MASTER OF SCIENCE
IN ENGINEERING

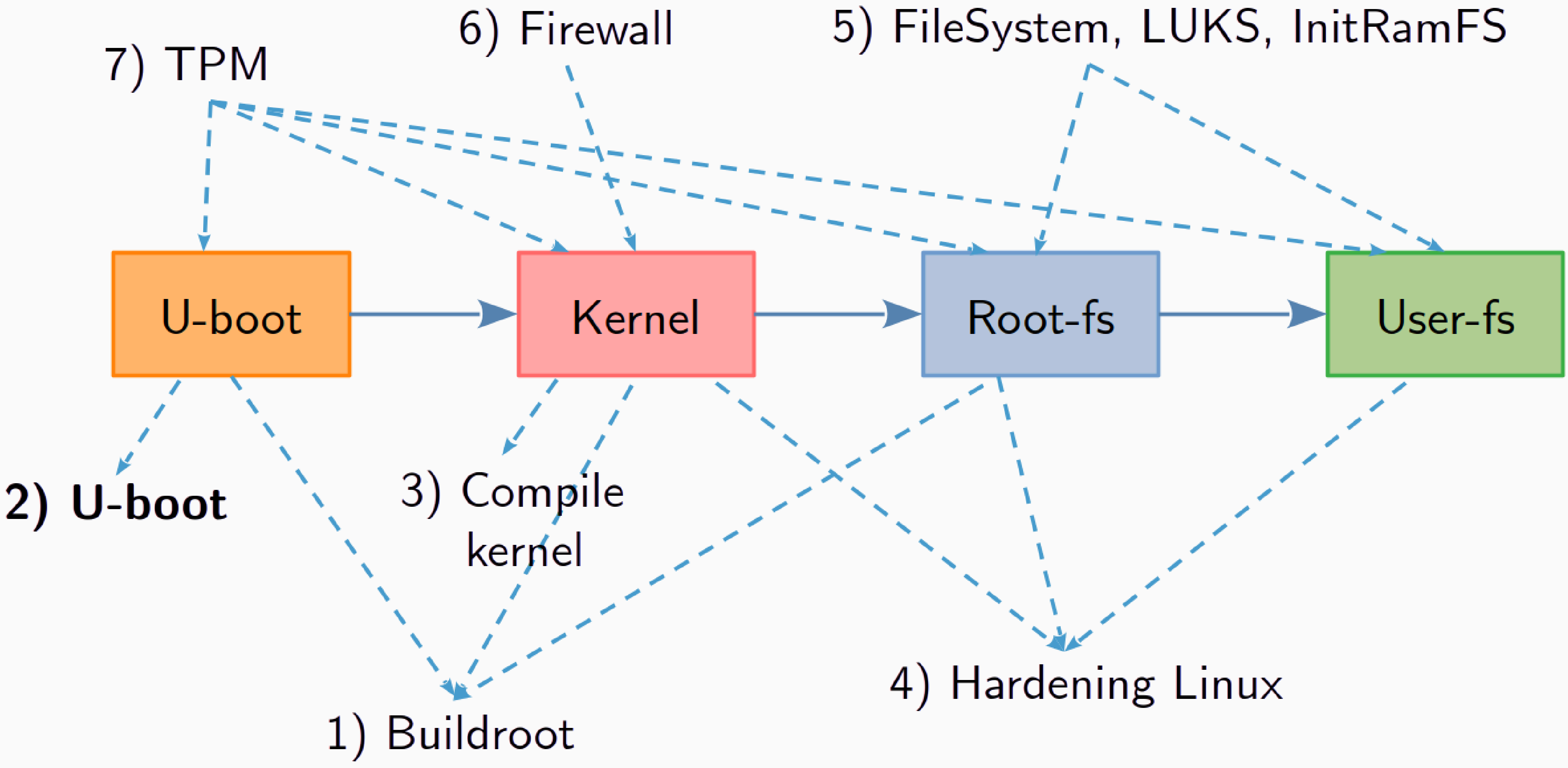
U-Boot & image formats

Florent Glück, Jean-Roland Schuler

October 18, 2025

swissuniversities

Overview


7) TPM
6) Firewall
5) FileSystem, LUKS, InitRamFS
U-boot
Kernel
Root-fs
User-fs
2) U-boot
3) Compile kernel
4) Hardening Linux
1) Buildroot

Introduction

What is a bootloader?

- Purpose of a bootloader:
  - basic hardware initialization
  - loading (+ possibly decompression) of an application binary, usually a kernel from flash storage, network, etc.
  - execution of the binary
- Most bootloaders provide a shell with various commands:
  - loading data from storage or network, memory inspection, write to flash, boot kernel, hardware diagnostics, etc.

Initial boot sequence on embedded systems

- CPU has an integrated 1st stage boot code in ROM (vendor specific):
  - BootROM on NanoPi, BootROM on AT91, "ROM code" on OMAP, etc.
  - exact details are CPU-dependent
- Boot code loads 2nd stage bootloader from storage device into internal SRAM (DRAM not initialized yet):
  - storage device typically: MMC, NAND, SD, SPI flash, UART, etc.
- 2nd stage bootloader:
  - limited in size due to hardware constraints (SRAM size)
  - provided either by CPU vendor or through community projects
  - initializes DRAM and low-level hardware
  - loads 3rd stage bootloader, typically U-Boot, into RAM

Full boot sequence on NanoPi NEO Plus2

1. BootROM[1] code loads from flash (SD card first, then eMMC) (sector 16) `sunxi-spl.bin` (Secondary Program Loader) firmware into SRAM (32KB max), then executes it
2. `sunxi-spl.bin` initializes DRAM and low-level hardware, then loads `u-boot.itb` (U-Boot) from flash into DRAM and executes it
3. U-Boot performs more hardware initialization (clocks, controllers, etc.), then loads a Linux kernel image and a Device Tree Blob (DTB) from flash into DRAM
4. U-Boot passes the DTB and kernel arguments to the Linux kernel, then executes it
5. The Linux kernel initializes the rest of the hardware, configures itself, then mounts the various filesystems (rootfs, tmpfs, userfs, etc.) and finally executes **`init`** the first user process

[1] https://github.com/ARM-software/u-boot/blob/master/board/sunxi/README.sunxi64

Das U-Boot bootloader

- The Universal Boot Loader, simply called **U-Boot**
- Open-source[1] generic-purpose bootloader for embedded systems
- By far the most used bootloader in the industry
- U-Boot's homepage: https://u-boot.org/
- U-Boot' source code: https://github.com/u-boot/u-boot

[1] Licensed as GPLv2

U-Boot basics

U-Boot configuration

- To configure U-Boot in Buildroot:

```
make uboot-menuconfig
```

  Upon exit, configuration saved in `output/build/uboot-xx/.config`

- U-Boot default values are defined in `Kconfig` files scatered throughout `output/build/uboot-xx/`
- Buildroot's `defconfig` file can contain U-Boot config options overriding default values
- If a fragment file for U-Boot is defined in Buildroot `(Bootloaders ---> Additional config fragment files)`, its content overrides any other values
  - config after applying fragment stored in `output/build/uboot-xx/.config`

U-Boot configuration file

- U-Boot configuration file is saved in output/build/uboot-xx/.config
- It can be inspected, but do not modify it:

```
$ more output/build/uboot-2020.10-rc5/.config
#
# Automatically generated file; DO NOT EDIT.
# U-Boot 2020.10-rc5 Configuration
#

#
# Compiler: aarch64-buildroot-linux-gnu-gcc.br_real (Buildroot 2022.08.3) 11.3.0
#
CONFIG_CREATE_ARCH_SYMLINK=y
# CONFIG_ARC is not set
CONFIG_ARM=y
# CONFIG_M68K is not set
# CONFIG_MICROBLAZE is not set
# CONFIG_MIPS is not set
# CONFIG_NDS32 is not set
```

Building U-Boot

- To specifically build U-Boot in Buildroot:

```
make uboot-rebuild
```

- Once U-Boot's build is complete, two files are created:

```
output/images/u-boot.itb
output/images/boot.scr
```

- If a SD card image must be generated, don't forget to run `make` once U-Boot is built

U-Boot prompt

A key press during the boot process allows to enter a command prompt:

```
U-Boot SPL 2020.10-rc5 (Sep 08 2025 - 18:39:41 +0200)
DRAM: 512 MiB
Trying to boot from MMC1

U-Boot 2020.10-rc5 (Sep 08 2025 - 18:39:41 +0200) Allwinner Technology

CPU:   Allwinner H5 (SUN50I)
Model: FriendlyARM NanoPi NEO Plus2
DRAM:  512 MiB
MMC:   mmc@1c0f000: 0, mmc@1c10000: 2, mmc@1c11000: 1
Loading Environment from FAT... OK
In:    serial
Out:   serial
Err:   serial
Net:   phy interface7
eth0: ethernet@1c30000

Hit any key to stop autoboot:  0
=>
```

U-Boot commands (1/2)

- Type help to list the supported commands
- Type help <cmd> to display a command's help

```
boot       - boot default, i.e., run 'bootcmd'
booti      - boot Linux kernel 'Image' format from memory
bootm      - boot application image (FIT) from memory
bootp      - boot image via network using BOOTP/TFTP protocol
bootz      - boot Linux kernel 'zImage' format from memory

ext2load  - load binary file from a Ext2 filesystem
ext2ls    - list files in a directory (default /)

ext4load  - load binary file from a Ext4 filesystem
ext4ls    - list files in a directory (default /)
ext4size  - determine a file's size

fatinfo   - print information about filesystem
fatload   - load binary file from a dos filesystem
fatls     - list files in a directory (default /)
fatmkdir  - create a directory
fatrm     - delete a file
fatsize   - determine a file's size
```

U-Boot commands (2/2)

```
iminfo     - print header information for application image (FIT)
md         - memory display
mm         - memory modify (auto-incrementing address)
mmc        - MMC sub system
mmcinfo    - display MMC info
ping       - send ICMP ECHO_REQUEST to network host
printenv   - print environment variables
reset      - Perform RESET of the CPU
run        - run commands in an environment variable
saveenv    - save environment variables to persistent storage
source     - run script from memory
tftpboot   - boot image via network using TFTP protocol
...
```

Examples:

```
ext4ls mmc 0:1     # list files from ext4 filesystem located on SD card's 1st partition
                   # NOTE: writing "mmc 0:1" is equivalent to writing "mmc 0"
ext2ls mmc 0:2     # list files from ext2 filesystem located on SD card's 2nd partition
fatls mmc 1:1      # list files from fat filesystem located on eMMC's 1st partition
```

U-Boot environment

- U-Boot can be configured through **environment variables**:
  - affect various commands' behavior
  - very useful to save frequently used commands
  - simple scripts can be executed in U-Boot
- U-Boot environment variables are stored in the "environment" (stored on flash, SD card, EEPROM, etc.):
  - defined in the board configuration file
- Environment is loaded to RAM during U-Boot startup sequence:
  - can be modified and saved back for persistence
- Whether the environment can be saved to physical storage or not depends on U-Boot configuration

U-Boot environment variables

Example of U-Boot variables:

```
bootargs  - contains the arguments passed to the Linux kernel
bootcmd   - variable executed when 'boot' command is executed
bootdelay - delay before boot starts

ethaddr   - Ethernet MAC address of the first interface (can only be set once)

ipaddr    - client IP address for the tftpboot command
serverip  - server IP address for the tftpboot command

filesize  - size of the last file downloaded with 'tftpboot', 'bootp' or 'dhcp' commands

...
```

Commands related to environment variables

- `printenv` ⟶ displays the values of all variables
- `printenv <var>` ⟶ displays the value of a variable
- `setenv <var> <value>` ⟶ changes the value of a variable
- `setenv <var>` ⟶ deletes a variable
- `editenv <var>` ⟶ edits the value of a variable
- `saveenv` ⟶ save environment variables to persistent storage
- `echo $x` ⟶ displays the value of variable `x`
- `run x` ⟶ executes commands stored in variable `x`

Examples:

```
=> printenv bootcmd
bootcmd=run distro_bootcmd

=> echo $bootdelay
2
```

```
=> setenv bootargs root=/dev/mmcblk0p2 ro

=> saveenv
Saving Environment to FAT... OK
```

U-Boot environment

- Reminder, to configure U-Boot in Buildroot:

```
make uboot-menuconfig
```

- U-Boot environment configuration:

  `make uboot-menuconfig` → `Environment`

```
[ ] Environment is not stored
[ ] Environment in EEPROM
[*] Environment is in a FAT filesystem
[ ] Environment is in a EXT4 filesystem
[ ] Environment in flash memory
[ ] Environment in an MMC device
[ ] Environment in a NAND device
...
```

U-Boot and the Linux kernel

How to boot the Linux kernel?

U-Boot command to boot Linux kernel depends on kernel image format[1]:

- `Image`: uncompressed, primarily for ARM64
  - No decompression needed at runtime → faster boot
  - Uncompressed → uses more storage space, slower transfer
  - Requires the **`booti`** command
- `zImage`: compressed, primarily for ARM32
  - Typically created using: `cat Image | gzip -9 > zImage`
  - Saves storage space, faster transfer
  - Self-decompression → slightly slower boot, more CPU usage
  - Requires the **`bootz`** command

[1] There is also `vmlinux` which is in the original ELF format, but it's usually not used

Booting the Linux kernel: booti

```
=> help booti
booti - boot Linux kernel 'Image' format from memory

Usage:
booti [addr [initrd[:size]] [fdt]]
    - boot Linux flat or compressed 'Image' stored at 'addr'
  The argument 'initrd' is optional and specifies the address
  of an initrd in memory. The optional parameter ':size' allows
  specifying the size of a RAW initrd.
  Currently only booting from gz, bz2, lzma and lz4 compression
  types are supported. In order to boot from any of these compressed
  images, user have to set kernel_comp_addr_r and kernel_comp_size environment
  variables beforehand.
  Since booting a Linux kernel requires a flat device-tree, a
  third argument providing the address of the device-tree blob
  is required. To boot a kernel with a device-tree blob but
  without an initrd image, use a '-' for the initrd argument.
```

Usage example: `booti 0x40080000 - 0x4FA00000`

Booting the Linux kernel manually: example

1. Set the necessary kernel arguments using the `bootargs` variable:

```
setenv bootargs console=ttyS0,115200 earlyprintk root=/dev/mmcblk0p2 rootwait
```

2. Load the kernel image to DRAM (here, from a VFAT filesystem):

```
fatload mmc 0:1 $kernel_addr_r Image
```

3. Load the DTB to DRAM:

```
fatload mmc 0:1 $fdt_addr_r sun50i-h5-nanopi-neo-plus2.dtb
```

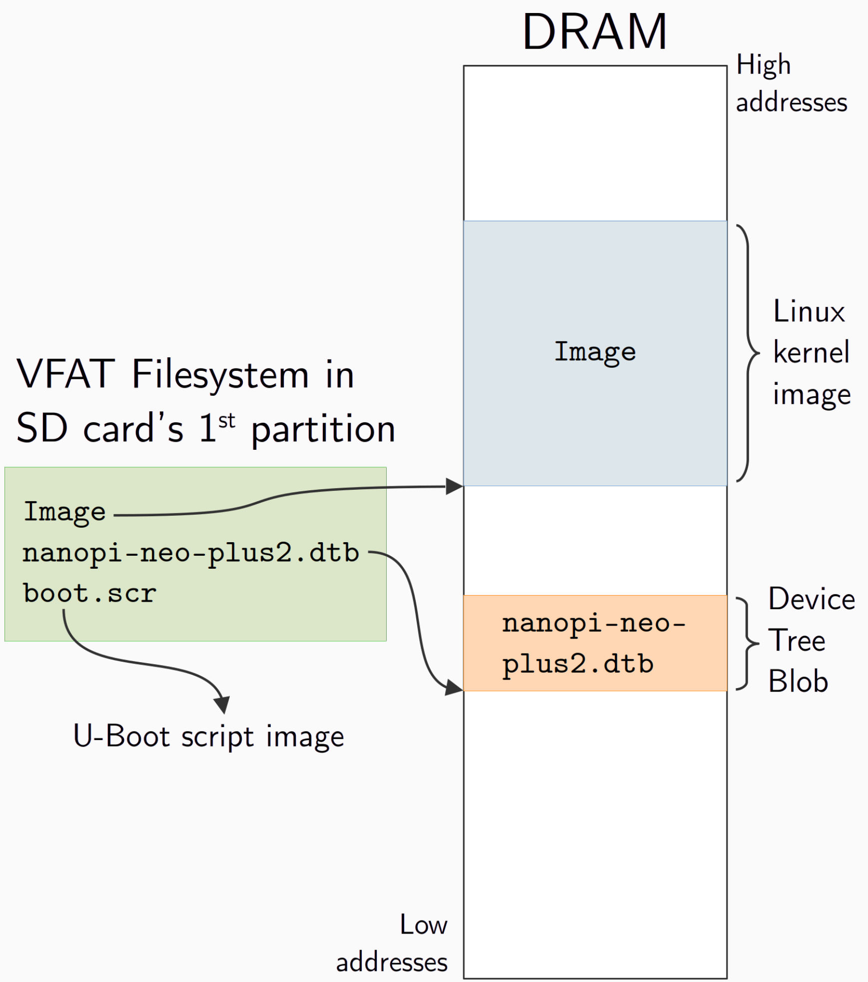
4. Boot the kernel by specifying the kernel and DTB addresses in DRAM:

```
booti $kernel_addr_r - $fdt_addr_r
```

Loading kernel and Device Tree Blob

```
=> fatls mmc 0:1
39475712   Image
    22620   sun50i-h5-nanopi-neo-plus2.dtb
      279   boot.scr

=> echo $kernel_addr_r
0x40080000

=> echo $fdt_addr_r
0x4FA00000
```

```
fatload mmc 0:1 $kernel_addr_r Image
fatload mmc 0:1 $fdt_addr_r sun50i-h5-
nanopi-neo-plus2.dtb
```



Booting the Linux kernel via a script: example

1. Set the necessary kernel arguments using the `bootargs` variable:

```
setenv bootargs 'console=ttyS0,115200 earlyprintk root=/dev/mmcblk0p2
rootwait'
```

2. Insert commands into the `bootcmd` variable:

```
setenv bootcmd 'fatload mmc 0:1 $kernel_addr_r Image ; fatload mmc 0:1
$fdt_addr_r sun50i-h5-nanopi-neo-plus2.dtb ; booti $kernel_addr_r -
$fdt_addr_r'
```

3. Run the `boot` command (which runs the `bootcmd` variable):

```
boot
```

U-Boot scripts

U-Boot scripts

U-Boot scripts can be:

- Created on the host
- Compiled on the host (using `mkimage`)
- Loaded by U-boot
- Executed by U-Boot (using `source`)

This is another way of booting a Linux kernel

U-Boot script: example

buildroot/board/friendlyarm/nanopi-neo-plus2/boot.cmd is an example of U-Boot script:

```
1 setenv bootargs console=ttyS0,115200 earlyprintk root=/dev/mmcblk0p2 rootwait
2 fatload mmc 0:1 $kernel_addr_r Image
3 fatload mmc 0:1 $fdt_addr_r sun50i-h5-nanopi-neo-plus2.dtb
4 booti $kernel_addr_r - $fdt_addr_r
```

- L1: specifies the kernel command line arguments U-Boot will pass to the Linux kernel
- L2: loads from the VFAT filesystem located on the 1st partition of the SD card, the kernel image `Image` at DRAM address `kernel_addr_r`
- L3: Loads the DTB (Device Tree Blob) `sun50i-h5-nanopi-neo-plus2.dtb` from the same location as `Image` at DRAM address `fdt_addr_r`
- L4: Boot the Linux kernel located at DRAM address `kernel_addr_r` using the DTB at DRAM address `fdt_addr_r`

Compiling an U-Boot script

- Example extracted from `buildroot/board/friendlyarm/nanopi-neo-plus2/post-build.sh`
- This line from `post-build.sh` compiles `boot.cmd` into `boot.scr`:

```
mkimage -C none -A arm64 -T script -d boot.cmd boot.scr
```

  - Arguments:

`-C`	compression type
`-A`	hardware architecture
`-T`	image type
`-d`	input file

Booting the Linux kernel, using a compiled script

- Say `boot.scr` is the compiled version of `boot.cmd` seen previously and is stored on a SD card
- Insert commands to load and execute the script into `bootcmd`:

```
setenv bootcmd 'fatload mmc 0:1 0x4FC00000 boot.scr ; source 0x4FC00000'
```

- Run the `boot` command:

```
boot
```

Flattened Device-Tree (FDT) & Flattened Image (FIT)

Flattened Device-Tree (FDT)

- The Flattened Device Tree (FDT) **describes the hardware**
  - introduced in Linux kernel 2.6.24 in 2008
- Required by the Linux kernel for its configuration
- The FDT is composed of two files:
  - `.dts`: **Device Tree Source** file (text) → FDT' **source** code
  - `.dtb`: **Device Tree Blob** file (binary) → **compiled** version of the `.dts`
- The `dtc` compiler is required to compile a `.dts` into a `.dtb`, e.g.:

```
dtc board.dts -o board.dtb
```

- A DTB can be decompiled (to the standard output) with:

```
dtc -I dtb board.dtb
```

Device Tree Source file: example

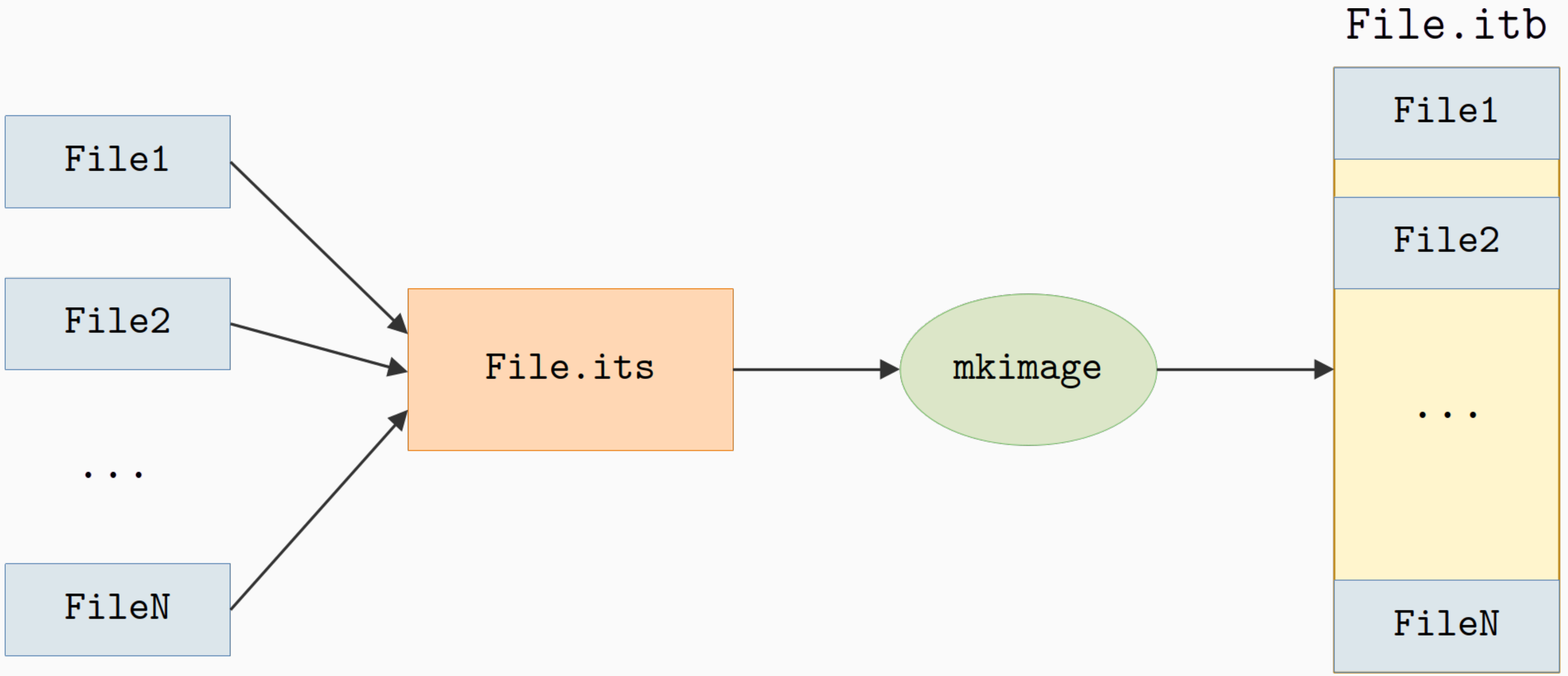
output/build/linux-6.3.6/arch/arm64/boot/dts/allwinner/sun50i-h5-nanopi-neo-plus2.dts:

```
/dts-v1/;
#include "sun50i-h5.dtsi"

#include <dt-bindings/gpio/gpio.h>
#include <dt-bindings/input/input.h>
#include <dt-bindings/pinctrl/sun4i-a10.h>

/ {
        model = "FriendlyARM NanoPi NEO Plus2";
        compatible = "friendlyarm,nanopi-neo-plus2", "allwinner,sun50i-h5";

        aliases {
                ethernet0 = &emac;
                serial0 = &uart0;
        };

        chosen {
                stdout-path = "serial0:115200n8";
        };

        leds {
                compatible = "gpio-leds";

                led-0 {
                        label = "nanopi:green:pwr";
                        gpios = <&r_pio 0 10 GPIO_ACTIVE_HIGH>;
```

Flattened Image Tree (FIT): what?

- After the introduction of FTD in Linux, the FIT (Flattened Image Tree) binary file format was created
- FIT allows to insert different files into a single file
- FIT is composed of two files:
  - `.its`: an **Image Tree Source** file (text)
    - describes which files to insert into the `.itb` file
  - `.itb`: an **Image Tree Blob** file (binary)

Building an Image Tree Blob (itb) from an Image Tree Source (its)



FIT: why and how?

- The FIT format allows more flexibility in handling images of various types
- Enhances integrity protection of images with hash functions, rsa signature, sha256, sha1, md5, crc32, etc.
- The `mkimage` command reads a `.its` file and creates a `.itb` file:

```
mkimage -f File.its -E File.itb
```

FIT source file: example

output/build/uboot-2020.10-rc5/u-boot.its:

```
/dts-v1/;
/ {
    description = "Configuration to load ATF before
U-Boot";
    images {
        uboot {
            description = "U-Boot (64-bit)";
            data = /incbin/("u-boot-nodtb.bin");
            type = "standalone";
            arch = "arm64";
            compression = "none";
            load = <0x4a000000>;
        };
        atf {
            description = "ARM Trusted Firmware";
            data = /incbin/("/workspace/buildroot/
output/images/bl31.bin");
            type = "firmware";
            arch = "arm64";
            compression = "none";
            load = <0x44000>;
            entry = <0x44000>;
        };
```

```
        fdt_1 {
            description = "sun50i-h5-nanopi-neo-
plus2";
            data = /incbin/("arch/arm/dts/sun50i-
h5-nanopi-neo-plus2.dtb");
            type = "flat_dt";
            compression = "none";
        };
    };
    configurations {
        default = "config_1";
        config_1 {
            description = "sun50i-h5-nanopi-neo-
plus2";
            firmware = "uboot";
            loadables = "atf";
            fdt = "fdt_1";
        };
    };
};
```

Listing the content of a FIT file: example

- The dumpimage[1] tool can be used to list the content of an `.itb` file:

```
$ dumpimage -l u-boot.itb
FIT description: Configuration to load ATF before U-Boot
Created:         Sun Sep  7 21:56:17 2025
 Image 0 (uboot)
  Description:  U-Boot (64-bit)
  Created:      Sun Sep  7 21:56:17 2025
  Type:         Standalone Program
  Compression:  uncompressed
  Data Size:    582200 Bytes = 568.55 KiB = 0.56 MiB
  Architecture: AArch64
  Load Address: 0x4a000000
  Entry Point:  unavailable
 Image 1 (atf)
  Description:  ARM Trusted Firmware
  Created:      Sun Sep  7 21:56:17 2025
  Type:         Firmware
```

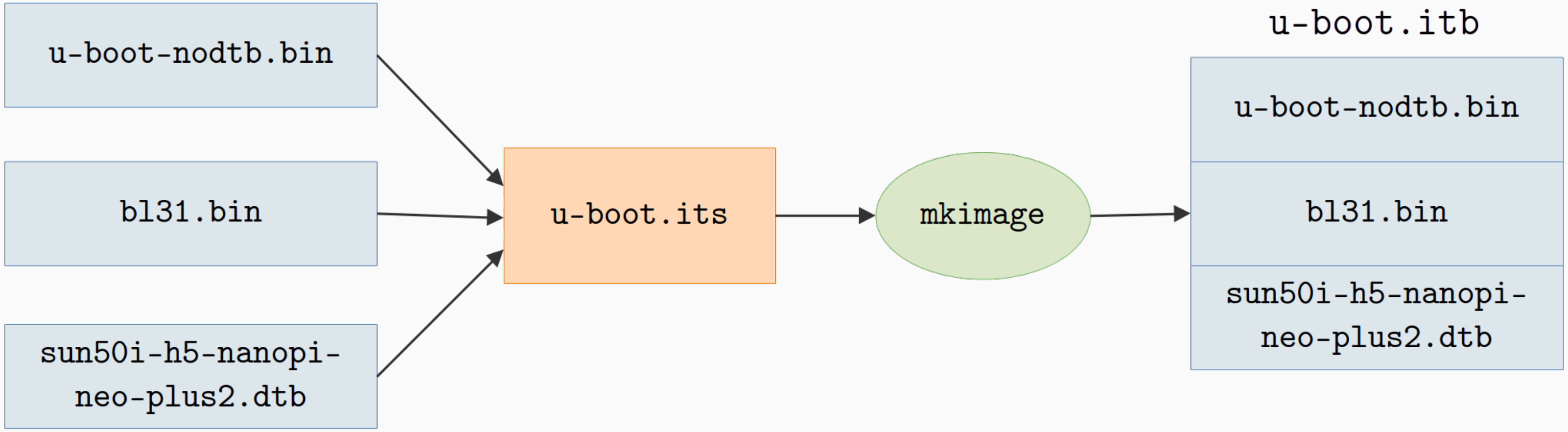
[1] Available in the `u-boot-tools` package on Debian/Ubuntu Linux

Listing the content of a FIT file: example

```
  Compression:  uncompressed
  Data Size:    20484 Bytes = 20.00 KiB = 0.02 MiB
  Architecture: AArch64
  OS:           Unknown OS
  Load Address: 0x00044000
 Image 2 (fdt_1)
  Description:  sun50i-h5-nanopi-neo-plus2
  Created:      Sun Sep  7 21:56:17 2025
  Type:         Flat Device Tree
  Compression:  uncompressed
  Data Size:    26119 Bytes = 25.51 KiB = 0.02 MiB
  Architecture: Unknown Architecture
 Default Configuration: 'config_1'
 Configuration 0 (config_1)
  Description:  sun50i-h5-nanopi-neo-plus2
  Kernel:       unavailable
  Firmware:     uboot
  FDT:          fdt_1
  Loadables:    atf
```

FIT file content: example

- Based on the previous example, `u-boot.itb` contains:
  - `u-boot-nodtb.bin` → U-Boot bootloader
  - `bl31.bin` → ARM Trusted Firmware (ATF) EL3 runtime firmware
  - `sun50i-h5-nanopi-neo-plus2.dtb` → Device Tree Blob



U-Boot and FIT files

- U-Boot's `iminfo` command displays information about a FIT image:

```
=> iminfo 0x50000000
## Checking Image at 50000000 ...
   FIT image found
   FIT description: Linux kernel and FDT blob
    Image 0 (my_kernel)
     Description:  Linux 6.3.6
     ...
     Hash value:   c07b4732d3194a16014c6ba6c6d3bc66ae458ea12c69cdbe11dadda872964ec5
   ...
```

- U-Boot's `bootm` command boots a Linux kernel stored in a FIT image by specifying the adress at which the FIT image was loaded:

```
bootm 0x50000000
```

FIT file and the Linux kernel (1/3)

How to create a FIT file for the Linux kernel and Flattened Device Tree?

- An entry defining where to find the kernel, its architecture and where to load it, must be specified, e.g.:

```
my_kernel {
    description = "Linux 6.3.6";
    data = /incbin/("./Image");
    type = "kernel";
    arch = "arm64";
    os = "linux";
    compression = "none";
    load = <0x40080000>;
    entry = <0x40080000>;
    hash-1 {
        algo = "sha1";
    }
};
```

FIT file and the Linux kernel (2/3)

- An entry defining where to find the Flattened Device Tree Blob and where to load it, must be specified, e.g.:

```
my_fdt {
    description = "Flattened Device Tree Blob";
    data = /incbin/("./sun50i-h5-nanopi-neo-plus2.dtb");
    type = "flat_dt";
    arch = "arm64";
    compression = "none";
    load = <0x4FA00000>;
    entry = <0x4FA00000>;
    hash-1 {
        algo = "sha1";
    }
};
```

FIT file and the Linux kernel (3/3)

- A configuration entry must specify the kernel and FDT to use:

```
configurations {
    default = "my_config";
    my_config {
        description = "Insert_here_your_description";
        kernel = "my_kernel_entry";
        fdt = "my_fdt_entry";
    };
};
```

- **Multiple configurations** can be present, each referencing different kernels and FDT!
- Having both kernel and FDT inside a FIT file allows to load them and boot them both at once

Booting a Linux kernel in a FIT file

- Like any other FIT file, the Image Tree Blob (.itb) is generated from the Image Tree Source (.its) using `mkimage`
- The .itb must be loaded in DRAM, for instance using `fatload` or `ext4load` commands
- U-Boot can then boot the kernel along its FDT using `bootm` with the address where the .itb was loaded, e.g.:

```
=> bootm 0x50000000
## Loading kernel from FIT Image at 50000000 ...
   Using 'my_config' configuration
   Trying 'my_kernel_entry' kernel subimage
     Description:  Linux 6.3.6
     Type:         Kernel Image
   ...
```

FIT file: handling multiple configurations

- As mentioned earlier, **multiple configurations** can be present in a FIT file, each referencing different kernels and FDT
- Offers lots of **flexibility**: distribute a single FIT image, usable in different ways, e.g. production/debug build, multiple FDT which can be used on multiple hardware platforms, etc.
- To boot a given configuration, postfix the address with the configuration's name, e.g. to boot `my_config_2`:

```
bootm 0x50000000#my_config_2
```

Checksums in FIT files

- `Image` and `zImage` are prone to silent data corruption, which can go **unnoticed**
- FIT format supports configurable checksum algorithms (SHA256, SHA1, CRC32, etc.) to protect any artifacts (kernel, FDT, etc.)
- U-Boot uses the checksums during boot to detect silent corruption
- To protect an artifact using checksum, add the following node to it (with the checksum algorithm of your choice):

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

SD card image generation

Buildroot configuration for Nano Pi NEO Plus2

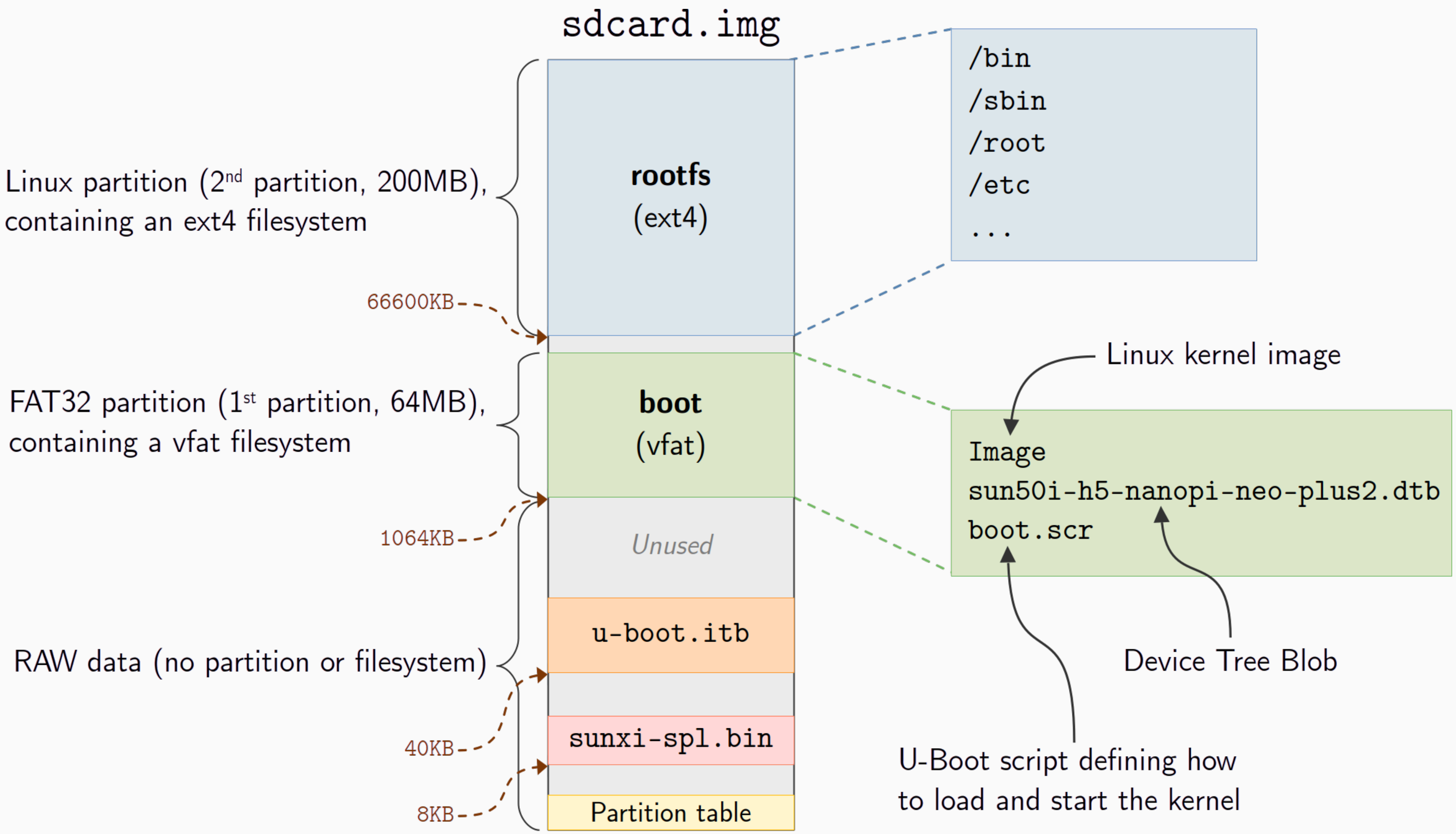
- Buildroot runs `mkimage` on `buildroot/board/friendlyarm/nanopi-neo-plus2/boot.cmd` to generate `boot.scr` which is executed by U-Boot
- Buildroot configured to execute `buildroot/support/scripts/genimage.sh` after filesystem images creation:
  - `genimage.sh` configured to read configuration in `buildroot/board/friendlyarm/nanopi-neo-plus2/genimage.cfg`
  - `genimage.cfg` describes how to create the SD card image
- Generated images files and firmwares (`sdcard.img`, `boot.vfat`, `rootfs.ext4`, `sunxi-spl.bin`, `bl31.bin`, `u-boot.itb`, `boot.scr`, `Image`) are located in `buildroot/output/images`

Nano Pi NEO Plus2: genimage.cfg

```
image boot.vfat {
    vfat {
        files = {
            "Image",
            "sun50i-h5-nanopi-neo-plus2.dtb",
            "boot.scr"
        }
    }

    size = 64M
}

image sdcard.img {
    partition spl {
        in-partition-table = "no"
        image = "sunxi-spl.bin"
        offset = 8K
    }
```

```
    partition u-boot {
        in-partition-table = "no"
        image = "u-boot.itb"
        offset = 40K
        size = 1M # 1MB - 40KB
    }

    partition boot {
        partition-type = 0xC
        bootable = "true"
        image = "boot.vfat"
    }

    partition rootfs {
        partition-type = 0x83
        image = "rootfs.ext4"
    }
}
```

Nano Pi NEO Plus2: SD card image (sdcard.img)



Nano Pi NEO Plus2 boot sequence

1. BootROM loads & executes `sunxi-spl.bin`
2. `sunxi-spl.bin`, the Secondary Program Loader (SPL):
   - Initializes DRAM
   - Loads `u-boot.itb` (contains `bl31.bin`, `u-boot-nodtb.bin`, `sun50i-h5…plus2.dtb`)
   - Executes `bl31.bin`
3. `bl31.bin`, the ARM Trusted Firmware (ATF) EL3 runtime firmware:
   - Sets up exception levels, power management, secure monitor services, etc.
   - Executes `u-boot-nodtb.bin`
4. `u-boot-nodtb.bin`, the U-Boot bootloader:
   - Performs additional hardware initialization
   - Loads and executes `boot.scr` which:
     - loads the Linux kernel `Image` and Device Tree Blob `sun50i-h5…plus2.dtb`
     - passes DTB and arguments to the Linux kernel, then boots it

U-Boot binary generation

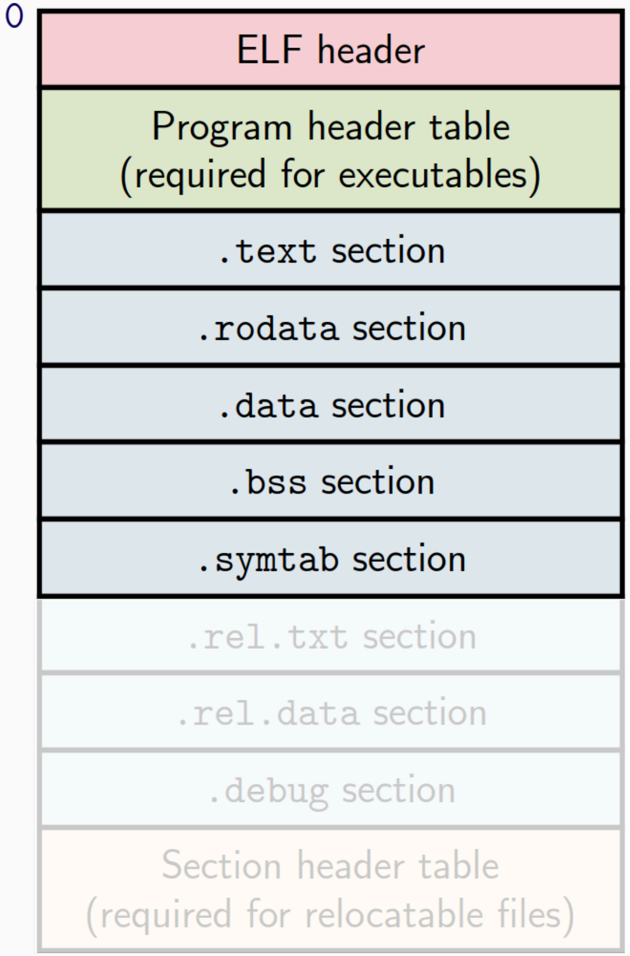
ELF file format (1/2)

ELF header

- file type (.o, .so, exec), machine type, word size, byte ordering, etc.

Program header table

- segments virtual addresses (sections), segment sizes
- .text section → code
- .rodata section → read only data
- .data section → initialized global variables
- .bss section → uninitialized global variables
  - occupies no space!
- .symtab section
  - symbol table
  - function and static variable names
  - section names and locations

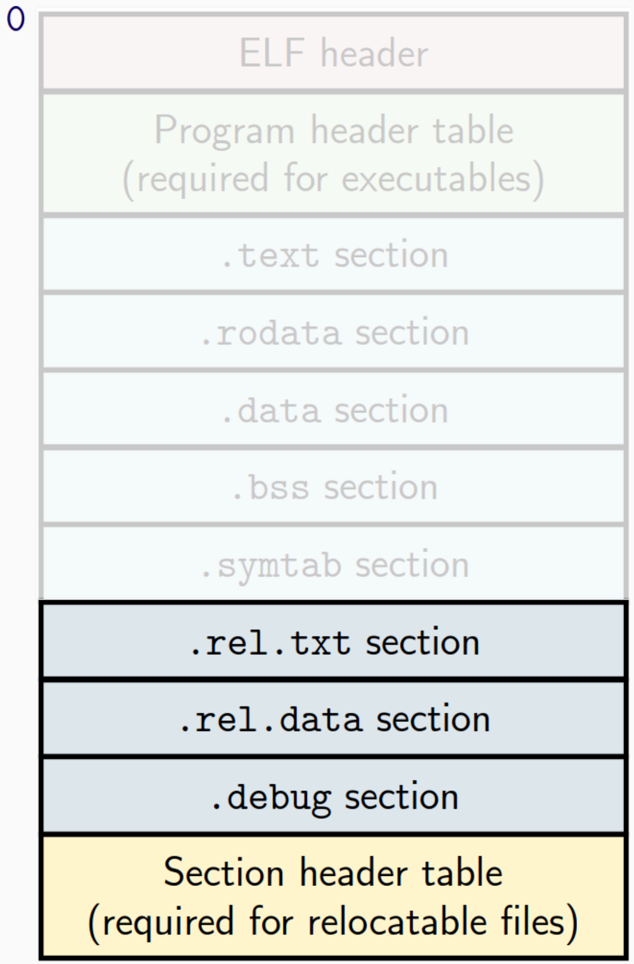


ELF file format (2/2)

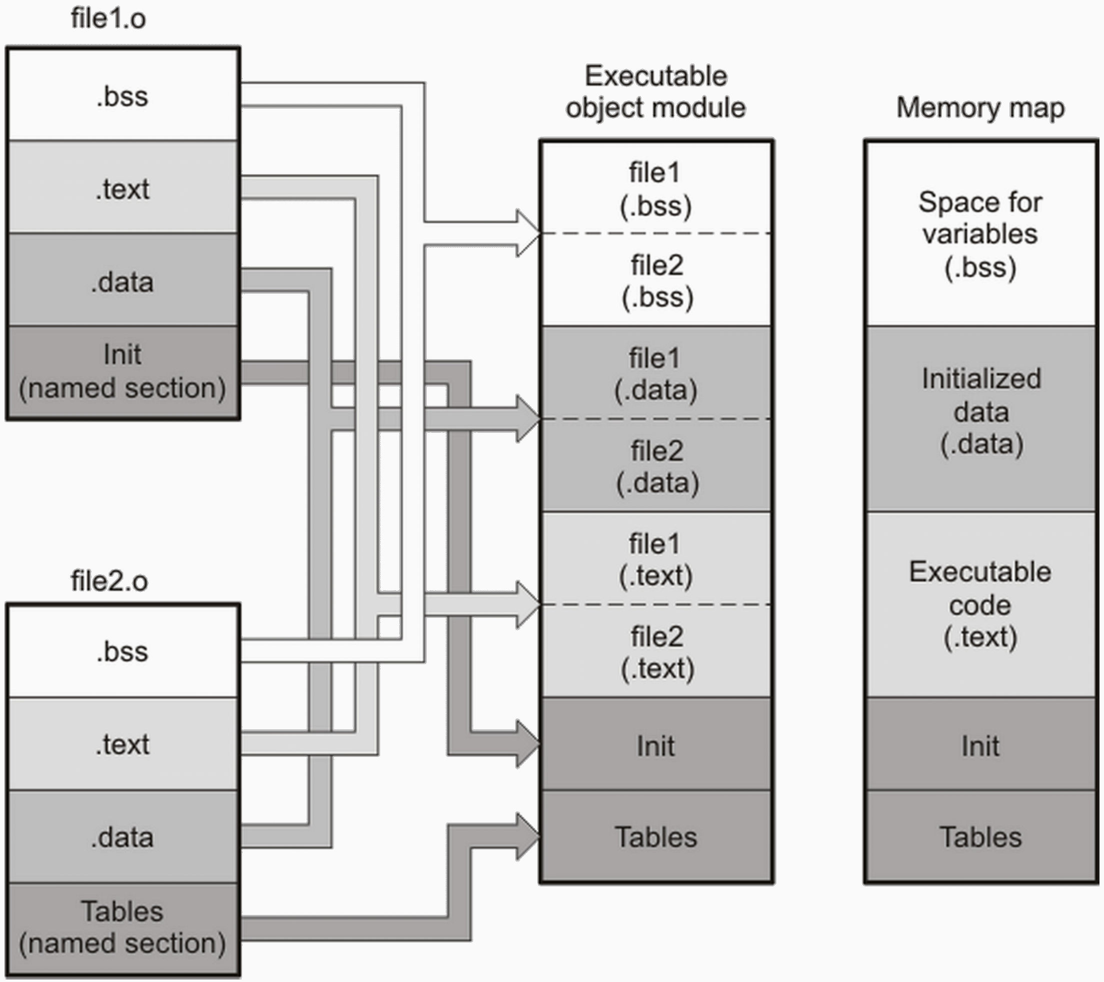
- `.rel.text` section
  - relocation info for `.text` section
  - addresses of instructions that'll need to be modified in the executable
- `.rel.data` section
  - relocation info for `.data` section
  - addresses of pointer data that will need to be modified in the merged executable
- `.debug` section
  - info for symbolic debugging (`gcc -g` option)

Section header table

- offsets and sizes of each section



ELF: from relocation to execution



U-Boot binaries

- The U-Boot build process creates several binaries: `u-boot`, `u-boot.bin`, `u-boot-dtb.bin`, `u-boot-nodtb.bin`, etc.
- The `u-boot` file (in `output/build/uboot-xx/`) is an ELF binary executable:
  - **structured** format: contains a header and various sections
  - allows a loader to know where and how sections must be loaded in RAM
  - `readelf` command can be used to inspect ELF files
- U-Boot binaries ending in `.bin` are unstructured **raw** binary executables (contain no header, nor sections)

Binaries: symbols & security considerations

- ELF executable files contain various symbols:
  - Debugging symbols (required for debugging)
  - Function and variable symbols (read by `objdump` and `nm`)
  - Dynamic linking symbols (required for linking to libraries)
- **Only dynamic linking symbols are required for execution**
- To **improve security** it's advisable to **remove** unecessary symbols from ELF files (debugging, functions and variables symbols)
- The `strip` utility is used to remove symbols from an ELF file, e.g.:

```
aarch64-linux-strip u-boot -o u-boot.stripped
```

U-Boot raw binaries

- `sunxi-spl.bin` the Secondary Program Loader **can only** read FIT files and **raw** binaries
  - It cannot parse and load ELF files such as `u-boot`
- Consequently, a **raw** binary must be created from `u-boot` ELF file:
  - `u-boot.bin` and `u-boot-nodtb.bin` are **raw** binaries
- How to generate a **raw** binary from an ELF file?

U-Boot raw binary generation

- The `objcopy` utility can generate **raw** binaries from ELF binaries:
  - Excerpt from `man objcopy`:

    objcopy *can be used to generate a raw binary file by using an output target of binary (e.g., use -O binary). When objcopy generates a raw binary file, it will essentially* ***produce a memory dump of the contents of the input object file****.* ***All symbols and relocation information will be discarded****. The memory dump will start at the load address of the lowest section copied into the output file.*

- Always use tools from the toolchain dedicated to the target system!
  - toolchain located in `buildroot/output/host/bin`

```
# Generate a raw binary from u-boot ELF binary
aarch64-linux-objcopy -O binary u-boot u-boot-nodtb.bin
```

Hardening U-Boot

Buffer overflow

```c
char buffer_overflow() {
    char tab[16];

    for (int i = 0; i < 26; i++)
        tab[i] = 42;

    return tab[0];
}

int main() {
    return buffer_overflow();
}
```

How to detect such buffer overflows?

Buffer overflow

```c
char buffer_overflow() {
    char tab[16];
    for (int i = 0; i < 26; i++)
        tab[i] = 42;
    return tab[0];
}

int main() {
    return buffer_overflow();
}
```

How to detect such buffer overflows?

- By telling the compiler to generate code for stack overflow checks
- Also called stack canary code

Checking for buffer overflows

- `gcc` can be configured to emit special code to check for buffer overflows
- The family of `-fstack-protector` options add a guard variable named **canary** to functions
  - Inserted on the stack at function entry and checked before returning to the caller

Option	Protection scope	Overhead
`-fstack-protector`	Only functions with local char arrays or `alloca`	low
`-fstack-protector-strong`	Functions with arrays (any type), references to local addresses, or dynamic sizes	medium
`-fstack-protector-all`	All functions	high

Buffer overflow detection example

```c
// overflow.c
char buffer_overflow() {
    char tab[16];
    for (int i = 0; i < 26; i++)
        tab[i] = 42;
    return tab[0];
}

int main() {
    return buffer_overflow();
}
```

```
$ gcc -Wall -fstack-protector-all overflow.c -o overflow && ./overflow
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

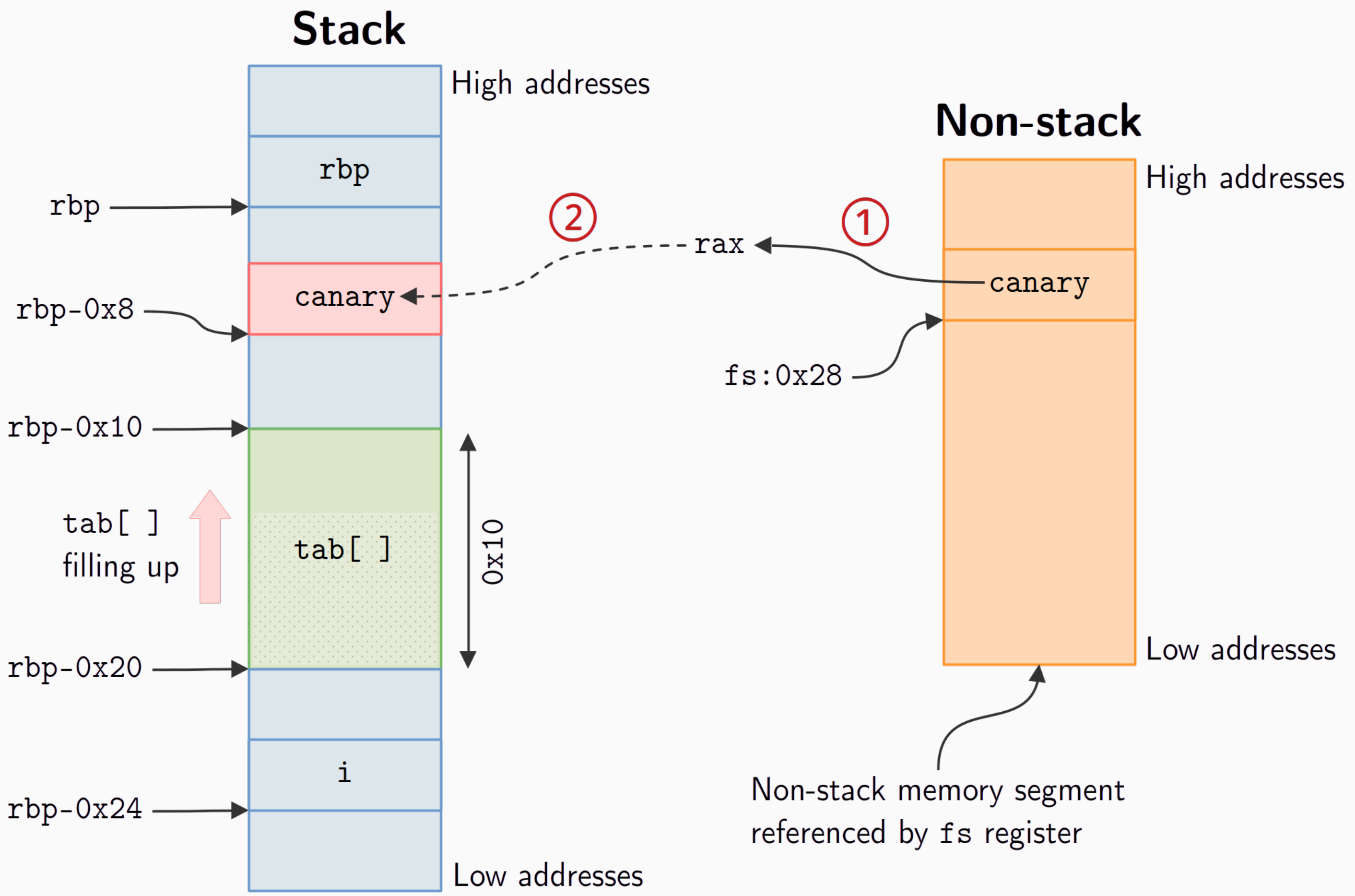
To disassemble the code:

```
objdump -M intel -d overflow
```

Generated canary code (Intel x86-64/AMD64/Intel64)

```
0000000000001149 <buffer_overflow>:
    1149: f3 0f 1e fa           endbr64
    114d: 55                    push   rbp
    114e: 48 89 e5              mov    rbp,rsp
    1151: 48 83 ec 30           sub    rsp,0x30
    1155: 64 48 8b 04 25 28 00  mov    rax,QWORD PTR fs:0x28          // reads canary value in rax
    115c: 00 00
    115e: 48 89 45 f8           mov    QWORD PTR [rbp-0x8],rax        // save canary value on the stack
    1162: 31 c0                 xor    eax,eax
    1164: c7 45 dc 00 00 00 00  mov    DWORD PTR [rbp-0x24],0x0       // loop counter (i = 0)
    116b: eb 0e                 jmp    117b <buffer_overflow+0x32>
    116d: 8b 45 dc              mov    eax,DWORD PTR [rbp-0x24]       // eax = i
    1170: 48 98                 cdqe                                  // rax = eax
    1172: c6 44 05 e0 2a        mov    BYTE PTR [rbp+rax*1-0x20],0x2a // tab[i] = 0x2a (42)
    1177: 83 45 dc 01           add    DWORD PTR [rbp-0x24],0x1       // i++
    117b: 83 7d dc 19           cmp    DWORD PTR [rbp-0x24],0x19      // i == 0x19 (25) ?
    117f: 7e ec                 jle    116d <buffer_overflow+0x24>    // jmp 116d if i <= 0x19
    1181: 0f b6 45 e0           movzx  eax,BYTE PTR [rbp-0x20]        // eax = tab[0]
    1185: 48 8b 55 f8           mov    rdx,QWORD PTR [rbp-0x8]        // rdx = saved canary value
    1189: 64 48 2b 14 25 28 00  sub    rdx,QWORD PTR fs:0x28          // subtract original canary value
    1190: 00 00
    1192: 74 05                 je     1199 <buffer_overflow+0x50>    // jmp 1199 if subtract == 0
    1194: e8 b7 fe ff ff        call   1050 <__stack_chk_fail@plt>    // otherwise -> stack_chk_fail
    1199: c9                    leave
    119a: c3                    ret
```

Stack smashing detection using a canary



Analyzing U-Boot compilation options

```
make uboot-rebuild V=1
```

```
/workspace/buildroot/output/host/bin/aarch64-buildroot-linux-gnu-gcc -Wp,-MD,spl/arch/arm/cpu/armv8/.fwcall.o.d -nostdinc -isystem /workspace/buildroot/output/host/lib/gcc/aarch64-buildroot-linux-gnu/11.3.0/include -Ispl/include -Iinclude -I./arch/arm/include -include ./include/linux/kconfig.h -DKERNEL -DUBOOT -DCONFIG_SPL_BUILD -Wall -Wstrict-prototypes -Wno-format-security -fno-builtin -ffreestanding -std=gnu11 -fshort-wchar -fno-strict-aliasing -fno-PIE -Os -fno-stack-protector -fno-delete-null-pointer-checks -Wno-stringop-truncation -Wno-maybe-uninitialized -fmacro-prefix-map=./= -g -fstack-usage -Wno-format-nonliteral -Wno-address-of-packed-member -Wno-unused-but-set-variable -Werror=date-time -ffunction-sections -fdata-sections -DARM -mstrict-align -ffunction-sections -fdata-sections -fno-common -ffixed-r9 -fno-common -ffixed-x18 -pipe -march=armv8-a -DLINUX_ARM_ARCH=8 -I./arch/arm/mach-sunxi/include -DKBUILD_BASENAME=‘‘“fwcall”’’ -DKBUILD_MODNAME=‘‘“fwcall”’’ -c -o spl/arch/arm/cpu/armv8/fwcall.o arch/arm/cpu/armv8/fwcall.c
```

- `-fno-stack-protector` → **does not** emit extra code to check for buffer overflows, such as stack smashing attacks

Changing U-Boot compilation options

- Buildroot features a “Stack Smashing Protection” option in its configuration
- Unfortunately: only for userspace packages (running on top of the Linux kernel) → doesn't apply to U-Boot which uses its own `Kconfig` and `CFLAGS`
- Instead, U-Boot build **must be modified *by hand*** to emit code for stack overflow checks:
  - Adding compilation options to generate stack protection checks is not enough: the linker will complain about `__stack_chk_guard` and `__stack_chk_fail` symbols not being defined
  - This is because U-Boot is compiled with baremetal options (i.e. no OS support), thus the required supporting code is missing

Adding stack protection checks to U-Boot

- Modify U-Boot's root `Makefile` to add stack protection checks
  - Done by adding the option to the `KBUILD_CFLAGS` variable
- Add a new source file `common/stackprot.c` which must contain the implementation of the `__stack_chk_fail` function and the `__stack_chk_guard` global variable[1]
- Modify U-Boot `common/Makefile` so that `stackprot.o` is added to the list of object files to be linked
  - Done by adding the object file to the `obj-y` variable

[1] See https://github.com/u-boot/u-boot/tree/u-boot-2023.07.y/common

Resources

- U-Boot documentation: https://docs.u-boot.org/en/latest/
- https://wiki.friendlyelec.com/wiki/index.php/NanoPi_NEO_Plus2
- https://www.thegoodpenguin.co.uk/blog/u-boot-fit-image-overview/
- Examples of FIT config files in: `buildroot/output/build/uboot-xx/doc/uImage.FIT/`
- Secure and flexible boot with U-Boot bootloader: https://elinux.org/images/8/8a/Vasut--secure_and_flexible_boot_with_u-boot_bootloader.pdf