From Reset Vector to Kernel

Navigating the ARM Matryoshka

Ahmad Fatoum – a.fatoum@pengutronix.de

https://www.pengutronix.de

The SoC: NXP i.MX8MM

- ARM SoCs are very diverse, we will take the i.MX8MM as an example
- Multi-core: ARM Cortex-A53, Cortex-M4
- ARMv8.0-A
- Comparatively open documentation

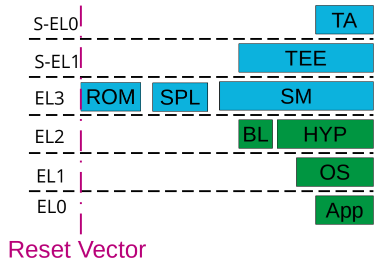
  → Good open source software support

[MNT Reform]

[Purism Librem 5]

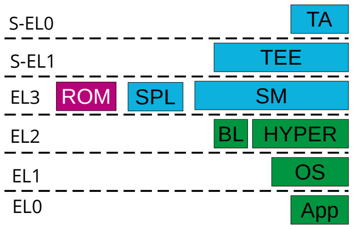
In the beginning was the Reset Vector

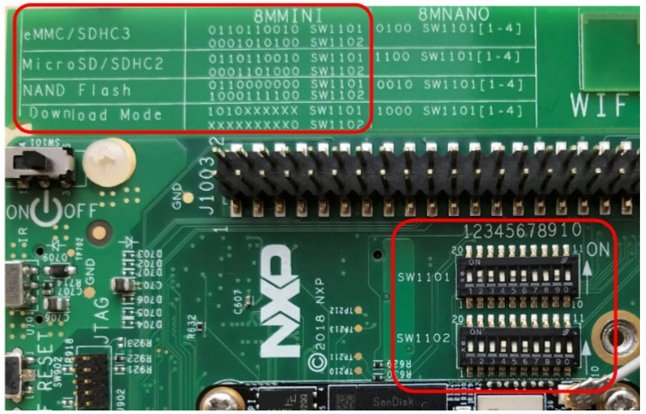
- An Implementation-defined address execution starts from
- Needs to point to something directly executable
  - On-Chip SRAM → Needs to be pre-loaded (JTAG, co-processor)
  - Memory mapped flash:
    - Effort: Needs to be pre-programmed
    - Cost: Often dedicated storage chip needed
    - Security: If the user reprograms it, they own the system
  - Mask boot ROM:
    - Effort: Can be used as In-System Programmer
    - Cost: Software on it can boot from many different sources
    - Security: If you trust it, you can verify the bootloader



Boot ROM

- Configure PLLs, Clocks, Stack, Interrupts
- Basic system initialization, e.g. latencies for on-chip SRAM
- Sample eFuses and strapping pins to determine boot mode
- Try configured boot media in order



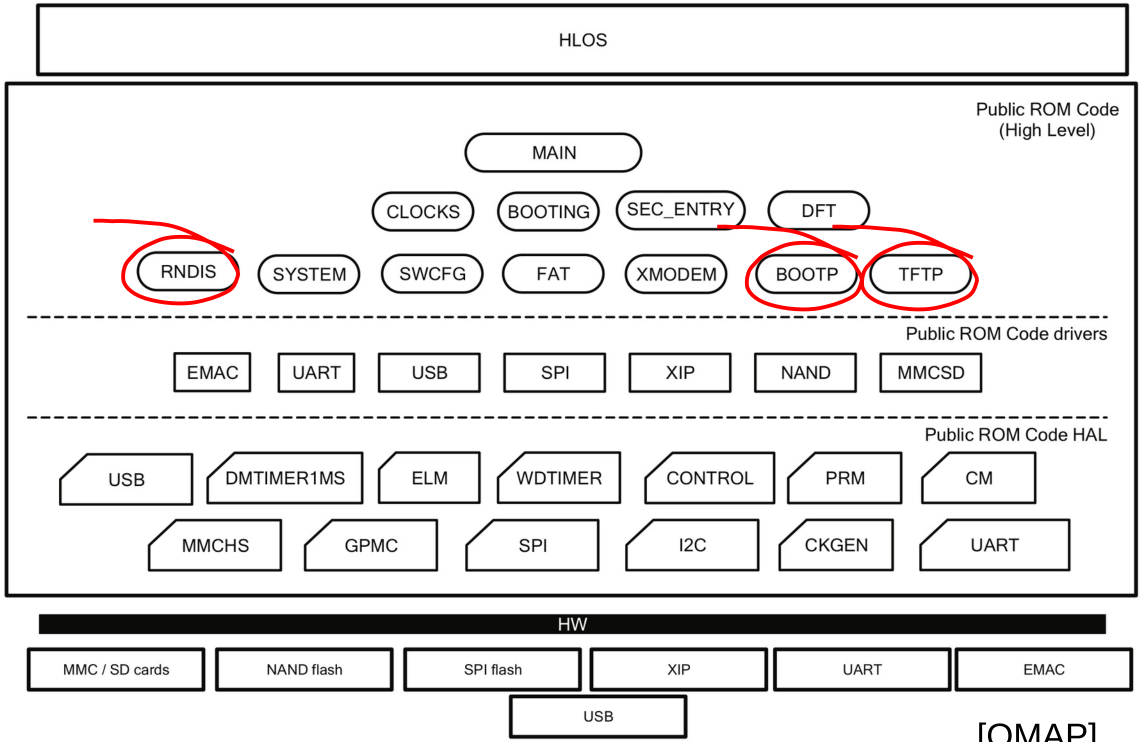


[EVK]

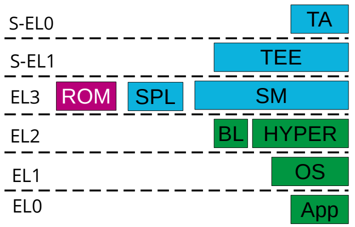
Boot ROMs are complex

- My favourite so far: OMAP tftp-over-usb-ethernet-gadget

Figure 26-1. Public ROM Code Architecture



[OMAP]



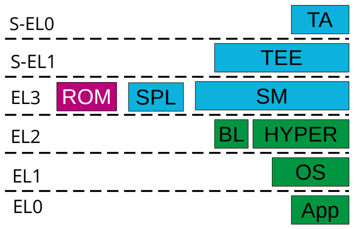
Boot ROMs are fallible

- Everything has bugs
- Sometimes workarounds are possible

```
ARM: imx8mq: reclock ARM PLL to 800MHz

The BootROM sets up the ARM PLL to run at 1.6GHz and then uses the
divider after the PLL the achieve a CPU clock rate of 800MHz. New Linux
kernels (>= 5.8) switch to a clock path that bypasses the divider, as
the divider should not be used for CPU clock frequencies >1GHz. If the
BootROM setup is left unchanged this causes the CPU clock to jump to
the full 1.6GHz until CPUfreq takes over and reprograms the PLL. This
rate is outside of the chip specification and leads to crashes.

Fix this by reclocking the ARM PLL to 800MHz.
```

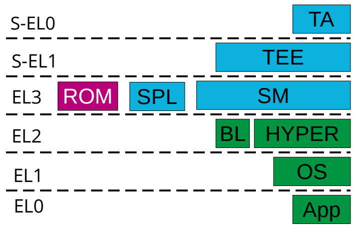
- Sometimes not:

```
CVE-2017-7932

An improper certificate validation issue was discovered in NXP i.MX [...]. When the
device is configured in security enabled configuration, under certain conditions it
is possible to bypass the signature verification by using a specially crafted
certificate leading to the execution of an unsigned image.
```



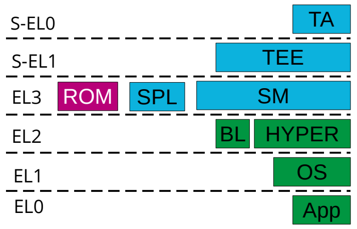
i.MX: Image Vector Table (IVT)

- Located within boot device at (type-specific) fixed offset
- Contains pointers to
  - Device Configuration Data (DCD)
  - Next-stage Bootloader Binary
  - Load Address
  - Command Sequence File (for Secure Boot)
  - NXP-Signed HDMI PHY firmware
- Specifies whether Boot image is a plugin

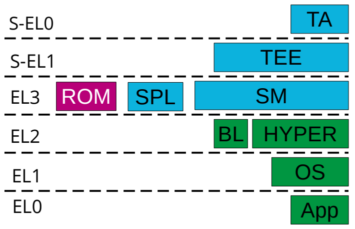


i.MX: Device Configuration Data (DCD)

- Bytecode interpreted by i.MX BootROMs
- Executed before copying boot program into memory
- Can do memory access to whitelisted regions
  - write 8/16/32 bit literal
  - poll until `(*addr & mask) == mask`
  - set/clear bits
- Can be sufficient to set up a SDRAM controller and load code to DDR3 directly
  - Not enough to configure voltages via I²C PMIC however
  - Not flexible enough to properly setup DDR4



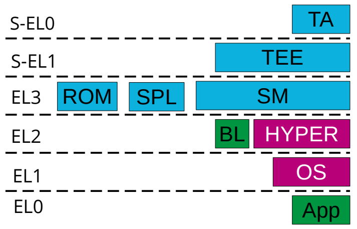
i.MX: Plugin Image



- Run ARM code, for when DCD can't set up SDRAM
- returns control to BootROM
- Avoids having to do own authentication in early boot
- Still need to reimplement access to boot medium
  - Newer i.MX8M SoCs support ROM API to reuse BootROM driver
- Not used in barebox

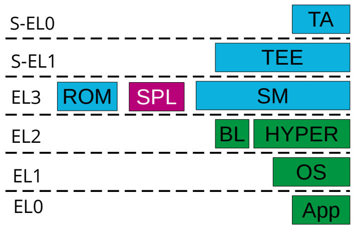
What now?

- We now know all we need to run our own code
- What's missing for Linux?
  - Setup and initialise the RAM
  - Install secure monitor for CPU power management
  - Setup the device tree
  - Jump to the kernel image



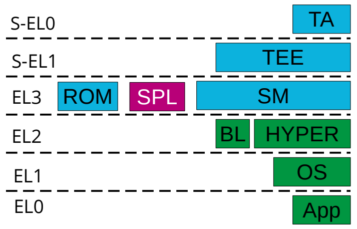
Setup and initialise RAM

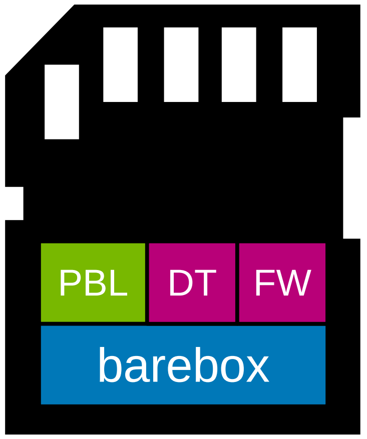
- Configure voltage regulators
  - Depending on model, initial values can be fused
  - Runtime configuration usually via I²C
- Configure DDR DRAM controller
  - Vendor provides tool and spreadsheet to generate register configuration
  - DDR PHY microcontroller firmware blob used for training
- We will need some code running outside DRAM to initialise it:
  - DCD commands: Used with i.MX6
  - ARM code: Needed for proper communication with the DDR4 PHY µC



Secondary Program Loader (SPL)

- Also called first-stage bootloader (BootROM is 0th Stage)
- Runs from SRAM or eXecuted-In-Place (XIP) from supported flash and sets up DRAM
- For barebox, the PreBootloader (PBL) is used for this:
  - normally extracts barebox to end of SDRAM
    - Allows for compressed multiplatform bootloader images
  - Depending on platform, is extended to:
    - Pass hardware device description to chainloaded barebox proper
    - Do low-level and SDRAM init
    - Load early boot firmware

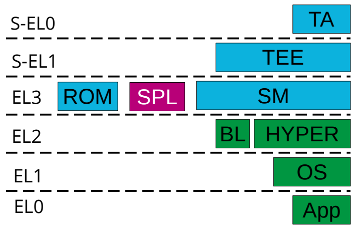


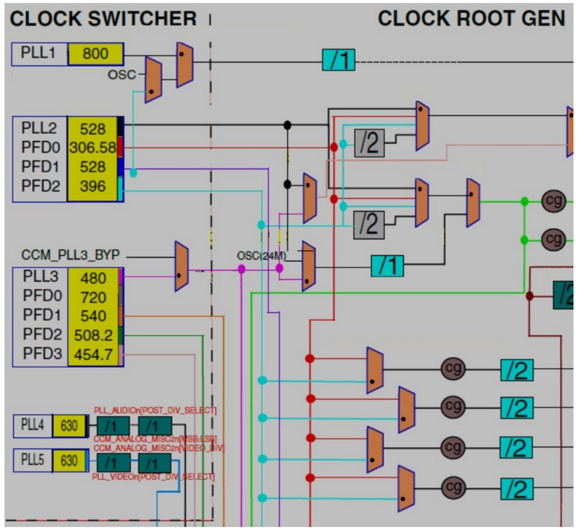


PBL: SoC init

```
    imx8mm_cpu_lowlevel_init();
    imx8mm_early_clock_init();
```

- Configure EL0/1 for non-secure operation
- Configure next lower level as AArch64
- Disable traps, MMU, caches ... etc.
  - Registers may be left in bad state by BootROM
- Initialize clock tree for PBL usage

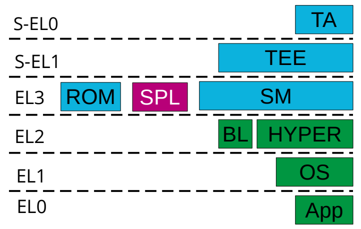




[MX6-CLK]

PBL: C and first peripheral

```
relocate_to_current_addr();
setup_c();
setup_uart();
```

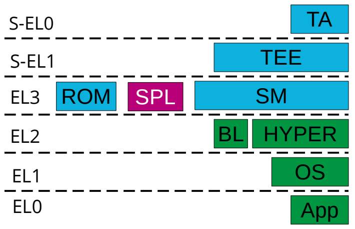


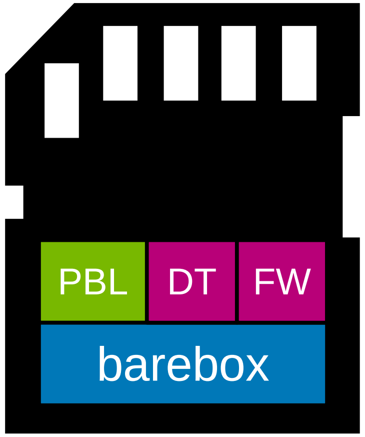
- Set up C environment
- Enable early serial output
  - imx8m_early_setup_uart_clock
  - imx8mm_setup_pad: Pin muxing
  - imx8m_uart_setup: e.g. default Baudrate
  - pbl_set_putc: Store global function pointer for output

PBL: chainloading

- Configure PMIC
- Configure SDRAM controller
- BootROM already configured some boot medium

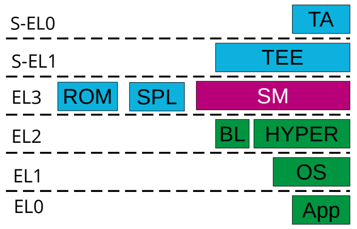
  → reuse, don't reconfigure from scratch
- BootROM-loaded PBL chainloads full barebox from same boot source
  - Serial → Provide USB gadget to continue boot
  - eMMC → chainload full barebox image
- What about secondary CPU cores though?





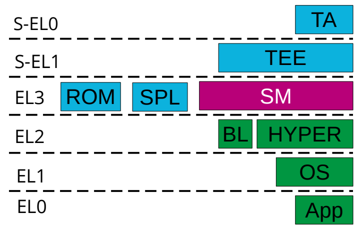
ARM PSCI

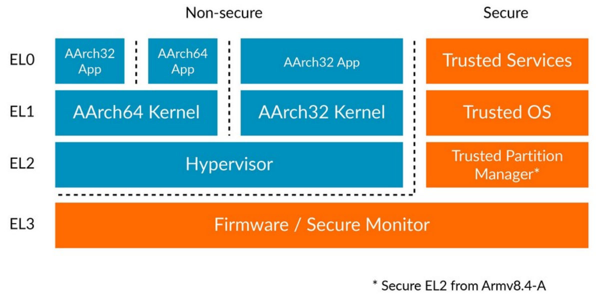
- Control of secondary cores highly SoC-specific
- Native and Virtualized Kernel need different power management drivers
- Solution: Generic interface
  - If called from Hypervisor (EL2), trap into secure monitor (EL3)
  - If called from Kernel (EL1), trap into Hypervisor (EL2)
- Secure Monitor runs in isolated secure world



TrustZone

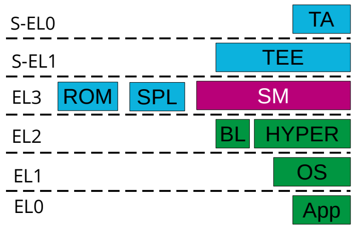
- Partitions system into secure and non-secure worlds
  - Linux and your normal applications run in non-secure world
  - Non-secure world can trap into secure monitor to access secure resources
    - Power Management
    - Clock and Reset Handling
    - Hacks the vendor couldn't get into mainline Linux





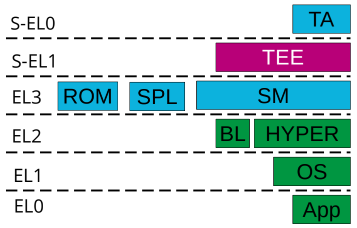
Trusted Firmware - A

- Reference Implementation for secure world firmware
- Its actual function is SoC-dependent. Common part: execution in secure (trusted) world. On i.MX8M:
  - provides secure monitor
  - handles CPU power management via PSCI
  - configures interrupt controller
  - loads, and forwards secure monitor calls to, Trusted Execution Environment (OS for secure world)



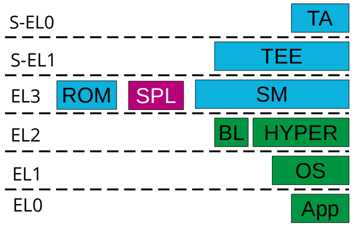
OP-TEE

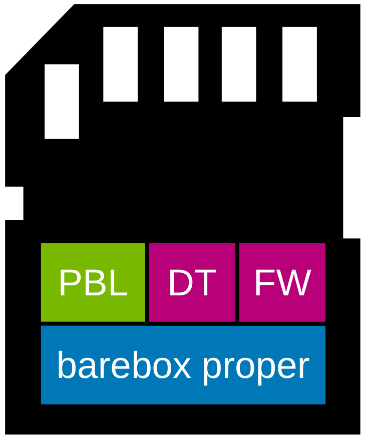
- Increasing need for tamper-resistant trusted applications (TAs)
  - Handling sensitive data, e.g. TLS keys, disk encryption
  - Streaming DRM-protected Media
  - Trusted Platform Module
  - Verified boot support
- Solution
  - Run „trustlets“ in secure EL0 which use standardized API to communicate with trusted OS
  - Non-secure userspace can access services via secure monitor calls
- OP-TEE is an open-source implementation of the GP TEE specification



PBL: TF-A Handling

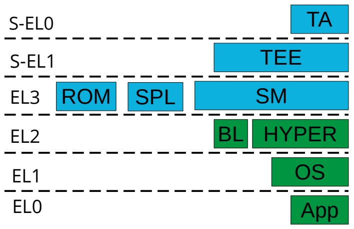
- TF-A binary is linked into barebox PBL
- PBL copies full barebox to TF-A return address in SDRAM
- TF-A sets up secure monitor in EL3 and returns control to barebox in non-secure EL2
- Now full barebox runs and because it's EL2, not EL3:
  - Enables MMU and caches
  - Verifies correctness of barebox proper hash on secure boot
  - Extracts barebox proper to end of SDRAM
  - Jumps to barebox proper and pass it device tree





Device Tree

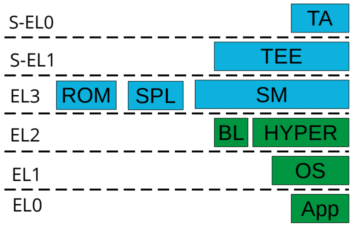
- Not all hardware is discoverable
- Hardware description in C code doesn't scale
- Original Open Firmware spec describes how hardware can bundle its own device description and drivers
- Linux extended the hardware description part to replace static board description in C code on ARM
  - Has its own specification now
  - Bindings and device trees still mainly maintained in the Linux source tree
    - also used by FreeBSD, barebox, U-Boot, TF-A, OP-TEE, Zephyr, …
    - not specific to ARM: Used on RISC-V, PowerPC, MIPS, even few x86, …
- Mandatory for all new ARM Linux platforms



Device Tree: Example

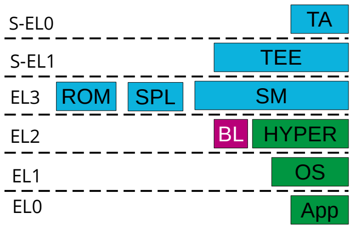
```
/* SoC device tree */
flexspi: spi@30bb0000 {
        compatible = "nxp,imx8mm-fspi";
        reg = <0x30bb0000 0x10000>, <0x8000000 0x10000000>;
        reg-names = "fspi_base", "fspi_mmap";
        interrupts = <GIC_SPI 107 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&clk IMX8MM_CLK_QSPI_ROOT>,
                 <&clk IMX8MM_CLK_QSPI_ROOT>;
        clock-names = "fspi", "fspi_en";
        #address-cells = <1>;
        #size-cells = <1>;
};
```

```
/* Board device tree */
&flexspi {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_flexspi>;

        flash@0 {
                reg = <0>;

                compatible = "jedec,spi-nor";
                spi-max-frequency = <80000000>;
                spi-tx-bus-width = <4>;
                spi-rx-bus-width = <4>;
        };
};
```



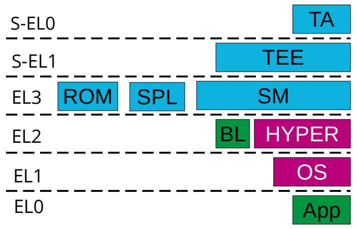
barebox proper

- Basically a stripped down simplified kernel
  - Most drivers ported from Linux, can probe from DT
  - No interrupts, only cooperative multitasking
  - Unix-like abstractions: virtual file system, block layer, character devices, file descriptors and shell
- But with goodies for facilitating boot and development
  - Redundant watchdog-supervised boot sequences
  - Shared state with OS for variable exchange
  - Hardware manipulation primitives
  - Device Tree fixups
  - Verified Boot
  - All linked into single binary



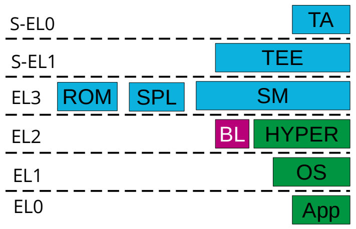
Linux: call the kernel image

- Bootloader needs to:
  - Mask interrupts
  - Initialize standard ARM timer
  - Load kernel at offset specified in header
  - Disable MMU, Data Caches
  - Initialize CPU registers for either EL2 or EL1
    - e.g. x0 for device tree blob
- Jump to it

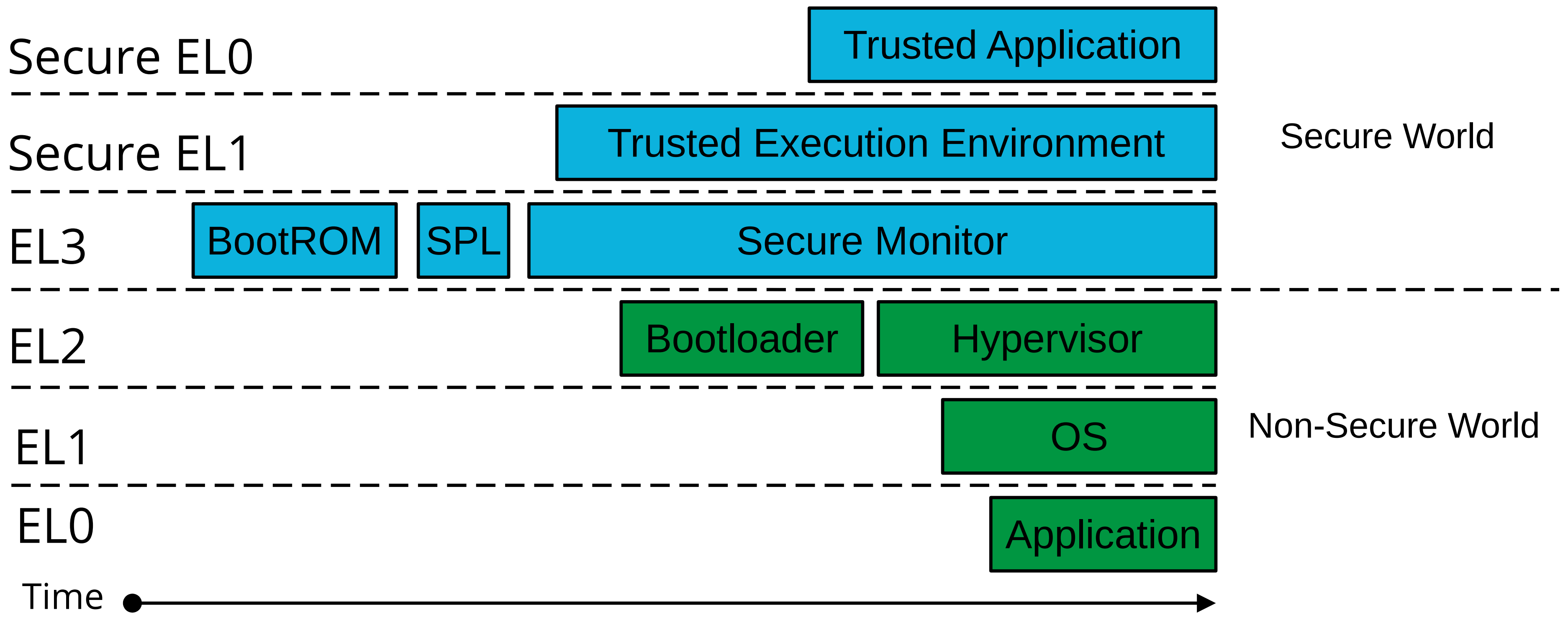


What else? UEFI on ARM

- Distros like their GRUB
  - → Have bootloader (U-Boot, Tianocore) provide minimal UEFI environment
  - → Compile your usual UEFI payload for ARM
- UEFI is almost its own operating system
  - Embedded/Server Base Boot Requirements (EBBR/SBBR) define subsets
- For embedded vendors:
  - Solves some already solved problems differently
  - You are already responsible for the whole stack → Each extra component is extra maintenance burden
- For commercial SBCs:
  - Useful for running general purpose distros
  - But doesn't yet solve the hard problems If you don't have dedicated flash for firmware
    - Can't use a universal OS image
    - Can't do runtime variable storage
    - How do you update GRUB in a fail-safe manner?
- For ARM Servers:
  - taken one step further by using ACPI and extending it with device tree properties
  - enables booting old long term stable distributions on newer server hardware

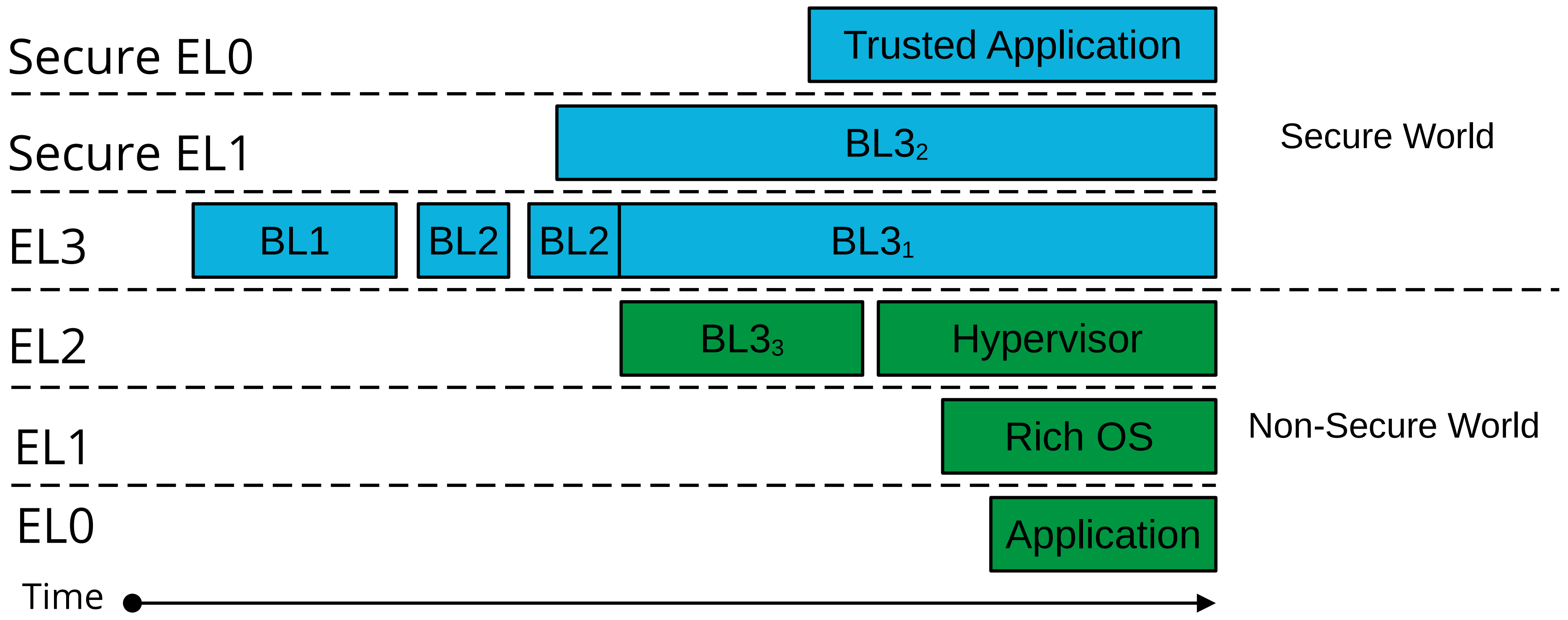


Boot Flow



Secure EL0
Trusted Application
Secure EL1
Trusted Execution Environment
Secure World
EL3
BootROM
SPL
Secure Monitor
EL2
Bootloader
Hypervisor
EL1
OS
Non-Secure World
EL0
Application
Time

Boot Flow (with ARM terms)



Beyond Booting: Further Watching

- Beyond "just" Booting: barebox Bells and Whistles (Ahmad Fatoum)
- HOWTO build a product with OP-TEE (Rouven Czerwinski)
- Runtime Services: Share System Resources on Multi-Processor System (Lionel Debieve)
- Device Tree: hardware description for everybody (Thomas Petazzoni)
- EBBR: Standard Boot for Embedded Platforms (Alexander Graf, Grant Likely)

Image Resources

- [MNT Reform]: https://www.crowdsupply.com/mnt/reform
- [Purism Librem 5]: https://puri.sm/products/librem-5/
- [EVK]: https://static6.arrow.com/aropdfconversion/3eb66c6ff328259ad060ed3a1cb5b2e34dcaa693/imx8mmevkhug.pdf
- [OMAP]: https://www.yumpu.com/en/document/view/18631588/chapter-26-initializationpdf
- [MX6-CLK]: https://www.shuzhiduo.com/A/mo5kwRnn5w/
- [ARM-EL]: https://developer.arm.com/architectures/learn-the-architecture/exception-model/execution-and-security-states

Thanks!

Questions?

https://www.pengutronix.de