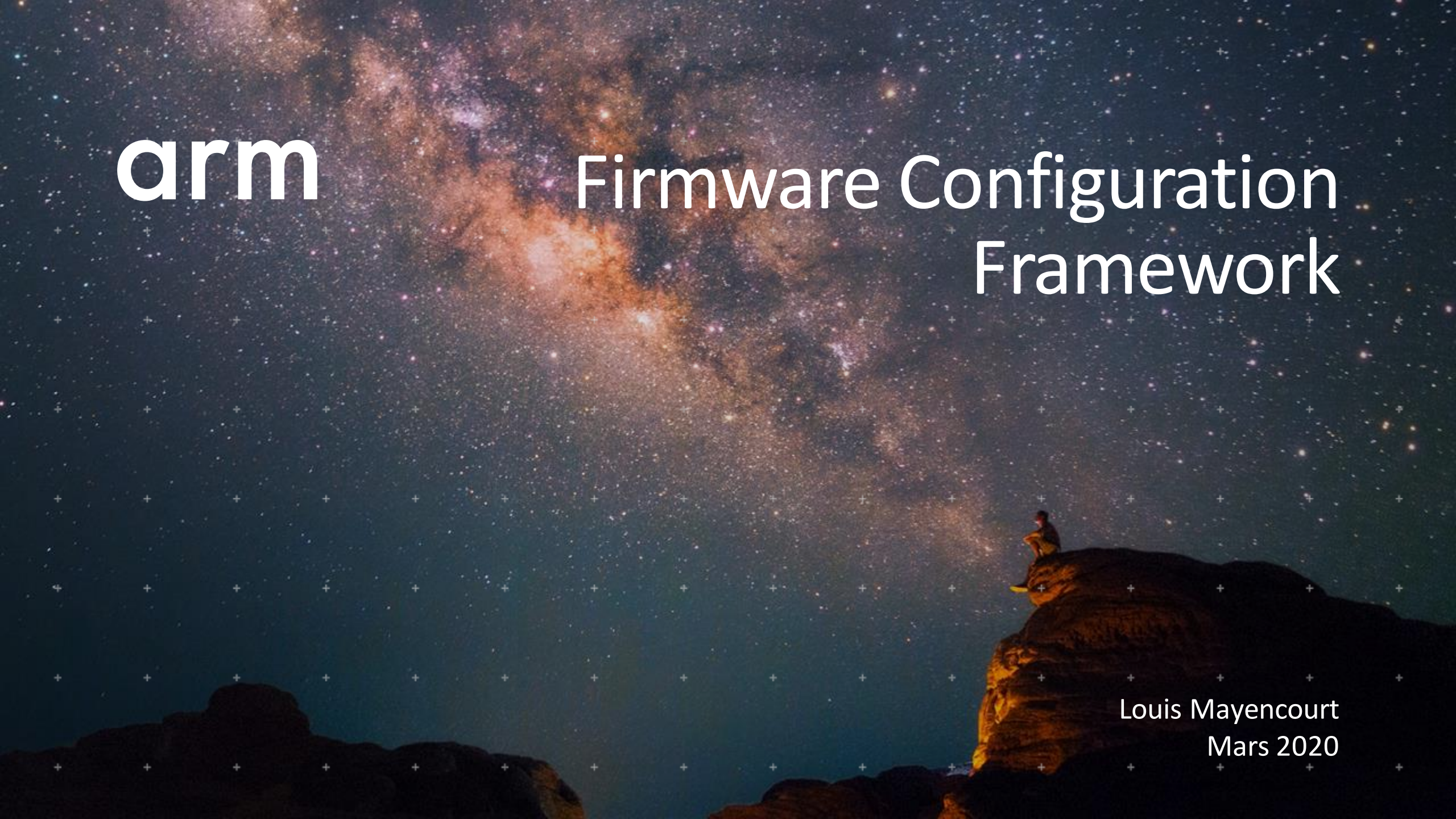


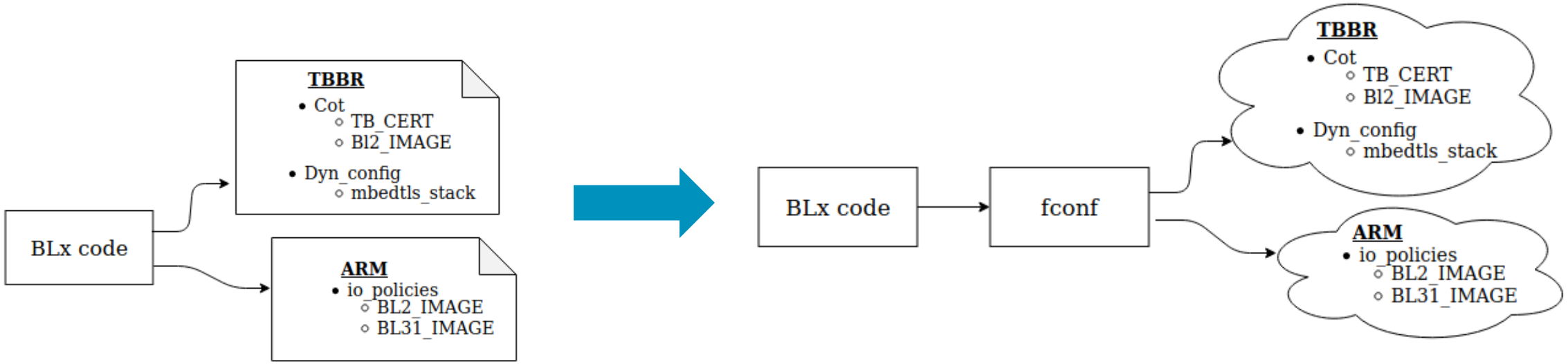
Firmware Configuration Framework

Louis Mayencourt

Mars 2020

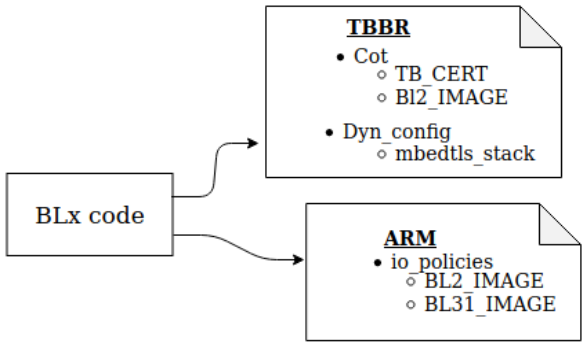
Idea

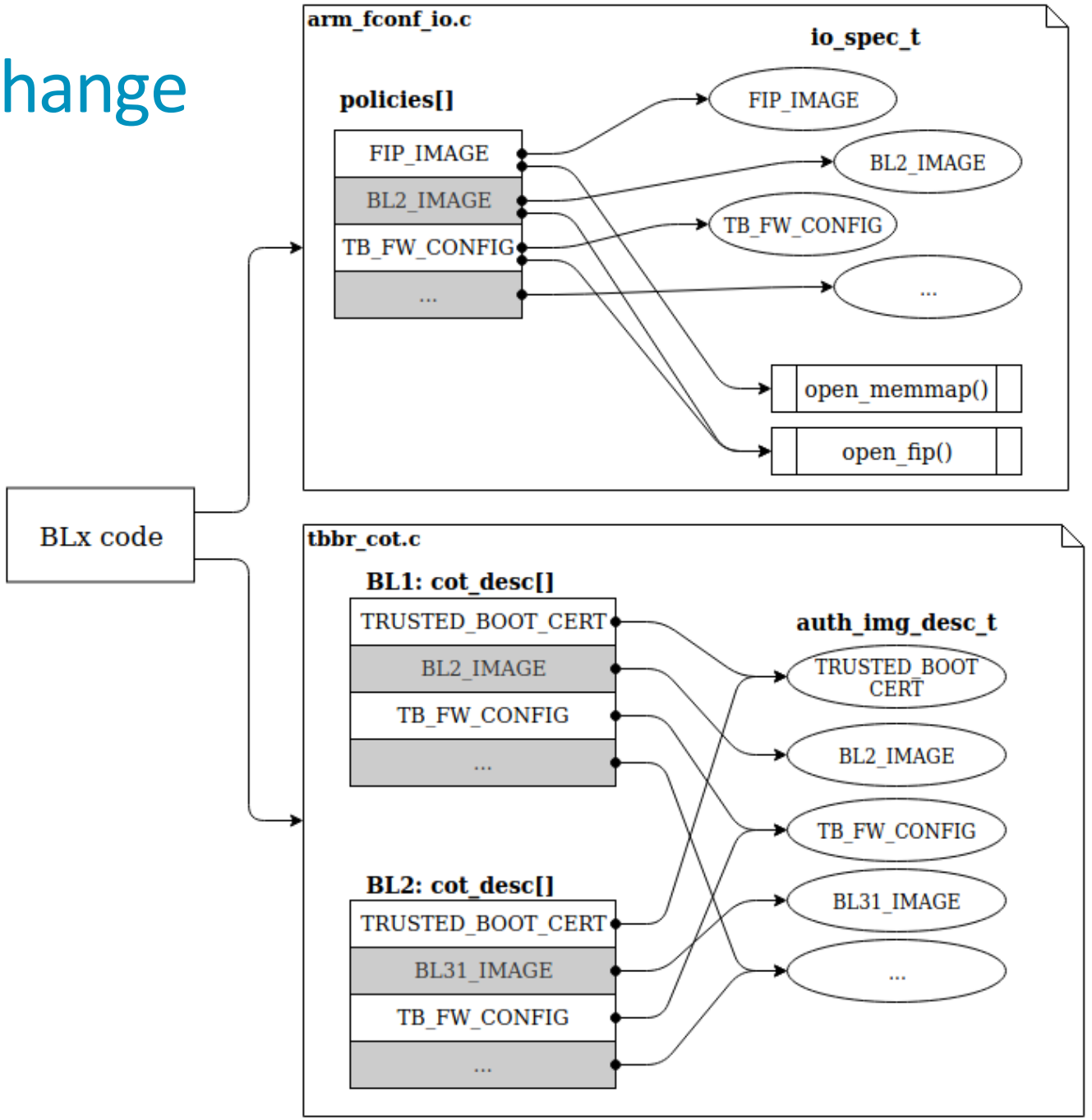
- Use a "data abstraction layer" to access the configuration data (io policies, chain of trust, uart, ...) into different data-representation. The API should allow the BLx images to access data by "key" like get_parameter(key).



What we want to change

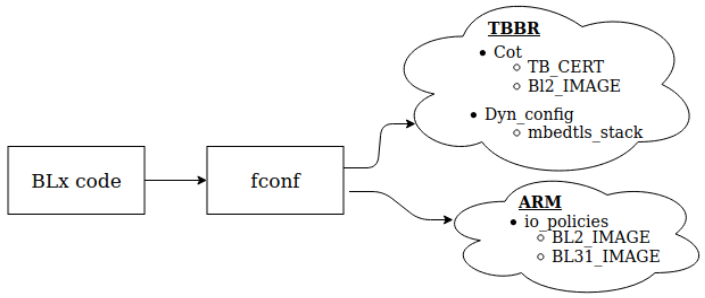
- Code have direct acccess to data structures.





Proposed design

- Data can be defined in the code or in DTB.



Design considerations

- C data struct is the default data representation.
  - Keep backward compatibility with current implementation.
  - Device tree need a storage anyway.
- Access to data must be efficient and safe.
  - Use a macro allows build-time checks and no runtime penalty.
    - The code is however more difficult to follow.
- Data structure are filled once with a *populate()* function.
  - Based on subscriber-publisher pattern.
    - Use linker to register publisher at build-time.
  - Function called by common code.
- BL1 data must be defined at build-time.
  - CoT and io_policies are required to load config file.
- BL2/BL31 can use dynamic configuration and build-in data.

Interfaces

- Data acces (and definition):
  - *FCONF_GET_PROPERTY(a, b, c)*
- Load the configuration DTB:
  - *fconf_load_config()*
- Data population:
  - *fconf_populate(const char *config_name, uintptr_t config)*
  - *FCONF_REGISTER_POPULATOR(config, name, callback)*

Interfaces

- Data acces and definition:
  - *FCONF_GET_PROPERTY(namespace, sub-namespace, property)*
    - 2 levels macro to access property c, from "namespace" a and "sub-namespace" b
    - Example: tbbr.cot.bl2, arm.io_policies.bl31, arm.uart0.baudrate

Include/lib/fconf/fconf.h

```
// Common getter
#define FCONF_GET_PROPERTY(a,b,c)    a##__##b##_getter(c)
```

Include/lib/fconf/fconf_tbbr_getter.h

```
// Keep backward compatibility
#define tbbr__cot_getter(id)    cot_desc_ptr[id]


// redirection to a property structure
#define tbbr__dyn_config_getter(id)    tbbr_dyn_config.id

struct tbbr_dyn_config_t {
   uint32_t disable_auth;
   uintptr_t mbedtls_heap_addr;
   size_t mbedtls_heap_size;
} tbbr_dyn_config;
```

Plat/arm/board/fvp/include/fconf_getter.h

```
// Platform specific getter
#define arm__io_policies_getter(id)    &policies[id]

struct plat_io_policy {
   uintptr_t *dev_handle;
   uintptr_t image_spec;
   int (*check)(const uintptr_t spec);
};

extern struct plat_io_policy policies[];
```


arm

Interfaces

- Load the configuration DTB:
  - *fconf_load_config()*
    - Used in BL1 to load *fw_config.dtb*
  - Information in *fw_config* can be used to load more configuration DTBs:
    - TOS_FW config
    - HW config
    - ...

Include/lib/fconf/fconf.h

```
void fconf_load_config();
```

Plat/arm/board/fvp/fdts/fvp_fw_config.dts

```
// config index
dtb-registry {
    compatible = "arm,dyn_cfg-dtb_registry";
    tos_fw-config {
        load-address = <0x0 0x4001200>;
        max-size = <0x200>;
        id = <TOS_FW_CONFIG_ID>;
    };
    hw-config {
        load-address = <0x0 0x82000000>;
        max-size = <0x01000000>;
        id = <HW_CONFIG_ID>;
    };

// Some properties
 arm-io_policies {
            fip-handles {
                compatible = "arm,io-fip-handle";
                scp_bl2_uuid = <0x3dfd6697 0x49e8be89 0xa1785dae 0x13826040>;
                bl31_uuid = <0x6d08d447 0x4698fe4c 0x5029959b 0x005abdcb>;
                ....
             };
  };
```

Interfaces

- Data population:
  - *fconf_populate(const char *config_type, uintptr_t config)*
    - Called once in BL2 and BL31 entry.
    - Call every registered "*populator*" with a matching "*type*", usually a dtb name: TB_FW, HW_CONFIG,…
  - *FCONF_REGISTER_POPULATOR(config, name, callback)*
    - Used to register common and platform specific "populator" function

Include/lib/fconf/fconf.h

```
#define FCONF_REGISTER_POPULATOR(config, name, callback)
  __attribute__((used,section(".fconf_populator")))          \
  const struct fconf_populator name##__populator = {         \
     .config_type = #config,                                 \
     .info = #name,                                          \
     .populate = callback                                    \
  };


// Call registered populator for a specific config_type
void fconf_populate(const char *config_type, uintptr_t config);
```

lib/fconf/fconf_tbbr_getter.c

```
int fconf_populate_tbbr_dyn_config(uintptr_t config) {
   /* read tb_fw config dtb and fill tbbr_dyn_config struct */
}
FCONF_REGISTER_POPULATOR(TB_FW, tbbr, fconf_populate_tbbr_dyn_config)
```

Plat/arm/common/fconf/arm_io_getter.c

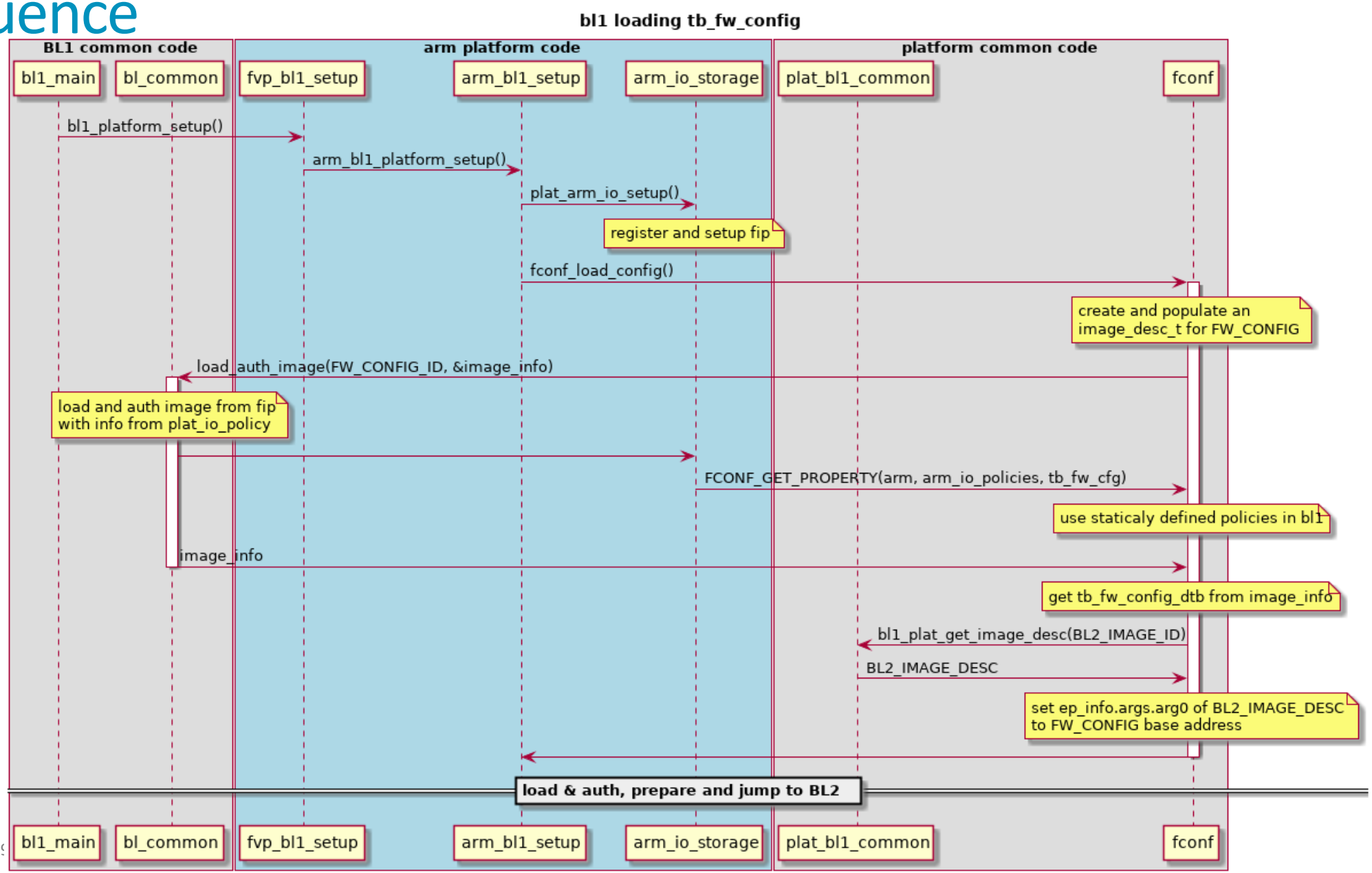
```
int fconf_populate_arm_io_policies(uintptr_t config) { … }
FCONF_REGISTER_POPULATOR(TB_FW, arm_io, fconf_populate_arm_io_policies)
```

Plat/arm/board/fvp/fconf/fconf_hw_config_getter.c

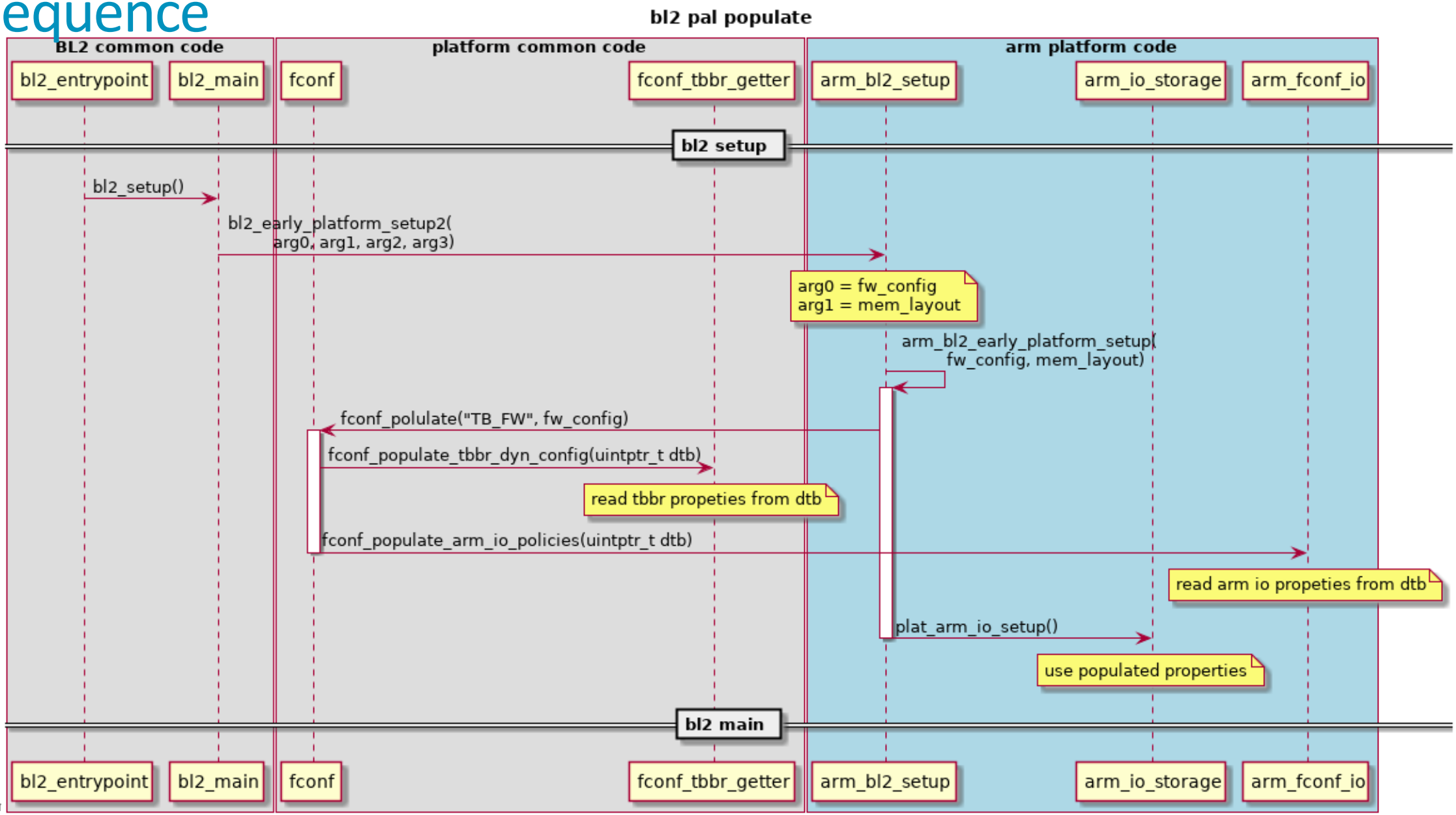
```
int fconf_populate_topology(uintptr_t config){...}
FCONF_REGISTER_POPULATOR(HW_CONFIG, topology, fconf_populate_topology);
```


arm

BL1 sequence



BL2 sequence



Namespace

- *FCONF_GET_PROPERTY*(**namespsace**, subnamespace, property)
- Namespace divided in two categories
  - Common code properties
    - TBBR
    - Dynamic configuration
    - …

    -> Can be called from everywhere.
  - Platform properties
    - Arm
    - Nvidia
    - Qemu
    - …

    -> Can only be called from platform code !!
- Subnamespace can be used freely

Features using fconf:

- Dynamic configuration
- Arm io policies
- Hardware topology extracted from HW_CONFIG
- Measured boot
- And more to come !

Possible improvement:

- Implement a FCONF_*SETTER* a mechanism
- Only use *fw_config* in Blx hand-off
  - Prototype almost ready
- Chain of trust in DTB
- ...

arm

For more info:

https://trustedfirmware-a.readthedocs.io/en/latest/components/fconf.html

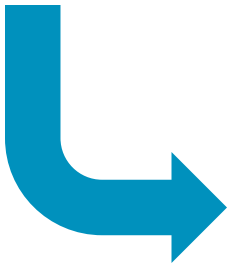
arm

Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
תודה

2. Configs Hand-Off

	BL1 -> BL2	BL2 -> BL31	BL31 -> BL33
arg0	TB_FW_CONFIG	bl_params_t	NT_FW_CONFIG
arg1	mem_info	SOC_FW_CONFIG	HW_CONFIG
arg2	HW_CONFIG	HW_CONFIG	-
arg3	-	Magic number	-

- Each config uses 1 arg.
  - Not scalable
  - No unified hand-off between different blx images
- No consistent use of the register between different blx images.



- First step:
  - Only provide FW_CONFIG
    - Extract other configs base address from FW_CONFIG.
    - No configs number limitations.
    - Unified behavior.

	BL1 -> BL2	BL2 -> BL31	BL31 -> BL33
arg0	FW_CONFIG	bl_params_t	NT_FW_CONFIG
arg1	mem_info	FW_CONFIG	HW_CONFIG
arg2	-	-	-
arg3	-	Magic number	-

arm

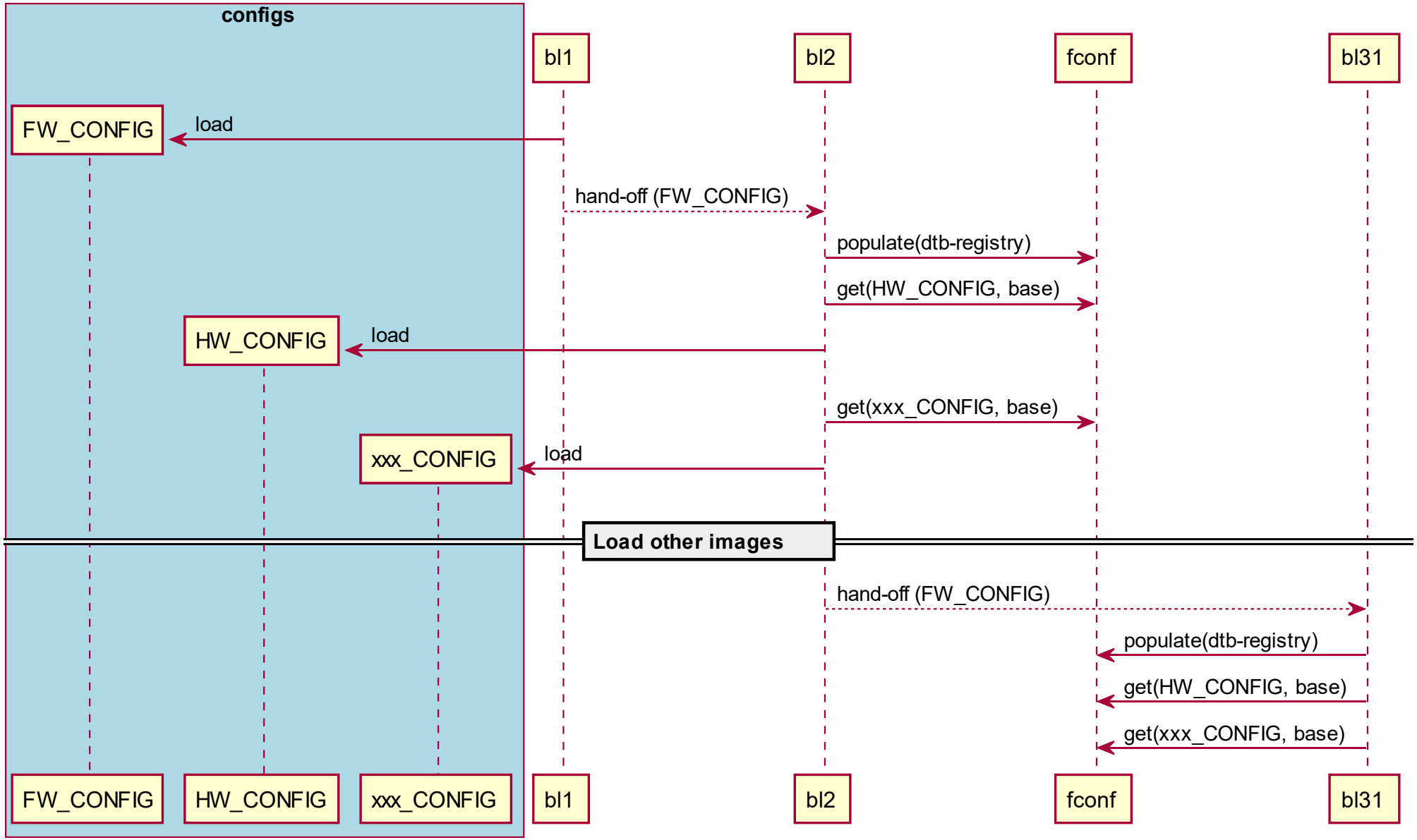
2. Configs Hand-Off (2)

Information are already in FW_CONFIG dtb !

```
dtb-registry {
    compatible = "arm,dyn_cfg-dtb_registry";

    tb_fw-config {
        load-address = <0x0 0x4001010>;
        max-size = <0x200>;
        id = <TB_FW_CONFIG_ID>;
    };
    hw-config {
        load-address = <0x0 0x82000000>;
        max-size = <0x01000000>;
        id = <HW_CONFIG_ID>;
    };
    soc_fw-config {
        load-address = <0x0 0x04001000>;
        max-size = <0x200>;
        id = <SOC_FW_CONFIG_ID>;
    };
    tos_fw-config {
        load-address = <0x0 0x04001200>;
        max-size = <0x200>;
        id = <TOS_FW_CONFIG_ID>;
    };
    nt_fw-config {
        load-address = <0x0 0x80000000>;
        max-size = <0x200>;
        id = <NT_FW_CONFIG_ID>;
    };
};
```

2. Configs Hand-Off (3)



arm