Safe update of firmware blobs in bootloader stack

Marek Vasut

May 14th, 2025

Content

- Bootloader stack and blob problem
- U-Boot SPL porting
- OpTee-OS start from U-Boot
- Falcon boot mode

Bootloader stack

Bootloader stack – past, present, future

Bootloader stack

- Software that runs between POWER ON and OS kernel
- Evolved over time and grew in complexity
- Turned into multi-component software
- Expanded onto multiple cores

Bootloader stack – past

Past

- Bootloader was a single program, e.g. U-Boot
- Bootloader was stored in direct mapped memory, e.g. NOR
- On POWER ON, CPU starts executing NOR content
- Bootloader directly interacted with hardware
- Bootloader did hardware initialization
- Bootloader loaded and started kernel
- Kernel directly interacted with hardware
- Kernel did more hardware initialization

Bootloader stack – present

Present

- Bootloader stages are composed of multiple programs from multiple projects with disparate licensing
  - BootROM built into SoC – Closed source
  - TFA (BL2, BL31, ...) – BSD-3-Clause
  - TEE – OpTee-OS BSD-2-Clause
  - U-Boot (and SPL, TPL, ...) – GPL-2.0-or-later
  - ...
- Bootloader stages are stored in various memory devices
- On POWER ON, CPU starts executing BootROM
- Some bootloader stages may directly interact with hardware
- Some bootloader stages may do hardware initialization
- Some bootloader stages may load and start kernels
- Kernel may directly interact with hardware
- Kernel may do more hardware initialization

Bootloader stack – future

Future

- Bootloader stages are running on different CPU cores
  - Cortex-M or Cortex-R safety cores always on
  - Cortex-A application cores as needed
  - ...
- On POWER ON, safety core starts executing BootROM
- Safety cores provide e.g. SCMI service for HW interaction
- Secure hardware blocks are controlled by safety cores only (clock, pinmux ...)
- Non-Secure hardware may be controlled by application cores
- Bootloader interacts with safety cores via some ABI to access secure hardware
- Kernel interacts with safety cores via some ABI to access secure hardware

Exception levels

- Bootloader stages run in different Exception Levels, EL:
  - EL3 – highest level, unrestricted access to everything (FW)
  - EL2 – hypervisor (e.g. U-Boot and Linux)
  - EL1 – virtualized OS (e.g. virtualized Linux)
  - EL0 – applications
  - Switch from higher EL to lower EL – exception return
  - Switch from lower EL to higher EL – exception instruction
  - ARM DDI 0487 D1.1 (aarch64) and G1.2 (aarch32) [LINK]
  - ARMv7-A has PL (privilege levels) PL0, PL1, PL2
- SMC – Trigger synchronous exception in EL3 (Secure monitor)
- HVC – Trigger synchronous exception in EL2 (Hypervisor)
- Secure and Non-Secure/Normal worlds, spans into busses

Services

- Bootloader stages may provide services:
  - BootROM built into SoC – Fixed address function calls
  - TFA BL31 – PSCI/SCMI provider
  - TEE – PSCI/SCMI provider
  - U-Boot – PSCI provider
- Lower EL software can call those services
- Call is done either via direct function call, or SMC/HVC/...
- PSCI – Power State Coordination Interface
  (CPU on/off/virt, power control)
- SCMI – System Control and Management Interface
  (clock, pinmux, regulators ...)
- Interfaces to these services are an ABI
- Non-standard ABI extensions in vendor packages do exist
- Broken ABI changes across vendor package versions do happen

Firmware ABI

The firmware ABI must be stable, otherwise a system update can lead to either

- Kernel only update:
  - New kernel expects new ABI, old bootloader provides old ABI, system fails to boot
  - Recovery kernel can likely still use old ABI
  - System can be reinstalled with old kernel version
  - Recovery likely possible
- Bootloader and kernel update
  - New kernel expects new ABI, new bootloader provides new ABI, system may boot
  - But if new kernel fails to boot ...
  - Recovery kernel expects old ABI, new bootloader provides new ABI, recovery system fails to boot
  - Recovery NOT possible, brick

The plan

Reorder the bootloader stack

- FROM:
  - BootROM built into SoC – EL3
  - TFA – EL3 - EL2
  - TEE – EL2
  - U-Boot – EL2
  - Linux – EL2
- TO:
  - BootROM built into SoC – EL3
  - U-Boot SPL – EL3
  - U-Boot – EL3
  - TEE – EL2
  - Linux – EL2
- Falcon boot mode:
  - BootROM built into SoC – EL3
  - U-Boot SPL – EL3
  - TEE – EL2
  - Linux – EL2

The implementation

There are three platform specific issues to solve:

- Replace TFA BL2 with U-Boot SPL
  - Makes Falcon boot mode easily possible
- Start OpTee-OS from U-Boot as part of fitImage
  - Makes A/B update easily possible
- Enable Falcon boot mode in SPL
  - Significantly reduces attack surface

Replace TFA BL2 with U-Boot SPL

Replace TFA BL2 with U-Boot SPL

- TFA BL2 performs hardware initialization
- U-Boot SPL can perform the same initialization
- Implement clock/pinmux/regulator/. . . drivers in U-Boot DM (Driver Model)
- Implement DRAM driver in U-Boot DM
- Drivers are often already available for similar hardware

Replace TFA BL2 with U-Boot SPL – STM32MP13xx

STM32MP13xx – very similar to STM32MP15xx:

- Pinmux driver already present, enable in SPL
- Clock driver depends on SCMI, had to be extended
- DRAM controller driver extended to support 16bit bus
- BootROM image format changed, STM32Image v2 added to U-Boot `mkimage` tool
- Patches are posted , see next slide

Replace TFA BL2 with U-Boot SPL – patches

Fixes and preparatory work:

- image: Fix FIT image loadable section custom processing
- ARM: Align image end to 8 bytes to fit DT alignment
- ARM: stm32: Drop unnecessary space
- ARM: stm32: Fix SYSRAM size on STM32MP13xx
- ARM: stm32: Fix DBGMCU macro on STM32MP13xx
- ARM: stm32: Auto-detect ROM API table on STM32MP15xx
- clk: stm32mp13: Fix typo in STM32MP13 RCC driver

Actual implementation:

- clk: stm32mp13: Add SPL support and clock tree init to STM32MP13 RCC driver
- ram: stm32mp1: Add STM32MP13xx support
- tools: stm32image: Add support for STM32 Image V2.0
- ARM: stm32: Add STM32MP13xx SPL and OpTee-OS start support
- ARM: bootm: Add support for starting Linux through OpTee-OS on ARMv7a

Replace TFA BL2 with U-Boot SPL – Implementation

SPL is size constrained environment:

- Be mindful of SRAM size constraints
  - Avoid writes outside of SRAM
  - Avoid stack overflows (may overwrite SPL text too)
  - Consider heap size, do not malloc()ate too much
- Consider U-Boot `DEBUG_UART` – early UART output
  - `printf()` is still the king of debuggers
- Find a way to start SPL and perform fast test/debug cycles
  - STM32MP13xx – Either DFU or JTAG
    - DFU – [dfu-util link]
    - JTAG – [OpenOCD link]

Replace TFA BL2 with U-Boot SPL – Development

Start U-Boot SPL on STM32MP13xx using `dfu-util`:

```
make -j$(nproc) && \
  dfu-util -w -a 0 -D u-boot-spl.stm32 -R
```

Start U-Boot SPL on STM32MP13xx using `OpenOCD`, part 1:

```
./src/openocd -s tcl/ \
              -f tcl/interface/ftdi/flyswatter2.cfg \
              -f tcl/target/stm32mp13x.cfg \
              -f myscripts.cfg \
              -c "adapter speed 20000" \
              -c "reset_config trst_and_srst" \
              -c "jtag_ntrst_delay 20"
```

Replace TFA BL2 with U-Boot SPL – OpenOCD

OpenOCD is scriptable, use it to automate testing:

```
1  cat myscripts.cfg
2  # Start SPL (abbreviations are faster to type too)
3  proc sp {} {
4          targets 2
5          reset halt
6          load_image /u-boot/spl/u-boot-spl.bin 0x2ffe0000 bin
7          arm core_state arm
8          resume 0x2ffe0000
9  }
10
11 # Start U-Boot
12 proc su {} {
13         halt
14         load_image /u-boot/u-boot.bin 0xc0100000 bin
15         arm core_state arm
16         resume 0xc0100000
17 }
```

Replace TFA BL2 with U-Boot SPL – OpenOCD

```
$ telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
> sp
JTAG tap: stm32mp13x.tap tap/device found: 0x6ba00477 \
  (mfg: 0x23b (ARM Ltd), part: 0xba00, ver: 0x6)
JTAG tap: stm32mp13x.clc.tap tap/device found: 0x06501041 \
  (mfg: 0x020 (STMicroelectronics), part: 0x6501, ver: 0x0)
stm32mp13x.cpu: ran after reset and before halt ...
stm32mp13x.cpu: MPIDR level2 0, cluster 0, core 0, \
  multi core, no SMT
target halted in Thumb state due to debug-request, \
  current mode: Supervisor
cpsr: 0x800001f3 pc: 0x00002050
MMU: disabled, D-Cache: disabled, I-Cache: disabled
>
```

Remove PSCI and SCMI dependencies from U-Boot

- Majority of dependencies are in drivers
- Drivers in SPL cannot depend on PSCI/SCMI
- Reuse SPL driver modifications in U-Boot

Start OpTee-OS from U-Boot as part of fitImage

OpTee

- [OpTee about link]
- OP-TEE is a Trusted Execution Environment (TEE) designed as companion to a non-secure Linux kernel
- Multiple components:
  - OpTee-OS - library/runtime
  - Trusted applications (on top of TEE)
  - Untrusted Linux kernel (i.e. the usual)
  - Linux userspace (and optional TEE client)
- OpTee-OS is the interesting part

OpTee-OS

- OpTee-OS - library/runtime
- Loaded early on by early bootloader stages (TFA BL2, U-Boot SPL)
- On ARMv7-A, contains its own SM (Secure Monitor) which handles SMC
- On ARMv8-A and newer, depends on external SM to handle SMC and trigger OpTee-OS
- Calls into SM trigger OpTee-OS as if calling a library
- OpTee-OS can trigger external SM to access services

OpTee-OS start on ARMv7-A

- OpTee-OS generates multiple files during its build
- Pick the correct file, most likely `tee-raw.bin`
- Load the file at the correct destination address (link address)
- Set up parameters passed to OpTee-OS and then to next stage:
  - `r1` – Machine ID, set to 0xffffffff
  - `r2` – Usually DT pointer in DRAM
  - `lr` – Next stage address in DRAM
  - [Linus Walleij arm32 kernel boot article link]
- Jump to the OpTee-OS address:
  - OpTee-OS does its own initialization
  - OpTee-OS installs SMC handler and SM
  - OpTee-OS triggers SM using SMC call
  - SM returns to NS world using `rfefd` return from exception
  - The `rfefd` pops stack and returns to `lr` above
  - This can directly jump to Linux kernel

OpTee-OS start on ARMv7-A – Simple test

Start OpTee-OS from either OpenOCD or U-Boot:

```
cat myscripts.tcl
proc st {} {
        halt
        load_image /optee_os/out/arm-plat-stm32mp1/core/tee-raw.bin 0xde000000 bin
        arm core_state arm
        resume 0xde000000
        sleep 1000
        halt
        # Dump CPU registers and check the stack:
        reg
        mdw [dict values [get_reg -force {sp_mon}]] 2
}
```

```
# U-Boot
STM32MP> tftpb $loadaddr 192.168.1.1:tee-raw.bin
STM32MP> cp.b $loadaddr 0xde000000 $filesize
STM32MP> dcache off && icache off && go 0xde000000
STM32MP> I/TC: Early console on UART#4
I/TC:
I/TC: Embedded DTB found
I/TC: OP-TEE version: 4.4.0-3-g15ef367ad-dev \
  (gcc version 14.2.0 (Debian 14.2.0-19)) #1 Fri May  2 23:44:17 UTC 2025 arm
I/TC: WARNING: This OP-TEE configuration might be insecure!
I/TC: WARNING: Please check https://optee.readthedocs.io/en/latest/architecture/porting_guidelines.html
I/TC: Primary CPU initializing
I/TC: WARNING: All debug accesses are allowed
I/TC: Platform stm32mp1: flavor 135F_DHCOR_DHSBC - DT stm32mp135f-dhcor-dhsbc.dts
I/TC: DTB enables console (non-secure)
I/TC: Primary CPU switching to normal world boot
```

fitImage

fitImage

- Multi-component image type based on DT
- Capable of bundling multiple images into single container: kernel images, ramdisks, DTs, firmware blobs, ...
- U-Boot is capable of booting fitImage
- OpenEmbedded core is capable of generating fitImage
- Recommended for any contemporary system using U-Boot

fitImage – images

fitImage .its image tree source

```
1  /dts-v1/;
2
3  / {
4    description = "Linux kernel and FDT blob";
5
6    images {
7      kernel-1 {
8        description = "Linux kernel image for ARM64";
9        data = /incbin/("./arch/arm64/boot/Image");
10       type = "kernel";
11       arch = "arm64";
12       os = "linux";
13       compression = "none";
14       load = <0x50200000>;
15       entry = <0x50200000>;
16       hash-1 {
17         algo = "crc32";
18       };
19     };
20     fdt-1 {
21       description = "Flattened Device Tree blob";
22       data = /incbin/("./arch/arm64/boot/dts/renesas/r8a779g0-white-hawk.dtb");
23       type = "flat_dt";
24       arch = "arm64";
25       compression = "none";
26       hash-1 {
27         algo = "crc32";
28       };
29     };
30   };
31 ...
```

fitImage – configurations

fitImage .its image tree source

```
/dts-v1/;

/ {
  description = "Linux kernel and FDT blob";

  images {
    kernel-1 {
      ...
    };
    fdt-1 {
      ...
    };
    fdt-2 {
      ...
    };
    ...
  };

  configurations {
    default = "conf-1";
    conf-1 {
      description = "Boot Linux kernel with FDT blob 1";
      kernel = "kernel-1";
      fdt = "fdt-1";
    };
    conf-2 {
      description = "Boot Linux kernel with FDT blob 2";
      kernel = "kernel-1";
      fdt = "fdt-2";
    };
    ...
  };
};
```

OpTee-OS start on ARMv7-A – fitImage

The fitImage support for OpTee-OS is upstream, add node:

```
/dts-v1/;
/ {
...
  images {
...
    tee-1 {
      description = "OP-TEE";
      data = /incbin/("/optee_os/out/arm-plat-stm32mp1/core/tee-raw.bin");
      type = "tee";
      arch = "arm";
      compression = "none";
      os = "tee";
      load = <0xde000000>;
      entry = <0xde000000>;
      ...
  };};

  configurations {
    default = "conf-1";
    conf-1 {
...
      loadables = "tee-1";
...
    };
  };
};
```

OpTee-OS start on ARMv7-A – fitImage

Generate fitImage:

```
# Use external data with fitImage
mkimage -E -f fit-image.its /tftp/fitImage
```

Boot:

```
=> tftpboot $loadaddr 192.168.1.1:fitImage
=> bootm $loadaddr
# ... and hang
```

fitImage and OpTee-OS on ARMv7-A

- U-Boot currently loads OpTee-OS as a loadable to target address
- OpTee-OS is not triggered before booting Linux kernel
- Linux kernel boot code in U-Boot has to be extended:
  - Instead of direct jump to Linux kernel, jump through OpTee-OS
  - Use `U_BOOT_FIT_LOADABLE_HANDLER()` to record OpTee-OS load address
  - Introduce `boot_jump_linux_via_optee()` start OpTee-OS
- These modifications trigger OpTee-OS in case it was loaded, performs its setup and then returns to Linux kernel and starts Linux kernel
- Linux kernel can then access OpTee-OS via SMC and access its PSCI/SCMI services

fitImage and OpTee-OS on ARMv7-A – jump code

```
1  // arch/arm/lib/bootm-optee.S
2  ENTRY(boot_jump_linux_via_optee)
3      mov     r4, r3
4      mov     lr, r0
5      mov     r3, #0
6      mov     r0, #0
7      bx      r4
8  ENDPROC(boot_jump_linux_via_optee)
```

```
1  // arch/arm/lib/bootm.c
2  static bool boot_jump_via_optee;
3  static unsigned long boot_jump_via_optee_addr;
4
5  static void boot_jump_linux(struct bootm_headers *images, int flag) {
6  ...
7      if (boot_jump_via_optee)
8          boot_jump_linux_via_optee(kernel_entry, machid, r2, boot_jump_via_optee_addr);
9  ...
10 }
11
12 static void arch_tee_image_process(ulong image, size_t size) {
13     boot_jump_via_optee = true;
14     boot_jump_via_optee_addr = image;
15 }
16 U_BOOT_FIT_LOADABLE_HANDLER(IH_TYPE_TEE, arch_tee_image_process);
```

OpTee-OS start on ARMv7-A – Boot test I

Start OpTee-OS from fitImage:

```
STM32MP> tftpb $loadaddr 192.168.1.1:fitImage && bootm $loadaddr
...
## Loading kernel (any) from FIT Image at c2000000 ...
   Using 'conf-1' configuration
   Trying 'kernel-1' kernel subimage
     Description:  Linux kernel
     Created:      2025-05-11   0:30:29 UTC
     Type:         Kernel Image
     Compression:  uncompressed
     Data Start:   0xc20004c8
     Data Size:    9232920 Bytes = 8.8 MiB
     Architecture: ARM
     OS:           Linux
     Load Address: 0xc0008000
     Entry Point:  0xc0008000
     Hash algo:    crc32
     Hash value:   33fea6d6
   Verifying Hash Integrity ... crc32+ OK
## Loading fdt (any) from FIT Image at c2000000 ...
   Using 'conf-1' configuration
   Trying 'fdt-1' fdt subimage
     Description:  Flattened Device Tree blob
     Created:      2025-05-11   0:30:29 UTC
     Type:         Flat Device Tree
     Compression:  uncompressed
     Data Start:   0xc28ce6e0
     Data Size:    65990 Bytes = 64.4 KiB
     Architecture: ARM
     Hash algo:    crc32
     Hash value:   0be45dea
   Verifying Hash Integrity ... crc32+ OK
   Booting using the fdt blob at 0xc28ce6e0
Working FDT set to c28ce6e0
```

OpTee-OS start on ARMv7-A – Boot test II

Start OpTee-OS from fitImage:

```
...
## Loading loadables (any) from FIT Image at c2000000 ...
   Trying 'tee-1' loadables subimage
     Description:  OP-TEE
     Created:      2025-05-11   0:30:29 UTC
     Type:         Trusted Execution Environment Image
     Compression:  uncompressed
     Data Start:   0xc28de8a8
     Data Size:    456544 Bytes = 445.8 KiB
     Hash algo:    crc32
     Hash value:   bcaa3d55
   Verifying Hash Integrity ... crc32+ OK
   Loading loadables from 0xc28de8a8 to 0xde000000
   Loading Kernel Image to c0008000
   Loading Device Tree to cffec000, end cffff1c5 ... OK
Working FDT set to cffec000
No RNG device

Starting kernel ...

I/TC: Early console on UART#4
...
```

OpTee-OS start on ARMv7-A – Boot test III

Start OpTee-OS from fitImage:

```
Starting kernel ...

I/TC: Early console on UART#4
I/TC:
I/TC: Embedded DTB found
I/TC: OP-TEE version: 4.4.0-3-g15ef367ad-dev (gcc version 14.2.0 (Debian 14.2.0-19)) #1 Sat May 10 ...
I/TC: WARNING: This OP-TEE configuration might be insecure!
I/TC: WARNING: Please check https://optee.readthedocs.io/en/latest/architecture/porting_guidelines.html
I/TC: Primary CPU initializing
I/TC: WARNING: All debug accesses are allowed
I/TC: Platform stm32mp1: flavor 135F_DHCOR_DHSBC - DT stm32mp135f-dhcor-dhsbc.dts
I/TC: DTB enables console (non-secure)
I/TC: Primary CPU switching to normal world boot
I/TC: Reserved shared memory is disabled
I/TC: Dynamic shared memory is enabled
I/TC: Normal World virtualization support is disabled
I/TC: Asynchronous notifications are enabled
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 6.12.26-00018-gdeda01d71ca8 (marex@pc) (arm-linux-gnueabi-gcc ...
...
[    0.907311] optee: probing for conduit method.
[    0.907328] optee: revision 4.4 (15ef367a)
[    0.914062] optee: Asynchronous notifications enabled
[    0.914078] optee: dynamic shared memory is enabled
...
```

fitImage and OpTee-OS and TZC and ETZPC

- OpTee-OS is security sensitive code
- It is mandatory to configure TrustZone correctly
- TrustZone is configured at multiple levels
- TZ filters used to control bus-level access (ETZPC on STM32MP13xx)
- TZ filters used to control DRAM access (TZC on STM32MP13xx)
- Filtering often based on S/NS and AXI or similar bus IDs

A/B update

A/B update

- A/B update of OpTee-OS bundled into fitImage is trivial
- fitImage is a single file stored in filesystem,
  or a run of blocks on MTD device
- There can be multiple copies of different fitImages
- U-Boot can pick and boot one of the fitImages
- fitImage selection can be based e.g. on boot counter
  see `bootcmd` and `altbootcmd`
- Even if the OpTee-OS implements some sort of broken PSCI/SCMI extension, this is now also fine – the kernel which likely depends on that extension is bundled together with the broken PSCI/SCMI extension provider in the same fitImage and is updated in lockstep
- This avoids ABI mismatch between the kernel and firmware (PSCI/SCMI provider)

Enable Falcon boot mode in SPL

Too much code in security sensitive context

- U-Boot SPL and U-Boot together contain a lot of code
- Code running in security sensitive context has to be tiny
- Such code has to be easy to inspect and understand
- Reduce the amount of code running in security sensitive context:
  - Do not run U-Boot at all
  - Boot fitImage from SPL directly

Falcon boot mode

U-Boot SPL is capable of booting fitImage directly

- SPL is capable of fitImage authentication/decryption
- SPL can boot various images embedded in fitImages, Linux, TEE, TFA BL31, ...
- SPL can select and boot Linux or U-Boot fallback fitImage
- SPL is also capable of A/B switching, bootargs passing, ...

Falcon boot mode – enablement

Enable `CONFIG_SPL_OS_BOOT` and related options in U-Boot config:

```
cat .config
...
CONFIG_SPL_FIT=y
...
CONFIG_SPL_OS_BOOT=y
...
CONFIG_SPL_FALCON_BOOT_MMCSD=y
CONFIG_SPL_SYS_MMCSD_RAW_MODE=y
CONFIG_SYS_MMCSD_RAW_MODE_U_BOOT_USE_SECTOR=y
CONFIG_SYS_MMCSD_RAW_MODE_U_BOOT_SECTOR=0x22
CONFIG_SYS_MMCSD_RAW_MODE_KERNEL_SECTOR=0x2000
...
```

```
// Implement fallback selector in board/ board file.
int spl_start_uboot(void)
{
        /* U-Boot / Linux selector , 0 is Falcon , 1 is U-Boot */
       return 0;
}
```

Falcon boot mode – Jump via OpTee-OS

- Linux has to be started through OpTee-OS
- Modify `arch/arm/lib/spl.c jump_to_image_linux()`
- Modification similar to U-Boot is necessary in SPL
- The `bootm-optee.S` jump code is reused
- TZC and ETZPC must be configured in SPL already
- Patches posted:
  - spl: fit: Add ability to jump to Linux via OpTee-OS on ARMv7a

Falcon A/B switching

- Override `board_spl_mmc_get_uboot_raw_sector()`
- Select A or B copy based on boot counter
  (e.g. non-volatile CPU register)

Conclusion

- TFA BL2 can be replaced by U-Boot SPL
- OpTee-OS can be started from U-Boot as part of fitImage
  - Makes development easier
  - Avoids update dangers in production
- OpTee-OS can be started from SPL as part of fitImage
  - Reduces amount of code in privileged mode
  - Improves platform security

Thank you for your attention

Marek Vasut <marek.vasut+er25@mailbox.org>