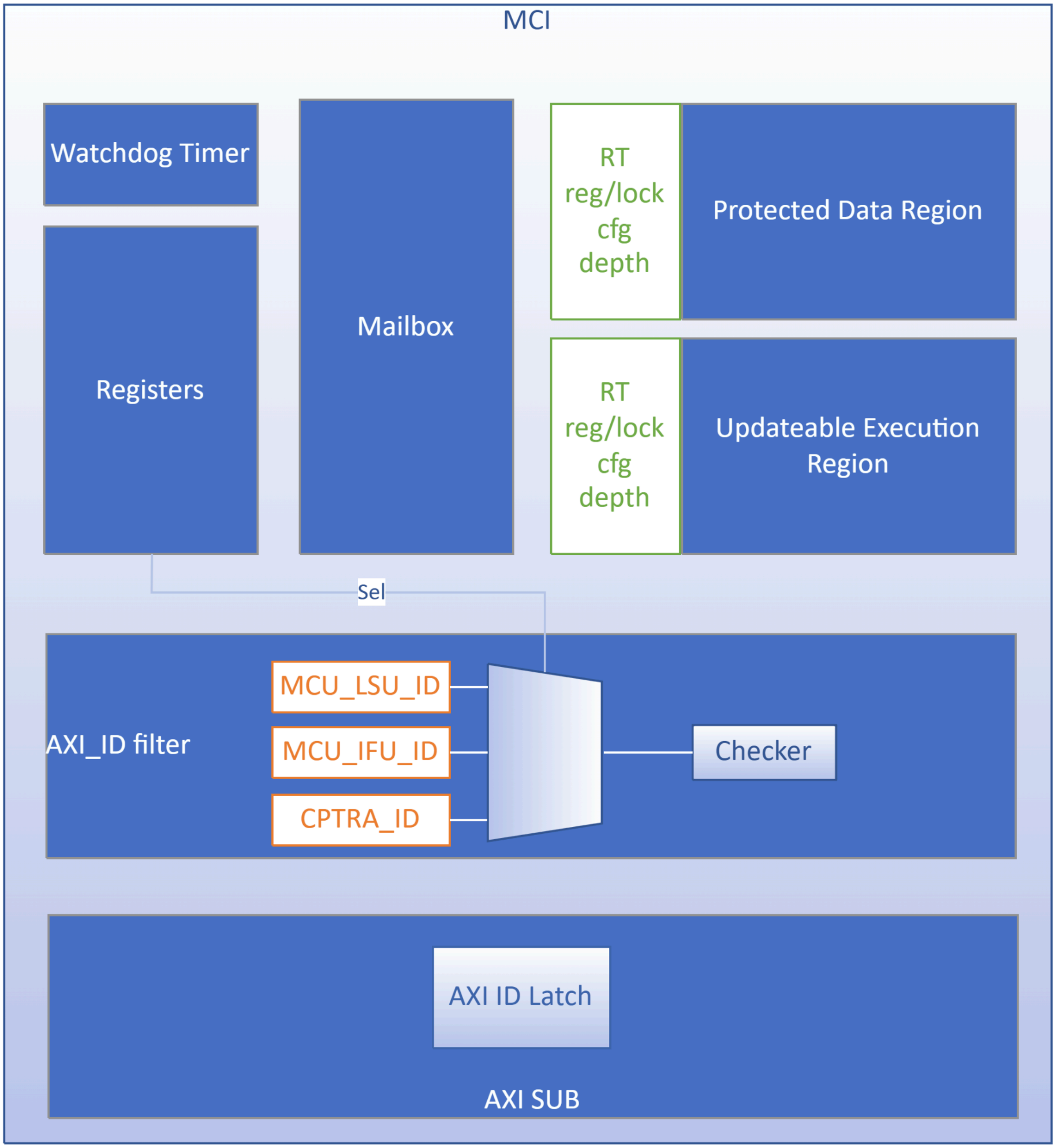
- **LSU_BUS_ID (5’h01), LSU_BUS_PRTY (6’h02), LSU_BUS_TAG (8’h03)**: These parameters configure the LSU bus. The bus ID is set to 5'h01, the bus priority is set to 6'h02, and the bus tag is set to 8'h03.
- **LSU_NUM_NBLOAD (9’h004), LSU_NUM_NBLOAD_WIDTH (7’h02)**: These parameters configure the number of non-blocking loads in the LSU. The number of non-blocking loads is set to 9'h004 and the width is set to 7'h02.
- **LSU_SB_BITS (9’h011)**: This parameter defines the number of bits in the LSU scoreboard. The number of bits is set to 9'h011.
- **LSU_STBUF_DEPTH (8’h04)**: This parameter defines the depth of the LSU store buffer. The depth is set to 8'h04.
- **NO_ICCM_NO_ICACHE (5’h00)**: This parameter disables both the ICCM and ICache when set to 5'h00.
- **PIC_2CYCLE (5’h00)**: This parameter enables or disables 2-cycle operation in the Programmable Interrupt Controller (PIC). The 2-cycle operation is disabled (5'h00).
- **PIC_BASE_ADDR (36’h060000000)**: This parameter defines the base address of the PIC. The base address is set to 36'h060000000.
- **PIC_BITS (9’h00F)**: This parameter defines the number of bits in the PIC. The number of bits is set to 9'h00F.
- **PIC_INT_WORDS (8’h01)**: This parameter defines the number of interrupt words in the PIC. The number of interrupt words is set to 8'h01.
- **PIC_REGION (8’h06)**: This parameter defines the region of the PIC. The region is set to 8'h06.
- **PIC_SIZE (13’h0020)**: This parameter defines the size of the PIC. The size is set to 13'h0020, which means the PIC can store 32 entries.
- **PIC_TOTAL_INT (12’h01F), PIC_TOTAL_INT_PLUS1 (13’h0020)**: These parameters define the total number of interrupts in the PIC. The total number of interrupts is set to 12'h01F and the total number plus one is set to 13'h0020.
- **RET_STACK_SIZE (8’h08)**: This parameter defines the size of the return stack. The size is set to 8'h08, which means the return stack can store 8 entries.
- **SB_BUS_ID (5’h01), SB_BUS_PRTY (6’h02), SB_BUS_TAG (8’h01)**: These parameters configure the scoreboard bus. The bus ID is set to 5'h01, the bus priority is set to 6'h02, and the bus tag is set to 8'h01.
- **TIMER_LEGAL_EN (5’h01)**: This parameter enables or disables the timer. The timer is enabled (5'h01).

Manufacturer Control Interface (MCI)

Overview

The Manufacturer Control Interface (MCI) is a critical hardware block designed to supplement the Manufacturer Control Unit (MCU) within a System on Chip (SoC). The primary functions of the MCI include providing an SRAM bank, facilitating restricted communication through a mailbox from external entities, and managing a bank of Control/Status Registers (CSRs). Additionally, the MCI incorporates a Watchdog Timer and a Boot Sequencing Finite State Machine (FSM) to manage timing and control during the SoC boot sequence after power application. This boot sequence encompasses reset deassertion, initialization of the Fuse Controller, initialization of the Lifecycle Controller, and enabling the JTAG block for debugging and manufacturing processes.

The following diagram illustrates the internal components of the MCI.



Sub-block Descriptions

Control/Status Registers (CSRs)

The Control/Status Registers (CSRs) within the MCI are designed to provide critical control and status monitoring functions for the SoC. These registers include configuration settings, status indicators, and control bits that allow communication and management of the various operations of the MCI. The CSR bank is accessible via the AXI interface and is mapped into the memory space to facilitate straightforward access and manipulation.

TODO: Migrate reg descriptions to HTML, generated from RDL

Capabilities

Reports MCU configuration values

Boot Status

Reports FSM state for Boot Sequencer

ERROR FATAL/NON_FATAL

HW REV ID

FW REV ID

WDT Timeout0

WDT Timeout1

WDT CTRL

WDT STATUS

Mailbox AXI_ID [0-4]

Mailbox AXI_ID Lock [0-4]

MCU Reset Request

[0] Req: Writable by Caliptra, on set to 1, intr to MCU asserts. Once MCU writes “Ack”, MCU reset asserts.

[1] Clr: Writable by Caliptra; autoclears; on set to 1, “[0] Set” is cleared, MCU reset deasserts

MCU Reset Ack

[0] Ack: Writable by MCU, MCU reset asserts (when Req also 1).

MCU Reset Reason

Populated by Caliptra to indicate reset was requested for purpose of FW update (auto-populated based on Req/Ack).

MCU_RUNTIME_LOCK

Reset value: 0

Writable only by Caliptra, only when already at 0. (W1S, cleared only by MCU FW Update reset).

With a value of 0, only the Caliptra AXI ID is allowed to access the Updateable Execution SRAM region in the MCU.

Writing 1 to this register enables the MCU LSU and IFU AXI IDs to access the Updateable Execution SRAM Region. AXI IDs are provided as integration parameter/macro to MCU.

Writing to this register also indicates to MCU ROM that it may perform context switch to RT image.

FW_SRAM_EXEC_REGION_SIZE

Reset value: Full span of the FW SRAM.

Reset signal: SoC reset (NOT FW Update reset).

May only be written by MCU ROM. Enforced by a check against MCU_RUNTIME_LOCK value.

Min value: 4KiB

Max value: Total size of the FW SRAM

CALIPTRA_BOOT_GO

Reset value: GO=0 (= cptra_rst_b = 0)

Write-1-to-set, writable only by MCU (will be written by ROM -only-, gated by AXI ID).

Once set, Caliptra reset is deasserted (cptra_rst_b = 1).

Boot Sequencer Finite State Machine (FSM)

The Boot Sequencer FSM is responsible for the orderly and controlled boot process of the SoC. This state machine ensures that all necessary initialization steps are completed in the correct sequence after power application. The boot sequence includes the following steps:

- Reset deassertion

- Fuse Controller Initialization

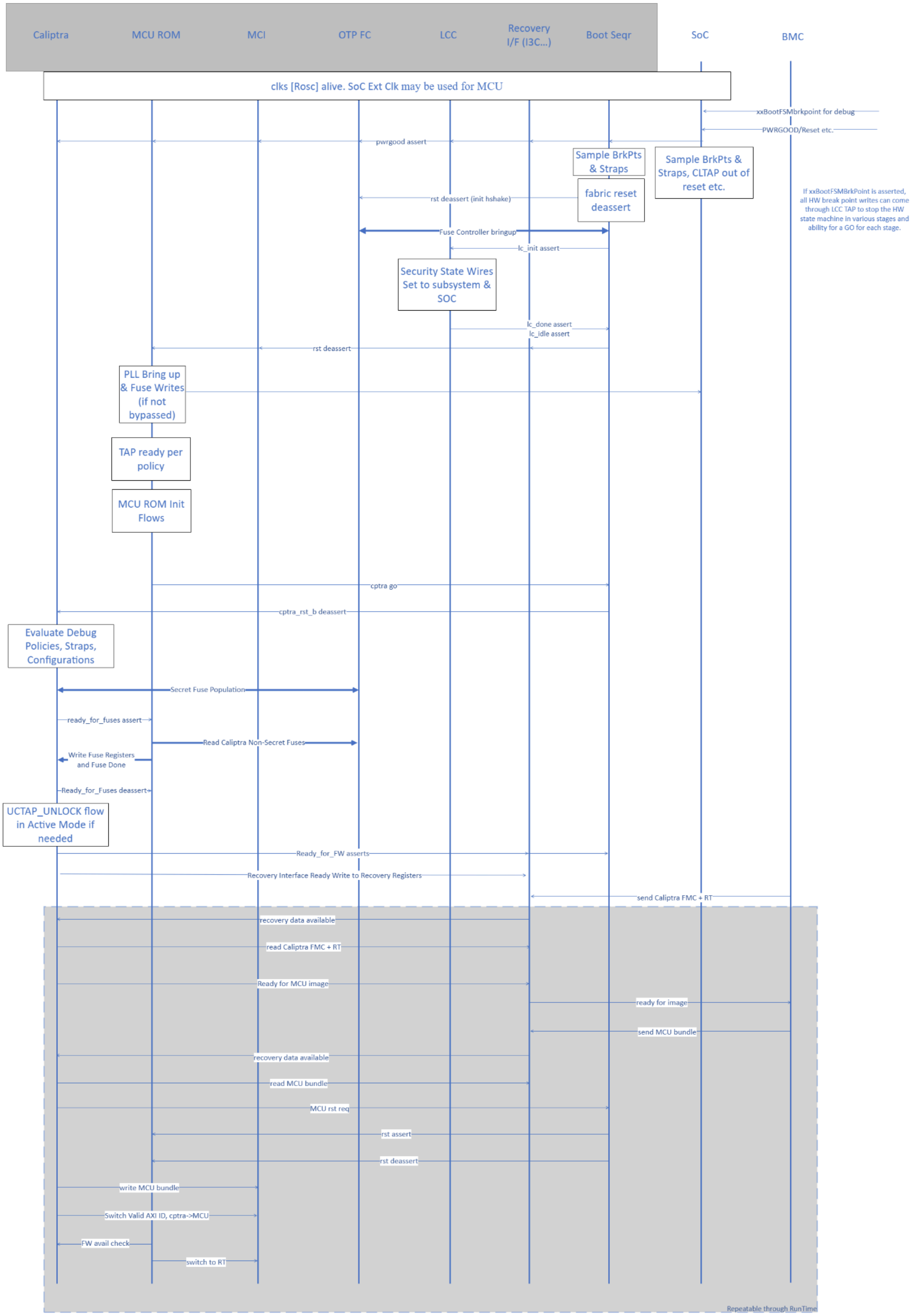
- Lifecycle Controller Initialization

- Enabling the JTAG interface for debug and manufacturing purposes

The boot sequence also allows for an SoC to boot on an external reference clock or an internal ring oscillator, and switch to a PLL clock later during the boot cycle. MCU may perform steps to enable a PLL clock and perform the clock transition.

The following boot flow diagram illustrates the FSM operations.

TODO: Breakpoints for Debug/Manuf


Caliptra
MCU ROM
MCI
OTP FC
LCC
Recovery I/F (I3C...)
Boot Seqr
SoC
BMC
clks [Rosc] alive. SoC Ext Clk may be used for MCU
xxBootFSMbrkpoint for debug
PWRGOOD/Reset etc.
pwrgood assert
Sample BrkPts & Straps
Sample BrkPts & Straps, CLTAP out of reset etc.
If xxBootFSMBrkPoint is asserted, all HW break point writes can come through LCC TAP to stop the HW state machine in various stages and ability for a GO for each stage.
rst deassert (init hshake)
fabric reset deassert
Fuse Controller bringup
lc_init assert
Security State Wires Set to subsystem & SOC
lc_done assert
lc_idle assert
rst deassert
PLL Bring up & Fuse Writes (if not bypassed)
TAP ready per policy
MCU ROM Init Flows
cptra go
cptra_rst_b deassert
Evaluate Debug Policies, Straps, Configurations
Secret Fuse Population
ready_for_fuses assert
Read Caliptra Non-Secret Fuses
Write Fuse Registers and Fuse Done
Ready_for_Fuses deassert
UCTAP_UNLOCK flow in Active Mode if needed
Ready_for_FW asserts
Recovery Interface Ready Write to Recovery Registers
send Caliptra FMC + RT
recovery data available
read Caliptra FMC + RT
Ready for MCU image
ready for image
send MCU bundle
recovery data available
read MCU bundle
MCU rst req
rst assert
rst deassert
write MCU bundle
Switch Valid AXI ID, cptra->MCU
FW avail check
switch to RT
Repeatable through RunTime

Watchdog Timer

The Watchdog Timer within the MCI is a crucial component designed to enhance the reliability and robustness of the SoC. This timer monitors the operation of the system and can trigger a system reset if it detects that the system is not functioning correctly. The Watchdog Timer is configurable through CSRs and provides various timeout periods and control mechanisms to ensure the system operates within defined parameters.

Mailbox

The Mailbox component of the MCI allows for secure and restricted communication between external SoC entities and the MCU. This communication channel is essential for exchanging control messages, status updates, and other critical information that the MCU will use monitor system boot, firmware updates, and security critical operations. The Mailbox is designed to ensure that only authorized entities can access and communicate through it, preserving the integrity and security of the SoC operations.

Mailbox logic is adapted from the Caliptra Mailbox and follows the same programming flow and rules as defined for Caliptra.

MCU SRAM

The MCU SRAM provides essential data and instruction memory for the Manufacturer Control Unit. This SRAM bank is utilized by the MCU to load firmware images, store application data structures, and create a runtime stack. The SRAM is accessible via the AXI interface and is mapped into the MCI's memory space for easy access and management. Exposing this SRAM via a restricted API through the SoC AXI interconnect enables seamless and secured Firmware Updates to be managed by Caliptra.

AXI ID filtering is used to restrict access within the MCU SRAM based on system state and accessor. Access permissions are based on the AXI_ID that is enabled through the MCU_RUNTIME_LOCK register (either the Caliptra AXI_ID, or the MCU IFU/LSU AXI IDs). Any write attempt by an invalid AXI_ID is discarded and returns an error status. Any read attempt returns 0 data and an error status.

The MCU SRAM contains two regions, a Protected Data Region and an Updateable Execution Region, each with a different set of access rules.

The span of each region is dynamically defined by the MCU ROM during boot up. Once MCU has switched to running Runtime Firmware, the RAM sizing is locked until any SoC-level reset. ROM uses the register FW_SRAM_EXEC_REGION_SIZE to configure the SRAM allocation.

The Updateable Execution Region may only be read/written by Caliptra prior to setting the MCU_RUNTIME_LOCK register and may only be read/written by the MCU IFU or MCU LSU after MCU_RUNTIME_LOCK is set. The Protected Data Region may never be accessed by Caliptra or the MCU IFU. Only the MCU LSU is allowed to read or write to the Protected Data Region, regardless of whether MCU ROM or MCU Runtime firmware is running.

Memory Map

The following memory map defines the address offsets for accessing all internal blocks of the MCI via the AXI interface. The memory map provides static offsets while allowing integrators to flexibly allocate memory sizes up to 512 KiB.

TODO: Define some supported offsets/spans and include in testplan

Internal Block	**Address Offset**
CSRs	0x0000
Mailbox	0x80000
MCU SRAM	0x100000

Port List

Interface	**Signal Name**	**Direction**	**Description**
clk		in	
soc_rst_b		in	

SoC Reset Outputs	axi_rst_b	out	
	fc_rst_b	out	
	lcc_rst_b	out	
	tap_rst_b	out	
	mcu_rst_b	out	
	cptra_rst_b	out	
SoC Reset Window Outputs	axi_rst_win	out	
	fc_rst_win	out	
	lcc_rst_win	out	
	tap_rst_win	out	
	mcu_rst_win	out	
	cptra_rst_win	out	
LCC I/F	lc_init		
	lc_done		
	lc_idle		
AXI Sub			
Interrupts	TBD		

Reset Domain Crossing

Each reset output will be accompanied by a reset window signal. Attached logic that is reset by the the MCU boot sequencer ensures that all signals driven by that reset are stabilized to an idle value immediately on observing the reset window, to avoid metastability events on loads from different reset domains.

Integrator Guide

The integration of the MCI into the SoC requires careful planning and coordination with other blocks in the system. Integrators must define an address map for the AXI-attached interconnect to ensure that all components can communicate effectively. Additionally, integrators must set static AXI ID values for the MCU and other agents, such as Caliptra, that will interact with the MCI. These AXI ID values are critical in restricting access to the MCI, ensuring that only permitted entities can access its resources.

The MCI interacts with other SoC components through the following interfaces:

- AXI Interface: Used for directly accessing the CSRs, Mailbox, and SRAM.

- JTAG Interface: Enabled during the boot sequence for debugging and manufacturing processes.

- Mailbox Interface: Facilitates restricted communication with external entities.

- Interrupts: Routed to MCU to generate notifications about system-level events.

Integrators must ensure that the AXI-attached interconnect is configured to allow seamless access to the MCI while maintaining security and integrity. This involves defining address ranges, setting up access permissions, and configuring AXI ID values to control access.

Subsystem HW security Construction

Also To be documented:

- ROM specific flow detail integration into the document