Trusted firmware for A-profile systems

Generic boot

Trainer: Sumit Garg
Linaro Support and Solutions Engineering

Linaro
Developer Services

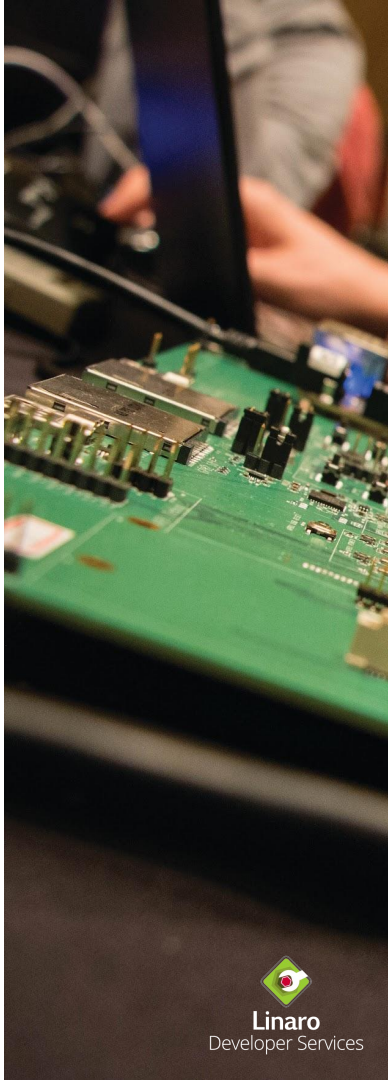
Logistics

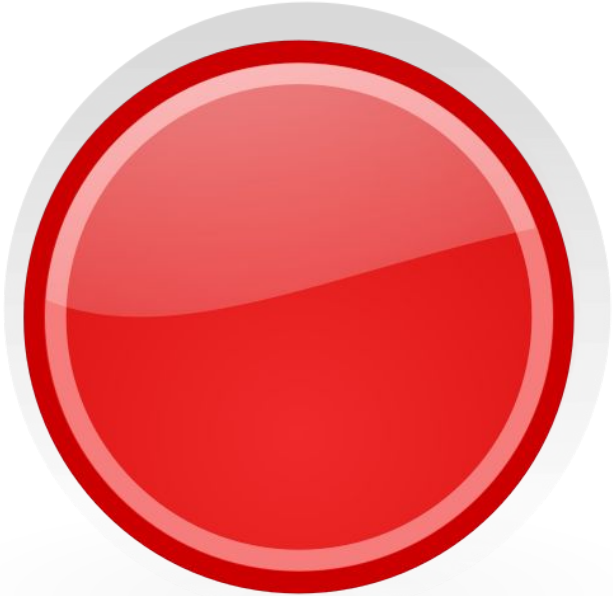
These sessions will be recorded. Turn off your camera if you do not want to appear in the recording.

Questions welcome! There is time allocated for Q&A at the end of today's session but you can ask relevant questions verbally or in the chat as we go.

Please keep your microphone muted when not speaking!

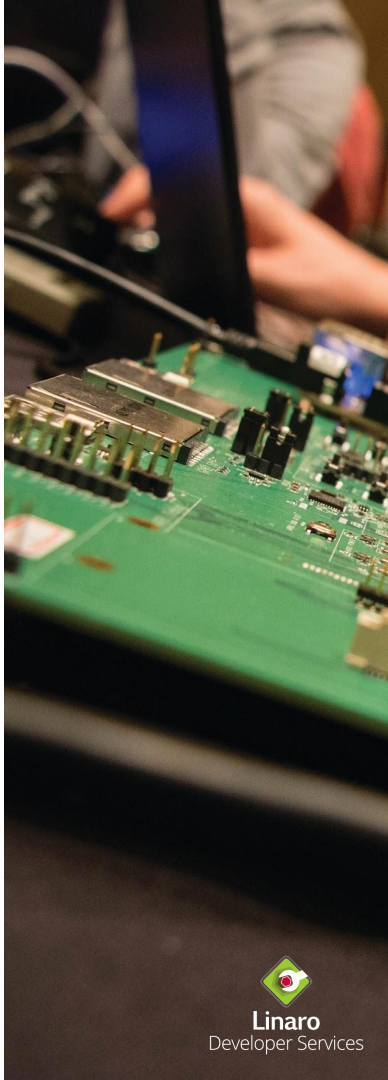
Slides and lab resources can be downloaded from:
https://fileserver.linaro.org/s/fE6iBYca8bYqrrk





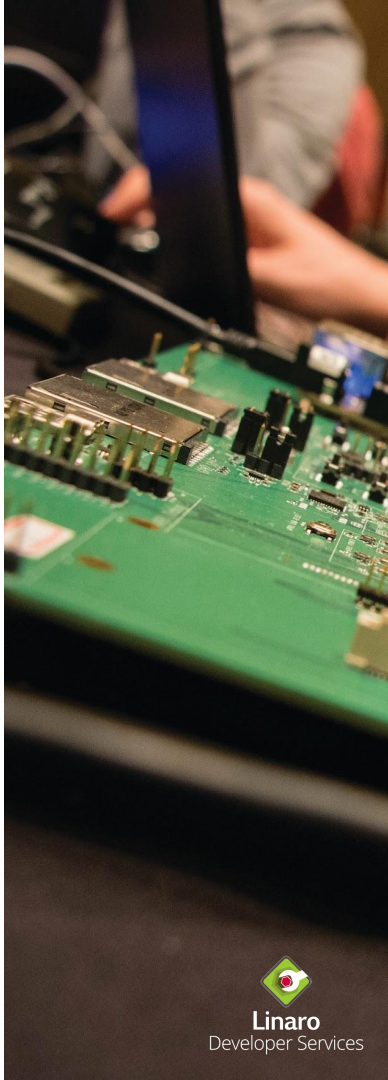
Training modules overview

1. Introduction to TF-A
2. **Generic boot**
3. Firmware security
4. Secure/Realm world interfaces



Generic boot

- **Boot flows**
  - **Bootloaders image terminology**
- Image organisation
  - Firmware Image Package (FIP)
- Console API framework
  - Log levels
  - Crash reporting
- IO storage abstraction layer
- BL2 image parameter passing
- Locking primitives
- Device tree
  - Firmware Configuration Framework



Bootloaders terminology

BL1	Application processor (AP_* prefix optional) level 1 bootloader (boot ROM)
BL2	Application processor level 2 bootloader (trusted boot f/ware)
BL31	Application processor EL3 payload (monitor firmware)
BL32	Application processor S-EL1 payload (trusted OS)
BL33	Application processor NS-EL1/EL2 normal world bootloader
SCP_BL1	System coprocessor level 1 bootloader
SCP_BL2	System coprocessor payload

Bootloaders terminology - II

NS_BL1U	Application processor level 1 updater (boot ROM, Non-secure EL1)
NS_BL2U	Application processor level 2 updater (2nd stage updater, Non-secure EL1)
BL2U	Application processor level 2 updater (Secure EL1)
MCP_BL1	Management Control Processor level 1 firmware (boot ROM)
MCP_BL2	Management Control Processor level 2 firmware (RAM firmware)

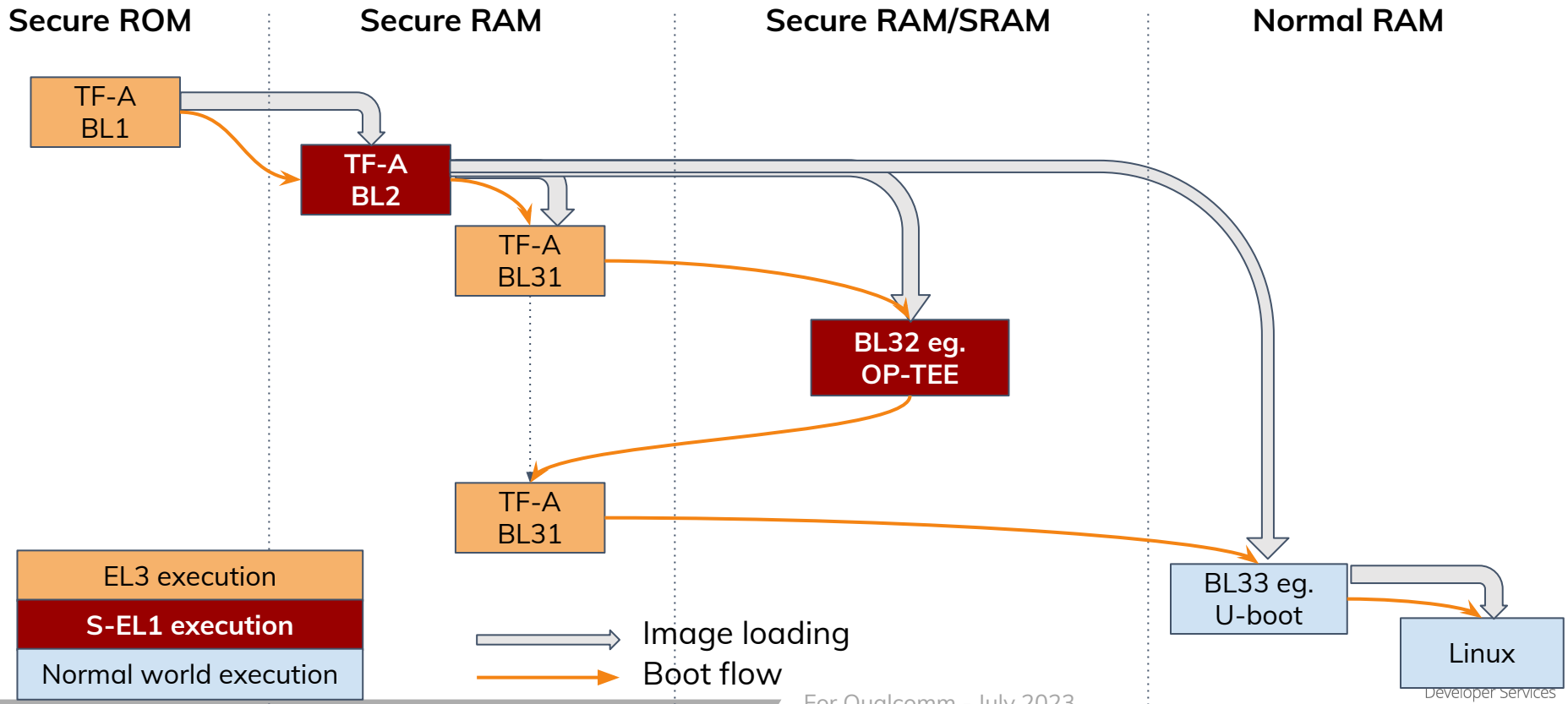
Cold/warm boot

Cold boot is boot from power on (full initialization)

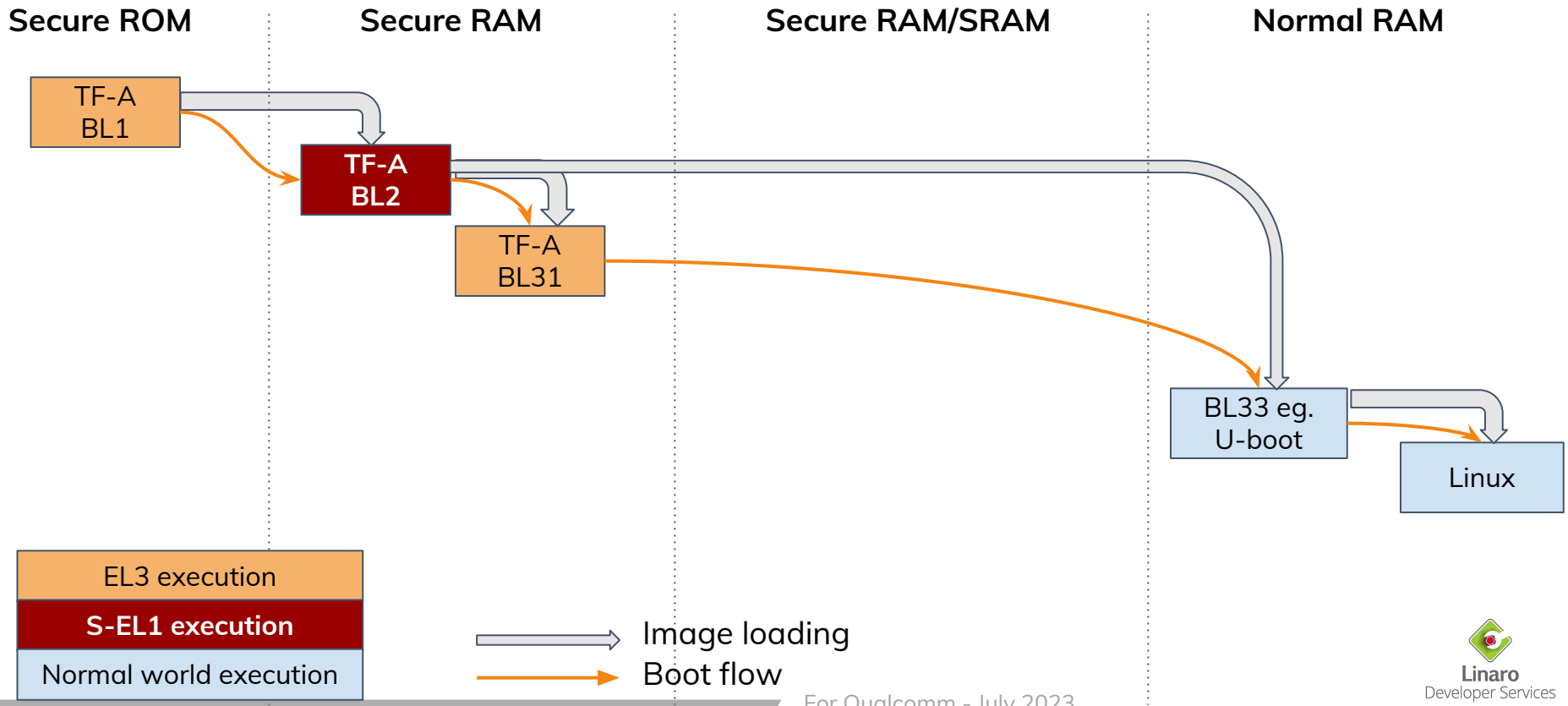
Warm boot is bringing a new core online within a running system

https://trustedfirmware-a.readthedocs.io/en/latest/getting_started/image-terminology.html

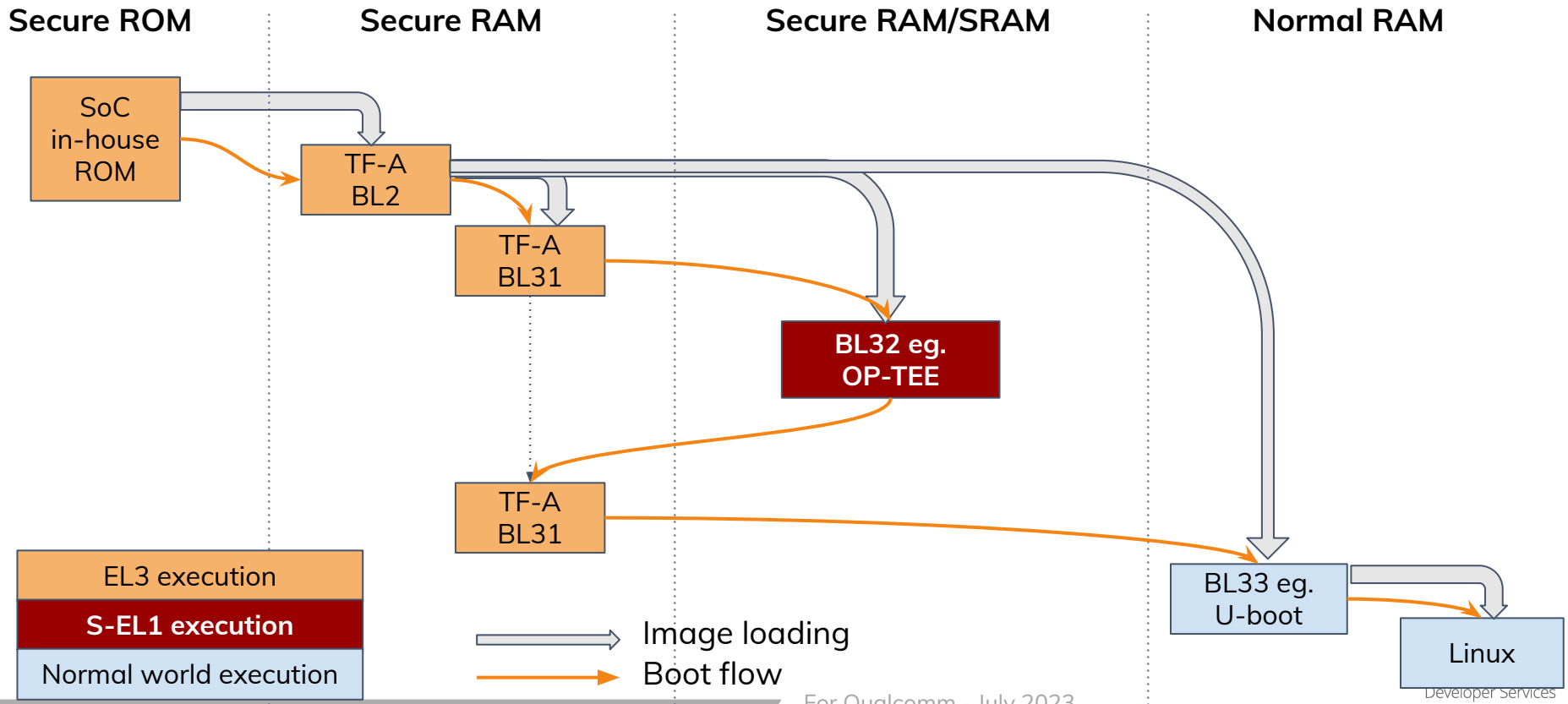
Main boot flow



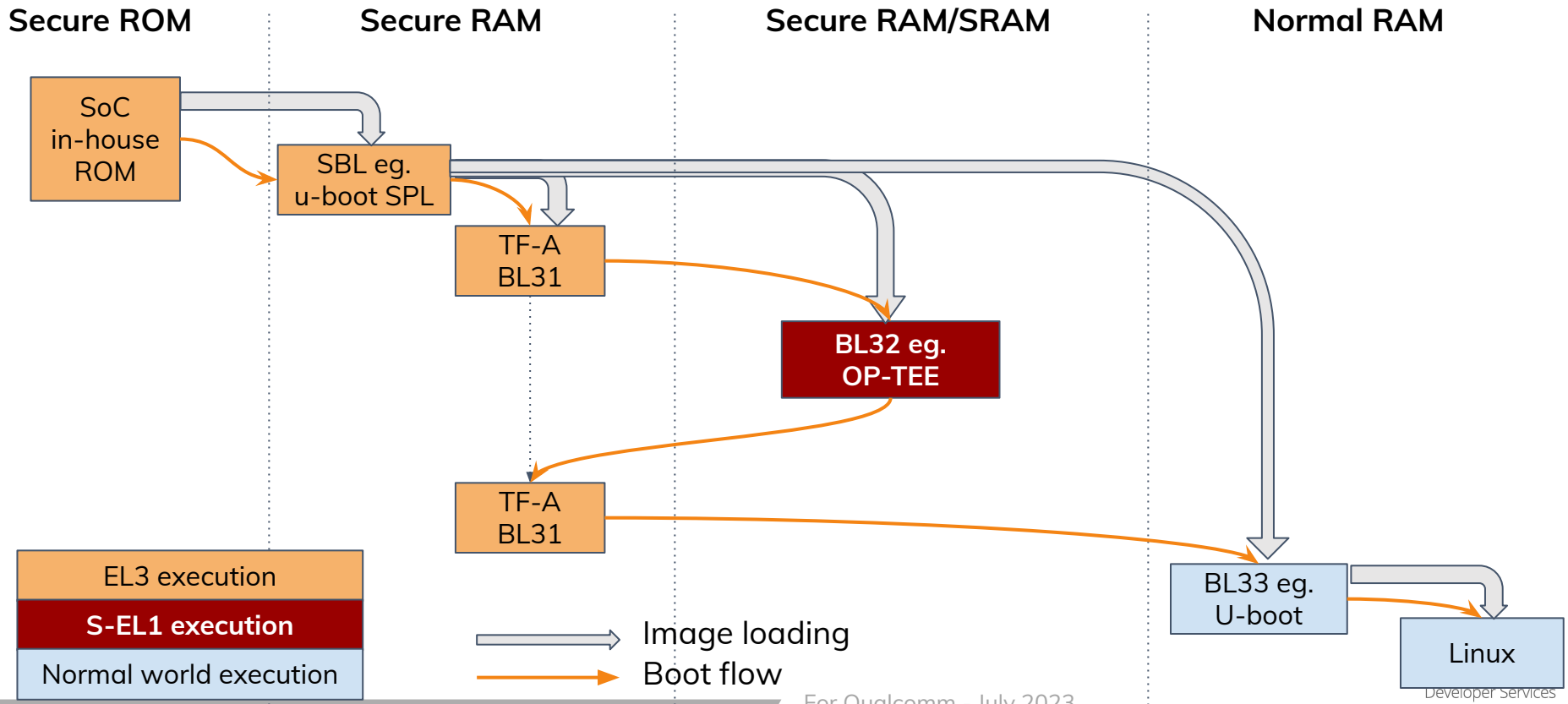
Boot flow without secure OS



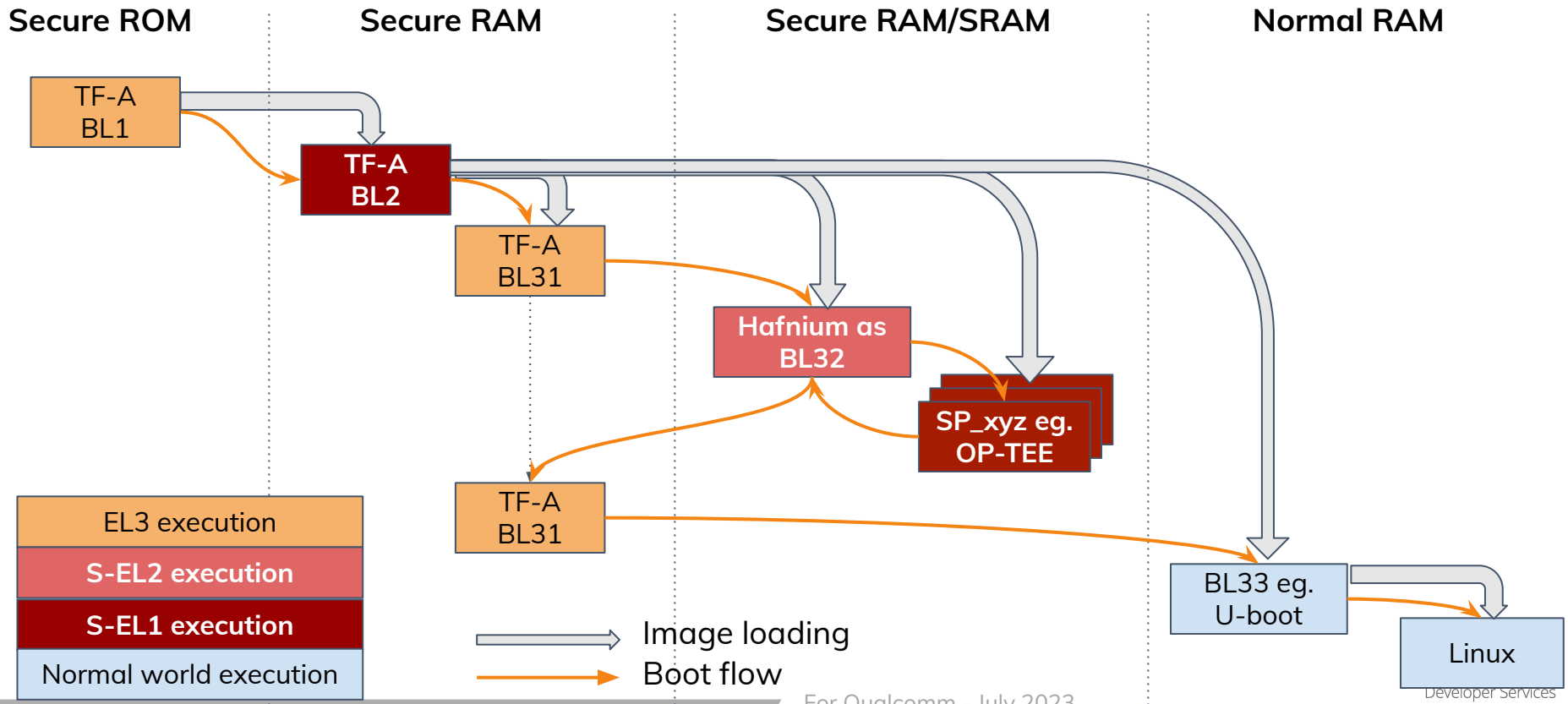
Boot flow with custom BootROM: RESET_TO_BL2



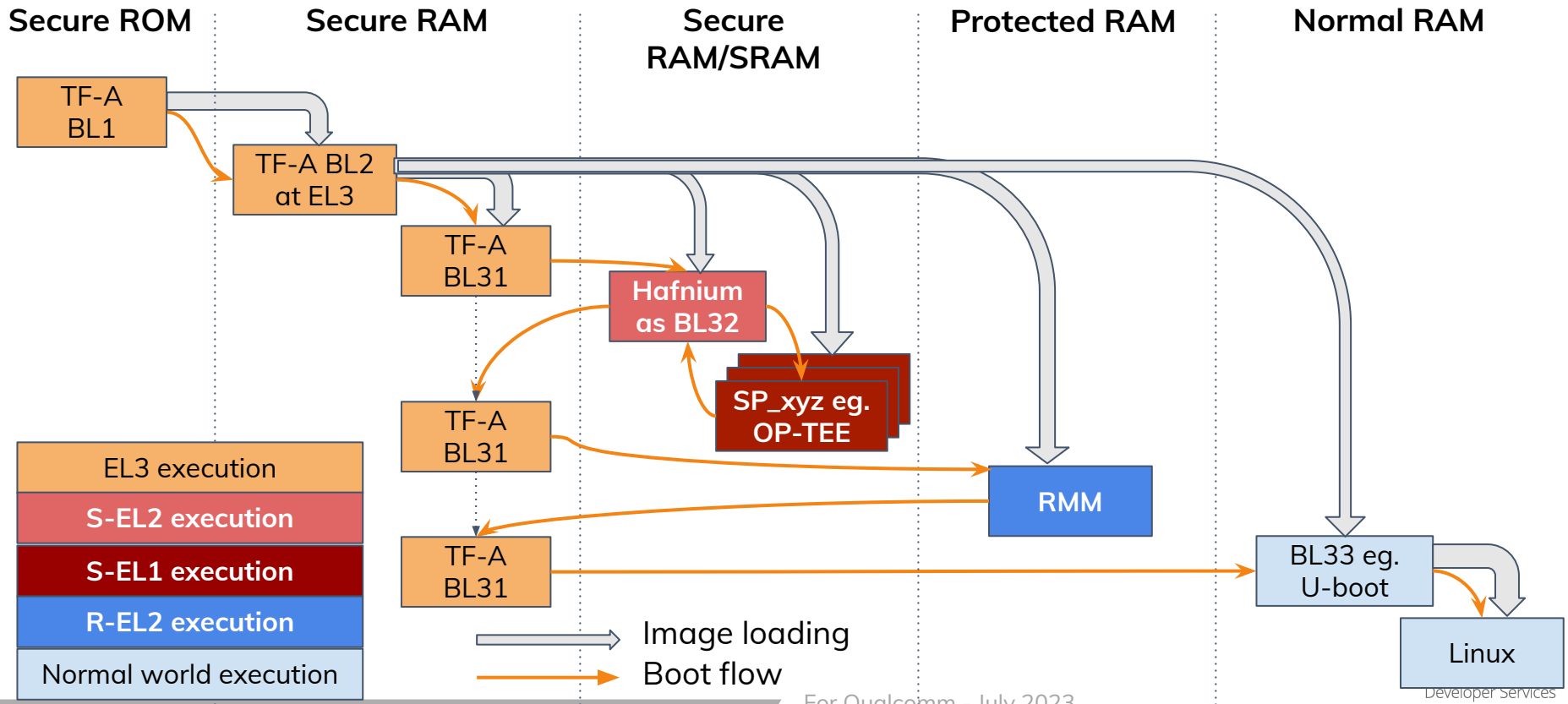
Boot flow with custom bootloader: RESET_TO_BL31



Boot flow with S-EL2 (Armv8.4 onwards)



Boot flow with S-EL2 and RMM (Armv9)



TF-A BL1: Level 1 bootloader (boot ROM)

BL1 is a reference implementation of the **first unmodifiable piece of code** executing on application processor at **EL3 exception level**. It is commonly referred as Boot ROM.

Its primary purpose is to:

- Perform the **minimum initialization** necessary.
- Load and authenticate an updateable **second stage bootloader (BL2)** image into an executable **SRAM or on-chip RAM** location.
- **Hand-off control** to the BL2 image.

However, most SoC comes with their **proprietary** implementation of boot ROM. But TF-A BL1 can be taken as a reference to implement BL2 image authentication and loading in a **generic manner**.

TF-A BL2: Level 2 bootloader

BL2 is the reference second stage bootloader. It is commonly referred to as "**Trusted boot firmware**". By default it runs at **S-EL1** exception level but can be configured to run at **EL3** via BL2_RUNS_AT_EL3. Its purpose:

- Responsible for **initializing DRAM**, has corresponding silicon vendor specific drivers.
- Does the bulk of loading next stage firmwares/bootloaders (**BL31, BL32, BL33, RMM etc**) into DRAM. It has a **generic I/O framework** which supports drivers from multiple silicon vendors.
- Responsible for **authenticating and optionally decrypting** firmware images, provides a reference implementation for **Trusted Board Boot Requirements (TBBR)**.
- **Hand-off control** to the EL3 Runtime Firmware (BL31).

TF-A BL31: EL3 runtime firmware

BL31 is the **most privileged** exception level where the **runtime resident** monitor firmware executes. TF-A provides **reference implementation** for EL3 monitor firmware.

Provide **runtime services**, such as:

- Arm architectural services
- Standard services such as **PSCI**, **SDEI**, **MM**, **TRNG** etc.
- SiP/OEM specific services
- Foundation to build:
  - Trusted Execution Environment (**TEE**)
  - **Realms** for Confidential Compute Architecture (**CCA**)

to lower exception levels (EL1/EL2 and S-EL1/S-EL2)

- **EL1 -> Rich OS** or **EL2 -> Hypervisor**
- **S-EL1 -> Trusted OS** or **S-EL2 -> Secure hypervisor**

BL32: S-EL1 payload

Pre Armv8.4 architectures

BL32 represents an **optional Trusted OS payload** executing at S-EL1 exception level. BL31 runtime firmware commonly provides a **Trusted OS specific dispatcher** responsible for:

- **Initializing** the Trusted OS.
- **Routing** requests and responses between the Trusted OS and REE OS.
- Trusted OS specific **context management**.

Upstream supported Trusted OSes:

- **OP-TEE OS** and dispatcher as **opteed**, a Trusted Firmware community project.
- **Google Trusty** and dispatcher as **trusty**.
- **Nvidia Trusted Little Kernel (TLK)** and dispatcher as **tlkd**.
- **ProvenCore micro-kernel** and dispatcher as **pncd**.

BL32: S-EL2 payload

Post Armv8.4 architectures

With the advent of Secure world hypervisor (**S-EL2**), it is now possible to have **multiple Trusted OSes** executing and **isolated** from each other. The major use-case is to provide isolated trusted OS service corresponding to each **virtual machine** running in normal world.

Here BL32 represents S-EL2 payload aka **Secure Partition Manager Core (SPMC)**. **Hafnium** is a reference SPMC implementation, a Trusted Firmware community project. BL31 runtime firmware provides a corresponding **Secure Partition Manager Dispatcher (SPMD)** responsible for:

- **Initializing** the SPMC.
- Routing **Firmware Framework-A (FF-A)** protocol messages among SPMC and REE hypervisor.
- SPMC specific **context management**.

BL33: Normal world bootloader

BL33 represents the normal world bootloader whose primary purpose is to **load and boot** normal world OS.

Major normal world bootloaders known to work with TF-A:

- U-boot
- EDK2
- Coreboot

RMM: Realm Management Monitor firmware

Armv9 architecture: RME extension

The RMM is a software component that runs at **Realm EL2** and forms part of a system which implements the **Arm Confidential Compute Architecture** (Arm CCA).

RMM provides a **Realm Management Interface (RMI)** towards normal world hypervisor to initiate and control realm VM. All the resource management (eg. memory) for realm VM happens via this interface. BL31 provides **RMM dispatcher (RMMD)** to relay messages among RMM and normal world hypervisor.

RMM provides a **Realm Service Interface (RSI)** towards realm VM to request services such as:

- Realm **attestation report** which also includes platform attestation.
- **Memory management** requests from the realm VM to the RMM.

Generic boot

- Boot flows
  - Bootloaders image terminology
- **Image organisation**
  - **Firmware Image Package (FIP)**
- Console API framework
  - Log levels
  - Crash reporting
- IO storage abstraction layer
- BL2 image parameter passing
- Locking primitives
- Device tree
  - Firmware Configuration Framework

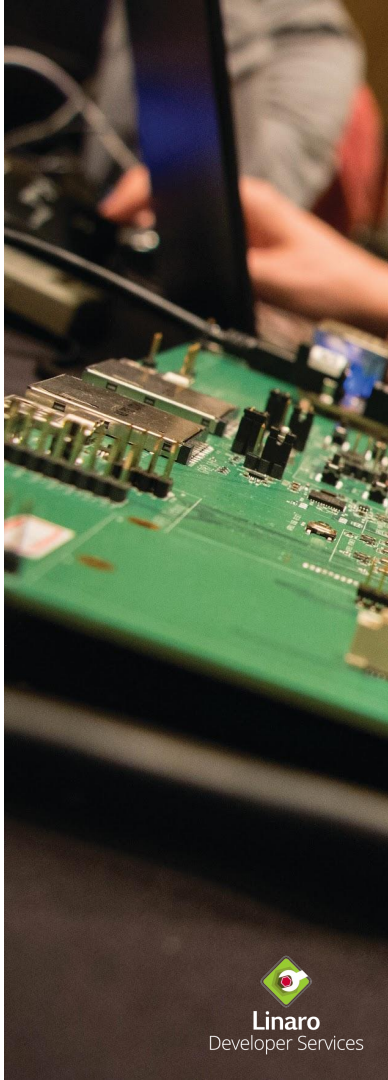
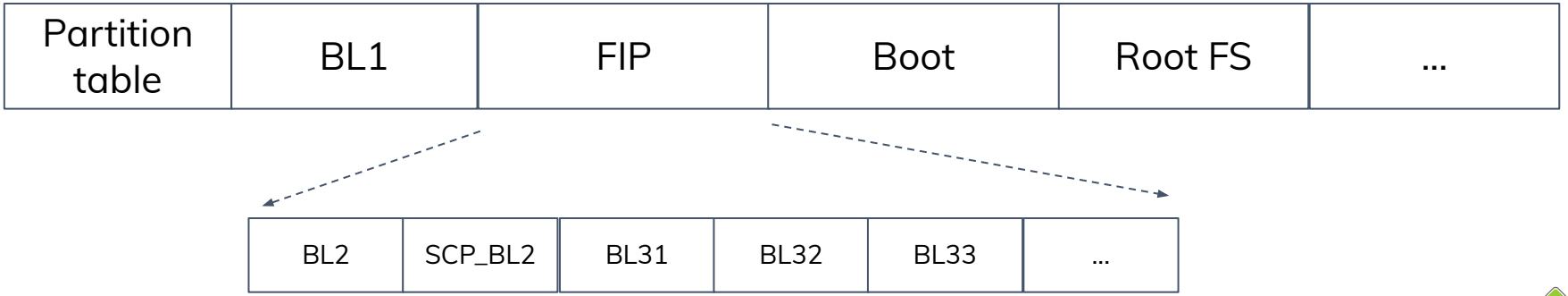


Image organisation

In general design, BL2 is loaded by BL1. Afterwards BL2 loads sequential firmware/bootloader images to SRAM or DRAM.

Firmware Image Package (FIP) is a packaging format used by TF-A to package multiple firmware images in a single binary. A typical FIP image may contain SCP_BL2, BL2, BL31, BL32 and BL33.



Firmware Image Package (FIP)

All the images **bundled** into the FIP are organised in a self-describing format containing image specific information. The number and type of images that should be packed in a FIP is **platform-specific** and may include TF-A images and other firmware images required by the platform.

FIP features:

- **Self-describing** format.
- Flexible for **adding or removing images** to support varying boot flows as per platform requirements.
- Support for **Trusted Board Boot (TBB)** with certificates.
- TF-A provides **fiptool** to generate package for bundling images with a self-describing header.

fiptool: Supported commands

```
$ cd trusted-firmware-a
$ ./tools/fiptool/fiptool help
     usage: fiptool [--verbose] <command> [<args>]
     Global options supported:
       --verbose    Enable verbose output for all commands.

     Commands supported:
       info         List images contained in FIP.
       create       Create a new FIP with the given images.
       update       Update an existing FIP with the given images.
       unpack       Unpack images from FIP.
       remove       Remove images from FIP.
       version      Show fiptool version.
       help         Show help for given command.
```

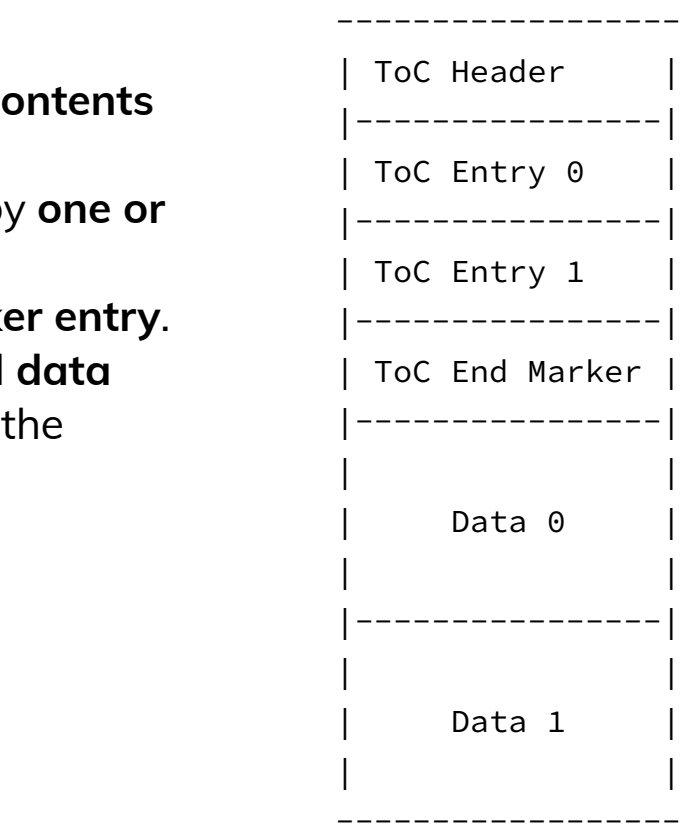
fiptool: A typical usage example

```
$ cd trusted-firmware-a
$ make PLAT=XXX SPD=opteed \
      SCP_BL2=.../lpm3.img \
      BL32=.../tee-header_v2.bin \
      BL33=.../u-boot.bin all fip


$ tools/fiptool/fiptool create \
      --align 512 \
      --tb-fw .../bl2.bin \
      --scp-fw .../lpm3.img \
      --soc-fw .../bl31.bin \
      --tos-fw .../tee-header_v2.bin \
      --nt-fw .../u-boot.bin .../fip.bin
```

Usually it's not needed to directly invoke fiptool command, when compiling TF-A with option "fip", it will automatically invoke fiptool to generate FIP binary.

FIP format

- The FIP layout consists of a **Table of Contents (ToC)** followed by payload data.
- The ToC itself has a **header** followed by **one or more table entries**.
- The ToC is terminated by an **end marker entry**.
- All ToC entries describe **some payload data** that has been appended to the end of the binary package.



FIP format - II

ToC header fields:

`**name**`: The name of the ToC. This is currently used to validate the header.
`**serial_number**`: A non-zero number provided by the creation tool
`**flags**`: Flags associated with this data.
Bits 0-31: Reserved
Bits 32-47: Platform defined
Bits 48-63: Reserved

ToC entry fields:

`**uuid**`: All files are referred to by a predefined Universally Unique IDentifier [UUID] . The UUIDs are defined in `include/tools_share/firmware_image_package.h`. The platform translates the requested image name into the corresponding UUID when accessing the package.
`**offset_address**`: The offset address at which the corresponding payload data can be found. The offset is calculated from the ToC base address.
`**size**`: The size of the corresponding payload data in bytes.
`**flags**`: Flags associated with this entry. None are yet defined.

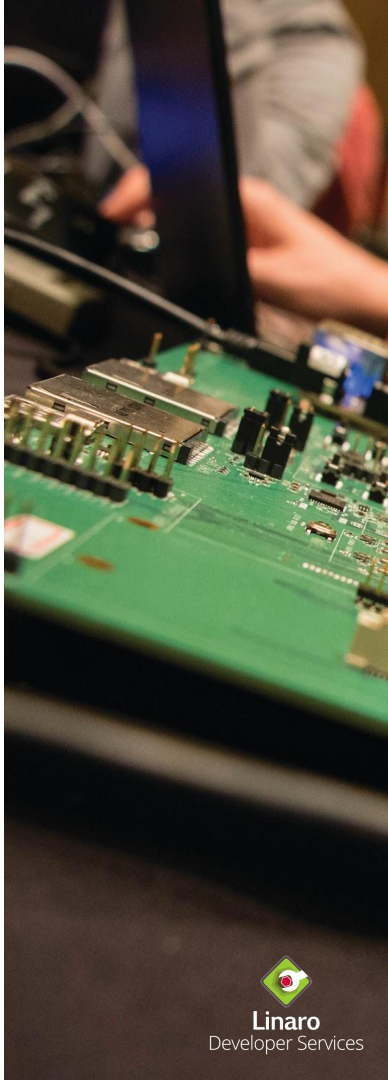
FIP format - III

fip binary format


```
$ tools/fiptool/fiptool info .../fip.bin
Trusted Boot Firmware BL2: offset=0x400, size=0x14611, cmdline="--tb-fw"
SCP Firmware SCP_BL2: offset=0x14C00, size=0x35088, cmdline="--scp-fw"
EL3 Runtime Firmware BL31: offset=0x49E00, size=0xC021, cmdline="--soc-fw"
Secure Payload BL32 (Trusted OS): offset=0x56000, size=0x1C, cmdline="--tos-fw"
Secure Payload BL32 Extra1 (Trusted OS Extra1): offset=0x56200, size=0x97758,
cmdline="--tos-fw-extra1"
Secure Payload BL32 Extra2 (Trusted OS Extra2): offset=0xEDA00, size=0x0,
cmdline="--tos-fw-extra2"
Non-Trusted Firmware BL33: offset=0xEDA00, size=0xF0000, cmdline="--nt-fw"
```

Generic boot

- Boot flows
  - Bootloaders image terminology
- Image organisation
  - Firmware Image Package (FIP)
- **Console API framework**
  - **Log levels**
  - **Crash reporting**
- IO storage abstraction layer
- BL2 image parameters passing
- Locking primitives
- Device tree
  - Firmware Configuration Framework



Console API framework

Every boot loader stage (BLx) **initializes UART driver** to enable console for logging. This is **bare minimum** initialization that every TF-A platform does. It is possible to support **multiple consoles**.

TF-A provides a console API framework consisting of following APIs:

- **console_<driver>_{register/unregister}()**: Main API for any BLx stage to register and unregister console driver. These are **mostly implemented in assembly** as there are places within the firmware where stack may not be readily available. It will map:
  - console_putc() -> console_<driver>_putc()
  - console_getc() -> console_<driver>_getc()
  - console_flush() -> console_<driver>_flush()
- **console_set_scope()**: TF-A support multiple console states depending on execution state: CONSOLE_FLAG_**BOOT**, CONSOLE_FLAG_**RUNTIME** and CONSOLE_FLAG_**CRASH**.

Log levels

At compile time, TF-A provides **options to configure log levels** depending on environment you are compiling TF-A for like debug or development or production. Corresponding build option:

LOG_LEVEL: Chooses the log level, which controls the amount of console log output compiled into the build. Values being:

- 0 (LOG_LEVEL_**NONE**)
- 10 (LOG_LEVEL_**ERROR**)
- 20 (LOG_LEVEL_**NOTICE**)
- 30 (LOG_LEVEL_**WARNING**)
- 40 (LOG_LEVEL_**INFO**)
- 50 (LOG_LEVEL_**VERBOSE**)

All log output up to and including the selected log level is compiled into the build. The default value is **40 in debug builds** (DEBUG=1) and **20 in release builds** (DEBUG=0).

Crash reporting

BL31 implements a scheme for reporting the processor state when an **unhandled exception** is encountered. The reporting mechanism attempts to preserve all the register contents and report it via a **dedicated UART (crash console)**.

A dedicated **per-CPU crash stack** is maintained by BL31 and this is retrieved via the per-CPU pointer cache. The implementation attempts to **minimise the memory** required for this feature. The file **crash_reporting.S** contains the implementation for crash reporting.

Crash reporting - II

Possible reasons for a TF-A crash:

- **Unexpected interrupts**
  - Interrupt is happened on EL3 mode
  - Interrupt is routing into EL3, but no interrupt handling is configured
- **Sync exception**
  - Sync exception, usually are triggered by firmware bug: Data aborts, Instruction aborts, Unalignment fault, etc
  - Unknown SMC Call
- **Async exception**
  - External bus errors
- Software can trigger crash reports by calling **panic()**

Registers included in crash reports

General Purpose Regs

X30
X0 - X29

EL3 Regs

SCR_EL3	DAIF	TTBR0_EL3
SCTLR_EL3	MAIR_EL3	ESR_EL3
CPTR_EL3	SPSR_EL3	FAR_EL3
TCR_EL3	ELR_EL3	

Non EL3 Regs

SPSR_EL1	CPACR_EL1	TCR_EL1	AFSR0_EL1	CTX_CNTKCTL_EL1
ELR_EL1	CSSELR_EL1	TPIDR_EL1	AFSR1_EL1	CTX_FP_FPEXC32_EL2
SPSR_ABT	SP_EL1	TPIDR_EL0	CONTEXTIDR_EL1	SP_EL0
SPSR_UND	ESR_EL1	TPIDRRO_EL0	VBAR_EL1	ISR_EL1
SPSR_IRQ	TTBR0_EL1	DACR32_EL2	CTX_CNTP_CTL_EL0	
SPSR_FIQ	TTBR1_EL1	IFSR32_EL2	CTX_CNTP_CVAL_EL0	
SCTLR_EL1	MAIR_EL1	PAR_EL1	CTX_CNTV_CTL_EL0	
ACTLR_EL1	AMAIR_EL1	MPIDR_EL1	CTX_CNTV_CVAL_EL0	

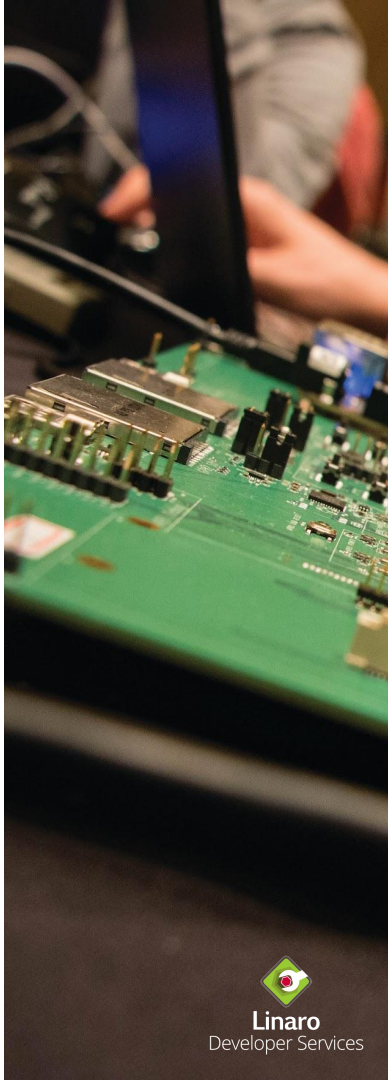
Short break

Based on feedback from previous courses...

... we added this to the middle...

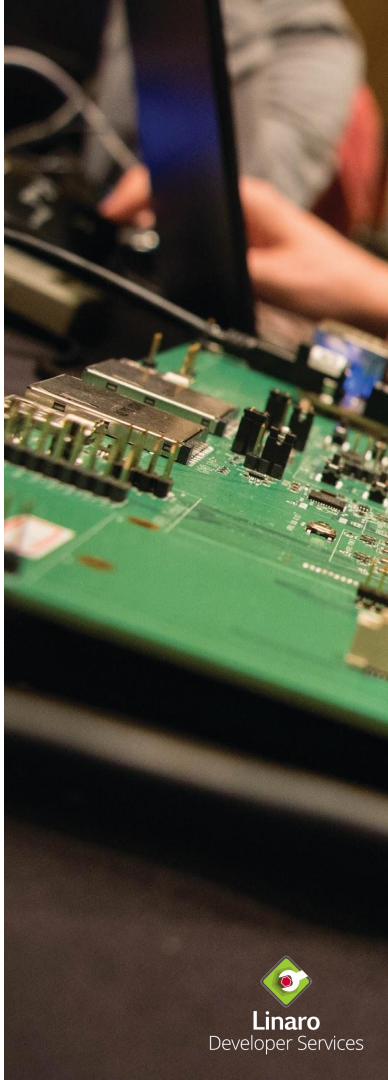
... because it is important ...

... and after two hours it is hard to recollect bits

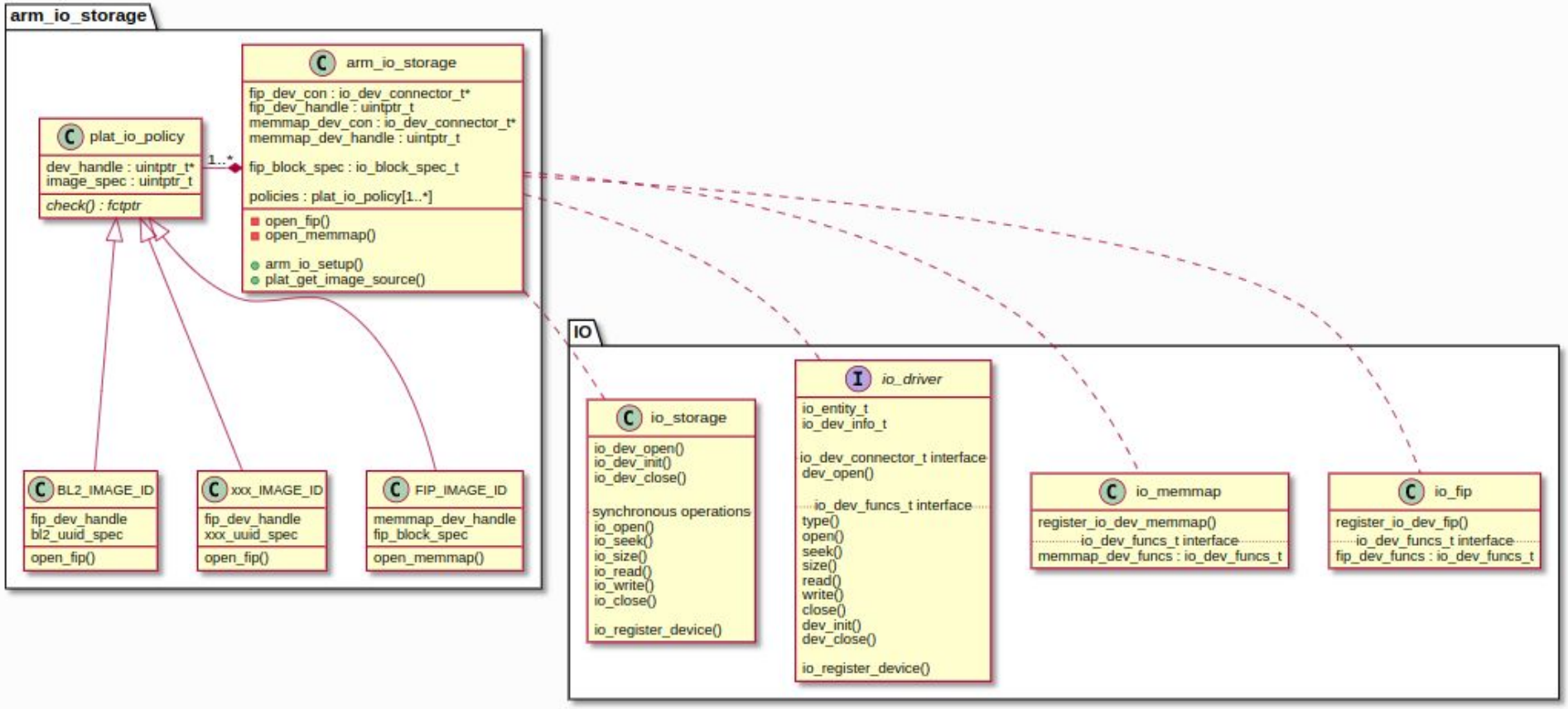


Generic boot

- Boot flows
  - Bootloaders image terminology
- Image organisation
  - Firmware Image Package (FIP)
- Console API framework
  - Log levels
  - Crash reporting
- **IO storage abstraction layer**
- BL2 image parameters passing
- Locking primitives
- Device tree
  - Firmware Configuration Framework



IO storage abstraction layer



IO storage abstraction layer - II

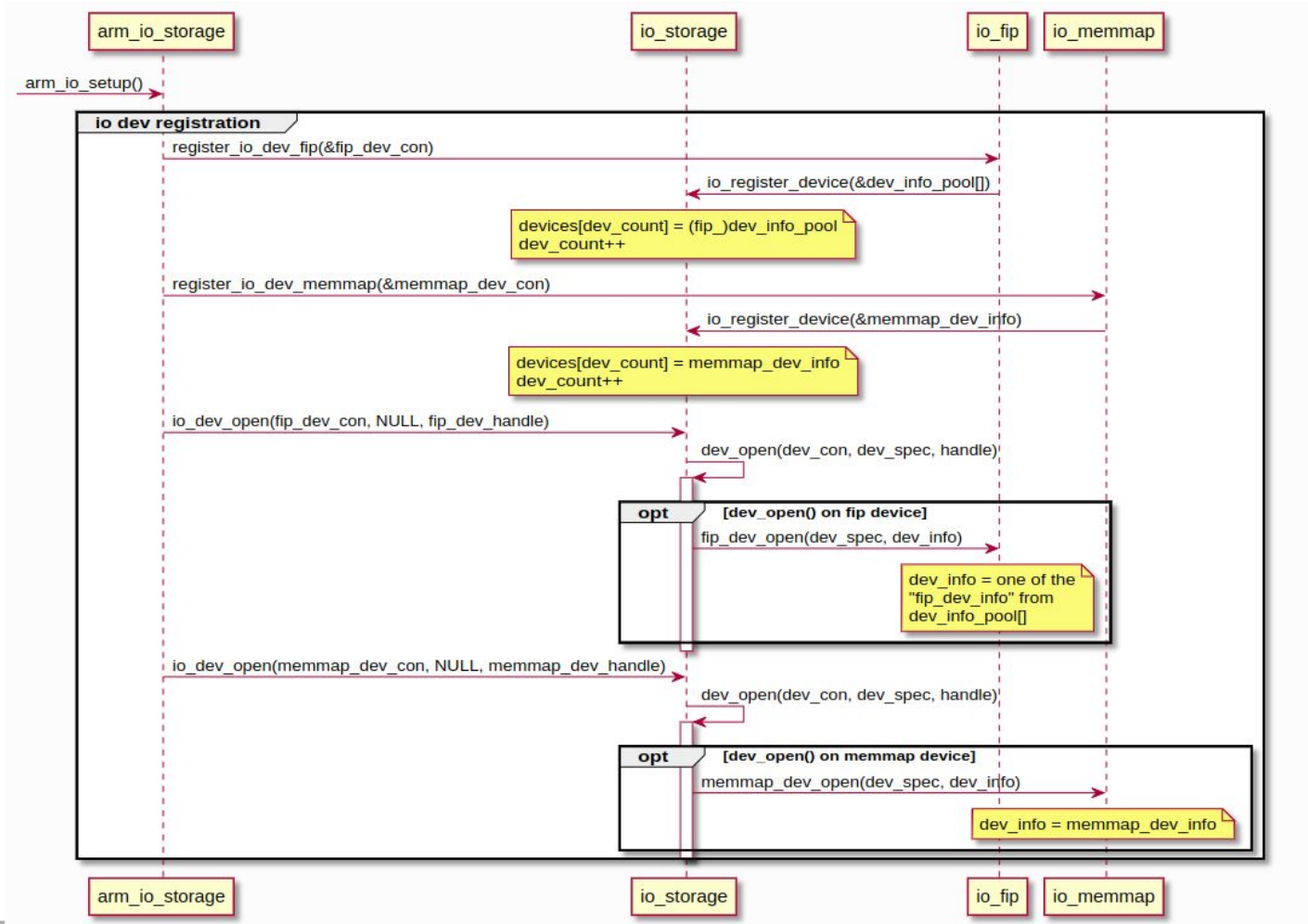
In order to **improve platform independence and portability**, TF-A provides a storage abstraction layer to load data from non-volatile platform storage. Currently storage access is only required by **BL1 and BL2 phases** and performed inside the **load_image()** function in **bl_common.c**.

- Mandatory for platform to implement at least one storage driver
  - Common IO storage library: **`drivers/io/io_storage.c`**
  - IO driver files: **`drivers/io/`**

- Each platform should **register devices and their drivers** via the IO storage abstraction layer. The drivers are initialized in bootloader specific **`blx_platform_setup()`** functions via **`io_dev_init()`** invocation.

- The abstraction layer **supports** `io_open()`, `io_close()`, `io_read()`, `io_write()`, `io_size()` and `io_seek()` APIs.

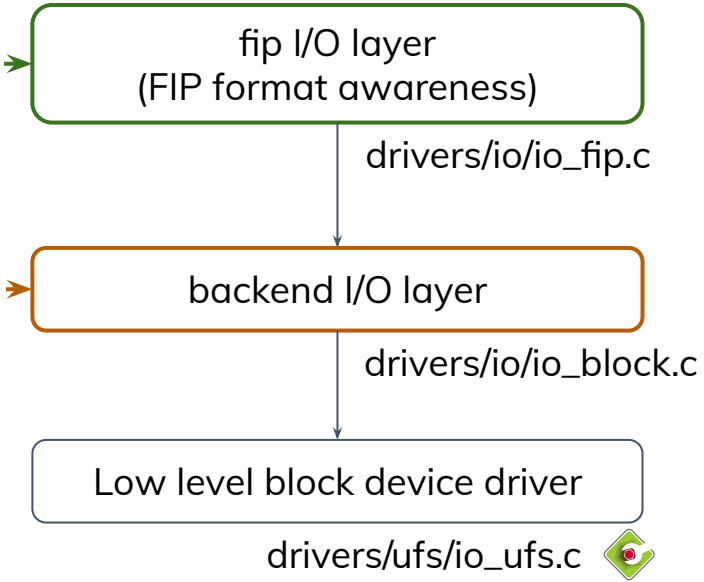
Platform IO storage driver registration



Example IO storage driver policy

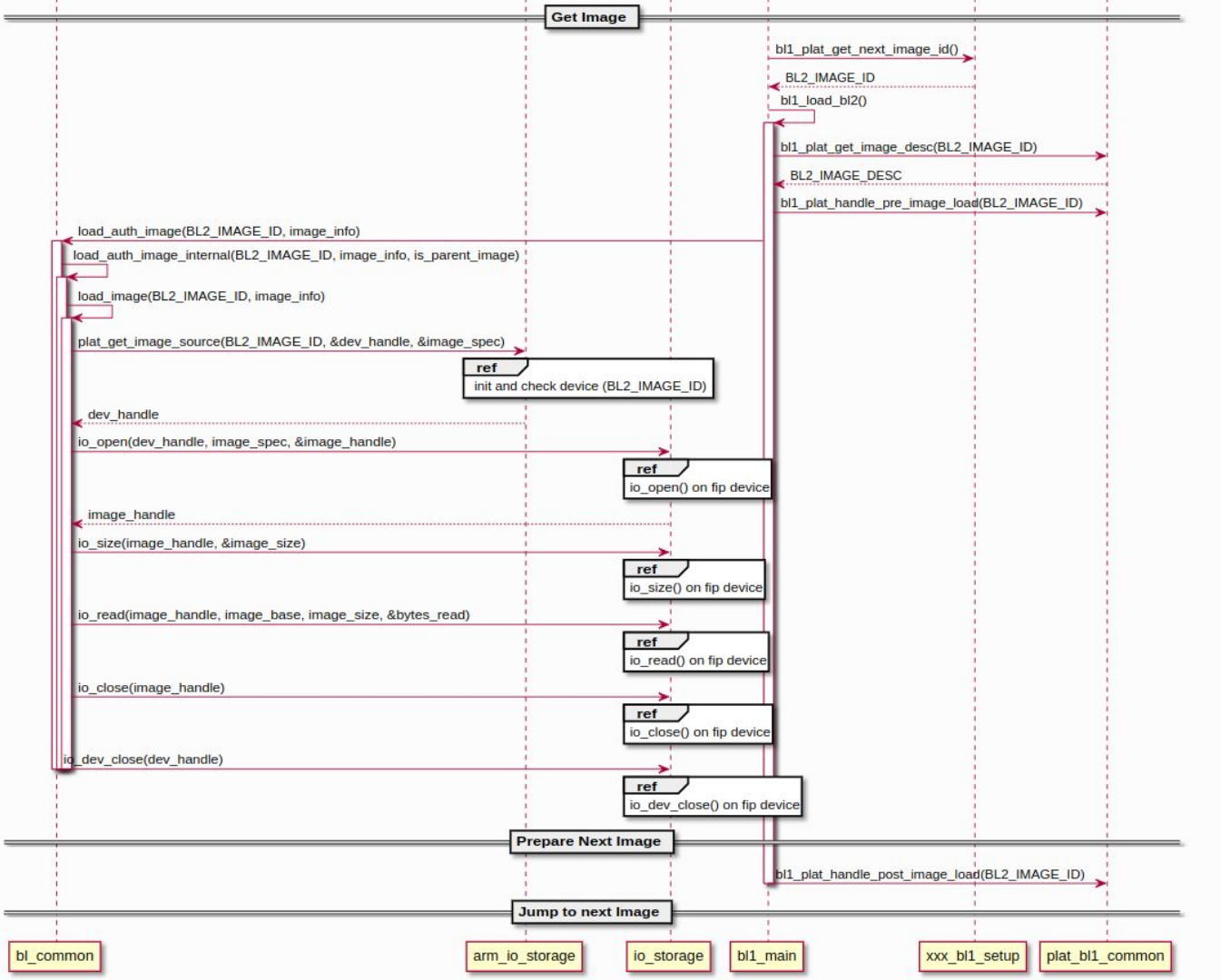
```
static const struct plat_io_policy policies[] = {

        [FIP_IMAGE_ID] = {
                &backend_dev_handle,
                (uintptr_t)&fip_block_spec,
                open_backend
        },

        [BL2_IMAGE_ID] = {
                &fip_dev_handle,
                (uintptr_t)&bl2_uuid_spec,
                open_fip
        },

        [SCP_BL2_IMAGE_ID] = {
                &fip_dev_handle,
                (uintptr_t)&fuse_bl2_uuid_spec,
                open_fip
        },
        …
};
```

Platform storage I/O driver:
plat/nxp/common/setup/ls_io_storage.c



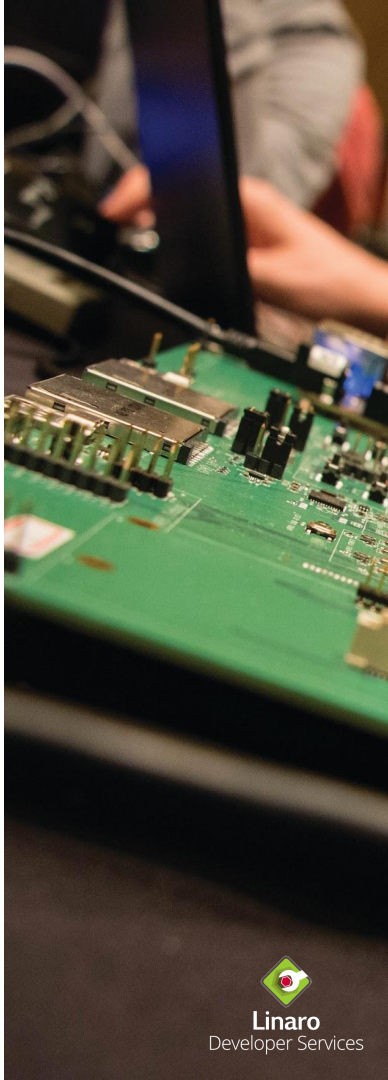
Linaro
Developer Services

Generic image loading flow



Generic boot

- Boot flows
  - Bootloaders image terminology
- Image organisation
  - Firmware Image Package (FIP)
- Console API framework
  - Log levels
  - Crash reporting
- IO storage abstraction layer
- **BL2 image parameters passing**
- Locking primitives
- Device tree
  - Firmware Configuration Framework



BL2 - Parameters for images

bl_mem_params_node_t

`unsigned int image_id`
`image_info_t image_info`
`entry_point_info_t ep_info`
`unsigned int next_handoff_image_id`
`bl_load_info_node_t load_node_mem`
`bl_params_node_t params_node_mem`

Predefined images information for platform specific, it gives out every image's ID, image base address and size, jumping configurations, and the linked info cross multiple images.

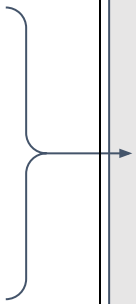
→ Image node on the loading list.

→ Image node on the parameter list, which is passed from BL2 to BL31.

BL2 - Parameters for images - II

bl_mem_params_node_t

unsigned int image_id
image_info_t image_info
entry_point_info_t ep_info
unsigned int next_handoff_image_id
bl_load_info_node_t load_node_mem
bl_params_node_t params_node_mem



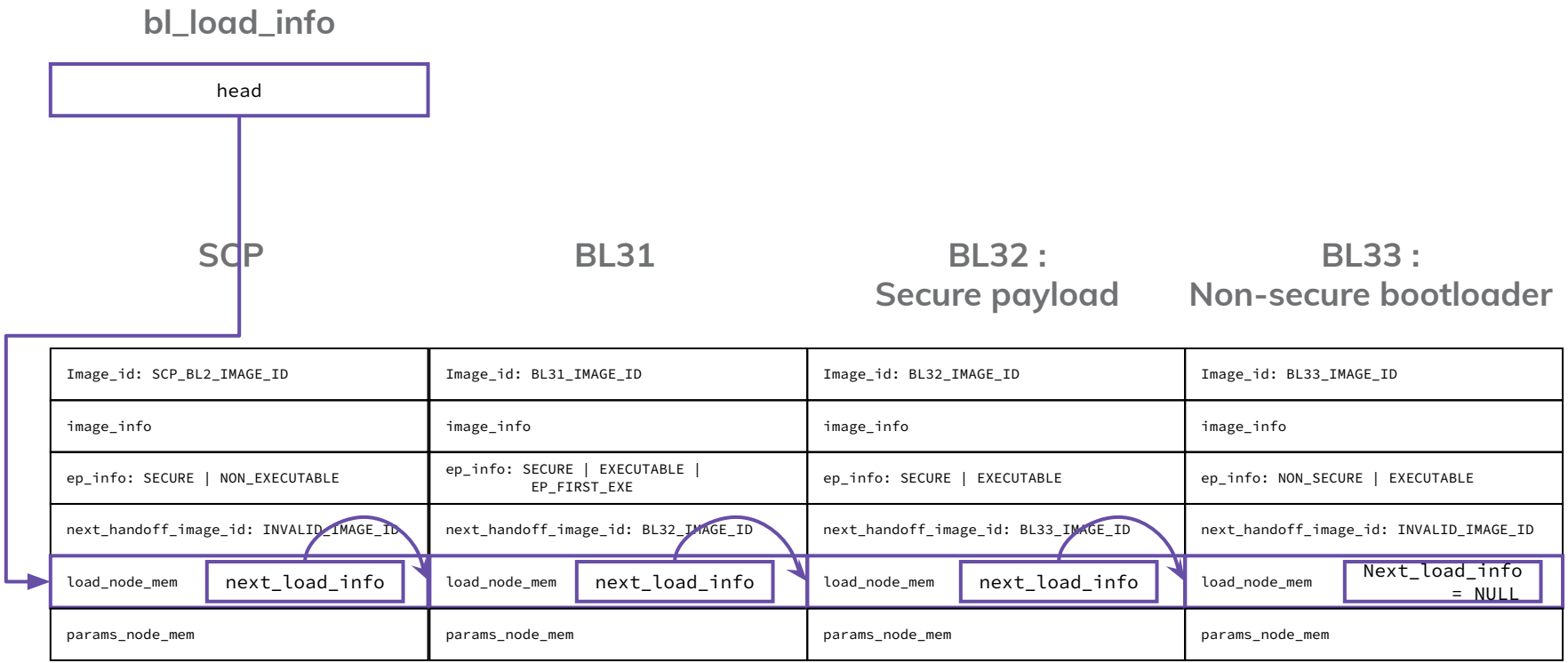
plat/hisilicon/hikey960/hikey960_bl2_mem_params_desc.c

```
static bl_mem_params_node_t bl2_mem_params_descs[] = {

#ifdef SCP_BL2_BASE
        /* Fill SCP_BL2 related information if it exists */
    {
            .image_id = SCP_BL2_IMAGE_ID,

            SET_STATIC_PARAM_HEAD(ep_info, PARAM_IMAGE_BINARY,
                    VERSION_2, entry_point_info_t,
                    SECURE | NON_EXECUTABLE),

            SET_STATIC_PARAM_HEAD(image_info, PARAM_IMAGE_BINARY,
                    VERSION_2, image_info_t, IMAGE_ATTRIB_PLAT_SETUP),
            .image_info.image_base = SCP_BL2_BASE,
            .image_info.image_max_size = SCP_BL2_SIZE,

            .next_handoff_image_id = INVALID_IMAGE_ID,
    },
#endif /* SCP_BL2_BASE */

…
};


REGISTER_BL_IMAGE_DESCS(bl2_mem_params_descs)
```

BL2 - Platform predefined image information

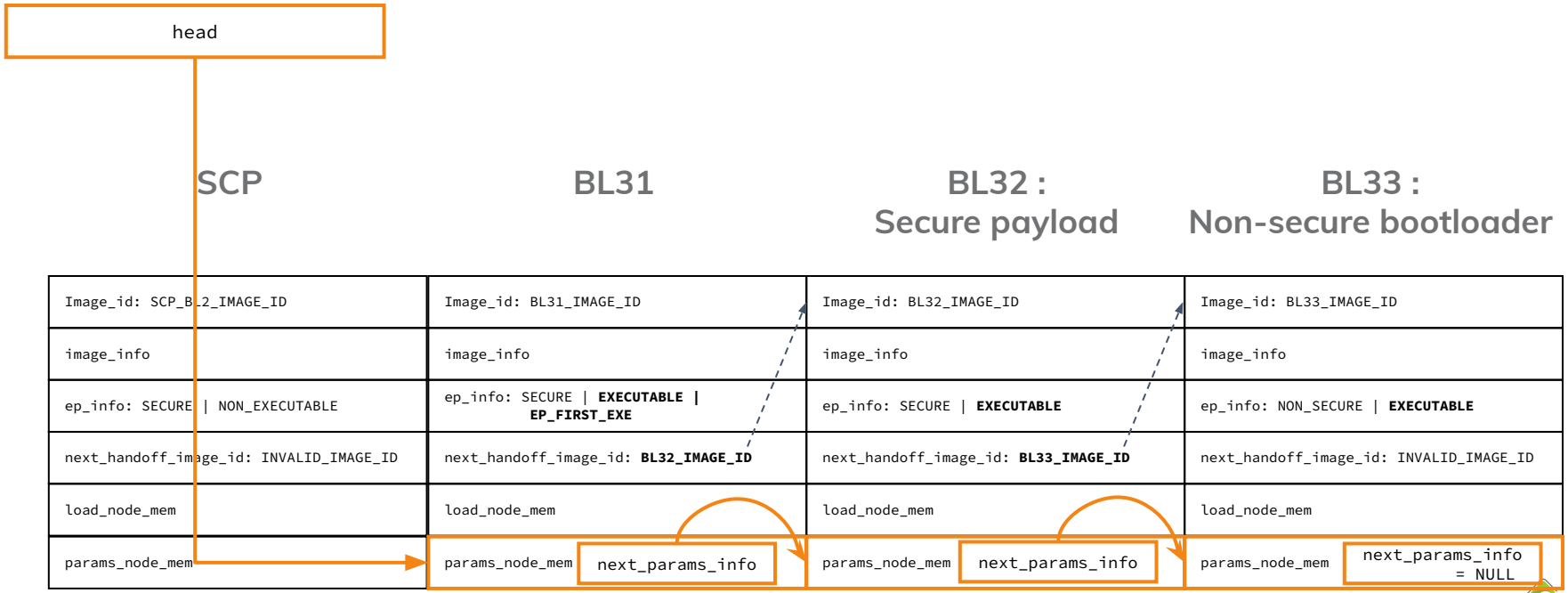
SCP	BL31	BL32 : Secure payload	BL33 : Non-secure bootloader
Image_id: SCP_BL2_IMAGE_ID	Image_id: BL31_IMAGE_ID	Image_id: BL32_IMAGE_ID	Image_id: BL33_IMAGE_ID
image_info	image_info	image_info	image_info
ep_info: SECURE | NON_EXECUTABLE	ep_info: SECURE | EXECUTABLE | EP_FIRST_EXE	ep_info: SECURE | EXECUTABLE	ep_info: NON_SECURE | EXECUTABLE
next_handoff_image_id: INVALID_IMAGE_ID	next_handoff_image_id: BL32_IMAGE_ID	next_handoff_image_id: BL33_IMAGE_ID	next_handoff_image_id: INVALID_IMAGE_ID
load_node_mem	load_node_mem	load_node_mem	load_node_mem
params_node_mem	params_node_mem	params_node_mem	params_node_mem

BL2 - Load image list



BL2 - Parameter list

bl2_to_next_bl_params



BL2 - Pass parameter list to BL31

bl2_to_next_bl_params

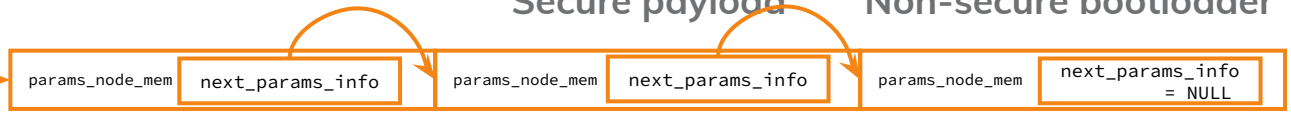
Pass pointer to BL3 through arg0 (x0)

```
struct entry_point_info *bl2_load_images(void)
{
        …

        /* Populate arg0 for the next BL image if not already provided */
        if (bl2_to_next_bl_params->head->ep_info->args.arg0 ==
        (u_register_t)0)
                bl2_to_next_bl_params->head->ep_info->args.arg0 =
                                        (u_register_t)bl2_to_next_bl_params;

        plat_flush_next_bl_params();
        …
}
```

BL31

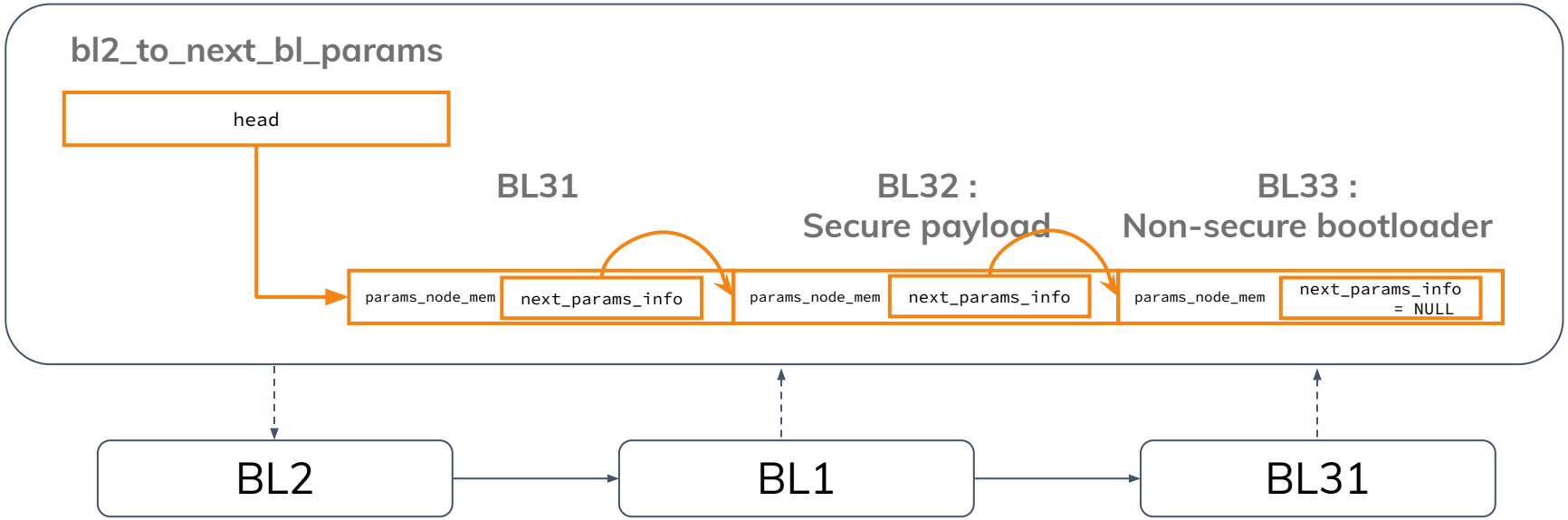
BL32 : Secure payload

BL33 : Non-secure bootloader



Linaro
Developer Services

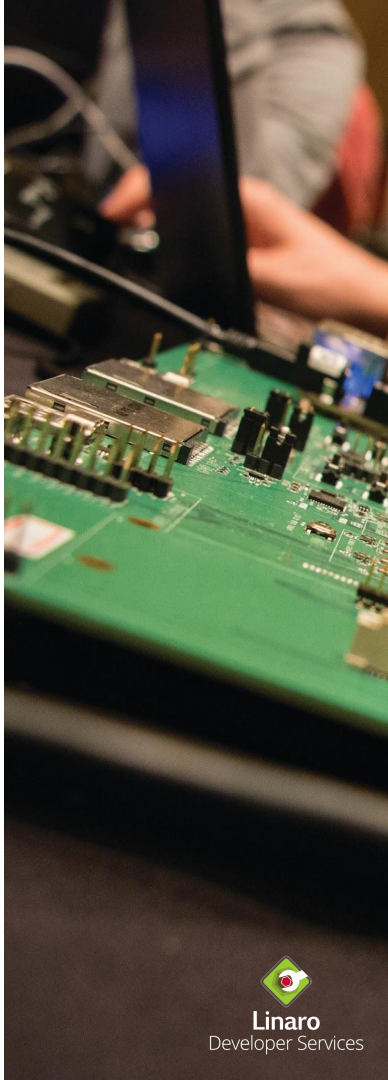
Aside: requirement for identical memory mapping



The parameter list is compound in BL2 but it's shared and accessed by BL1 (for printing log) and BL31; for easier accessing the list without converting address space in BL1/BL31, it's required to use identical memory mapping in trusted firmware.

Generic boot

- Boot flows
  - Bootloaders image terminology
- Image organisation
  - Firmware Image Package (FIP)
- Console API framework
  - Log levels
  - Crash reporting
- IO storage abstraction layer
- BL2 image parameter passing
- **Locking primitives**
- Devicetree
  - Firmware Configuration Framework



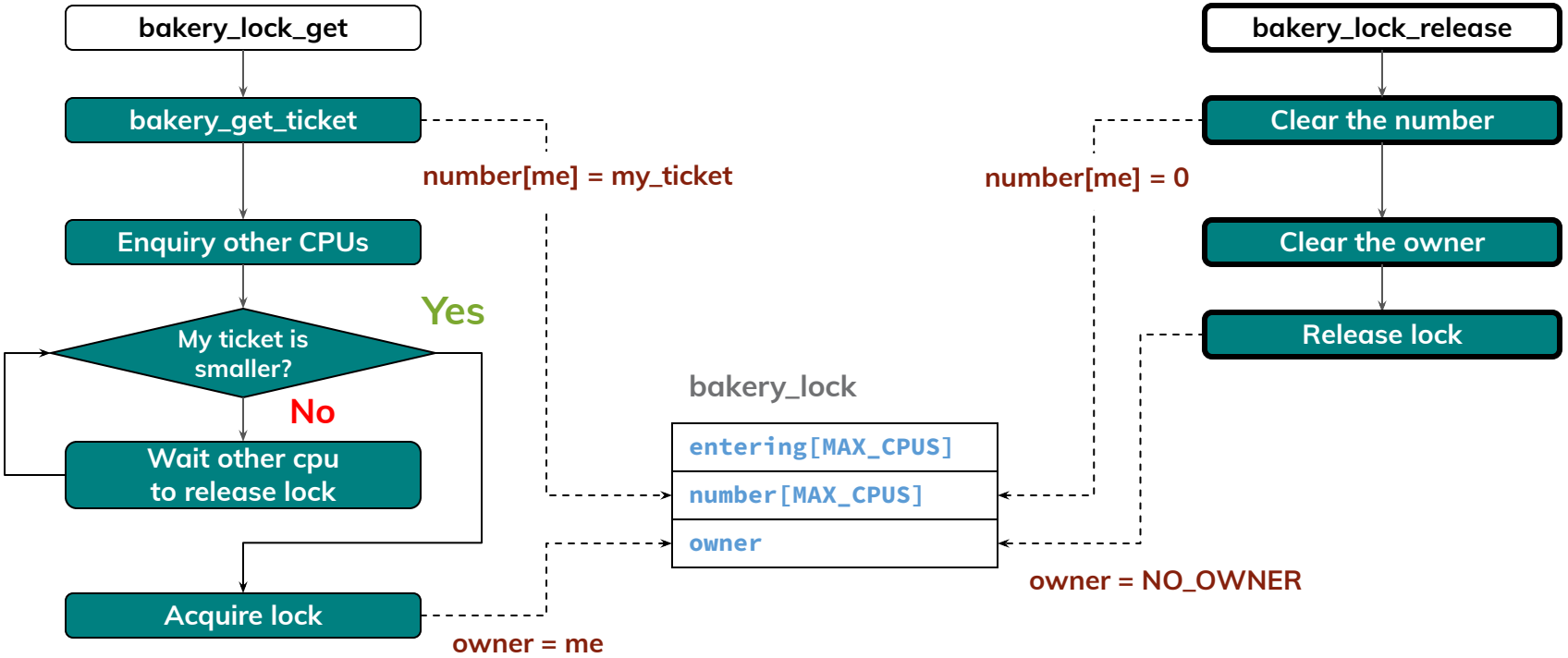
Locking in TF-A?

- **Why locking** is needed in TF-A?
  - Locking required for CPU affinity structure in PSCI core layer
  - SoC specific device driver for exclusive accessing registers
- **Cannot use ldrex/strex** instructions, why?
  - Usually in firmware, some CPUs are in coherency domain and other CPUs are out of coherency domain (In kernel, all CPUs are in coherency domain!)
  - ldrex/strex depends on **exclusive monitor**. If cpu has exited from cache coherency domain then it needs DDR controller to support exclusive monitor. But DDR controller commonly doesn't have exclusive monitor.
- **Locking for CPU's affinity structure**
  - Affinity level 0: uses spin lock to protect CPU on / off in case
  - Affinity level 1: Every cluster affinity structure has one lock
  - Affinity level 2: have a global lock for whole system
  - No reverse locking when acquire multiple locks

Locking primitives

- **Use software locking** algorithm
  - Many mature algorithms (voting lock, bakery lock)
- **Locking interfaces**
  - `void` **`bakery_lock_init`**`(bakery_lock_t *bakery);`
  - `void` **`bakery_lock_get`**`(bakery_lock_t *bakery);`
  - `void` **`bakery_lock_release`**`(bakery_lock_t *bakery);`
- The locking structure itself can be placed into **coherent/normal** memory region
  - It's decided by the platform build configuration **`USE_COHERENT_MEM`**
  - When the locking structure is placed in the normal memory, the locking operations perform **software cache maintenance** on the lock data structure

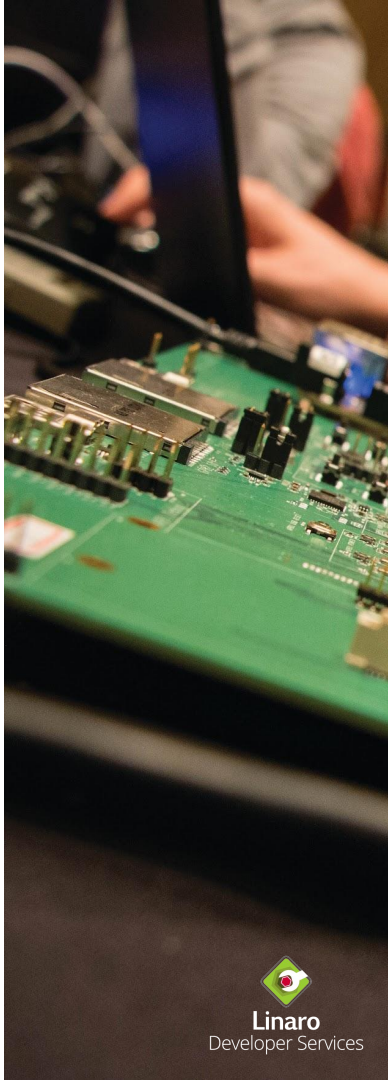
Bakery locking algorithm



Linaro
Developer Services

Generic boot

- Boot flows
  - Bootloaders image terminology
- Image organisation
  - Firmware Image Package (FIP)
- Console API framework
  - Log levels
  - Crash reporting
- IO storage abstraction layer
- BL2 image parameter passing
- Locking primitives
- **Devicetree**
  - **Firmware Configuration Framework**



Devicetree

A devicetree (DT) is a **tree data structure** with nodes that describe the devices in a system. Each node has **property/value pairs** that describe the characteristics of the device being represented. Each node has exactly one parent except for the root node, which has no parent.

The major usage of DT within TF-A is to **hold platform specific** firmware configuration data. Traditionally, all that platform data used to be defined **statically** in the corresponding firmware. **Firmware CONfiguration Framework (FCONF)** allows to provide an abstraction layer via DT for platform specific data.

Firmware CONfiguration Framework (FCONF)

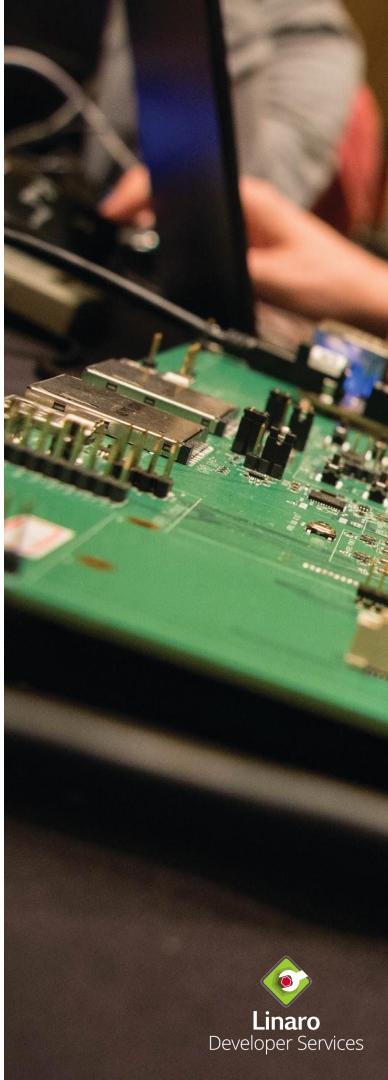
FCONF is an abstraction layer for platform specific data, allowing a **“property”** to be queried and a value retrieved **without** the requesting entity knowing what backing store is being used to hold the data.

It is used to bridge new and old ways of providing platform-specific data such as:

- (TBBR) **Chain of Trust data**: tbbr.cot.trusted_boot_fw_cert
- (TBBR) **dynamic configuration info**: tbbr.dyn_config.disable_auth
- **Arm io policies**: arm.io_policies.bl2_image
- **GICv3 properties**: hw_config.gicv3_config.gicr_base

The FCONF properties are **loaded from devicetree blob (DTB)** which itself is loaded from FIP. So the **IO layer** must be initialized prior to accessing FCONF properties. **Arm and STM32 platforms** already have support for FCONF properties.

Lab sessions



Reminder: Preparation and boot

Preparatory steps remains similar as for TFA-01 lab sessions.

Boot cmdline:

```
qemu-system-aarch64 \
    -machine virt,secure=on -cpu cortex-a57 \
    -smp 4 -nographic -m 1G -bios flash.bin \
    -drive \
     file=./core-image-tfa-qemuarm64-secureboot.wic.qcow2,if=virtio,format=qcow2 \
    -netdev user,id=eth0,hostfwd=tcp::2222-:22 \
    -device virtio-net-device,netdev=eth0
```

Don’t forget to source the SDK environment setup script!

LAB1 - Explore TF-A boot flow

Build TF-A from source code

```
$ . /poky/sdk/path/environment-setup-cortexa57-poky-linux

$ git clone https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git
$ cd trusted-firmware-a
$ git checkout -b v2.9.0 v2.9.0

$ LDFLAGS= make CFLAGS= PLAT=qemu DEBUG=1 \
        BL33=$SDKTARGETSYSROOT/boot/u-boot.bin BL32_RAM_LOCATION=tdram \
        all fip
$ dd if=build/qemu/debug/bl1.bin of=flash-src.bin bs=4096 conv=notrunc
$ dd if=build/qemu/debug/fip.bin of=flash-src.bin seek=64 bs=4096 conv=notrunc
```

Attach GDB during TF-A boot

Launch Qemu with cmdline:

```
qemu-system-aarch64 \
    -machine virt,secure=on -cpu cortex-a57 \
    -smp 2 -nographic -m 1G -bios trusted-firmware-a/flash-src.bin \
    -drive \
     file=./core-image-tfa-qemuarm64-secureboot.wic.qcow2,if=virtio,format=qcow2 \
    -netdev user,id=eth0,hostfwd=tcp::2222-:22 \
    -device virtio-net-device,netdev=eth0 \
    -gdb tcp::1234 -S
```

"**-gdb**" option: Accept gdb connection over "**tcp**" port no. "**1234**". QEMU defaults to starting the guest without waiting for gdb to connect; use "**-S**" too if you want it to not start execution.

"**-S**" option: Freeze CPU at startup (use 'c' to start execution).

Attach debugger during TF-A boot

Launch GDB in another terminal:

```
$ . /poky/sdk/path/environment-setup-cortexa57-poky-linux
$ cd trusted-firmware-a
$ aarch64-poky-linux-gdb
```

GDB setup commands:

```
(gdb) target remote localhost:1234
(gdb) add-symbol-file build/qemu/debug/bl1/bl1.elf
(gdb) add-symbol-file build/qemu/debug/bl2/bl2.elf
(gdb) add-symbol-file build/qemu/debug/bl31/bl31.elf
```

Explore TF-A boot flow: GDB breakpoints

Key TF-A boot flow breakpoints:

```
(gdb) b bl1_main
(gdb) b bl1_load_bl2
(gdb) b bl1_prepare_next_image
(gdb) b bl2_main
(gdb) b bl2_load_images
(gdb) hbreak bl31_main
(gdb) hbreak runtime_svc_init
(gdb) hbreak
bl31_prepare_next_image_entry
```

Boot up by writing “continue” in GDB:

```
(gdb) c <enter>
```

Use additional gdb commands to explore further.

```
print <expression>, info locals
backtrace
step, next, finish
list, display <expression>
```

If you step over something interesting and miss the details then you can always reboot, set a new breakpoint and take a closer look!

LAB2 - Build and analyze your own FIP

Use fiptool explicitly to create fip.bin

The TF-A makefile option: “`fip`” **implicitly** builds a “`fip.bin`” using fiptool. Now you have to build “`fip.bin`” **manually**.

Some **hints**:

- fiptool is present here:
  - `<trusted-firmware-a clone path>/tools/fiptool/fiptool`
- BL1 image
  - `<trusted-firmware-a clone path>/build/qemu/debug/bl1.bin`
- BL2 image
  - `<trusted-firmware-a clone path>/build/qemu/debug/bl2.bin`
- BL31 image
  - `<trusted-firmware-a clone path>/build/qemu/debug/bl31.bin`
- BL33 image
  - `$SDKTARGETSYSROOT/boot/u-boot.bin`

Boot and analyze your own fip.bin

Create “`flash-src.bin`” using your own “`fip.bin`” and see if it boots successfully.

`fiptool` is very handy tool to **analyze and update** contents of “`fip.bin`”. Try out following commands:

- `./tools/fiptool/fiptool` **`info`**
- `./tools/fiptool/fiptool` **`update`**
- `./tools/fiptool/fiptool` **`unpack`**
- `./tools/fiptool/fiptool` **`remove`**

Thank you

support@linaro.org

Linaro
Developer Services