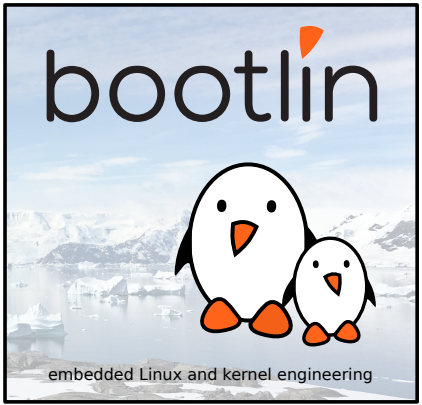
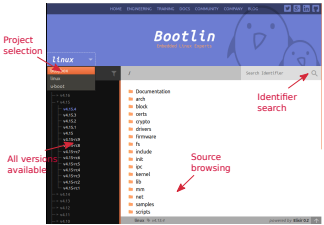
Understanding U-Boot Falcon Mode

Michael Opdenacker
michael.opdenacker@bootlin.com



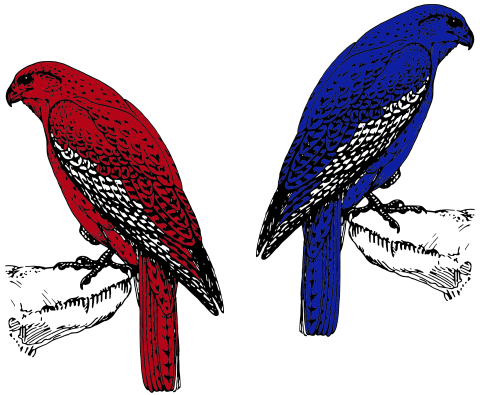
Michael Opdenacker

- Founder and Embedded Linux engineer at Bootlin:
  - Embedded Linux **expertise**
  - **Development**, consulting and training
  - Focusing **only on Free and Open Source Software**
- Free Software contributor:
  - Current maintainer of the Elixir Cross Referencer, making it easier to study the sources of big C projects like the Linux kernel. See https://elixir.bootlin.com
  - Co-author of Bootlin's freely available embedded Linux and kernel training materials (https://bootlin.com/docs/)
  - Former maintainer of GNU Typist



Goal: boot faster!

U-Boot Falcon Mode is about reducing the time spent in the bootloader.



Falcons are the fastest animals on Earth!

Image credits: https://openclipart.org/detail/287044/falcon-2

Example: booting on Microchip SAMA5D36

You first need to understand how your SoC boots:

Figure 11-2. NVM Bootloader Sequence Diagram

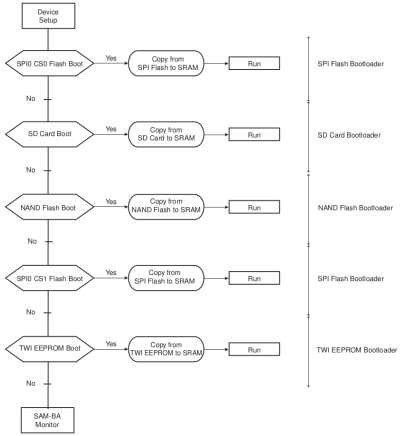
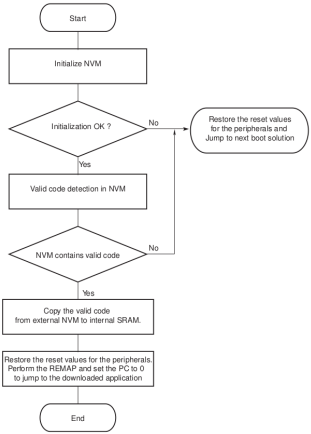
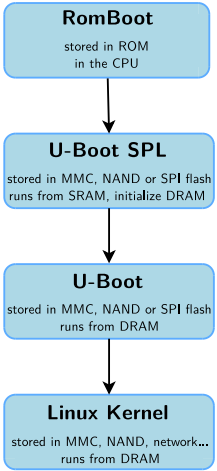


Figure 11-3. NVM Bootloader Program Diagram



Source: Microchip SAMA5D36 datasheet
https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11121-32-bit-Cortex-A5-Microcontroller-SAMA5D3_Datasheet_B.pdf

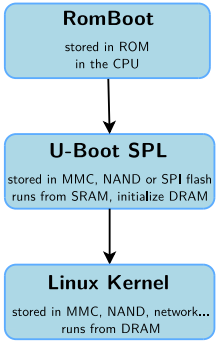
Normal and Falcon boot on Microchip SAMA5D3



Boot process with U-Boot

- **RomBoot**: tries to find a valid bootstrap image from various storage sources, and load it into SRAM (DRAM not initialized yet). Size limited to the SRAM size (here 64 KB).
- **U-Boot SPL** (*Secondary Program Loader*): runs from SRAM (inside the SoC). Initializes the DRAM controller plus storage devices (MMC, NAND), loads the secondary bootloader into DRAM and starts it. Much bigger size limits!
- **U-Boot**: runs from DRAM. Initializes other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to DRAM and starts it.
  This is the part that can be skipped
- **Linux Kernel**: runs from DRAM. Takes over the system completely (the bootloader no longer exists).

This scheme applies to all modern SoCs.



Boot process without U-Boot (*Falcon mode*)

Falcon mode advantages and drawbacks

- Main advantage: since Linux and U-Boot are both loaded to RAM, U-Boot's *Falcon Mode* mainly saves time by directly loading Linux from the SPL instead of loading and executing the full U-Boot first.
- Drawback: you lose the flexibility brought by the full U-Boot. You have to follow a special procedure to update the kernel binary, DTB and kernel command line parameters.
- Advantage: the interactivity offered by the full U-Boot is not necessary on a production device. Falcon boot works in the same way on all SoCs on which U-Boot SPL is supported. This makes it easier to apply this technique on all your projects!

What U-Boot does (1)

U-Boot has multiple ways of preparing the kernel boot:

- *ATAGS* - The old way (before Device Tree)
  U-Boot prepares the Linux kernel command line (`bootargs`), the machine ID and other information for Linux in a tagged list, and passes its address to Linux through a register.
- *Flattened Device Tree* - The standard way
  - U-Boot checks the device tree loaded in RAM or directly provides its own.
  - U-Boot checks the specifics of the hardware (amount and location of RAM, MAC address, present devices...), possibly loads corresponding Device Tree overlays, and modifies (fixes-up) the Device Tree accordingly.
  - U-Boot stores the Linux kernel command line (`bootargs`) in the `chosen` section in the Device Tree.

What U-Boot does (2)

- *FIT Image* - The new way
  - U-Boot loads the kernel(s), device tree(s), initramfs image(s), signature(s) from a single file (*FIT Image*)
  - That's used for secure booting, for booting recovery images, etc.
  - U-Boot also implements Device Tree fix-ups, of course.

Using the `spl export` command in U-Boot, you can do such preparation work ahead of time.

- In this presentation, we just cover standard Device Tree booting.
- U-Boot also has support for FIT Image loading in the SPL, but that may still be a bit experimental, and such code must fit within your maximum allowable size for the SPL.
  See `arch/arm/cpu/armv8/fsl-layerscape/doc/README.falcon`

Falcon mode usage overview (1)

Here are the generic steps you need to go through:

- Recompile U-Boot with support for Falcon Mode (`CONFIG_SPL_OS_BOOT`), with support for `spl export` (`CONFIG_CMD_SPL`), and for the way you want to boot.
- Also make sure that `CONFIG_SPL_SIZE_LIMIT` is set (find the SRAM size for your CPU, `0x10000` for SAMA5D36), otherwise, U-Boot won't complain when the SPL is bigger.
- Build the kernel legacy `uImage` file from `zImage` (see next slides)
- Set the kernel command line (`bootargs` environment variable)
- Load the `uImage`, initramfs (if any) and Device Tree images to RAM as usual.

Falcon mode usage overview (2)

Continued...

- Have U-Boot execute the preprocessing before booting Linux, but stop right before doing it:

  ```
  spl export fdt <kernel-addr> <initramfs-addr> <dtb-addr>
  ```

- Save the exported data (*ARGS*) from RAM to storage, in *Flattened Device Tree* form, so that the SPL can load it and directly pass it to the Linux kernel. The below environment variables will help:
  - `fdtargsaddr`: location of *ARGS* in RAM
  - `fdtargslen`: size of *ARGS* in RAM
- If supported by your board (code explanations given later), set your `boot_os` environment variable to `yes/Yes/true/True/1` to enable direct OS booting.

spl export example output

```
=> fatload mmc 0:1 0x21000000 uImage
5483584 bytes read in 530 ms (9.9 MiB/s)
=> fatload mmc 0:1 0x22000000 dtb
27795 bytes read in 7 ms (3.8 MiB/s)
=> setenv bootargs console=ttyS0,115200
=> spl export fdt 0x21000000 - 0x22000000
## Booting kernel from Legacy Image at 21000000 ...
   Image Name:   Linux-5.12.6
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    5483520 Bytes = 5.2 MiB
   Load Address: 20008000
   Entry Point:  20008000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 22000000
   Booting using the fdt blob at 0x22000000
   Loading Kernel Image
   Loading Device Tree to 2fb2c000, end 2fb35c92 ... OK
subcommand not supported
subcommand not supported
   Loading Device Tree to 2fb1f000, end 2fb2bc92 ... OK
Argument image is now in RAM: 0x2fb1f000
```

Image credits:
https://openclipart.org/detail/292953/horus

How to create the uImage file

Microchip SAMA5D3 Xplained board example

- Need to know the loading address that should be used for your board. Usually on ARM32, it's the starting physical address of RAM plus `0x8000`.
- Either generate it from the Linux build system:

```
make LOADADDR=0x20008000 uImage
```

- Or generate it using U-Boot's `mkimage` command:

```
mkimage -A arm -O linux -C none  -T kernel \
-a 0x20008000 -e 0x20008000 \
-n "Linux-5.12.6" \
-d arch/arm/boot/zImage arch/arm/boot/uImage
```

U-Boot code changes to support a new board (1)

Your `board/<vendor>/<board>/<board>.c` file must at least implement the `spl_start_uboot()` function.
Here's the most typical example:

```
#ifdef CONFIG_SPL_OS_BOOT
int spl_start_uboot(void)
{
	if (CONFIG_IS_ENABLED(SPL_SERIAL_SUPPORT) && serial_tstc() && serial_getc() == 'c')
		/* break into full u-boot on 'c' */
		return 1;

	if (CONFIG_IS_ENABLED(SPL_ENV_SUPPORT)) {
		env_init();
		env_load();
		if (env_get_yesno("boot_os") != 1)
			return 1;
	}
	return 0;
}
#endif
```

U-Boot code changes to support a new board (2)

If you cannot fit support for an environment in the SPL, the `spl_start_uboot()` function can be simpler:

```
#ifdef CONFIG_SPL_OS_BOOT
int spl_start_uboot(void)
{
	if (CONFIG_IS_ENABLED(SPL_SERIAL_SUPPORT) && serial_tstc() && serial_getc() == 'c')
		/* break into full u-boot on 'c' */
		return 1;

	return 0;
}
#endif
```

U-Boot code changes to support a new board (3)

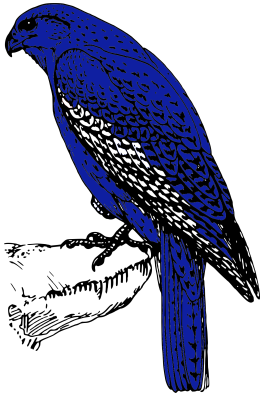
Or even, if reading characters from the serial line doesn't work:

```
#ifdef CONFIG_SPL_OS_BOOT
int spl_start_uboot(void)
{
        return 0;
}
#endif
```

You may also need extra defines to be set, but you will find which ones are missing at compile time.

How to fall back to U-Boot

- If supported by your board, hit the specified key on the serial console and back in U-Boot, disable the `boot_os` environment variable. That's it.
- Otherwise, try to cause OS loading to fail. The easiest way is to erase the kernel binary and if needed the `spl export` output.
- If this doesn't work, re-compile and update the SPL without Falcon mode support, or temporarily modify the `spl_start_uboot()` function to always return `1`. This way, you don't lose your configuration.



Booting from raw MMC - Proposed storage layout

For use on Microchip SAMA5D3 Xplained

Offset (512 b sector)	Offset (bytes)	Contents
0x0	0	MBR (Master Boot Record)
0x100	128 KiB	SPL ARGS
0x200	256 KiB	`u-boot.img`
0x1000	2 MiB	uImage
0x4000	16 MiB	Start of FAT partition

- A FAT partition is required to store the SPL file (`boot.bin`). SAMA5D36 doesn't support an SPL file on raw MMC (unlike i.MX6).
- Caution: partition offsets should be a multiple of the *segment* size, as indicated by the device's `preferred_erase_size` attribute under `/sys/bus/mmc/devices/`.

Booting from raw MMC - Configuration

U-Boot configuration (starting from `sama5d3_xplained_mmc_defconfig`):

```
CONFIG_SPL_OS_BOOT=y
SPL_SIZE_LIMIT=0x10000
CONFIG_SPL_LEGACY_IMAGE_SUPPORT=y
CONFIG_SPL_MMC_SUPPORT=y
CONFIG_CMD_SPL=y
CONFIG_CMD_SPL_WRITE_SIZE=0x7000
CONFIG_SYS_MMCSD_RAW_MODE_U_BOOT_SECTOR=0x200
# CONFIG_SPL_FS_FAT is not set
```

include/configs/sama5d3_xplained.h

```
#define CONFIG_SYS_MMCSD_RAW_MODE_ARGS_SECTOR 0x100  /* 256 KiB */
#define CONFIG_SYS_MMCSD_RAW_MODE_ARGS_SECTORS (CONFIG_CMD_SPL_WRITE_SIZE / 512)
#define CONFIG_SYS_MMCSD_RAW_MODE_KERNEL_SECTOR 0x1000 /* 2 MiB */
#define CONFIG_SYS_SPL_ARGS_ADDR 0x22000000
```

Booting from Raw MMC - Writing to raw storage

On your GNU/Linux host:

- Write U-Boot (using the same block size as sector size, to get the same offsets):
  `sudo dd if=u-boot.img of=/dev/sdc bs=512\ seek=512 conv=sync`
- Write `uImage`:
  `sudo dd if=uImage of=/dev/sdc bs=512 \ seek=4096 conv=sync`
- Reminder: in our case (SAMA5D36), the SPL is copied to `boot.bin` in a FAT partition.

On your U-Boot target, after `spl export`:

- Select the right MMC device for `mmc write`:

```
=> mmc list
Atmel mci: 0 (SD)
Atmel mci: 1

=> mmc dev 0
switch to partitions #0, OK
mmc0 is current device
```

- Check the size of ARGS

```
=> printenv fdtargslen
```

- Divide it by the sector size (512), and convert it to hexadecimal (round it up), and use the value to save the ARGS to raw MMC:

```
=> mmc write ${fdtargsaddr} 0x100 0x67
```

- **Caution**: the last argument of `mmc write` is a **number of sectors**. If you pass a number of bytes, you'll erase your FAT partition!

Booting from Raw MMC - Results and notes

Reference test

- Loading `zImage` and `dtb` from FAT through `fatload` and using a zero `bootdelay`:

```
setenv bootdelay 0
setenv bootcmd 'fatload mmc 0:
1 0x21000000 zImage; fatload mmc 0:
1 0x22000000; bootz 0x21000000 -
0x22000000'
```

- Not completely fair because we have the filesystem overhead, but that's the standard / easiest way on MMC. We could have loaded images from raw MMC, but that's very inconvenient.
- Best result (using `grabserial`):

```
[3.452681 0.000099] Please press Enter to
activate this console.
```

Falcon boot test

- Best result:

```
[3.191228 0.000134] Please press Enter to
activate this console.
```

- We saved 261 ms, but that's disappointing.
- Adding instrumentation to the SPL allowed us to understand why:
  - Time to load the kernel from U-Boot / FAT: 530 ms
  - Time to load the kernel from SPL / raw MMC: 1.010 ms
- Here the specific MMC driver in SPL has poor performance (lack of DMA?)
- We had much better results on different hardware, such as saving 1.2s on i.MX6, and 478 ms on TI AM3359 (Beagle Bone Black).

Booting from raw NAND - Configuration

Proposed NAND layout
For use on Microchip SAMA5D3 Xplained

Offset	Size	Contents
0x0	256 KiB	SPL (`spl/u-boot-spl.bin`)
0x40000	1 MiB	U-Boot (`u-boot.bin`)
0x140000	128 KiB	U-Boot redundant environment
0x160000	128 KiB	U-Boot environment
0x180000	128 KiB	Original DTB or CMD
0x1a0000	6.375 MiB	uImage
0x800000		Other partitions

Notes:

- Only the SPL offset is hardcoded
- All others can be configured differently
- Offsets must be a multiple of the erase block size (128 KiB)

U-Boot configuration

```
CONFIG_SPL_OS_BOOT=y
SPL_SIZE_LIMIT=0x10000
CONFIG_ENV_OFFSET=0x160000
CONFIG_ENV_OFFSET_REDUND=0x140000
CONFIG_SPL_LEGACY_IMAGE_SUPPORT=y
CONFIG_SPL_NAND_SUPPORT=y
CONFIG_SPL_NAND_DRIVERS=y
CONFIG_SPL_NAND_BASE=y
CONFIG_CMD_SPL_WRITE_SIZE=0x7000
CONFIG_CMD_SPL_NAND_OFS=0x180000
```

(starting from `sama5d3_xplained_nandflash_defconfig`)

include/configs/sama5d3_xplained.h

```
/* Generic settings */
#define CONFIG_SYS_NAND_U_BOOT_OFFS     0x40000

/* Falcon boot support on raw NAND */
#define CONFIG_SYS_NAND_SPL_KERNEL_OFFS 0x1a0000
```

Booting from raw NAND - Results and notes

- Reference test
  - To be fair, using a zero `bootdelay` and the exact `zImage` and `dtb` size:
    ```
    setenv bootdelay 0
    setenv bootcmd 'nand read 0x21000000 0x1a0000 0x53ac00; nand read 0x22000000 0x180000 0x6c93; bootz 0x21000000 - 0x22000000'
    ```
  - Best result (using `grabserial`):
    ```
    [4.320618 0.000470] Please press Enter to activate this console.
    ```
- Falcon boot test
  - Best result (using `grabserial`):
    ```
    [3.768543 0.000125] Please press Enter to activate this console.
    ```
  - We saved 552 ms!

U-Boot code and debugging Falcon Mode

- Depending on how you boot, read the corresponding code:
  - `common/spl/spl_mmc.c`
  - `common/spl/spl_nand.c`
  - Other files in `common/spl/`
- If booting doesn't work, the easiest way is to add `puts();` lines to trace strategic functions and check return values. You'll get the messages in the serial console.

Issues and lessons learned (1)

- *SPL storage driver performance*: not on all platforms, but at least here on Microchip SAMA5.
- *Features limited by space*: what can be done with Falcon booting is not limited by U-Boot features, but by how much code can fit in the limited SRAM.
  This is why I couldn't show Falcon booting from a FAT partition, because adding support for this filesystem and disk partitions to the SPL doesn't fit in the maximum size possible on the particular platform chosen for the demo.
- *U-Boot initializations*: in addition to the FDT fixups without which the Linux kernel may not boot, the Linux kernel may also rely on some initializations performed by U-Boot. Before such dependencies can be removed by updating kernel drivers, you may need to hardcode such initializations in the SPL, provided you have enough space!

Issues and lessons learned (2)

- *Limited automation*: while the `uImage` file can be updated automatically in the storage image, any change in the kernel command line or Device Tree must go through the `spl export` command **on the board**. The FDT fixups done by U-Boot are not trivial to reproduce. This makes it difficult to prepare production images without a manual step in U-Boot.
- *No decompression*: U-Boot currently doesn't seem to support decompression in the SPL. If your architecture doesn't support kernel self-decompression and relies on the bootloader (e.g. arm64 or riscv), Falcon mode won't be available.
- *Side note*: Found that U-Boot's `bootm` was noticeably slower than `bootz` (+170 ms)

Further work

- Improve raw MMC read performance in the SPL on Microchip SAMA5
- Didn't try with what U-Boot calls the *Raw* kernel images yet, supported with `CONFIG_SPL_RAW_IMAGE_SUPPORT`. Assuming this corresponds to the `arch/arm/boot/Image`
- Didn't try FIT Image support in SPL yet. Will try on an SoC with more space for SPL (i.MX)

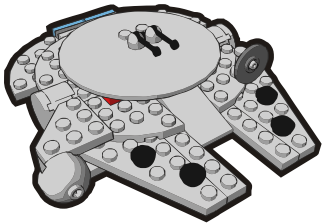


Image credits:
https://openclipart.org/detail/224913/clip-is-a-brick-star-wars-millenium-falcon-set-4488

References

- Bootlin's U-Boot patch to support Falcon boot on SAMA5D3 Xplained: https://lists.denx.de/pipermail/u-boot/2021-May/451107.html
- Bootlin's *Embedded Linux Boot Time Optimization* course with free training materials: https://bootlin.com/training/boot-time/
- U-Boot's doc/README.falcon file
- Linus Walleij: *How the ARM32 kernel decompresses*: https://people.kernel.org/linusw/how-the-arm32-linux-kernel-decompresses

This Is How You **Win** the Time War

Questions?
Suggestions?
Comments?

Michael Opdenacker
michael.opdenacker@bootlin.com

https://bootlin.com/pub/conferences/2021/lee/