Booting blobs between U-Boot and Linux

Marek Vasut

February 1st, 2025

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]
- SMC – Trigger synchronous exception in EL3 (Secure monitor)
- HVC – Trigger synchronous exception in EL2 (Hypervisor)

Services

- Bootloader stages may provide services:
  - BootROM built into SoC – Fixed address function calls
  - TFA – BL31 PSCI/SCMI provider
  - TEE – 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

Secure and non-secure resources

- Bootloader stages may protect resources:
  - Configure TrustZone to restrict access to peripherals
  - Configure higher EL MMU tables to restrict access to memory ranges
- U-Boot was always meant as a boot monitor and debug tool
- If access to resources is blocked by higher EL blobs, this does not work
- EL and resource protection can be a good thing:
  - U-Boot could run in EL2 and use EL3 as access fault handler (suggested by Caleb)

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 – EL3
  - TFA – EL3 - EL2
  - TEE – EL2
  - Linux – EL2

The implementation

There are two platform specific issues to solve:

- U-Boot may already depend on PSCI/SCMI calls
- TFA BL31/TEE may be started before U-Boot

U-Boot PSCI/SCMI dependency removal

- Very doable unless PSCI/SCMI provider closed source
- Many PSCI/SCMI providers do not guarantee source access
- Functionality accessed via PSCI/SCMI from software in EL2 is implemented in the PSCI/SCMI provider running in EL3 via direct register accesses
- Direct register access available in U-Boot running in EL3 too
- Implement clock/pinmux/regulator/. . . drivers in U-Boot DM (driver model)
- Run U-Boot in EL3 without calling PSCI/SCMI services

U-Boot PSCI/SCMI provider

- On ARM64, PSCI is mandatory, SCMI optional
- PSCI and possibly SCMI provider must start before Linux
- U-Boot running in EL3 can act as PSCI provider
- U-Boot can act as fallback PSCI provider until user one starts
- Full PSCI implementation is SoC specific
- Basic PSCI implementation can be completely generic
- `arch/arm/mach-renesas/psci-rcar64.c`

Starting TFA BL31 from U-Boot

- Two ways of starting TFA BL31 from U-Boot in EL3:
  - Direct via U-Boot shell (for development)
    - The same U-Boot executable can run in EL3 and EL2
    - Build TFA BL31 with destination address matching U-Boot proper load address (`TEXT_BASE`)
    - Load TFA BL31 to memory in U-Boot shell
    - Disable caches – `=> dcache off ; icache off`
    - Jump to TFA BL31 – `=> go 0xaddress`
    - TFA BL31 switches from EL3 to EL2
    - U-Boot is started again, this time in EL2
  - Direct jump to Linux through TFA BL31 using fitImage
    - The fitImage container contains TFA BL31, Linux, DTs, ...
    - Support for starting TFA BL31 PSCI provider as part of fitImage posted [LINK to 3 patch series]
    - fitImage hooks set up TFA BL31 jump address to U-Boot kernel jump code

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 load and start

- U-Boot copies images referenced in selected fitImage configuration to their destination addresses in memory
- U-Boot runs loadable handler function for each image
- TFA BL31 firmware image type has been posted [LINK]
- Vendor specific TFA BL31 forks may require extra setup before being started
- TFA BL31 loadable handler and jump prep handler has been added for Renesas R-Car V4H

fitImage TFA BL31 image type

```
diff --git a/boot/image.c b/boot/image.c
@@ -183,6 +183,7 @@ static const table_entry_t uimage_type[] = {
        {       IH_TYPE_FDT_LEGACY, "fdt_legacy", "legacy Image with Flat Device Tree ", },
        {       IH_TYPE_RENESAS_SPKG, "spkgimage", "Renesas SPKG Image" },
        {       IH_TYPE_STARFIVE_SPL, "sfspl", "StarFive SPL Image" },
+       {       IH_TYPE_TFA_BL31, "tfa-bl31",  "TFA BL31 Image", },
        {       -1,                     "",             "",                     },
 };

diff --git a/include/image.h b/include/image.h
@@ -232,6 +232,7 @@ enum image_type_t {
        IH_TYPE_FDT_LEGACY,             /* Binary Flat Device Tree Blob in a Legacy Image */
        IH_TYPE_RENESAS_SPKG,           /* Renesas SPKG image */
        IH_TYPE_STARFIVE_SPL,           /* StarFive SPL image */
+       IH_TYPE_TFA_BL31,               /* TFA BL31 image */

        IH_TYPE_COUNT,                  /* Number of image types */

diff --git a/boot/image-fit.c b/boot/image-fit.c
@@ -2167,6 +2167,7 @@ int fit_image_load(struct bootm_headers *images, ulong addr,
        type_ok = fit_image_check_type(fit, noffset, image_type) ||
                  fit_image_check_type(fit, noffset, IH_TYPE_FIRMWARE) ||
                  fit_image_check_type(fit, noffset, IH_TYPE_TEE) ||
+                 fit_image_check_type(fit, noffset, IH_TYPE_TFA_BL31) ||
                  (image_type == IH_TYPE_KERNEL &&
                   fit_image_check_type(fit, noffset, IH_TYPE_KERNEL_NOLOAD));
```

fitImage TFA BL31 loadable handler

Loadable handler

- Called after the TFA BL31 firmware blob got loaded from fitImage to destination address in memory
- May set up hand-off tables or parameters for the firmware
- Can be defined for any image type using macro `U_BOOT_FIT_LOADABLE_HANDLER()`

```
1  static void tfa_bl31_image_process(ulong image, size_t size)
2  {
3      /* Set up parameters passed to TFA BL31 by current stage */
4      struct bl2_to_bl31_params_mem *blinfo = ...;
5
6      blinfo->... = ...;
7      blinfo->... = ...;
8      /* Cache TFA BL31 load address for armv8_switch_to_el2_prep() */
9      tfa_bl31_image_addr = image;
10 }
11
12 /* Define a handler for IH_TYPE_TFA_BL31 */
13 U_BOOT_FIT_LOADABLE_HANDLER(IH_TYPE_TFA_BL31, tfa_bl31_image_process);
```

fitImage kernel jump prep handler

- Jump handler prep is called shortly before Linux kernel starts
- The handler runs in EL3
- The handler does final setup and jumps to an entry point
- The entry point can be TFA BL31 entry point
- TFA BL31 is expected to return into U-Boot, but in EL2
- U-Boot is re-entered right past `armv8_switch_to_el2`
- U-Boot is re-entered without valid stack pointer anymore
- U-Boot `armv8_switch_to_el2` runs and jumps to Linux

```
1  void armv8_switch_to_el2_prep(u64 args, u64 mach_nr, u64 fdt_addr, u64 arg4, u64 entry_pt, u64 es_flag)
2  {
3      typedef void __noreturn (*image_entry_noargs_t)(void);
4      image_entry_noargs_t image_entry = (image_entry_noargs_t)(void *)tfa_bl31_image_addr;
5      struct bl2_to_bl31_params_mem *blinfo = ...;
6
7      /* Destination address in arch/arm/cpu/armv8/transition.S right past the first bl
8       * in armv8_switch_to_el2() to let the rest of U-Boot pre-Linux code run. The code
9       * does run without stack pointer! */
10     const u64 ep = ((u64)(uintptr_t)&armv8_switch_to_el2) + 4;
11
12     /* Set up kernel entry point and parameters: x0 is FDT address, x1..x3 must be 0 */
13     blinfo->bl33_ep_info.pc = ep;
14     ...
15     /* Jump to TFA BL31 */
16     image_entry();
17 }
```

fitImage TFA BL31 usage

- Integration into the fitImage ITS source is a firmware node
- fitImage is built using `$ mkimage -f fit.its fit.itb`

```
/dts-v1/;

/ {
    description = "Linux kernel with FDT blob and TFA BL31";

    images {
        kernel-1 { ... };
        fdt-1 { ... };
        atf-1 {                    /* This is the TFA BL31 image */
            description = "TFA BL31";
            data = /incbin/("../build/plat/release/bl31.bin");
            type = "tfa-bl31"; /* <---------- This is the TFA BL31 image type */
            arch = "arm64";
            os = "arm-trusted-firmware";
            compression = "none";
            load = <0x46400000>;
            entry = <0x46400000>;
        };
    };

    configurations {
        default = "conf-1";
        conf-1 {
            description = "Boot Linux";
            kernel = "kernel-1";
            fdt = "fdt-1";
            loadables = "atf-1"; /* <--------- This is the TFA BL31 loadable */
        };
    };
};
```

fitImage TFA BL31 boot example – U-Boot

```
u-boot=> tftpboot 0x48000000 v4h/fitImage && bootm 0x48000000
...
## Loading kernel from FIT Image at 48000000 ...
...
## Loading fdt from FIT Image at 48000000 ...
...
Working FDT set to 493ff200
## Loading loadables from FIT Image at 48000000 ...
   Trying 'atf-1' loadables subimage
     Description:  ARM Trusted Firmware
     Type:         TFA BL31 Image
     Compression:  uncompressed
     Data Start:   0x4940f190
     Data Size:    131128 Bytes = 128.1 KiB
     Hash algo:    crc32
     Hash value:   58f07ac9
   Verifying Hash Integrity ... crc32+ OK
   Loading loadables from 0x4940f190 to 0x46400000
   Loading Kernel Image to 50200000
   Loading Device Tree to 0000000057fec000, end 0000000057ffeede ... OK
Working FDT set to 57fec000

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x414fd0b1]
...
```

fitImage TFA BL31 boot example – Linux

```
[    0.000000] psci: probing for conduit method from DT.
[    0.000000] psci: PSCIv1.1 detected in firmware.
[    0.000000] psci: Using standard PSCI v0.2 function IDs
[    0.000000] psci: Trusted OS resident on physical CPU 0x0
[    0.000000] psci: SMC Calling Convention v1.2
...
[    0.012046] smp: Bringing up secondary CPUs ...
[    0.020272] Detected PIPT I-cache on CPU1
[    0.020337] GICv3: CPU1: found redistributor 100 region 0:0x00000000f1080000
[    0.020369] CPU1: Booted secondary processor 0x0000000100 [0x414fd0b1]
[    0.024282] Detected PIPT I-cache on CPU2
[    0.024332] GICv3: CPU2: found redistributor 10000 region 0:0x00000000f10a0000
[    0.024352] CPU2: Booted secondary processor 0x0000010000 [0x414fd0b1]
[    0.028264] Detected PIPT I-cache on CPU3
[    0.028291] GICv3: CPU3: found redistributor 10100 region 0:0x00000000f10c0000
[    0.028304] CPU3: Booted secondary processor 0x0000010100 [0x414fd0b1]
[    0.028355] smp: Brought up 1 node, 4 CPUs
[    0.028390] SMP: Total of 4 processors activated.
[    0.028393] CPU: All CPU(s) started at EL2
[    0.028397] CPU features: detected: 32-bit EL0 Support
[    0.028401] CPU features: detected: Data cache clean to the PoU not required for I/D coherence
[    0.028407] CPU features: detected: CRC32 instructions
[    0.028411] CPU features: detected: RCpc load-acquire (LDAPR)
[    0.028415] CPU features: detected: LSE atomic instructions
[    0.028419] CPU features: detected: Privileged Access Never
[    0.028424] CPU features: detected: Speculative Store Bypassing Safe (SSBS)
[    0.028463] alternatives: applying system-wide alternatives
[    0.029869] CPU features: detected: Hardware dirty bit management on CPU0-3
...
root@linux-v4h:/# nproc
4
```

A/B update

- A/B update of TFA BL31 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
- Even if the BL31 implements some sort of broken PSCI extension, this is now also fine – the kernel which likely depends on that extension is bundled together with the broken PSCI extension provider in the same fitImage and is updated in lockstep
- This avoids ABI mismatch between the kernel and firmware (PSCI provider)

End

Thank you for your attention

Marek Vasut <marek.vasut+fosdem25@mailbox.org>