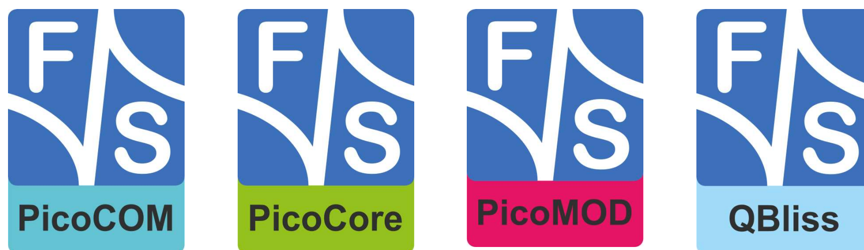
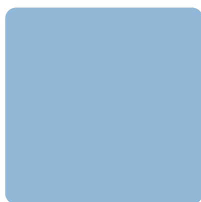
Linux on F&S Boards

Introduction

Version 0.24
(2026.05.27)





© F&S Elektronik Systeme GmbH
Untere Waldplätze 23
D-70569 Stuttgart
Germany

Phone: +49(0)711-123722-0
Fax: +49(0)711-123722-99

About This Document

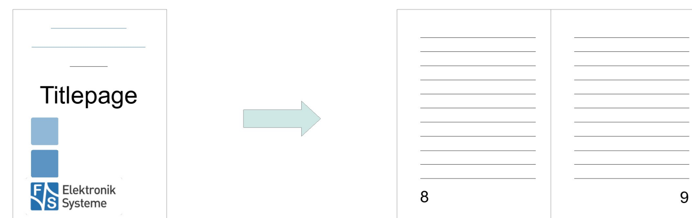
This document shows how to use the bootloaders, Linux system and peripherals on F&S boards and modules and how to update the system software. It also covers how to build bootloader, Linux kernel and root filesystem as well as own applications for the device.

Remark

The version number on the title page of this document is the version of the document. It is not related to the version number of any software release! The latest version of this document can always be found at http://www.fs-net.de.

How To Print This Document

This document is designed to be printed double-sided (front and back) on A4 paper. If you want to read it with a PDF reader program, you should use a two-page layout where the title page is an extra single page. The settings are correct if the page numbers are at the outside of the pages, even pages on the left and odd pages on the right side. If it is reversed, then the title page is handled wrongly and is part of the first double-page instead of a single page.



Typographical Conventions

We use different fonts and highlighting to emphasize the context of special terms:

`File names`

Menu entries

```
Board input/output
```

`Program code`

```
PC input/output
```

`Listings`

```
Generic input/output
```

`Variables`

History

Date	V	Platform	A,M,R	Chapter	Description	Au
2019-08-16	0.1	*	A	*	Initial Version. Large parts are moved from the FirstSteps document over here, other parts are newly written	HK
2019-11-15	0.2		M	3.6, 10	Use <arch> instead of fsimx6 in file names	HK
				*	Minor fixes all across the text	HK
2019-12-12	0.3	fsimx7ulp	A	3.7, 4	Add notes for PicoCoreMX7ULP	PJ
			A	5.7.3	Add chapter RNDIS	PJ
			A	5.10.3	Add commands to copy files from USB stick to local eMMC	PJ
2019-12-18	0.3	*	A	2.2	Add chapter “What is Embedded Linux?”	HK
			A	3.8	Add chapter about mass storage devices in general	HK
			M	5.4	Rework chapter for MTD partitions	HK
			A	5.10	Add graphics to show how downloading works in each case	HK
2020-03-24	0.4	*	A	8.18	Add distinguish between Yocto and Buildroot	PG
			M	10.2	Adjust documentation for new Yocto version 2.4	PG
2020-04-22	0.5	*	A	3.1	Add comment about license of the VirtualBox Extension Pack	HK
			M	3.6	Update release versions	HK
			R	4.2	Remove duplicate bad block description (already in 3.8.1)	HK
			M, A	6	Add detailed descriptions for install/update/recover and how to switch eMMC to Enhanced Mode; reorder subchapters	HK
			A	10	Add hint to build own configurations and not simply use ours	HK
2020-08-19	0.6	fsimx6ul	M	3.8.2	Add documentation for NBoot eMMC support and the original note for eMMC only devices with NBoot verions 43 or higher	KM
			M	4.6	Add necessity of MBR and FAT partition for U-Boot	KM
			A	4.8	Add chapter “Create Partitions”	KM
			A	4.9	Add chapter “Access Partitions with USB”	KM
			M	5.6	Add commands to erase environment in eMMC	KM
			M	5.9.3	Add documentation on fatwrite and ums command	KM
			A	5.10.2	Add chapter “TFTP download, Store in eMMC Flash”	KM
			A	5.10.5	Add chapter “Copy individual filesystem images over UMS to local eMMC”	KM
			M	10.1.4	Add defconfig for eMMC only devices	KM
			A	11	Add chapter “eMMc memory layout”	KM
2021-04-26	0.7	fsimx8mm	M	1.1	Add fsimx8mm architecture	PJ
			M	3.6	Remove minimal images, Add sdcard images, add distro flag	PJ
			M	4.0	Add PicoCoreMX8MM which has no NBoot	PJ
			M, A	3.6	Change distro description and add update file names	HK
			M	3.8.2	Use eMMC-Layout with NBoot, PicoCoreMX7ULP is special	HK
			D	5.6	Remove erasing of UBootEnv	HK
			A	5.6.1	Add general env command	HK
			A	5.6.2	Add mmcdev to list of special variables	HK
			A	5.6.3	Add env default -a to delete/restore environment	HK
			A	5.9.4	Move writing to FAT partition to extra sub-chapter	HK
			A	5.10	Add extra chapter for ums	HK
			M	5.11.2	Fix graphic	HK
			A	6.8	Add chapter for command fsimage	HK
2021-06-28	0.8	fsimx8mm	A	3.8.2	Add information about sysimage	PG
2021-08-05	0.9	fsimx6ul	M	1.1	Add PicoCOMA7 and PicoCoreMX6UL100	PJ
			M	3.6	Add correct difference between NBoot I.MX6 and I.MX8	PJ
			M	7.8	Add fsimx6ul for fsimage command	PJ
2021-10-18	0.10	all	A	6.4.7	Add information about building install.scr with Buildroot	PG
			A	9,9.2	Add information about Weston/Wayland	PG
			A	9.1.1	Add information about Weston/Wayland	HK
			A	10.1.4,	Add Information about building U-Boot with Buildroot	PG
			A	10.1.6		
			M	4.6,4.7,4.8, 4,9,5.11.2/4 /5	Fix wrong display of sysimage entry	PG
2021-11-08	0.11	fsimx8mn	M	3.8.2	Add UBoot offset for fsimx8mn in sysimg	KM
			M	5.11.2	Add note about writing a bigger root filesystem	KM
			M	10.2.4	Exclude all fsimx8 architectures from fbdev	KM
			M	10.2.5	Add a mention of Wayland/Weston to fus-image-std	KM

2022-01-31	0.12	fsimx6	M	1.1	Add efusA9r2	PJ
2022-05-18	0.13	Fsimx7ulp all	M M	5.11.4 6.4.7	Add comment about new eMMC layout Add information about Yocrto install script	PG
2022-09-09	0.14	fsimx8mm	A	8.20	Add instructions to use the RDP for Buildroot	DA
2022-09-13	0.15	fsimx8mm	A,M	8.20	Add instructions to use the RDP for Yocto Minor changes concerning the wording	DA
2022-10-18	0.16	ALL	M	ALL	Reworked the entire document, fixed figure bugs and converted it to the .docx format	DA
2022-11-25	0.17	fsimx8mp	A	1.1	Add table entry for picocoremx8mp(r2), armstonemx8mp	AD
2023-09-05	0.18	ALL	M	ALL	Revert version 0.16 and use .odt format again	PG
2023-09-05	0.19	ALL	M	3.8.2 10.2 8	Add reserved area to sysimage table Add information for Yocto 4.0 Add imx8 and Yocto4.0 information	PG
2023-10-09 2023-10-26 2023-10-27 2023-10-30	0.20	ALL	A R M M M M R/A/M	8.15 4.6, 4.8, 4.9, 5.11.2, 5.11.5 1.1 3.7 4 6.8 5.11 → 7	Add ALSA settings for the wm8960 Remove duplicate text sections caused by obsolete regions, in fact remove all regions that were originally pointing to external .odt files Add fsimx8mn and PicoCoreMX8MMr2 to architectures table Keep NBoot part simple, move specific stuff to chapter 4 Add ARM32/64 handling, improve introduction to NBoot Rewrite whole fsimage sub-chapter Move chapter 5.11 to new chapter 7 and rewrite large parts	DA HK HK HK HK HK HK
2023-11-14	0.21	ALL	M	9.21	Add information about bluetoothctl	PG
2023-12-19	0.22	ALL	M	7.2	Add information fsimx6ul emmc-uboot	PG
2025-09-02	0.23	ALL	A	11.2.4	Add chapter about docker	PG
2026-05-27	0.24	ALL	A/M/R	11.1	Modernize Buildroot chapter with Yocto chapter as a basis	DD

V	Version
A,M,R	Added, Modified, Removed
Au	Author

Table of Contents

1 Introduction 1

1.1 F&S Board Families and CPU Architectures....1

1.2 Scope of This Document....3

2 Embedded Linux 4

2.1 What Is Linux?....4

2.2 What is Embedded Linux?....6

2.3 The Command Line Paradigm....7

2.4 Manufacturer Versions....7

2.5 Linux Kernel Layout....9

2.6 Linux Licenses....10

2.7 Patents, Designs, Logos....11

3 Linux On F&S Boards 13

3.1 Linux PC....13

3.2 Connections Between Board and PC....14

3.3 The Parts of a Linux System....16

3.4 Buildroot and Yocto....17

3.5 F&S Download Area....19

3.6 Release Content....21

3.7 Boot Sequence....25

3.8 Mass Storage Devices....26

3.8.1 NAND....26

3.8.2 eMMC....29

3.8.3 SD Card....31

3.8.4 USB Media....31

4 Working With NBoot 33

4.1 NBoot on ARM32 (Vybrid, i.MX6)....34

4.1.1 Entering NBoot....34

4.1.2 Show Bad Blocks....35

4.1.3 Erase Flash....36

4.1.4 Serial Download....36

4.1.5 USB Download....38

4.1.6 Save Bootloader Image....39
4.1.7 Execute Bootloader....40
4.1.8 Create Partitions....40
4.1.9 Access Partitions with USB....40
4.2 NBoot on ARM64 (i.MX8)....41
4.2.1 Handling NBoot....42
5 Working With U-Boot 43
5.1 Entering U-Boot....43
5.2 Commands....44
5.3 Command History....46
5.4 MTD Partitions....46
5.5 UBI Concepts....48
5.6 Environment....52
5.6.1 Environment Variables....52
5.6.2 Special Variables....53
5.6.3 Save the Environment....54
5.6.4 Using Variables in Commands....54
5.6.5 Running Commands in a Variable....55
5.7 Network Configuration....56
5.7.1 Set MAC Address....56
5.7.2 Set IP addresses....57
5.7.3 RNDIS....58
5.8 Image Download....60
5.8.1 Network Download With TFTP or NFS....61
5.8.2 Low-level Read From USB Device....63
5.8.3 Low-Level Read From SD Card....65
5.8.4 Low-level Read From UBI Volume....68
5.8.5 High-Level Read From Filesystem....68
5.9 Image Storage....70
5.9.1 Save to NAND Flash MTD Partition....71
5.9.2 Save to UBI Volume....72
5.9.3 Save to SD Card or eMMC....72
5.9.4 High-Level Write To Filesystem....72

5.10 Exporting eMMC as USB Mass Storage Device....73
5.11 Booting the Linux System....73
6 Special F&S U-Boot Features 75
6.1 Simplified $loadaddr....75
6.2 Allow Wildcards in FAT Filenames....76
6.3 Improved NAND Driver....77
6.4 The Install/Update/Recover Mechanism....78
6.4.1 Regular Boot Process....79
6.4.2 Install Process....79
6.4.3 Update Process....80
6.4.4 Recover Process....81
6.4.5 Configuring the Install/Update/Recover Mechanism....83
6.4.6 The update Command....87
6.4.7 Creating an Appropriate U-Boot Script Image....87
6.5 Linux Boot Strategies....89
6.5.1 Kernel Settings....90
6.5.2 Device Tree Settings....91
6.5.3 Rootfs Settings....92
6.5.4 Mode Settings....92
6.5.5 Console Settings....92
6.5.6 Login Settings....92
6.5.7 Network Settings....93
6.5.8 Init Settings....93
6.5.9 Extra Settings....94
6.5.10 Boot Strategy Examples....94
6.6 NAND Layout Strategies....95
6.6.1 Linux Kernel In MTD partition....97
6.6.2 Linux Kernel In Raw UBI Volume....97
6.6.3 Linux Kernel In Root Filesystem....98
6.7 Configure eMMC For Enhanced Mode....99
6.8 Command fsimage To Handle F&S Images....103
6.8.1 fsimage on ARM32 (PicoCoreMX6UL[100], PicoCoreMX6SX)....103
6.8.2 fsimage on ARM64 (i.MX8)....104

7 Common Ways of System Installation 109

7.1 Installation to NAND Flash....109
7.2 Installation to eMMC....110
7.2.1 Installation From SD Card/USB Pendrive With Same Layout....111
7.2.2 Installation From SD Card/USB Pendrive With System Image File....113
7.2.3 Installing Individual Files....114
7.3 F&S Update and Installation Scripts....115

8 Working With Linux 117

8.1 Busybox....117
8.2 Device Trees....117
8.3 Devices And Device Drivers....117
8.4 Old Way With Board Support Code....118
8.5 New Way With Device Trees....121
8.6 Advantages of Device Trees....123
8.7 Built-in Drivers And Kernel Modules....123

9 Using the Standard System and Devices 124

9.1 procfs....125
9.2 Sysfs....125
9.3 bdinfo....126
9.4 Serial....126
9.5 CAN....126
9.6 Ethernet....127
9.7 WLAN....127
9.8 Qt support....128
9.9 Video Processing (gstreamer support)....128
9.10 I2C....131
9.11 SPI....132
9.12 USB Stick (Storage)....133
9.13 RTC....134
9.14 GPIO....134
9.15 Sound....136
9.16 Pictures....137
9.17 TFTP....138

9.18 Telnet....138
9.19 SSH....138
9.20 VNC....139
9.21 RDP....140
9.22 Bluetooth....141

10 Graphical Environments 143

10.1 Rendering....143
10.1.1 Weston/Wayland....143
10.1.2 X Server technology....144
10.1.3 Qt with OpenGL, EGL, EGLFS and more....144
10.2 Weston/Wayland Configuration....145
10.2.1 General behaviour....145
10.2.2 Weston touch calibration....146
10.2.3 Xwayland....147
10.3 X.Org Server Configuration....147
10.3.1 General behaviour....147
10.3.2 GPU support....148
10.3.3 X.Org Server touch calibration....148
10.4 Qt Environment....148
10.4.1 Qt touch configuration....149
10.4.2 Running Qt5 Example....149

11 Compiling the System Software 151

11.1 Buildroot....152
11.1.1 Install Cross-Compile Toolchain....152
11.1.2 Installing the mkimage Tool....153
11.1.3 Unpacking the Source Code....153
11.1.4 Running Docker....154
11.1.5 Building with Buildroot....154
11.1.6 Cleaning a Buildroot build....156
11.1.7 Rebuild a single package....157
11.1.8 Add Packages to an Image....157
11.1.9 Compiling the Toolchain....158
11.1.10 Compiling U-Boot....158

11.1.11 Compiling the Linux Kernel....159
11.2 Yocto....160
11.2.1 Installing the mkimage Tool....161
11.2.2 Unpacking the Source Code....161
11.2.3 Download Main Yocto....162
11.2.4 Running Docker....162
11.2.5 Configure Yocto for an F&S architecture....163
11.2.6 Build a Yocto Image....164
11.2.7 Rebuild A Single Package....166
11.2.8 Add Packages to an Image....167
11.2.9 Useful utilities....167
11.2.10 Further reading....168
12 Appendix 169
List of Figures....169
List of Tables....170
Listings....171
Important Notice....172

1 Introduction

1.1 F&S Board Families and CPU Architectures

F&S offers a whole variety of boards and modules. Figure 1 shows the Single Board Computers (SBC) NetDCU and armStone.

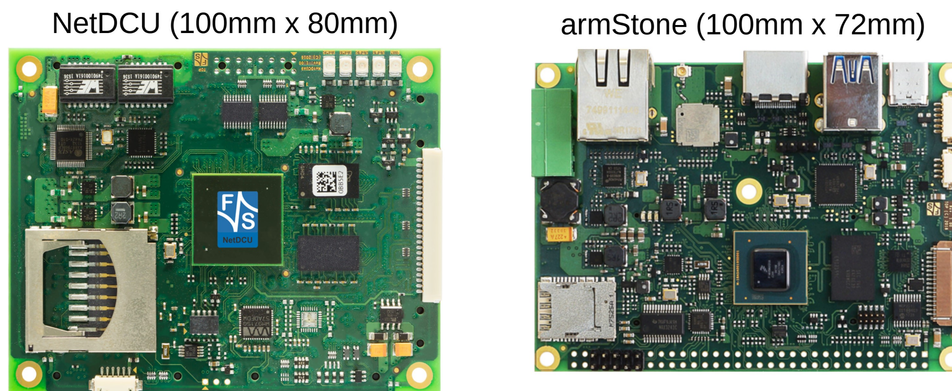


Figure 1: Single Board Computers

Figure 2 shows the Systems on Module (SOM) PicoMOD, QBliss, efus, PicoCOM, and Pico-Core.

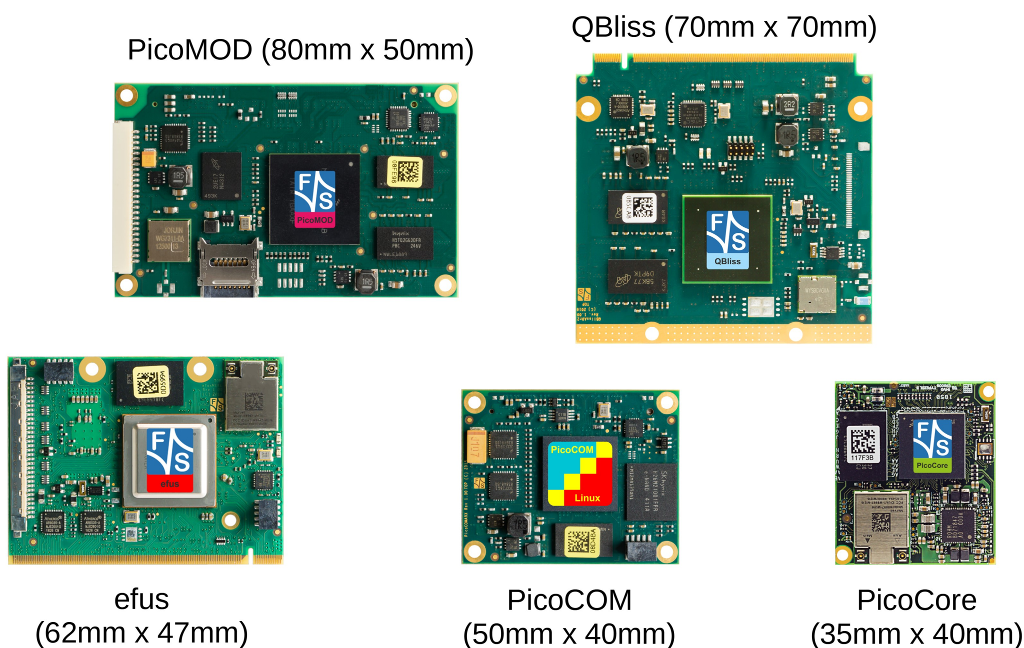


Figure 2: Systems on Module

Each form factor consists of a whole family of boards who have many common features and are often even pin compatible. Linux is available for all of these platforms. In the past, there

was a separate Linux release for each single platform. This was quite cumbersome and also caused a lot of duplicated work, because releases often only differed in small parts, for example in the UART port number for the serial debug port.

As a result, F&S has invested quite some effort in the software to reduce the differences. For example we added a directory with board specific information to the sysfs in `/sys/bdinfo`, so that userspace software has a single point of information to decide for such things as the debug port number. With the effect that now all boards with the same CPU – or rather SoC (System on Chip) – can actually use the same release. We call this *architecture releases*. All the boards of the same architecture can use the same sources, and the binaries can be used on any board of this architecture. Please note the difference: *board families* are grouped by form factor, *architectures* are grouped by SoC type, i.e. they usually contain boards of different families.

Table 1 shows all architectures that are currently supported by F&S. More architectures will be added as soon as new SoCs are used on F&S boards and modules. F&S works hard to ensure that the differences from one architecture to another are as small as possible. In fact most parts of the software use the same source packages. This makes switching from one platform to another rather straightforward and allows the customer to always choose the board that is suited best for a specific application without having to deal with a new development environment all the time.

Architecture	CPU	Platforms
fsvybrid	NXP Vybrid VF6xx	PicoCOMA5, NetDCUA5, armStoneA5, PicoMODA5, PicoMOD1.2
fsimx6	NXP i.MX6	efusA9, efusA9r2, QBlissA9, QBlissA9r2, armStoneA9, armStoneA9r2, PicoMODA9, NetDCUA9
fsimx6sx	NXP i.MX6-SoloX	efusA9X, PicoCOMA9X, PicoCoreMX6SX
fsimx6ul	NXP i.MX6-UltraLite	efusA7UL, PicoCOM1.2, PicoCoreMX6UL, PicoCOMA7, PicoCoreMX6UL100
fsimx7ulp	NXP i.MX7ULP	PicoCoreMX7ULP
fsimx8mm	NXP i.MX8M-Mini	PicoCoreMX8MM, PicoCoreMX8MMr2
fsimx8mn	NXP I.MX8M-Nano	PicoCoreMX8MN
fsimx8mp	NXP i.MX8M-Plus	PicoCoreMX8MP, PicoCoreMX8MPr2, armStoneMX8MP, efusMX8MP

Table 1: F&S Architectures

1.2 Scope of This Document

This document is a generic approach to explain the F&S Linux environment on F&S boards and modules. We will have a look at Linux in general, at the bootloaders NBoot and U-Boot, and especially how software is installed. Then we will look at the build environments Buildroot and Yocto and how the system software is built. And finally we will show how some specific peripherals will be used, for example I²C, CAN, SPI, sound, the gstreamer multimedia infrastructure and the graphics environment.

This is a rather new document, so some places may still be a little vague and some topics are still missing completely. But we will continue to add to the text over time and thus gradually will generate a kind of reference book that comprehensively covers all aspects of Linux on F&S boards.

Please note that this text is not specific to one of the architectures, we try to be as general as possible. Of course, we have to distinguish from time to time when things on different architectures differ a lot, but basically all architecture dependent stuff is covered in the appropriate FirstSteps documents. Especially all the links to hardware specific documents can be found there.

2 Embedded Linux

2.1 What Is Linux?

Linux is an Open Source Operating System. Which means the source code of the Linux kernel itself and also of thousands of userspace applications, the so-called *Userland*, is available for free. Linux is running on most CPU architectures, including x86 and ARM, but also on many other architectures from small CPUs for IoT devices (Internet-of-Things) to large Main Frames and High Performance Computers (HPC). Linux is feature-rich and fully scalable. The number of applications is correspondingly diverse.

- Set-top boxes, media players, DVD/Blu-ray players, TVs
- Network devices, routers
- Storage devices (NAS)
- Home automation
- Industrial and medical applications

The Linux kernel including the device drivers also forms the basis for Android, so in principle all Android devices, i.e. smartphones, tablets, etc., are also Linux devices. In fact most people already use Linux devices, perhaps without knowing it (see Figure 3).

Figure 3: Some Linux devices

All these applications ensure that Linux continues to evolve at a rapid pace. Video and audio codecs are often available for Linux first, also WLAN and Bluetooth drivers. In fact nowadays even Linux itself is the first OS available for most Systems on Chip (SoC), because the SoC

manufacturers use Linux to test and verify their hardware. So when the chip is released to the public, there is also immediately a first version of all necessary Linux device drivers available.

Linux development is done by the “Community”. On one hand, the Community consists of professional software developers who are employed by companies and earn their money by developing software for Linux. On the other hand, there are many private enthusiasts who contribute code to Linux that they developed in their spare time. In this way, about every nine to ten weeks a new version of the Linux kernel is released. This is the so-called *Mainline Kernel*.

Despite this variety of people working on the kernel, the quality is at a very high standard. If you want to contribute a change to the kernel, you must first post it to a mailing list. There the code is checked by many people for bugs, shortcomings and even coding style. In most cases there are still complaints and the code has to be revised again. Often several iterations of these revisions have to be performed until the code satisfies the requirements of the kernel maintainers and is then actually integrated into the next kernel release.

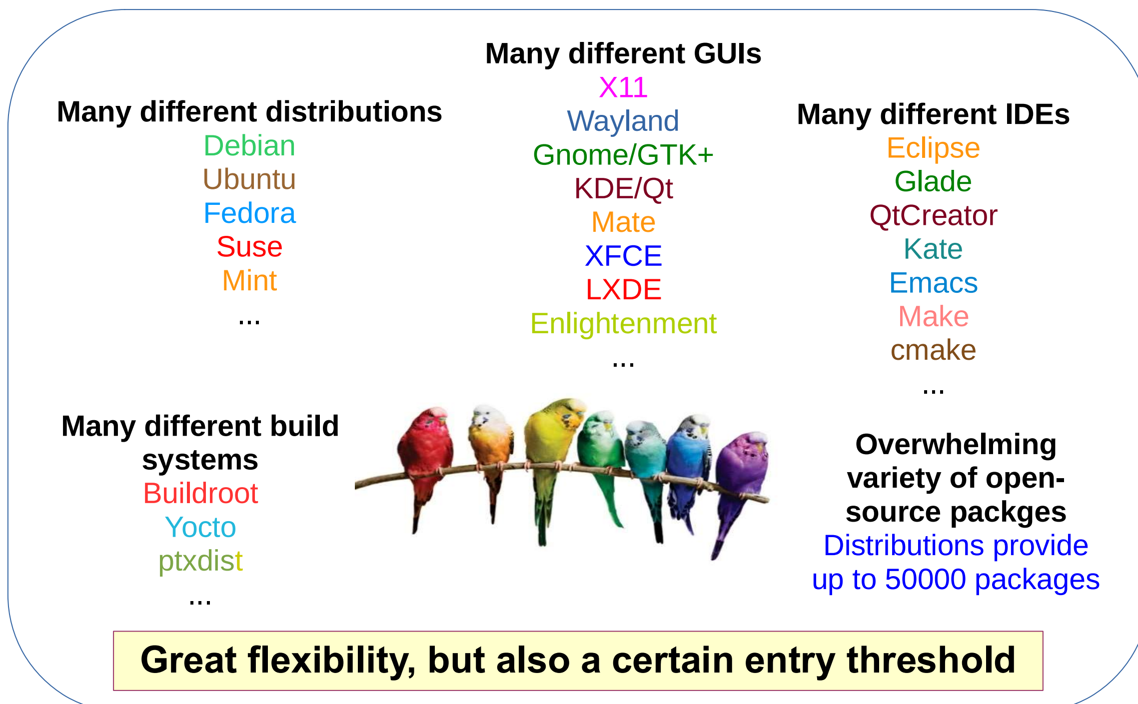


Figure 4: Linux diversity

One of the biggest advantages of Linux is its diversity. As you can see in Figure 4, there are many distributions with different objectives, many graphical user interfaces, many build systems, many development environments and an overwhelming amount of open source software packages in the Linux ecosystem. So you can choose from a huge amount of libraries, approaches, versions, variants and other options. This flexibility is gigantic.

On the other hand, this is also a problem. Many people miss the red thread that leads them directly to the goal. Especially Linux beginners often find themselves helpless in front of a huge mountain of possibilities and don't know where to start. This problem is very difficult to

overcome. It simply needs experience. If you work with Linux, you will definitely have to try a lot, read a lot, research a lot, compare a lot. But since you can almost always find code that you can reuse, this effort is well invested and saves a lot of time in the end. And after a while you will find your individual way through the Linux jungle.

F&S tries to help by providing workshops, documentation and a support forum. Attending these workshops and carefully reading the documentation may save you many hours of own research and unnecessary tries. If there are still unanswered topics, you can ask in our forum. Our software and hardware engineers will answer quickly and competently.

2.2 What is Embedded Linux?

An Embedded Linux System basically does not differ from a regular Desktop Linux System. It uses the same source code, it uses the same compilers and software tools and it even uses the same Userland packages. However there are often some constraints that need attendance.

- Embedded Systems usually have less RAM. It is counted in megabytes, not gigabytes. This requires to keep the system small, only start those background services that are really necessary, sometimes software is even optimized for size instead of speed. Also leave out unnecessary parts of software. For example if you know that your application will never need 3D features or web assembly, you can compile a smaller version of the web browser without these elements.
- Embedded Systems usually do not have a hard disc drive, only small flash media. So a full-blown distribution that needs several gigabytes of storage is most probably not appropriate. Even the hundreds of common command line tools take too much space and are often replaced by a compact single tool called Busybox.
- Embedded Systems are often battery-powered. So it is important that all power-saving functions actually work. Sleeping modes like Suspend-to-RAM are essential and fast wake-up features need to be implemented. On the desktop, this is often not a priority topic.
- Embedded Systems are often connected to industrial buses like serial ports, SPI, I²C, CAN and similar, but do not have many PCI lanes, if any at all. So common hardware extensions known from PCs can not be used. On the other hand the SoCs in Embedded devices have lots of built-in periphery, so that additional external hardware is often not necessary at all. This means different device drivers are needed compared to a regular desktop PC.
- Embedded Devices are often headless, i.e. do not have a display. So other means of communicating with the device need to be established. If they have a display, then most probably a touch screen is used, not a mouse. This has to be thought of when designing the GUI. Often no support for high-performance graphics is available, or it is significantly less performant than on a desktop PC. The same is true for multi-media features like video playback.
- Embedded Systems are often not based on x86 CPUs, but on ARM or other SoCs. So we need cross-compilation and different libraries.
- Embedded devices are often used in rough environments, for example outdoors where it can be very cold or hot, or even wet. It is also possible that the power supply

is unstable or the power is simply cut off by pulling the plug. This needs very robust filesystems, and also recovery capabilities. Maybe regular filesystems need to be mounted read-only, which is not possible with regular distributions.

An Embedded Linux system is therefore very similar to a regular desktop Linux system, only the weighting of the individual objectives is different.

2.3 The Command Line Paradigm

As already mentioned there are many different graphical user interfaces under Linux. This makes it difficult to develop graphical tools to configure and manage the system. Even for small tasks you would have to program many variants, one for each graphics system. The instructions would also be correspondingly complicated: "If you use graphical user interface A, then use tool X and click first here, then there. If you use graphical user interface B, then use tool Y and click here, then there, then there and finally there." And everything would have to be accompanied by a long series of screenshots. In fact, that would be completely impracticable.

That is why Linux decided from the beginning to use small command line tools. Each of these tools does its job, and only that. This makes these tools very compact and rather trivial to write. (KISS: "Keep it small and simple"). If you want to have a graphical variant, just add a graphical frontend, but still call the command line version in the end.

This approach has several advantages.

- Instructions how to achieve something are rather simple: just show the commands to use. The user can even copy the text from the document and paste it in his own command shell. Done! So documentation and help instructions are easy to write. No complex screenshots are necessary, that may even change over time.
- If there is an error in the command line tool, just fix it and all graphical versions are also fixed immediately.
- You can use these commands in scripts for automated processes, for example to prepare a Docker container, do a complex software build or just make some recurring tasks easier.
- You can control and manage also headless systems, i.e. systems that do not have a display or graphical user interface at all. You just need a serial port or a network shell like `telnet` or `ssh`.

So when you work with Linux, you will definitely get in contact with the command line. But that is not a sign of backwardness. It is a paradigm without which the diversity of Linux would not be manageable. And after a while you will be astonished how powerful and efficient the work with the command line is.

2.4 Manufacturer Versions

Unfortunately the mainline kernel release cycle every nine or ten weeks is too fast for some manufacturers. If a SoC manufacturer is about to release a completely new chip, software development may take several months, sometimes even years. During this time, they test

and verify the hardware, write device drivers and port and modify several software packages. If they also had to handle a new kernel version every two months, this would lead to chaos.

This is why the manufacturer takes the then current mainline kernel (or the latest LTS version with long term support) when he starts with the software porting. Then he works for several months and at the end he releases his own version of the kernel. This is still the same version, but now his source code contains all the modifications and drivers that are needed for this new SoC. So this is the manufacturer version of the kernel.

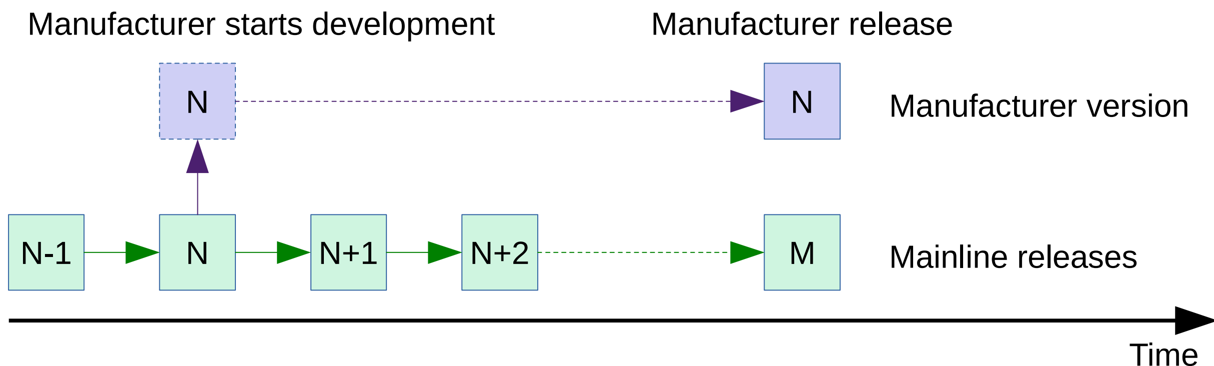


Figure 5: Mainline and manufacturer versions diverge

Of course in the meantime, the mainline kernel had quite a few more releases, maybe ten or even more. So the version numbers of mainline and manufacturer kernel diverge (see Figure 5). The mainline kernel has the newest features, but the manufacturer kernel is the only one that supports this new SoC.

When we at F&S create a board with the new SoC, we are forced to use the manufacturer version of the kernel, because we need the supporting code for the SoC. But on the other hand we also have other hardware that is not necessarily restricted to the manufacturer version. We often port back newer drivers for this hardware to provide the best possible experience for the customer.

The same is true for the Userland. For example it requires modifications to the gstreamer multimedia infrastructure to support the video hardware encoding and decoding acceleration that is available in some SoCs. Again there may be changes or even own versions of the manufacturer, and again F&S has to provide additional modifications so that everything works right out of the box.

This results in the following stack of modifications (see also Figure 6):

- The mainline version provides the basic functionality
- The manufacturer version provides support for his SoCs and evaluation boards
- The F&S version provides support for F&S boards and modules

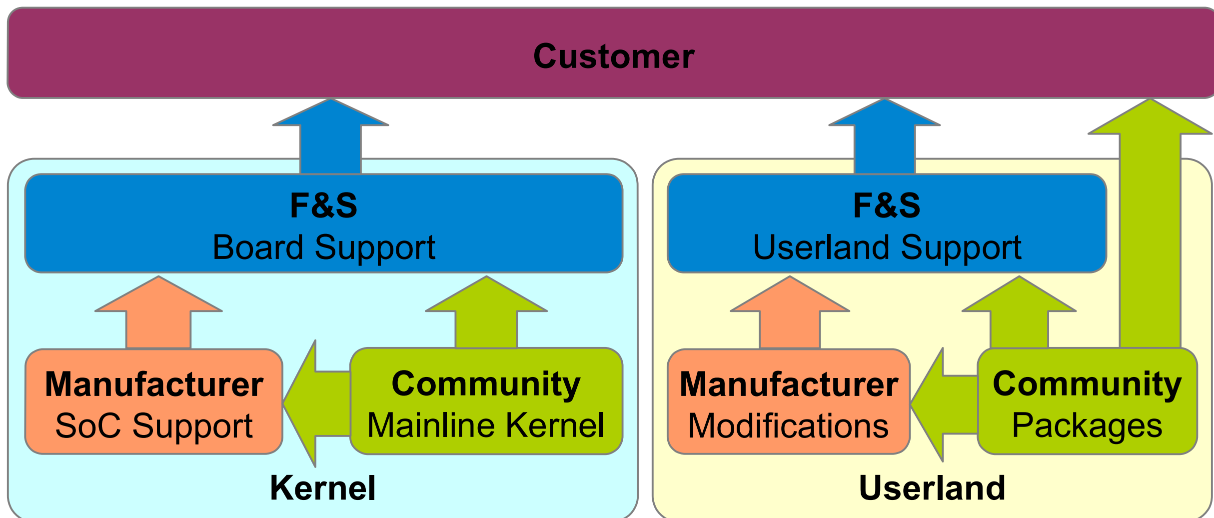


Figure 6: Mainline versions, manufacturer versions and F&S support

The goal is that the customer no longer has to worry about device drivers and the kernel, but can start directly with the development of his own application.

2.5 Linux Kernel Layout

Figure 7 shows the structure of the Linux kernel. Whenever a Userland application wants to access the hardware, it is not allowed to do so directly, it has to ask the Linux kernel to do this. The kernel then checks if the software is privileged to access the hardware and synchronizes any concurrent accesses.

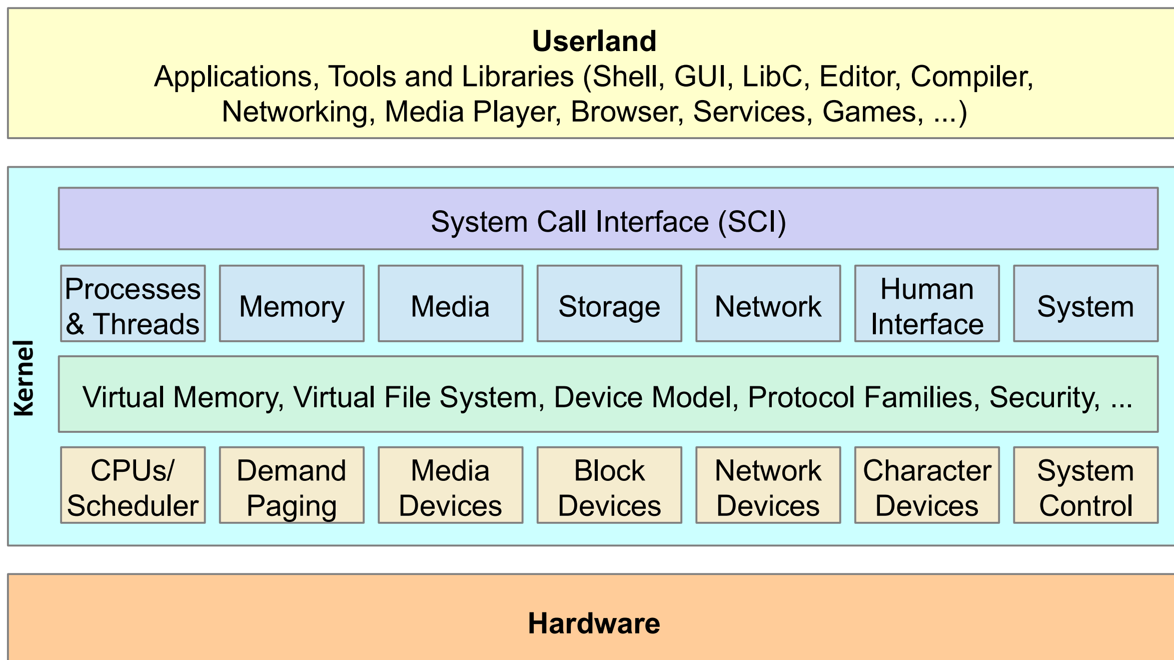


Figure 7: Hardware accesses must go through the Linux kernel

To talk to the hardware, the Linux kernel uses device drivers. Each type of hardware may require an own device driver. Even hardware with the same functionality may differ. A UART device on a PCI slot may look completely different to a UART device of one SoC, and this may again look totally different to the UART device of a second SoC. So there are many different UART drivers included in the kernel for all the different UART hardware devices. But to the Userland side, the kernel will present all these different UART devices in a generic form. All UART devices will have a read and a write function, a function to set the baudrate, a function to set the flow control (none, XON/XOFF, RTS/CTS, etc.) and so on. So one task of the kernel is to abstract from the underlying hardware and present the different devices in a uniform, virtual form.

The kernel also manages all other resources, such as RAM, storage, processes and threads, input devices (keyboards, mice, touchpanels), output devices (displays, LEDs), GPIOs, etc. Whenever some software has to handle these things, it has to access the kernel. This is done via the System Call Interface. However these calls are actually part of the C library. The C library is tightly coupled with the kernel. For example if the user does a call to function `malloc()`, this will implicitly do a system call and instructs the kernel to provide some RAM from the heap.

Or if you want to send some data to a UART port, simply open the appropriate device under `/dev`, probably configure the port with `tcsetattr()` and then write the data to the file descriptor. In fact each of these library functions will do at least one kernel call that actually does the real work. So in fact you will never get in contact with the System Call Interface directly,

2.6 Linux Licenses

The license for the Linux Kernel itself is the GNU General Public License (GPL) Version 2. This license gives the right to everyone to use and modify the code. And it gives the right, to re-publish the code, but only if you do not restrict the right to use and modify the code in any way. This means you have to provide all licensees (your customers) with the source code if they request it, otherwise they obviously can not use and modify the code. In other words if you only use the Linux kernel for yourself, you can do with it what you like. But if you pass on a modified version, then you have to also pass on the full source code with all your modifications. Which means you have to make your modifications public, not pro-active, but on request. You can not add something to the kernel that is Closed Source.

Software in the Userland is different. You can choose whatever license you want to publish your software application. However if you want to reuse any existing code, either as source code or as a binary library, you carefully have to check if the license of this code suits your needs. There is lots of code that is published under a license that does not require you to do anything (MIT, Apache, BSD). Other licenses may require that you mention the used software in your code or documentation. And there are licenses that require you to fully open-source your own code (GPL).

So it is very important, that you check the licenses of all code that you use. If the license requests anything that you do not want to fulfil, than you must not use this code. Either look for some other code or write this code on your own. For example if you use code that is published under GPL, then you must make your own code public. This is usually not what you want for industrial or medical applications. If you use a library that is published under the

LGPL (Lesser GNU General Public License), then you can use this library also in a closed-source application, as long as the library is linked as a shared library. Then you only have to publish and make available any code that you add to the library itself, but not the code of your main application.

Some companies use more than one license for their code. For example The Qt Company has three different licenses for their Qt5 software and you can choose which license you want to use for your software product.

- GPLv3. This license can only be used if you want to do an open-source project.
- LGPLv3. This license allows closed-source development if Qt5 is linked as shared library. However not all Qt5 modules are licensed under LGPL. Especially newer parts for modern 3D-based user interfaces and cloud based internet protocols are not yet available. So you might be restricted to older Qt5 features if you want to use this license.
- Qt Commercial License: This license is not for free. It costs money per developer seat and also per device sold. But you have no obligations or restrictions, you can use from Qt5 whatever you like.

So it is of course possible to write closed-source applications in Linux. But be careful what code you reuse. Also note that some third-party code is not for free.

> **Note**
>
> The license is only relevant if you directly use code in your application, either by directly including the sources in your own code or by a library that is linked to your code. But in the case of a stand-alone program that is started independently, the license does not matter. For example you can execute an independent GPL program at any time without having to disclose anything of your own code.

2.7 Patents, Designs, Logos

Unfortunately “Open Source” does not necessarily mean “No Costs”. The Linux system itself is for free. Any costs involved with Linux are mainly costs by service providers for additional support or software features. But there are some topics that may not be for free.

Video and Audio Codecs

Video and audio codecs are often protected by patents.

- MPEG-2 (DVD), MPEG-4 (H.263, H.264, H.265) have to be licensed from the MPEG License Association
- WMV (VC1) and WMA have to be licensed from Microsoft
- Dolby Digital (AC3) has to be licensed from Dolby Laboratories
- Digital Theater Sound (DTS) has to be licensed from DTS, Inc.

Some Codecs are supposed to be free.

- Google guarantees that VP8 and VP9 Codecs, used in WebM, can be used freely. Google will cover all costs in case any patent claims (e.g. from patent trolls) arise.
- The patents on MP3 expired in April 2017. Since then MP3 can be used freely.

Graphics and Computing Libraries

Also Graphics and Computing Libraries may not be free.

- OpenGL, OpenVG and Vulkan have to be licensed from Khronos Group
- OpenCL has to be licensed from Khronos Group
- CUDA has to be licensed from Nvidia; the same may be true for machine learning libraries

Fonts and Images

Fonts and graphical images may be covered by copyrights or design patents. If you want to use them, you have to license them. Especially you might be tempted to use common fonts like Arial or Courier New. But these fonts are copyrighted by Microsoft. When using a Windows system, you are of course allowed to use these fonts. But not necessarily in a Linux system. So please make sure if using such fonts is allowed. There are often free versions of similar fonts available, for example the Liberation font family for Linux (Liberation Sans instead of Arial, Liberation Mono instead of Courier New, etc.)

Logos for Hardware Support

If you want to officially use logos for USB, WiFi, HDMI, Bluetooth and similar, you are usually obliged to participate in the appropriate association, requiring an annual fee. In addition, conformity tests may be required, to guarantee that the standard specifications are not violated. For example WiFi/WLAN chips must conform to the regulatory specifications for the country they are used in, i.e. the maximum allowed radio transmission strength must not be exceeded and only the released frequency channels may be used.

Important Notice

The default configurations and sample images that F&S provides in their BSPs are not meant for final distribution. They may contain audio and video codecs that are not free, they may contain Qt software that would require the commercial license and they may use software tools or hardware drivers (for example WLAN software) that would require conformance tests or software licenses.

So please create your own images by removing all parts that you do not need and adding all extra parts that you need. You are responsible for paying all license fees as required by the software you publish to your end customers.

3 Linux On F&S Boards

3.1 Linux PC

When working with an F&S board, development is done on a PC and then software is transferred to and executed on the board (see Figure 8). This means we need to do cross-compilation, i.e. compile code on an x86-CPU, but for the ARM-CPU of the board. This has two impacts.

1. When building software, the configuration step can not check the environment of the final system on-the-fly. Such parameters have to be given beforehand. This makes cross-compilation slightly more complicated in some cases.
2. The resulting software can not be executed and tested on the PC, it can only run on the board itself.

However the higher performance of the PC when building software is a much better benefit. A PC has more CPU cores, more speed per core, more memory, more hard disc space. And this is really necessary when building your own root filesystem. For example when building a Yocto system with the Chromium browser, you need about 100 GB of hard disc space, at least 10 GB of RAM and this takes about 10 to 16 hours on a 3 GHz 64-bit Quad-Core Intel Core i7 with SMT (Hyperthreading). Try to do this on a board with a 1 GHz 32-bit Single-Core ARM CPU, with 1 GB of RAM, no hard disc and with only 512 MB of NAND flash or 4 GB of eMMC. Other than that only rather slow external storage media like USB stick or SD card is available, that will also need to be used as swap space. This will take ages to build. And remember, you might need to do this more than once.

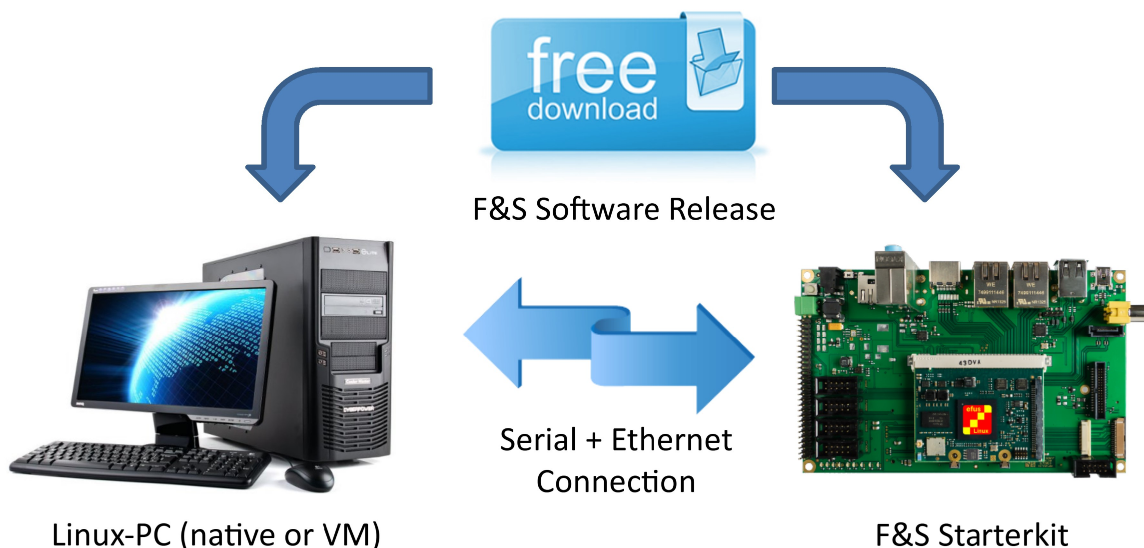


Figure 8: F&S development environment

When working with Linux on the board, you also need a Linux based PC for the software development. However this PC does not necessarily need to be a native Linux PC. It can also be a virtual machine running on a native Windows PC. In fact this configuration has some advantages, because then also Windows tools can be used, too. For example F&S provides

some tools that were originally used with Windows Embedded Compact based boards and are therefore only available for the Windows platform (`DCUTerm`, `NetDCUUSBLoader`).

The type of distribution that is used on the PC does not matter. If you have any preference, feel free to use it. We at F&S are using Fedora, simply because we like this distribution, especially with the MATE GUI. The Linux system is running as virtual machines in VirtualBox (Oracle) under a Windows host. VirtualBox is available for free. (But please be careful, because the VirtualBox Extension Pack that is used to pass on USB devices into the VM is *not* free when used in commercial products.)

This is why we also have pre-configured virtual machine images with Fedora and also some documentation that explains how to configure everything in Fedora. So if you do *not* have any preference, we recommend using Fedora because there we have the most knowledge and can help best.

Note

You will find the document `AdvicesForLinuxOnPC.pdf` on the F&S website that explains how to set up a Linux based development machine.

F&S also offers a pre-configured virtual machine image based on VirtualBox. Just install the free software VirtualBox on your PC. The operating system of this PC does not matter, VirtualBox is available for all major systems. Then download the image from our server and import it in VirtualBox. As a result you will have a virtual machine running a Fedora Linux and with all sources of a current release installed and pre-compiled.

There is a document `Quickstart with F&S Development Machine.pdf` that explains this procedure in more detail.

As you already have seen in Chapter 2.4 with the manufacturer kernel, it may take some time until software is available for ARM platforms or specific SoC variants. This is not only true for the kernel, but also for the Userland. Software is almost always developed for PCs and it takes time until patches for other CPU architectures slowly migrate into the original packages. It also takes time until build environments like Buildroot and Yocto update to newer packages. And then it takes again some time, until we at F&S have ported all this stuff to our boards and modules.

As a result it is often not the best choice to have the newest Linux distribution on your PC. If compiler versions (GCC), other build tools or libraries are too new, you may get problems when building some older packages. A good choice is an LTS version that is about one or two years old.

3.2 Connections Between Board and PC

On the other side there is the board. The Starterkit provides all necessary cables that you need for development. In case of a module, also a baseboard is part of the Starterkit. The baseboard brings all signals of the module to common connectors, like SD card slots, USB Host Type A, Ethernet RJ45 or Jack connectors for audio. Other signals, that do not have a well-defined connector, like SPI, I²C, CAN or GPIOs, are available on two-row pin connectors.

For the specific location of each connector on each platform, especially the serial debug port, please refer to the FirstSteps document of your board. There are images that show the connectors on the boards, modules and Starterkit baseboards.

You will use at least a serial connection between PC and board. The cable is connected to the serial debug port on the board side. If your PC does not have a serial port you need to install a PCIe card with a serial port or you need to use a USB-to-serial adapter (see Figure 9). Please note that some cheap adapters do not work reliably because they have a very high deviation from the standard baud rate.

Figure 9: Serial port options for PCs without native port

The serial connection is used to log-in and control the board with simple text commands. This means we also need a terminal program on the PC to enter these commands and see the results. We recommend a terminal program that supports a 1:1 binary download and also supports ANSI Escape Sequences for colour and text highlighting. Examples are:

- `TeraTerm` (Windows)
- `PuTTY` (Windows/Linux, does not support 1:1 download)
- `minicom` (Linux, does not support 1:1 download, but not needed in Linux)

F&S also provides a small terminal program for Windows called `DCUTerm`. You can find `DCUTerm` in the Tools-Section of the Download Area (in *My F&S*). However `DCUTerm` does not support ANSI Escape Sequences, which means the output of a Linux command like `ls` is nearly unreadable. Instead of different colours for different file types, you will see a mixture of file names and verbatim escape sequences. Also accessing the command history with the up and down arrow keys is not possible in `DCUTerm`. So `DCUTerm` is not suited very well for Linux. However it supports a 1:1 binary download. So `DCUTerm` is actually a good companion for `PuTTY`. Use `DCUTerm` for serial downloads and `PuTTY` for everything else.

The serial protocol used on the serial debug port is 115200 baud, 1 start, 1 stop bit, no flow control. So use these settings in you terminal program.

In most cases, during development you will also have an Ethernet connection to the board to download larger images. These files are usually transferred with TFTP, so you need to install some TFTP server software on the Linux PC. If you also want to use NFS, for example when exporting the root filesystem from the PC, you also need an NFS server running. These services are usually available in the package repository of the PC Linux distribution and can easily be activated. If you want to download files via TFTP from Windows, there is a small program called `tftpd32` by Philippe Jounin that can be used as TFTP server.

3.3 The Parts of a Linux System

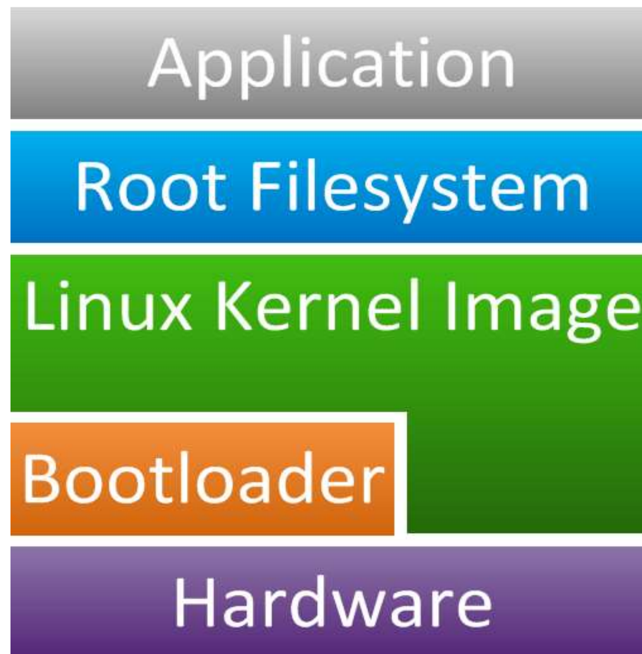


Figure 10: Components of a Linux system

If you look at Figure 10, you will see that a typical Linux system has several layers. At the bottom, there is the hardware of the platform. A bootloader initializes and configures the hardware and then starts the Linux kernel that contains all the device drivers, controls the memory and data storage and handles process execution. All system and userspace programs, tools and data files are located in a filesystem that is usually called *Root Filesystem.* Finally the customer application is executing the function that the device is dedicated for.

On a regular desktop Linux, there may be an additional layer, the initial ram disk (initrd). This is a small intermediate root filesystem that is sometimes needed before the real root filesystem can be activated. However in the default configuration of all F&S boards, this additional layer is not necessary. We can directly start into the main root filesystem.

The Linux system from F&S covers the following parts:

Hardware

The platform manufactured by F&S.

Bootloader

The bootloader is split into two parts: a small first level loader called NBoot that simply calls the main bootloader and the main bootloader itself, called U-Boot. U-Boot activates the hardware, loads the Linux kernel and executes it. U-Boot is also used to download and install the images for Linux kernel and root filesystem. It can also boot the board in different modes, for example locally or via the network by downloading the images on-the-fly.

Linux Kernel

This is a Linux kernel modified to support F&S boards. The Linux kernel image is the operating system of the device. It provides the device drivers, filesystems, multitasking and all I/O features that the board supports. Beside the kernel there are device trees. The device tree describes the hardware of the platform. It tells the kernel what devices have to be activated. While a kernel image may be used across different platforms, every platform needs its own device tree file.

Root Filesystem

We use a Buildroot or Yocto based root filesystem. The root filesystem is the filesystem that you see after the kernel has booted. It contains the userspace programs, services, libraries and configuration files required to run the Linux system and applications. The default root filesystem supplied with the board has a Busybox for starting the system and to provide all standard command line tools, some ALSA tools for sound, gstreamer for audio and video processing, a rudimentary X-Server to show some graphical user interface after startup and of course all the shared libraries like glibc.

Currently the bootloader U-Boot and the root filesystems based on Buildroot are mostly combined for all F&S platforms, but the Linux kernel is still different for some architectures.

3.4 Buildroot and Yocto

Usually when you are using Linux on a desktop, you are actually working with a specific Linux distribution like Ubuntu, Debian, Suse, Fedora, or similar. There are more than 100 distributions available, most of them with a specific purpose, for example best hardware support, easiest installation, most stable software, and so on. A distribution means that someone has assembled a large collection of software, has subdivided all the software in small packages and makes these packages available as a software repository. You can decide what packages to use and you can download and install additional packages later at any time and you can also uninstall unused packages. The distribution also provides updates, for example to fix security issues or to have newer versions with additional features available.

This results in a very flexible and universal system that is suited for nearly every purpose. However if you want to use an embedded system, you most probably have a specific application in mind. And in such a case a full-blown distribution also has a few disadvantages.

- A distribution is mainly targeted for desktops. This means installation, configuration and usage often implies some kind of display and many components imply that they can show their output on a Graphical User Interface (GUI).
- This also means that all components use a very comprehensive configuration. For example a web browser will support 3D, Web Assembly, multimedia and similar stuff, resulting in a rather large binary. And this binary will pull in a lot of libraries and thus a lot of additional packages. So even if you know that you will never need 3D, you can not uninstall the 3D library package alone because you could not start the browser without this library. The package manager of the distribution will therefore automatic-

ally uninstall the browser package, too, if you uninstall the 3D library. This means a distribution can not easily be stripped down to a small system that is specifically targeted for your application. You have to carry a lot of extra ballast around that can also cause additional security issues, even if you never use it.

- Distributions often start a lot of background services that are not required on an embedded system. This slows down the boot process and increases the RAM footprint unnecessarily.
- A generic distribution for ARM often does not support specific hardware features of specific SoC variants. Then you probably can not use hardware accelerated video decoding or similar things.
- Distributions assume that they have lots of RAM and storage memory. To guarantee that a distribution will run on an embedded system, you must also have boards with large amounts of memory, making the devices more expensive.
- Distributions assume swap space. This is usually not available on an embedded system.
- Distributions rely on an initrd, making the boot process unnecessary complex and slowing down the boot process again.
- Distributions are not aware that they are probably running on flash memory based media. So they write to log files arbitrarily often, the `/tmp` directory is not in a ramdisk, they do not optimize write cycles and they do not care for wear-levelling. This might reduce the lifetime of a flash memory based device considerably.
- Distributions assume that they are shut down in a proper way. If you simply switch such a device off, they will lose data. Shutting down without losing data implies a read-only mounted root filesystem, which is not possible with regular distributions.
- Distributions use the regular set of command line tools. Each tool is rather small, but if you have hundreds of them, this sums up to a considerable amount of memory, also increasing the memory footprint unnecessarily. They do not use Busybox.

From this point of view, a distribution is not the best solution. It is large (a minimal system with graphics is about 1 GB of size), it is slow and it may not support SoC specific hardware acceleration. But there are other build environments that actually do address these issues. Two of them are Buildroot and Yocto and both are available from F&S.

The basic idea of these environments is that you create an own small distribution that just contains those packages that you really need and even these packages are configured in a way that no unnecessary parts are included. This results in very small Linux systems. Our standard image with X11 and a small window manager called matchbox, gstreamer1 for multimedia, ALSA for sound, a set of tools to access CAN, I²C and SPI and a picture viewer, is just about 90 MB in size.

We will deal with Buildroot and Yocto in more detail in later chapters. For now it is sufficient to know that F&S provides releases for both environments. There are releases for Buildroot and releases for Yocto. So you can decide which way to go and which release to download. If you buy a Linux Starterkit from F&S, it will contain the newest Buildroot release by default.

3.5 F&S Download Area

If you want to download hardware and software documentation, go to our main website

http://www.fs-net.de

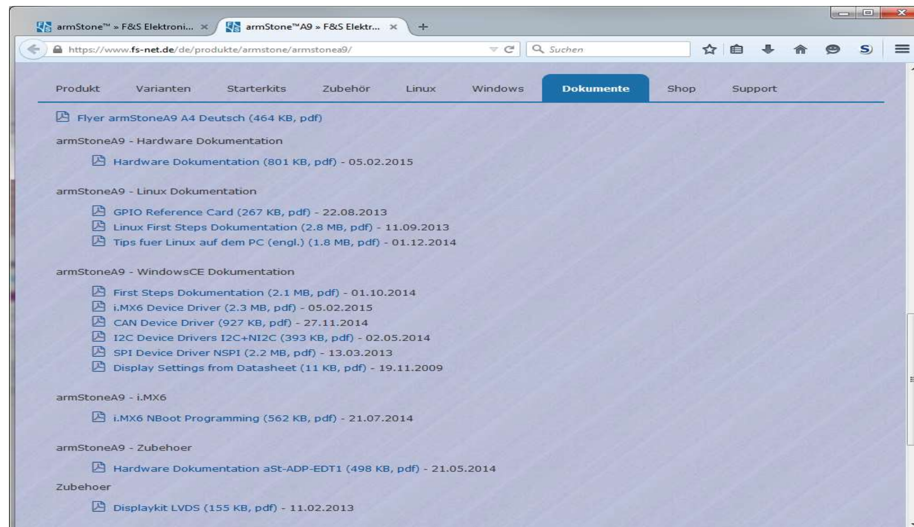


Figure 11: Download documents from F&S website

Select *Products* from the menu at the top, then the board family and finally your specific board. The top half of the screen will now show the board and its features. And in the lower half of the screen you will find an additional menu. Select *Documents* there (see Figure 11).

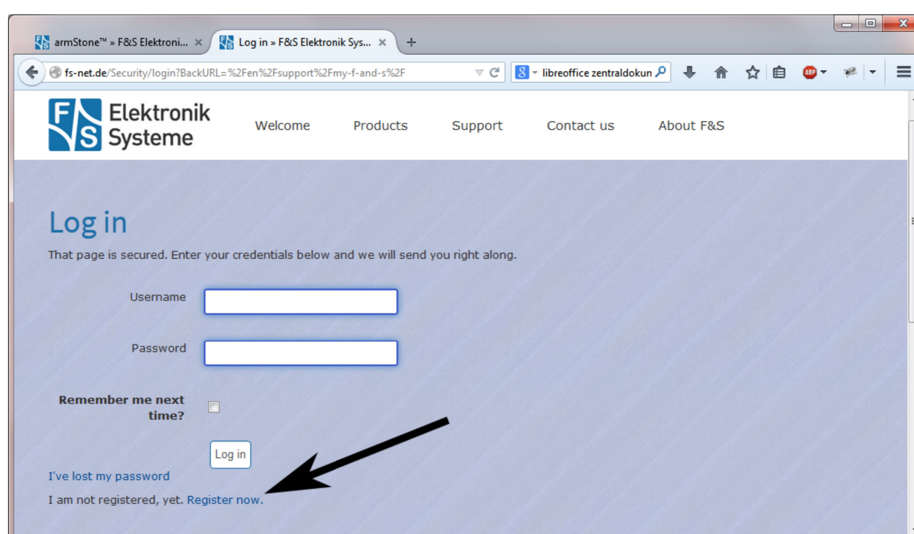


Figure 12: Register with F&S website

To download any software, you first have to register with the website. Click on *Login* right at the top of the window and then on the text “I am not registered, yet. Register now” (see Figure 12).

In the screen appearing now, fill in all fields and then click on *Register*. You are now registered and can use the personal features of the website, for example the Support Forum where you can look for solutions to any problems and where you can ask your own questions. These questions are usually answered by the *F&S Support Team* or also sometimes by other users.

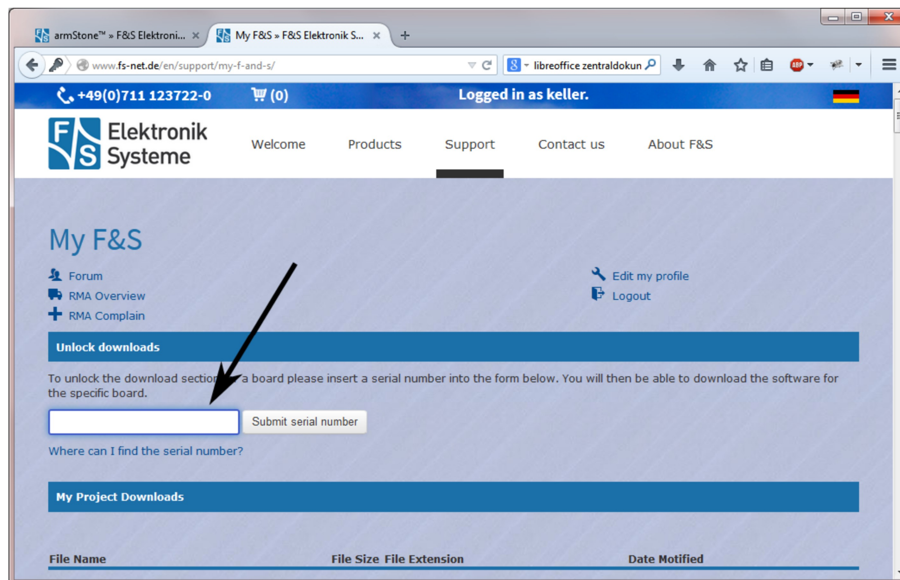


Figure 13: Unlock software with the serial number

After having logged in, you are at your personal page, called “My F&S”. You can always reach this place by selecting *Support* → *My F&S* from the top menu. Here you can find all software downloads that are available for you. In the top sections there are private downloads for you or your company (may be empty) and in the bottom section you will find generic downloads for all registered customers.

To get access to the software of a specific board, you have to enter the serial number of one of these boards (see Figure 13). Click on “Where can I find the serial number” to get pictures of examples where to find this number on your product. Enter the number in the white field and press *Submit serial number*. This enables the software section for this board type for you. You will find Linux, Windows CE, and all other software and tools available for this platform like `DCUTerm` or `NetDCUUsbLoader`.

First click on the type of your board, e.g. armStone*A9*, then on *Linux*. Now you have the choice of *Buildroot* or *Yocto*. Click on one of these. This will bring up a list of all our releases. Old releases up to year 2018 had V<x>.<y> as version identifier for Buildroot and Y<x>.<y> for Yocto. New releases since 2019 use B<year>.<month> for Buildroot and Y<year>.<month> for Yocto. We will abbreviate this as <v> from now on.

Select the version you want, which is most probably the newest version, for example *fsimx6-B2020.04* if you are using a board from the fsimx6 architecture. We will use <arch> for the architecture part of the filename from now on. This will finally show two archives that can be downloaded.

<arch>-<v>.tar.bz2 .... This is the main release itself, containing all sources, the binary images, the documentation and the toolchain.

`sdcard-<arch>-<v>.tar.bz2`.... Files that can be stored on an SD Card or USB stick to allow for easy installation.

The SD card archive is meant for the case that you just quickly want to have the binaries for installation. We used this archive in the FirstSteps documentation. But these files are also contained in the main release. So if you consider downloading the main release anyway, don't bother with the SD card file.

3.6 Release Content

These tar archives are compressed with bzip2. To see the files, you first have to unpack the archives

```
tar xvf <arch>-<v>.tar.bz2
```

This will create a directory `<arch>-<v>` that contains all the files of the release. They often use a common naming scheme:

`<package>-<platform>-<v>.<extension>`

With the following meaning:

`<package>`.................................. The name of the package (e.g. `uboot`, `linux`, `rootfs`). If it is a source package, we also add the version number of the original package that our release is based on, for example `linux-5.4.70`.

`<platform>`................................ The name of a board, if the package is only valid on one board (e.g. `armStoneA9`); or the name of an architecture, if the package is valid on different boards of the same architecture (e.g. `fsimx6`), or the string `f+s` or `fus` if the package is architecture independent.

`<v>`... Release version, consisting of a letter (`B` for Buildroot based releases, `Y` for Yocto based releases) and the year and month of the release (e.g. `Y2021.04`).

`<extension>`.............................. The extension of the package (e.g. `.bin`, `.tar.bz2`, etc.). Please note that some file types do not have an extension, for example the `zImage` file of the Linux kernel.

Some filenames also contain a distro component as part of the package name. The distro defines a special variant of the compilation process, mainly in Yocto. We use this to set the graphical subsystem, e.g. `x11`, `fb`, `wayland` or `xwayland`. So for example there may exist several different Qt5 images, one on top of X11 (distro `x11`), one on top of Wayland (distro `wayland`) and one where Qt does the rendering to the framebuffer itself without the help of an underlying graphical subsystem (distro `fb`).

The following table lists the files that you get after unpacking the release archive. Entries in purple are only part of a Buildroot release, entries in green are only part of a Yocto release. To avoid having a too excessive list, we use the wildcard * in some entries to refer to a whole group of similar file names that only differ in the name of the board or module.

There are some differences between 64-bit releases (ARMv8, e.g. i.MX8) and 32-bit releases (ARMv7, e.g. i.MX6).

Directory/File	Description
/	**Top directory**
Readme-buildroot-f+s.txt	Release information (Buildroot)
setup-buildroot	Script to unpack Buildroot source packages to a build directory
Readme-yocto-f+s.txt	Release information (Yocto)
setup-yocto	Script to unpack Yocto source packages to a build directory
binaries/	**Images to be used with the board directly**
nboot-<arch>-<v>.fs nboot<arch>_<v>.bin nboot<arch>_mmc_<v>.bin	NBoot bootloader image (ARMv8) NBoot bootloader image (ARMv7 nand boot) NBoot bootloader image (ARMv7 emmc boot, only for fsimx6sx and fsimx6ul)
uboot-<arch>-<v>.fs	U-Boot image (ARMv7 releases use .nb0)
Image-<arch>-<v>	Linux kernel image (ARMv7 releases use zImage-<arch>-<v>)
rootfs_min-<arch>-<v>.ubifs	Minimal root filesystem image (UBIFS format)
rootfs_min-<arch>-<v>.ext4	Minimal root filesystem image (EXT4 format)
rootfs_std-<arch>-<v>.ubifs	Standard root filesystem image (UBIFS format)
rootfs_std-<arch>-<v>.ext4	Standard root filesystem image (EXT4 format)
emmc-std-<arch>-<v>.sysimg	Full media image for eMMC including partition table and all partitions based
fus-image-std-<distro>-<arch>-<v>.ubifs	Standard root filesystem image (UBIFS format)
fus-image-std-<distro>-<arch>-<v>.ext4	Standard root filesystem image (EXT4 format)
emmc-std-<distro>-<arch>-<v>.sysimg	Full media image for eMMC, including partition table and all partitions

Directory/File	Description
`install-<arch>-<v>.scr`	Install script for U-Boot (mkimage format)
`*-<v>.dtb`	Device trees, one file for each supported board.
`sdcard/`	**Files to copy to SD card**
`nboot-<arch>.fs` `nboot-<arch>.bin`	NBoot loader, name as expected by NBoot (ARMv8) NBoot loader, name as expected by NBoot (ARMv7)
`uboot-<arch>.fs`	U-Boot, name as expected by NBoot (ARMv7 releases use ubotmx6.nb0 or similar here)
`Image-<arch>`	Linux kernel, name as expected by install script (ARMv7 releases use `zImage-<arch>` here)
`rootfs-<arch>.ubifs`	Rootfs for NAND, name as expected by install script
`emmc-<arch>.sysimg`	Full media image for eMMC, name as expected by install script
`install.scr`	Install script for U-Boot, name as expected by U-Boot's update and install system
`*.dtb`	Device trees, one file for each board; names as expected by install script, i.e. without version number
`sources/ (on github in new releases)`	**Source packages**
`u-boot-2018.03-<arch>-<v>.tar.bz2`	U-Boot source
`linux-5.4.70-<arch>-<v>.tar.bz2`	Linux kernel source
`Buildroot-2021.02-<arch>-<v>.tar.bz2`	Buildroot based rootfs source code
`yocto-3.0-fus-<v>.tar.bz2`	F&S layer for Yocto
`examples-fus-<v>.tar.bz2`	Examples source code
`install.txt`	Source code of the install script
`toolchain/`	**Cross-compilation toolchain**
`fs-toolchain-8.3-armv8ahf.tar.bz2`	F&S toolchain to use with `<arch>` (ARMv7 releases have `fs-toolchain-8.3-armv7ahf.tar.bz2`)
`mkimage`	Program needed for U-Boot script images
`dl/`	**Additional packages for Buildroot/Yocto**

Directory/File	Description
*	If there are additional packages required for Buildroot or Yocto, they will appear here
doc/	**Documentation**
<arch>-FirstSteps_eng.pdf	First Steps document
LinuxOnFSBoards_eng.pdf	The Linux environment on F&S boards and modules (this text here)
spdx/	**SBOM in SPDX format**

Table 2: Content of the created release directory

Remark

The files in subdirectory `sdcard` are actually only symbolic links to the according files in the `binaries` directory. However if you copy the content to an actual SD card, the referenced files will be copied and you get a completely normal SD card content.

If you have problems unpacking the `sdcard` directory, for example on a Windows based system that does not know about symbolic links, you can either copy the original files and rename them accordingly. Or you can unpack the separate sdcard archive, that just contains this directory.

3.7 Boot Sequence

Figure 14 shows the typical boot sequence of the board.

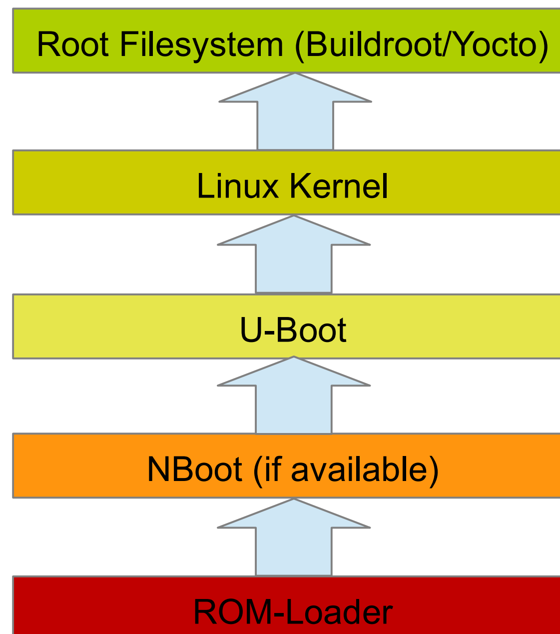


Figure 14: Boot sequence

Let us go through each step one by one, starting at the bottom.

1. The ROM-Loader is code that is located in a ROM inside of the chip. It is usually capable of loading some piece of code from a boot device (NAND flash, SD card, eMMC, or similar). However at this point of time the dynamic RAM (DRAM) is not yet initialized. The chip does not even know if there is any DRAM attached at all. So it can only load this code to some rather small internal RAM (IRAM). A regular U-Boot bootloader with more than 500KB will not fit in there. This means we need a smaller step-stone loader first, which is called NBoot on F&S systems. This step-stone loader is loaded to IRAM and then executed
2. NBoot (short for Native-Boot) is a rather small step-stone bootloader. It detects and activates the DRAM and then loads the main bootloader U-Boot.

   Some boards like the PicoCoreMX7ULP do not have the NBoot step. On these boards, U-Boot is directly loaded by the ROM-Loader. Of course all NBoot features are missing then.
3. U-Boot is the main bootloader. It is used to load and execute the Linux kernel and the device tree. It is also used to install the Linux images and to configure the boot procedure.
4. The Linux kernel provides the Linux operating system including the device drivers. It will activate the peripherals and finally mounts the root filesystem.

5. The root filesystem contains the code for the `init` process and all the Userland software. The `init` process will finalize the boot process, for example it will start the background services, the GUI and probably the main application.

In the next chapters we will have a more detailed look at each of these elements.

3.8 Mass Storage Devices

Most F&S boards do not provide SATA or PCIe ports. So using a Hard Disc Drive (HDD) or Solid State Drive (SSD) is not possible. This means the system software, i.e. boot loaders, Linux kernel image, device trees, and the root filesystem, must be stored elsewhere.

3.8.1 NAND

The most common storage device on F&S boards is the NAND flash memory. A NAND flash is subdivided into pages and blocks. A page is the smallest unit that can be read and written, a block is the smallest unit that can be erased. Data can be read arbitrarily often, but before data can be re-written, it must be erased. Typical sizes are 2 KB for a page and 64 pages or 128 KB for a block (see Figure 15).

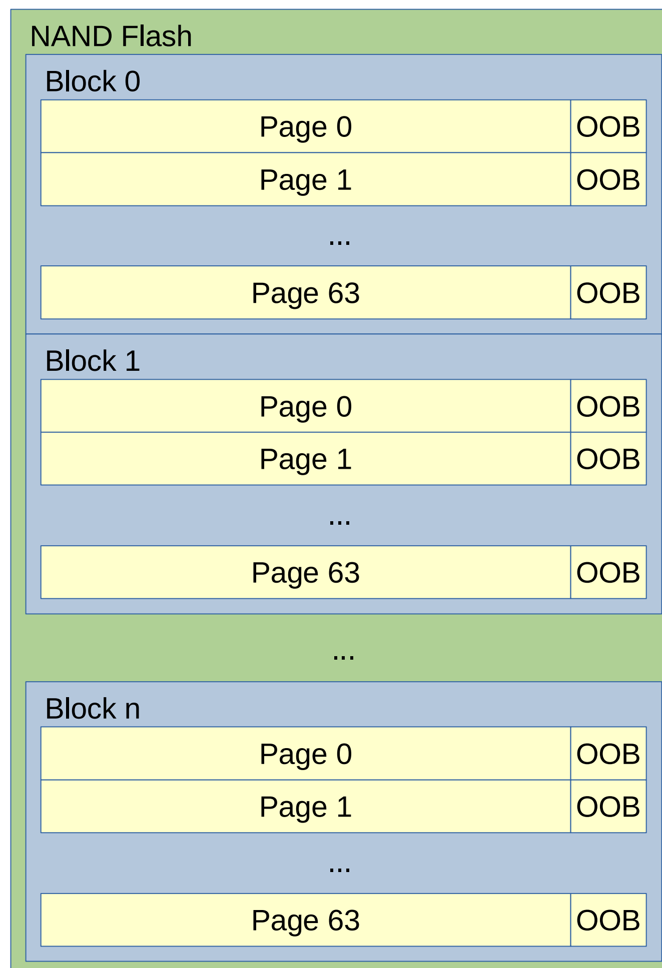


Figure 15: NAND flash architecture

NAND is cheap because it does not guarantee that all bits actually work perfectly. A certain amount of bit errors is accepted, and the user is supposed to use Error Correction Codes (ECC) to detect any bit flips in the data. When data is written to a page, the ECC sum is computed and stored with the regular data. For this purpose, the page has additional space, the so-called spare area or Out-Of-Band (OOB) area, which is at least 64 Bytes for a 2 KB page. When the data is read back later, the ECC is computed again and compared to the stored ECC of the spare area. If the sums are the same, then the data contained no error and everything was correct. But if the sums differ, then one or more bits have flipped. The ECC algorithm was designed in a way, that the comparison of the sums can immediately tell which bits are affected. So we can simply toggle back these bits and then we have the correct data again.

The complexity of the ECC sum decides how many bit errors can be detected and corrected. A simple sum can for example only detect and correct one single bit error. A more complex sum can detect and fix up to 20, 30 or even more bit errors. But of course then the ECC sum is bigger and needs more space to store. So the size of the spare area often limits the number of bit errors that can be corrected. F&S uses a setting that allows to detect and correct at least 16 bit errors in a page of 2 KB. As long as the number of errors does not exceed this limit, the data can be restored and the NAND flash is working correctly.

F&S always uses Single Level Cell (SLC) NAND flash, where each memory cell stores exactly one bit of data. So there are only two states in the cell, a low and a high voltage meaning 0 and 1. This is a very safe way of storing data, but such NAND flash is more expensive compared to MLC, TLC or even QLC NAND flash, that stores two, three or even four bits of data in one single memory cell. MLC uses four different voltages to indicate the four different possibilities 00, 01, 10 and 11 of two bits. Accordingly TLC uses 8 different voltages for three bits and QLC even 16 different voltages for four bits (see Figure 16).

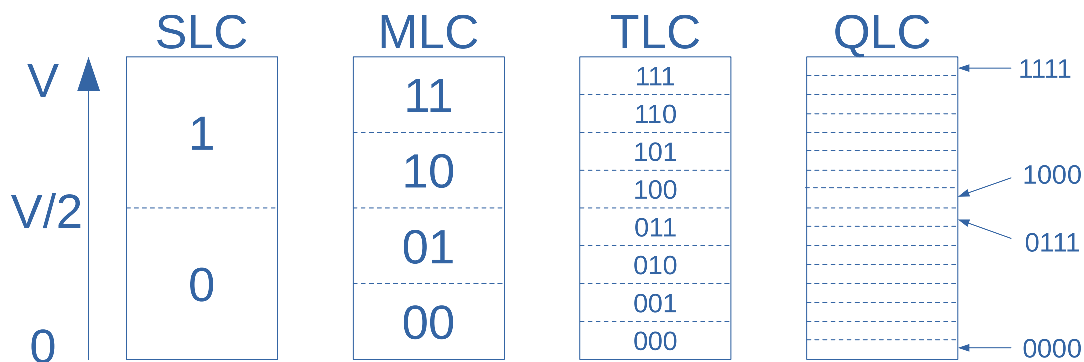


Figure 16: Bit states in SLC, MLC, TLC, QLC NAND flash cells

The more bits are stored in one memory cell, the more likely it is that these bits are corrupted. For example a small change in the charge of a QLC cell may be interpreted as 0111 instead of 1000, which will introduce four bit errors in one go. The same change in the charge of an SLC cell is so small that the two states 0 and 1 can still be correctly distinguished. So SLC is significantly more safe and will introduce less bit errors over time. This means the NAND flash memory on F&S boards is a very reliable flash memory.

Changes in the charging state of a NAND cell can occur when writing to neighbouring cells or even by repeatedly reading the data of the cell itself. Each read access minimally discharges the cell, so that after many million accesses, bits may actually flip. The whole process is ac-

celerated by heat, so if the flash memory is used in a very warm or even hot environment, the data retention time decreases.

However this is not a permanent damage. If the charge is renewed, then the cell will work again as before. This is why F&S has introduced a so-called *Block Refresh* procedure in the bootloaders that renews the data cells of a region if the bit error count is getting too high. F&S also uses filesystem variants that do similar actions.

NAND flash has a limited number of erase cycles. Depending on the NAND flash type, this may be in the range of several tens of thousands (SLC) over several thousands (MLC) to only a few hundreds (TLC, QLC). With an increasing number of erase cycles, some bits may stop working. Which means over time, the number of bit errors within some pages will rise. And some day, the ECC algorithm that is used will not be capable of correcting these errors any more. In this case, this page is irretrievably damaged. In most cases the whole block is worthless then. It can not be used without risking damage to the data. Such a block has to be marked as bad.

In fact it is possible that right from the beginning there are some blocks that have an intolerable high error rate. Then the manufacturer already has marked these blocks as bad. This is not a sign of a damaged device, it just reduces the available space slightly. The software will handle this by skipping bad blocks when installing. Filesystems like UBIFS also keep a certain amount of blocks as reserve and will use one of the reserve blocks if a regular block becomes bad at run-time.

Note

Having a few bad blocks in a NAND device is completely normal and no reason for a complaint. Only if the number of bad blocks is more than about 5% of the whole space, or if there is a long sequence of bad blocks in a row, this may cause software problems.

NAND flash is usually sub-divided into smaller regions, like partitions on a Hard Disc Drive. These regions are similarly called *MTD Partitions*. MTD is the short form for *Memory Technology Devices,* which is a Linux term for all kinds of flash memories (NAND, NOR, QSPI, etc.).

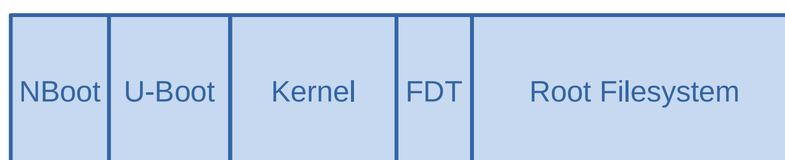


Figure 17: Common MTD partition layout in NAND flash

The rough structure of the MTD partition layout on F&S boards is shown in Figure 17. There is a partition for the bootloaders NBoot and U-Boot, a partition for the Linux kernel, one for the device tree and one for the main root filesystem. Further MTD partitions can be added arbitrarily, for example for application code, application data, image or video data, log file storage, etc. In fact the real layout has some more partitions, we will handle this later in the chapter to U-Boot.

3.8.2 eMMC

Some boards are equipped with additional eMMC memory and some new boards use eMMC instead of NAND for booting. eMMC works similar to an SD card, but has additional features. For example it can use up to eight data lines (instead of four on an SD card), can use double data rate, and the data region can be subdivided into hardware partitions.

eMMC also consists of NAND flash memory, but it also contains a built-in controller chip that takes care of all the internal NAND flash handling. You don't have to worry about NAND blocks, pages, OOB or ECC. So from the user point of view, you can use an eMMC memory like a Hard Disc Drive where the data space is accessed as blocks (or sectors) of 512 bytes each. But you have to keep in mind that it is still NAND memory with limited erase cycles, where ECC is done, where data has to be erased before re-writing, where bits can flip and where cell refreshing needs to be done from time to time. You just don't see this because it takes place in the background by the internal NAND flash controller.

In theory, eMMC memory can be built up from any kind of NAND flash memory. But in real life, at least MLC NAND is used, sometimes even TLC or QLC NAND. This means that eMMC provides more storage space, e.g. 4 GB or more, but also the data retention time is not as good as on an SLC NAND flash. It can be compared to SD cards or USB sticks, that use the same kind of NAND flash memory. So keep in mind that all those memories are less reliable, especially when exposed to high temperatures. And it is important to power up these devices from time to time so that the internal NAND flash controller has a chance to refresh the memory cells.

If eMMC is equipped in addition to (raw) NAND on an F&S board, then the system still boots from NAND, i.e. the bootloaders NBoot and U-Boot still remain in NAND flash. But all other images like kernel, device tree and root filesystem can be stored in either eMMC or NAND flash, as you like. Figure 18 shows a common layout. By using a partition table like Master Boot Record (MBR) or GUID Partititon Table (GPT), the User area of the eMMC memory is divided into (software) partitions for System (Linux kernel and device trees), roof filesystem and other user data.

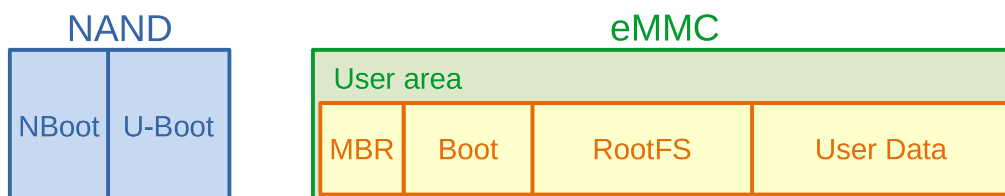


Figure 18: Common F&S layout with NAND & eMMC

If eMMC is the only flash memory on a board, then the system will also boot from eMMC. In this case we take advantage of the fact that the data region of an eMMC can be subdivided into hardware partitions. Each hardware partition behaves like a separate storage device with its own size and data that begins at sector 0.

By default, an eMMC device has already four hardware partitions: a user partition `User`, two boot partitions `Boot1` and `Boot2`, and a partition for security information called `RPMB` (Replay Protected Memory Block). But you can add up to four Generic Partitions `GP1` to `GP4`. The space for them is taken from the User Space. This results in up to eight partitions.

Partition ID	Name	Meaning
0	`User`	User Space
1	`Boot1`	Boot information (e.g. bootloader)
2	`Boot2`	Second set of boot information, either backup or alternative boot sequence
3	`RPMB`	Security information (Replay Protected Memory Block)
4	`GP1`	Generic Partition 1
5	`GP2`	Generic Partition 2
6	`GP3`	Generic Partition 3
7	`GP4`	Generic Partition 4

Table 3: Hardware partitions on eMMC memories

Please note that changing the hardware partition layout is a write-once option for an eMMC. It can only be done exactly once and will then remain in this state forever.

When changing the layout, the partitions can also be switched to the so-called *Enhanced Mode*. In this mode, the memory cells are used in a different way, so that data is stored more reliably, but on the cost of less available memory. In fact on most eMMCs this is implemented by the so-called Pseudo-SLC mode. Instead of the regular MLC variant, where two bits are stored in each cell, the cell only stores one bit in Enhanced Mode, like on a SLC cell. Of course all regions that are switched to Enhanced Mode are only half the size then. For example a 4 GB eMMC will only have 2 GB of available space after switching the whole data area to Enhanced mode. The partitions `Boot1`, `Boot2` and `RPMB` are in Enhanced Mode by default to make the boot process more reliable. So there is no need to change this.

Of course you can still install a software partition table on top of each hardware partition, further subdividing it into software partitions. `Boot1`, `Boot2` and `RPMB` are rather small, so it does not make much sense there, but it is very useful for the `User` partition.

Figure 19 shows the general layout for F&S Boards with eMMC only. We stay with the default hardware partitions, but like above further subdivide the `User` partition into software partitions. The bootloaders NBoot and U-Boot and the U-Boot Environment are stored in the `Boot1` and `Boot2` hardware partitions now. The `System` part contains the Linux kernel and the device tree(s).

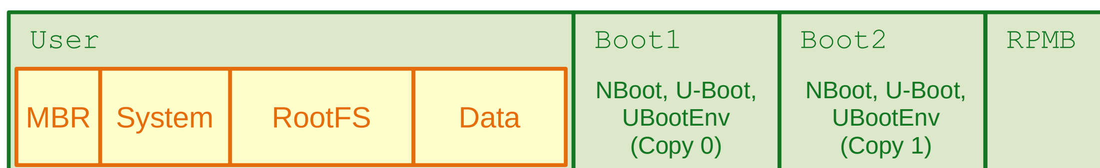


Figure 19: Default F&S layout for boards with eMMC only

The `RPMB` hardware partition is only used when working with a Trusted Execution Environment like opTEE.

Note

On PicoCoreMX7ULP there is no NBoot. So instead of NBoot, U-Boot is located in the `Boot1` hardware partition and is booted directly.

On fsimx6ul and fsimx6sx architectures, support for eMMC only modules like PicoCoreMX6MX and PicoCoreMX6UL was added in NBoot version 43.

3.8.3 SD Card

SD-Cards are rather similar to eMMC memory. But of course they are removable devices. They are available in different speed grades. Default Speed and High Speed use regular 3.3 V power supply. Beginning with UHS-I, the voltage needs to be reduced to 1.8 V, so these speeds up to 200 Mbps can only be used on boards that support voltage switching. However UHS-II cards for still higher speeds need a completely different slot design with additional differential signals. These signals are not available on the SD card slots on F&S boards, which means these cards can only be used in compatibility mode with the standard set of signals and do not profit from their higher speed.

Please note that the speed is not coupled with the memory size of the card (SDHC, SDXC). A regular SD card is not automatically restricted to regular speed and an SDHC High Capacity card does not automatically support High Speed. There may also be regular SD cards that can handle High Speed and High Capacity cards that can only handle regular speed.

SD cards can be used on F&S boards to store data on them, like eMMC. Preparing the data is also rather similar and you can use regular partition tables for software partitions, similar to Figure 18. But there are no hardware partitions, there is only one large region of data, like the `User` partition on eMMC. And the cards do not support Enhanced Mode. Which means the data on an SD card is not stored as reliably as on eMMC in Enhanced Mode or as on SLC NAND.

Important

The slot concept of SD cards only works properly if the card is removed from the slot from time to time. Then the contact springs of the slot and the contact pads of the card will grind clean again. If this does not happen and the card remains in the slot for a long time, the contacts will corrode and sooner or later communication problems with the card will arise, making the whole system unstable. This is extremely annoying if it happens after a few years and you have to exchange the card in the field. Therefore, F&S does not recommend installing the system software (kernel, device tree, root filesystem) on an SD card. Instead you should use the soldered on-board NAND or eMMC.

3.8.4 USB Media

F&S boards do support USB mass storage devices. Whether these devices are based on real rotating discs (HDD) or on NAND flash media (SSD or USB sticks/pen drives) does not matter. They are all handled in the same way. For NAND media, the same restrictions apply as before. For example you should activate them from time to time to give the internal NAND

flash controller a chance to refresh the data cells. Rotating discs on the other hand are susceptible to impacts, for example when used in portable devices.

USB devices are well-suited to provide large amounts of data like videos, pictures or other graphical data. Or you can use them for installing the system software in production or for updating the software in the field. There is quite some support for this in U-Boot. There are no hardware partitions on USB devices, only software partitions. A partitioning scheme when using USB for the system software would look similar to Figure 18, but again F&S does not recommend this for similar reasons as on SD cards.

4 Working With NBoot

NBoot, which is short for “Native Boot”, is a small first-level bootloader. It is loaded by the native loading mechanism of the ROM-Loader of the SoC and is running before the main bootloader, typically U-Boot (see Figure 20). NBoot is always pre-installed on every board that is shipped from F&S, like a BIOS on an x86 platform. In most cases, the user does not have to worry about it and unless in some rare cases, it does not need to be changed or updated in any way.

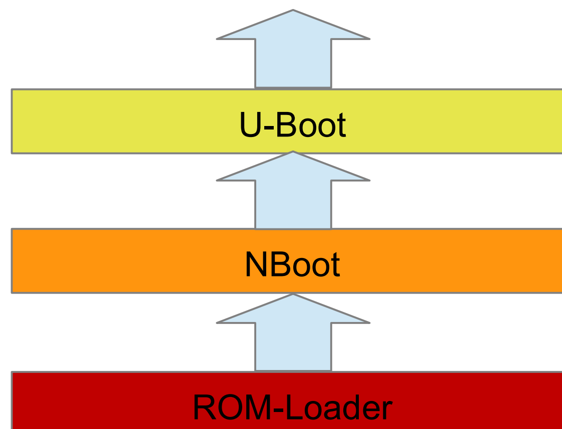


Figure 20: NBoot boot stage

At this early stage in the boot process, no Dynamic RAM is available yet, which is why the ROM- Loader has to load this code to some internal On-Chip RAM. This On-Chip RAM is typically limited in size and would be too small for the whole U-Boot, which explains why this extra step is necessary.

The first task of NBoot is to activate DRAM and all other peripherals that are needed to load U-Boot. The second task is to actually load U-Boot itself and check its integrity, for example by validating the signature in case of Secure Boot. On newer architectures, also additional files like board-configurations, DRAM timings and training data, firmwares for companion chips, the ARM Trusted Firmware (ATF) and the Trusted Execution Environment are loaded in this stage. Of course all these images need to be checked for validity, too. And finally, NBoot starts U-Boot.

So NBoot is actually getting more and more complex. It is handling all the low-level stuff where the user is typically not interested in, it serves as a Secure Boot hub and it also helps in the production process for the first time installation, for example by providing functions to blow one-time configuration fuses.

Note

The PicoCoreMX7ULP has no NBoot. To be able to use special concepts like low-power-boot and parallel boot of Cortex-A and Cortex-M cores, we have decided to keep the boot process as similar as possible to the way NXP does it. So if you are using this board, you can skip this whole chapter.

As already said, NBoot activates the Dynamic RAM (DDR) and NAND or eMMC flash on the board. Because these memory chips are often only available for a short period of time before going end-of-life, F&S has validated a whole variety of these chips and even new RAM or flash types are added from time to time. This may require modified activation sequences and therefore a new NBoot version. The advantage of hiding this initialization in NBoot is that no other software component needs to be changed in such cases.

It is rather common to never touch NBoot and leave it at the version that was shipped with the board. This NBoot can definitely handle all hardware aspects of this board. However updating NBoot does not do any harm. Every new NBoot should be fully compatible to the previous versions.

> **Please note**
>
> Never ever do a *downgrade* of NBoot. If a board is equipped with a new version of NBoot, then this might be needed by new RAM or flash components that are mounted on this board. If you downgrade NBoot, the older NBoot version may not be capable of initializing these components and the board may fail to boot. Then you might have to send in the board for repair.

In the past on our 32-bit architectures, NBoot was a proprietary software developed by F&S. But starting with our 64-bit boards, NBoot was converted to be based on U-Boot's SPL. As an effect, the usage of NBoot currently differs between ARM32 and ARM64. We are working on porting more and more stuff from the old version to the new version, but this is work in progress and is not fully completed yet.

In the following sub-chapters, we will discuss the usage of NBoot on ARM32 and ARM64.

4.1 NBoot on ARM32 (Vybrid, i.MX6)

The F&S boards and modules based on 32-bit CPUs are also available with Windows Embedded Compact. The same NBoot is used for Linux and Windows Embedded Compact and only after NBoot, the boot process splits up. Windows uses a main bootloader called EBoot and then starts the Windows operating system, while Linux uses U-Boot as main bootloader and then starts the Linux operating system.

NBoot provides a common way to install the main bootloader on the board, no matter what architecture the board has. As long as NBoot is on the system, you do not need any architecture specific hardware or software tools to install the Linux system. You can always download the main bootloader with a serial cable or a USB cable. When erasing the memory, NBoot is supposed to stay on the board.

4.1.1 Entering NBoot

NBoot usually works completely invisible. You will not see a single byte of output from it unless there is some error, for example if there is no valid main bootloader installed. Or of course if you want to stop deliberately in NBoot. That is what we want to do now.

Open the serial connection in your terminal program. Then press and hold key *s* (lower-case S). While holding this key, switch on power of the board (or press the reset button). This should bring you to the NBoot menu. You will see something like this (output is taken from efusA9, the real messages may vary slightly depending on the software version):

```
F&S Nand Loader VN38 built Jul 15 2019 11:20:16

efusA9 Rev. 1.20

...

Please select action

'd' -> Serial download of bootloader
'E' -> Erase flash
'B' -> Show bad blocks

Use NetDCUUsbLoader for USB download
```

Listing 1: NBoot menu

This means you have three choices now. Show a list of bad blocks by pressing *B* (upper-case b), erase everything but NBoot itself from the flash memory by pressing *E* (upper-case e) or download a bootloader file from the PC. In the latter case you can do this either via the serial cable by pressing *d* (lower-case D) or via a USB cable with the help of a Windows Tool from F&S called `NetDCUUSBLoader`. The following sections will show these options in more detail.

Please note that the menu depends on the context. For example after you have downloaded a bootloader image, the menu shows additional entries.

4.1.2 Show Bad Blocks

The NAND flash memory on the board is used as a kind of hard disc replacement, i.e. at least NBoot and U-Boot are stored there. In the default configurations, also the Linux kernel, device tree and the root filesystem is stored there, but it is also possible to locate those somewhere else.

As shown in chapter 3.8.1, NAND flash memory is rather inexpensive because it does not guarantee that really every bit works perfectly. It is simply assumed that some bits or even whole blocks may be damaged. While bits can be corrected by using ECC (Error Correction Codes) and do not matter much, bad blocks can not be used at all and thus reduce the amount of available memory. So it might be of interest to know how many bad blocks there are.

NBoot command *B* (upper-case b) shows the list of blocks that are marked bad. So typing character *B* in the terminal program will give something like this:

```
Bad blocks: None
```

Of course if there are any bad blocks, you will see the block numbers here.

4.1.3 Erase Flash

NBoot command *E* (upper-case e) erases the whole flash memory. This includes all Linux images (kernel, device tree, root filesystem, any own partitions), U-Boot and the U-Boot environment. So only do this if you really know that you do not lose any important data. Of course NBoot itself will be preserved and remains in NAND flash (see Figure 21).

NBoot | preserved
U-Boot
Linux-Kernel | erased
Root FIlesystem

Figure 21: NAND flash erase command

While erasing, the progress is shown by an increasing number of dots, one dot for each erased block.

```
Erasing.........................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
.........................................Done
```

> **Note**
>
> If the whole NAND flash is erased in this way, also the MAC addresses for network devices are lost. They are stored in the U-Boot environment and this environment is erased, too. So please do not forget to create appropriate environment variables again after U-Boot is re-installed. Please refer to the U-Boot chapter for this.

4.1.4 Serial Download

NBoot command *d* (lower-case D) will start the serial download. You can download any bootloader image, U-Boot for Linux, EBoot for Windows Embedded Compact and even a new version of NBoot itself. After having pressed the key, you have 60 seconds to start the down-

load in your terminal program. If it takes longer, the download command times out and the menu is shown again.

> **Note**
>
> If you forget to type *d* or if the download is started after the command timed out, every byte of the sent file is interpreted as an own NBoot command. You definitely do not want this! Just imagine that one of these commands is *E*.

Downloading needs to be done as a 1:1 binary download. This means each byte has to be sent as 8-bit value to the board, without any modifications and without adding or removing any data bytes. Normal transmission of terminal programs is often modified by adding or removing Linefeed or Carriage Return characters at line endings or by converting special keys for cursor movements (up, down, left, right, end, home) into escape sequences. Such a download mode is *not* suited.

The following steps are different, depending on the terminal program that you use.

- In `DCUTerm`, go to File → *Transmit Binary File...* and open the file that you want to download. This starts the serial download.
- In `TeraTerm`, go to *File* → *Send File...* and activate the *Binary* option checkbox. This is important! Then open the file you want to download. This starts the serial download.
- In Linux, you can copy the file directly to the serial port device. This is possible even if the terminal program does not support a 1:1 download function. For example if your serial port is `/dev/ttyS0`, just use a separate shell and enter:

  ```
  dd if=ubotmx6.nb0 of=/dev/ttyS0.
  ```

  Just note that you must not enter any characters in the terminal program while download is in progress. The character would also be sent to the serial port and would be inserted at a random position in the sequence of bytes. This would shift the remaining file content and would result in a damaged and unusable download.

During download, progress is shown by an increasing number of dots and the number of transmitted bytes from time to time.

```
................................................................ 65536 Bytes
................................................................ 131072 Bytes
................................................................ 196608 Bytes
................................................................ 262144 Bytes
................................................................ 327680 Bytes
................................................................ 393216 Bytes
................................................................ 458752 Bytes
................................................................ 524288 Bytes
Success, checksum: 0xaa7d

>>> U-Boot image loaded (524288 bytes) <<<

Please select action
'f' -> Save image to flash
'x' -> Execute image
'd' -> Serial download of bootloader
```

```
'E' -> Erase flash
'B' -> Show bad blocks

Use NetDCUUsbLoader for USB download
```

When download is complete, you see the menu again, which now has additional entries. In the example above, a U-Boot image was loaded, and the menu shows that you can either save the image to NAND flash or execute U-Boot now.

4.1.5 USB Download

There is also the option to download the bootloader image via USB, which is of course much faster. This requires a software counterpart on the PC side called `NetDCUUSBLoader` (see Figure 38). Unfortunately this software is currently only available for Windows, not for Linux. You can download this program from the `Tools` section of the *My F&S* download area on the F&S website.

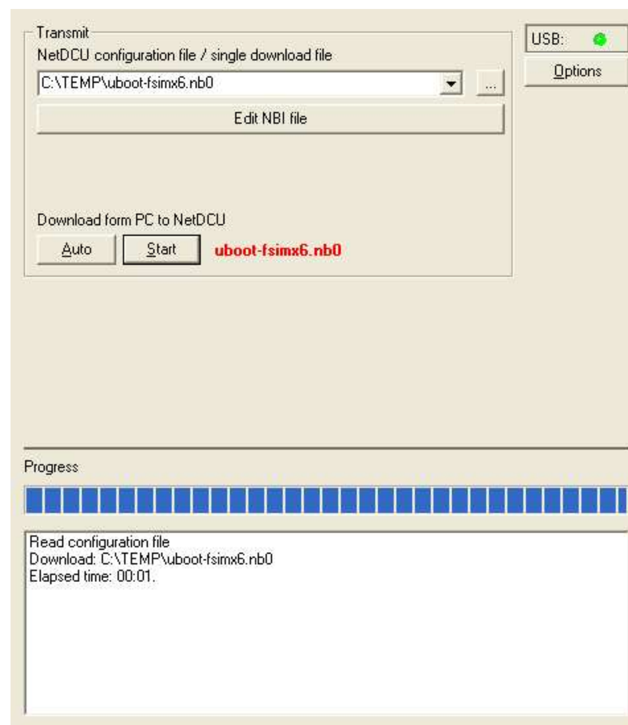


Figure 22: NetDCUUSBLoader

Start the `NetDCUUSBLoader`. Then connect the USB cable. Use the Type A connector on the PC side and the Type B Mini connector with your board. Some F&S boards have a Type B Micro connector or a Type C connector instead. The board will show:

```
Connected at high-speed
```

In `NetDCUUSBLoader`, the USB connection status should go to green, too. Then select the file that you want to send by pressing on the three dots. Finally press *Start* to trigger the transmission. As you can see, in this case the initiative is going from the `NetDCUUSBLoader` program. You do not need to press any key in NBoot.

NBoot will show

```
download complete
USB download of 524288 bytes...Success

>>> U-Boot image loaded (524288 bytes) <<<

Please select action
'f' -> Save image to flash
'x' -> Execute image
'd' -> Serial download of bootloader
'E' -> Erase flash
'B' -> Show bad blocks

Use NetDCUUsbLoader for USB download
```

Download only takes about one second. Then you see the menu again, which now has additional entries. In the example above, a U-Boot image was loaded, and the menu shows that you can either save the image to NAND flash or execute U-Boot now.

4.1.6 Save Bootloader Image

Save a loaded bootloader image by pressing *f* (lower case F). When the image is loaded to RAM, NBoot automatically detects the image type and shows the type every time the menu is shown. Therefore NBoot can automatically decide where the image has to be saved in the save command *f*. If an NBoot image is loaded, NBoot replaces its own code. This is possible because NBoot is actually running from RAM in that moment and only the NAND data is replaced. The currently running NBoot will continue to run until a downloaded image is started or the whole board is restarted.

Saving an image should show something like this. Of course the message can differ depending on the type of loaded image.

```
Saving U-Boot to NAND
Success
```

The image still remains in RAM. So it still can be started directly with command *x*.

Note

The type check of an image is not very sophisticated. If it has the correct size and a correct ID (some magic number near the beginning of the file), it is assumed to be a valid image. This can be misleading, though. For example all bootloaders for all variants of i.MX6 SoCs have the same size and same ID. So it is possible to store an image for an fsimx6ul U-Boot on a regular fsimx6 board. Of course this will not result in a working system.

This is especially dangerous if the image is an NBoot image. If you happen to download the wrong NBoot image, the board will not boot anymore afterwards and may have to be sent in for repair. Therefore be particularly careful when updating Nboot.

To save the U-Boot a valid MBR and FAT partition must be present. You can create empty partitions in NBoot with the command `w` (lower-case W). (Refer to Chapter 4.1.8)

4.1.7 Execute Bootloader

If a valid bootloader image is loaded, it can be executed by command *x* (lower-case X). If the loaded image was an NBoot image, this will immediately start the new instance of NBoot. If the image was a regular main bootloader image like U-Boot, then execution is passed on to this bootloader. No further output is visible in NBoot.

4.1.8 Create Partitions

For eMMC only devices a FAT partition and MBR must be present at the `User` partition to write U-Boot. To initialize the partitions press *w* (lower-case W). Automatically a FAT partition with the size 40 MB and an ext4 partition with the size of 800 MB will be created. You can define more partitions with partition type and size (e.g. 0x83 for ext4 and 40960 for 40 kB). The characters typed in will not show on the terminal. If you don't need any further partitions, you can simply press Enter without typing in values for type and size. When all partitions are defined, a prompt is shown to validate the writing of the partitions. This will result in overwriting any current partition table.

```
Create MBR
Creating boot partition 0 (FAT16, 40MiB)
creating partition  0 with type 0x06 and size 40 MiB
Creating rootfs partition 1 (Linux, 800MiB)
creating partition  1 with type 0x83 and size 800 MiB
enter additional data partition 2
filesystem type:filesystem size in MiB:
WARNING: writing the MBR will delete all data!

Enter 'y' to proceed, any other key to exit
writing MBR
writing FAT
```

4.1.9 Access Partitions with USB

If a valid MBR (Master Boot Record) is stored in `User` partition of eMMC, the NBoot can emulate partitions as an USB-Stick by executing the command Z (upper-case Z). The following message is shown, indicating that MSD (Mass Storage Device) is enabled.

```
enable MSD
```

If the USB port is connected, every partition on the `User` partition will be registered by the Host-PC as an USB-Stick.

4.2 NBoot on ARM64 (i.MX8)

The boot process on ARM64 is rather more complicated than on ARM32. This is shown in Figure 23.

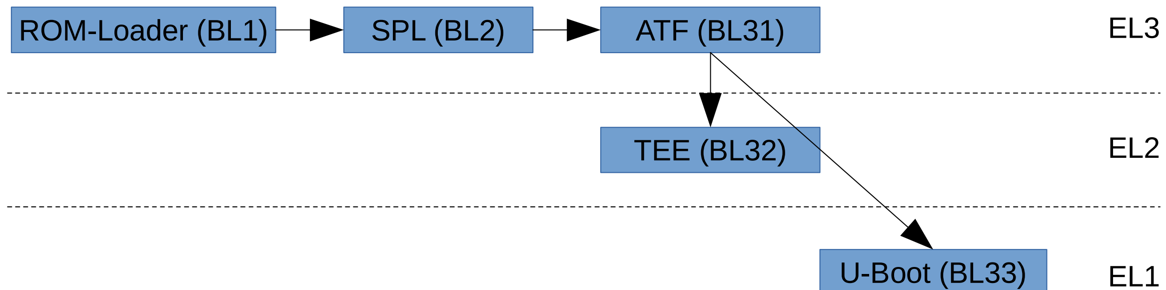


Figure 23: ARM64 (ARMv8) boot process and Execution Levels

The new boot process allows more complex systems to be set up, for example with different operating systems controlled by a Hypervisor. This requires to have limits on what each operating system is allowed to do. For example if one operating system wants to change the clock speed of a CPU Core, this may contradict with the idea of what the other operating system wants to do. So if any of these two operating systems would be allowed to do this change itself, it would also change the behaviour of the other operating system, too. This must not be allowed. So we need a moderator who keeps track of what both operating systems want and then in the end actually performs the desired task like changing the clock speed. This is the point where the ARM Trusted Firmware (ATF) and the Trusted Execution Environment (TEE) come in. By running in high Execution Levels (EL2 and EL3), only these instances have access to all the hardware that is shared, like CPU Cores, the operating systems themselves are running on a lower Execution Level (EL1) with only restricted access to that hardware.

And there are even more software parts involved. This results in the following list of images, that is not comprehensive and always depends on the type of SoC.

- Second Program Loader (SPL)
- ARM Trusted Firmware (ATF)
- Trusted Execution Environment (TEE)
- DRAM timing and training data
- Firwmares for companion chips (SCU, SECO, HDMI)
- Board-specific configuration information (BOARD-CFG, BOARD-ID)
- U-Boot

Most of this stuff is very low-level and the typical user is not interested in this at all. The board simply should start, no matter how this works in full detail. This is the reason why we also will not go into more detail here. Apart from U-Boot, we have put all the other stuff in our NBoot and this is again pre-installed on each board that is shipped. So as a user, you do not have to care about all this low-level stuff. You just need to install your Linux system and, if necessary, your own U-Boot. Nothing else.

4.2.1 Handling NBoot

NBoot on ARM64 is based on U-Boot's SPL. So when the system boots, NBoot will show a short SPL greeting message before quietly loading all the software parts and starting U-Boot. So the next message is already U-Boot showing up.

Different to the ARM32 variant, NBoot on ARM64 does not support a menu yet. So it is not possible to stop in NBoot and execute all the interactive commands that are available on ARM32. On ARM64, NBoot definitely needs an U-Boot to perform a successful boot. If U-Boot is lost, NBoot alone is not capable of installing anything again. In that case, it requires additional software tools like the USB installation program `uuu` to restore the board.

So actually there is not much to tell here, the whole installation process is explained later in the next chapters as part of U-Boot.

5 Working With U-Boot

The bootloader U-Boot actually started as *PPC-Boot*, which was a bootloader for PowerPC based SoCs. At some point in time, support for other CPU architectures like ARM was added, and the bootloader was renamed to *Universal Boot*, or U-Boot for short. “U-Boot” is also the German word for a submarine. So this is a nice wordplay because a submarine remains invisible under the surface and only comes up to load new supplies, like a bootloader remains invisible in normal life and only comes to view when new images are to be loaded.

U-Boot is used for downloading and storing all necessary binary images (kernel, root filesystem, etc.) on the board, handles the basic system configuration, and is responsible for executing the Linux Operating System.

5.1 Entering U-Boot

To work with U-Boot, you also need the serial connection to the board and the terminal program. See Chapter 3.2 on page 14 for instructions how to do this. When the board is switched on, the first output that you see on the serial connection is from U-Boot.

```
U-Boot 2018.03 (Aug 16 2019 - 17:12:21 +0200) for F&S

CPU:   Freescale i.MX6DL rev1.3, 996 MHz (running at 792 MHz)
CPU:   Extended Commercial temperature grade (-20C to 105C)
Reset: POR
Board: efusA9 Rev 1.20 (LAN, eMMC, 2x DRAM)
I2C:   ready
DRAM:  1 GiB
NAND:  256 MiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1, FSL_SDHC: 2
Loading Environment from NAND... OK
In:    serial
Out:   serial
Err:   serial
Net:   FEC [PRIME]

Hit any key to stop autoboot:  3
```

The number at the end of the last line slowly counts down. To enter U-Boot, you have to send an arbitrary character on the serial line to the board before the count reaches zero. This brings you to the command line interface of U-Boot.

If you do not press a character, the system will try to boot Linux. If this fails, you will also be brought to the U-Boot command line. Otherwise the Linux system will come up.

Remark

In the following sections we will show commands to be typed as a framed text with yellow background. For example:

```
help
```

This means you have to type `help`, followed by the *Return* or *Enter* key.

Sometimes we also combine input and output in one yellow region. In this case, the command that has to be typed is in bold and the output of the board is shown in regular font.

```
efusA9 # version
U-Boot 2018.03 (Aug 08 2019 - 17:17:06 +0200) for F&S
arm-linux-gcc (for F+S boards and modules) 7.4.0
GNU ld (for F+S boards and modules) 2.31.1
```

This means the board shows the command prompt, consisting of the board name and a hash mark. The command you have to type is `version` which is why this part is in bold. And finally the board outputs the remaining lines as result of the command.

To avoid the impression that a command is only available on a specific board, we actually drop the board name and only take the hash mark as command prompt. So in this document, this would in fact show as

```
# version
U-Boot 2018.03 (Aug 08 2019 - 17:17:06 +0200) for F&S
arm-linux-gcc (for F+S boards and modules) 7.4.0
GNU ld (for F+S boards and modules) 2.31.1
```

As a simple rule when entering a command, skip the hash mark `#` if the command is preceded by one.

5.2 Commands

U-Boot has a command line interface. Commands consist of a keyword and parameters. The keyword tells what to do and the parameters define the behaviour of the action. A parameter can be a memory address where to store some data, the name of a file that should be loaded from an SD card, an IP address for a network access, an option to modify the data output, and so on.

The following example tells U-Boot to transfer a file named `rootfs.ubifs` via TFTP protocol from a TFTP server with IP address 10.0.0.5 to the board and store the content in RAM at address `0x11000000`.

```
tftpboot 0x11000000 10.0.0.5:rootfs.ubifs
```

Commands may be abbreviated, as long as they are unambiguous. So the above command `tftpboot` can also be written as `tftp`. And addresses are automatically interpreted as hex values, so the `0x` can also be omitted. So the following command does the same:

```
tftp 11000000 10.0.0.5:rootfs.ubifs
```

U-Boot also has a default load address. So if the address in RAM does not matter, it can also be omitted and the file is loaded to the default load address. Usually the IP address of the TFTP server is also well known (by setting an environment variable), so the above command can be further simplified to the following command which is much easier to type:

```
tftp rootfs.ubifs
```

The following Table 4 shows an incomprehensive list of available U-Boot commands.

Command	Function
`setenv`	Set environment variables (command line, network parameters, ...)
`saveenv`	Store environment to persistent storage (NAND)
`mtdparts`	Access MTP partitions (add, delete, show, ...)
`nand`	Access NAND flash (read, write, info, bad blocks, ...)
`ubi`	Access UBI volumes (create, remove, read, write, ...)
`usb`	Access USB storage devices (select, show, read, write, ...)
`mmc`	Access MMC/SD card devices (select, show, read, write, …)
`tftpboot/nfsboot`	Load from TFTP/NFS server (Ethernet)
`ls`	List files on media with a filesystem (FAT, EXT2/4, UBIFS)
`load`	Load a file from a filesystem (FAT, EXT2/4, UBIFS)
`update`	Search for update/install/recover script and execute
`bdinfo`	Show board information
`clocks`	Show information on clock rates
`md/cmp/cp/mw/mm`	Show/compare/copy/write/modify memory
`source`	Execute a script file
`fdt`	Manipulate a device tree
`run`	Run commands from an environment variable
`boot/bootm`	Start the default boot process/boot an image in RAM
`help`	Show list of available commands or help for a single command

Table 4: Incomprehensive list of U-Boot commands

In fact the command line interface is a real small shell called *hush* that supports a syntax similar to the Bourne Shell `sh`. For example several commands can be written as a sequence, separated by semicolons. It also supports conditional execution (`if … then .. else … fi`, `||`, `&&`) and loop constructs (`for … in … do … done`, `while … do … done`, `until … do … done`). It can even execute script files, for example to perform complex installation or update procedures.

Of course explaining all these shell-like features is beyond the scope of this document. More information about U-Boot can be found at http://www.denx.de/wiki/DULG/Manual or in the `doc` directory in the source code package of U-Boot. We will concentrate on some basic features of U-Boot and on special features only available on F&S boards.

5.3 Command History

U-Boot remembers the last few commands that you typed. So if you want to repeat a previous command, simply use the keys *Arrow Up* ↑ and *Arrow Down* ↓ to bring back one of the previous command lines. Then you can modify the command and execute it again by pressing *Return*. This may speed up your work considerably.

Unfortunately this command history is not saved permanently and the list is also not very long, so you only can select from the last few commands of the current session.

> **Note**
>
> To use the command history, the terminal program needs to support ANSI escape sequences for the cursor keys. This is true for `PuTTY`, `TeraTerm` and `minicom`, but not for `DCUTerm`.

5.4 MTD Partitions

If the board is equipped with NAND flash, then the NAND flash is used for many different purposes. The binaries of NBoot, U-Boot, the Linux Kernel and the root filesystem are stored there at specific positions. For example on the fsimx6 architecture, the Kernel image is stored at offset 0x240000 and can use up to 0x800000 bytes, i.e. 8 MB. When we want to read this region from NAND flash to RAM, we can use the following command:

```
nand read $loadaddr 240000 800000
```

But as it would be cumbersome to memorize all the starting offsets and sizes of all these regions, especially as these values may change with a new release, the regions have been given meaningful names. For example the kernel region has been given the name `Kernel`. So instead of typing the NAND flash offset and the size in NAND commands, you can simply use the region name. This makes working with these regions much easier. For example the above command can be simplified to

```
nand read $loadaddr Kernel
```

Actually with this definition, these regions divide the NAND flash into partitions like on a Hard Disc Drive. These regions are therefore called *MTD partitions*.

With command `mtdparts`, you can add or remove such partitions, or list the current MTD partitions. Please note that the command `mtdparts` does not actually change anything in NAND flash itself, it just refers to a list of names, offsets and sizes that it has set up in RAM. There is also no partition table in the NAND. The list is only stored in an environment variable called `mtdparts`. So you need to save the environment to make changes permanent.

In Chapter 3.8.1, we have already seen the rough MTD partition layout on F&S boards. Figure 24 shows the real layout in more detail. The sizes are not proportional, `TargetFS` is actually much bigger than all other partitions together.

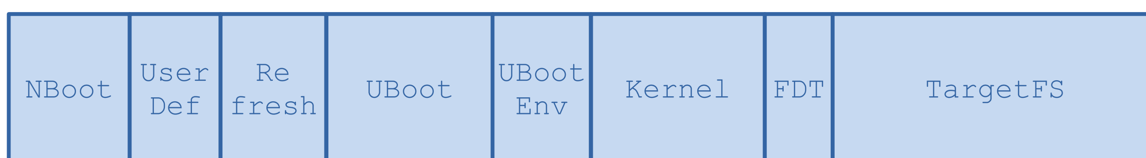


Figure 24: Detailed MTD partiton layout on F&S boards

Even if the layout is rather similar on most F&S boards, the start offsets and sizes may vary. Table 5 shows the default partition list for fsimx6 boards. But please note that these values may be adjusted in newer releases from time to time as needed. This is why you should never use offsets or sizes directly, just the partition names.

Name	Start	Size	Meaning
`NBoot`	`0x00000000`	`0x00040000`	Region where NBoot is stored
`UserDef`	`0x00040000`	`0x000c0000`	Small space for user defined configurations. Can be used for Cortex-M4 programs on boards with Asymmetric Multi-Processing
`Refresh`	`0x00100000`	`0x00040000`	Space to temporarily store blocks during the Block Refresh procedure.
`UBoot`	`0x00140000`	`0x000c0000`	Region where U-Boot is stored
`UBootEnv`	`0x00200000`	`0x00040000`	Environment of U-Boot
`Kernel`	`0x00240000`	`0x00800000`	Linux kernel image
`FDT`	`0x00a40000`	`0x001c0000`	Device Tree image (FDT=Flat Device Tree)
`TargetFS`	`0x00c00000`	`(remaining free space)`	Space for root filesystem

Table 5: NAND flash partitioning

The `TargetFS` partition occupies the remaining free space of the flash. Its size therefore depends on the size of the NAND flash that your board is equipped with. Partition `Refresh` is reserved for internal use by NBoot and U-Boot, so please do not store any data there. Partition `NBoot` is especially protected so that you do not unintentionally erase or modify the NBoot bootloader. If you try to write to this partition, it will be reported as read-only and if you try to read from it, you will only see bytes with 0xFF.

On boards equipped with a Cortex-M core for Asymmetric Multi-Processing (AMP), partition `UserDef` can be used to store a Cortex-M binary. This will be loaded and executed by NBoot at a very early stage, before even U-Boot is loaded. On all other boards, this partition is free and you can use it for your own purposes.

Partition `UBootEnv` contains the non-volatile environment variables. When you call `saveenv`, then the current environment is saved to this partition. If you erase this partition, U-Boot will start with the compiled-in default environment the next time. Please note that this will also remove the network settings, especially the MAC address(es), so you have to set them again at the next start.

5.5 UBI Concepts

As already mentioned in the NBoot chapter, NAND flash is rather inexpensive. Unfortunately this requires some effort with Error Correction Codes (ECC) and still the number of erase cycles is limited. Now imagine the MTD partition layout of Figure 25.

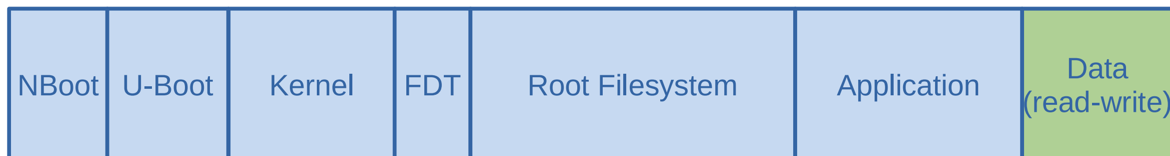


Figure 25: NAND with MTD partitions only

In addition to the bootloader and kernel partitions (which are represented slightly simplified here again), you would probably have a partition for your root filesystem, a partition for your application and a partition for your data. All but the data partition could be read-only to avoid any unnecessary erase cycles. But the data partition needs to be read-write, because you want to store data there on a regular basis.

If you only want to write small amounts of data, a small data partition is sufficient. However this also means, that only a small number of NAND blocks will be used for it. And if you write rather often, then these blocks will be erased and re-written over and over again. With the result that these blocks will be worn out some day. Your data partition will show more and more errors, then bad blocks and eventually it will fail completely. Which is a pity, because there are many many blocks in all the other partitions that are still in good shape. They only have been written once or twice when the software was installed. But you can not use them because they are in other partitions.

So from the point of data security, small partitions are desired. From the point of wear-levelling, large partitions are desired. This annoying contradiction is solved by using UBI.

UBI is a management layer on top of flash memories that deals with all the special issues of these kinds of memories, especially wear-levelling. The idea is that you group as many NAND blocks as possible and put an UBI on it. UBI will keep a list of how many times each block was erased and will prefer blocks with low erase count over blocks with high erase count when writing new data. All blocks of the UBI space will take part in wear-levelling, even blocks that are marked as read-only by the filesystem.

This means a logical block will be mapped to one physical block first, but when wear-levelling asks for, it will be relocated to a different physical block, making the previous physical block with a low erase count available for new data writes. In the end, there are no blocks with an excessive erase count, all blocks will have rather similar counts.

UBI makes sure that data is not lost when relocating blocks, even if the power goes down right in the middle of writing a block. How exactly this is done is beyond the scope of this document but some kind of atomic block replacement algorithm guarantees that the system either sees the data on the old or the new position, so that the UBI area can always be brought back to a consistent state without the need of a time-consuming filesystem check.

Note

Only data that was already in the NAND flash is not lost, because either the old or the new copy is seen at the next start and UBI makes sure to take the right copy. However when writing new data to the flash memory, Linux buffers large parts of that data in RAM first. UBI can not prevent that such unwritten data is lost in case of a power failure. You have to take care by your software design that times of inconsistent state are as short as possible.

Due to all these relocations, after a while it is no longer possible to say which logical block is located on which physical block. Only UBI knows this. From an outside view, the blocks are completely unsorted. Hence the name UBI, which stands for “Unsorted Block Image”.

UBI can subdivide its space into smaller entities called volumes. A volume is similar to an MTD partition, but the data blocks of a volume can be located anywhere within the UBI. So even with volumes, all blocks of the UBI area take part in wear-levelling. This is a big difference to the MTD scenario above.

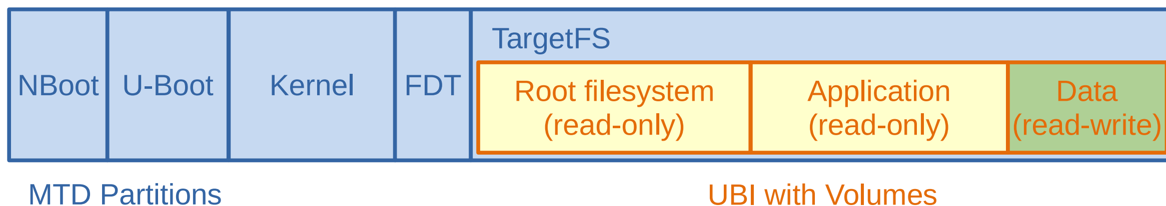


Figure 26: NAND layout with UBI volumes

This results in the common NAND layout from Figure 26. Again the bootloaders and Linux images are located in MTD partitions. But now the whole remaining part called `TargetFS` is one big UBI. And the root filesystem, application and data partitions are only UBI volumes.

As you can see, UBI is installed on top of an MTD partition. Which means we have to tell the UBI subsystem in U-Boot, what MTD partition to use. In our case this is `TargetFS`.

```
# ubi part TargetFS
ubi0: attaching mtd1
ubi0: scanning is finished
ubi0: attached mtd1 (name "mtd=7", size 244 MiB)
ubi0: PEB size: 131072 bytes (128 KiB), LEB size: 126976 bytes
ubi0: min./max. I/O unit sizes: 2048/2048, sub-page size 2048
ubi0: VID header offset: 2048 (aligned 2048), data offset: 4096
ubi0: good PEBs: 1952, bad PEBs: 0, corrupted PEBs: 0
ubi0: user volume: 1, internal volumes: 1, max. volumes count: 128
ubi0: max/mean erase counter: 7/4, WL threshold: 4096, image sequence
number: 0
ubi0: available PEBs: 0, total reserved PEBs: 1952, PEBs reserved for
bad PEB handling: 40
```

If `TargetFS` already contained an UBI, this UBI will be reused. Otherwise a new empty UBI will be created. If creation fails, try to erase the `TargetFS` partition first by using the following command. On a new device when no wear-levelling has happened, this is no problem.

```
nand erase.part TargetFS
```

> **Note**
>
> In general, erasing an MTD partition with a valid UBI on it is not a good idea. UBI relies on the fact that it keeps a list of erase counts. If you low-level erase the MTD partition, these erase counts are lost which means all subsequent wear-levelling is based on wrong data. So especially if the device has already been in operation for some time, do not erase such a partition unless really really necessary and everything else has failed.

After the partition is linked to the UBI subsystem, you can list existing volumes:

```
# ubi info layout
Volume information dump:
        vol_id          0
        reserved_pebs   1908
        alignment       1
        data_pad        0
        vol_type        3
        name_len        6
        usable_leb_size 126976
        used_ebs        1908
        used_bytes      242270208
        last_eb_bytes   126976
        corrupted       0
        upd_marker      0
        name            rootfs
Volume information dump:
        vol_id          2147479551
        reserved_pebs   2
        alignment       1
        data_pad        0
        vol_type        3
        name_len        13
        usable_leb_size 126976
        used_ebs        2
        used_bytes      253952
        last_eb_bytes   2
        corrupted       0
        upd_marker      0
        name            layout volume
```

This shows a volume called `rootfs` and a special internal `layout volume` that is used to manage all volumes. This internal volume can be compared to a partition table that holds information about the available partitions, or volumes in this case.

This setup is the default configuration on F&S boards. The whole UBI on the `TargetFS` partition is defined as one big volume `rootfs` that holds the root filesystem for Linux. We will now create a different layout with three volumes, like in the example above.

Name	Size	Meaning
`application`	`0x01400000`	Application code, read-only data (20 MiB)
`data`	`0x00A00000`	Read-write Data, log files, configurations (10 MiB)
`rootfs`	Remaining space	Root filesystem, system commands, libraries

Table 6: Different UBI volume layout

The first step is to remove the previous volume `rootfs`. Of course this will make the previous content unavailable, so be careful to back up all data first that you still need.

```
# ubi remove rootfs
Remove UBI volume rootfs (id 0)
```

Then we add the new volumes. All volumes but the last need a specific size. If the size argument is not given, the volume will use all remaining free space of the UBI, which only makes sense for the last volume, because after that there is no more space for further volumes.

The sequence of creation does not matter much, because all the data will be stored in an unsorted heap anyway. We use `rootfs` as last volume because in our example this is the volume that takes the remaining free space.

```
# ubi create application 0x01400000
Creating dynamic volume application of size 20971520
# ubi create data 0x00a00000
Creating dynamic volume data of size 10485760
# ubi create rootfs
No size specified -> Using max size (210653184)
Creating dynamic volume rootfs of size 210653184
```

An UBI volume is only the space to store some data in it. It is not a filesystem. Theoretically it would be possible to store an arbitrary filesystem image in an UBI volume. However most filesystems like FAT or EXT4 are not aware of the fact that they run on flash memory. They write data more often than necessary and do not group data in chunks that help to reduce write and erase cycles. And some of these filesystems, for example FAT, are very vulnerable if they are simply switched off. They will get into an inconsistent state and need a time-consuming check at the next start before they can be used again. This would somehow counteract the idea of UBI that is more stable in such situations.

Therefore, UBI is used either with read-only filesystems such as Squashfs, or with UBIFS, a filesystem which has been especially designed for UBI.

U-Boot is not capable of creating a filesystem image by its own. Neither for EXT2 or EXT4, nor for FAT or UBIFS. It always needs pre-made filesystem images, which then only need to be stored on the board. But if an UBIFS is available in a UBI volume, then U-Boot can access the files in it. We will see later in the high-level filesystem access how this is done.

5.6 Environment

Configuration in U-Boot is done by setting variables. The set of all variables is called environment and can be stored in flash so that it is again available at the next start. In case of NAND flash, it is stored in MTD partition `UBootEnv`. In case of eMMC, it is stored at some specific offset in the `Boot1` hardware partition.

The environment is handled by the command `env`. However most `env` commands have an alias for historical reasons, and actually most people still use these aliases nowadays. So we will also use the aliases in our examples.

5.6.1 Environment Variables

A variable is set to a value with the command

```
env set <variable> <value>
```

However the alias `setenv` is more common.

```
setenv <variable> <value>
```

The `<value>` is basically an arbitrary string. For example the following command will set variable `my_var` to the value “Hello world”

```
setenv my_var Hello world
```

The value of a variable can be shown with command

```
env print <variable>
```

Again the alias `printenv` is more common. For example

```
# printenv my_var
my_var=Hello world
```

This shows that the current value of `my_var` is the string `Hello world`. The command

```
printenv
```

without any parameters will list all variables. Well, nearly all. A variable name starting with a '.' character (dot) denotes a so-called hidden variable. Hidden variables are not shown in this list. If you want to see these variables, too, you have to use:

```
printenv -a
```

This will show all variables, visible or hidden. F&S uses the hidden feature for the boot strategy variables. There are quite a lot of them and if they were visible all the time they would overcrowd the variables list unnecessarily and make it less legible.

Removing a variable is done by providing an empty value in the `setenv` command. So

```
setenv my_var
```

will remove the variable `my_var` again that was set above.

5.6.2 Special Variables

Some variables have a special meaning for U-Boot and require a special syntax. Setting them will have an immediate effect on U-Boot. For example the variable `baudrate` holds the baud rate of the serial communication line. The value can only be a valid number for a serial baud rate like 19200, 38400 or 115200. Setting a new value with

```
setenv baudrate 19200
```

will immediately switch to the new speed of 19200 bps, so you would have to adapt the speed setting of your terminal program, too.

Some variables are also automatically set or modified when specific commands are executed. For example the set of currently active MTD partitions is held in a variable called `mdtparts`. Each time when the `mtdparts` command is used to modify the partition table, the `mtdparts` variable is automatically updated to reflect the new state. If you show the content of this variable with ...

```
# printenv mtdparts
mtdparts=mtdparts=gpmi-nand:256k(NBoot)ro,768k(UserDef),
256k(Refresh)ro,768k(UBoot)ro,256k(UBootEnv)ro,8m(Kernel)ro,
1792k(FDT)ro,-(TargetFS)
```

… you will see that it is a rather long string with a strange syntax. In fact this is the same syntax that is required for the Linux kernel command line, which makes this variable quite handy for this purpose.

Table 7 shows some basic U-Boot environment variables. You may need to adapt some of them to reflect your local environment. For example you usually have to modify the network variables to be compatible with your local network.

Variable	Meaning
`ipaddr`	The IP address of your board
`serverip`	The IP address of your TFTP and/or NFS server (usually your development PC)
`gatewayip`	The IP address of your gateway; the device in your network that knows how to access the internet (usually your router)
`netmask`	The network mask used for your network (usually 255.0.0.0 for local network 10.x.x.x or 255.255.255.0 for local network 192.168.n.x)
`ethaddr`	MAC address of your board.
`bootcmd`	Boot command; this command will be executed when typing command `boot` or when you let U-Boot boot automatically
`mtdparts`	NAND flash MTD partitions
`loadaddr`	RAM address to load any images to/from
`fdtaddr`	RAM address to load the device tree to

Variable	Meaning
bootargs	Basic kernel parameters
bootfile	Default name of the kernel
bootfdt	Default name of the device tree
bootdelay	Number of seconds to wait for a keypress before booting automatically
arch	Architecture of your board, i.e. fsimx6
filesize	The size (in hex) of the last file loaded
platform	Platform name, e.g. armstonea9, qblissa9, picomoda9, efusa9
mmcdev	The MMC port where the system boots from (typically the eMMC port)
preboot	A command that is executed before the board is booted.

Table 7: Important U-Boot settings

5.6.3 Save the Environment

If you change the content of environment variables with `setenv`, this is only valid for the current session. At the next start the previous environment would be activated again. If you also want to save the new settings for the future, then you have to use `saveenv`.

```
# saveenv
Saving Environment to NAND... Erasing NAND…
Erasing at 0x220000 -- 100% complete.
Writing to NAND... OK
OK
```

As you can see, this will store the environment in NAND flash in MTD partition `UbootEnv`.

If you have played around with the environment and want to go back to the original state, then simply type

```
env default -a
```

This will return all variables to their compiled in default state. Please note that on F&S boards, some variables will have the value `undef` now. You have to restart the board to get them set to their board-specific value again. Of course this will also remove the MAC address stored in variable ethaddr. You have to set it again to the correct value. This is handled in Chapter 5.7.

5.6.4 Using Variables in Commands

The value of a variable can also be referenced in other commands by preceding the variable name with a `$`. Then the content of the variable is substituted at this place. For example the variable `loadaddr` holds the default load address as a hex number. This address is auto-

matically used in many commands if the address parameter of the command is omitted. However the syntax of some commands does not allow omitting the address parameter. In this case you can use `$loadaddr` to refer to this default address. We have already seen this above in the command to load the `Kernel` MTD partition.

```
nand read $loadaddr Kernel
```

Here the content will be stored at the default address in RAM.

The name of a variable may additionally be enclosed with curly brackets `{ }`. This is convenient if the scope of the name would be ambiguous otherwise. For example if you have a variable called `state` and you want to print the state with the string `"_mode"` appended to the content, you have to write it as

```
echo ${state}_mode
```

because

```
echo $state_mode
```

would search for a variable named `state_mode`.

5.6.5 Running Commands in a Variable

Some commands have to be typed over and over again. For example the sequence to download a kernel and to store it to the Kernel MTD partition:

```
tftp zImage
nand erase.part Kernel
nand write $loadaddr Kernel $filesize
```

In fact it is possible to store this sequence in an environment variable and run it at any time. We will use a variable with the simple one-character name `r`:

```
setenv r tftp zImage\; nand erase.part Kernel\; nand write \$loadaddr Kernel \$filesize
```

Here you have to note two things. First we "quote" the semicolons with a backslash `\`. Otherwise the semicolon would denote the end of the `setenv` command and the remaining part of the command would be executed immediately. By quoting, we remove the special meaning of the semicolon as end-of-command marker and a single semicolon is inserted in the variable content.

And secondly we have also quoted the `$` from the variable names `loadaddr` and `filesize`. Again, if we do not do this, the current content of the variable would be inserted into the command right now. But we want the content to be inserted later, when we actually execute the command sequence. So here again a quoting backslash.

Now we can execute this sequence of commands at any time with

```
run r
```

This even works recursively. So you can run a variable that in turn runs the contents of another variable. F&S uses this concept for the boot strategies. And U-Boot is implicitly using this, too. For example when booting the board, the content of variable `bootcmd` is automatically run.

5.7 Network Configuration

When working with a network device like Ethernet, we need to configure a few things. First of all the network device needs a MAC address. MAC addresses are unique around the whole world, each network device needs its own address. These addresses are administered by IEEE. MAC addresses may either be built in a network device, or they may have to be provided externally. F&S has obtained a pool of MAC addresses from IEEE and assigns one or more addresses to each board. If a board has one network device, it gets one MAC address, if it has two network devices, it gets two (subsequent) MAC addresses, and so on.

Hardware MAC addresses are independent of the network protocol that is used on the device. However in most cases we will use the Internet Protocol. This protocol uses an own layer of addresses, the so-called IP addresses. Globally visible devices also need a unique IP address, but devices that are only part of a local network can be assigned a dynamic address, usually by querying a DHCP server. So a device may have one IP address at one day and a different IP address at another day.

In practical life, nobody uses IP addresses directly. Instead internet names (URLs) are used. The system software will then query a name server to resolve the name, i.e. convert internet names to an IP addresses and vice versa. For example at the time of writing, the IP address of the F&S web server www.fs-net.de was 94.130.76.119.

U-Boot is only capable of IPv4 protocol, not IPv6. And it is not capable of talking to a name server. This means we have to deal with IP addresses directly. DHCP is only possible in a very restrictive way when using TFTP.

MAC addresses as well as IP addresses are stored in the U-Boot environment. If the environment was lost, for example by erasing the whole NAND flash with *E* command in NBoot or by erasing the `UBootEnv` MTD partition, then it is mandatory to set at least the MAC address(es) again.

5.7.1 Set MAC Address

Each network device needs its own MAC address. U-Boot can only handle Ethernet based network devices, but Linux also uses WLAN or Bluetooth devices that may also need MAC addresses. F&S uses the U-Boot environment to store *all* these MAC addresses, also those needed for non-Ethernet devices. U-Boot will pass on these values to Linux via the device tree.

If one MAC address is needed, it is stored in variable `ethaddr`. If a second MAC address is required, it is stored in variable `eth1addr`. If a third MAC address is required, it is stored in variable `eth2addr`. And so on. As you can see, the entry for the first address unfortunately does not follow the naming rule of the other variables. For historical reasons this variable is called `ethaddr` instead of `eth0addr`.

A MAC address consists of twelve hexadecimal digits (0 to 9 and A to F), that are often grouped in pairs and separated by colons. The first six digits for F&S boards are always the same: 00:05:51, which is the official MAC address code for the F&S company. The remaining six digits can be found on the 2D-bar-code sticker directly on your board (see Figure 27). The full MAC address for this sticker would be 00:05:51:07:93:4B.

Figure 27: Bar-code sticker

If a board needs more than one MAC address, the sticker only mentions the first address. The remaining addresses are then the consecutive MAC addresses, i.e. the addresses where the last number is increased by one each. F&S uses stickers of different colours to denote how many MAC addresses are available.

- A white sticker is for one MAC address
- A yellow sticker is for two MAC addresses
- A pink sticker is for three MAC addresses
- An orange sticker is for four MAC addresses

The following example sets two MAC addresses and stores the current environment (including the newly set MAC addresses) in NAND flash. Of course you have to replace `xx:yy:zz` with the six hex digits from the bar-code sticker on your board. And in the second command, you have to determine the next higher value zz+1 and directly type the result. U-Boot can not do any arithmetic on MAC addresses.

```
setenv ethaddr 00:05:51:xx:yy:zz
setenv eth1addr 00:05:51:xx:yy:zz+1
saveenv
```

Warning

If you do not set these unique addresses, default addresses are used that are the same for all boards of this type. This will definitely lead to problems in real networking scenarios.

Do not use more addresses than the colour of the sticker allows. Any subsequent addresses are used by other F&S boards.

Remark

Some customer specific boards may have the MAC addresses pre-programmed in the fuses (OTP Memory) of the SoC. Here the appropriate variables will always be set automatically, even if the NAND flash is erased. You can still override this by setting the `ethaddr` variables manually.

5.7.2 Set IP addresses

IPv4 addresses are built from four numbers from 0 to 255, separated by dots. Examples would be 10.0.1.122 or 192.168.1.1.

When downloading files, at least two devices are involved: the server that sends the file and the client that receives the file. In our case, the server is often the development PC and the client is the board. However if server and client are not in the same subnet, a third device is involved: the gateway. The gateway is the one device in the local subnet that knows how to get outside of the subnet to other destinations. So we need to configure up to three IP addresses in variables `serverip`, `ipaddr` and `gatewayip`.

However, the local client also needs another piece of information, namely whether the remote station is in the local subnet or not. This is done by setting the `netmask` variable. The local and the remote IP addresses are each bitwise AND-combined with `netmask`. If the res-

ults are the same, both devices are in the same subnet and can talk directly to each other. If the results differ, then the devices are in different subnets and must go via the gateway to communicate.

Example 1

The board with `ipaddr` 10.0.0.252 should download files from the server with `serverip` 10.0.1.122. The 10.x.y.z subnet can have up to 2^{24} devices, i.e. all IP addresses that start with 10 are in the same subnet. This means the `netmask` has to be set to 255.0.0.0. A `gatewayip` is not needed.

```
setenv ipaddr 10.0.0.252
setenv serverip 10.0.1.122
setenv netmask 255.0.0.0
```

You can use `ping` to verify that your network access is working.

```
# ping 10.0.1.122
Using FEC0 device
host 10.0.1.122 is alive
```

Example 2

The board with `ipaddr` 192.168.1.17 should download files from the server with `serverip` 192.168.2.2. The network is divided in small subnets, only devices with the same first three numbers are in the same subnet. Which means `netmask` has to be set to 255.255.255.0. The gateway in subnet 192.168.1.x has `gatewayip` 192.168.1.1. This is a configuration that is often found in private homes or small companies.

```
setenv ipaddr 192.168.1.17
setenv serverip 192.168.2.2
setenv gatewayip 192.168.1.1
setenv netmask 255.255.255.0
```

Again you can use `ping` to verify that the configuration is working.

5.7.3 RNDIS

Some F&S boards do not provide an Ethernet port. And even if they provide a WLAN interface, this can not be used in U-Boot, because WLAN drivers are not included in U-Boot because of their complexity. On these boards, e.g. the PicoCoreMX7ULP, you can use RNDIS over USB as a replacement network device.

The *Remote Network Driver Interface Specification* (RNDIS) is a protocol introduced by Microsoft, mostly used on top of USB. It provides a virtual Ethernet interface to a remote terminal, usually a PC. This remote PC can then either serve data directly, or it can be used as gateway to relay any requests into the regular LAN. The F&S board is the device side and the PC is the host side of the USB communication (see Figure 28). RNDIS is implemented in Windows, Linux, and FreeBSD operating systems, so all of them are suited as remote partner in the RNDIS communication.

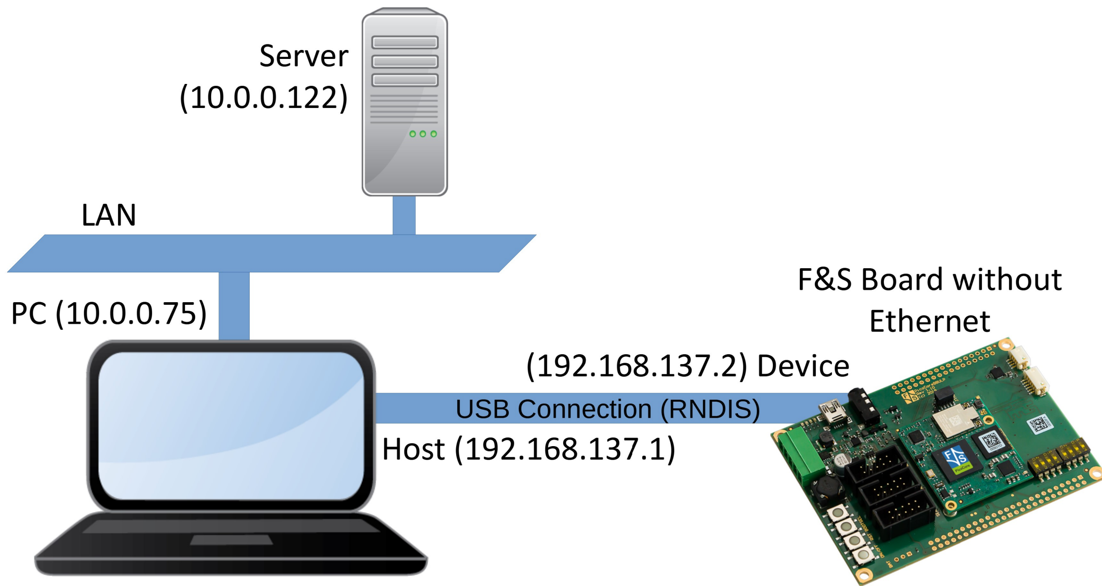


Figure 28: Network access via RNDIS

If RNDIS is set up, you can use the regular network commands of U-Boot, e.g `ping`, `tftp`, `nfs`, etc. U-Boot will automatically establish the USB connection and executes the command. Then it shuts down the USB connection again. This shows that RNDIS has quite some overhead in U-Boot, as the communication is started for each command again. Or in other words, the USB device will only be visible on the host side for the short period while the command is running. After that, the device disappears again. This is not perfect, but sufficient for the occasional software download during development.

The environment variables shown in Table 8 are necessary for the RNDIS communication.

Environment Variable	Value
`ethaddr`	00:05:51:07:55:83
`usbnet_devaddr`	00:05:51:00:00:01
`usbnet_hostaddr`	00:05:51:00:00:02
`ipaddr`	192.168.137.2
`gatewayip`	192.168.137.1
`serverip`	10.0.0.122 (example)

Table 8: Necessary environment variables for RNDIS

The board and the PC will open their own subnet to communicate with each other. This needs to be different to the subnet where the PC is included in the LAN. So if the LAN is using the 10.0.0.x subnet for example, you can not use this for the direct communication, too. When Microsoft Windows is installed on the host PC side, Windows will use 192.168.137.1

for the PC by default. This means variable `ipaddr` must be in the same IP range. So simply choose 192.168.137.2.

This means PC and board are now both in subnet 192.168.137.x and can talk to each other. But to be able to talk to the server 10.0.0.122 in the 10.0.0.x subnet of the LAN, it requires a gateway setting. If the virtual RNDIS network adapter on the host PC is bridged to the real hardware network adapter, then the PC can take the role of the gateway. So set variable `gatewayip` to the IP address of the PC, which is 192.168.137.1.

The variables `usbnet_devaddr` and `usbnet_hostaddr` are hard-coded in U-Boot, but you are able to overwrite them if needed. These are some unused MAC addresses from the F&S MAC address range that can be used for the RNDIS communication. Variable `ethaddr` is not actually used in the communication, because `usbnet_devaddr` is used in RNDIS communications instead, but the regular network commands will complain if `ethaddr` is not set. The address above is the default value that U-Boot uses in these cases.

If not configured already, you can set the values with these commands:

```
setenv usbnet_devaddr 00:05:51:00:00:01
setenv usbnet_hostaddr 00:05:51:00:00:02
setenv ethaddr 00:05:51:07:55:83
setenv ipaddr 192.168.137.2
setenv gatewayip 192.168.137.1
```

When everything is set up, connect the USB device cable and verify the connection. You can use `ping` to verify that your network access is working.

```
# ping 10.0.0.122

Using usb_ether device
host 10.0.0.122 is alive
```

5.8 Image Download

One of the main purposes of U-Boot is to install the images that build the operating system. In case of the default F&S Linux, this is the Linux kernel image, the device tree image and the root filesystem image. They are all installed to NAND flash. Custom configurations may consist of more images and may want to install these images to different places. But no matter how many images are to be installed, each image basically needs two steps

1. Load the image to RAM
2. Save the image to the appropriate device, e.g. NAND flash

So first we will look at the loading part. There are quite a few possibilities how to download an image.

- Network download with TFTP or NFS
- Low-level read from a USB device
- Low-level read from an SD card
- Low-level read from a UBI volume

- High-level read from a filesystem (FAT, EXT2, EXT4, UBIFS)

Let us look at these options in more detail. Later we will see how these images can be saved.

5.8.1 Network Download With TFTP or NFS

For TFTP or NFS, the network has to be configured, i.e. environment variables `ipaddr`, `serverip`, `gatewayip`, `netmask` and of course the MAC addresses in the `ethaddr` variables have to be set appropriately (see Chapter 5.7 on page 56).

We also assume that TFTP and/or NFS services are correctly set up and running on the server. When you use the F&S Virtual Development Machine or follow the steps in the `AdvicesForLinux_eng.pdf` documentation, then the setting is as follows

- Files to be downloaded with TFTP have to be stored in `/tftpboot` on the server.
- Files to be downloaded with NFS have to be stored in `/rootfs` on the server.

To download a file with TFTP use the `tftpboot` command (or `tftp` for short).

```
tftp <loadaddr> <serverip>:<file>
```

In this case, the file `<file>` will be downloaded from the server with `<serverip>` to RAM address `<loadaddr>`. If `<loadaddr>` is omitted, the address is taken from variable `loadaddr`. If `<serverip>` and the colon are omitted, the IP address is taken from variable `serverip`. Even `<file>` can be omitted, then a generic filename is generated from the variable `ipaddr` converted to 8 hex digits appended by extension `.img`. For example a board with `ipaddr` set to 10.0.0.252 will use the file name `0A0000FC.img` (10=0x0A, 0=0x00, 0=0x00, 252=0xFC).

A common case is to just give the file name to download. The following example will download a file `zImage` from the server configured in the environment.

```
# tftp zImage
Using FEC device
TFTP from server 10.0.0.168; our IP address is 10.0.0.252
Filename 'zImage'.
Load address: 0x11000000
Loading:
  #################################################################  1466 KiB
  #################################################################  2934 KiB
  #################################################################  4402 KiB
  #########################################################
  3.1 MiB/s
done
Bytes transferred = 5831584 (0x58fba0)
```

As you can see, while downloading the file, progress is indicated by hash marks and the amount of already transmitted bytes from time to time. At the end a summary is given.

Actually U-Boot can do a DHCP request before starting a TFTP download. In this case use command `dhcp` instead of `tftp`. It has the same syntax but updates variables `ipaddr`,

`netmask` and `gatewayip` according to the answers from the DHCP server before loading the file via TFTP. Of course if you want to download several files, only the first command needs to be `dhcp`, the other commands can again be `tftp`.

The following example will load a kernel image and a device tree and starts the Linux system that is built by these two images. On F&S boards, variable `bootfile` is set to the name of the kernel image and variable `bootfdt` to the name of the device tree image. Variables `loadaddr` and `fdtaddr` are set to some meaningful RAM addresses for these images. As you can see the resulting commands are completely architecture independent and can be run on any F&S board.

```
# dhcp $loadaddr $bootfile
BOOTP broadcast 1
BOOTP broadcast 2
*** Unhandled DHCP Option in OFFER/ACK: 43
*** Unhandled DHCP Option in OFFER/ACK: 43
DHCP client bound to address 10.0.0.84 (709 ms)
Using FEC device
TFTP from server 10.0.0.168; our IP address is 10.0.0.84
Filename 'zImage'.
Load address: 0x11000000
Loading:
  #################################################################  1466 KiB
  #################################################################  2934 KiB
  #################################################################  4402 KiB
  #########################################################
  3.3 MiB/s
done
Bytes transferred = 5830096 (0x58f5d0)
# tftp $fdtaddr $bootfdt
Using FEC device
TFTP from server 10.0.0.168; our IP address is 10.0.0.84
Filename 'efusa9dl.dtb'.
Load address: 0x12000000
Loading:
  #
  1.3 MiB/s
done
Bytes transferred = 44854 (0xaf36)
# bootm $loadaddr - $fdtaddr
## Booting kernel from zImage at 11000000
## Flattened Device Tree blob at 12000000
   Booting using the fdt blob at 0x12000000
   Loading Kernel Image ... OK
   Using Device Tree in place at 12000000, end 1200df35
   Setting run-time properties
## Overwriting property gpmi-nand@00112000/fus,ecc_strength from device tree!
## Overwriting property bdinfo/ecc_strength from device tree!
## Keeping property bdinfo/board_name from device tree!
Starting kernel ...
```

The command to download a file with NFS is `nfsboot` (or `nfs` for short).

```
nfs <loadaddr> <serverip>:<path>
```

This will download the file given with the full path `<path>` from server with `<serverip>` to RAM address `<loadaddr>`. Again if `<loadaddr>` is ommited, the address is taken from variable `loadaddr`. If `<serverip>` and the colon are omitted, the IP address is taken from variable `serverip`.

A common case is to just give the file name to download. The following example will download a file `zImage` from the server configured in the environment that has exported directory `/rootfs`.

```
# nfs /rootfs/zImage
Load address: 0x11000000
Loading:
  #################################################################  1024 KiB
  #################################################################  2048 KiB
  #################################################################  3072 KiB
  #################################################################  4096 KiB
  #################################################################  5120 KiB
  ##
done
Bytes transferred = 5264904 (0x505608)
```

As you can see, output is rather similar to the TFTP case.

5.8.2 Low-level Read From USB Device

A USB device (USB stick or USB hard disc drive) can be connected to a USB host port of the board. But as U-Boot does not support interrupts, the USB port must be polled. So it can not automatically detect a device that is plugged in, you have to start the USB subsystem manually. The USB subsystem is handled with the `usb` command.

```
# usb start
starting USB...
USB0:   Port not available as HOST.
USB1:   USB EHCI 1.00
scanning bus 1 for devices... 3 USB Device(s) found
        scanning usb for storage devices... 1 Storage Device(s) found
```

This will scan all available USB host ports and check for mass storage devices. The output in the example above shows that there are two USB ports `USB0` and `USB1`, but `USB0` is the device port and is not available as host port. On `USB1`, there are three devices found and one of them is a storage device.

To get more information about the connected devices, call:

```
# usb tree
USB device tree:
  1  Hub (480 Mb/s, 0mA)
  |  u-boot EHCI Host Controller
```

```
  |
  +-2  Hub (480 Mb/s, 2mA)
    |
    +-3  Mass Storage (480 Mb/s, 200mA)
                  USB DISK 2.0 07B9020B22F42A4C
```

This shows the devices in a tree-like structure. We see the root hub as device 1, an additional hub as device 2 and the storage device as device 3.

To see some information of the devices in general, use the following command:

```
# usb info
1: Hub,  USB Revision 2.0
 - u-boot EHCI Host Controller
 - Class: Hub
...
```

To see more information of the storage devices only, use:

```
# usb storage
Device 0:
  Interface: USB
  Vendor:
  Product: USB DISK 2.0
  Revision: PMAP
  Type: Removable Hard Disk
  Capacity: 3700.6 MB = 3.6 GB (7579008 x 512)
```

It is important to know the storage device ID, because you have to use this when loading data from the device. Storage device IDs are counted from zero, so in most cases it will be device 0. But of course on boards with more than one host port or when using an external hub, you can connect more than one storage device. Then you need the correct ID to be able to talk to each device independently.

A low-level read needs the address in RAM where to load the data to, the sector number on the USB device where to start reading from and the sector count, i.e. how many sectors should be read. Please note that all numbers are in hex, even if no `0x` is prepended. The following command reads 0x5000 (=20480) sectors starting from sector 0x200 (=512) and stores the data at the RAM address given by variable `loadaddr`.

```
# usb read $loadaddr 0x200 0x5000
usb read: device 0 block # 512, count 20480 ... 20480 blocks read: OK
```

Please note that from this low-level view of the `usb` command, you can only see the whole device as one big space with absolute sector numbers. If you have a partition table on the device, you can not access data relative to a partition start, you always have to give absolute numbers. But at least you can list the partitions with the following command:

```
# usb part
Partition Map for USB device 0  --   Partition Type: DOS
```

```
Part     Start Sector     Num Sectors     UUID            Type
  2      92240            10640           7b0e0f1b-02     ef Boot
```

In this example, the Stick is equipped with a valid partition table that holds one partition. The partition starts on sector 92240 and is 10640 sectors long. Unfortunately U-Boot uses decimal numbers here, which is not very consistent.

Remark

If you replace a USB stick with a different one, you have to restart the USB subsystem with

```
# usb reset
resetting USB...
```

This will automatically rescan the devices.

5.8.3 Low-Level Read From SD Card

SD cards, or MMC devices in general, are handled with the `mmc` command. So handling an SD card and eMMC memory is completely the same. To see all available MMC ports, use the command:

```
# mmc list
FSL_SDHC: 0 (SD)
FSL_SDHC: 1
FSL_SDHC: 2
```

This shows that there are three MMC ports 0 to 2. The name `FSL_SDHC` is given by the driver and does not matter. One device is the current device and all `mmc` commands will refer to it. By default this is port 0.

Note

MMC devices in U-Boot may be numbered differently to Linux. U-Boot enumerates them in the sequence of initialization, while Linux uses the hardware port number. Devices called `mmc 0`, `mmc 1` and `mmc 2` in U-Boot can for example be named `mmc 3`, `mmc 0` and `mmc 1` in Linux. So be careful in cases where you need a Linux name, for example when preparing the Linux command line in variable `bootargs`.

F&S initializes the ports in U-Boot in an order so that the first port is always a slot for replaceable media. Non-removable devices like eMMC are last in the list. The idea is that the first MMC device is often used for updates by inserting a card with new software. Then scripts are easier to implement if they can use port 0 directly without having to query the right port number first. A non-removable device like eMMC is less probably used as update media.

Because U-Boot has no support for interrupts, it can not automatically detect when you plug in or replace an SD card. If the card is already inserted at boot time, it is automatically scanned. The same is true if you switch to a different port. But if you insert or replace a card at runtime, you have to discover it manually. This is done with the following command:

```
# mmc rescan
```

To get information about the device, enter the following command:

```
# mmc info
Device: FSL_SDHC
Manufacturer ID: 3e
OEM: 482d
Name: SMI
Bus Speed: 50000000
Mode : SD High Speed (50MHz)
Rd Block Len: 512
SD version 2.0
High Capacity: No
Capacity: 957 MiB
Bus Width: 4-bit
Erase Group Size: 512 Bytes
```

This shows a low capacity SD card with approximately 1 GB of space and high speed support.

To low-level read some data from an MMC device, you need the address in RAM where the data should be stored, the starting sector and the number of sectors to read. Please note that all values are hex, even if you do not prepend 0x. The following command will read 0x2000 (=8192) sectors starting at sector 0x100 (256) and stores the data at the RAM address given by environment variable `loadaddr`.

```
# mmc read $loadaddr 0x100 0x2000
MMC read: dev # 0, block # 256, count 8192 ... 8192 blocks read: OK
```

Please note that from this low-level view of the `mmc` command, you can only see the whole device as one big space with absolute sector numbers. If you have a partition table on the device, you can not access data relative to a partition start, you always have to give absolute numbers. But at least you can list the partitions with the following command:

```
# mmc part
Partition Map for MMC device 0  --   Partition Type: DOS

Part    Start Sector    Num Sectors     UUID            Type
  1     255             1959681         00000000-01     06
```

In this example, the card is equipped with a valid partition table that holds one partition. The partition starts on sector 255 and is 1959681 sectors long. Unfortunately U-Boot uses decimal numbers here, which is not very consistent.

If you want to use a different device, you first have to switch the current device to the new port. The following command will switch to the device on port 2:

```
# mmc dev 2
switch to partitions #0, OK
mmc2(part 0) is current device
```

In our example, the device on port 2 is an eMMC memory. As shown in Chapter 3.8.2, eMMC memory has more features compared to regular SD cards. For example it can use up to eight data lines (instead of four), can use double data rate, and the data region can be subdivided into hardware partitions. Each hardware partition behaves like a separate device. That means each hardware partition has its own size and the data in it begins at sector 0. You can even install a regular partition table on each hardware partition, further subdividing it into software partitions.

When we list the device information of this eMMC, we get the following result:

```
# mmc info

Device: FSL_SDHC
Manufacturer ID: fe
OEM: 14e
Name: MMC04
Bus Speed: 52000000
Mode : MMC High Speed (52MHz)
Rd Block Len: 512
MMC version 4.4.1
High Capacity: Yes
Capacity: 3.6 GiB
Bus Width: 8-bit
Erase Group Size: 512 KiB
HC WP Group Size: 4 MiB
User Capacity: 3.6 GiB
Boot Capacity: 2 MiB ENH
RPMB Capacity: 128 KiB ENH
```

This info shows, that the device has 3.6 GiB of space in the User region, 2 MiB of space for each Boot partition, and 128 KiB of space in the RPMB region. In addition, the Boot partitions and the RPMB partition use Enhanced Mode (also called Pseudo-SLC mode). This makes data in these partitions more reliable for the cost of less available space.

When selecting an eMMC device using the `mmc dev` command as seen before, this will automatically select partition 0, which is the user region. However you can add the partition ID as an additional argument to select a different hardware partition. For example to access the `Boot1` partition, you have to type

```
# mmc dev 2 1

switch to partitions #1, OK
mmc2(part 1) is current device
```

Now reading data would use the `Boot1` hardware partition.

Important

Adding GP partitions or activating Enhanced Mode (Pseudo-SLC) on eMMC can be done with the `mmc hwpartition` command. But please note that this is a write once operation. So if these changes are completed, they can never be undone again and remain

forever. So please carefully consider if you really need this. If you do it wrongly, you can unintentionally reduce the available size of the eMMC considerably.

If you just need some partitions, using a software partition table on the User area is often the better and easier solution.

5.8.4 Low-level Read From UBI Volume

Reading data from a UBI volume is rather simple. You just need to tell which MTD partition holds the UBI…

```
ubi part TargetFS
```

...and then you can read the data. For example to read volume `rootfs`, simply call

```
# ubi read $loadaddr rootfs
Reading from volume rootfs ... OK, 210653184 bytes loaded to 0x11000000
```

If you just want to read a limited number of bytes, you can add the count as an additional argument.

```
# ubi read $loadaddr rootfs 0x20000
Reading from volume rootfs ... OK, 131072 bytes loaded to 0x11000000
```

An example where reading a volume can be useful is a recovery scenario. Imagine your main filesystem has severe damage and does not start any more. But you have a backup of your original root filesystem in a separate volume. Then you simply need to read this backup volume now and use it to replace the contents of the volume with the damaged root filesystem.

5.8.5 High-Level Read From Filesystem

The commands `usb`, `mmc` or `ubi` only provide a low-level view on the devices, i.e. on a sector-by-sector basis. But usually we want to read images that are stored as files in the filesystem on these devices. This is possible by using the filesystem commands. These commands allow listing the directory contents and loading a file. U-Boot supports FAT, EXT2, EXT4 and UBIFS filesystems.

Historically there were separate commands for each filesystem, i.e. `fatls` and `fatload` for FAT, `ext2ls` and `ext2load` for EXT2, `ext4ls` and `ext4load` for EXT4 and `ubifsls` and `ubifsload` for UBIFS. These commands still exist, but now there is a common interface that can be used for any filesystem. These commands are simply called `ls` and `load`. You only have to give the device type and the device ID, supplemented by the partition number if necessary. The remaining syntax is the same, no matter what filesystem is actually present on the device.

Before being able to use these commands, the device must be available. Which means you need to call `usb start` for USB devices, `mmc rescan` for MMC devices and `ubi part` for UBI. But UBI needs an additional step to actually have access to the UBIFS filesystem.

```
ubifsmount ubi0:rootfs
```

The syntax is taken from Linux, where there can be several different UBI devices. However in U.Boot, only one UBI can be active at a time, so the prefix is always `ubi0:`, followed by the volume name.

The following examples show the usage of `ls` and `load`.

List the files on the first USB storage device:

```
# ls usb 0

/:
 <DIR>        System Volume Information/
      36910   efusa7ul.dtb
      36449   efusa7ull.dtb
        890   install.scr
      65536   nbotmx6u.bin
      34219   picocom1.2.dtb
      35333   picocoremx6ull.dtb
   72884224   rootfs-fsimx6ul.ubifs
     524288   ubotmx6u.nb0
    5586048   zImage-fsimx6ul
```

Load the file `zImage-fsimx6ul` to the RAM address given by variable `loadaddr`:

```
# load usb 0 $loadaddr zImage-fsimx6ul

Loading /zImage-fsimx6ul ... done!
5586048 bytes read in 599 ms (8.9 MiB/s)
```

List the files on the device on MMC port 0:

```
# ls mmc 0

/:
 <VOLUME>     SD Card
     262144   uboot.nb0
         47   noaction.txt
  103436638   EW2011-1080p50-crf10.avi
```

Load the file `uboot.nb0` to the RAM address given by variable `loadaddr`.

```
# load mmc 0 $loadaddr uboot.nb0

Loading /uboot.nb0 ... done!
262144 bytes read in 35 ms (7.1 MiB/s)
```

List the files in the currently mounted UBIFS

```
# ls ubi 0

<DIR>        5224  Tue Jul 09 11:02:21 2019  bin
<DIR>         608  Sun Jul 07 20:09:08 2019  dev
<DIR>        2528  Tue Jul 09 11:02:56 2019  etc
<DIR>        3592  Tue Jul 09 11:02:49 2019  lib
<DIR>         160  Sun Jul 07 20:09:08 2019  mnt
<DIR>         160  Sun Jul 07 20:09:08 2019  opt
…
```

```
<DIR>          544  Tue Jul 09 11:02:49 2019  usr
<DIR>          160  Sun Jul 07 20:09:08 2019  proc
<DIR>         6456  Tue Jul 09 11:02:49 2019  sbin
<DIR>          160  Sun Jul 07 20:09:08 2019  root
<LNK>           11  Tue Jul 09 10:19:31 2019  linuxrc
<LNK>            3  Tue Jul 09 09:58:03 2019  lib32
<DIR>          160  Sun Jul 07 20:09:08 2019  media
```

Load the file bin/busybox

```
# load ubi 0 $loadaddr bin/busybox
773212 bytes read in 115 ms (6.4 MiB/s)
```

If the device has a software partition table on it, you can add the partition number to the device ID, separated by a colon. Partitions are counted from one, so the first partition on device 0 is 0:1.

For example to list the files on partition 2 of the second USB storage device (ID 1) use this command::

```
ls usb 1:2
```

You can also use virtual partition 0. This is a shortcut to the first partition, if the device contains a partition table, or to the whole device otherwise. In fact when using the device ID without a partition number, this is the same as referring to partition 0. So `mmc 0` is the same as `mmc 0:0`. In most cases, the device ID alone is sufficient.

If an image is too large to fit into RAM in one go, you can also load only parts of a file. Just add the number of bytes to load as an additional argument. And if loading should not start at the beginning of the file, also add a byte offset.

To load 0x2000 (=8192) bytes from the beginning of the file:

```
# load mmc 0 $loadaddr uboot.nb0 0x2000
Loading /uboot.nb0 ... done!
8192 bytes read in 10 ms (799.8 KiB/s)Loading /uboot.nb0 ... done!
```

To load 0x1000 (=4096) bytes from offset 0x8000 of the file:

```
# load mmc 0 $loadaddr uboot.nb0 0x1000 0x8000
Loading /uboot.nb0 ... done!
4096 bytes read in 14 ms (285.2 KiB/s)
```

5.9 Image Storage

In the previous section we have seen different ways how images can be downloaded to RAM. Now we will see how images can be saved from RAM to some non-volatile memory. These are the possible options:

- Save to a NAND flash MTD partition
- Save to a UBI volume

- Save to an SD Card or eMMC

One of the important things when saving data is that we need to know how much data we need to save. U-Boot helps us with this. Most download commands will set the environment variable `filesize` to the number of bytes that were transferred. So we can use this variable to refer to the size of the currently downloaded file.

> **Note**
>
> Writing to a filesystem (FAT, EXT2, EXT4, UBIFS) is not possible in U-Boot. This is a deliberate decision. U-Boot should be used to bring images *to* the board, not to bring data away *from* the board. This helps to prevent unauthorized copies of already installed software.

5.9.1 Save to NAND Flash MTD Partition

NAND flash can only write new data to areas that do not contain any old data. So if we want to replace existing data, we have to erase this old data first.

The easiest way is to have an own MTD partition for each image that needs to be stored. Then you only need the partition name and the size of the data to write. If you want to store more than one image in a partition, you have to keep track of all the addresses or offsets yourself.

We assume that a kernel image was loaded to RAM at the address given by variable `loadaddr`. The following commands will erase the Kernel partition and stores the image there.

```
# nand erase.part Kernel

NAND erase.part: device 0 offset 0x240000, size 0x800000
Erasing at 0xa20000 -- 100% complete.
OK

# nand write $loadaddr Kernel

NAND write: device 0 offset 0x240000, size 0x58fb88
 5831560 bytes written: OK
```

The first command erases the MTD partition `Kernel` and the second command stores the image there. We only store the real size of the image given by variable filesize, not the full area of the `Kernel` partition. This is slightly faster.

> **Note**
>
> It is important that the partition is slightly larger than the image to be saved there. NAND flash can always have bad blocks. These blocks are skipped when writing, which means if an image would fit exactly into the partition, it will no longer fit if there are any bad blocks.
>
> This is why F&S uses partitions that are larger than actually required by the appropriate images, to be prepared for potential bad blocks. For example the `UBoot` partition is

768 KiB when the U-Boot image is only 512 KiB. This allows for up to two bad blocks in this region, if a block has 128 KiB.

5.9.2 Save to UBI Volume

Before being able to write to an UBI volume, we have to tell U-Boot where the UBI volume is located.

```
ubi part TargetFS
```

Then we can save the image. We assume a root filesystem image was loaded to the address given by variable `loadaddr` and should be saved to the `rootfs` volume. The size in variable `filesize` was set by the download command before.

```
# ubi write $loadaddr rootfs $filesize
Writing to volume rootfs ... OK, 91930624 bytes stored
```

As you can see it is not necessary to erase the volume first. UBI will automatically erase the blocks that it needs.

5.9.3 Save to SD Card or eMMC

Writing data to the SD card or eMMC memory is rather similar to reading data from it. First you have to make sure that the correct MMC port is selected with `mmc dev` and the device is correctly detected with `mmc rescan`. Then you can write the data.

We assume that a partition image with 512 MiB (=0x00100000 sectors) is available in RAM and should be written at sector 0 to the MMC device. Please note that the variable `filesize` is of no help here, because we have to give the size as sector count, not as byte count.

```
# mmc write $loadaddr 0 0x00100000
MMC write: dev # 2, block # 0, count 1048576 ... 1048576 blocks written: OK
```

Again we do not need to erase the target before writing the data. This is automatically done by the controller inside of the device.

Filesystem images for SD cards or eMMC memory are usually rather large and often do not fit into RAM as a whole. In this case you have to split the image in smaller parts, read each part separately and use an own `mmc write` command for each part.

5.9.4 High-Level Write To Filesystem

On some boards, for example the PicoCoreMX7ULP, U-Boot, the Linux kernel and the device tree are stored in a FAT software partition on eMMC. These boards typically support high-level writing to FAT filesystems. This works similar to reading. For example if the FAT filesystem is located on the first partition on the boot device in `mmcdev`, then the following command will write data from RAM at the address in `loadaddr` to the file `zImage`.

```
write mmc $mmcdev:1 $loadaddr zImage $filesize
```

Again there is the old command `fatwrite` instead of the filesystem independent `write` command if you prefer this.

5.10 Exporting eMMC as USB Mass Storage Device

But there is also a completely different option to get data from and to such a FAT partition: USB Mass Storage Device. If you have a USB cable connected between the USB Device port of the board and a USB Host port of your PC, then you can use the `ums` command to export the whole eMMC as Mass Storage Device.

```
# ums 0 mmc $mmcdev
UMS: LUN 0, dev 1, hwpart 0, sector 0x0, count 0x748000
-
```

From now on the board behaves like a USB stick and the PC will open a window with the content. Now you can use drag and drop to copy files between your PC and the board.

For example if you want to store a new Linux kernel version, simply replace the kernel image, e.g. `zImage`, with the new version.

When you are done, simply use the appropriate eject sequence for USB media on your PC to finish any pending writes and then stop the `ums` command in U-Boot with *Ctrl-C*. Then you you are back on the command line to enter new commands.

5.11 Booting the Linux System

Starting the Linux system is rather simple. Just start the board and after a few seconds, U-Boot will automatically boot the kernel. Or if you are already in the U-Boot command line interface, then simply use command

```
boot
```

This actually executes the commands that are stored in variable `bootcmd`. So if you want to change the behaviour of the boot sequence, then just modify this variable. But in the end, there must be a command that actually transfers the execution to the Linux kernel. This is the `bootm` command. In the past without device trees, just the address of the kernel had to be given. This means the kernel image had to be loaded to RAM and then this RAM address had to be given as argument to `bootm`.

Nowadays, however, there is also the device tree image, in U-Boot often called *Flat Device Tree*, or FDT for short. This means two such images need to be loaded to RAM and the RAM address of the device tree has to be given as argument to `bootm`, too. Actually, the device tree address is the third argument to `bootm`, not the second. Because the second argument can be used for the initial ramdisk.

An initial ramdisik can be used in complicated boot sequences when access to the real root filesystem requires loading some drivers. Then you would create a rather small preliminary root filesystem image that only contains all necessary command line tools, scripts, drivers and the `init` program. This root filesystem would be loaded completely to RAM before the kernel is started, hence the name initial ramdisk, or initrd for short. Then Linux would use this

initrd and the programs in it to mount the final root filesystem, switch to this final root filesystem and finally drop the initrd.

However the boot sequence on F&S boards is simple enough that we can directly mount the final root filesystem, an initrd image is not needed. This means we have to load the kernel image and the device tree to RAM, usually from MTD partitions `Kernel` and `FDT`, and then we can call `bootm`.

```
nand read $loadaddr Kernel
nand read $fdtaddr FDT
bootm $loadaddr - $fdtaddr
```

As you can see we use variables `loadaddr` and `fdtaddr` to provide meaningful addresses for the images. And note the `-` (dash) character as second argument to tell `bootm` that we do not use an initrd image.

Note

Normally when using a kernel image of type zImage, you would have to use a different boot command called `bootz` with the same syntax. However F&S has modified the `bootm` command so that it also works with zImages. This means there is no difference to previous releases where we used the uImage type for the kernel and where `bootm` was mandatory.

6 Special F&S U-Boot Features

F&S has an enhanced U-Boot to work more reliably and to be used more intuitively. The following list shows a short summary.

- The often used expression `$loadaddr` can be abbreviated as a simple dot.
- A new FAT driver uses less RAM and allows wildcards in FAT filenames.
- An improved NAND driver speeds up NAND access and makes it more reliable by adding block refresh.
- The Install/Update/Recover mechanism provides automatic software installation in the production, automatic updates in the field and automatic recovery to the delivery status or the previous version in case of damaged system files.
- Linux Boot Strategies provide different ways to start the system. For example during development when images change on a regular basis, it may be useful to download all images over the network on the fly. However in the final stand-alone system, the images need to be available locally.
- Different NAND Layout Strategies provide common scenarios to store kernel and device tree images.
- Allow percent values and hex numbers when setting eMMC to Enhanced Mode.
- Special command `fsimage` to handle F&S images

We will show these features in more detail in the upcoming sections.

6.1 Simplified $loadaddr

The variable `$loadaddr` is needed so many times in so many U-Boot commands, that F&S has made an extension to U-Boot to make its usage easier. Instead of having to type `$loadaddr` all the time, you can simply type a single dot. So for example instead of

```
nand read $loadaddr Kernel
```

you can simply type

```
nand read . Kernel
```

Please note that this is no standard behaviour of U-Boot. When using U-Boot versions of other manufacturers, or even in old U-Boot versions of F&S, the dot may be interpreted as the value zero, which is not what you want.

In our examples throughout this document we will still use the long form `$loadaddr` for two reasons. A single dot may easily be overlooked and considered “dirt” on the paper and skipped at the input. And using the long form makes the intention of the command more clear that a variable and some memory address is accessed. So we do not use the short form for didactic reasons.

6.2 Allow Wildcards in FAT Filenames

USB sticks or MMC devices (SD cards) are typically formatted with the FAT filesystem. The original U-Boot has quite some problems when handling long filenames (VFAT entries). For example if two filenames start with the same sequence of more than eight characters, U-Boot may load the wrong file because it will not detect the difference.

Because of this insufficiency, F&S has completely rewritten the FAT access code. It consumes far less memory for buffers in RAM (one FAT sector = 512 bytes compared to two clusters of up to 32 KB each in mainline U-Boot).

In addition, it can also work with wildcards (see Table 9).

Wildcard	Description	Example
`*`	Matches any number of arbitrary characters, even zero	`ub*txt` will match `ub123txt`, `uboot.txt` and even `ubtxt`, but not `uboot.nb0`
`?`	Matches exactly one arbitrary character	`ub??txt` will match `ub12txt` and `ubi.txt`, but not `uboot.txt`

Table 9: Wildcards in FAT names

For example if you look at the content of an SD card with a kernel image, root file system and a U-Boot image stored there it will show the following list:

```
armStoneA9 # mmc rescan
armStoneA9 # ls mmc 0
/:
     524288  uboot-fsimx6.nb0
    4497696  zImage-fsimx6
   57266176  rootfs-fsimx6.ubifs
```

So for example to load the `rootfs-fsimx6.ubifs` file from this SD card, simply type

```
load mmc 0 $loadaddr root*
```

Or load the file that ends in `nb0` with

```
load mmc 0 $loadaddr *nb0
```

If the wildcard expansion results in more than one filename, this will only work for the `ls` command. But when trying to load a file, this will result in an error.

```
armStoneA9 # ls mmc 0 *fsimx6*

/:
    4497696  zImage-fsimx6
   57266176  rootfs-fsimx6.ubifs

armStoneA9 # load mmc 0 $loadaddr *fsimx6*

/: Ambiguous matches for "*fsimx6*"
```

Note

The concept of wildcard characters in filenames would also be handy with other filesystems like EXT2/EXT4 or UBIFS, and the code is even written in a way that other filesystems could take advantage of it. Unfortunately at the moment only the F&S FAT driver actually uses the wildcard code.

6.3 Improved NAND Driver

Compared to the regular Linux releases by NXP, F&S has considerably improved the NAND access in NBoot, U-Boot and Linux. It uses ECC with high error correction capabilities (at least 16 bit errors can be corrected in a 2K page), is much faster and it also has block refresh for regular MTD partitions now. This means that an increasing number of bitflips caused by merely reading the data (read-disturbances) is detected and corrected by automatically re-writing the block.

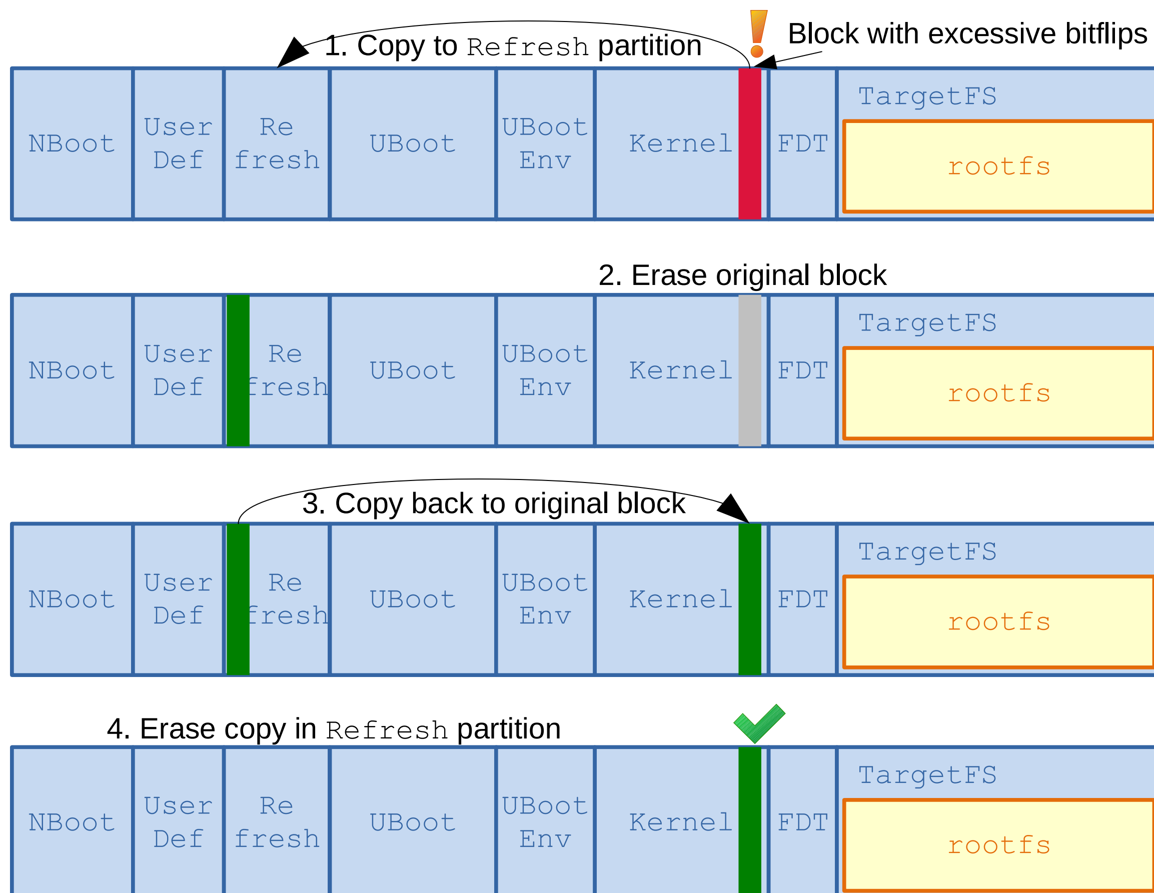


Figure 29: Block Refresh with intermediate safety copy

Much care is taken that no data is ever lost during this refresh procedure, even if there is a power failure while the refresh is in progress. The data of the original block is first copied to a reserved backup block in the `Refresh` partition, then the original block is erased and finally

the data is copied back to the original block and the backup copy is erased (see Figure 29). So data is not simply held in RAM, at any time there is at least one valid copy of the data in non-volatile memory, either in the original block or in the backup block. If the refresh process is interrupted, the next time it will either be continued or repeated from start, depending on the stage where the interruption happened. If the block data was only buffered in RAM, it would be lost in this case.

Because of this Block Refresh, data integrity of regular MTD partitions is better than ever before, and this is even true for the bootloaders NBoot and U-Boot. They also profit from this block refresh.

This means F&S has also increased the data safety for all data that is *not* handled by UBI. And because of this we still keep the Linux kernel image in an MTD partition in our standard configuration. We believe it is similar safe as with UBI, but this method does not have the penalty of a slower boot time. In our view this is the best compromise between speed and data safety.

6.4 The Install/Update/Recover Mechanism

Installing software on an empty board and updating software to a new version are common tasks when working with an embedded system. F&S has added an install, update and recover mechanism to U-Boot to automate these tasks as far as possible.

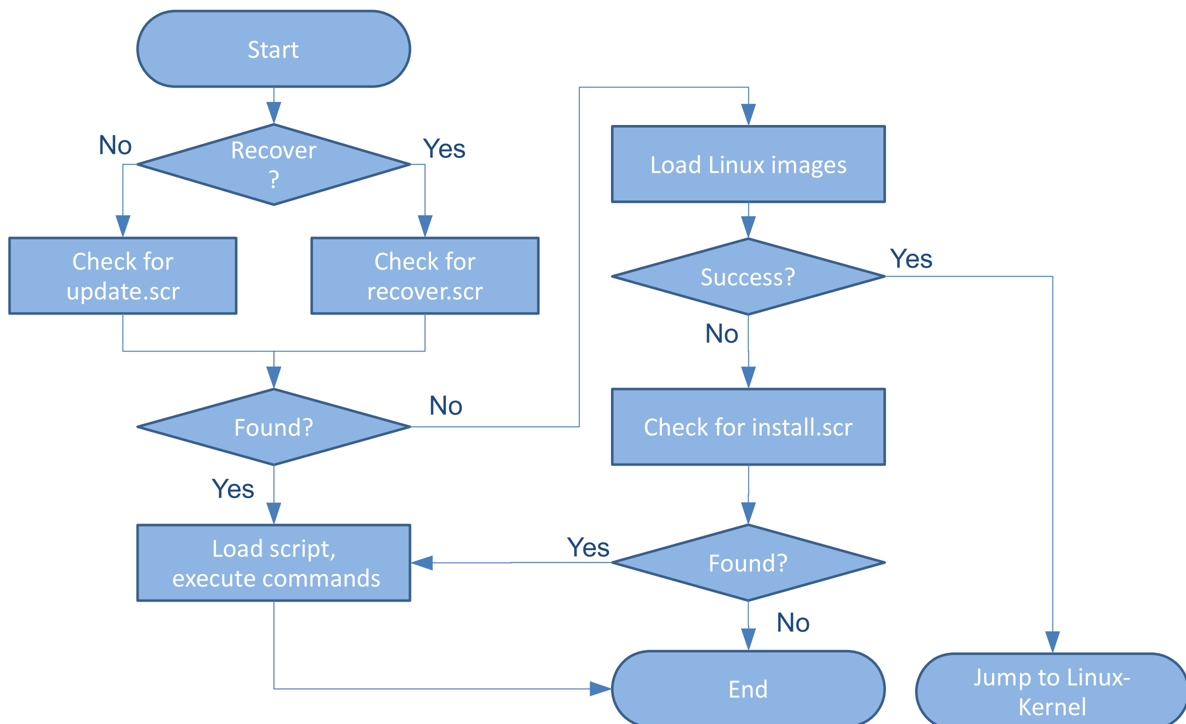


Figure 30: Generic Update/Install/Recover flow

How does it work? When U-Boot starts, it follows these steps.

1. It checks at several locations for the script file `update.scr`. If the file is found, it is executed. The file is typically used to do a system update. Alternatively it can look for a file `recover.scr` to do a system recovery.
2. If no such script is found, U-Boot tries to boot with variable `bootcmd`. This is the common case and constitutes the regular start of the Linux system.
3. If booting fails, U-Boot looks at several locations for the script file `install.scr`. If it is found, it is executed. The file is typically used to install Linux on an empty system.

The general flow of this mechanism is shown in Figure 30. It looks more complicated than it actually is. We will look at it step by step.

6.4.1 Regular Boot Process

First we will show what a regular boot process will look like. U-Boot looks for an update script, but it will not find any. So it will simply load the Linux kernel and jumps to it. That's all (see Figure 31).

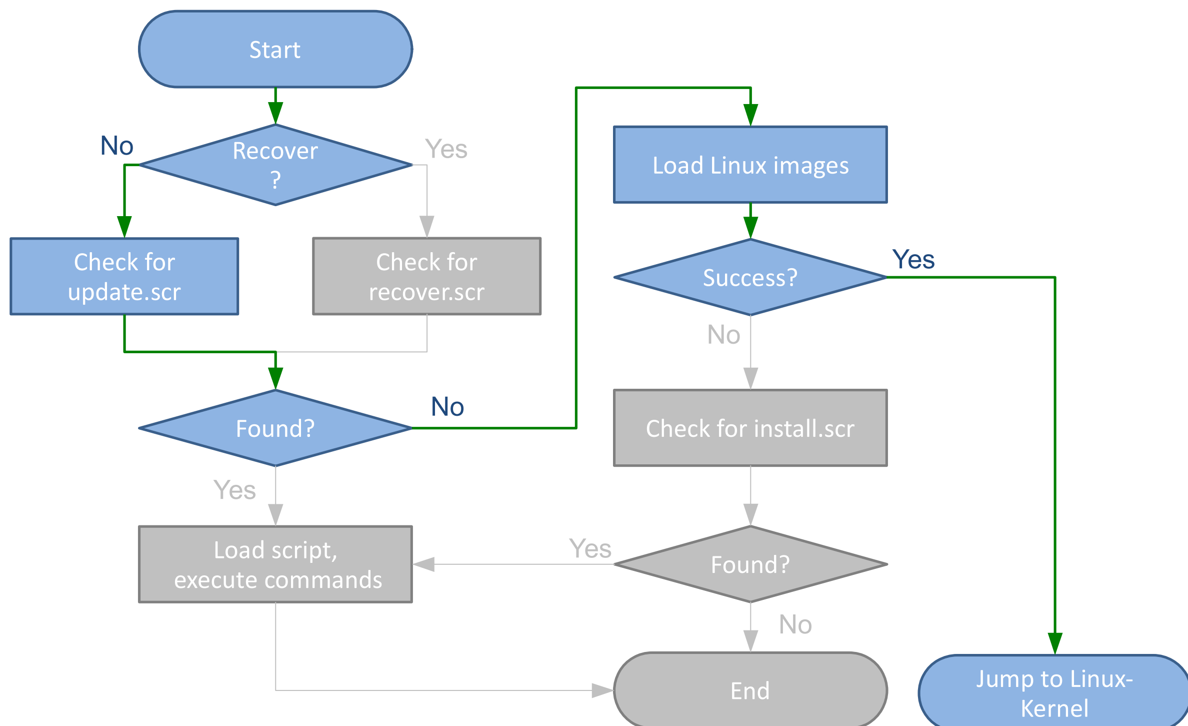


Figure 31: Regular Boot Process

6.4.2 Install Process

Apart from Starterkits that include a full set of software, F&S typically ships its boards with only the bootloaders NBoot and U-Boot installed. This means you need to install the final software yourself as part of your own production process. In fact, this is rather easy by using the so-called *Install Process*. You just have to make sure that an install script, typically called `install.scr`, is available at some predefined location, for example on a USB stick. We will

see later what different locations are possible. If the file is found, it is loaded and the U-Boot commands inside are executed (see Figure 32).

The install script itself is responsible for doing all the steps to perform the installation of the system software (kernel, device tree, root filesystem, etc.). U-Boot has no deeper knowledge of where to get any files or images and where to store them. It also does not know what to do if anything fails. All this stuff must be part of the script. This keeps the installation mechanism itself rather simple.

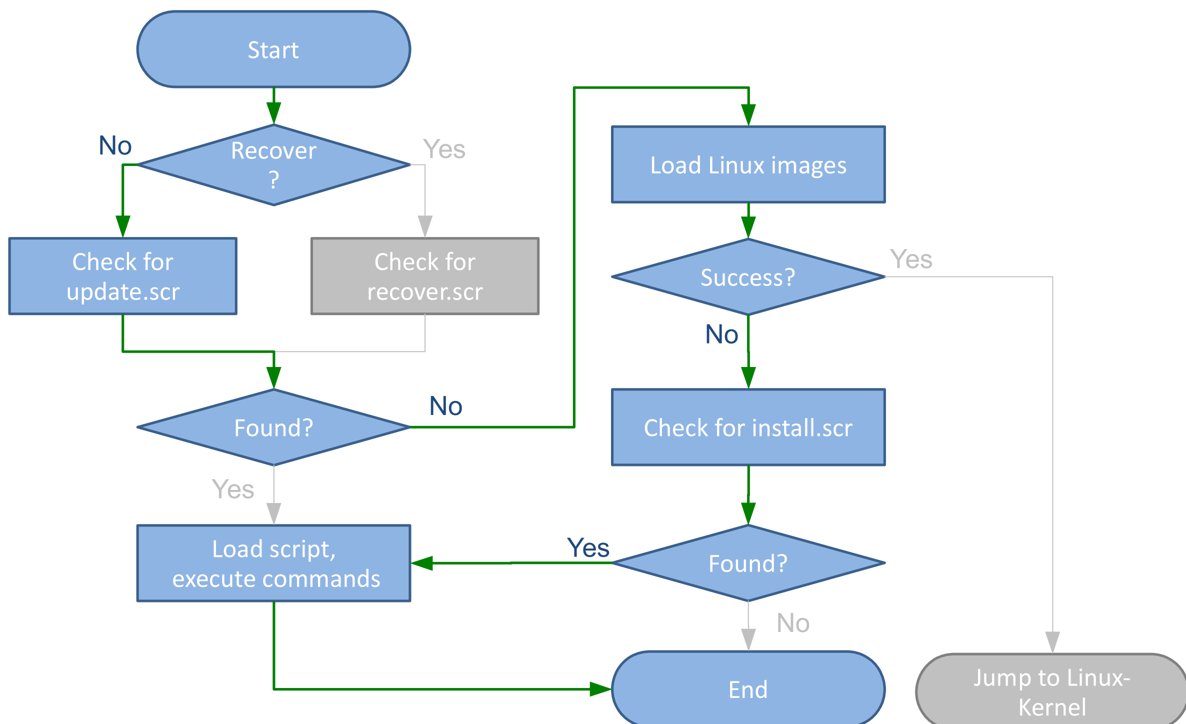


Figure 32: Install Process

After the installation is done, it is a good idea to restart the system with command `reset` as the last command of the script. The installation process may have updated U-Boot itself, including the U-Boot environment. If the board is completely rebooted this way, it makes sure that everything, including the Linux system, can actually be started successfully from now on. Because of the now valid system software, U-Boot will perform a regular boot process into Linux. There you can perform final high-level installation or configuration steps that may not be possible in U-Boot alone.

If there is no install script, or if the install script ends without starting the system in one way or the other, U-Boot simply stops at the command prompt and waits for commands.

6.4.3 Update Process

As we have seen, the Installation Process is only started if there is no valid Linux system. But if a system in the field should be updated, there is already a valid Linux system on the board. This is why U-Boot additionally looks for an update script *before* the Linux system is started. The script, which is typically called `update.scr`, is loaded to RAM and executed if found.

This is then called an *Update Process* (see Figure 33). Again the script itself is responsible for doing all the steps to perform the update and handle failures.

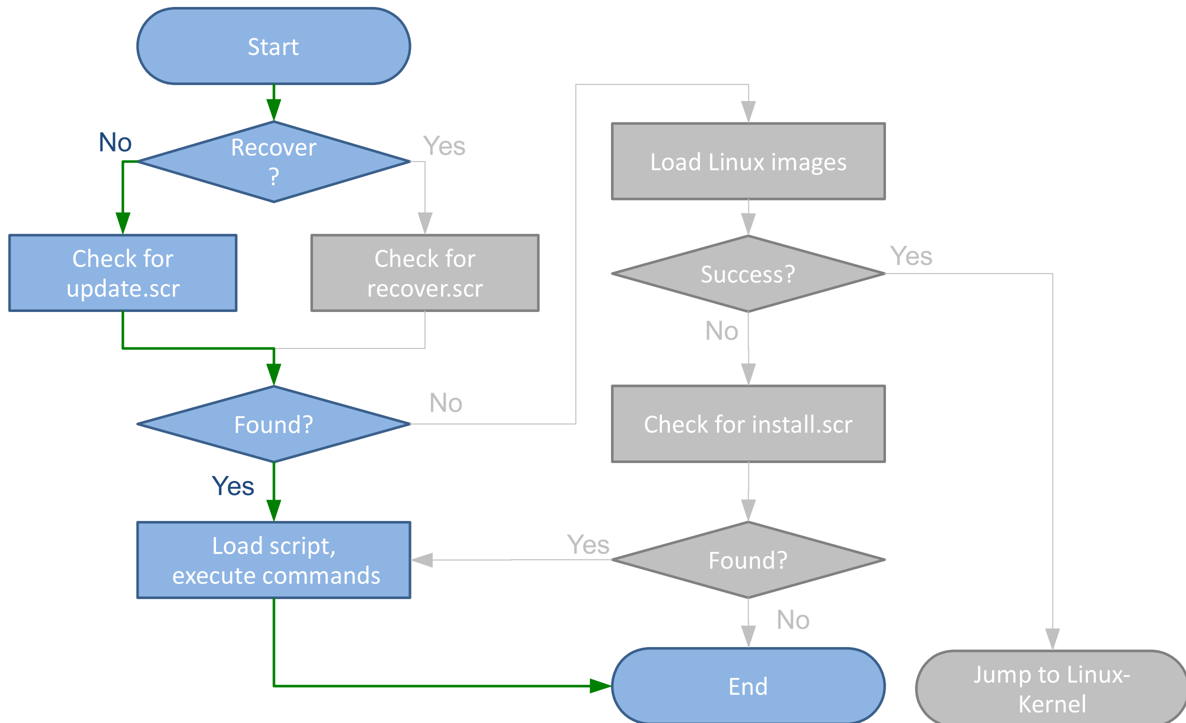


Figure 33: Update process

The difference between Install Process and Update Process is simply that U-Boot checks for the update script before the system start and for the install script only if the system start fails. Loading and executing the script itself is completely the same.

Remark

After the update process is done, the update script should either start the Linux system by calling command `boot`, or instruct the user to remove the update media and restart the system manually. It is *not* a good idea to simply execute the `reset` command to reboot the system. Because different to the Install Process, here the update script will be found again at the next start and the update process will repeat over and over.

6.4.4 Recover Process

There is yet a third update variant that we call *Recover Process*. This type assumes that you have installed some kind of fall-back system software. This is nothing that is there by itself, you have to actively prepare your devices before shipping so that this fall-back system is available. For example you can keep the originally shipped software version as a copy on the board. Then if something goes wrong, you can recover from the error by going back to the original software. The recover process then must copy the backup version to the working version (see Figure 34).

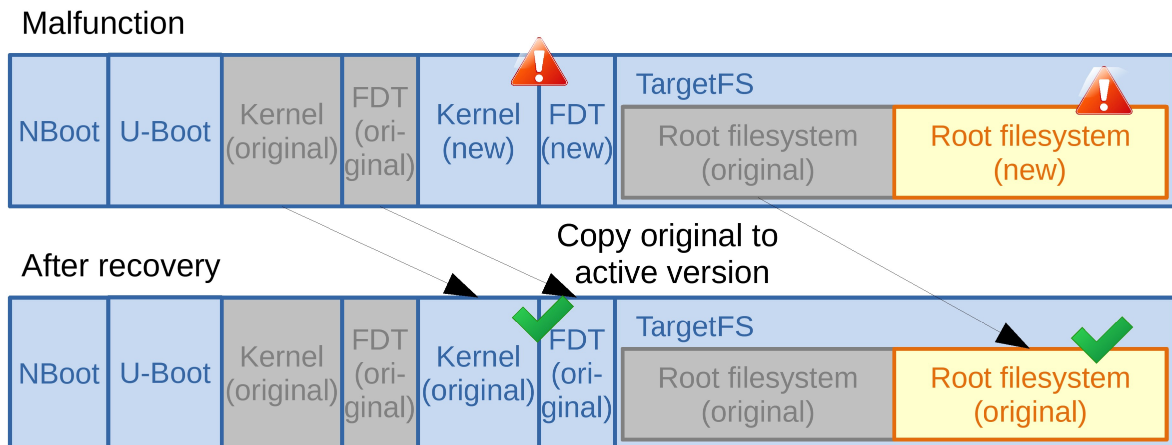


Figure 34: Recover by copying backup version to active version

Or you can use a kind of ping-pong activation. You always have an active version and an inactive version. The active version is the system that is running. An update however will install to the inactive version. If anything fails before the update is complete, the active version is still valid and can still be started. Only if the update process can be fully completed without error, then you switch the roles of the active and inactive versions. Then you start the newly installed version and the previous version gets inactive.

If, despite everything, something still goes wrong, then you can recover by simply switching back the roles and thus re-activating the previous version (see Figure 35).

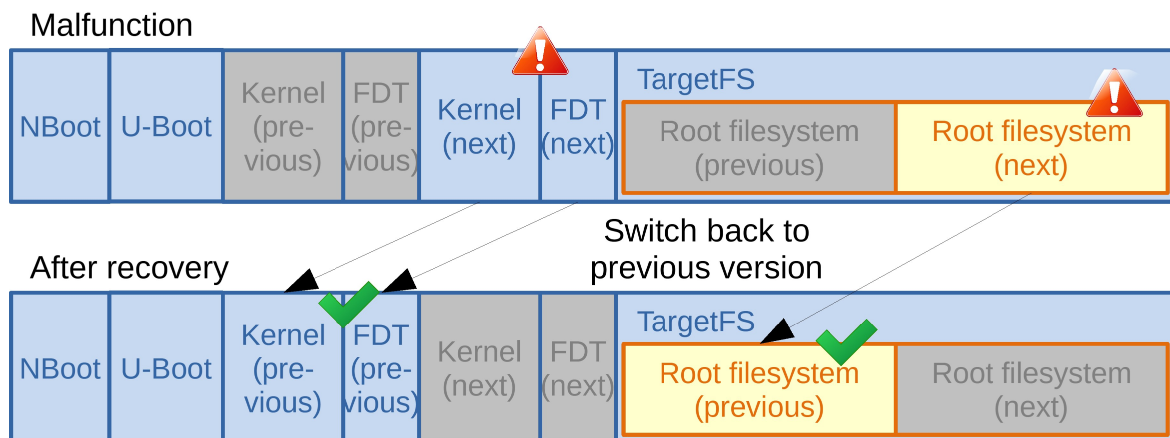


Figure 35: Recover by re-activating the previous version

Now you simply have to tell U-Boot to start the Recover Process. This may be implemented by a recessed button that can only be pressed with the help of a pen or a bent-open paper-clip, or by a key switch that can only be accessed by a privileged service person, or after opening the casing.

All these implementations have in common that in the end only a GPIO input pin changes state. So when U-Boot starts, it can also check for this GPIO pin. If the pin is in regular state, the normal check for update is done as shown above. But if the pin is in recover state, U-Boot looks for a recover script, typically called `recover.scr`, and executes it if found (see Figure 36). Again the recover script is responsible to do all steps required to recover the system to a working state.

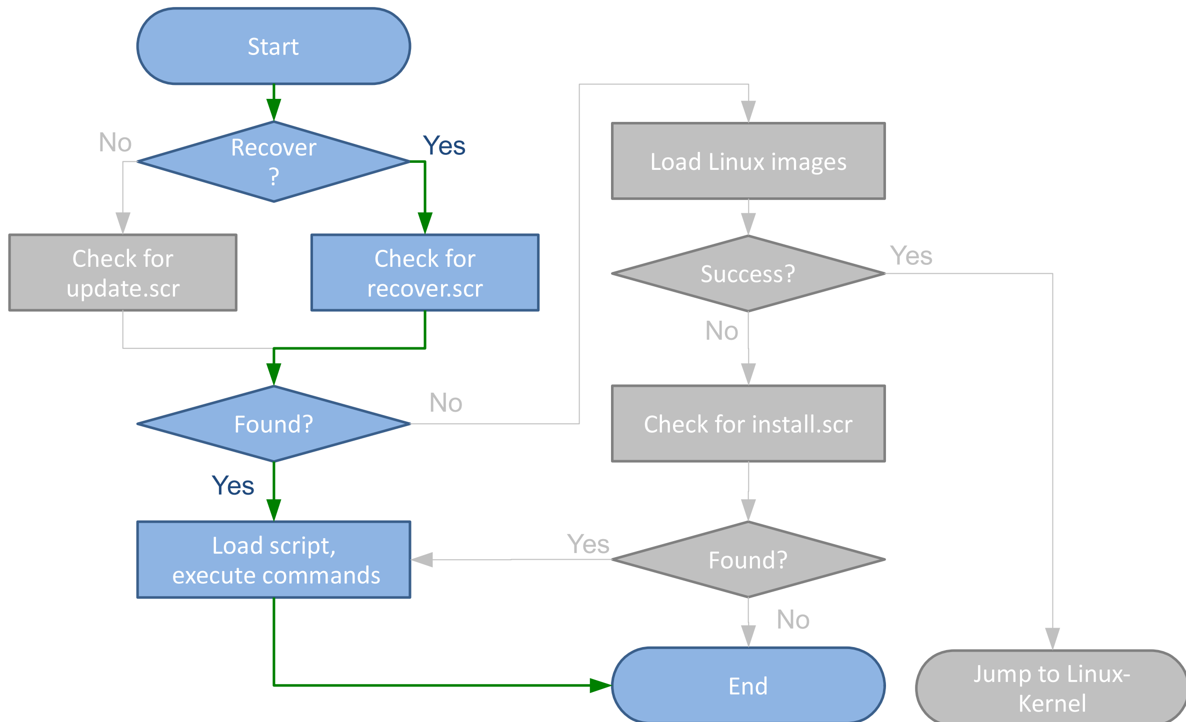


Figure 36: Recover process

6.4.5 Configuring the Install/Update/Recover Mechanism

In general, the Install/Update/Recover mechanism can be configured by setting several U-Boot environment variables. An overview is in Table 10.

Variable	Description	Action if not set
`recovergpio`	GPIO pin and signal level on this pin to use for recover detection	No recover check is done
`installscript`	Name of the install script	Use `install.scr`
`updatescript`	Name of the update script	Use `update.scr`
`recoverscript`	Name of the recover script	Use `recover.scr`
`installaddr`	RAM address to load install script to	Use `$loadaddr`
`updateaddr`	RAM address to load update script to	Use `$loadaddr`

Variable	Description	Action if not set
`recoveraddr`	RAM address to load recover script to	Use `$loadaddr`
`installcheck`	Locations where to look for install script (Default setting: `ram@<addr>,mmc,usb`)	No install is done
`updatecheck`	Locations where to look for update script (Default setting: `mmc,usb`)	No update is done
`recovercheck`	Locations where to look for recover script (Default setting: `ram@<addr>,mmc,usb`)	No recover is done

Table 10: Variables to configure Install/Update/Recover Mechanism

To define which GPIO pin is to be checked and whether to trigger the Recover Process in high or low level state, you have to set the environment variable `recovergpio`. Just give the GPIO number, followed by an optional dash `-` or underscore `_`, followed by `high` or `low`. Only the first letter of the last word is actually checked. `H` or `h` is high, everything else (including nothing at all) is low.

Examples

Start Recover Process if GPIO #123 is high level

```
setenv recovergpio 123_high
```

Start Recover Process if GPIO #65 is low level

```
setenv recovergpio 65-low
```

Start Recover Process if GPIO #13 is low level

```
setenv recovergpio 13
```

Start Recover Process if GPIO #31 (0x1f) is high level

```
setenv recovergpio 0x1fh
```

Remark

Some customer specific boards already handle the recover recognition in NBoot. Then U-Boot gets this information directly from NBoot and it is not necessary to set this variable.

If you want a different name for any of the scripts, just set the corresponding environment variable `installscript`, `updatescript` or `recoverscript` accordingly. For example if the update script should be named `my_update.uboot` instead of `update.scr`, then simply call:

```
setenv updatescript my_update.uboot
```

By default, the scripts are loaded to the default address given by variable `loadaddr`. However if this makes any problems, you can set a different load address by setting the corres-

ponding variable `installaddr`, `updateaddr` or `recoveraddr`. For example to load the recover script to address 0x85000000, execute the following command:

```
setenv recoveraddr 85000000
```

Finally the locations, where U-Boot should look for the scripts can be given in the variables `installcheck`, `updatecheck` and `recovercheck`. If such a variable is empty, then no check is done, effectively disabling this kind of update. By default, the install and recover scripts look at an architecture specific RAM address, at the first MMC port and at the first USB storage device. The update script only looks at the first MMC port and the first USB storage device.

Attention

When boards are shipped by F&S, install, update and recover are enabled. This is intended so that you can use these mechanisms to automatically install your own system software in your production. But do not forget to change the default settings afterwards to suit your own update policy. Especially if you do not want to enable any automatic update for your product in the field, then you have to clear (unset) the variables `installcheck`, `updatecheck` and `recovercheck`!

Table 11 shows all possible locations for script files and how the necessary parameters are given in the check variable. You can also give a list of locations, separated by commas, then U-Boot will look at all these locations one after the other. The first script that is found is loaded and executed.

Location	Description
`mmc[<dev_num>[:<part_num>]]`	An SD card or eMMC on any MMC port. If no device number (port) is given, use port 0, i.e. the first port. If no partition number is given, use 0, which either refers to the first partition of a device or the whole device if it does not contain a software partition table. Regular partition numbers are counted from 1.
`usb[<dev_num>[:<part_num>]]`	Any USB storage device. If no device number is given, use 0, i.e. the first storage device. If no partition number is given, use 0, which either refers to the first partition of a device or the whole device if it does not contain a software partition table. Regular partition numbers are counted from 1.
`nand[<dev_num>[:<part_num>]][+<offset>]`	Any MTD partition on NAND given by device number and partition number plus an optional offset.
`<part_name>[+<offset>]`	Any MTD partition on NAND given by the partition name plus an optional offset
`nand[<dev_num>[:<part_num>].ubi(<vol_name>)`	Any UBIFS in the volume with the given name in the UBI on top of the MTD partition given by device and

	partition number
`<part_name>.ubi(<vol_name>)`	Any UBIFS in the volume with the given name in the UBI on top of the MTD partition given by the partition name
`ram[@<addr>]`	Any RAM address. If no address is given, use the value of variable `loadaddr`.
`tftp`	Any TFTP server configured manually by the U-Boot environment variables
`dhcp`	Any TFTP server configured by a DHCP request
`nfs`	Any NFS server configured manually by the U-Boot environment variables

Table 11: Syntax for install/update/recover script locations

Examples

For update script, look at the first USB storage device only

```
setenv updatecheck usb
```

For install script, look at the second partition on the device on MMC port 2 and on the third partition on the second USB storage device (devices are counted from 0, so the second device has number 1). You can use blanks between the device name and the number and also after the comma, if you like.

```
setenv installcheck mmc2:2, usb1:3
```

For recover script, look at the UBI volume `backup` (on top of the MTD partition `TargetFS`), and if not found, also try a TFTP server that is configured in the U-Boot variables.

```
setenv recover TargetFS.ubi(backup),tftp
```

Note

It sounds tempting to directly specify a network server for updates, where the update script and the images are uploaded when a new version is available. However the update check will be done on every boot cycle of the board. If the board is not online, for example if the network cable is not plugged in, or if the server does not respond, then U-Boot will wait for several seconds before continuing the boot process. This will delay every regular boot considerably. This is not what you want.

Either do the download in Linux and store the files somewhere locally where U-Boot will find the update script at the next boot, or only enable the update check by changing the `updatecheck` variable when you are sure that there is actually a new version available.

6.4.6 The update Command

Sometimes you also want to trigger the Install, Update or Recover Process manually, for example when you want to test your scripts. This can be done by using the `update` command. If you just use `update`, it will obviously start an Update Process. If you append `.install` or `.recover`, it will start an Install or Recover Process. You can give the location where to look for the appropriate script as first argument, the name of the script as second argument and the load address as third argument. This will override the values of the environment variables.

Please be careful when entering the location. If you add blanks like in `mmc 0`, then you have to enclose the whole argument in double quotes, or otherwise the 0 would be interpreted as the next argument, which is the script name. This is why you can omit all blanks in the location string so that you also do not need any double quotes.

Examples

Do an Update Process with standard settings.

```
update
```

Start a Recover Process from MMC port 1, and optionally via TFTP. Look for the script file `test.scr`. Load it to address 0x83000000.

```
update.recover mmc1,tftp test.scr 83000000
```

Show the help for command `update` with a detailed description of the configuration options.

```
help update
```

6.4.7 Creating an Appropriate U-Boot Script Image

To prepare a script to be used with any of the above methods, you need to create a text file with all the U-Boot commands that should be executed. But unfortunately this text file can not be executed by U-Boot directly, you first need to convert it to a U-Boot script image.

A simple update script for nand could look like this:

```
# Load kernel image and store in partition Kernel
load ${updatedev} . zImage-${arch}
nand erase.part Kernel
nand write . Kernel ${filesize}

# Load device tree blob and store in partition FDT
load ${updatedev} . ${platform}.dtb
nand erase.part FDT
nand write . FDT ${filesize}

# Create UBI with volume rootfs on partition TargetFS
nand erase.part TargetFS
ubi part TargetFS
ubi create rootfs
```

```
# Load root filesystem and store in UBI volume rootfs
load ${updatedev} . rootfs-${arch}.ubifs
ubi write . rootfs ${filesize}

# Set default configuration: kernel and fdt from NAND, rootfs from ubifs
run .kernel_nand
run .fdt_nand
run .rootfs_ubifs

# Remove update variable and save environment
setenv updatedev
saveenv

# Done
echo "Installation complete"
echo
echo "Please set/verify ethernet address(es) now and call saveenv"
```

As you can see, we are using variables `arch` for the architecture and `platform` for the platform name. These variables are automatically set by our U-Boot and allow us to use the same install script for almost all our platforms.

In addition, during an update process, the variable `updatedev` is set to the location, where the script itself was loaded from. So for example if the script was loaded from MMC port 0, then `updatedev` will be set to `mmc 0`. So we need not know beforehand when writing the script which kind of update device is actually used later. Unfortunately this is only possible for MMC and USB sources. As you can see, we need to take care and delete this variable before saving the environment, or else the variable would always be set.

> **Note**
>
> Please make sure that you use Linux line endings with `<LF>` only, not Windows line endings with `<CR><LF>`. Using the wrong line endings may cause each command to be executed twice, similar to the U-Boot command line where just pressing *Return* typically repeats the previous command.

Starting with release fsimx6-B2021.10 and fsimx8mn-Y2021.10, the install script is built with Buildroot and Yocto.

Buildroot

For Buildroot, the script source is located at

```
board/f+s/common/install.txt
```

A different source can be used by setting the INSTALL_TXT_PATH variable in the boards finalscript_* file to the new source location.

The output install.scr can be found at

```
output/images/install.scr
```

Yocto

The source file install.txt can be found at

```
sources/meta-fus/recipes-config/images/files/install.txt
```

in the Yocto directory.

The output install.scr can be found at

```
tmp/deploy/images/fsimx7ulp/install.scr
```

in the Yocto-Build directory.

Older releases:

The text version `install.txt` of our standard F&S `install.scr` file can be found in the `sources` directory of our releases. It has the following content.

The conversion of the text file to a U-Boot script image is done with the following command on your development PC:

```
mkimage -A arm -O u-boot -T script C none -n "F&S install script" \
        -d install.txt install.scr
```

`mkimage` is a tool that is built when compiling U-Boot. So you will find it in the `tools` subdirectory of U-Boot. We also have stored a copy in the `toolchain` directory of our releases. The command adds a description, a CRC checksum and some other information to the file. The file could also be compressed if necessary, but we do not do this because the script is very short and compression does not make much sense. You can do this if you want to hide the content. By default, the content can be easily extracted from the script image.

The result of the command is the file `install.scr` that can be used in U-Boot.

6.5 Linux Boot Strategies

When U-Boot boots the system, it automatically executes the commands stored in environment variable `bootcmd` and passes the contents of variable `bootargs` as command line to the Linux kernel. By modifying the contents of these variables, the boot process can be changed. For example by using a different boot command in `bootcmd`, the kernel can be loaded from a different place. And by using a different `bootargs` content, the Linux system can load the root filesystem from a different place or can do other things differently.

When working with an Embedded System, there are different situations that require different boot behaviour. For example while developing the application software, the board is usually connected permanently to the LAN and it may be convenient to load the ever changing root filesystem via NFS and have write access to it. Later when development is over, the system should be switched to a self contained environment where the root filesystem is loaded from NAND flash and is read-only.

To cope with these different situations, F&S has developed different *boot strategies*. You can easily switch between these strategies by running commands that are stored in environment variables.

The following settings can be modified independently from each other.

- From where to load the kernel (nand, ubi, ubifs, tftp, nfs, mmc, usb).
- From where to load the device tree (none, nand, ubi, ubifs, tftp, nfs, mmc, usb).
- From where to load the root filesystem in Linux (ubifs, nfs, mmc, usb).
- How to mount the root filesystem in Linux (ro, rw).
- Where to show the console in Linux (none, serial, display).
- Where to start the login prompt in Linux (none, serial, display).
- How to start the network in Linux (off, manual, dhcp).
- The name of the init program (init, linuxrc)

To understand how this actually works, we have to take a look at the `bootcmd` variable. The contents of this variable are executed when the system boots. On F&S boards it has the following content.

```
run set_bootargs; run kernel; run fdt
```

So it first runs the content of variable `set_bootargs`. As the name already suggests, this will handle all settings of the `bootargs` variable which is used as kernel command line later when the Linux system starts. Then it runs the contents of variable `kernel` which loads the kernel image from different sources into RAM. And finally it runs the contents of variable `fdt` which loads the device tree and executes the kernel image from RAM

So let's have a look at the `set_bootargs` variable, too

```
setenv bootargs ${console} ${login} ${mtdparts} ${network} ${rootfs}
                ${mode} ${init} ${extra}
```

Here the contents of the `bootargs` variable (= command line) is assembled by combining several other variables. So basically it is possible to change the command line content by modifying the content of one or more of these variables. And it is possible to change the kernel source by modifying the `kernel` variable.

We have prepared a set of predefined environment variables that will allow setting each of these parts to meaningful values. So there are several settings to change the `console` variable, several settings to change the `login` variable, and so on. This results in quite a lot of environment variables, which is why we have chosen to hide them so that they do not overload a common printenv listing. This means all these variable names start with a `.` (dot) and you can only see them with

```
printenv -a
```

The following sections will show the predefined variables. Of course you can also set different values or add more variables if the predefined settings do not suit your needs.

6.5.1 Kernel Settings

These settings listed in Table 12 modify the `kernel` variable and tell U-Boot where to load the kernel image from. If a filename is required, it is taken from variable `bootfile`. Other settings may also be taken from other variables, for example the network settings.

Kernel	Description
`.kernel_mmc`	Tell U-Boot to load kernel from MMC (SD card)
`.kernel_usb`	Tell U-Boot to load kernel from USB drive
`.kernel_nfs`	Tell U-Boot to load kernel via network with NFS protocol
`.kernel_tftp`	Tell U-Boot to load kernel via network with TFTP protocol
`.kernel_nand`	Tell U-Boot to load kernel from MTD partition `Kernel`
`.kernel_ubi`	Tell U-Boot to load kernel from the UBI volume named `kernel`
`.kernel_ubifs`	Tell U-Boot to load kernel from the root filesystem, subdirectory `/boot`; the root filesystem is assumed to be UBIFS formated and located in the UBI volume named `rootfs`.

Table 12: Variables to set kernel source

6.5.2 Device Tree Settings

The settings listed in Table 13 modify the `fdt` variable and tell U-Boot where to load the device tree from and how to start the kernel. If a filename is required, it is taken from variable `bootfdt`.

Variable	Description
`.fdt_none`	Tell U-Boot to load no device tree (for example if the device tree is already appended to the kernel image)
`.fdt_mmc`	Tell U-Boot to load device tree from MMC (SD card)
`.fdt_usb`	Tell U-Boot to load device tree from USB drive
`.fdt_nfs`	Tell U-Boot to load device tree via network with NFS protocol
`.fdt_tftp`	Tell U-Boot to load device tree via network with TFTP protocol
`.fdt_nand`	Tell U-Boot to load device tree from MTD partition `FDT`
`.fdt_ubi`	Tell U-Boot to load device tree from the UBI volume `fdt`
`.fdt_ubifs`	Tell U-Boot to load device tree from the root filesystem, subdirectory `/boot`; the root filesystem is assumed to be UBIFS formated and located in the UBI volume named `rootfs`.

Table 13: Variables to set device tree source

6.5.3 Rootfs Settings

The settings listed in Table 14 modify the `rootfs` variable as part of the `bootargs` variable and tell the kernel from where to load the root filesystem.

Variable	Description
`.rootfs_mmc`	Tell kernel to load root filesystem from MMC (SD card)
`.rootfs_usb`	Tell kernel to load root filesystem from a USB drive
`.rootfs_nfs`	Tell kernel to load root filesystem via network (NFS)
`.rootfs_ubifs`	Tell kernel to load root filesystem from a UBIFS filesystem located in a UBI volume

Table 14: Variables to set rootfs source

6.5.4 Mode Settings

The settings listed in Table 15 modify the `mode` variable as part of the `bootargs` variable and tell the kernel whether to allow write access to the root filesystem or not.

Variable	Description
`.mode_ro`	Tell kernel to mount root filesystem in read-only mode
`.mode_rw`	Tell kernel to mount root filesystem in read-write mode

Table 15: Variables to set set /deny write access for root filesystem

6.5.5 Console Settings

The settings listed in Table 16 modify the `console` variable as part of the `bootargs` variable and tell the kernel where to send console messages to.

Variable	Description
`.console_none`	Tell kernel to not show any boot messages at all
`.console_serial`	Tell kernel to send boot messages to the serial debug port
`.console_display`	Tell kernel to send boot messages to the display

Table 16: Variables to set console output

6.5.6 Login Settings

The settings listed in Table 17 modify the `login` variable as part of the `bootargs` variable and tell the kernel where to start a login prompt (`getty`).

Variable	Description
`.login_none`	Tell kernel to not start a login at all
`.login_serial`	Tell kernel to start login on the serial debug port
`.login_display`	Tell kernel to start login on the display

Table 17: Variables to set login prompt origin

6.5.7 Network Settings

The settings listed in Table 18 modify the `network` variable as part of the `bootargs` variable and tell the kernel how to start the Ethernet interface.

Variable	Description
`.network_off`	Tell kernel to not activate Ethernet; it must be started manually or by the network infrastructure in the root filesystem
`.network_on`	Tell kernel to start Ethernet with same settings as in U-Boot
`.network_dhcp`	Tell kernel to start Ethernet with a DHCP request

Table 18: Variables to set network activation

Note

When the network is enabled with this setting, this means that the kernel itself will start the network. This is very useful (and even necessary) if the root filesystem is accessed via network (e.g. when using `run .rootfs_nfs`), but may be undesired in most other cases. For example if the network cable is not plugged in, or if no DHCP server is responding, then the kernel will block the boot process indefinitely. So this is usually not what you want.

The regular way is to keep this network value here at “off” and let the network infrastructure in the root filesystem do the job of starting the network. This happens later, after the kernel has finished booting, has mounted the root filesystem and has passed the control on to the `init` program. Automatic activation of the network can then be achieved by modifying the settings in `/etc/network/interfaces` or similar.

6.5.8 Init Settings

The settings listed in Table 19 modify the `init` variable as part of the `bootargs` variable and tell the kernel whether which file should be used for the `init` process.

Variable	Description
`.init_init`	Tell kernel to use file `init`
`.init_linuxrc`	Tell kernel to use file `linuxrc`

Table 19: Variables to set init process file

6.5.9 Extra Settings

The `extra` variable as part of the `bootargs` variable allows to add arbitrary content to the kernel command line. There are no predefined variables, just set `extra` to the required content. For example to start Linux without showing the Tux logo on the display, use this:

```
setenv extra logo.nologo
```

6.5.10 Boot Strategy Examples

Boot kernel and device tree via TFTP, and root filesystem via NFS from the server defined in U-Boot:

```
run .kernel_tftp
run .fdt_tftp
run .rootfs_nfs
run .network_on
```

Boot kernel from MTD partition `Kernel`, device tree from MTD partition `FDT`, and the root filesystem from the UBIFS image in UBI volume `rootfs`. Do not start the network in Linux automatically. This is the default setting for F&S boards, for example in the Starterkits.

```
run .kernel_nand
run .fdt_nand
run .rootfs_ubifs
run .network_off
```

Boot kernel from the file `/boot/my_yummy_kernel` from the root filesystem (located in default position, i.e. in UBI volume `rootfs`). Use device tree `my_yummy_fdt` from the same location.

```
setenv bootfile my_yummy_kernel
setenv bootfdt my_yummy_fdt
run .kernel_ubifs
run .fdt_ubifs
```

Start the login prompt on the display instead of on the serial line:

```
run .login_display
```

Of course you usually need to call `saveenv` to save these settings permanently. Otherwise they will only be used for an immediate boot process started with the `boot` command. For example if you use the following command sequence, the system will start once in read-write mode and then in the future in read-only mode again.

```
run .mode_ro
saveenv
run .mode_rw
boot
```

This can be useful because if `openssh` is active in the Linux system, it tries to generate some cryptographic keys when started for the very first time. But if the root filesystem is mounted read-only, this is not possible. Then the key generation is skipped and `openssh` is not usable. But if it is started in read-write mode once, the keys can be generated and `openssh` is working correctly from then on. Just remember to safely shut down the Linux system in this one case, for example by calling command `halt`. Any subsequent runs can and will be done read-only again, because the saved setting of `mode` is read-only.

Sometimes running one of the predefined variables does not fit your needs. For example there is no predefined variable to send the console output to `/dev/ttymxc2` with 19200 baud. Then simply set the appropriate `bootargs` component directly, which is the `console` variable in this case.

```
setenv console console=/dev/ttymxc2,19200
```

6.6 NAND Layout Strategies

A great concern when using NAND flash is the fear of losing any data due to ageing processes. We have seen in chapter 3.8.1 that NAND uses ECC (Error Correction Codes) to be able to detect and correct a small number of bad or flipped bits. And that block refresh can be used to renew bits that have flipped due to discharging effects, for example after many million reads. Block refresh is especially important for read-only data that would never be rewritten (and thus renewed) otherwise.

But of course this also needs support from the software side. The Unsorted Block Image concept, or UBI for short, was especially designed to handle exactly these problems. ECC is used to handle bitflips and it provides automatic block refresh if the number of bitflips in a page of a block reaches a specific threshold. This is triggered either when regularly reading any data with many bitflips, or by a low-priority background task that checks all blocks from time to time for bitflips.

But UBI does even more. UBI provides wear-levelling. It keeps an erase counter for each block, and if a block has a high erase count, UBI tries to use an other block for this data. It even moves data of blocks with lower erase count so that these blocks can be used for erasing and writing new data, too. By doing this, UBI keeps all blocks of the UBI region at a similar erase count.

Of course this works best if the wear-levelling can use as many NAND blocks as possible. So having one big UBI works better than two smaller UBIs. And of course any extra MTD partitions, like `NBoot`, `UBoot`, `UbootEnv`, `Kernel` and `FDT` are not covered by this mechanism because they are not part of the UBI region.

In the past this was the reason why we introduced the possibility to move partitions `Kernel` and `FDT` to the UBI region so that they could participate in the UBI wear-levelling. For other partitions this is not possible. Partition `NBoot` is accessed by the ROM loader of the SoC and

Partition `UBoot` is accessed by `NBoot`. They both can not access UBI and therefore these partitions must stay separate MTD partitions.

However moving these images to UBI also has a disadvantage. To access the UBI region, U-Boot needs to scan the whole NAND flash to get enough information of where all data is located. And if these files are stored in the root filesystem, U-Boot actually needs to mount the root filesystem, which takes even longer. So moving these images to UBI results in a considerably slower boot time. This is why we keep kernel and device tree in separate MTD partitions. This is simply the fastest mode. And since F&S has added block refresh for arbitrary NAND data to NBoot and U-Boot, this is also no longer as important regarding data integrity as before. But we leave the choice to the customer.

Setting a different NAND layout works similar to the boot strategies. We have again some predefined variables that switch the appropriate setting if they are run. One set of variables handles the MTD partition layout (Table 20), one handles the UBI volume layout (Table 21). The difference is, that running these variables actually creates the set of MTD partitions or UBI volumes. So do not run them more than once.

Variable	Description
`.mtdparts_std`	Create separate MTD partitions `Kernel` and `FDT` for the Linux kernel and the device tree
`.mtdparts_ubionly`	Do not create MTD partitions `Kernel` and `FDT`, both kernel and root filesystem are supposed to be stored in the UBI located on top of `TargetFS`

Table 20: Variables to define the MTD partition table

Variable	Description
`.ubivol_std`	Only create the `rootfs` volume in the UBI on top of `TargetFS`, Linux kernel and device tree are supposed to be stored in the root filesystem
`.ubivol_ubi`	Create `kernel`, `fdt` and `rootfs` volumes in the UBI on top of `TargetFS`, i.e. kernel and device tree go to separate volumes

Table 21: Variables for UBI volume creation

The next sub-sections will show in detail how to set up three different strategies.

Note

The following sections modify the size of the MTD partition `TargetFS`, that is the base for the UBI. Of course we have to erase `TargetFS` in this case or UBI will get completely confused. But after having created the UBI once, you should never erase it again by erasing the contents of `TargetFS`. Erasing destroys the wear-levelling history that UBI builds over time, making the whole effort more or less useless. You can write to UBI volumes, you can even remove UBI volumes and create them again. That is all OK and UBI will

take care of all required wear-levelling information. But never erase the underlying `TargetFS` itself or you will lose all wear-levelling information.

6.6.1 Linux Kernel In MTD partition

Here the kernel and the device tree are located in separate MTD partitions (see Figure 37). They do not participate in UBI's wear-levelling but block refresh is done by U-Boot. Loading is very fast. This is how the system is set up by default.

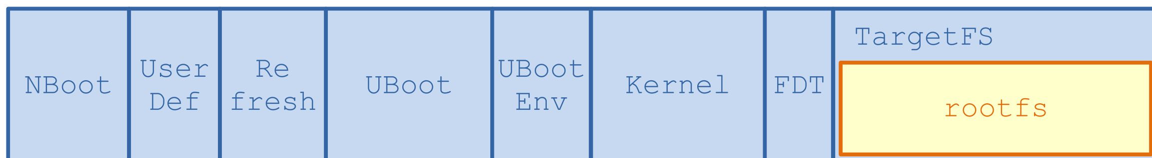


Figure 37: Linux kernel and device tree as separate MTD partitions

To prepare the system in this way, use the following commands.

```
run .mtdparts_std
nand erase.part TargetFS
run .ubivol_std
run .kernel_nand
run .fdt_nand
saveenv
```

To save the downloaded kernel and device tree images, use the `nand` command.

```
<download kernel image, e.g. with tftp>
nand erase.part Kernel
nand write $loadaddr Kernel $filesize

<download device tree image, e.g. with tftp>
nand erase.part FDT
nand write $loadaddr FDT $filesize
```

6.6.2 Linux Kernel In Raw UBI Volume

Here the kernel and the device tree are located in separate UBI volumes in the UBI region (see Figure 38). Block refresh is done by UBI and the images participate in UBI wear-levelling. Loading speed is medium, because UBI must be scanned in U-Boot.

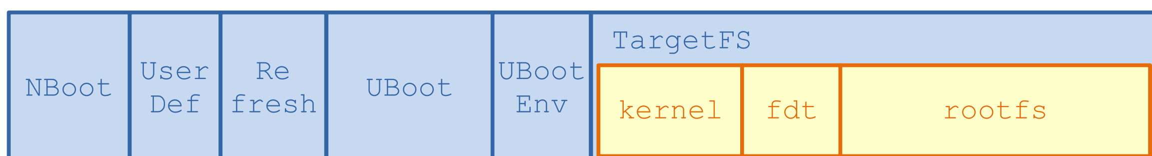


Figure 38: Linux kernel and device tree in separate UBI volumes

To prepare the system in this way, use the following commands:

```
run .mtdparts_ubionly
nand erase.part TargetFS
run .ubivol_ubi
run .kernel_ubi
run .fdt_ubi
saveenv
```

To save the downloaded kernel and device tree images, use the `ubi` command. Please note the different spelling of the volume names. Volume names are all lower-case while MTD partitions also have upper-case characters. This is done on purpose so that it is not possible to inadvertently use command `nand` with volume names and command `ubi` with MTD partition names.

```
<download image, e.g. with tftp>
ubi part TargetFS
ubi write $loadaddr kernel $filesize

<download device tree image, e.g. with tftp>
ubi write $loadaddr fdt $filesize
```

6.6.3 Linux Kernel In Root Filesystem

Here the kernel and the device tree are located as files inside of the root filesystem in the UBI region (see Figure 39). Block refresh is done by UBI and the images participate in UBI wear-levelling. Loading speed is low because UBI must be scanned and also the root filesystem must be mounted in U-Boot.

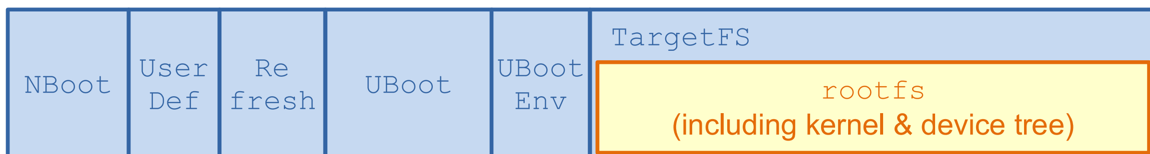


Figure 39: Linux kernel and device tree are simlpe files in the root filesystem

To prepare the system in this way, use the following commands:

```
run .mtdparts_ubionly
nand erase.part TargetFS
run .ubivol_std
run .kernel_ubifs
run .fdt_ubifs
saveenv
```

In this case the kernel and the device tree are part of the root filesystem, so they are written together with the other root filesystem files as part of the root filesystem image to the `rootfs` UBI volume. There is no way to write the files separately within U-Boot because U-Boot's implementation of UBIFS has no write support.

6.7 Configure eMMC For Enhanced Mode

As seen in Chapter 3.8.2, an eMMC has four predefined hardware partitions: `User`, `Boot1`, `Boot2` and `RPMB`. Furthermore up to four memory areas can be split off from the `User` space as separate Generic Partitions `GP1` to `GP4`. This results in up to eight hardware partitions (see Table 22).

Partition ID	Name	Meaning	Enhanced Mode
`0`	`User`	User Space	Optional (one continuous region)
`1`	`Boot1`	Boot information 1	Always
`2`	`Boot2`	Boot information 2	Always
`3`	`RPMB`	Security information	Always
`4`	`GP1`	Generic Partition 1	Optional (whole partition)
`5`	`GP2`	Generic Partition 2	Optional (whole partition)
`6`	`GP3`	Generic Partition 3	Optional (whole partition)
`7`	`GP4`	Generic Partition 4	Optional (whole partition)

Table 22: Hardware partitions on eMMC and Enhanced Mode

Most eMMC uses MLC, TLC or even QLC flash memory. This means that two, three or even four bits are stored per memory cell, resulting in a reduced data retention time. To increase data integrity, eMMC has the option to switch every hardware partition and one continuous region of the User Space to a so-called *Enhanced Mode*. Here data is stored with less states per cell, resulting in better data retention, but at the cost of available memory. This feature is often called Pseudo-SLC mode, and after activation, you only have half the amount of memory available. Partitions `Boot1`, `Boot2` and `RPMB` are already set to Enhanced Mode by the manufacturer, but all other partitions can be switched to Enhanced Mode, too, if desired.

Attention

This is a write-once option. Once configured, it remains forever and can not be undone. So be careful when doing this or you may end up with an unrecoverably misconfigured device.

Partition setup and Enhanced Mode activation have to be done in one single configuration step. U-Boot has a command `mmc hwpartition` for doing this. However the original version was very inflexible and needed absolute size values for all arguments. It was therefore nearly useless for an automatic configuration where the specific size of the memory is not known beforehand. It also used decimal numbers instead of hex numbers, which is very unusual for U-Boot, where numbers are typically hex numbers.

Let's show an example.

```
# mmc list
FSL_SDHC: 0
FSL_SDHC: 1
FSL_SDHC: 2
```

The eMMC memory always uses the highest MMC port number. So we first need to switch to port 2.

```
# mmc dev 2
switch to partitions #0, OK
mmc2(part 0) is current device
```

The output with the hardware partition 0 in parenthesis confirms that this is an eMMC and we are in the `User` partition.

```
# mmc info
Device: FSL_SDHC
Manufacturer ID: fe
OEM: 14e
Name: MMC04
Bus Speed: 52000000
Mode : MMC High Speed (52MHz)
Rd Block Len: 512
MMC version 4.4.1
High Capacity: Yes
Capacity: 3.6 GiB
Bus Width: 8-bit
Erase Group Size: 4 MiB
HC WP Group Size: 4 MiB
User Capacity: 3.6 GiB
Boot Capacity: 2 MiB ENH
RPMB Capacity: 128 KiB ENH
```

The MMC information shows us 3.6 GiB of `User` area that can be switched to Enhanced Mode. It also shows that each of the `Boot` partitions is 2 MiB and the `RPMB` partition is 128 KiB. These partitions are already in Enhanced Mode as you can see from the string `ENH` at the end of the last two lines. This setting was done by the manufacturer.

Entry `HC WP Group Size` shows the unit size that can be modified. When defining Generic Partitions or when defining a memory region in the User Space to switch to Enhanced Mode, they have to be multiples of this unit size, which is 4 MiB here.

Command `mmc hwpartition` needs exact values counted in sectors. So we should determine the number of available sectors first. 3.6 GiB is not accurate enough. Unfortunately there is no command to return this value, so we use a trick. We try to load data from a very high sector number that definitely does not exist on the device. This triggers an error message and the error message shows the maximum possible value. This is exactly the number that we need.

```
# mmc read $loadaddr 0x10000000 1
```

```
MMC read: dev # 1, block # 268435456, count 1 ... MMC: block num-
ber 0x10000001 exceeds max(0x728000)

0 blocks read: ERROR
```

This means we have 0x728000 sectors available, which is 7503872 in decimal. After switching to Enhanced Mode, only half the size is left, which is 3751936 sectors. So this is the maximum size that we can define.

We need to give a starting offset and the size of the area of the User Space that should be switched to Enhanced Mode. Both values can be increased or decreased in steps of 4 MiB, or 8192 sectors. The most common case would be to convert the whole User Space. So this is from sector 0, and the whole 3751936 sectors.

Nonetheless let us do a check first. By giving `check` as last argument, the following command does only verify the validity of the given values.

```
# mmc hwpartition user enh 0 3751936 check

Partition configuration:
        User Enhanced Start: 0 Bytes
        User Enhanced Size: 1.8 GiB
        No GP1 partition
        No GP2 partition
        No GP3 partition
        No GP4 partition
```

If there are no errors, you can set the values into the hardware by giving `set` as last argument. The output is rather similar.

```
# mmc hwpartition user enh 0 3751936 set

Partition configuration:
        User Enhanced Start: 0 Bytes
        User Enhanced Size: 1.8 GiB
        No GP1 partition
        No GP2 partition
        No GP3 partition
        No GP4 partition
```

As a final step, this setting must be confirmed once more by giving `complete` as last argument. Be careful, after this step there is no way back.

```
# mmc hwpartition user enh 0 3751936 complete

Partition configuration:
        User Enhanced Start: 0 Bytes
        User Enhanced Size: 1.8 GiB
        No GP1 partition
        No GP2 partition
        No GP3 partition
        No GP4 partition
Partitioning successful, power-cycle to make effective
```

As the output instructs us, we have to switch the power off and on again. Now the eMMC memory will come up with the new settings. If you query the MMC information again, it will show the following now:

```
# mmc dev 2
switch to partitions #0, OK
mmc2(part 0) is current device
# mmc info
Device: FSL_SDHC
Manufacturer ID: fe
OEM: 14e
Name: MMC04
Bus Speed: 52000000
Mode : MMC High Speed (52MHz)
Rd Block Len: 512
MMC version 4.4.1
High Capacity: Yes
Capacity: 1.8 GiB
Bus Width: 8-bit
Erase Group Size: 4 MiB
HC WP Group Size: 4 MiB
User Capacity: 1.8 GiB ENH
User Enhanced Start: 0 Bytes
User Enhanced Size: 1.8 GiB
Boot Capacity: 2 MiB ENH
RPMB Capacity: 128 KiB ENH
```

This shows that the whole User Space has been switched to Enhanced Mode and only 1.8 GiB of memory is left now.

Uff, finally done!

As you can see, this is rather cumbersome. Determining the sector count can not be done automatically and the conversion from hex values to decimal numbers also seems unnecessarily complicated.

This is the reason why F&S has enhanced this command. You can now give hex numbers by prefixing them with `0x`. (However the default without prefix is still decimal to stay compatible with mainline U-Boot.) And offset and sizes can be given in percent instead of absolute numbers by adding a `%` character. The whole procedure reduces to the following two commands:

```
# mmc hwpartition user enh 0 100% set
# mmc hwpartition user enh 0 100% complete
```

This converts the whole User Space (100%) starting from sector 0. That's all. You do not need to know anything at all about the specific sizes of the eMMC device. It will simply work.

6.8 Command fsimage To Handle F&S Images

As we have seen previously, storing a file to NAND and eMMC differs. It needs completely different commands:

- On NAND, it is command `nand write` and you need to specify a NAND offset or an MTD partition and the size in bytes.
- On eMMC, it is command `mmc write` and you need to specify a start block number and the size in blocks, which is diffcult to get, because `$filesize` only provides the size in bytes.

It gets even more complicated when storing NBoot, which consists of several parts where only a subset needs to be stored, but at different places. Storing these images manually requires a lot of knowledge of the flash layout. None of this is easy, and it is also error-prone.

By using the `fsimage` command, this will get super easy. Just load the NBoot or U-Boot image to RAM and then save it with `fsimage save`. Done. You do not have to care where everything has to go, it will simply work.

There are more subcommands available in `fsimage`:

Command	Function
`fsimage save`	Store NBoot or U-Boot image to the right place
`fsimage load [nboot]`	Load current NBoot image to RAM
`fsimage load uboot`	Load current U-Boot image to RAM
`fsimage list`	List sub-images of the current NBoot/U-Boot image in RAM
`fsimage board-id`	Show the BOARD-ID of the current board
`fsimage board-cfg`	List the board-configuration of the current board
`fsimage boot`	Show the current boot settings
`fsimage arch`	Show the current CPU architechture (e.g. fsimx8mm)

Table 23: fsimage sub-commands

6.8.1 fsimage on ARM32 (PicoCoreMX6UL[100], PicoCoreMX6SX)

On ARM32, only those boards that are booting from eMMC are equipped with a rudimentary `fsimage` command, i.e. PicoCoreMX6UL, PicoCoreMX6UL100 and PicoCoreMX6SX. Only `fsimage arch` and `fsimage save` are supported, and only U-Boot can be saved, not NBoot. The U-Boot extension has to be `.nb0`.

```
# tftp uboot.nb0
# fsimage save
```

6.8.2 fsimage on ARM64 (i.MX8)

On ARM64, U-Boot and NBoot images have the extension `.fs`. These images have a special F&S header prepended that gives information about the image type, size, a description, and images can even contain sub-images. In fact `nboot.fs` is a whole collection of smaller sub-images.

When `fsimage` saves these files, it always stores two copies. So if one copy fails to load, the boot process can fall back to the second copy. This makes the boot process very robust.

Here is an example when saving a U-Boot image. You do not have to care where the image is stored, it will automatically be stored to NAND or eMMC at the right place.

```
# tftp uboot.fs
...
# fsimage save
Found unsigned U-BOOT image at 0x40480000 (CRC32 header+image ok)
Booted from Primary SPL, so starting with copy 1
Saving copy 1 to mmc2:
  -- U-BOOT (hwpart 2) --
  Invalidating U-BOOT at offset 0x00180000 size 0x280000... OK
  Writing U-BOOT at offset 0x00180200 size 0x129d40... OK
  Completing U-BOOT at offset 0x00180000 size 0x200... OK
Saving copy 0 to mmc2:
  -- U-BOOT (hwpart 1) --
  Invalidating U-BOOT at offset 0x00180000 size 0x280000... OK
  Writing U-BOOT at offset 0x00180200 size 0x129d40... OK
  Completing U-BOOT at offset 0x00180000 size 0x200... OK
Saving U-Boot complete
```

The `fsimage save` command can also be used to save NBoot. In fact it is currently the only way to install a new NBoot version. Load the new image to RAM and call `fsimage save`. Again you do not need to know how this actually works and where the data is stored, it simply works. Now `fsimage save` has to do even more. It selects only those parts of NBoot that are relevant for this specific board and only saves these. If we ever change the layout in the future, `fsimage save` even knows how to handle this and will also move other files (U-Boot and Environment) if this is necessary for the new layout.

```
# tftp nboot-fsimx8mm-2023.10.fs
...
# fsimage save
Found unsigned NBOOT image at 0x40480000 (CRC32 header+image ok)
Found NBOOT version 2023.10
Booted from Primary SPL, so starting with copy 1
Saving copy 1 to mmc2:
  -- NBOOT (hwpart 2) --
  Invalidating NBOOT at offset 0x00088000 size 0xf8000... OK
  Writing BOARD-CFG at offset 0x00088200 size 0x460... OK
  Writing FIRMWARE at offset 0x00088660 size 0x40... OK
```

```
  Writing DRAM-INFO at offset 0x000886a0 size 0x40... OK
  Writing DRAM-TYPE at offset 0x000886e0 size 0x40... OK
  Writing DRAM-FW at offset 0x00088720 size 0x145b0... OK
  Writing DRAM-TIMING at offset 0x0009ccd0 size 0x1a10... OK
  Writing ATF at offset 0x0009e6e0 size 0x9220... OK
  Completing BOARD-CFG at offset 0x00088000 size 0x200... OK
  -- SPL (hwpart 2) --
  Invalidating SPL at offset 0x00048400 size 0x3fc00... OK
  Writing SPL at offset 0x00048600 size 0x1f000... OK
  Completing SPL at offset 0x00048400 size 0x200... OK
  Writing SECONDARY-SPL-INFO at offset 0x00008200 size 0x200... OK
  -- SPL (hwpart 1) --
  Invalidating SPL at offset 0x00048400 size 0x3fc00... OK
  Writing SPL at offset 0x00048600 size 0x1f000... OK
  Completing SPL at offset 0x00048400 size 0x200... OK
  Writing SECONDARY-SPL-INFO at offset 0x00008200 size 0x200... OK

Saving copy 0 to mmc2:
  -- NBOOT (hwpart 1) --
  Invalidating NBOOT at offset 0x00088000 size 0xf8000... OK
  Writing BOARD-CFG at offset 0x00088200 size 0x460... OK
  Writing FIRMWARE at offset 0x00088660 size 0x40... OK
  Writing DRAM-INFO at offset 0x000886a0 size 0x40... OK
  Writing DRAM-TYPE at offset 0x000886e0 size 0x40... OK
  Writing DRAM-FW at offset 0x00088720 size 0x145b0... OK
  Writing DRAM-TIMING at offset 0x0009ccd0 size 0x1a10... OK
  Writing ATF at offset 0x0009e6e0 size 0x9220... OK
  Completing BOARD-CFG at offset 0x00088000 size 0x200... OK
  -- SPL (hwpart 1) --
  Invalidating SPL at offset 0x00008400 size 0x3fc00... OK
  Writing SPL at offset 0x00008600 size 0x1f000... OK
  Completing SPL at offset 0x00008400 size 0x200... OK
  -- SPL (hwpart 2) --
  Invalidating SPL at offset 0x00008400 size 0x3fc00... OK
  Writing SPL at offset 0x00008600 size 0x1f000... OK
  Completing SPL at offset 0x00008400 size 0x200... OK

Saving NBoot complete
New BOARD-CFG is now active
```

The fsimage load command can be used to check the integrity of the stored images. The command tries to load both copies and verifies checksums and also compares the two copies. If one of the images (or a part thereof) fails to load, a warning is issued.

In the following example, the second copy of the BOARD-CFG is corrupt:

```
# fsimage load

Loading SPL from mmc2
  Loading copy 0 from hwpart 1 offset 0x00008400 size 0x1f200... OK
  Loading copy 1 from hwpart 2 offset 0x00048400 size 0x1f200... OK

Loading BOARD-CFG from mmc2
  Loading copy 0 from hwpart 1 offset 0x00088000 size 0x660... OK
  Loading copy 1 from hwpart 2 offset 0x00088000 size 0x660... BAD
CRC32 FAILED (-84)
  Warning! One copy corrupted! Saving NBoot again may fix this.
```

```
Loading FIRMWARE from mmc2
  Loading copy 0 from hwpart 1 offset 0x00088660 size 0x1f2a0... OK
  Loading copy 1 from hwpart 2 offset 0x00088660 size 0x1f2a0... OK

NBoot successfully loaded to 0x40480000
```

Nonetheless loading succeded, there was a valid copy available. If you call `fsimage save` now, NBoot will be saved again, fixing the corrupted parts while doing this. So after this sequence, NBoot should be fully functional again.

Remark

This sequence of `fsimage load` and then `fsimage save` will not work in Secure Boot mode. To be able to save NBoot, NBoot as a whole must be correctly signed. But as only parts of NBoot are actually stored on the board, the NBoot image loaded with `fsimage load` obviously only contains this subset of the original NBoot without a valid signature. Which means `fsimage save` will refuse this image because of the missing signature.

To get an impression on how complex a current NBoot file actually is, you can use command `fsimage list`. This command lists all sub-images of NBoot:

```
# tftp nboot-fsimx8mm-2023.10.fs
...
 # fsimage list
Content of F&S image at addr 0x40480000

offset   size     type (description)
------------------------------------------------------------
00000000 000610a0 NBOOT (fsimx8mm)
00000040 0001f200  SPL (fsimx8mm)
0001f280 00007aa0  BOARD-INFO (fsimx8mm)
0001f2c0 00000530   BOARD-CFG (PCoreMX8MM-FERT1)
0001f830 00000560   BOARD-CFG (PCoreMX8MM-FERT2)
0001fdd0 00000620   BOARD-CFG (PCoreMX8MM-FERT3)
00020430 00000620   BOARD-CFG (PCoreMX8MM-FERT4)
00020a90 00000600   BOARD-CFG (PCoreMX8MM-FERT9)
000210d0 00000570   BOARD-CFG (PCoreMX8MM-FERT12)
00021680 00000590   BOARD-CFG (PCoreMX8MM-FERT13)
00021c50 00000560   BOARD-CFG (PCoreMX8MM-FERT15)
000221f0 00000580   BOARD-CFG (PCoreMX8MMr2-FERT3)
000227b0 00000580   BOARD-CFG (PCoreMX8MMr2-FERT4)
00022d70 00000560   BOARD-CFG (PCoreMX8MMr2-FERT9)
00023310 00000570   BOARD-CFG (PCoreMX8MMr2-FERT12)
000238c0 00000590   BOARD-CFG (PCoreMX8MMr2-FERT13)
00023e90 00000560   BOARD-CFG (PCoreMX8MMr2-FERT15)
00024430 000005c0   BOARD-CFG (PCoreMX8MX-FERT5)
00024a30 00000600   BOARD-CFG (PCoreMX8MX-FERT6)
00025070 000005c0   BOARD-CFG (PCoreMX8MX-FERT7)
00025670 000005b0   BOARD-CFG (PCoreMX8MX-FERT8)
00025c60 00000560   BOARD-CFG (PCoreMX8MX-FERT17)
00026d60 00032fa0  FIRMWARE (fsimx8mm)
```

```
00026da0 00029d40   DRAM-INFO (fsimx8mm)
00026de0 0001c740    DRAM-TYPE (lpddr4)
00026e20 00014570     DRAM-FW (lpddr4)
0003b3d0 000019d0     DRAM-TIMING (lpddr4_k4f8e3s4hd_mgcl)
0003cde0 00002440     DRAM-TIMING (lpddr4_k4f8e304hb_mgcj)
0003f260 00002140     DRAM-TIMING (lpddr4_k4f6e3s4hm_mgcl)
000413e0 00002140     DRAM-TIMING (lpddr4_fl4c2001g_d9)
00043560 0000d580    DRAM-TYPE (ddr3l)
000435a0 0000c000     DRAM-FW (ddr3l)
0004f5e0 00000a60     DRAM-TIMING (ddr3l_2x_im4g16d3fdbg107i)
00050080 00000a60     DRAM-TIMING (ddr3l_2x_mt41k128m16tw)
00050b20 000091e0   ATF (fsimx8mm)
00059d40 00007360  EXTRA (fsimx8mm)
00059d80 000031b0   BASH-SCRIPT (addfsheader)
0005cf70 00004130   BASH-SCRIPT (fsimage)
```

To show the contents of the current BOARD-CFG, use this command:

```
# fsimage board-cfg

Found unsigned NBOOT image at 0x40480000 (CRC32 header+image ok)
Found NBOOT version 2023.10
FDT part of BOARD-CFG located at 0x404a0470

/ {
        board-cfg {
                board-cfg = "PCoreMX8MM-FERT4";
                board-cfg-version = <0x00000002>;
                board-name = "PicoCoreMX8MM-LPDDR4";
                dram-type = "lpddr4";
                dram-timing = "lpddr4_k4f8e3s4hd_mgcl";
                dram-size = <0x00000400>;
                dram-chips = <0x00000001>;
                boot-dev = "MMC3";
                fuse-bankword = <0x00010003>;
                fuse-mask = <0x10007fff>;
                fuse-value = <0x10002800>;
                have-emmc;
                have-sgtl5000;
                have-eth-phy;
                have-wlan;
                have-lvds;

                PCoreMX8MM-FERT4.110 {
                        board-rev = <0x0000006e>;
                };

                PCoreMX8MM-FERT4.120 {
                        board-rev = <0x00000078>;
                        have-rtc-pcf85063;
                };

                PCoreMX8MM-FERT4.130 {
                        board-rev = <0x00000082>;
                        have-rtc-pcf85263;
                };
        };
```

```
        nboot-info {
                version = "2023.10";
                support-crc32;
                save-board-id;
                uboot-with-fsh;
                uboot-emmc-bootpart;
                emmc-both-bootparts;

                nand {
                        spl-start = <0x00080000 0x00100000>;
                        spl-size = <0x00080000>;
                        nboot-start = <0x00180000 0x002c0000>;
                        nboot-size = <0x00140000>;
                        uboot-start = <0x00500000 0x00800000>;
                        uboot-size = <0x00300000>;
                        env-start = <0x00480000 0x004c0000>;
                        env-size = <0x00004000>;
                        env-range = <0x00040000>;
                };

                emmc-boot {
                        spl-start = <0x00008400 0x00048400>;
                        spl-size = <0x0003fc00>;
                        nboot-start = <0x00088000>;
                        nboot-size = <0x000f8000>;
                        uboot-start = <0x00180000>;
                        uboot-size = <0x00280000>;
                        env-start = <0x00000000>;
                        env-size = <0x00004000>;
                };

                sd-user {
                        spl-start = <0x00008400 0x00048400>;
                        spl-size = <0x0003fc00>;
                        nboot-start = <0x00088000 0x00448000>;
                        nboot-size = <0x000f8000>;
                        uboot-start = <0x00180000 0x00540000>;
                        uboot-size = <0x002c0000>;
                        env-start = <0x00440000 0x00444000>;
                        env-size = <0x00004000>;
                };
        };
};
```

7 Common Ways of System Installation

In this chapter we will present some common scenarios how the system software is typically installed on our boards. The variables `arch`, `platform`, `bootfile` and `bootfdt` supply valuable information to be able to write generic installation scripts. If we print these variables we see something like this:

```
# printenv arch platform bootfile bootfdt
arch=fsimx8mm
platform=picocoremx8mm-lpddr4
bootfile=Image
bootfdt=picocoremx8mm-lpddr4.dtb
```

The example shows that we are on the F&S i.MX8MM architecture, the board is a PicoCoreMX8MM with LPDDR4, the default kernel name is `Image` and the default device tree name is `picocoremx8mm-lpddr4.dtb`. These variables can be used to download the files in a very generic manner that will work on all F&S boards.

7.1 Installation to NAND Flash

This is the most common scenario when developing with our NAND based boards, typically on ARM32. You have new versions of some images available on your PC (Server) and want to download them via TFTP and store them in NAND flash (see Figure 40).

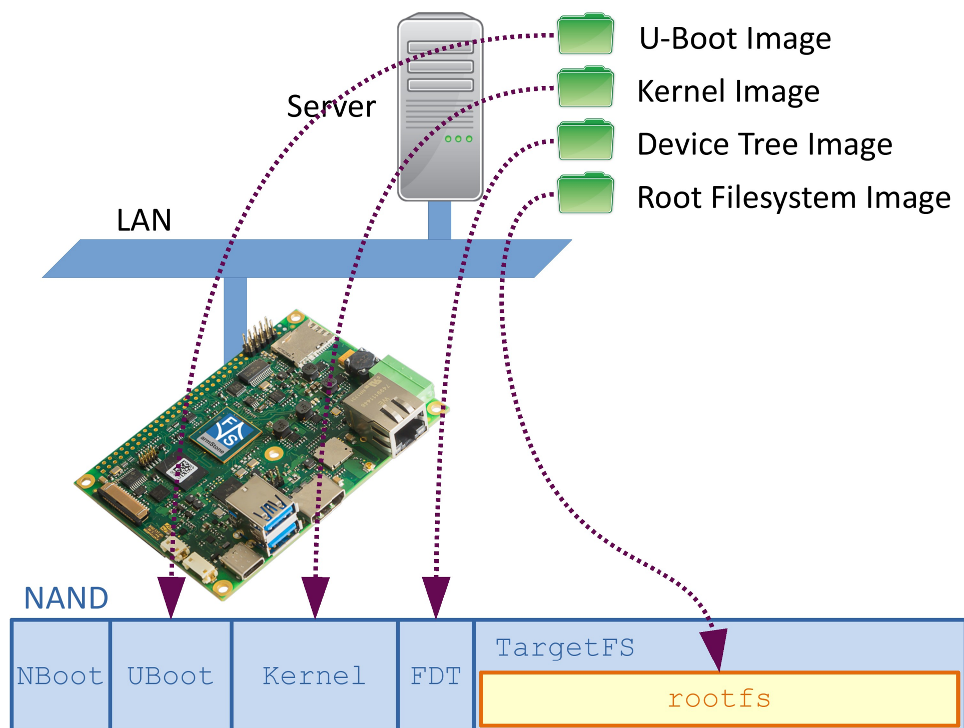


Figure 40: Download files with TFTP, store in NAND flash

There is no difference whether you want to install the whole system or just simply single files. Each of the following steps will install or replace one image type. Only the commands are shown, not the command output.

Replace U-Boot in MTD partition `UBoot` on ARM32:

```
tftp uboot.nb0
nand erase.part UBoot
nand write $loadaddr UBoot $filesize
```

Replace U-Boot in MTD partition `UBoot` on ARM64:

```
tftp uboot.fs
fsimage save
```

Replace Linux kernel image in MTD partition `Kernel`.

```
tftp $bootfile
nand erase.part Kernel
nand write $loadaddr Kernel $filesize
```

Replace device tree in MTD partition `FDT`.

```
tftp $bootfdt
nand erase.part FDT
nand write $loadaddr FDT $filesize
```

Replace the root filesystem in UBI volume `rootfs` on top of MTD partition `TargetFS`.

```
tftp rootfs-$arch.ubifs
ubi part TargetFS
ubi write $loadaddr rootfs $filesize
```

Of course instead of downloading the files via `tftp`, you can also use any other way, like loading them from a USB device or from an SD Card. Experience has shown that `tftp` is the fastest way.

7.2 Installation to eMMC

This is the scenario when developing with our eMMC based boards, typically on ARM64. Here the situation is slightly more complex, because the User part of eMMC is using software partitions like System, RootFS and Data (see Figure 41). While U-Boot is capable of working with partitions based on the Microsoft MBR partition table (Master Boot Record) and the GPT partition table (GUID Partition Table), there are still no commands to create these partitions and the partition table.

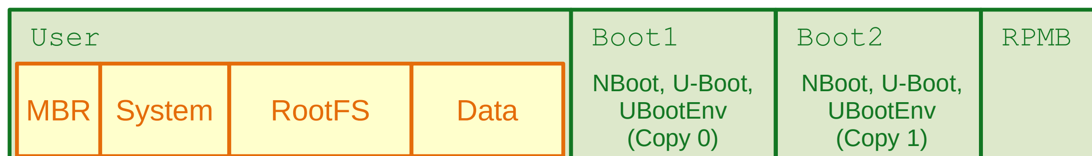


Figure 41: Typical eMMC layout

The only way to get the partition table installed is by creating the table on the PC and then copy the image to the board. The following table shows the partition layout as F&S is using in its default implementations, but you are of course free to choose a different layout.

	Offset in bytes (blocks)	Size in bytes (blocks)	Content
Partition Table	0x00000000 (0x0)	0x8000 (0x40)	MBR/GPT (32 KiB)
Reserved/ U-Boot	0x00008000 (0x40)	0x7f8000 (0x3fc0)	Raw data (8160 KiB)
System	0x00800000 (0x4000)	0x05800000 (0x2c000)	FAT32: Kernel, device trees (88 MiB)
RootFS	0x06000000 (0x30000)	n * 0x200 (n)	EXT4 (size configured in Yocto/Buildroot)
...			Data partition or similar

Table 24: Software partitions on eMMC

In fact when using an F&S predefined configuration in Yocto or Buildroot, the build process will already create a so-called `sysimg` that contains exactly this layout above. It has the partition table as MBR, the Reserved region, the System partition with Kernel and `.dtb` device trees already included and the RootFS partition with the root filesystem already included.

Remark

fsimx8
In the past, the Reserved region was used to store the U-Boot image. With NBoot after 2023.08, U-Boot was moved to the Boot1/2 hardware partitions and does not need any space in the User area anymore. However with respect to the remaining partitions, we decided to keep the Reserved area to stay compatible with the previous (and NXP's) layout.

fsimx6
U-Boot is still in the User partition at 0x00008000

Of course downloading these system images to the board using `tftp` like in the NAND case would theoretically also work. But unfortunately these system images are rather big and do not fit into RAM. And because `tftp` does not support downloading arbitrary parts of a file, this approach is typically not a valid option. In the following sections we will use an SD card or a USB pendrive to install these system images.

7.2.1 Installation From SD Card/USB Pendrive With Same Layout

In this scenario you use your development PC to prepare an SD card or USB pendrive with all partitions and files that you want to install on the board. You can use your preferred tools

to partition the device and create the filesystems, then you simply copy all the files to these partitions. Of course this step may be arbitrary complex until all necessary information is stored on the device.

The simplest way is to copy the `sysimg` to the SD card (or USB pendrive) using low-level commands, like `dd` in Linux and dd for Windows or HDD Raw Copy Tool on Windows. This means the device will have the same layout as later the eMMC.

Then you insert this card into the board. The idea is to do a low-level copy sector by sector of this SD card to the local eMMC memory. This means the actual data structure of the SD card, i.e. how many partitions there are, how big they are and what filesystem types they represent, is completely irrelevant at this point. The eMMC will have exactly the same structure afterwards. See Figure 42.

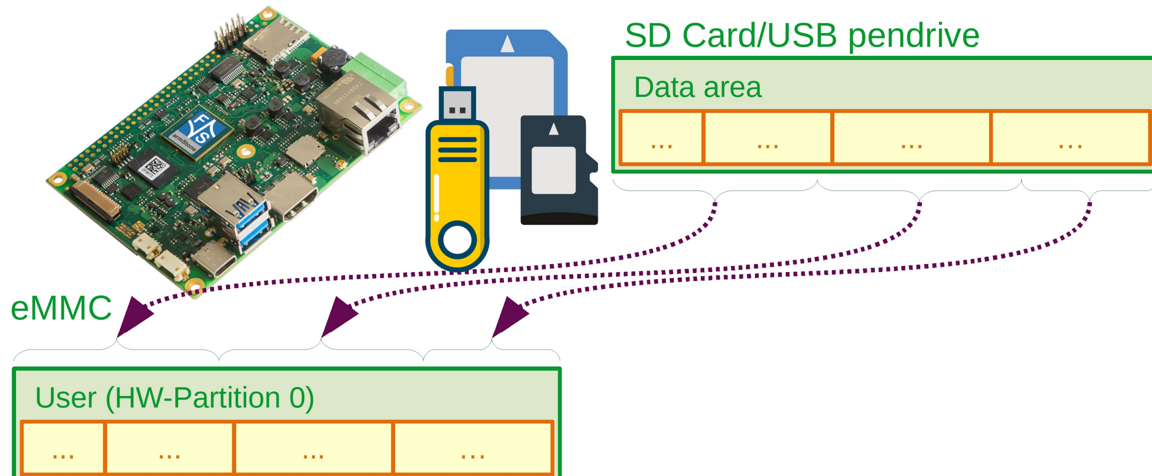


Figure 42: Low-level copy external SD card to internal eMMC

The biggest problem here is that the image of the whole SD card will not fit into RAM in one go. So it is necessary to split the data into smaller parts. Assume a 1 GB SD card (957 MiB = 0x001DE800 sectors) as source on MMC port 0. The eMMC device is on MMC port 2, RAM is 512 MiB. For example you can split the content in parts of 256 MiB (=0x00080000 sectors). The last part is slightly smaller then. The following sequence of commands will perform the copy. Only the commands are shown, not the output.

```
mmc dev 0
mmc read $loadaddr 0x0 0x80000
mmc dev 2
mmc write $loadaddr 0x0 0x80000
mmc dev 0
mmc read $loadaddr 0x80000 0x80000
mmc dev 2
mmc write $loadaddr 0x80000 0x80000
mmc dev 0
mmc read $loadaddr 0x100000 0x80000
mmc dev 2
mmc write $loadaddr 0x100000 0x80000
mmc dev 0
```

```
mmc read $loadaddr 0x180000 0x5e800
mmc dev 2
mmc write $loadaddr 0x180000 0x5e800
```

If you use a USB pendrive instead of an SD card, then the read commands would use `usb` instead of `mmc` and switching the MMC device with `mmc dev` in between would not be necessary. But loading small parts is still necessary.

The advantage of this solution is that you can use well-known tools on your PC to create the source device: partitioning, formatting, file copy, everything works as you are used to from your daily life when dealing with SD cards or USB sticks. Of course also graphical tools like `gparted` and your regular file explorer can be used.

The big disadvantage is that you also need to copy empty space. So for example if only 20% of the device is actually filled with data and the remaining part is just free space, then you still have to copy the entire device, because individual parts of the payload data may be distributed over the whole device. Copying these large images takes quite some time. And if you have to produce thousands of devices, this will sum up.

7.2.2 Installation From SD Card/USB Pendrive With System Image File

This scenario is rather similar to the previous one. But instead of actually setting up the SD card or USB pendrive with the same format as later in eMMC, you just create an image file of the final system layout, including the partition table and all software partitions. The `sysimg` is in fact an example for such a system image file. You simply copy this image file to the SD card or USB pendrive like any other file. It even does not matter if there are other files there, too, the device simply needs to have enough space for the system image.

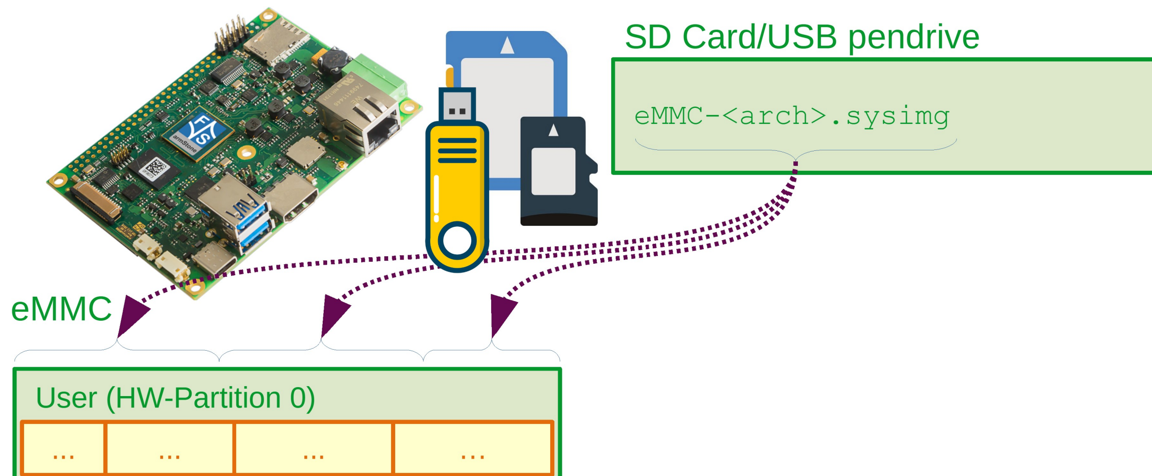


Figure 43: Copy sysimg to internal eMMC

Do not forget to unmount/eject the device correctly before pulling it from the development PC or you may have a corrupt image file and a non-functional system later.

The problem with the large file not fitting into RAM strikes again. Fortunately reading files from a filesystem allows reading arbitrary parts. You just have to give the size of the part and the offset where to start loading as additional arguments to the U-Boot `load` command.

A second problem is that the `load` command uses bytes to define offset and size, while the `mmc write` command uses blocks for offset and size, requiring a conversion.

Here is an example assuming the same sizes as before, i.e. an image with 1 GB image and splitting the file in parts of 256 MiB. Please note the byte values in `load` and the block values in `mmc write` and also the smaller last part. The first command selects the eMMC device for low-level writing. The high-level read from the filesystem can work independently from this, so there is no need to switch back and forth.

```
mmc dev $mmcdev
load mmc 0 $loadaddr eMMC-${arch}.sysimg 0x10000000 0x0
mmc write $loadaddr 0x0 0x80000
load mmc 0 $loadaddr eMMC-${arch}.sysimg 0x10000000 0x10000000
mmc write $loadaddr 0x80000 0x80000
load mmc 0 $loadaddr eMMC-${arch}.sysimg 0x10000000 0x20000000
mmc write $loadaddr 0x100000 0x80000
load mmc 0 $loadaddr eMMC-${arch}.sysimg 0x0bd00000 0x30000000
mmc write $loadaddr 0x180000 0x5e800
```

7.2.3 Installing Individual Files

In this scenario we want to install single files (see Figure 44).

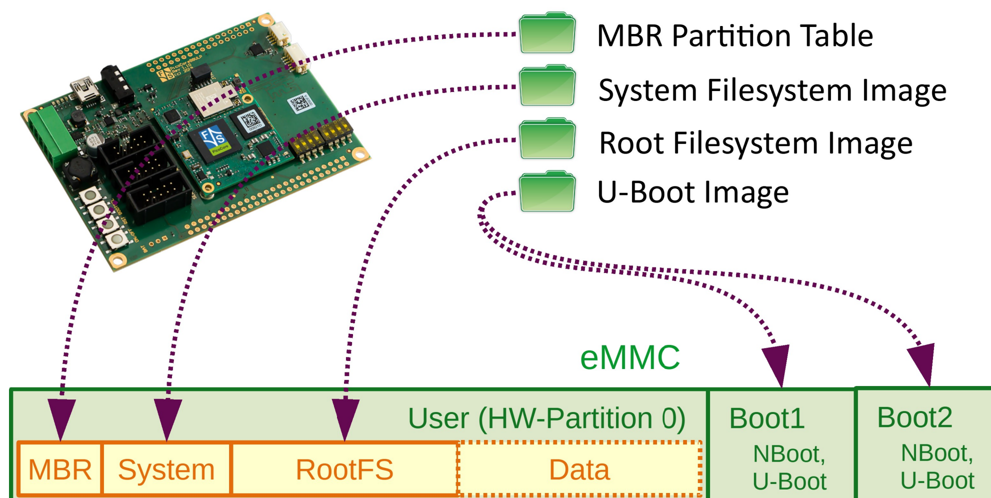


Figure 44: Installing individual files on eMMC

We assume that the system was installed at least once with one of the methods above. This means the partition table is valid and we can access the individual partitions. You can verify this with the following command:

```
# mmc part

Partition Map for MMC device 2  --   Partition Type: DOS

Part    Start Sector    Num Sectors     UUID            Type
  1     16384           180224          00000000-01     0c Boot
  2     196608          999426          00000000-02     83
```

You can see the System partition beginning at block 16384 (=0x4000), size 180224 (=0x2c000) blocks with type 0x0c (FAT32 with LBA) and the RootFS partition beginning at block 196608 (=0x30000) with type 0x83 (Linux).

From then on, you can also download and install individual files. As long as they fit into RAM, you can load them with `tftp`, from SD card or USB pendrive. If a file exceeds RAM size and you have to download it in smaller chunks, then `tftp` will not work. In the following examples we skip the download command and only show the commands to save the files.

To save a new U-Boot version

```
fsimage save
```

To save a new root filesystem.

```
setexpr blkcnt $filesize + 0x1ff
setexpr blkcnt $blkcnt / 0x200
mmc write $loadaddr 0x30000 $blkcnt
```

Please be careful, if the new image exceeds the partition size, you will overwrite a part of the next partition. In this case you may need to reinstall larger parts of the system.

Replacing the Linux kernel or a single device tree file is more complicated. The F&S version of U-Boot does not provide a `fatwrite` command. In our point of view, the main purpose of U-Boot is to bring data from a PC to the board for installation purposes. Unfortunately when there is a write command to a filesystem, it is also possible to bring data from the board to some external media like SD card or USB pendrive. In other words this makes it very easy to steal software. This is why we do not include the `fatwrite` command.

Unfortunately this also prohibits writing single files to the System partition that holds Linux kernel and device trees. So at the moment, there are two options: You could either use command `ums` to mount the System partition on your PC and replace the files there, or you can provide a new version of the whole System partition image and replace the whole partition. In the latter case, after loading the image to RAM, you can save it with:

```
setexpr blkcnt $filesize + 0x1ff
setexpr blkcnt $blkcnt / 0x200
mmc write $loadaddr 0x4000 $blkcnt
```

Remark

Buildroot already creates this System Partiton automatically. It is called `boot.vfat` and is located in the `images/` directory.

7.3 F&S Update and Installation Scripts

The previous examples have shown that it takes several steps to install the system software, especially if the images are large and need to be split. F&S provides an install script that simplifies the installation process considerably, for example by automatically splitting the `sysimg` in smaller parts and downloading the parts one after the other. The script works in conjunction with the Install/Update/Recover mechanism (see Chapter 6.4) and can be used to install the system from an SD card or USB pendrive.

You can also use this script as a basis for your own installation process, for example to be used in your production or even for updates in the field.

8 Working With Linux

###TODO###

F&S Improvements: bdinfo, Boot process, Init-System. /proc, filesystems, read-only rootfs, Buffer-Cache.

Some parts may go here, some to chapter 2.

Configuration of the kernel, device drivers as modules or directly as part of the kernel image.

How is Linux built, usually as part of Buildroot or Yocto, so that kernel modules are automatically stored in the rootfs without any additional steps.

8.1 Busybox

Busybox instead of single command line tools.

8.2 Device Trees

###TODO###

8.3 Devices And Device Drivers

The Linux device driver system is quite simple, but nonetheless very effective. Linux distinguishes *platform devices* from *device drivers*. The platform device is the piece of hardware itself, for example a CPU core, a memory, a system communication bus, a peripheral device, or an external chip. And the device driver is the code that can handle this platform device.

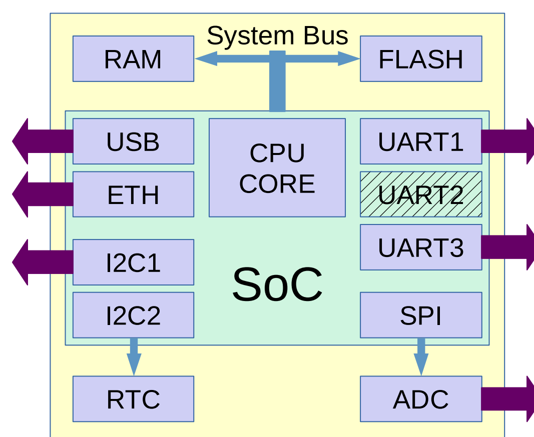


Figure 45: Platform devices on an imaginary hardware

In 45 you can see an imaginary hardware platform with all its platform devices, either within the System-on-Chip (SoC) or externally. Some of these platform devices have interfaces to the outside and some are just used internally. For a working system, Linux needs to know

about all these platform devices and whether they are used or not. For example in this platform, UART2, though available on the SoC, is not used, because it is not needed.

Linux needs to include the code for all device drivers for these devices. Of course one driver can handle several instances of the same platform device, like in the case of UART and I2C in the example above.

The latter part (including device drivers) is quite simple. Just call `make menuconfig` and activate the appropriate menu entries in the *Device Drivers* section. But the former part (providing the platform devices) is more challenging as it requires a deep knowledge of the specific hardware available on this platform and in the past always involved additional kernel code, the so-called *board support code*. But this has changed now to *device trees*.

8.4 Old Way With Board Support Code

For quite a long time, providing the platform devices for a specific platform was done by adding a board support file to the kernel. The functions of this regular C source code file were called during the boot phase of the kernel and board-specific code that knew all the details of the board hardware, created a list of all available platform devices and registered them with the kernel. Thus the board support code was part of the final kernel image. See 46.

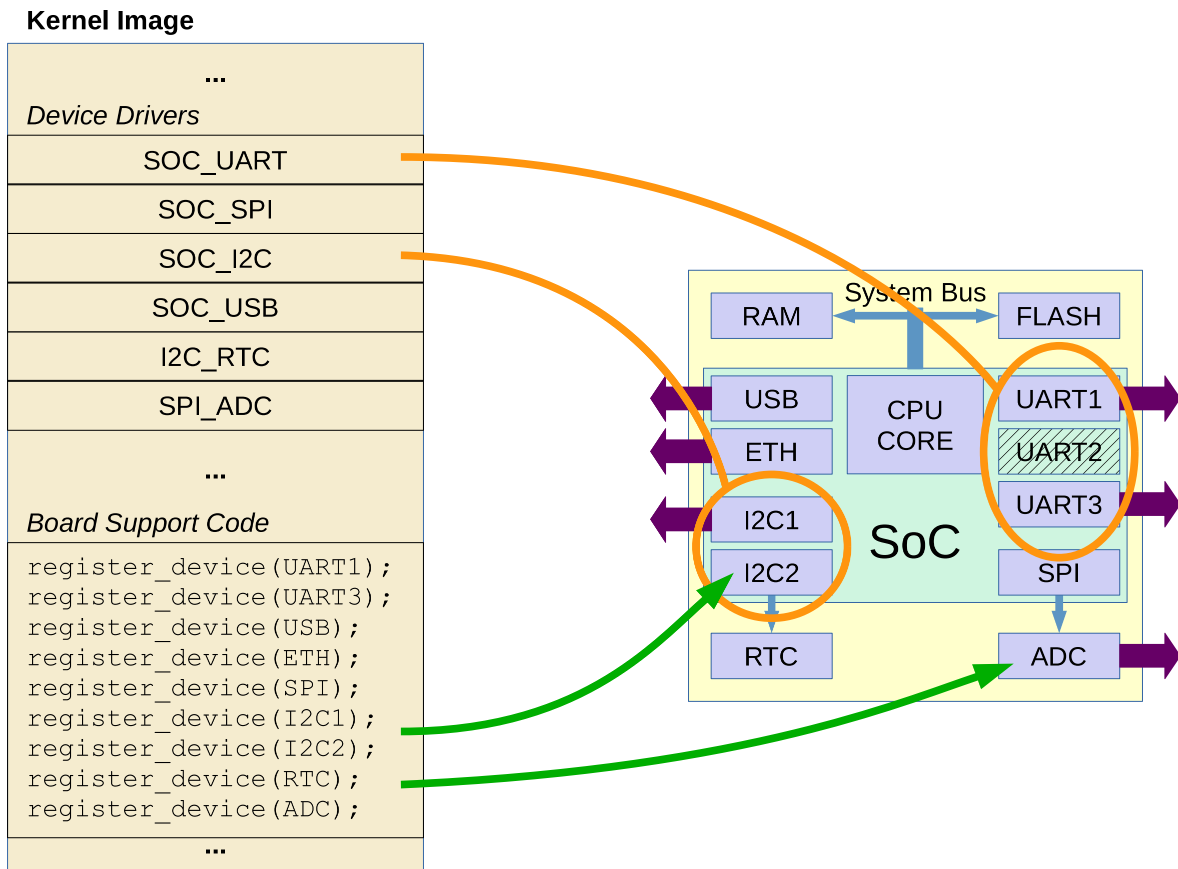


Figure 46: Old way: Device drivers and board support are both in the kernel image

Every time a platform device found a matching device driver, the device was activated. If either side was missing, the device was not available in the running kernel. For example UART2 was not available because it was not added as a platform device, even though the SOC_UART driver was present.

But in the world of embedded systems, this concept with only one board support file was very unsatisfactory. Embedded systems vary quite heavily, from very small and restricted systems to full-featured and very complex systems. As we can see in our example, already the core of an embedded system, the System-on-Chip (SoC), offers quite a lot of devices, but these devices differ considerably from manufacturer to manufacturer, providing more, less or simply different periphery. This scaled not very well with the platform specific support code and lead to a situation where embedded Linux systems were very self-concentrated. Each system had its own kernel that could only run on this one system and nowhere else.

In an attempt to improve the situation, Machine IDs were invented, especially in the ARM branch of Linux. For each platform that should officially be supported in Linux, the developers had to request a unique Machine ID. This Machine ID was passed from the bootloader to the kernel and based on this ID, the kernel decided which platform support code to start. This allowed to add several different board support files to one kernel image, and such a kernel image could run on a few different platforms. See 47.

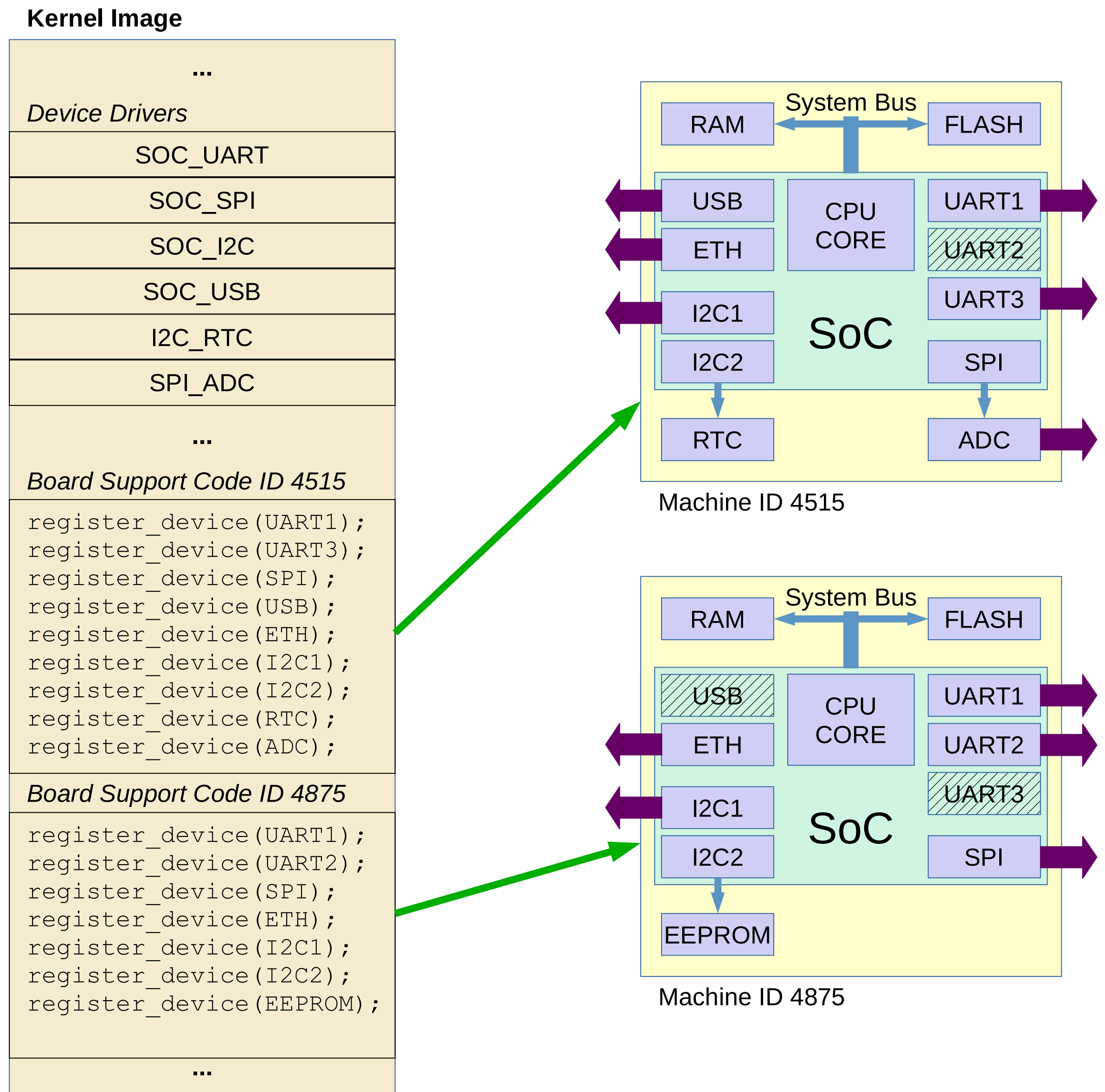


Figure 47: Old way: Board support for several platforms with Machine IDs

However this concept still had some major flaws. Even the smallest modification of the board configuration, like changing the conversion rate of an A/D Converter (ADC), the resolution of an LC-Display or the number of a GPIO pin, required changes in the board support source code and thus a recompilation of the kernel. And board support was only possible for existing platforms. Every new platform needed additional platform support code and therefore resulted in a new kernel version. And last but not least, people often added other code to the board support that did not fit anywhere else, just to avoid having to add an own driver file.

Hence the Linux people further considered all these aspects and finally came up with a completely new way of handling platform devices by using device trees.

8.5 New Way With Device Trees

The basic idea behind device trees is to separate all board specific stuff from the kernel. The kernel itself should be as generic as possible. Of course it still contains all the device drivers, but everything regarding platform devices is now removed from the kernel image and moved to a separate file called device tree. A device tree is basically nothing more than the list of platform devices that are available on a specific platform.

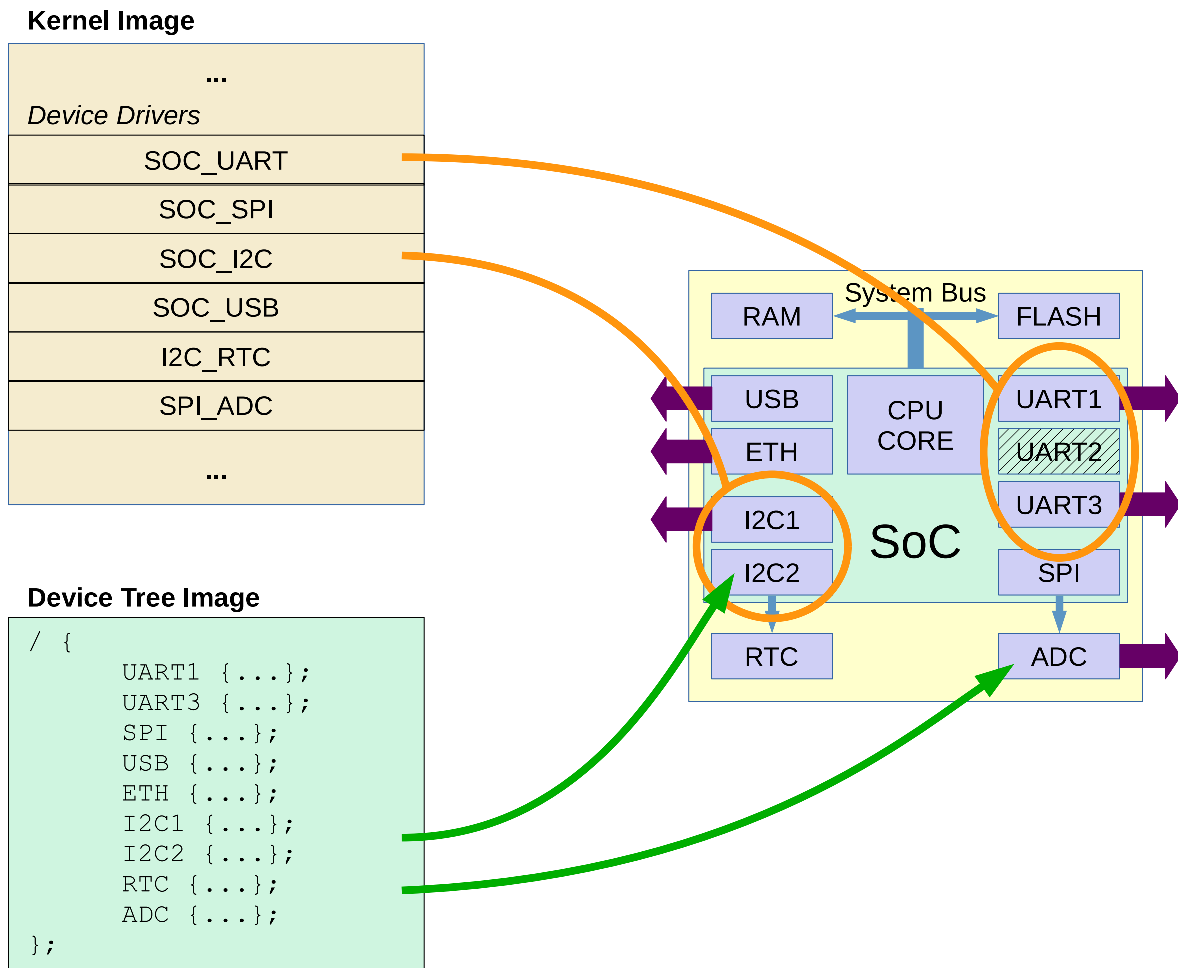


Figure 48: New way: Device tree is separate from the kernel image

The device tree is defined as a text file with a special syntax and the filename extension `.dts` (device tree source). The syntax is used to define nodes and properties. Each node represents a device of the platform, for example an A/D Converter (ADC). The properties tell how the node (=ADC) behaves in detail. For example one property of a node representing such an ADC may be the conversion rate, i.e. how many values are sampled per second.

This text file is converted by the device tree compiler `dtc` to a binary "blob" with filename extension `.dtb` (device tree blob). This binary file can either be appended to the kernel image file, or it can be handled separately. We at F&S have decided to handle it separately, because then you can use one and the same kernel image in combination with many different

device tree binaries and vice versa. In our opinion, the combined version contradicts the strict separation to a certain degree.

Like the image of the kernel itself, the device tree blob is downloaded to the board and stored in an own partition. When the system boots, the bootloader U-Boot loads both images to RAM and then passes the device tree address (like the optional initial ramdisk address) to the kernel.

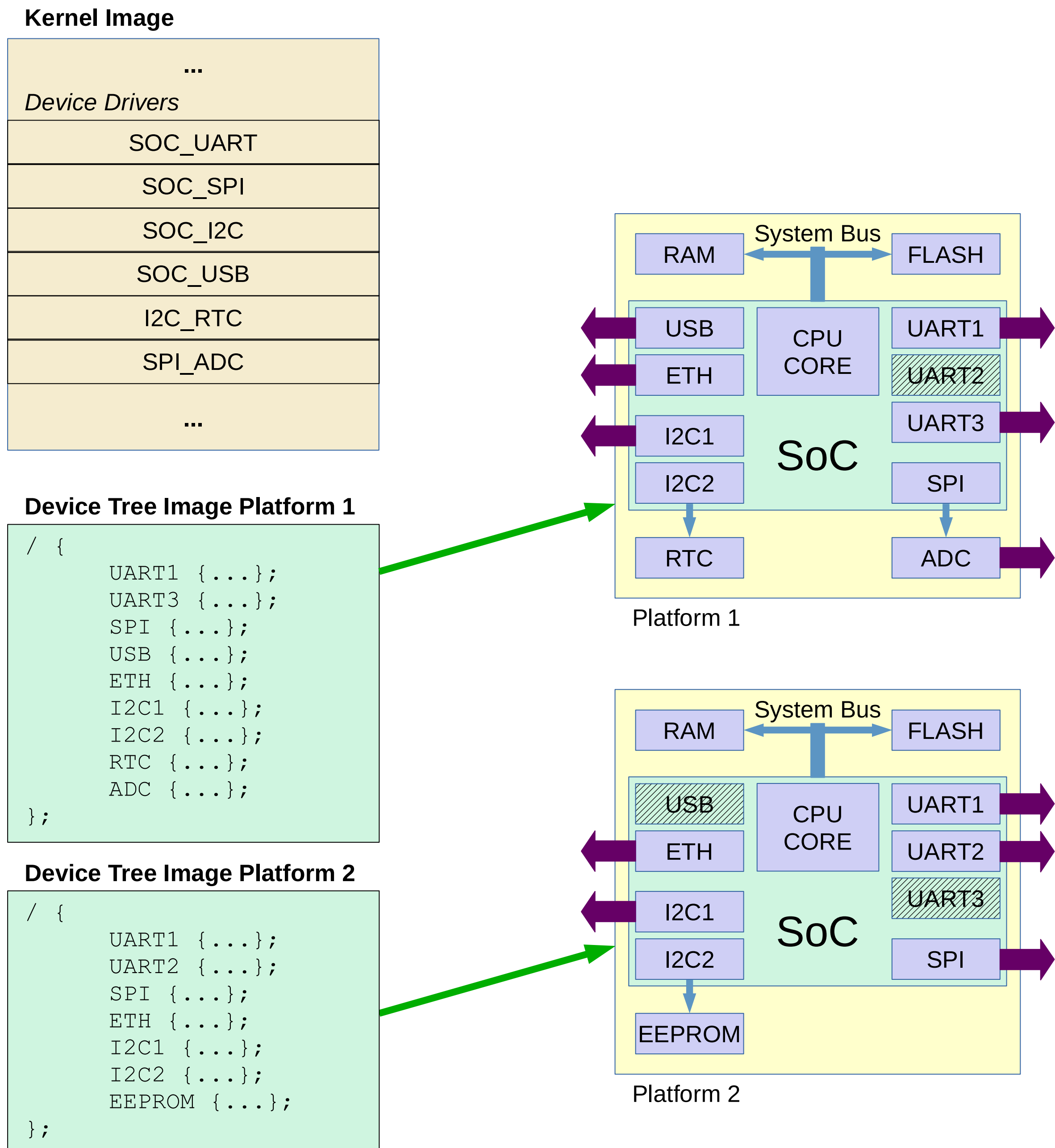


Figure 49: New way: One device tree per platform

8.6 Advantages of Device Trees

The biggest advantage is the flexibility. If anything in the board configuration changes, like the conversion rate of the ADC in the example above, then only the device tree file has to be recompiled with the device tree compiler, which is just a matter of seconds. And then only this file with a few kilobytes has to be replaced on the board. The main kernel image can remain the same.

To support several different platforms, we simply provide several device trees, one device tree per platform (see 49). For example if an F&S architecture consists of the boards PicoCOM, armStone and efus, then each of these boards has its own device tree file. But they can all share the same kernel image.

It is also very simple to create application specific device trees. For example you can prepare your own device tree, where all additional platform devices of your own application-specific hardware are added, the settings (properties) of the existing devices are modified and all devices that are not needed on your hardware are completely left out. Then just compile this new tree, replace it on the board, and you are done. And still the kernel can remain the same.

And such a kernel is also capable to support new boards that are not available yet. So in theory, adding a new board to our board family in the future just requires defining a new device tree. Only if we need a device driver that is missing in the current kernel, then we have to expand the kernel configuration and need to recompile the kernel.

We even have the vision that in the future we can have kernels that span different architectures. One single kernel image that can be used on fsimx6, fsimx6ul, fsimx6sx and maybe even on fsvybrid and other CPU types.

8.7 Built-in Drivers And Kernel Modules

###TODO###

Show difference, also explain grey area with respect to closed source modules.

9 Using the Standard System and Devices

Yocto:

The Yocto rootfs is mounted read-write by default. To build a read-only rootfs add the option "-m ro" when calling the fus-setup-release script to setup your Yocto build.

Buildroot:

By default, the standard root filesystem is mounted read-only. Therefore you can not create files unless you go to a directory like `/tmp` that is located in a RAM disk. This is to make the system as stable as possible. If the root filesystem is mounted read-only, it is usually no problem to just switch off the power.

If you want to remount the filesystem in read-write mode, just say

```
mount -o remount,rw /
```

Note the slash `/` that is denoting the mount point of the root directory. Now you can create files everywhere. But remember that written data is often buffered in RAM first and is not immediately stored on the media itself. If you simply switch off the power now, some still RAM buffered data may be lost. So in this case it is important to actually shut down the system with

```
halt
```

or restart with

```
reboot
```

Or you can remount the root filesystem back to read-only after applying the changes with

```
mount -o remount,ro /
```

All these commands will force the system to actually write any buffered data to the media.

The `/dev` directory is also built on top of a RAM disk. This allows the kernel to create and remove device entries dynamically. For example if a USB stick is attached, a device `/dev/sda1` is automatically created. And when the USB stick is unplugged, the device is also automatically removed again.

There are two systems in Buildroot that can do this. A very small and rudimentary system is provided by Busybox and is called `mdev`. But the more powerful system is an own package called `udev`. The `F&S` platforms use `udev` as they need features that are not well supported by `mdev`, for example dynamic loading of firmware files when devices are activated at runtime. Also the input daemon `evdev`, used for touch input, requires `udev` support.

9.1 procfs

Linux has a virtual filesystem called Procfs. It is mounted under `/proc` and provides information about the system in general and about each process that is currently active.

Get information about the CPU

```
cat /proc/cpuinfo
```

Shows the Linux version.

```
cat /proc/version
```

Shows the current memory usage.

```
cat /proc/meminfo
```

Lists the supported filesystems.

```
cat /proc/filesystems
```

9.2 Sysfs

Sysfs is a virtual file system in Linux. As the name suggests, it exports information about devices and drivers from the kernel device model to user space, and is also used for configuration.

Devices that want to share information or want to accept configuration settings, create subdirectories below the `/sys` directory. The subdirectories can contain virtual text files. So for example if a touch panel can accept some sensitivity configuration, it would create a file `sensitivity` there. By reading data from the file, we could query the current setting. And by writing a new value to the file, we could set a new sensitivity value.

Many things can be queried and modified in this way. In fact the Linux implementation of many tools and utilities often simply looks at the `/sys` directory to perform its task.

Most devices can be found in the subtree under `/sys/devices`, subdivided into categories like `cpu`, `platform`, `software`, `system` and `virtual`. These categories represent the *place* where the device is located in the system. There is a directory `/sys/class`. Here the devices are sorted by their *function*. The following example lists all available classes on fsimx6.

```
# ls /sys/class/

ata_device    firmware    mdio_bus      ptp            thermal
ata_link      gpio        mem           pwm            tty
ata_port      gpu_class   misc          regulator      ubi
backlight     graphics    mmc_host      rtc            udc
bdi           hwmon       mtd           scsi_device    vc
block         i2c-dev     mxc_ipu       scsi_disk      video4linux
bluetooth     ieee80211   mxc_vpu       scsi_host      vtconsole
```

```
devcoredump  input    net           sound        watchdog
dma          lcd      power_supply  spi_master
drm          leds     pps           spidev
```

For example access the RTC subsystem:

```
cat /sys/class/rtc/rtc0/date
```

Show the CPU core temperature (in 1/1000 °C):

```
cat /sys/class/thermal/thermal_zone0/temp
```

Show board specific information:

```
cat /sys/bdinfo/board_revision
```

9.3 bdinfo

###TODO###

9.4 Serial

The default speed is 115200 bit/s. On NXP CPUs, the devices are called `/dev/ttymxc<n> or /dev/ttyLP<n>`, where `<n>` is a number starting with 0. One port is usually used as serial debug port where all console messages are sent to. You can use input and output re-direction to use a serial port from the command line.

Example:

```
echo Hello > /dev/ttymxc2
```

Show characters that arrive on port `ttymxc2`:

```
cat < /dev/ttymxc2
```

Usually character input is line buffered, so you will only see information after sending Return.

Remark

The default setting for serial ports in Linux echo each character that arrives. So if you con-nect one serial port to a second serial port with a Null-Modem cable, sending a single char-acter will result in an endless loop. Each side will echo the character indefinitely. You can use `stty` to change this behaviour.

9.5 CAN

The CAN driver uses Socket CAN, i.e. the CAN bus is accessed as a network device, similar to an ethernet card. If the driver is available, you can find a `can0` device when issuing the command

```
ifconfig -a
```

But better use the newer ip program as the older ifconfig does not know anything more detailed about CAN controllers.

```
ip link
```

Before you can activate the CAN device, you have to set the baud rate. This requires the ip program. For example to set 125000 bit/s for CAN and activate, use this command:

```
ip link set can0 up type can bitrate 125000
```

Now you can create sockets that access the CAN device. Some examples are provided in the package can_utils in Buildroot (can_tx.c, can_rx.c, candump.c, cansend.c).

9.6 Ethernet

To activate the ethernet port in Linux, you have to configure the network device first. For example to use IP-Address 10.0.0.242, you can use the command

```
ifconfig eth0 10.0.0.242 netmask 255.0.0.0 up
```

Then you can use network commands, e.g.

```
ping 10.0.0.121
```

There is also a DHCP client included. To receive an IP address via DHCP just call:

```
udhcpc
```

If your board supports more than one network interface, you can add option -i to specify the appropriate interface. For example, to request IP data for interface eth1:

```
udhcpc -i eth1
```

9.7 WLAN

Wireless LAN is available on some F&S. Configure the WLAN adapter for your needs by using the ifconfig or ip program. Use program wpa_supplicant to set up all parameters required for WLAN access, like data encryption and login information.

The data for known WLAN connections is kept in the file /etc/wpa_supplicant.conf. If the root filesystem, is in read-write-mode, you can append the access information for a new network to this file. For example, if the SSID of the network is My WLAN and the passphrase is My secret access code, then you can issue:

```
wpa_passphrase "My WLAN" >> /etc/wpa_supplicant.conf
My secret access code
```

From now on, wpa_supplicant will automatically connect to this network if it is in range.

Start the wpa_supplicant:

Azurewave wlan chip

```
wpa_supplicant -B -D nl80211 -i mlan0 -c /etc/wpa_supplicant.conf
```

Silex wlan chip

```
wpa_supplicant -B -D nl80211 -i wlan0 -c /etc/wpa_supplicant.conf
```

`wpa_supplicant` stays active in the background and handles all WLAN tasks. For example, it scans all WLAN networks in range and checks if a known network is among them. After a few seconds, the WLAN connection should be established. Then you can start the network similar to an Ethernet interface, for example with:

Azurewave wlan chip

```
udhcpc -i mlan0
```

Silex wlan chip

```
udhcpc -i wlan0
```

There is a special program called `wpa_cli` that can talk to the running `wpa_supplicant` process. A single command can be given as argument to `wpa_cli`. For example, to show the list of currently available WLANs, call:

```
wpa_cli scan_results
```

Calling `wpa_cli` without arguments will go to interactive mode. Now you can enter `wpa_supplicant` commands directly, without having to type `wpa_cli` before each line.

The list of `wpa_supplicant` commands, that you can show by entering `help`, is rather long, way too much to handle it here. When you are done, type `quit` to leave interactive mode again.

9.8 Qt support

Qt is a cross-platform application framework that is widely used for developing applications mostly with a graphical user interface.

Buildroot

F&S provides a qt*_defconfig

Yocto

F&S provides a fus-image-qt* recipe.

9.9 Video Processing (gstreamer support)

Modules based on fsimx6 support hardware accelerated video processing in `gstreamer`. Table 25 holds a list of all i.MX6 specific `gstreamer1` plugins that use hardware acceleration.

Plugin name	Description
imxv4l2src	Video4Linux source (e.g. camera)
aiurdemux	VPU-based demultiplexer
vpudec	VPU-based video decoder
vpuenc_jpeg	VPU-based JPEG encoder
vpuenc_h263	VPU-based H.263 video encoder
vpuenc_h264	VPU-based H.264 video encoder
vpuenc_mpeg4	VPU-based MPEG4 video encoder
imxv4l2sink	Video4Linux video sink
imxeglvivsink	GPU based video sink
beepdec	Audio decoder
imxvideoconvert_g2d	G2D-based video converter
imxvideoconvert_ipu	IPU-based video converter

Table 25: VPU based GStreamer plugins

Modules based on fsimx8mp support hardware accelerated video processing in `gstreamer`. The following table holds a list of all iMX8MP specific gstreamer1 plugins that use hardware acceleration.

Plugin name	Description
v4l2src	Video4Linux source (e.g. camera)
aiurdemux	Multi-format demultiplexer for several video formats
imxvideoconvert_g2d	IMX g2d video converter (pixel format, rotation)
imxcompositor_g2d	IMX g2d video compositor (combine several videos, add background)
v4l2sink	Video4Linux video sink (overlay support)
waylandsink	Wayland video sink
beepdec	Audio decoder for several audio formats
vpu_enc_h264	IMX VPU-based AVC/H264 video encoder
vpuenc_vp8	IMX VPU-based VP8 video encoder
vpudec	IMX VPU-based video decoder

To play an AVI (h264 encoded) file for example the following command can be used for playback

```
gst-launch-1.0 filesrc location=video.avi typefind=true
    ! aiurdemux ! vpudec ! imxv4l2sink
```

Please note that the Video4Linux video sink (`imxv4l2sink`) will show the video content in an overlay framebuffer.

Of course, also a lot of regular gstreamer plugins are availabe. The following command will show all available plugins and supported video and audio formats. The output is rather lengthy.

```
gst-inspect-1.0
```

To get more information about a specific item, just add it after the command. For example to get information about the v4l2sink, enter:

```
gst-inspect-1.0 v4l2sink
```

Check if the video path is working:

```
gst-launch-1.0 videotestsrc ! fbdevsink
```

This command renders a kind of test image directly into the framebuffer. Use `v4l2sink` to render into a screen overlay. You can modify the test image by adding `pattern=<n>` to `videotestsrc`, where `<n>` is a number of 0 to 24. For example …

```
gst-launch-1.0 videotestsrc pattern=18 ! fbdevsink
```

… will show a moving ball on the screen.

Check if the audio path is working:

```
gst-launch-1.0 audiotestsrc ! alsasink
```

This will output a tone made of a sine curve. You can modify the output by adding `wave=<n>` to `audiotestsrc`, where `<n>` is a number of 0 to 12. This will use a different curve or use noise instead. For example …

```
gst-launch-1.0 audiotestsrc wave=9 ! alsasink
```

… will output White Gaussian Noise.

The easiest way to play a video or audio file is by using the `playbin` filter.

```
gst-launch-1.0 playbin uri=<name>
```

You have to give the file name in URI-style, i.e. prepend `file:` and use the absolute path. For example, if you are working as user `root`, who has his home directory in `/root`, and you have an MPEG-2 video called `video.mpg` there, then call:

```
gst-launch-1.0 playbin uri=file:/root/video.mpg
```

`playbin` will automatically select a suitable set of filters and plugins to play the file.

Please note that most video sinks do not accept interlaced videos. If your video is in interlaced format, you have to add a flag to tell playbin to deinterlace the video when playing. The flags value is a combination of several flags. The default value is 0x17, the flag for deinterlacing is 0x200. This is the final command:

```
gst-launch-1.0 playbin uri=file:/root/video.mpg flags=0x217
```

You can also give your own list of filters, which is slightly more complicated. For example to just play the video part `video.mpg` this would be:

```
gst-launch-1.0 filesrc location=video.mpg ! mpegpsdemux \
    ! mpegvideoparse ! mpeg2dec ! deinterlace ! videoconvert \
    ! fbdevsink
```

To play an audio file `sound.wav`:

```
gst-launch-1.0 filesrc location=sound.wav ! wavparse ! Alsasink
```

9.10 I2C

Most devices on an I²C bus are accessed by a device driver in Linux. But you can also have access to additional devices from userspace software. There are even command line tools to do this.

Show the available I²C buses:

```
i2cdetect -l
i2c-3   i2c             30a50000.i2c                    I2C
adapter
i2c-1   i2c             30a30000.i2c                    I2C
adapter
i2c-2   i2c             30a40000.i2c                    I2C
adapter
i2c-0   i2c             30a20000.i2c                    I2C
adapter
```

Show the available devices on bus `i2c-0`:

```
i2cdetect -y 0
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- -
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
70: -- -- -- -- -- -- -- -
```

So, there are no devices on `i2c-0`. Some devices are handled by a Linux driver, so they cannot be accessed from user space (therefore marked as “used” with `UU`). Now you can read and write data with `i2cget` and `i2cset` or read whole data areas with `i2cdump`.

9.11 SPI

SPI devices are often directly accessed by device drivers from the kernel. However, an SPI device can also be made available to user space via the usermode SPI driver called `spi-dev`. Such devices are available as `/dev/spidev<n>.<m>`, where `<n>` is the SPI bus number and `<m>` is the chip select number on this bus.

With `spi-config`, you can get information about the settings of your SPI device, and you can also change them.

```
spi-config -q -d /dev/spidev1.0
/dev/spidev1.0: mode=0, lsb=0, bits=8, speed=20000000, spiready=0
```

Set the device to use 8 MHz and SPI mode 3:

```
spi-config -m 3 -s 8000000 -d /dev/spidev1.0
```

Remark

The speed is only kept as long as the file is open. So, if you query the device again after the above command, it will still show `speed=20000000`. So, you have to keep the `spi-config` command running to see the change. For example:

```
spi-config -m 3 -s 8000000 -w -d /dev/spidev0.0 &
spi-config -q -d /dev/spidev1.0
/dev/spidev1.0: mode=3, lsb=0, bits=8, speed=8000000, spiready=0
fg
<Ctrl-C>
spi-config -q -d /dev/spidev1.0
/dev/spidev1.0: mode=3, lsb=0, bits=8, speed=20000000, spiready=0
```

Option `-w` in the first command keeps the command running, but the ampersand `&` sends it to the background. When you now issue the query command in the foreground, you can see the change to 8 MHz. After pulling the first command back to the foreground and stopping it with *Ctrl-C*, the speed is also back at 20 MHz.

The `spi-pipe` command is used to send data (from `stdin`) to the device and at the same time receive data from the device (to `stdout`). So as the name implies, it is intended to be used as part of a command pipeline

```
<command generating data> | spi-pipe | <command consuming data>
```

Or you can use I/O redirection to get the data to send from one file and store the received data in a second file. The following command will send 10 bytes of data from file `data.send`

as 2 packages of 5 bytes to the device. The received data, also 10 bytes, will be stored in file `data.received`.

```
spi-pipe -b 5 -n 2 < data.send > data.received
```

SD Card

Besides the size (regular SD and Micro SD) there are also two types of SD card slots: slots with and without a Card Detect (CD) signal. Slots without a CD pin can only be used for non-removable media. So the card is detected only if it is present at boot time. If it is inserted later, it is not detected anymore. Slots with a CD pin are meant for removable media. They can detect at runtime when a card is inserted or ejected.

If an SD card is detected in the system, a device `/dev/mmcblk0` is created. This device represents the whole card content. If the device also holds a partition, an additional device `/dev/mmcblk0p1` is created. This device represents this single partition only.

You can mount and unmount the card now. For example to mount the card on directory `/mnt`, you have to issue the following command:

```
mount /dev/mmcblk0p1 /mnt
ls /mnt
```

Now you can work with the device. Later, when you are done, you can unmount it again with

```
umount /mnt
```

9.12 USB Stick (Storage)

If a USB memory stick is inserted, it is available like a standard hard disk. Because there is usually no real hard disk connected, it is found as `/dev/sda`. If you have partitions on your USB stick, you have to access them as `/dev/sda1`, `/dev/sda2` and so on.

You can mount and unmount all the partitions now. For example to mount the first partition on directory `/mnt`, you have to issue the following command:

```
mount /dev/sda1 /mnt
```

To unmount again, issue:

```
umount /mnt
```

Remarks

If a USB storage device contains more than one partition, a device entry will be created for each partition, for example also `/dev/sda2` and `/dev/sda3`. In fact also a device entry `/dev/sda` without a number is available, that refers to the whole device, including all partitions. Some USB storage devices do not contain a partition table at all. Then only `/dev/sda` is created.

If more than one storage device is plugged in (for example via a USB hub), then the second device has the base name `sdb`, the third `sdc`, and so on.

9.13 RTC

Setting date:

```
date "2015-04-22 22:55"
```

Save current date to RTC:

```
hwclock -w
```

The time will automatically be loaded from the RTC at the next boot.

Note:

Make sure VBAT is connected to the module. Otherwise the RTC can not keep time.

9.14 GPIO

Sysfs

Note:

Deprecated since Linux kernel 5.15

You can setup and use GPIOs with the Sysfs system.

```
ls /sys/class/gpio
export        gpiochip128  gpiochip64   unexport
gpiochip0     gpiochip32   gpiochip96
```

Please refer the according “GPIO Reference Card“ document to know how the pins of the board correspond with the Sysfs-GPIO system.

Example:

Configure for example function pin COL0 (armStoneA9: J12 / PTD2) as output pin. The pin number on J12 is 3. The “GPIO Reference Card” shows in row `/sys/class/gpio/gpio#` the number 147. To get this pin into sysfs write:

```
echo 147 > /sys/class/gpio/export
```

This creates a new directory `gpio147` in `/sys/class/gpio` that is used for all further settings of this GPIO.

```
ls /sys/class/gpio/gpio147/
active_low  direction   edge         power        subsystem
uevent      value
```

Set pin as output:

```
echo out > /sys/class/gpio/gpio147/direction
```

Set pin to high level:

```
echo 1 > /sys/class/gpio/gpio147/value
```

Now the pin should have a high level (about 3.3V) which you can measure with a voltmeter. To set pin low again type:

```
echo 0 > /sys/class/gpio/gpio147/value
```

Now pin has a low level. To set a pin as input, write `in` into `direction`. Then you can read the current value with

```
cat /sys/class/gpio/gpio147/value
```

libgpio-tools

Note:

Available starting Linux kernel 5.15

To list all available GPIO chips run

```
gpiodetect
gpiochip0 - 32 lines:
line   0:      unnamed       unused   input  active-high
line   1:      unnamed "fts_reset_gpio" output active-high [used]
…
gpiochip1 - 32 lines:
line   0:      unnamed       unused   input  active-high
line   1:      unnamed       unused   input  active-high
…
```

You can name each GPIO by adding the "gpio-line-names" entry to the respective GPIO node in your Linux device tree. Please note that you always have to set the name for each GPIO of a chip.

```
&gpio1 {
     gpio-line-names =
      "GPIO_0","GPIO_1","GPIO_2","GPIO_3","GPIO_4","GPIO_5",
      "GPIO_6","GPIO_7","GPIO_8","GPIO_9","GPIO_J2_65","GPIO_11",
    …
};
```

Example

On PicoCoreMX8MP, use pin GPIO_1_54 on feature connector as output pin. This is pin 54 on the PicoCoreBBDSI connector J1, also available on pin 8 of the feature connector J11 on the PicoCoreBBDSI SKIT. The "GPIO Reference Card" for PicoCoreMX8MP in the row GPIO the GPIO1_IO1 which means that this GPIO is on gpiochip1 the IO number 1. If you have named your GPIO, you can also get this information by running

```
gpiofind <gpio_name>
```

To set the GPIO pin to high level run :

```
gpioset -m signal gpiochip1 1=1
```

Now the pin should have a high level (about 3.3V) which you can measure with a voltmeter. To end the gpioset press CTRL+c.

Please note, that the GPIO is only driven as long as gpioset is activie.

To set pin low again type:

```
gpioset -m signal gpiochip1 1=0
```

Now pin has a low level.

You can read the current value with

```
gpioget gpiochip1 1
```

9.15 Sound

You can use standard ALSA tools to play and record sound. There is a tool to test the sound output.

```
speaker-test -c 2 -t wav
```

This will say "Front left" and "Front right" on the appropriate line out channel. If you have a WAV file to play, you can use this command:

```
aplay <file.wav>
```

To record a file from microphone in (mono), just call

```
arecord -c 1 -r 8000 -f s16_le -d <duration> <file.wav>
```

SGTL500 Codec:

To record a file from line in, you first have to switch recording from microphone to line in. This can be done with

```
amixer sset 'Capture Mux' LINE_IN
```

WM8960 Codec:

There are two possible ways of configuring the WM8960. One is the interactive way with ALSA Mixer, but we'll concentrate on the shell commands.

To configure the general sound settings

```
amixer -c 0 set 'Headphone' 100%
amixer -c 0 set 'Speaker' 100%
amixer -c 0 set 'Playback' 100%
```

Every speaker (e.g. left and right) has it's own mixer. To activate them, execute

```
amixer -c 0 set 'Left Boost Mixer LINPUT3' unmute
amixer -c 0 set 'Left Output Mixer Boost Bypass' 100%
amixer -c 0 set 'Left Output Mixer LINPUT3' 100%
amixer -c 0 set 'Left Output Mixer PCM' unmute

amixer -c 0 set 'Right Boost Mixer RINPUT3' unmute
amixer -c 0 set 'Right Output Mixer Boost Bypass' 100%
amixer -c 0 set 'Right Output Mixer RINPUT3' 100%
amixer -c 0 set 'Right Output Mixer PCM' unmute
```

For the left and right channel.

To record a stereo file with high quality:

```
arecord -c 2 -r 48000 -f s16_le -d <duration> <file.wav>
```

To see what other controls are available, call amixer without arguments:

```
amixer
```

You can also use gstreamer to test sound.

```
gst-launch audiotestsrc ! alsasink
```

And to play a WAV file with gstreamer, you can use the following command:

```
gst-launch filesrc location=<file.wav> ! wavparse ! alsasink
```

9.16 Pictures

Buildroot:

There is a small image viewer program included called `fbv`. Just call it with the list of images to show. This will show a new image every ten seconds.

```
fbv -s 10 /usr/share/directfb-examples/*.png
```

To show possible program options use:

```
fbv --help
```

If you want to use a different framebuffer, set the variable `FRAMEBUFFER` accordingly. For example to use `/dev/fb2`, use the following command, before calling `fbv`.

```
export FRAMEBUFFER=/dev/fb2
```

Yocto:

Fbv is not available in Yocto. Instead, you can add the package fbida to your package

Add the following line to your conf/local.conf

CORE_IMAGE_EXTRA_INSTALL_append = " fbida"

```
fbi /<PATH_TO_IMAGE>.png
```

To show possible program options use:

```
fbi --help
```

fbi and fbv will write directly to the framebuffer, so you will have to make sure that no other graphical program like Weston/Wayland is running on the same virtual terminals.

You can change the virtual terminal with the command

```
chvt <1..63>
```

9.17 TFTP

There is a small program to download a file from a TFTP server. This can be rather useful to get some files to the board without having to use an SD card or a USB stick. For example to load a file `song3.wav` from the TFTP server with IP address 10.0.0.121, just call

```
tftp -g -r song3.wav 10.0.0.121
```

9.18 Telnet

If you want to use `telnet` to login from another PC, you have to start the telnet daemon

```
telnetd
```

However as this service is considered insecure, telnetd does not allow to log in as root. So you have to add a regular user, for example called "telnet".

```
adduser -D telnet
passwd -d telnet
```

The first commands adds the user "telnet" and the second command sets an empty password for this user.

Now you can log in from another PC with username telnet:

```
telnet <ipaddr>
```

9.19 SSH

Buildroot

You can connect to your device by SSH. But then you need to set a password for your root user

```
passwd
```

or create a new user by:

```
adduser <username>
```

Now you can connect via SSH by any host in the network with:

```
ssh <username>@<device-ip>
```

If for some reason the keys are expired you can calculate new ones. Therefore old keys have to be removed.

```
cd /etc
rm ssh_host_*
cd /etc/init.d
./S50sshd
```

Be careful with rm `ssh_host_*`. Just remove the files with the word `ssh_host_` and `key` in it. After that the startup script `S50sshd` should be executed again

```
/etc/init.d/S50sshd restart
```

> **Note**
>
> Please note that date and time must be valid on the board or login attempts with `ssh` will fail. The script `S50sshd` will only create new encryption keys if the directory `/etc` is writable. So if your rootfs is read-only, you have to remount it as read-write first before calling the script.

Yocto

In Yocto ssh is configured to allow root login and empty-password by default. Just connect via SSH by any host in the network with:

```
ssh <username>@<device-ip>
```

You can change these settings at:

```
vi /etc/ssh/sshd_config
```

9.20 VNC

> **Note**
>
> This is only supported in native X11 environments.

If you have no display attached or you want to connect by remote you can start the pre-installed (with the standard buildroot root filesystem) x11vnc program on your device by:

```
x11vnc
```

On any host in the network install a vnc-viewer program and connect to your board with:

```
vncviewer <device-ip>:0
```

9.21 RDP

Buildroot

To use the Remote Desktop Protocol, it is necessary to configure Buildroot. First go to the Buildroot directory and execute:

```
make menuconfig
```

then activate the RDP Compositor located at:

```
Target packages – Graphic libraries and applications (graphics/
test) – weston – RDP Compositor
```

Then reconfigure weston with:

```
make weston-reconfigure
```

When the process is finished, start the build process:

```
make
```

After the build has completed check if the **rdp-backend.so** file in

```
output/target/usr/lib/libweston-9
```

exists.

Save the rootfs on the board and boot the system. Establish a network connection between the development machine and the board. The RDP needs a security key. For this you must create a folder and generate a key:

```
cd /opt; mkdir rdp_keys
```

```
winpr-makecert -rdp -silent -path /opt/rdp_keys -n "rdp-security"
```

This command creates the two files:

```
rdp-security.crt
rdp-security.key
```

Now you have to start the weston service by executing:

```
weston --backend=rdp-backend.so --shell=desktop-shell.so --no-
clients-resize --rdp4-key=/opt/rdp_keys/rdp-security.key
```

The service is now up and running.

On the development machine you have to install the FreeRDP package:

```
sudo dnf install freerdp
```

To connect to the board you must start the client with:

```
xfreerdp /u:root /v:<BOARD-IP-ADDRESS>:3389
```

Yocto

First off, you must ensure that the FreeRDP package is installed on the board.

Simply add:

```
IMAGE_INSTALL_append = " freerdp"
```

to your **local.conf** file located in the build directory.

Next, you must create the **weston_<VERSION>.imx.bbappend** file in the

```
sources/meta-fus/recipes-graphics/wayland
```

directory. Create the folder wayland if necessary.

Add the following lines of code:

```
  EXTRA_OEMESON_remove = " -Dbackend-rdp=false"
  EXTRA_OEMESON_append = " -Dbackend-rdp=true"
  PACKAGECONFIG_append = " rdp"
  PACKAGECONFIG[rdp] = "-Dbackend-rdp=true, -Dbackend-rdp=false,
freerdp"
  EXTRA_OECONF_append = " --enable-rdp-compositor"
```

Save the file and run your build. The remaining steps to establish a connection are identical to the buildroot release. So please follow these steps described earlier in the buildroot section.

9.22 Bluetooth

Bluetooth is available on some modules. Buildroot and Yocto offer support for the official Linux Bluetooth protocol stack (bluez).

Before fsimx8mm-B2023.11

Silex wlan chip

To attach the bluetooth device you must first create a hci device. Physically bluetooth device is available on tty0:

```
hciattach ttymxc0 texas 38400 flow
```

Azurewave wlan chip

If not already done, start the Bluetooth daemon:

```
/usr/libexec/bluetooth/bluetoothd --compat &
```

After powering up the device, bluetooth is working generally.

```
hciconfig hci0 up
```

To scan for bluetooth devices you can use the hcitools.

```
hcitool scan
```

More information to bluez stack can be found at http://www.bluez.org .

After fsimx8mm-B2023.11

The hcitool is deprecated and not included in the bluez-utils package any more.

You can use the bluetoothctl command instead to bring up the bluetooth device.

If you still need the hci tools you can install the bluez-utils-compat package.

Azurewave wlan chip

If not already done, start the Bluetooth daemon:

```
/usr/libexec/bluetooth/bluetoothd --compat &
```

After powering up the device, bluetooth is working generally.

```
bluetoothctl power on
```

To scan for bluetooth devices you can use the bluetoothctl.

```
bluetoothctl scan on
```

More information to bluez stack can be found at http://www.bluez.org .

10 Graphical Environments

Linux offers a few different ways for applications to display their graphical user interface. A display server that uses protocols like X11 or Wayland can be used. Or the application can write directly to the frame buffer or render through EGL.

The default environment F&S uses is the X.Org Server. It is used when a root file systems is compiled with the '..._std_defconfig'. This configuration is destined for X applications.

As opposed to this the '..._qt5_defconfig' is destined for Qt applications. It doesn't contain a display server. There are several reasons for this. First the compilation of X server packages next to Qt5 packages is not intended in Buildroot and leads to conflicts. Second the common approach for Qt applications is to render directly through EGL. The Qt application runs in full-screen mode mostly and all configuration is done within the Qt framework. In this case a display server is omitted because it is not needed.

The third configuration for Buildroot F&S offers is the '..._min_defconfig'. It is a very small configuration. It contains almost no additional packages and no display server either. This configuration was designed to give a starting point for customers that want to have a very small and basic configuration they can extend. A good knowledge about configuration, needed packages and dependencies is needed therefore. Because it is a very basic configuration setting up another graphical environment or changing some settings is done by adjusting the '..._std_defconfig' '..._wayland_defconfig' or '...qt5_defconfig' most of the time.

Note

Releases, starting 2021 do not support X11 as graphical environment any more. Instead Weston/Wayland is used as graphical environment here.

All chapters about X11 do not apply to these releases!

10.1 Rendering

Rendering which is vital for showing up a GUI is a complex topic. There are lot's of techniques, protocols and coherence's that came by time. Let's have a quick look at the evolution, current technologies and how parts are connected.

10.1.1 Weston/Wayland

Wayland is another Graphical Environment. It is meant as the successor to X11, but without the overhead of a dedicated graphics library or a rendering server. This allows for a more efficient use of graphics hardware at the cost of less flexibility in networking capabilities.

In Wayland, all applications are rendering their window contents themselves. There is no strict rule on how to do this. They can use libraries like pixman, cairo, pango, freetype, or full blown widget libraries like Qt or GTK+, or OpenGL/OpenGLES for 3D features and even

Vulkan in the future. By using the Direct Rendering Infrastructure (DRI), Wayland programs can also get easy access to hardware acceleration. The result of the rendering is placed in a bitmap in RAM.

Then there is the so-called Compositor. The Compositor collects those parts of the bitmaps of each application that are visible on the screen, i.e. that are not covered by other windows and that are not off-screen, and copies them to the framebuffer. If an application updates some part of the content within its bitmap, for example as the result of some user action, it tells the Compositor about this and the Compositor will fetch the affected part of the application's bitmap again and refreshes the on-screen content with it.

So rendering is done by the applications, and composing the final screen content is done by the Compositor. And Wayland defines the protocols and data structures that are used for the communication between client applications and Compositor. Wayland itself does neither participate in drawing/rendering, nor is it responsible for the position and stacking of windows (the so-called scene graph).

The Compositor is part of the specific GUI. For example Gnome has its own Compositor called "mutter", KDE has its own Compositor named "kwin" and so on. The Compositor also takes the role of the Window Manager, so it is also responsible for the look and feel of the windows on the screen and the desktop in general. This is the part where the GUI systems bring in their specific concept of interaction with the user.

Wayland has an own reference implementation of a Compositor called Weston. This Compositor is used in cases where the selected GUI does not provide an own Compositor, which is the case on F&S boards.

So in fact most settings for Wayland on F&S boards are actuallydealing directly with Weston, which is why this GUI is often called Wayland/Weston.

10.1.2 X Server technology

In former times there was mainly 2D content. An application transferred it’s content to the X server that sent it directly to the graphic hardware. Therefore the X Server ran as root.

In the next time when OpenGL came up the **GLX** protocol was invented. It’s an extension of the X protocol that transfers OpenGL commands over the X Server protocol. 3D OpenGL content could be sent to the X Server and from there to the hardware now. Because the X Server is always standing in the middle and an application can’t interact with the graphic hardware directly this is called **indirect rendering**.

To solve that issue the **Direct Rendering Infrastructure (DRI)** came. OpenGL applications could render content faster by talking to the DRI directly. The X Server still had to communicate to the graphic hardware directly running as root. Framebuffer applications used the framebuffer driver.

Today the X Server, OpenGL applications and framebuffer applications use the direct rendering infrastructure to access the graphic hardware. Direct Rendering is very important especially for the performance of 3D applications. 2D X applications are often still using the indirect rendering because the performance is good enough. But 2D X applications can render via OpenGL too.

10.1.3 Qt with OpenGL, EGL, EGLFS and more

Let’s explain how Qt applications renders to the screen and explain a few terms therefore.

OpenGL is an API for rendering graphics mostly in 3D and also in 2D. It's an open specification. Mesa3D implements OpenGL via software rendering as well as drivers for hardware accelerated rendering for some graphic cards. The imx-vivante driver is not included in Mesa3D. It's an independent driver.

OpenGL ES is a subset of OpenGL. It's a 2D/3D rendering API for embedded systems. OpenGL ES 2.X provides full programmable 3D graphics. The vivante graphic card on i.MX6 and i.MX6SX SOCs is able to render OpenGL ES commands. Vivante provides (mostly closed source) libraries that implement the OpenGL ES API and communicate to the graphic hardware. In buildroot menuconfig you can activate some GPU examples in *Target Packages → Hardware handling → Freescale i.MX libraries → imx-gpu-viv.*

EGL is an API between a Khronos rendering API e.g OpenGL ES and a window system e.g. Wayland or the graphical environment of Android. EGL can handle multiple 2D and 3D rendering contexts. It manages buffers, synchronization and more. The EGL API does not contain platform-specifics. Usually the window manager keeps an eye on that. However a single Qt application may want to run without a window manager.

Therefore Qt provides **EGLFS** and so called **EGL device integration plugins** (e.g. linuxfb) that are loaded dynamically. Both solutions provide surfaces to draw to. In addition EGLFS can provide software-rendered windows similar to a window manager. The Qt documentation recommends EGLFS as “the recommended plugin for modern Embedded Linux devices that include a GPU” and prefer it over EGL device integration plugins.

10.2 Weston/Wayland Configuration

10.2.1 General behaviour

Weston/Wayland is compiled using the ‘..._wayland_defconfig with Buildroot or DISTRO=fus-imx-wayland in Yocto.

On start it shows a simple panel at the top of the screen, containing a clock on the right and an icon to open terminals on the left.

Weston is started automatically by default. If you have to start it manually call

```
weston --tty 7
```

The central file to configure Weston is located at

```
/etc/xdg/weston/weston.ini
```

The weston.ini file is composed of a number of sections which may be present in any order, or omitted to use default configuration values.

Rotate Display

To rotate the display 90 degrees clockwise, add the following lines to the [output] section:

```
[output]
```

```
name=[output name]        # fbev for imx6; DSI-1 for imx8¹
transform=rotate-[degree] # can be 90/180/270
```

Fullscreen

To remove the toolbar and start all applications in full-screen you can use the kiosk-shell

```
[core]
shell=kiosk-shell.so
```

Alternately you can remove the toolbar by adding the following lines

```
[shell]
panel-position=none
```

Applications then will get started in windows unless they request full screen.

10.2.2 Weston touch calibration

You can connect a 4-wire resistive touch or a capacitive touch to your device. When starting Linux for the first time you need to calibrate your touch with command

```
weston-touch-calibrator -v /dev/fb0
```

This will show a crosshair on the display in the top left corner. After you have touched it with your finger, the crosshair will move to the top right corner. Continue to touch these marks until all four corners are done. Now the touch is calibrated. The program will print a set of calibration data before exiting.

However this calibration is lost if you restart the board. To make the changes permanent, you have to add this to a udev rules file.

Create a new .rules file at /etc/udev/rules.d/ and add your calibration matrix values to it:

```
vi /etc/udev/rules.d/touchscreen.rules
```

```
ENV{LIBINPUT_CALIBRATION_MATRIX}="x.xxx x.xxx x.xxx x.xxx x.xxx x.xxx"
```

You may have to remove existing touchscreen rules.

> **Note**
>
> If weston-touch-calibrator is not installed to your Image in Yocto you can add the following line to you conf/local.conf and restart the build:
> IMAGE_INSTALL_append = " weston-examples"

10.2.3 Xwayland

It's still possible to run many X-applications on Wayland using the Xwayland server. Xwayland is a stand-alone Xserver without graphic-specific drivers. It sends the im-

[1] This may change regarding your display type

age data directly to Weston, which outputs it to the screen using hardware acceleration.

However, 3D-accelerated X11 applications are not supported on Xwayland. The native Wayland backend has to be used here.

Xwayland can be build by using the *..._xwayland_defconfig* with Buildroot or *DISTRO=fus-imx-xwayland* in Yocto.

You can test you Xwayland installation by running

```
xterm
```

10.3 X.Org Server Configuration

10.3.1 General behaviour

The default GUI compiled with '..._std_defconfig' just shows a rudimentary X-Window desktop under a Matchbox window manager. You can start a terminal program and a system load monitor from the starter menu. You can also click on the desktop icons and open them, but there are no further X applications installed. You can connect a USB mouse or USB keyboard (e.g. by using a USB hub) and move the mouse cursor or type commands to the terminal window.

The whole system is very simple and just demonstrates how a GUI could be implemented. We don't want to get a too large default root filesystem.

The X11 window manager is started automatically by default. If you have to start it manually call

```
startx
```

You can start some X applications on the command line e.g.:

```
export DISPLAY=:0
xclock &
xeyes &
xcalc &
```

In matchbox all applications are shown fullscreen. Matchbox does not support tiled or overlapped windows. You can switch between running applications by clicking on the top left drop-down menu.

The X server is started with script file /etc/init.d/S35x11. So if you don't want this GUI started at every boot, just rename this script to something that does not start with S and two digits. For example rename it to X35x11. Then you can rename it back any time you want

10.3.2 GPU support

To use the hardware accelerated graphics driver make sure that the following configuration lines are available in the `xorg.conf` file:

```
Section "Device"
    Identifier "i.MX Accelerated Framebuffer Device"
    Driver "vivante"
    Option "fbdev"              "/dev/fb0"
    Option "vivante_fbdev"      "/dev/fb0"
    Option "HWcursor"           "false"
EndSection
```

Listing 2: Hardware accelaration configuration in xorg.conf

10.3.3 X.Org Server touch calibration

You can connect a 4-wire resistive touch or a capacitive touch to your device. When starting Linux for the first time you need to calibrate your touch with command

```
xinput_calibrator
```

This will show a crosshair on the display in the top left corner. After you have touched it with your finger, the crosshair will move to the top right corner. Continue to touch these marks until all four corners are done. Now the touch is calibrated. The program will print a set of calibration data before exiting.

However this calibration is lost if you restart the board. To make the changes permanent, you have to add this to the `xorg.conf` file. This can be done by creating a sub directory `/etc/X11/xorg.conf.d` and adding a file `99-calibration.conf` that contains the calibration data that was reported by `xinput_calibrator`.

10.4 Qt Environment

If you use a Qt based root file system nothing will be shown on the display until you start a Qt application. Qt can be compiled with the linuxfb device integration plugin that simply writes to the framebuffer or with the recommended EGLFS variant. The first one is used by fsimxul_qt5_defconfig because i.MX6UltraLite doesn't have a 3D graphic engine. The latter one is used by '..._qt5_defconfig' of i.MX6 and i.MX6Solo X. If you have compiled a root file system containing both you can switch between them using the QT_QPA_PLATFORM argument. To start the gradients example using the framebuffer type:

```
QT_QPA_PLATFORM=linuxfb:fb=/dev/fb0 ./gradients
```

To start the gradients example using eglfs you can type:

```
QT_QPA_PLATFORM=eglfs ./gradients
```

Omitting the argument using EGLFS will be tried as default platform. This will not work for i.MX6UltraLite.

```
./gradients
```

The argument can also be set as environment variable:

```
export QT_QPA_PLATFORM=linuxfb:fb=/dev/fb0
```

Or it can be given as argument to the application:

```
./gradients -platform linuxfb
```

The environment variable will prefered over the application argument.

10.4.1 Qt touch configuration

You can connect a 4-wire resistive touch or a capacitive touch to your device. EVDEV is used for calibration. Because EGLFS knows platform-specifics like screen resolution it needs no calibration in most cases. The linuxfb platform plugin needs more attention.

Use the following argument on application start to rotate the touch orientation:

```
QT_QPA_EVDEV_TOUCHSCREEN_PARAMETERS="rotate=180" ./gradients
```

It is also possible to invert the x or y coordinates with the argument ‘invertx’ or ‘inverty’. Arguments can be chained using a colon as separator:

```
QT_QPA_EVDEV_TOUCHSCREEN_PARAMETERS="rotate=180:inverty" ./gradients
```

If you want to specify the input device the evdev touch handler should use supply the path like this:

```
QT_QPA_EVDEV_TOUCHSCREEN_PARAMETERS="/dev/input/event1" ./gradients
```

If no path is given, Qt tries to find the device by walking through the available nodes.

10.4.2 Running Qt5 Example

There are several Qt examples shipped with Qt that you can use to test your Qt environment and get a feel about functionality and behavior. For example the gradients example used in previous statements is located here:

```
/usr/lib/qt/examples/widgets/painting/gradients/
```

Sometimes an application has incorrect screen or window geometry settings and is not displayed in full screen. In those cases you need to adjust the height and width of the appliation. This is done with the following argument:

```
./gradients -qwindowgeometry 800x480
```

Sometimes the screen becomes dark and nothing can be seen. Most time this will happen when the CPU is idle for an amount of time. You can activate the CPU to run with full performance, so that the display will be available again.

```
echo "performance" > /sys/devices/system/cpu/cpu0/cpufreq/scal-
ing_governor
```

The following figure shows the gradients example running on a screen. If the touch is calibrated correctly you can change the settings on the right.

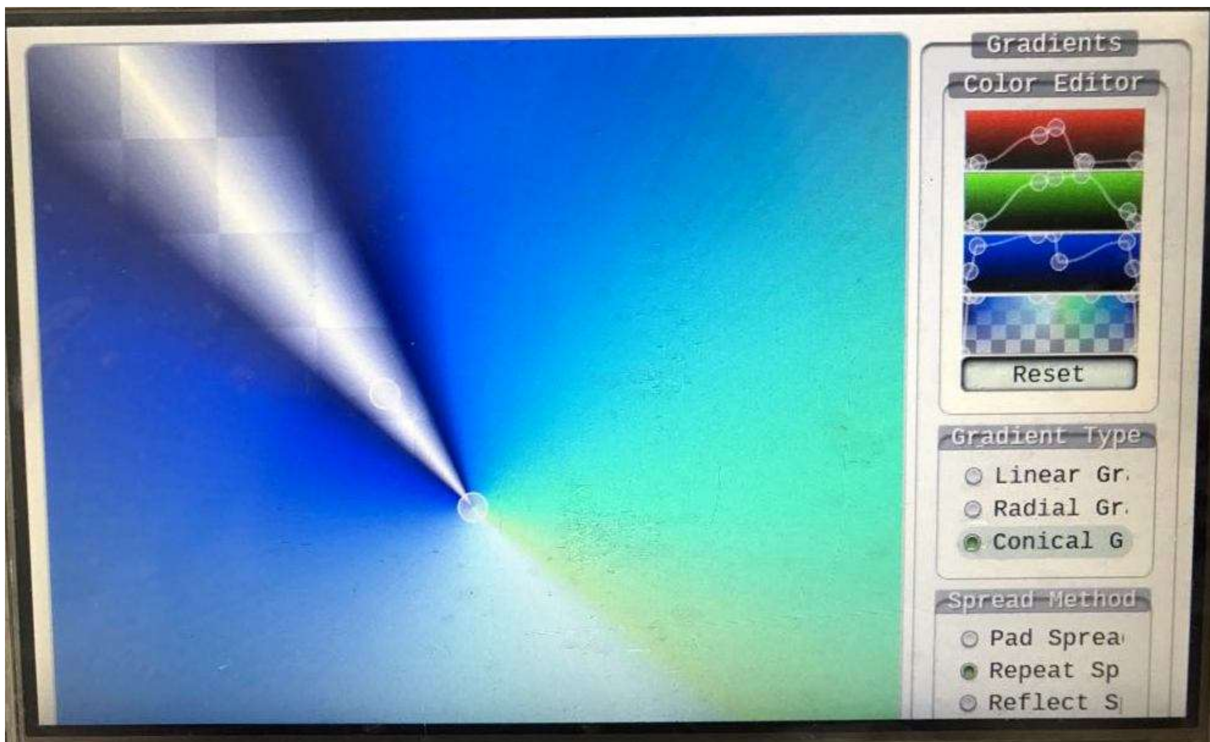


Figure 50: Qt Example (Gradients).

11 Compiling the System Software

When working for the first time with the build environment, you may be a little bit confused by all the different tools, toolchains and directories. For example you have a toolchain on the PC that generates code for the PC. And now we will install an additional toolchain that is on your PC that generates code for your board. This is called a cross-compile toolchain. (Theoretically it would also be possible to have a third toolchain that is on your board and generates code for the board, but we at F&S do not support this native compilation environment as compilation is rather slow when done on the board itself.)

All these toolchains have their own include files, their own libraries, their own configuration files and so on. Apparently it can happen very quickly, that you are in the wrong directory or that you use the wrong tool. But some of these commands may have very negative effects if executed in the wrong place. For example when you want to erase some configuration file that is meant for the `etc` directory on your board, but you accidentally erase the local file in `/etc` on your PC instead, then this may by dangerous or even fatal. The worst case is that you remove some larger parts and your development PC will not react to input anymore.

So we strongly recommend that you work as a normal user and not as the superuser “root” all the time. As a normal user you are not allowed to erase such system critical files and such commands simply won't do any harm. If you do have to enter privileged commands from time to time, please use the `sudo` command that grants super user rights for the next few minutes. And when having to type `sudo` in front of critical commands, then this automatically reminds you to be careful for this step.

Please see the documentation `AdvicesForLinuxOnPC_eng.pdf` for a description of how to set up `sudo`.

F&S supports the build environments Buildroot and Yocto to build the system software. Both have the same basic idea. The user defines what packages (tools, libraries, applications) he wants to include in his system by changing some configuration and then starts the build process. Then both environments do the following steps.

- Download all required source code packages from their original web sites all over the world
- Apply some patches so that the software can be cleanly cross-compiled
- Configure each package according to the user specification
- Compile all packages
- Create a filesystem image that can be downloaded to the board as root filesystem

However how the two environments work in detail is rather different. This is why we handle them in two different chapters.

Important

The configurations and images that F&S provides as part of the releases are meant as examples. They typically contain a small GUI, some multimedia environment and a set of tools to access interfaces like SPI, I²C, CAN, USB, Ethernet and WLAN.

However these configurations and images are not meant to be used in end products. We may have included software packages and audio/video codecs that need a license to be used. We most probably have included packages that you'll never need. There will be packages missing that you do need. And there is no security whatsoever. A system where the root user can log in without a password is definitely not secure.

F&S declines all responsibility for problems and damages resulting from the use of these configurations and images.

So we strongly recommend that you set up your own configuration with only those packages that you really need. You can use our minimal configuration as a starting point. Then add all packages that you need. And finally add packages for the security, like user management, firewall, SE Linux or similar.

11.1 Buildroot

11.1.1 Install Cross-Compile Toolchain

Note

Only necessary before B2026 releases, as from this version on we let Buildroot build its own toolchain. For Information on how to compile software outside of Buildroot on newer releases, continue at 11.1.9

The cross-compile toolchain is needed to compile U-Boot, the Linux Kernel and the Buildroot package. We recommend installing it globally for all users in directory `/usr/local/arm`. This is the directory that is preset in some configuration files. If you use a different directory, you may have to modify these configuration settings before being able to build packages, for example in Buildroot.

The global installation needs superuser rights, so we need to use `sudo`. First create the directory and unpack the file from the `toolchain` subdirectory.

```
sudo mkdir -p /usr/local/arm
sudo tar xvf fs-toolchain-8.3-armv7ahf.tar.bz2 -C /usr/local/arm
```

Now add this directory to your global `PATH` variable and set environment variables `ARCH` and `CROSS_COMPILE` for compiling the Linux kernel:

```
export PATH=$PATH:/usr/local/arm/fs-toolchain-8.3-armv7ahf/bin
export ARCH=arm
export CROSS_COMPILE=arm-linux-
```

Note

You probably have to edit some global or local bash profile to make these two environment changes permanent, for example `/etc/profile` or `~/.bashrc`.

11.1.2 Installing the mkimage Tool

The mkimage tool is used if the kernel should be compiled as uImage, as we did in the past. Additionally it is used to compile a U-Boot script. Copy the `mkimage` tool from the `toolchain` subdirectory to `/usr/local/bin`.

```
cp mkimage /usr/local/bin/
```

Check if `/usr/local/bin` is present in your `PATH` variable:

```
echo $PATH
```

If it is not there you have to extend your `PATH` variable:

```
PATH=$PATH:/usr/local/bin
```

This ensures temporarily that means for the current user in the current shell that the binary is found when calling mkimage.

> **Note**
>
> You probably have to edit some global or local bash profile to make this change permanent, for example `/etc/profile` or `~/.bashrc`.

11.1.3 Unpacking the Source Code

The source code packages are located in the `sources` subdirectory of the release archive. We will assume that you want to create a separate build directory where you extract the source code and build all the software. The easiest way is to extract U-Boot, Linux kernel, Buildroot and Examples next to each other, so that the top directories of their source trees are siblings.

We have prepared a shell script called `setup-buildroot` that does this installation automatically. Just call it when you are in the top directory of the release and give the name of the build directory as argument.

```
cd <release-dir>
./setup-buildroot <build-dir>
```

Add option `--dry-run` if you want to check first what this command will do. Then only a list of actions will be output but no actual changes will take place. For further information simply call

```
./setup-buildroot --help
```

> **Note**
>
> In earlier releases, this script was called `install-sources.sh`. We have renamed the script to be more consistent to Yocto where a similar script is called `setup-yocto`.

If you prefer to do the installation by hand, well, the script more or less executes the following commands, just with some more checks and directory switching.

```
mkdir <build-dir>
tar xf u-boot-2018.03-<arch>-<v>.tar.bz2
tar xf linux-4.9.88-<arch>-<v>.tar.bz2
tar xf buildroot-2019.05.1-<arch>-<v>.tar.bz2
tar xf examples-fus-<v>.tar.bz2
ln -s linux-4.9.88-<arch>-<v> linux-fsimx6
```

Note

Unpacking by hand is not supported anymore since B2026 releases and replaced by a git based system.

The symbolic link in the final command is required by Buildroot. It provides a constant name reference to the kernel source tree from the point of view of Buildroot, no matter how you actually call this directory. So for example if you want to use a different version of the kernel with a different directory name, just change the symbolic link to point to your other directory and Buildroot will automatically work with this version without having to change the Buildroot configuration.

11.1.4 Running Docker

Note

Only available starting B2026 releases

Starting F&S Buildroot 2025.02 releases can be build inside a docker container. This has the advantage, of a simpler and more consistent dependency management, a better reproducibility of the builds an an easier CI/CD pipline intigration.

First make sure that docker is installed on your building machine:

```
docker --version
```

If a docker version is printed run the following command to start the F&S Docker environment:

```
./setup-buildroot <build-dir> --docker
```

If you run this command for the first time on you building machine, a new docker container will be created. This may take some time.

When the docker container is available, you will get a new shell from the docker container inside your <build-dir> directory.

Change to the `buildroot-fus` directory to proceed with the next building steps.

```
cd ~/buildroot-fus
```

11.1.5 Building with Buildroot

Buildroot allows the creation of arbitrary root fileystems. In a menu based configuration system you can select which packets you want to include in the image. Buildroot knows all the

dependencies between the packages and will automatically add all necessary libraries. When you start the build process, Buildroot will perform the following steps.

- Download the selected source code packages from the original web sites all over the world
- Apply some patches to make compilation go smoothly when cross-compiling
- Build all selected packages
- Combine all binaries in a root filesystem image that you can download to the board

So basically the task for creating a root filesystem for your application is to select which packages you need, add your own application and let Buildroot build and combine everything into the filesystem image.

We have added four different <arch> configurations to Buildroot.

- <arch>_min_defconfig: A minimal one that just includes Busybox and the GLIBC library.
- <arch>_wayland_defconfig: The standard image that includes wayland support with additional features like gstreamer for multi-media, ALSA for sound support This is the version that is also included in our Starterkits and it is a good starting point for exploring the device and its capabilities.
- <arch>_xwayland_defconfig: Like the wayland image but with xwayland support for x11 applications.
- <arch>_qt6_defconfig: An Image with a set of QT libaries installed.

Again simply go to the Buildroot directory, issue a configuration command to tell Buildroot to build for the <arch> architecure, and then start the build process. The following commands will build the standard <arch> configuration.

```
cd buildroot-fus
make <arch>_wayland_defconfig
make
```

Please note that Buildroot needs network access to be able to download all the packages. It will save them in a download directory. This is usually the `dl` folder in the Buildroot directory, but you can change this by setting Linux environment variable `BR2_DL_DIR` to point to a different directory. Unless you delete this directory, Buildroot will keep the downloaded files there and will only do additional downloads if you add more packages to the root filesystem.

Note

If Buildroot fails to find a package for some reason, you can search the internet for it and download it manually from another site. Then store it in the `dl` directory and resume the build process with `make`. If F&S knows that a file is not available, we sometimes also add the file directly to our release in the `dl` directory.

Compiling Buildroot the first time may take quite a while, probably more than an hour even on a very fast computer, so don't be surprised. If you add the graphic environment QT to it, it

can easily be two hours or more. But later when only minor modifications need to be re-done, an “updating” compilation is usually done in a few minutes or even seconds.

The whole build process takes place in the directory `output/build`. Buildroot will also compile some utilities for the host PC needed for the build process. These files will be installed in `output/host`. Also the libraries that are built as intermediate steps are stored somewhere in the `output` directory. Buildroot actually also copies parts of the toolchain to this directory and builds some wrappers around them so that the toolchain will use the newly created libraries from Buildroot and not the globally installed libraries.

The final result are the root filesystem images in the sub-directory `output/images`. The default configuration will build two different images. A file called `rootfs.ubifs` that holds a UBIFS based filesystem meant to be stored in NAND flash on the board. And a file called `rootfs.ext4` that can be used on SD cards or that can be mounted locally on the PC to be exported to the board via NFS.

11.1.6 Cleaning a Buildroot build

Adding packages is usually straightforward. Just select the package in menuconfig, exit menuconfig by saving the config file, rebuild the root filesystem, done! However removing packages is slightly more complicated. Buildroot assembles all files that should go to the target board in `output/target`. This is the base for packing the root filesystem image later. But it does not keep track of what files of this `output/target` directory are installed by which package. If a package is added, the package itself will install any new files in this directory. If a package is recompiled, the package itself will re-install these files in this directory and the new files will simply overwrite the old ones. But what files should be deleted if a package is removed? Packages usually do not have an uninstall step in their build process. So Buildroot simply does not know this. And its solution is rather simple: it just leaves all files in `output/target` and never removes anything. So don't be surprised if you deselect some packages in menuconfig, rebuild everything and the root filesystem does not shrink in size at all. The reason is that all the old binary files are still there.

For that it actually makes sense to do a clean rebuild from time to time. Then the whole `output` directory including `output/target` is deleted and all packages are rebuilt from scratch. Because deselected packages are not built again then, the new filesystem image will reflect the new situation and will be considerably smaller.

```
make clean
make
```

Especially if you have finished your development, you should do a clean rebuild for your final root filesystem that will go into the field to get a minimal and optimized filesystem suited for your needs.

Some Notes About The Kernel Build Process in Buildroot

The kernel compilation process works a little bit different compared to all other Buildroot packages. Usually the source code of a package is downloaded from somewhere, extracted, patched, configured, compiled and added to the root filesystem. But the Linux build process for <arch> will not download a package. Instead it will use the Linux source directory next to the Buildroot directory that was unpacked in Chapter 11.1.3 on page 153. This is done by us-

ing rsync. Rsync replaces the variant with links that was used in older versions. The advantage not to use a regular package for the kernel is that modifications will still be available after a `make clean`. On a regular package changes made to the extracted version of a package will get lost after a `make clean`. For permanent changes patch files are used of course but obviously that is not practical for developing.

The first time rsync synchronizes kernel code to Buildroot will take about three minutes. If this is done once in the next time synchronizing will be faster of course.

11.1.7 Rebuild a single package

Sometimes it is necessary to rebuild a single package. When adding patches, changing the configuration of the config or increasing a package version, Buildroot does not see the change and will skip the package. To rebuild with your current configuration you can use the reconfigure make targets in buildroot:

```
make <package name>-reconfigure
```

e.g.

```
make linux-reconfigure
```

if you need to restart from a completely clean state in a package you can do this with the following commands for removing build data

```
make <package>-clean
```

or the whole directory

```
make <package>-dirclean
```

afterwards you can rebuild the package either with

```
make <package>
```

or the full Image

```
make
```

11.1.8 Add Packages to an Image

If you want to add or remove packages, or if you just want to change some settings, call

```
make menuconfig
```

Then do your modifications, exit menuconfig by saving the new configuration and then rebuild by calling

```
make
```

again.

11.1.9 Compiling the Toolchain

Buildroot can compile its own Toolchain. This is done when building a buildroot Image, but can also be done by invoking its target after setting the defconfig:

```
make <arch>_defconfig
```

Afterwards you can build the toolchain by either building a whole image or by invoking its make target

```
make gcc-final
```

This will build the toolchain used by buildroot, by adding it to your PATH variable you can use it outside of buildroot.

```
export PATH=$PATH:<build dir>/output/host/bin
```

Afterwards you have access to the buildroot build tools for cross compilation.

11.1.10Compiling U-Boot

Starting B2021 releases the U-boot is built as part of the Buildroot build process. (See11.1.5)

You can still build U-boot this way, but t is not necessary any more.

In your build directory, simply go to the U-Boot source directory, run a configuration command to tell U-Boot to compile for the <arch> architecture, and then start the build process. This can be done with the following three commands.

```
cd u-boot-fus
make <arch>_defconfig
make
```

Note

Releases older then B2026 will have different names for the U-Boot directory, but the following commands are the same.

Older Releases might also contain an <arch>_mmc_defconfig for Boards booting from mmc. Those are not necessary on current releases.

On arm Boards, this will build the file `uboot.nb0`, that can be downloaded to your board, either in NBoot or the currently installed U-Boot. The file is padded to exactly 512 KB in size (524288 bytes). The build process will also create a slightly smaller `uboot.fs`. This file will only work for serial download in NBoot, but then download is slightly faster due to the smaller size.

On arm64 Boards `uboot.fs` will be generated as the Image to be installed. The size in this Image is dynamic.

11.1.11 Compiling the Linux Kernel

Usually you would build the Linux Kernel as part of the Buildroot build process (see next chapter). Then automatically all kernel driver modules will be included in your root filesystem. However sometimes you may want to compile the kernel separately from Buildroot. Then the sequence is as follows.

Go to the Linux source directory and run a configuration command to tell Linux to build for the <arch> architecture. Please verify that you have set environment variable `ARCH` to the value `arm` and environment variable `CROSS_COMPILE` to the value `arm-linux-` or the make tool will search the wrong directory for the default configuration. Finally start the build process.

```
cd linux-fus
make <arch>_defconfig
make
```

> **Note**
>
> Releases older then B2026 will have different names for the Linux directory, but the following commands are the same.

The final kernel image can be found in `arch/arm/boot/`. The Images are called `zImage` on arm and `Image` on arm64

If you want to modify some kernel settings you can call

```
make menuconfig
```

and then recompile the kernel with

```
make
```

To build the device tree, use the device tree name with a vendor prefix as target for `make`. In the F&S environment for i.MX6, this name is built by the board name, the abbreviation for the CPU type (`dl` for Solo/DualLite, `q` for Dual/Quad) and finally filename extension `.dtb`, all in lowercase. For example to build the efusA9 device tree for Solo or DualLite CPU, call

```
make fus/efusa9dl.dtb
```

The resulting file can be found in `arch/arm/boot/dts/fus/efusa9dl.dtb`.

If you are not sure, what is the right name for the device tree, check variable `platform` in U-Boot. The result (appended by `.dtb`) can be used to identify the device tree.

```
printenv platform
```

In the sdcard subdirectory there are prebuilt .dtb files where you can have a look how the naming pattern works.

11.2 Yocto

Yocto is a system to build a Linux distribution. So it is not a Linux distribution by itself, but it will help building a Linux distribution. This includes the toolchains, the bootloader, the kernel, the root filesystems and Yocto can even build a package repository where the end user can install additional packages from.

Yocto is organized in layers. Layers may exist next to each other, but usually they have a hierarchical order, where the next layer is based on the previous layer. The F&S Yocto release is actually only the NXP/Freescale Release BSP, where an additional F&S layer is added on top to provide support for the boards and modules from F&S. The NXP/Freescale Release BSP itself is based on the regular Yocto release and simply adds some NXP/Freescale layers, to support the NXP CPUs and Evaluation Boards. Yocto itself is again based on Open Embedded, and adds some additional Yocto specific software layers, called "Poky".

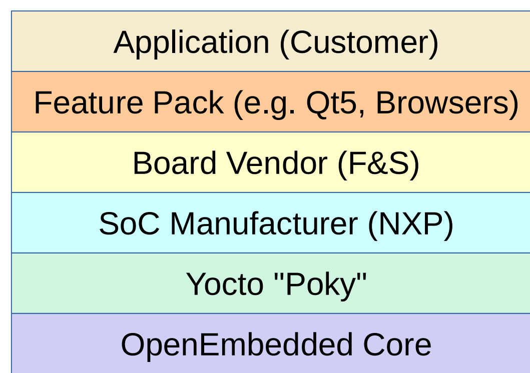


Figure 51: Layer architecture of Yocto

So at the core, Yocto is an Open Embedded based system. Open Embedded uses recipes, like they are used when baking a cake. The recipe describes what you need as ingredients, and then it tells step by step what tasks have to be done to produce the final cake. So a recipe in Yocto also names the ingredients (source packages, patch files, configuration settings, etc.) and then tells step by step what tasks have to be done to produce the target. For example how sources are unpacked and patched, how they are "converted" to binaries, how tools are installed to help the build process, how different files are grouped to root filesystems or other images, and so on. To add new packages or build strategies, you simply have to add some more recipes. This makes Yocto/Open Embedded a rather powerful build system where you can have any influence that you like.

On the other hand you actually have to write a seperate recipe even for small tasks. Even if you just want to assemble your own set of packages to build your private root filesystem, you need to write a recipe. There is no easy menu or GUI based package selection like for example in Buildroot. Everything works with recipes. Of course you can build some default images, but at the moment where you want to do something different, you have to modify an existing recipe or you have to write your own one.

So you actually have to learn how to write recipes if you want to work with Yocto. There is no way around this. It may also be necessary to learn how configurations and layers are written, because the most simple way to add your own application is by adding a separate layer for it, on top of the provided Yocto eco system.

These recipes use an own scripting language, based on Python. And there is a program called `bitbake` that can parse these recipes and execute the steps described in them. So building a Yocto package, root filesystem or even full target image as a whole, you have to use `bitbake`.

11.2.1 Installing the mkimage Tool

The `mkimage` tool is used if the kernel should be compiled as uImage, as we did in the past. Additionally it is used to compile a U-Boot script. Copy the `mkimage` tool from the `toolchain` subdirectory to `/usr/local/bin`.

```
cp mkimage /usr/local/bin/
```

Check if `/usr/local/bin` is present in your `PATH` variable:

```
echo $PATH
```

If it is not there you have to extend your `PATH` variable:

```
PATH=$PATH:/usr/local/bin
```

> **Note**
>
> You probably have to edit some global or local bash profile to make this change permanent, for example `/etc/profile` or `~/.bashrc`.

11.2.2 Unpacking the Source Code

The source code packages are located in the `sources` subdirectory of the release archive. We will assume that you want to create a separate build directory where you extract the source code and build all the software.

> **Note**
>
> Yocto needs quite a lot of space. The standard root filesystem as provided by F&S takes about 100GB to compile. If you also want to use a web browser like Chromium or Firefox, it takes up to 200 GB. And if you want to have a second instance, e.g. for a second hardware or a different configuration, you need twice the amount. So use a partition that has enough free space.

We have prepared a shell script called `setup-yocto` that does this unpacking automatically. Just call it when you are in the top directory of the release and give the name of the build directory as argument.

```
cd <release-dir>
./setup-yocto <build-dir>
```

Add option `--dry-run` if you want to check first what this command will do. Then only a list of actions will be output but no actual changes will take place. For further information simply call

```
./setup-yocto --help
```

By the way, the script rejects to run if it would overwrite existing files or directories and asks the user to remove these files first. If you are sure that it is OK to delete these existing files, you can use option `--force` to tell the script to continue nonetheless.

This is a sample output of the script when unpacking in a directory called `build`, directly within the release directory. So `build` will be right next to `binaries`, `sources`, `doc`, etc.

```
[fsimx6-Y1.0]$ ./setup-yocto build
```

11.2.3 Download Main Yocto

Note

This is not necessary for Yocto 4.0 releaes and later.

As already mentioned, the F&S Yocto system consists of different layers. But only the F&S layer is actually included in the F&S Yocto release tar archive. It provides our U-Boot, Linux kernel, machine descriptions and some Yocto modifications and enhancements to support F&S boards. Everything else, in other words the main part of Yocto, is downloaded from the internet.

We have added a script that downloads the main Yocto and then adds the F&S layer, that was unpacked in the previous step, on top of it.

```
cd <release-dir>/yocto-X.Y-fus-<v>
./yocto-download
```

This will take a few minutes, depending on your internet speed.

The download is done by using a small tool called `repo` that is designed to handle several GIT repositories in parallel. So the first step of the script is to download the `repo` tool. Then it fetches the repo description for the NXP/Freescale Release BSP and finally lets `repo` download all the different GIT repositories with the Yocto layers.

As a final step, the F&S layer is appended to `bblayers.conf` file.

11.2.4 Running Docker

Note

Only available starting Yocot 5.0

Starting F&S Yocto 5.0 releases can be build inside a docker container. This has the advantage, of a simpler and more consistent dependency management, a better reproducibility of the builds an an easier CI/CD pipline intigration.

First make sure that docker is installed on your building machine:

```
docker --version
```

If a docker version is printed run the following command to start the F&S Docker environment:

```
./setup-yocto <build-dir> --docker
```

If you run this command for the first time on you building machine, a new docker container will be created. This may take some time.

When the docker container is available, you will get a new shell from the docker container inside your <build-dir> directory.

Change to the `yocto-fus` directory to proceed with the next building steps.

```
cd ~/yocto-fus
```

11.2.5 Configure Yocto for an F&S architecture

F&S also uses architectures in Yocto, too. So you do not configure for a specific board, but you do configure for a whole F&S architecture <arch>. The resulting code will run on all boards that are part of this architecture. So what is called a machine in terms of Yocto, is actually our architecture.

Simply call

```
DISTRO=<distro> MACHINE=<arch> source ./fus-setup-release.sh [-b <build-dir>]
[-m <fs-mode>]
```

Please note that this script needs to be sourced, which means it must be run *in* the current shell to be able to modify the existing environment.

NXP uses the Yocto concept of distros to differentiate between the possible display solutions. F&S handles this in a similar way, but as we have our own layer we also use our own distro names. However they simply replace “fsl” with “fus”. See Table 26 for a list of possible values.

`<distro>`	Meaning
`Fus-imx-x11` `(Only Yocto 2.4)`	Use X11 graphics
`fus-imx-wayland`	Use Wayland graphics
`fus-imx-xwayland`	Use Wayland with X11 support (no EGL on X11 applications)
`Fus-imx-fb` `(Only Yocto 2.4)`	Use plain framebuffer graphics (useful for Qt, but not supported in all fsimx8 architectures)

Table 26: Possible distros in the F&S Yocto layer

`<build-dir>` is the name of the directory where the given distro will be built. It is possible to build different distros in the same Yocto environment. Then these distros will share the meta data and they all use the same download directory, which saves disk space. Therefore we recommend a meaningful directory name, for example one that starts with `build` and holds the name of the distro and the architecture, like `build-x11-<arch>`. If no directory name is given, the build direr will be named “ build-<arch>-<distro>” by default.

In Yocto the default filesystem mode is read-writeable. If you want a read-only filesystem, you can configure your build-environment by setting the <fs-mode> parameter to “ro”. If you want to to change the filesystem mode in an existing build-environment simply add or remove the line

EXTRA_IMAGE_FEATURES += "read-only-rootfs "

from your <build-dir>/conf/local.conf file and rebuild your image.

The above command will automatically switch to the newly created directory and sets the environment so that everything runs smoothly. However this is only valid in the current shell. You have to set up this environment in every newly started shell that should be used to build this Yocto image, for example if you restart your PC. This is done by going to the Yocto top directory and calling

```
source ./setup-environment <build-dir>
```

11.2.6 Build a Yocto Image

Yocto comes with a set of predefined demo images. Each image builds a more or less comprehensive set of libraries, tools and applications and then creates a filesystem image that can be downloaded to the board. There are images prepared by F&S, but also images from Freescale/NXP and of course also the basic images provided by Yocto (Poky). The images that we have tested and that are supposed to work on our boards are listed in Table 27.

Image	Meaning
`core-image-minimal`	Just command line interface and a few command line tools
`fus-image-std`	Standard F&S image with X11 (or Wayland) server, matchbox window manager (or Weston compositor), gstreamer, can-utils and a few small applications
`Fus-image-qt5/` `fus-image-qt6`	Qt5 based image Qt6 based image

Table 27: Some Yocto images

To build one of the demo images, simply call

```
bitbake <image-name>
```

where `<image-name>` is one of the images listed above.

First `bitbake` parses all existing recipes. This may take a few minutes on the first invocation. But as `bitbake` builds a cache for all the recipes, this step will only take a few seconds in subsequent calls, unless one of the recipes is changed.

Then `bitbake` determines what tasks have to be done to reach the desired target. A task may be fetching a package via the internet, unpack a package, configure, compile, install a package or similar steps. It then generates a list of all these tasks.

Finally `bitbake` executes this list of tasks. If you have a multi-core system, `bitbake` can work in parallel. It knows the dependencies between the tasks and if possible, it already executes the next task(s) in parallel, up to a certain limit that can be given in the configuration. But please keep in mind that each task itself may be a complicated build process that may already use parallel execution, for example a `make` with many parallel threads. So don't overdo the settings, because the number of parallel `bitbake` tasks will multiply with the number of parallel executions within the tasks themselves.

Building such an image will download all the required source packages, patch them, configure them, compile them, and assemble them in a package system and in target images. This includes binaries for the bootloader U-Boot, the Linux kernel, the device trees, and the root filesystem. Even the toolchain, that is used for compiling everything, is built during this process. Therefore building a Yocto image can take quite a long time, even several hours. For example the `fus-image-std` takes about three hours to build on a PC with a rather fast quad core CPU with hyper-threading.

The resulting files can be found in subdirectory `tmp/deploy/images`.

File	Meaning
`uboot.nb0`	U-Boot binary (suited for NBoot)
`(z)Image`	Linux kernel image
<boardname>`-*.dtb`	Device trees for all boards of architecture <arch>
`<image-name>-<arch>.ubifs`	Image to be installed in NAND flash
`<image-name>-<arch>.ext4`	EXT3 image, e.g. to be used via NFS
`<image-name>-<arch>.sysimg`	Image to be installed on an SD/eMMC card

Table 28: Resulting files

If execution fails at some task, `bitbake` finishes all the other tasks that are running in parallel, but it will not start any new tasks. When all pending tasks are completed, it will stop with an error message. `bitbake` logs all steps that it does in log files, one file per task. So you can reconstruct at which step the build process failed. Then you can fix the problem and when you call `bitbake` again, it will skip the tasks that it had already finished successfully and restart at the point where it had left off.

You can also interrupt the build process by hand, by typing *Ctrl-C*. However `bitbake` will *not* stop immediately. Instead like in the error case, it first finishes all running tasks and then

stops. To really stop immediately, you have to press *Ctrl-C* twice. In both cases you can continue by issuing the `bitbake` command again.

Usually you can also build additional images without having to erase the previous results. Just call `bitbake` with the new image name. Again only those steps that are new for the new image are done by `bitbake`.

Remark

Some images are rather large and may not fit into the NAND flash memory of your board or module. So for example `fsl-image-multimedia` results in a file of about 160 MB which does not fit into the 128 MB of the regular armStoneA9, so it can not be run from NAND flash. Such a rootfs can only be mounted via NFS or SD card then.

An even more complex example is `fsl-image-machine-test` This image creates an UBIFS image with a size of about 300 MB. But by default our UBIFS images are configured to be at most 256 MB of size. Therefore to compile this image, you have to modify the file `sources/meta-fus/conf/distro/include/fus-common.inc` to avoid an overflow of the UBIFS rootfs. Replace the line

```
MKUBIFS_ARGS = "-m 2048 -e 126976 -c 2048"
```

with this content:

```
MKUBIFS_ARGS = "-m 2048 -e 126976 -c 2800"
```

11.2.7 Rebuild A Single Package

Actually the target that is given to the `bitbake` command is just the name of a recipe. Image names are also just recipe names. But of course you can also execute smaller entities, for example a smaller sub-target or even a single package. Everything that has a separate recipe can be processed separately with `bitbake`.

Usually when a package is built, different tasks (or stages) are executed. For example `fetch`, `unpack`, `patch`, `configure`, `compile`, `strip`, `install`, `package`, `deploy`. The list may vary from package to package. For example the Linux kernel package has additional tasks to build the kernel modules and the device tree binaries.

You can show the list of possible task of a package with the command

```
bitbake -c listtasks <package>
```

For example:

```
bitbake -c listtasks linux-fus
```

To tell bitbake just to execute one task of a recipe, you give the name of the task after option `-c`. For example to just do the compile task of a package, just call

```
bitbake -c compile <package>
```

A special task that is part of nearly every recipe is the task named `clean`. It is used to remove the package that is built by this recipe from the system again. Before being able to recompile an already existing package, it must be removed. Therefore to rebuild a single pack-

age, you usually execute the following sequence of commands. They clean and recompile the package and then add it again to the final target image.

```
bitbake -c clean <package>
bitbake -c compile <package>
bitbake <image-name>
```

Here are some other examples, using the `linux-fus` package. To call `menuconfig` for the kernel, use

```
bitbake -c menuconfig linux-fus
```

The kernel is an own image. To copy the kernel to the `deploy/images` directory:

```
bitbake -c deploy linux-fus
```

Open a shell in the package directory where you can issue arbitrary commands:

```
bitbake -c devshell linux-fus
```

11.2.8 Add Packages to an Image

```
If you want to add a package to your local build (in addition to the regular
image content), you have to go to your distro directory and open the file ./
conf/local.conf. Here you can add the package by listing it in
IMAGE_INSTALL_append. Sometimes you also have to do additional steps like en-
abling a specific software license in variable LICENSE_FLAGS_WHITELIST. If the
software is in a separate layer, check if the layer is already in conf/bblayer-
s.conf and add the layer if it is missing.
```

For example to add the Chromium browser on top of `fus-image-std`, the following two lines need to be present in the *BBLAYERS* variable in `bblayers.conf`. (This should be the case if you set up the build directory with `fsl-setup-release.sh` like shown above.)

For example to add the Chromium browser on top of `fus-image-std` you have to add following line to `local.conf`

```
# add chromium to rootfs
CORE_IMAGE_EXTRA_INSTALL += “chromium-ozone-wayland”
```

Please note that building this image with Chromium needs lots of memory and time.

11.2.9 Useful utilities

If you run

```
source ../yocto-f+s-utilities
```

in the top directory of Yocto, you will get the following small helper functions.

Function	Meaning

`list_fus_target_boards`	Show a list of all supported F&S boards/architectures
`list_target_images`	Show all recipes with "image" in their name
`list_all_recipes`	List all available recipes (list is long!)
`list_target_image_packages`	List all the packages that are part of the target image given as argument

Table 29: F&S helper functions for Yocto

For example to determine which packages are part of fus-image-std, call

```
list_target_image_packages fus-image-std
```

11.2.10 Further reading

There is quite a lot of documentation available at the Yocto homepage

- https://www.yoctoproject.org/documentation
- https://www.yoctoproject.org/documentation/active

In addition you can also have a look at the documentation of the NXP/Freescale Community BSP on github:

- https://github.com/Freescale/Documentation

There is also a fine tool called `bb` that allows inspecting Yocto variables, list recipes and providers, show `bitbake` logs, search packages and recipes, show package contents and dependency structures, edit recipes, `include` and `bbappend` files, and similar things. You can clone it from github:

```
git clone https://github.com/kergoth/bb
```

Then read `README.md` for installation instructions.

12 Appendix

List of Figures

Figure 1: Single Board Computers....1
Figure 2: Systems on Module....1
Figure 3: Some Linux devices....4
Figure 4: Linux diversity....5
Figure 5: Mainline and manufacturer versions diverge....8
Figure 6: Mainline versions, manufacturer versions and F&S support....9
Figure 7: Hardware accesses must go through the Linux kernel....9
Figure 8: F&S development environment....13
Figure 9: Serial port options for PCs without native port....15
Figure 10: Components of a Linux system....16
Figure 11: Download documents from F&S website....19
Figure 12: Register with F&S website....19
Figure 13: Unlock software with the serial number....20
Figure 14: Boot sequence....25
Figure 15: NAND flash architecture....26
Figure 16: Bit states in SLC, MLC, TLC, QLC NAND flash cells....27
Figure 17: Common MTD partition layout in NAND flash....28
Figure 18: Common F&S layout with NAND & eMMC....29
Figure 19: Default F&S layout for boards with eMMC only....30
Figure 20: NBoot boot stage....33
Figure 21: NAND flash erase command....36
Figure 22: NetDCUUSBLoader....38
Figure 23: ARM64 (ARMv8) boot process and Execution Levels....41
Figure 24: Detailed MTD partiton layout on F&S boards....47
Figure 25: NAND with MTD partitions only....48
Figure 26: NAND layout with UBI volumes....49
Figure 27: Bar-code sticker....57

Figure 28: Network access via RNDIS....59
Figure 29: Block Refresh with intermediate safety copy....78
Figure 30: Generic Update/Install/Recover flow....79
Figure 31: Regular Boot Process....80
Figure 32: Install Process....81
Figure 33: Update process....82
Figure 34: Recover by copying backup version to active version....83
Figure 35: Recover by re-activating the previous version....83
Figure 36: Recover process....84
Figure 37: Linux kernel and device tree as separate MTD partitions....98
Figure 38: Linux kernel and device tree in separate UBI volumes....99
Figure 39: Linux kernel and device tree are simlpe files in the root filesystem....99
Figure 40: Download files with TFTP, store in NAND flash....111
Figure 41: Typical eMMC layout....112
Figure 42: Low-level copy external SD card to internal eMMC....114
Figure 43: Copy sysimg to internal eMMC....115
Figure 44: Installing individual files on eMMC....116
Figure 45: Platform devices on an imaginary hardware....118
Figure 46: Old way: Device drivers and board support are both in the kernel image....120
Figure 47: Old way: Board support for several platforms with Machine IDs....121
Figure 48: New way: Device tree is separate from the kernel image....122
Figure 49: New way: One device tree per platform....123
Figure 50: Qt Example (Gradients)....151
Figure 51: Layer architecture of Yocto....160

List of Tables

Table 1: F&S Architectures....2
Table 2: Content of the created release directory....24
Table 3: Hardware partitions on eMMC memories....30
Table 4: Incomprehensive list of U-Boot commands....45
Table 5: NAND flash partitioning....47
Table 6: Different UBI volume layout....51
Table 7: Important U-Boot settings....54

Table 8: Necessary environment variables for RNDIS.....60
Table 9: Wildcards in FAT names.....77
Table 10: Variables to configure Install/Update/Recover Mechanism.....85
Table 11: Syntax for install/update/recover script locations.....87
Table 12: Variables to set kernel source.....92
Table 13: Variables to set device tree source.....93
Table 14: Variables to set rootfs source.....93
Table 15: Variables to set set /deny write access for root filesystem.....93
Table 16: Variables to set console output.....93
Table 17: Variables to set login prompt origin.....94
Table 18: Variables to set network activation.....94
Table 19: Variables to set init process file.....95
Table 20: Variables to define the MTD partition table.....97
Table 21: Variables for UBI volume creation.....97
Table 22: Hardware partitions on eMMC and Enhanced Mode.....100
Table 23: fsimage sub-commands.....104
Table 24: Software partitions on eMMC.....113
Table 25: VPU based GStreamer plugins.....130
Table 26: Possible distros in the F&S Yocto layer.....163
Table 27: Some Yocto images.....164
Table 28: Resulting files.....165
Table 29: F&S helper functions for Yocto.....167

Listings

Listing 1: NBoot menu.....35
Listing 2: Hardware accelaration configuration in xorg.conf.....149

Important Notice

The information in this publication has been carefully checked and is believed to be entirely accurate at the time of publication. F&S Elektronik Systeme assumes no responsibility, however, for possible errors or omissions, or for any consequences resulting from the use of the information contained in this documentation.

F&S Elektronik Systeme reserves the right to make changes in its products or product specifications or product documentation with the intent to improve function or design at any time and without notice and is not required to update this documentation to reflect such changes.

F&S Elektronik Systeme makes no warranty or guarantee regarding the suitability of its products for any particular purpose, nor does F&S Elektronik Systeme assume any liability arising out of the documentation or use of any product and specifically disclaims any and all liability, including without limitation any consequential or incidental damages.

Products are not designed, intended, or authorised for use as components in systems intended for applications intended to support or sustain life, or for any other application in which the failure of the product from F&S Elektronik Systeme could create a situation where personal injury or death may occur. Should the Buyer purchase or use a F&S Elektronik Systeme product for any such unintended or unauthorised application, the Buyer shall indemnify and hold F&S Elektronik Systeme and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, expenses, and reasonable attorney fees arising out of, either directly or indirectly, any claim of personal injury or death that may be associated with such unintended or unauthorised use, even if such claim alleges that F&S Elektronik Systeme was negligent regarding the design or manufacture of said product.