# High-end Bootloaders

## Contents

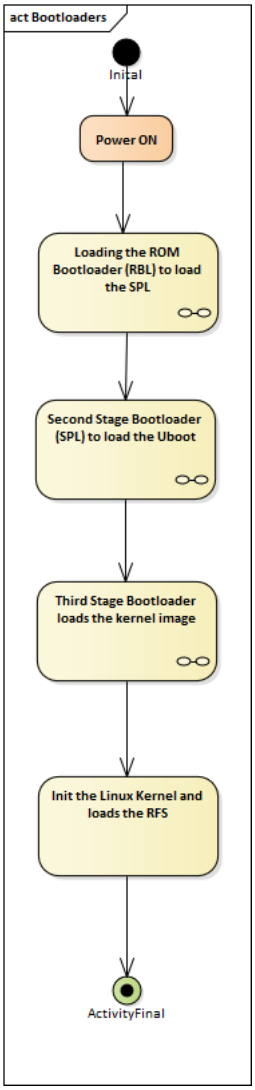


## Listings

# Contents

# 1 The Linux booting process

In order to make the Linux kernel up and running, it will be loaded according to the following stages/process




1. RBL - ROM bootloader or called the first stage bootloader
   This kind of bootloader resides in the ROM or flash, and It's most likely created by the SOC vendor.
   It's main purpose is to initialize the environment (e.g. Clock init, WDG timer, and the stack setup) for the second stage bootloader and then to run it.

   - It initializes the stack, clock, and the WDG timer
   - Search in available boot interfaces for the SPL
   - Loads the SPL (according to its image header) from the provided interface to the internal RAM memory.

- Executes and gives the control to the SPL.

Note: *You need to visit your SoC datasheet for boot sequence, and the different boot options and interfaces (e.g. SPI booting, Ethernet Booting, eMMC, SD memory, ...)*

2. SPL - Second stage bootloader or Memory bootloader
   It initializes the needed environment for the third stage bootloader and then run it.

   Let's say that the SPL resides in the SD card, then the RL will search on the file with a specific image header then loaded it to the internal SoC RAM. It's purpose

   - It initializes the console to print the boot messages (e.g. UART)
   - It initializes the external RAM memory preparing for the third stage bootloader to run
   - Copy the third stage bootloader image according to its image header (contains start address and image size) from the boot interface to the external RAM
   - Executes and gives the control to the third stage bootloader.

   Q: *Why do we need SPL ?*
   A: *The second stage bootloader has limited size reaches few Kbytes which can't be reached by Uboot size*

3. Third stage bootloader, like grub or Uboot in our case it's Uboot It's responsible for loading the Linux image and passes the command line arguments to it.

   - It initializes some peripherals like the USB or i2C or Ethernet (as configured) to load the kernel image from. (for example boot over ethernet)
   - Loads the Linux kernel for the defined sources to the External RAM, passes the command line parameters to it and then boot. Note: Booting and Configuring the uBoot at boot time can be modified by the uBoot configuration file **"uEnv.txt"** (you can override the default behaviors of uBoot by this file)

4. The Linux Kernel bootstrapper, it's main function is to decompress the Linux Kernel

5. The RFS - Root File System

# 2 Demo1 - Booting use case “raspberry pi 1 Model B Rev 2”

## 2.1 Info about the SoC used, and the booting options

The used HW is raspberry pi 1 Model B (2011.12 Model B Revision 2.0) board with the BC

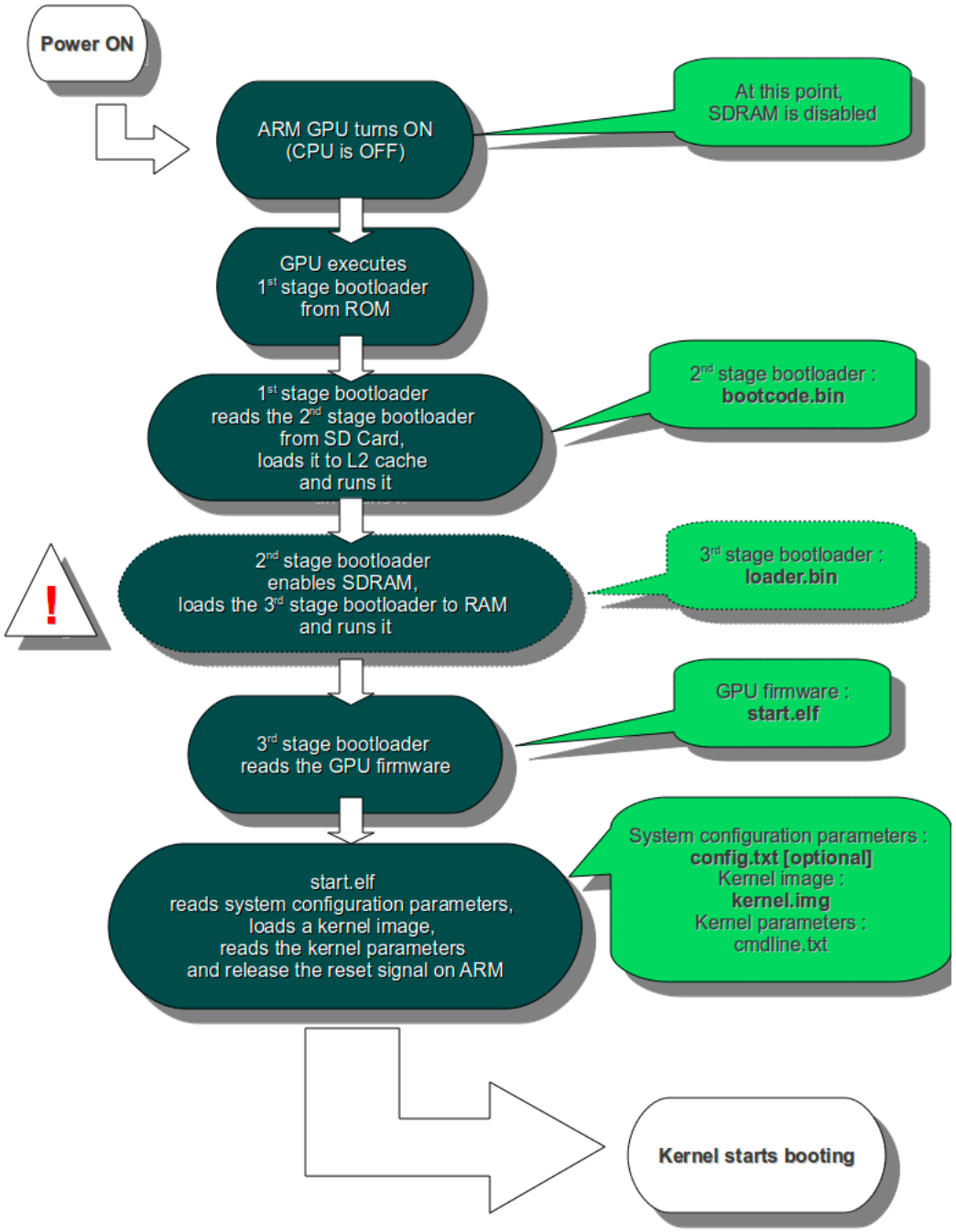
## 2.2 The Pi boot Process

1. After Power ON, the GPU starts and loads the ROM bootloader which loads the 2nd stage bootloader **”bootcode.bin”** into the L2 Cache (Till now the External RAM is not initialized yet)

2. the 2nd Stage bootloader enables the external SDRAM and loads 3rd stage bootloader **”loader.bin”** into it.

3. 3rd Stage bootloader runs the GPU firmware **”start.elf”**, which loads the kernel image based on the configuration file **”config.txt”** which has the information of the kernel to load

4. The 3rd stage bootloader reads **config.txt**, **cmdline.txt** and the **.dtb** (related to your board) If the dtb file exists, and then runs the **”kernel.img”** file. Note: In our case we will use the **config.txt** file run uBoot instead of the kernel.img file, then let the uBoot load the kernel.img.


Power ON
ARM GPU turns ON
(CPU is OFF)
At this point,
SDRAM is disabled
GPU executes
1st stage bootloader
from ROM
1st stage bootloader
reads the 2nd stage bootloader
from SD Card,
loads it to L2 cache
and runs it
2nd stage bootloader :
bootcode.bin
!
2nd stage bootloader
enables SDRAM,
loads the 3rd stage bootloader to RAM
and runs it
3rd stage bootloader :
loader.bin
3rd stage bootloader
reads the GPU firmware
GPU firmware :
start.elf
start.elf
reads system configuration parameters,
loads a kernel image,
reads the kernel parameters
and release the reset signal on ARM
System configuration parameters :
config.txt [optional]
Kernel image :
kernel.img
Kernel parameters :
cmdline.txt
Kernel starts booting

**For more info about the Pi boot sequence**

1. The boot flow
2. The contents of the boot partition, and the prebuild images for the Pi bootloaders here
3. The config.txt file

## 2.3 Building uBoot for Pi

### 2.3.1 Before we start - the prerequisites

- The needed HW
  1. raspberry pi board
  2. FTDI Usb-Serial converter (and download and install the FTDI driver) to communicate to the kernel in the board
- The needed tools
  1. The cross-platform compliers (will be discussed later)
  2. Uboot tools (for **mkimage** tool)

Listing 1: Uboot tools installation

```
$ sudo apt-get install u-boot-tools
```

  3. Needed libs to install in Linux

Listing 2: Config tools

```
$ sudo apt-get install bc build-essential git libncurses5-dev
    lzop perl
```

  4. Install the device tree complier

Listing 3: Config tools

```
$ sudo apt-get install device-tree-compiler

# or you can pick the latest source and install it your self
$ git clone git://git.kernel.org/pub/scm/utils/dtc/dtc.git -b
    v1.4.1
$ cd dtc/
$ make
$ export PATH=$HOME/dtc/:$PATH
```

5. Serial client SW like Putty or Minicom

The build process for uBoot is fairly simple, just three main steps;

1. setup the toolchain.
2. configure uBoot.
3. compile it!, that's it.

### 2.3.2 1. Setting up the Cross-Platform Complier

In order to compile Uboot or the Linux kernel for an embedded device, we need to cross-platform toolchain, for ARM we have variety set of toolchains as follow:

- Using the ARM foundation tools (baremetal tools)

Listing 4: Use ARM foundation cross-compliers

```
$ sudo apt-get install gcc-arm-none-eabi binutils-arm-none-eabi
    gdb-arm-none-eabi openocd
$ export CROSS_COMPILE=arm-none-eabi-
$ export ARCH=arm
```

- Using Linux foundation tools

Listing 5: Use Linux foundation ARM cross-compliers

```
$ apt-get install gcc-arm-linux-gnueabi
$ export CROSS_COMPILE=arm-linux-gnueabi-
$ export ARCH=arm
```

- Use Raspberrypi tools

Listing 6: Use Raspberrypi tools

```
cd $HOME
git clone git://github.com/raspberrypi/tools rpi-tools

export
    CROSS_COMPILE=$HOME/rpi-tools/arm-bcm2708/arm-bcm2708hardfp-linux-gnueabi/bin/arm-bcm2708har
export
    CROSS_COMPILE=$HOME/rpi-tools/arm-bcm2708/arm-bcm2708-linux-gnueabi/bin/arm-bcm2708-linux-gn
export
    CROSS_COMPILE=$HOME/rpi-tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/arm-linux-
```

- Using Linaro tools (We will be using this as it has a new version of GCC)

Listing 7: Use Linaro ARM cross-compliers

```
$ wget -c
    https://releases.linaro.org/components/toolchain/binaries/5.2-2015.11-2/arm-linux-gnueabihf/
$ tar xvf
    gcc-linaro-5.2-2015.11-2-x86_64_arm-linux-gnueabihf.tar.xz
$ ln -s gcc-linaro-5.2-2015.11-2-x86_64_arm-linux-gnueabihf
    gcc-linaro
```

For any used toolchain, you need to export the **ARCH**, and **CROSS_COMPILE** environmental varaibles.

Listing 8: Setup the environment

```
export ARCH=arm
export PATH=~/gcc-linaro/bin/:$PATH
export CROSS_COMPILE=arm-linux-gnueabihf-

# In Linaro, for 64bit Raspberry pi set them as follow
export ARCH=arm64
export PATH=~/gcc-linaro/bin/:$PATH
export CROSS_COMPILE=aarch64-linux-gnu-
```

Also note that every time you compile you need to set the mentioned environmental variables to the corresponding values. or you can make as follow.

Listing 9: Setup the environment

```
$ echo "export ARCH=arm" >> ~/export_compiler
$ echo "export PATH=~/gcc-linaro/bin/:$PATH" >> ~/export_compiler
$ echo "export CROSS_COMPILE=arm-linux-gnueabihf-" >>
    ~/export_compiler


$ source ~/export_compiler
```

**Note: compiler toolchains notations**

while using a toolchains, you might noticed that they have a notation or you can say naming conventions, and we can try to standardize as follow:
**arch[-vendor][-os]-abi**:

- **arch** is for architecture: arm, mips, x86, i686 ... etc.
- **vendor** is tool chain supplier: apple,
- **os** is for operating system: linux, none (bare metal)
- **abi** is for application binary interface convention: eabi, gnueabi, gnueabihf

Examples:
**arm-none-linux-gnueabi**; is he toolchain that can be installed in Debian-based systems using a package manager like apt (the package is called gcc-arm-linux-gnueabi). This toolchain targets the ARM architecture, has no vendor, creates binaries that run on the Linux operating system, and uses the GNU EABI.
**x86_64-w64-mingw32**; x86_64 architecture means AMD64, w64 is actually mingw-w64 used as a "vendor" here, mingw32 is the same as win32 API for gcc's perspective.
**arm-none-eabi**; This toolchain targets the ARM architecture, has no vendor, does not target an operating system (i.e. targets a "bare metal" system), and complies with the ARM eabi.

Finally you need to notice that they are all the same in our case.

### 2.3.3 2. Configure uBoot for Pi

First we need to fetch the uBoot source from its mainline repo.

Listing 10: Setup the environment

```
$ mkdir uboot-src
$ cd uboot-src
$ git clone -b 2016.11-toradex git://git.toradex.com/u-boot-toradex.git
$ cd u-boot-toradex
```

The next part is to configure uBoot for a certain board, for Pi 1 we use default configurations as follow

Listing 11: Setup the environment

```
# use the default configurations
$ make rpi_defconfig # for Pi 1 use rpi_defconfig
```

```
#compile the device tree
$

# or you can instead make you own configuration
$ make menu_config
```

Note: For other target you can use *make help* in your uBoot root source tree, it will list all the makefile targets. or use one of the following:

- rpi_0_defconfig -for Pi 0
- rpi_defconfig - for Pi 1
- rpi_2_defconfig - for Pi 2
- rpi_3_defconfig - for Pi 3

### 2.3.4 Compile uBoot

Listing 12: Setup the environment

```
$ make distclean
$ make u-boot.bin
```

## 2.4 U-boot environment parameters and the uEnv.txt

The Uboot environmental variables file uEnv.txt is used to configure or override the uBoot behavior.
The file is straight forward key-pair (name=value) configurations (environmental variables) and embedded commands.

Listing 13: Sample uEnv.txt file

```
# Some varaibles for the 0:2 is is the second partition in mcc0 interface
bootpart=0:2
bootdir=/boot
bootfile=zImage

# The console is passed to the kernel to print the debug messages
# at ttyO0 and with rate of 115200 (as our serial port configured)
console=ttyO0,115200n8

# Load addresses for fdt and kernel images
fdtaddr=0x88000000
loadaddr=0x82000000
fdtfile=fdt-file.dtb

#root file for the mmc device for loading the "rootfs" location
mmcroot=/dev/mmcblk0p2 ro
```

```
# Args passed to kernel
mmcargs=setenv bootargs console=${console}

#Loading the FDT file into the fdt address
loadfdt=load mmc ${bootpart} ${fdtaddr} ${bootdir}/${fdtfile}

# Loading the image
loadimage=load mmc ${bootpart} ${loadaddr} ${bootdir}/${bootfile}

# the main uBoot command, it runs the above commands
uenvcmd=if run loadfdt; then echo Loaded ${fdtfile}; if run loadimage;
    then run mmcargs; bootz ${loadaddr} - ${fdtaddr};
```

```
cpsw, usb_ether
Hit any key to stop autoboot:  0
mmc0 is current device
SD/MMC found on device 0
reading uEnv.txt
371 bytes read in 4 ms (89.8 KiB/s)
Loaded environment from uEnv.txt
Importing environment from mmc ...
Running uenvcmd ...
4394080 bytes read in 437 ms (9.6 MiB/s)
24884 bytes read in 25 ms (971.7 KiB/s)
## Booting kernel from Legacy Image at 82000000 ...
   Image Name:   Angstrom/3.8.13/beaglebone
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    4394016 Bytes = 4.2 MiB
   Load Address: 80008000
   Entry Point:  80008000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 88000000
   Booting using the fdt blob at 0x88000000
   Loading Kernel Image ... OK
   Using Device Tree in place at 88000000, end 88009133

Starting kernel ...
```

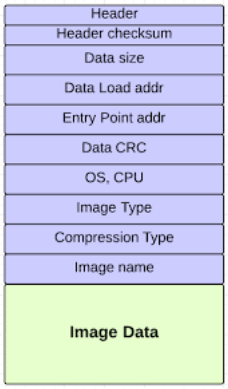
As you can see, the line "reading uEnv.txt", and then in order loading the kernel and the FDT into memory.

## 2.5 boot.scr.uimg File

## 2.6 U-boot Kernel Image types

The Linux kernel can be formed into different images for the uBoot to understand as following:

- Linux Kernel RAW image (called uImage) and this is not used by uBoot
- uBoot legacy compressed image (called zImage), and it's the same as uImage but with the uBoot header create by the uBoot tool *mkimage* (discussed in the example)




Note: the Legacy Format Image header can be found in "/include/image.h" in the uBoot source tree, and it contains:

Listing 14: Legacy Format Image header struture in /include/image.h

```
/*
 * Legacy format image header,
 * all data in network byte order (aka natural aka bigendian).
 */
typedef struct image_header {
   uint32_t ih_magic; /* Image Header Magic Number */
   uint32_t ih_hcrc; /* Image Header CRC Checksum */
   uint32_t ih_time; /* Image Creation Timestamp */
   uint32_t ih_size; /* Image Data Size  */
   uint32_t ih_load; /* Data Load Address  */
   uint32_t ih_ep;    /* Entry Point Address  */
   uint32_t ih_dcrc; /* Image Data CRC Checksum */
   uint8_t    ih_os;     /* Operating System   */
   uint8_t    ih_arch; /* CPU architecture  */
   uint8_t    ih_type; /* Image Type     */
   uint8_t    ih_comp; /* Compression Type  */
   uint8_t    ih_name[IH_NMLEN]; /* Image Name   */
} image_header_t;
```

- FIT Image, or the uBoot new image format which is used in Verified (secure) booting

## 2.7 The Flatten Device Tree

It's a file used to describe all the platform (board) peripherals in a static way in order for the Linux Kernel to know them. and they are board specific.
the details for every board is detailed in a DTS (device tree source), and then complied with a device tree complier (dtc) to form the Device Tree Binary (DTB)

Listing 15: FDT syntax example

```
/dts-v1/;
/include/ "common.dtsi";

/ {
    node1 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        a-byte-data-property = [0x01 0x23 0x34 0x56];
        cousin: child-node1 {
            first-child-property;
            second-child-property = <1>;
            a-string-property = "Hello, world";
        };
        child-node2 {
        };
    };
    node2 {
        an-empty-property;
        a-cell-property = <1 2 3 4>; /* each number (cell) is a uint32 */
        child-node1 {
            my-cousin = <&cousin>;
        };
    };
};

/node2 {
    another-property-for-node2;
};
```

For more info about how to write a device tree, refer to The used HW is the Pi documentation. Note: The DTS is detailing the passive elements on the board which means something like the USB will not be listed there, as the USB device is not in the device tree as it has the ability to announce the Linux kernel its existence at runtime.

## 2.8 Serial booting

Without using SD Card, Serial booting means transferring the image to the board through the serial port (UART), so the host system will contain the bootloader, Linux image, DTB file, and RFS.

And you need to note that here we will use a RAM based filesystem to work with our kernel.

**Note: Serial file transfer protocols**

In order to transfer data/files through UART, so we might be using one of the serial file transfer protocols like xModem, yModem, or zModem, kermit... .

So after booting up with the RBL, it listens on the UART using the xmodem protocol for example waiting for the first stage bootloader from the host machine, if there is a response then it will load the first stage bootloader into it's internal RAM, then the first stage bootloader will wait again through the xModem to get the second stage bootloader (e.g. uBoot), and the same for the RFS.

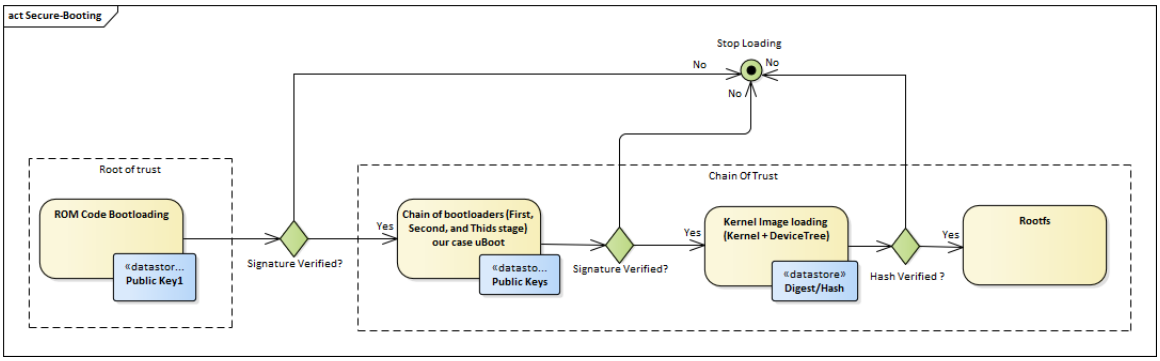
### 2.8.1 Serial Booting using Raspberry pi using minicom

## 2.9 TFTP Booting

Using TFTP protocol you can transfer the Linux kernel image to your board from your host. and your board will act as TFTP client with a defined IP address, and your host will be the server with a known IP address.

In order to do that we will put the **SPL, uBoot and uEnv.txt** in the SD Card, and the Linux Kernel, DTB file, and RFS in our host machine at this location /var/lib/tftpboot.

# 3 Secure booting




## 3.1 Chain of trust

The chain of trust is identified as a security verification method to verify a chain of processes from starting the CPU till we reach the applications up and running in a chained order which every node verifies that the next node will be trusty loaded. and the security is established by requiring that no code will be executed by firmware unless it has been signed by a trusted key.

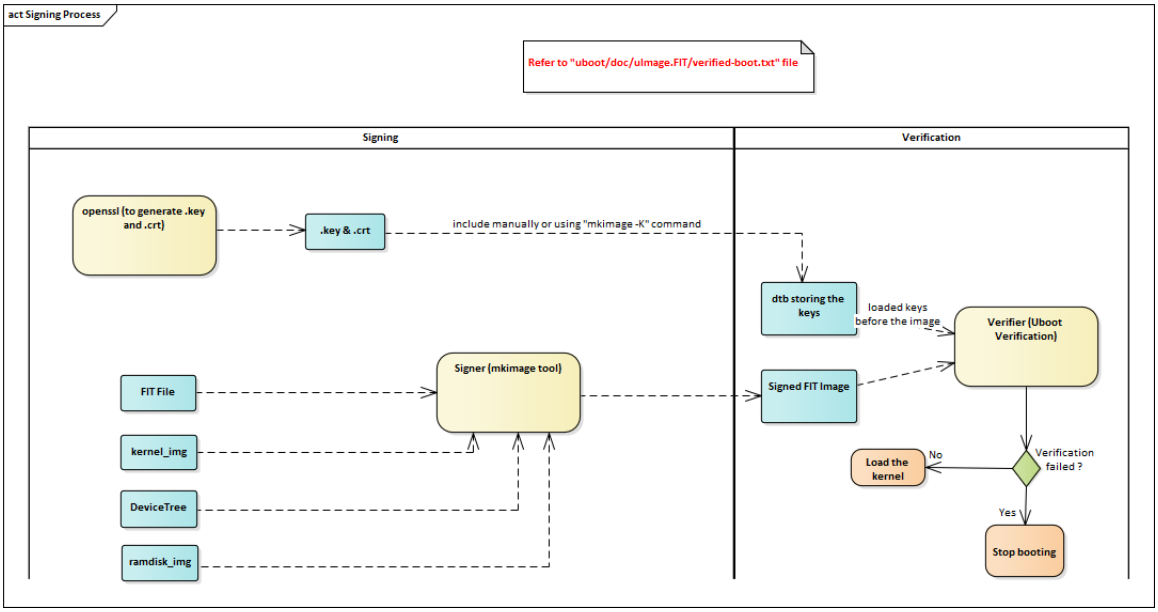
The chain of trust uses a public key cryptography to verify the signature of the next loaded node, if the signature is invalid, then this mean that the next node is intentionally modified.

## 3.2 Root of trust

In order to achieve a secure chain of trust, you need to make sure that the root that initialize all the secure boot process is made by a secure root/source. The root of trust is ideally based on a hardware-validated boot process to ensure the system can only be started using code from an immutable source.

## 3.3 Uboot Verified Boot - Demo on Pi

Note: Our example with be base on RSA encryption algorithm.
To setup uBoot in the secure process, you need to follow the following fair and simple steps:




1. Configuring uBoot. Before compiling uBoot. you need to configure uBoot to use FIT images and RSA algorithm. The following parameters should be configured:

   - Enable FIT: **CONFIG_FIT** - enable support for the FIT uImage format
   - Enable verified boot:
     - CONFIG_FIT_SIGNATURE: enables signature verification of FIT images
     - CONFIG_RSA: enables the RSA algorithm used for FIT image verification
   - Enable FDT: CONFIG_OF_CONTROL, CONFIG_OF_SEPARATE

2. Describe a FIT image: A FIT image is a extension of the old legacy image and it is tree like image that contains all files (called sub-images) required to boot a kernel, such as the kernel image, device-tree-blob and possibly a ramdisk (initrd). It follows the DeviceTree language, just create a **.its** file and follow the following example:

Listing 16: Simple FIT file

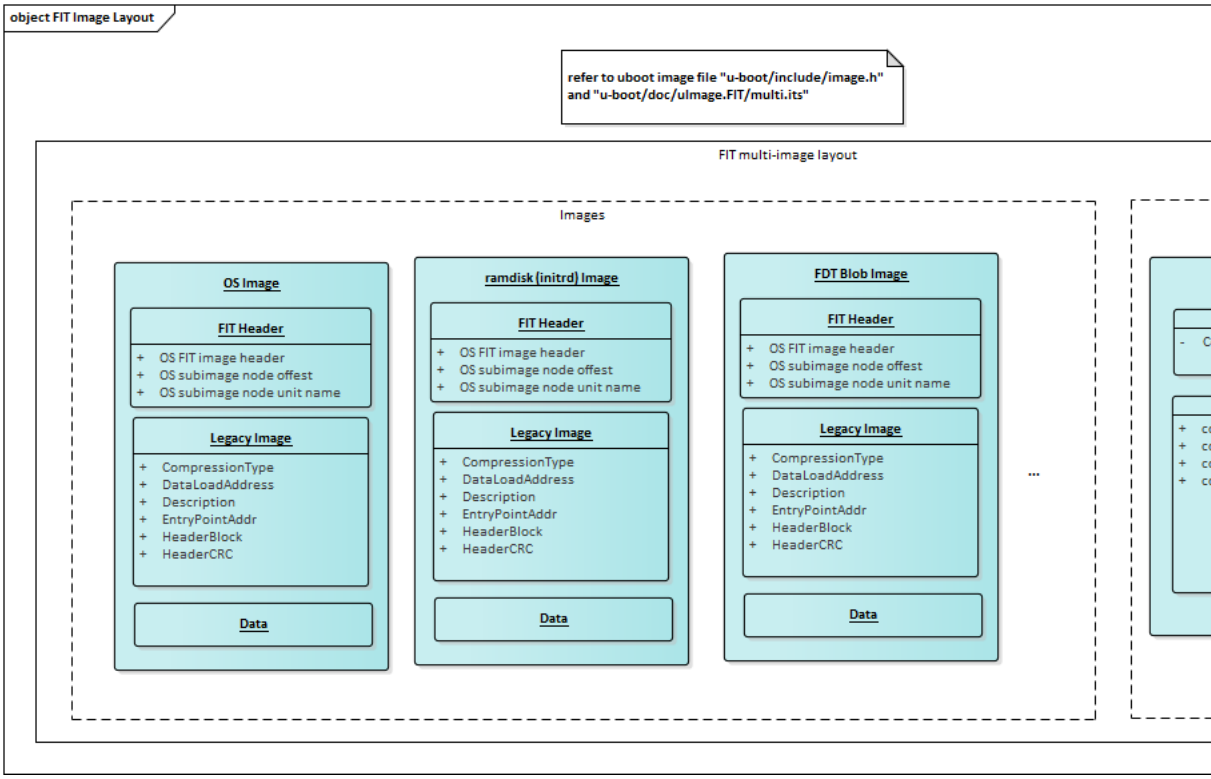
```
/*
 * Simple U-Boot uImage source file containing a single kernel and
     FDT blob
 */

/dts-v1/;
```

```
/ {
   description = "Simple image with single Linux kernel and FDT
       blob and ramdisk";
   #address-cells = <0x1>;

   images {
      kernel@1 {
         description = "Vanilla Linux kernel";
         data = /incbin/("/boot/zImage");
         type = "kernel";
         arch = "arm";
         os = "linux";
         compression = "none";
         load = <0x00008000>;
         entry = <0x00008000>;
         hash@1 {
            algo = "crc32";
         };
         hash@1 {
            algo = "sha1";
         };
      };
      fdt@1 {
         description = "Flattened Device Tree blob";
         data = /incbin/("/boot/dtb/armada-388-clearfog.dtb");
         type = "flat_dt";
         arch = "arm";
         compression = "none";
         hash@1 {
            algo = "sha1";
         };
         hash@1 {
            algo = "crc32";
         };
      };
   };

    ramdisk@1{
        description = "Ramdisk Image";
         data = /incbin/("/boot/ramdisk.image.gz");
         type = "ramdisk";
         arch = "arm";
         os = "linux";
         compression = "none";
         load = <0x00800000>;
         entry = <0x00800000>;
         hash@1 {
            algo = "sha1";
         };
```

```
        hash@1 {
           algo = "crc32";
        };
    };

   configurations {
      default = "conf@1";
      conf@1 {
         description = "Boot Linux kernel with FDT blob and
             ramdisk";
         kernel = "kernel@1";
         fdt = "fdt@1";
            ramdisk = ramdisk@1
      };
   };
};
```

Note: You can refer to **"doc/uImage.FIT/source_file_format.txt"** in the uBoot documentation folder, as a full description to create FIT formatted file.

and you can refer to the FIT image layout as follow:




It contains two main parts, images with a new FIT header to describe the image and its offset, and a configuration set

3. Use the Uboot tool "mkimage" to create a FIT image from the defined FIT description file in the previous step:

Listing 17: Generate a FIT image

```
mkimage -f kernel_fdt.its fitImage
```

4. Generate the key-pair using a signer *openssl*, the generated file will be a key ".key" and certificate ".crt".

Listing 18: Generate Keys

```
mkdir keys
openssl genpkey -algorithm RSA -out keys/dev.key -pkeyopt
    rsa_keygen_bits:2048 -pkeyopt rsa_keygen_pubexp:65537
openssl req -batch -new -x509 -key keys/dev.key -out keys/dev.crt
```

5. Sign the generated FIT image. you can attach the keys by extending the FIT file as follow, and then regenerate the FIT image

Listing 19: Extend the FIT file

```
/*
 * Simple U-Boot uImage source file containing a single kernel and
     FDT blob
 */

/dts-v1/;

/ {
   description = "Simple image with single Linux kernel and FDT
       blob and ramdisk";
   #address-cells = <0x1>;

   images {
      kernel@1 {
         description = "Vanilla Linux kernel";
         data = /incbin/("/boot/zImage");
         type = "kernel";
         arch = "arm";
         os = "linux";
         compression = "none";
         load = <0x00008000>;
         entry = <0x00008000>;
         hash@1 {
            algo = "sha1";
         };
         signature@1 {
            algo = "sha1,rsa1048";
                key-name-hint = "dev";
         };
      };
      fdt@1 {
         description = "Flattened Device Tree blob";
         data = /incbin/("/boot/dtb/armada-388-clearfog.dtb");
         type = "flat_dt";
         arch = "arm";
         compression = "none";
         hash@1 {
            algo = "sha1";
         };
         signature@1 {
            algo = "sha1,rsa2048";
                key-name-hint = "dev";
         };
      };
   };

    ramdisk@1{
        description = "Ramdisk Image";
```

```
        data = /incbin/("/boot/ramdisk.image.gz");
        type = "ramdisk";
        arch = "arm";
        os = "linux";
        compression = "none";
        load = <0x00800000>;
        entry = <0x00800000>;
        hash@1 {
           algo = "sha1";
        };
        signature@1 {
           algo = "sha1,rsa1048";
               key-name-hint = "dev";
        };
    };

   configurations {
      default = "conf@1";
      conf@1 {
         description = "Boot Linux kernel with FDT blob and
             ramdisk";
         kernel = "kernel@1";
         fdt = "fdt@1";
            ramdisk = ramdisk@1
      };
   };
};
```

Listing 20: Generate the signed kernel image

```
mkimage -f kernel_fdt_signed.its -k keys fitImage
```

6. You can skip previous step by using the following "-f and -F" options instead, to sign the image after creating it.

   The -F option is used "Indicates that an existing FIT image should be modified. No dtc compilation is performed and the -f flag should not be given. This can be used to sign images with additional keys after initial image creation." (refer to mkimage man page here)

7. Include the public key in uBoot device tree file before the verification of the image, the uBoot expects the keys to be loaded first, that's why we need to include the public key in the device tree file, and you we can achieve that by using the "-K capital k" options:

Listing 21: Include the key in the dtb

```
mkimage -f kernel_fdt_signed.its -k keys -K dts/dt.dtb -r image
    fitImage
```

Note: the -r option is used to tell uBoot at boot time which signature is used for certain image, otherwise uBoot will boot any signed, unsigned or wrongly signed images. if you used a signed configuration you need also to include here. Refer to mkimage man page: "Specifies that keys used to sign the FIT are required. This means that they must be verified for the image to boot. Without this option, the verification will be optional (useful for testing but not for release)."

8. Finally try your target.

**Signed Configurations to avoid mix and match attack**

While signing images is useful, it does not provide complete protection against several types of attack. For example, it it possible to create a FIT with the same signed images, but with the configuration changed such that a different one is selected (mix and match attack). It is also possible to substitute a signed image from an older FIT version into a newer FIT (roll-back attack).

refer to the uBoot configuration: "uboot/doc/uImage.FIT/signature.txt"

## 3.4 Playing with uboot commands

1. Load command

```
U-Boot# help load
load - load binary file from a filesystem

Usage:
load <interface> [<dev[:part]> [<addr> [<filename> [bytes [pos]
    - Load binary file 'filename' from partition 'part' on devi
        type 'interface' instance 'dev' to address 'addr' in mem
      'bytes' gives the size to load in bytes.
      If 'bytes' is 0 or omitted, the file is read until the en
      'pos' gives the file byte position to start reading from.
      If 'pos' is 0 or omitted, the file is read from the start
      All numeric parameters are assumed to be decimal,
      unless specified otherwise using a leading "0x".
U-Boot# load mmc 0:2 0x82000000 /boot/uImage
4310624 bytes read in 520 ms (7.9 MiB/s)
```

2. Memory Dump command

```
U-Boot# help md
md - memory display

Usage:
md [.b, .w, .l] address [# of objects]
U-Boot# md 0x82000000 4
82000000: 56190527 ba031e65 ca8df751 20c64100    '..Ve...Q....A
U-Boot# md 0x82000000 10
Unknown command 'md' - try 'help'
U-Boot# md 0x82000000 10
82000000: 56190527 ba031e65 ca8df751 20c64100    '..Ve...Q....A
82000010: 00800080 00800080 9c919f36 00020205    ........6.....
82000020: 73676e41 6d6f7274 382e332f 2f33312e    Angstrom/3.8.1
82000030: 67616562 6f62656c 0000656e 00000000    beaglebone....
U-Boot#
```

3. imi Comamnd

```
U-Boot# help imi
iminfo - print header information for application image

Usage:
iminfo addr [addr ...]
    - print header information for application image starting a
      address 'addr' in memory; this includes verification of t
      image contents (magic number, header and payload checksum
U-Boot# imi 0x82000000

## Checking Image at 82000000 ...
   Legacy image found
   Image Name:   Angstrom/3.8.13/beaglebone
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    4310560 Bytes = 4.1 MiB
   Load Address: 80008000
   Entry Point:  80008000
   Verifying Checksum ... OK
U-Boot#
```

4. Run command

For the list of uBoot command visit the uBoot documentation.