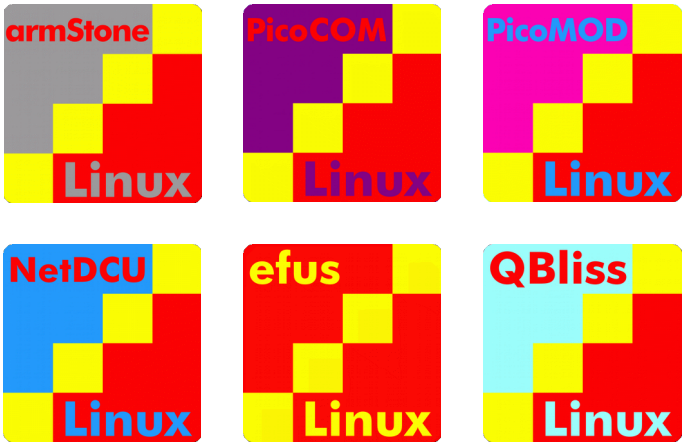
F&S Vybrid Linux

First Steps

Version 1.4
(2016-10-21)



Untere Waldplätze 23
D-70569 Stuttgart
Germany

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

About This Document

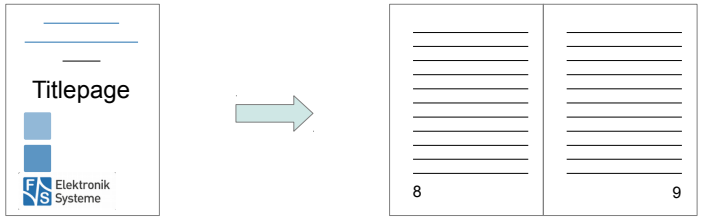
This document shows how to bring up F&S boards and modules under Linux, how to update firmware and how to use the system and the devices. It covers also compiling bootloader, Linux kernel and root filesystem as well as how to build your own applications for the device.

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
2013-05-08	1.0	armStoneA5 and PicoCOMA5	A	*	First Version. V0.1	DK
2013-09-20	1.1	armStoneA5, Pico COMA5, NetDCUA5				HK
2014-08-22	1.2	fsvybrid	A, M	all	Overall updates: improved introduction, explain architecture releases, new F&S Download Area, show automatic installation with an SD card, improve manual installation procedure, introduce the new system for switching boot strategies, short introduction of the update/install/recover mechanism of F&S, add CAN usage	HK
2014-09-11	1.3		M	all	Fix footer line	HK
2016-09-14	1.3	fsvybrid	A,M	6.15, 7.1, 10	Minor changes, to be more specific, SSH, add installing mkimage tool Compiling Buildroot,	PH
2016-10-17	1.4	fsvybrid	A,M	all	Add more information and additional explanation, new design	PH

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

Table of Contents

1 Introduction

F&S offers a whole variety of Systems on Module (SOM) and Single Board Computers (SBC). There are different board families that are named NetDCU, PicoMOD, PicoCOM, armStone, QBliss, and efus.

Family	Type	Size
NetDCU	Single Board Computer	80 mm x 100 mm
PicoMOD	System on Module	80 mm x 50 mm
PicoCOM	System on Module	40 mm x 50 mm
armStone	Single Board Computer	100 mm x 72 mm (PicoITX)
QBliss	System on Module	70 mm x 70 mm (Qseven)
efus	System on Module	62 mm x 47 mm

Table 1: F&S Board Families

Linux is available for all of these platforms. In the past, there was a separate Linux release for each platform. This was quite cumbersome and also caused a lot of duplicated work.

At the moment, F&S combines releases for platforms with the same CPU – or rather SoC (System on Chip), as these microcontroller units are far more than just a CPU – as so-called 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 CPU type, i.e. they usually contain boards of different families.

At the moment the following F&S architectures are supported:

Architecture	CPU	Platforms
fss5pv210	Samsung S5PV210	PicoMOD7A, NetDCU14, armStoneA8
picocom4	Samsung S3C2416	PicoCOM4
fsvybrid	NXP Vybrid VF6xx	PicoCOMA5, NetDCUA5, armStoneA5, PicoMODA5, PicoMOD1.2
fsimx6	NXP i.MX6	efusA9, QBlissA9, QBlissA9r2, armStoneA9, armStoneA9r2, PicoMODA9
fsimx6sx	NXP i.MX6-SoloX	efusA9X, PicoCOMA9X
fsimx6ul	NXP i.MX6-UltraLite	efusA7UL

Table 2: F&S Architectures

The final goal is to combine all architectures in one global *Multi-Platform* Linux release, where the different platforms and architectures can be built from one and the same source package.

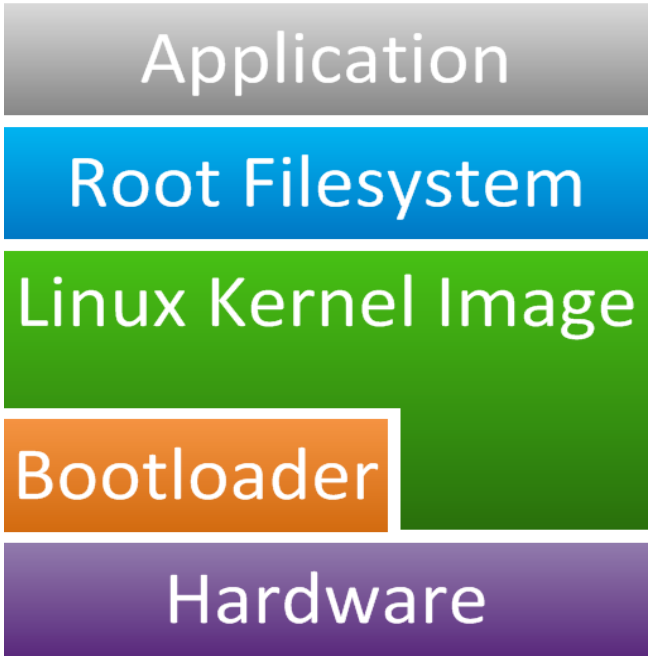


Figure 1: Components of a Linux system

If you look at Figure 1, 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.

So the Linux infrastructure is far more than just the kernel. The Linux distribution 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. See chapter 3.1 for more information about Nboot. Uboot activates the hardware, loads the Linux kernel and executes it. Uboot download and install the images for Linux kernel and root filesystem. It can also boot the board from different devices, for example from a server across the network.

Linux Kernel

This is a Linux kernel modified to support our 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, DirectFB with a few examples, 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 Multi-Platform already, but the Linux kernel is still different on each architecture. This is mainly because these architectures require special hardware drivers that are only available in manufacturer specific kernel releases and are not yet ported to the mainline kernel.

F&S works hard to get those different versions more closely together and even now the development environment is already very similar for all platforms, even across the architectures. 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.

This document describes the *fsvybrid* architecture. That means all F&S boards and modules based on the NXP Vybrid SoC, i.e. the PicoCOMA5, armStoneA5, NetDCUA5 and a couple

of customer specific boards. We will show you how you get started with such a platform and how you compile your first programs.

2 Development Environment

To seriously work and develop with F&S boards and modules, you need a Linux based PC for the software development, a terminal program to enter commands on the command line, a TFTP server to download files to the board and an NFS server to provide files and directories over the network easily.

We at F&S use:

Linux PCs based on Fedora Linux as development machines. They are running as virtual machines in VirtualBox (Oracle) under a Windows host.

- Putty or TeraTerm as terminal program connected to the serial console of the board to enter commands for U-Boot and Linux.
- TFTP server and NFS server provided by the Linux distribution.

In addition we sometimes use the following Windows software

- `DCUTerm` (terminal program for Windows made by F&S) to download U-Boot by serial line.
- `TeraTerm` as an alternative to `putty`.
- `NetDCUUsbLoader` (by F&S) to download files via USB in NBoot.
- `TFTPD` (by Philippe Jounin) as TFTP Server.

Note

You will find the document `AdvicesForLinuxOnPC.pdf` in the doc subdirectory in the release archive that explains how to set up a linux based development machine. Or you can download it from the our web site at http://fs-net.de

2.1 Starterkit

2.1.1 armStoneA5

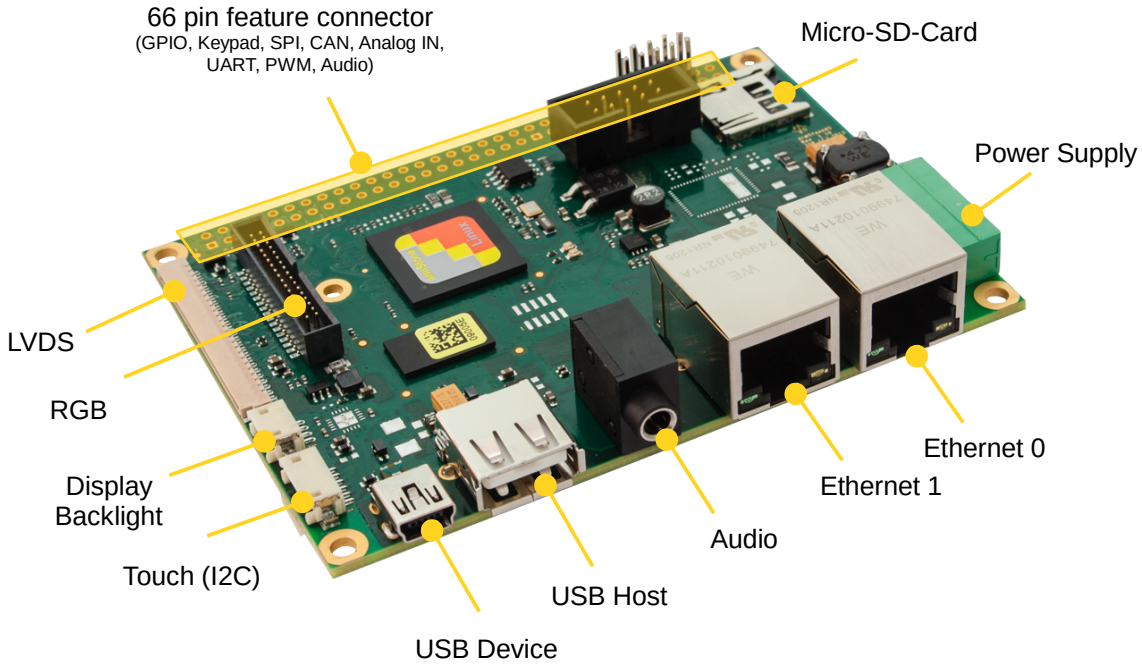


Figure 2: armStoneA5

2.1.2 PicoCOMA5

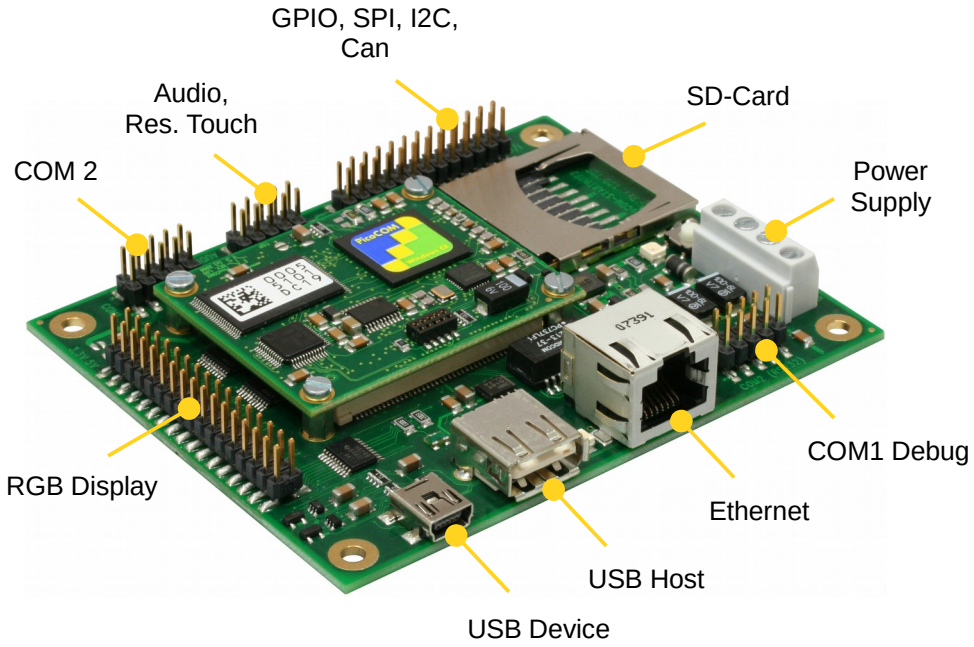


Figure 3: PicoCOMA5 with SKIT

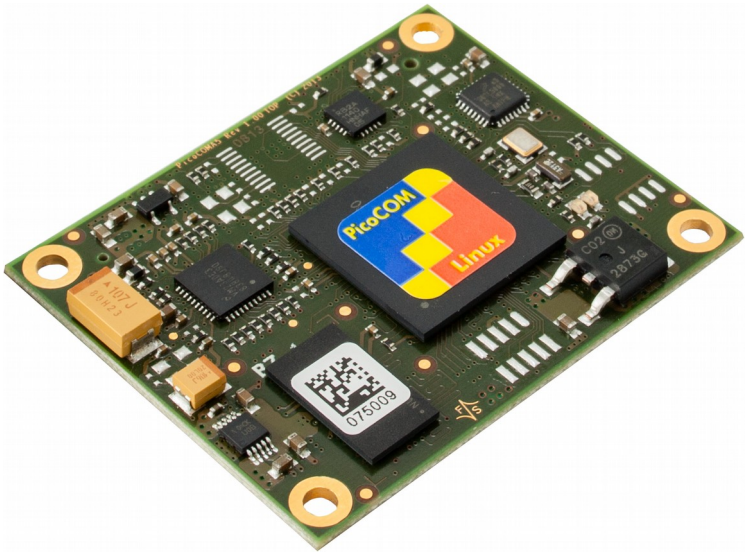


Figure 4: PicoCOMA5 module

2.1.3 NetDCUA5

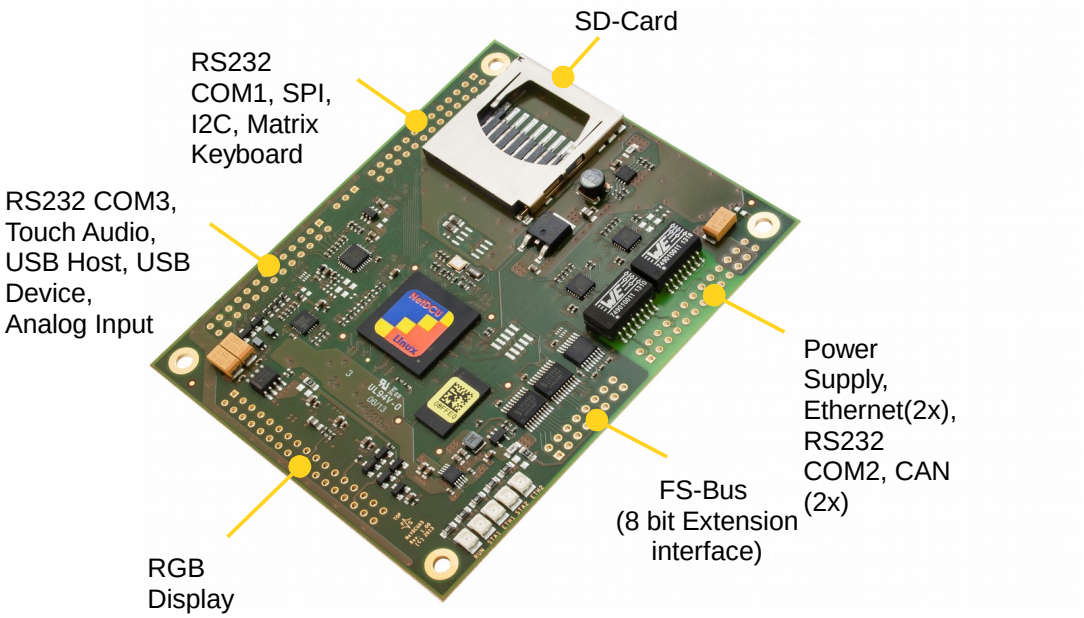


Figure 5: NetDCUA5

2.2 F&S Download Area

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

http://www.fs-net.de

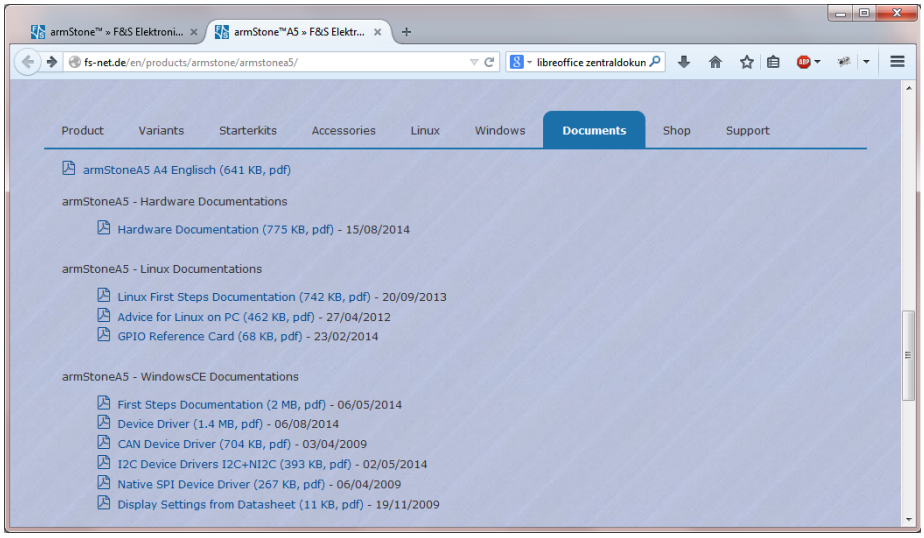


Figure 6: Download documents from F&S website

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 where you can select *Documents* (see Figure 6).To download any software, you first have to register with the website. Click on *Login* right at the top of the window and on the text “I am not registered, yet. Register now” (Figure 7).

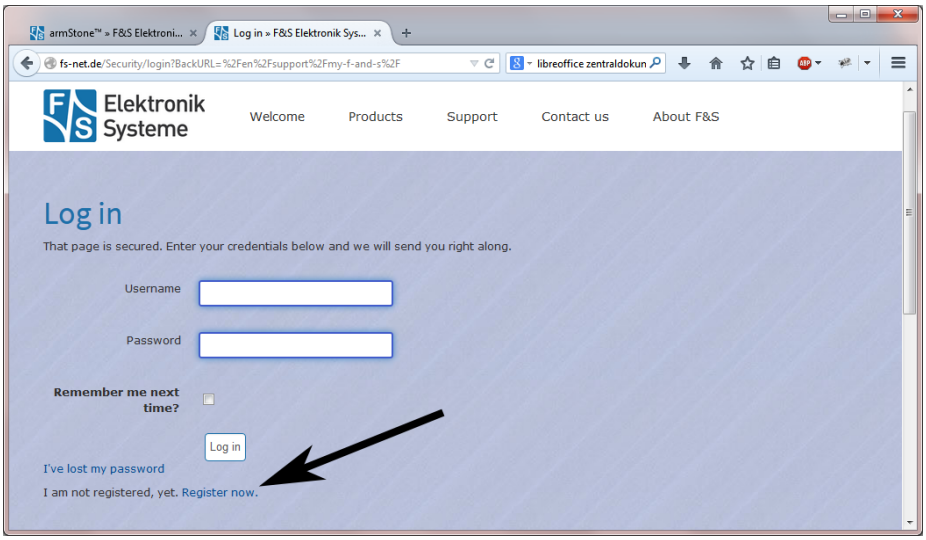


Figure 7: Register with F&S website

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.

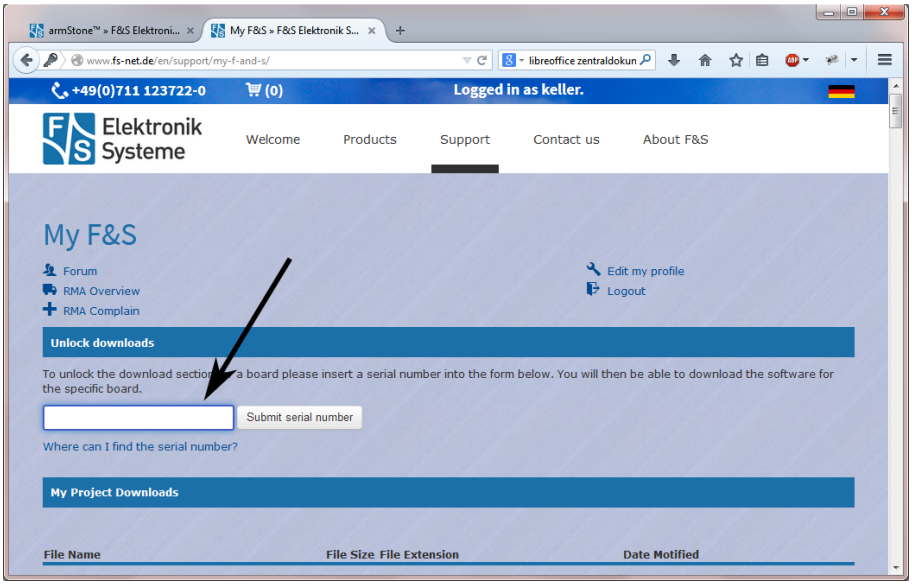


Figure 8: Unlock software with the serial number

After logging 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.

To get access to the software of a specific board, you have to enter the serial number of one of these boards (see Figure 8). 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.

Here you will always find our newest software releases. When you look at our Linux releases, you will find a file list of all our releases and a README text. There are usually two files related to a release.

`fsvybrid-V<x>.<y>.tar.bz2`
: This is the main release itself containing all sources, the binary images, the documentation and the toolchain.

`sdcard-fsvybrid-V<x>.<y>.tar.bz2`
: Files that can be stored on an SD Card 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. 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.

2.3 Release Content

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

```
tar xvf fsvybrid-V<x>.<y>.tar.bz2
```

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

`<package>-<platform>-V<x>.<y>.<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.

`<platform>`.....The name of a board, if the package is only valid on one board (e.g. `armStoneA5`); or the name of an architecture, if the package is valid on different boards of the same architecture (e.g. `fsvybrid`), or the string `f+s` if the package is architecture independent.

`V<x>.<y>`.....The major and minor number of the release (e.g. V2.0)

`<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 `uImage` file of the Linux kernel.

The following table lists the files that you get after unpacking the release archive.

Directory/File	Description
`binaries/`	**Images to be used with the platform directly**
`nbootvyb115_<v>.bin`	Architecture specific NBoot, 115200baud
`uboot-fsvybrid-V<x>.<y>.nb0`	U-Boot (bootloader) image
`uImage-fsvybrid-V<x>.<y>`	Kernel image (to be used with U-Boot)
`rootfs_std-fsvybrid-V<x>.<y>.ubifs`	Standard root filesystem (UBIFS format) to be stored in NAND flash memory
`rootfs_std-fsvybrid-V<x>.<y>.ext2`	Standard root filesystem (ext2 format), suited to be used via NFS
`rootfs_min-fsvybrid-V<x>.<y>.ubifs`	Minimal root filesystem (UBIFS format) to be stored in NAND flash memory

`rootfs_min-fsvybrid-V<x>.<y>.ubifs`	Minimal root filesystem (ext2 format), suited to be used via NFS
`install-fsvybrid-V<x>.<y>.scr`	Install script (U-Boot autoscript image)
`sources/`	**Configurations and sources**
`u-boot-<packageversion>-fsvybrid-V<x>.<y>.tar.bz2`	U-Boot source with modifications
`linux-<packageversion>-fsvybrid-V<x>.<y>.tar.bz2`	Linux kernel source with modifications
`buildroot-<packageversion>-fsvybrid-V<x>.<y>.tar.bz2`	Buildroot package with modifications
`examples-fus-V<x>.<y>.tar.bz2`	Examples package
`install.txt`	Install script (text version)
`toolchain/`	**Cross-Compilations toolchain**
`Fs-toolchain-5.2.0-armv7ahf.tar.bz2`	F&S toolchain to use with fsvybrid
`mkimage`	Program needed for autoscript images Also needed to compile kernel as uImage
`doc/`	**Documentation**
`Vybrid_FirstSteps_eng.pdf`	First Steps document
`AdvicesForLinuxOnPC_eng.pdf`	Document describing how to install services and tools on a Linux PC to be used with F&S Linux boards.
`doc/Hardware/`	**Hardware Documentation**
`PicoCOMA5_Hardware_eng.pdf`	Hardware description for PicoCOMA5
`PicoCOM_Startinterface_eng.pdf`	Hardware description for starterkit carrier board for PicoCOMA5
`PCOMnetA5_Hardware_eng.pdf`	Schematics for PCOMnetA5 carrier board with two ethernet ports
`NetDCUA5_Hardware_eng.pdf`	Hardware description for NetDCUA5
`NetDCU_SINTF14_eng.pdf`	Hardware description for starterkit carrier board for NetDCUA5

armStoneA5_Hardware_eng.pdf	Hardware description for armStoneA5
doc/GPIO-Reference-Cards/	**GPIO Documentation**
PicoCOMA5-GPIO-ReferenceCard_eng.pdf	Pin to GPIO mapping for PicoCOMA5
NetDCUA5-GPIO-ReferenceCard_eng.pdf	Pin to GPIO mapping for NetDCUA5
armStoneA5-GPIO-ReferenceCard_eng.pdf	Pin to GPIO mapping for armStoneA5
doc/Schematics/	**Hardware Schematics**
NetDCU-SINT14_Schematic.pdf	Schematics for starterkit carrier board for NetDCUA5
PicoCOM2_3_4-Startinterface_V1.00_Schem+BOM.pdf	Schematics for starterkit carrier board for PicoCOMA5
sdcard/	**Files to copy to SD card (symbolic links to binaries directory)**
nbotvyb.bin	NBoot loader, name as expected by NBoot
ubotvyb.nb0	U-Boot, name as expected by NBoot
uImage-fsvybrid	Linux kernel, name as expected by install.scr
rootfs-fsvybrid.ubifs	Rootfs, name as expected by install.scr
install.scr	Install script to install kernel and rootfs
/	
Readme.txt	Release notes
install-sources.sh	Script to unpack all source packages into a build directory

Table 3: 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 fully 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 SD card archive, that just contains this directory.

3 Bringing Up the System

When you get a Starterkit from F&S, the Linux system is usually pre-installed. In this case you can skip this chapter. But if you are switching over from an other operating system, if you are upgrading from a previous release, or if your board is empty for some other reason, the following sections describe how to install the provided standard images of the release on your platform.

There is a very simple automatic installation possible if you have an SD card or usb stick available. Or you can use the manual installation procedure via serial line and TFTP. We will describe one after the other.

Both installation procedures use the serial connection to give commands to NBoot and U-Boot. Connect the serial debug port to your PC. Please refer to chapter 2.1 for the location of the COM ports. Use 115200 baud, 1 start, 1 stop bit, no flow control. To transfer the images to the board use a terminal program that can download binary images 1:1 without using any serial protocol like kermit, xmodem or similar. A suitable terminal program for example is `DCUTerm.exe` from F&S or `TeraTerm`. You can find `DCUTerm.exe` in the Tools-Section of the Download Area (in *MyF&S*).

Later, when working regularly with the board, you can use a different terminal program like `Putty`, `TeraTerm` or similar. In fact you actually *should* use a different program then, because `DCUTerm.exe` does not support escape sequences for switching text color and then the output of commands like `ls` is nearly unreadable. Also accessing the command history is not possible with the up and down keys in `DCUTerm.exe`.

3.1 Entering NBoot

NBoot is a small first-level bootloader that is running before the main bootloader. It is the same for Linux and Windows CE and always remains on the board even if the whole flash memory is erased. By default NBoot is rather invisible and just loads and starts the main bootloader when the board is switched on. But it can also be used to download and store a new bootloader.

As long as NBoot is available on the board, it is always possible to bring up the whole system again. 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's what we want to do now.

Open the serial connection. 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 into the first-level bootloader NBoot. You should see something like this (output is taken from armStoneA9, the real messages may vary slightly depending on the software version):

```
F&S Nand Loader VN13 built Aug 12 2015 11:05:37
armStoneA5 Rev. 1.10

...
```

```
Please select action
'd' -> Serial download of bootloader
'E' -> Erase flash
'B' -> Show bad blocks

Use NetDCUUsbLoader for USB download
```

Listing 1: NBoot menu

3.2 Automatic Installation Using an SD Card

As you have seen in the previous chapter, F&S provides an SD card archive as separate tar file and also as part of the regular release archive. When using these files, installation of a Linux system can be done nearly automatically by only pressing a few buttons in NBoot.

First prepare the SD card. You need an SD card formatted with the FAT filesystem. Then copy all files from `sdcard-fsvybrid-V<x>.<y>.tar.bz2` or from the `sdcard` subdirectory of the regular release archive to the card. Do not create any subdirectories, the files have to reside directly in the top directory of the card.

Then simply follow these steps:

1. Enter NBoot (see chapter 3.1 on page 14).
2. Erase the flash by pressing *E* (upper-case e). This removes everything that was on the board before, for example an old installation. Don't be afraid, this won't erase NBoot itself.
3. Now press *c* (lower-case C). This will load the U-Boot image from SD card to RAM.
4. Press *f* (lower-case F). This will save U-Boot to NAND flash.
5. Press *x* (lower-case X). This will execute U-Boot.
6. Now the installation procedure of U-Boot will kick in and installs all the remaining Linux images automatically. This will take about a minute.

As you see, it only takes four keys: *E* → *c* → *f* → *x*.

Remark

At the time of writing, the NBoot of the fsimx6 architecture unfortunately was still missing SD card support and could therefore not load files from SD card directly. If this is the case you can not perform step 3 as described above. You have to download the file through serial download. See chapter 3.3.3 on page 17 for a description of how to do this.

Actually, if you are doing it this way, then you can also use a USB stick instead of an SD card.

3.3 Manual Installation

If you do not have an SD card available or if you want to understand how the installation procedure works in detail, you can also install everything manually.

There are several ways to download the files to the board, for example via a USB connection, with serial download, loading the files from an SD card or a USB stick, or by Ethernet transfer with the TFTP or NFS protocol. Unfortunately not all variants are available in all installation steps and presenting all variants for each step would also be very confusing here in a first steps document. Therefore we will only present a single variant where we use serial download in NBoot and then TFTP transfers in U-Boot. This is how it is usually done during normal development, so you will need a similar sequence later anyway.

This assumes that you have set up the TFTP service on your PC correctly and that you have copied the appropriate files to the TFTP directory there. Doing this is beyond the scope of this manual. We have an additional document `AdvicesForLinuxOnPC_eng.pdf` that gives some advice on how to do this. In addition you will need a Terminal as described in chapter 3. Please refer to chapter 2.1 on page 6 for the location of the serial ports.

Installing the Linux system manually involves five steps

1. Update NBoot (if necessary)
2. Erase flash

Install the Linux bootloader U-Boot

Install the Linux kernel image

Install the root file system

3.3.1 Check NBoot Version and Update If Necessary

Enter NBoot (see chapter 3.1 on page 14). NBoot will show a greeting text and a short menu. Please verify the version of NBoot. By default the Board is shipped with the latest version of Nboot installed. If the version on the board is older than the version that is contained in the downloadable tar archive it is recommended to install this newer version. In general upgrading NBoot might not be necessery depending on Board, placing, Nboot version and more but as we are constantly improving stability it is recommended.

To update NBoot, press *D* (upper-case d). This will show some message similar to this:

```
Waiting for NBoot...
```

Now in `DCUTerm.exe`, select from the file menu *Transmit Binary File...* and select `nbootvyb115-<v>.bin`. This starts the serial download. When download is complete, save NBoot by pressing *f* (lower-case *F*). This should show something like this:

```
Saving Nboot...Success
```

Warning

It is important to download the file in binary mode, where bytes are sent 1:1 to the board. Do *not* transfer in text mode, because there some characters like linefeeds are replaced and will lead to a non-working file, and thus probably to a non-responding board at the next boot! So be careful here.

Now restart the board and stop again in Nboot.

3.3.2 Erase Flash

If you have changed NBoot or if you are switching to very different versions, it is a good idea to erase the whole flash now by pressing *E* (upper-case e). This removes everything that was on the board before. Don't be afraid, this won't erase NBoot itself.

3.3.3 Load and Save U-Boot

While still in NBoot, press *d* (lower-case D). This will show some message similar to this:

```
Waiting for bootloader...
```

Now in `DCUTerm.exe`, select from the file menu *Transmit Binary File...* and select `uboot-fsvybrid-V<x>.<y>.nb0`. This starts the serial download. When complete, save U-Boot by pressing *f* (lower case F). This should show

```
Saving U-Boot...Success
```

Note

Again it is important to download the file in binary mode, where bytes are sent 1:1 to the board. Do *not* transfer in text mode, because there some characters like linefeeds are replaced and will lead to a non-working U-Boot. Then you have to repeat the procedure.

3.3.4 Load and Save Linux Kernel Image

As the U-Boot image is still loaded in RAM from the previous step, you can directly start U-Boot by pressing *x* (lower case X). This will show something like this:

```
U-Boot 2014.07-g2a84ed8 (Oct 17 2016 - 14:16:42) for F&S

CPU:   Freescale Vybrid 600 family rev1.1 at 396 MHz
Reset: Power-on Reset
Board: armStoneA5 Rev 1.10 (400 MHz, 1x DRAM, 2x LAN, 2x CAN)
DRAM:  256 MiB
WARNING: Caches not enabled
NAND:  128 MiB
MMC:   FSL_SDHC: 0
In:    serial
Out:   serial
```

```
Err:   serial
Net:   FEC0 [PRIME], FEC1
Hit any key to stop autoboot:  0

armStoneA5 #
```

To download the Linux kernel image by TFTP you have to set the networking parameters. Table 4 shows the possible settings.

Environment Variable	Meaning
`ipaddr`	The IP address of your board
`serverip`	The IP address of your TFTP and/or NFS server (usually your PC)
`gatewayip`	The IP address of your gateway; this is the device in your network that knows how to access the internet (usually your router)
`netmask`	The network mask 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)

Table 4: Network parameters in U-Boot

You have to adjust at least values for `ipaddr`, `serverip` and probably `netmask`. Of course you have to use values that fit your local network environment. The following examples are for a network based on IP-Adress 10.x.x.x. You probably have 192.168.1.x or some other IP address. If you want to store this variable settings permanently call *saveenv* afterwards.

```
armStoneA5 # setenv ipaddr 10.0.0.27
armStoneA5 # setenv serverip 10.0.0.126
```

For example you can use `ping` to verify that your network access is working.

```
armStoneA5 # ping 10.0.0.126
Using FEC0 device
host 10.0.0.126 is alive
```

After that you can download `uImage-fsvybrid-V<x>.<y>` to RAM, erase the old content of the Kernel partition and write the kernel image to the `Kernel` partition. Replace `<x>` and `<y>` with the version of your release, in our example it is `V2.2`. The two values `$loadaddr` and `$filesize` are automatically set by U-Boot. `$loadaddr` is the default address in RAM where images can be loaded to, and `$filesize` gets set by all download commands to the size of the downloaded file. This is quite convenient, especially when writing generic scripts where the size of a file is not known beforehand.

```
armStoneA5 # tftp uImage-fsvybrid-V2.2
Using FEC0 device
TFTP from server 10.0.0.126; our IP address is 10.0.0.27
Filename 'uImage_fsvybrid-V2.2'.
Load address: 0x10800000

Loading: ##############...##
```

```
done

Bytes transferred = 4497696 (0x44a120)

armStoneA5 # nand erase.part Kernel
NAND erase.part: device 0 offset 0x240000, size 0x5c0000
Erasing at 0x7e0000 -- 100% complete.
OK

armStoneA5 # nand write $loadaddr Kernel $filesize
NAND write: device 0 offset 0x240000, size 0x44a120
 4497696 bytes written: OK
```

3.3.5 Load and Save Root File System

Downloading and flashing the root filesystem image works similar to the Linux kernel image and the device tree. But it will be stored in a UBI (Unsorted Block Image), not directly in an MTD partition. Therefore we have to use the UBI tools.

So first create a UBI on the `TargetFS` partition with volume `rootfs` in it. A UBI volume is like a partition inside of a UBI, so you can split the UBI into smaller parts if you want to. The standard configuration simply uses the whole UBI as one big volume named `rootfs`.

```
armStoneA5 # ubi part TargetFS
Creating 1 MTD partitions on "nand0":
0x000000900000-0x000008000000 : "mtd=5"
UBI: attaching mtd1 to ubi0
UBI: physical eraseblock size:   131072 bytes (128 KiB)
UBI: logical eraseblock size:    126976 bytes
UBI: smallest flash I/O unit:    2048
UBI: VID header offset:          2048 (aligned 2048)
UBI: data offset:                4096
UBI: empty MTD device detected
UBI: create volume table (copy #1)
UBI: create volume table (copy #2)
UBI: attached mtd1 to ubi0
UBI: MTD device name:            "mtd=5"
UBI: MTD device size:            119 MiB
UBI: number of good PEBs:        952
UBI: number of bad PEBs:         0
UBI: max. allowed volumes:       128
UBI: wear-leveling threshold:    4096
UBI: number of internal volumes: 1
UBI: number of user volumes:     0
UBI: available PEBs:             939
UBI: total number of reserved PEBs: 13
UBI: number of PEBs reserved for bad PEB handling: 9
UBI: max/mean erase counter: 0/0

armStoneA5 # ubi create rootfs
Creating dynamic volume rootfs of size 119230464
```

Now download the root filesystem image `rootfs_std-fsvybrid-V<x>.<y>.ubifs` and store it in the `rootfs` volume. Again replace `<x>` and `<y>` with your release number. In our example it is `V2.2`

```
armStoneA5 # tftp rootfs_std-fsvybrid-V2.2.ubifs
Using FEC0 device
TFTP from server 10.0.0.117; our IP address is 10.0.0.27
Filename 'rootfs_std-fsvybrid_V2.2.ubifs'.
Load address: 0x10800000

Loading: #######...##

done
Bytes transferred = 32514048 (1f02000 hex)

armStoneA5 # ubi write $loadaddr rootfs $filesize

Volume "rootfs" found at volume id 0.
```

By the way if you get some messages about bad blocks, don't worry. NAND flash memory often has some bad bits. This is taken care of by the software and these blocks are automatically skipped. A few bad blocks are completely normal and no reason for reclamation.

3.4 Set Mac Adress

When the U-Boot environment is erased, also the MAC address for the ethernet chip is lost. We definitely have erased this environment when we used NBoot command *E* in our installation procedure, either in chapter 3.2 or in chapter 3.3.2. In such a case we have to set it again now and save it permanently.

The MAC address is a unique identifier for a network device. Each network device has its own address that should be unique across the whole world. So each network port on each board needs a unique MAC address.

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 bar-code sticker directly on your board (see Figure 9).

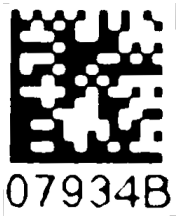


Figure 9: Bar-code sticker

The following two commands will set the MAC address and stores the current environment (including the newly set MAC address) 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.

```
armStoneA5 # setenv ethaddr 00:05:51:xx:yy:zz
armStoneA5 # saveenv
```

Warning

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

Remark

Some customer specific boards may have the MAC-Address pre-programmed in OTP Memory of the i.MX CPU. Here the `ethaddr` variable will be set automatically. But you can still override this by setting `ethaddr` manually.

3.5 Manually Updating

Of course you can also update only a single system file. For example when you have modified the root filesystem by a new build of your application software, only the root filesystem image needs to be updated, all other files can remain the same. You can use the same procedures like in the chapters above for doing that. Here are some hints about that.

When you are updating an MTD partition, do not forget to erase it first. Otherwise the new content will be mixed with the old content and the result will not work.

You can update a UBI volume without erasing it first. The UBI tools will automatically erase any old content first. So just tell UBI with the `ubi part` command which MTD partition to use, then download the new file and use the `ubi write` command.

You can also update an U-Boot image directly in U-Boot. You need not do this necessarily from NBoot. Only NBoot must be updated from NBoot.

You can neither read nor write the NBoot partition from U-Boot or from Linux. This is meant as a security feature so that you can not accidentally erase or overwrite NBoot.

4 U-Boot

U-Boot bootloader 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.

4.1 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 behavior 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 `0x10800000`.

```
tftpboot 0x10800000 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 10800000 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 usually be further simplified to

```
tftp rootfs.ubifs
```

which is much easier to type.

The following Table 5 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, ...)

Command	Function
mmc	Access MMC/SD card devices (select, show, read, write, …)
tftpboot/nfsboot	Load from TFTP/NFS server (Ethernet)
fatls/ext2ls/ls	List files on FAT/EXT2/arbitrary formatted device (USB,SD)
fatload/ext2load/load	Load from FAT/EXT2/arbitrary formatted devices (USB, SD)
ubifsload	Load from UBI volumes formatted with UBIFS (NAND)
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 5: 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.

4.2 MTD Partitions

The NAND flash of the board 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 the Kernel image is stored at offset 0x200000 and can use up to 0x400000 (about

4mb). When we want to read this region from NAND flash to RAM, we can use the following command:

```
nand read $loadaddr 200000 400000
```

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 giving 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 harddisk. And because this is not restricted to NAND flash, but also works for several other types of non-volatile memories, that are commonly called *Memory Technology Devices* in the Linux world (MTD for short), these regions are called *MTD partitions*.

With command `mtdparts`, you can add or remove such partitions, or list the current MTD partition table. Please note that the command `mtdparts` itself does not actually change anything in NAND flash, it just organizes the list of names, starting offsets and sizes. Table 6 shows the default partition list for fsvybrid boards.

Name	Start	Size
`NBoot`		
`UserDef`		
`Refresh`		
`UBoot`		
`UBootEnv`		
`Kernel`		
`TargetFS`		

Table 6: NAND flash partitioning

The `TargetFS` partition occupies the remaining free space on the flash. Its size therefore depends on the size of the NAND flash that your board is equipped with.

4.3 Environment

Configuration in U-Boot is done by setting variables. The set of all variables is called environment and can be stored in NAND flash so that it is again available at the next start. It is

stored in MTD partition `UBootEnv`. If you want to erase the whole environment, simply erase this partition with

```
nand erase.part UBootEnv
```

Then U-Boot will start with the built-in default environment the next time.

4.3.1 Environment Variables

A variable is set to a value with the command

```
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 `printenv`. For example

```
printenv my_var
```

will now show 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 they would overcrowd the 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.

4.3.2 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 let's say

```
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 the variable with

```
printenv mtdparts
```

you will see that it has the same (rather complex) syntax as is required for the Linux kernel command line, which makes it 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 networking variables to be compatible with your local network.

Environment 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 if U-Boot command menu is not entered on startup
mtdparts	NAND flash partitions
loadaddr	RAM address to load the kernel image (uImage) to/from
bootargs	Basic kernel parameters
arch	Architecture of your board, i.e. fsimx6
platform	Platform name, e.g. armstonea9, qblissa9, picomoda9, efusa9
preboot	A command that is executed before the board is bootet.

Table 7: Important U-Boot settings

4.3.3 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`.

4.3.4 Running Commands in a Variable

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

```
tftp uImage
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. Let's call the variable simply with the one letter `r`.

```
setenv r tftp uImage\; 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 immediately executed. 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 Special F&S U-Boot Features

5.1 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 behavior. 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 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 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 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 if the predefined settings do not suit your needs.

5.1.1 Kernel Settings

These settings listed in Table 8 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 tol 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 8: Variables to set kernel source

5.1.2 Console Settings

The settings listed in Table 9 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 9: Variables to set console output

5.1.3 Login Settings

The settings listed in Table 10 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 10: Variables to set login prompt origin

5.1.4 MTD partition settings

This is simply the common mtdparts variable that holds the current MTD partition table. This variable uses already the correct syntax and is meant to be used in the kernel command line. You can not change anything here for the boot strategy.

5.1.5 Network Settings

The settings listed in Table 11 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
.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 11: Variables to set network activation

5.1.6 Rootfs Settings

The settings listed in Table 12 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 NFS
`.rootfs_ubifs`	Tell kernel to load root filesystem from a UBIFS filesystem located in a UBI volume

Table 12: Variables to set rootfs source

5.1.7 Mode Settings

The settings listed in Table 13 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 13: Variables to set set /deny write access for root filesystem

5.1.8 Init Settings

The settings listed in Table 14 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 14: Variables to set init process file

5.1.9 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
```

5.1.10 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 .rootfs_nfs
run .network_on
```

Boot kernel from MTD partition `Kernel` 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 .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`).

```
setenv bootfile my_yummy_kernel
setenv bootfdt my_yummy_fdt
run .kernel_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.

```
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
```

5.2 NAND Layout Strategies

A great concern when using NAND flash is the fear of losing any data due to aging processes. NAND flash is a rather inexpensive technology with the disadvantage that there may be non-working bits or bits that change their value over time. This is usually compensated by using redundant Error Correction Codes (ECC). With ECC it is possible to correct a small number of bad or flipped bits. The more redundant data is added, the more bit errors can be corrected.

But ECC is not the only way to improve NAND flash behavior. Having a decent wear-leveling algorithm, that distributes erase and write accesses across the whole NAND device instead of “wearing down” single blocks and pages also increases the lifetime of a NAND flash.

The number of bad bits in NAND flash can rise with time even when data is only read! Yes, you heard right, reading data of a NAND flash many million times may lead to bit flips. These so-called *read-disturbances* can be eliminated by erasing the block and writing the data again. This refreshes the bit charge and the bits will work again for the next several million read cycles. This method is called block refresh and it should especially be applied to read-only data that would never be rewritten (and thus renewed) otherwise.

The Unsorted Block Image concept UBI was especially designed to handle exactly these problems. 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 data with many bitflips, or by a low-priority background task that checks all blocks from time to time for bitflips. And UBI provides wear-leveling that tries to keep erase cycles for all NAND blocks at a similar level. This works best if the wear-leveling has as many NAND blocks as possible at its disposition. So having one big UBI works better than two smaller UBIs.

Of course UBI can only use the blocks that belong to the UBI. So all separate MTD partitions that hold any data can not take part in UBI's wear-leveling and block refresh. So from this point of view we should put as much data into UBI as possible and keep seperate NAND regions as small as possible. As UBI is only available in U-Boot and in Linux, we can not extend UBI to the parts of data that have to be read by the CPU (ROM loader) and NBoot. Which means the partitions for `NBoot`, `UserDef`, `Refresh`, `UBoot` and `UBootEnv` need to stay separately. But we can move the `Kernel` partition to UBI. Then at least the kernel region can take part in UBI's block refresh and wear-leveling.

It should be mentioned that moving the kernel image to UBI also has a disadvantage. To be able to load the kernel from UBI, we must call `ubi part` somewhere in U-Boot. This command scans the whole UBI which takes quite some time, depending on the `TargetFS` partition size. So moving the kernel to UBI actually results in a considerably slower boot time.

So the remaining part of this chapter shows how F&S U-Boot can be configured for different storage places of the kernel image, either as a separate MTD partition or in UBI. For doing this we are using in fact a similar concept like in 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, i.e. with or without the `Kernel` partition. This is shown in Table 15. And Table 16 shows a second set of variables that is responsible for creating the necessary UBI volumes, i.e. `rootfs` only or `kernel` and `rootfs`.

Variable	Description
.mtdparts_std	Create a separate MTD partition Kernel for the Linux Kernel
.mtdparts_ubionly	Do not create MTD partition Kernel, both kernel and root filesystem are meant to reside in the UBI located on top of TargetFS

Table 15: Variables to define the MTD partition table

Variable	Description
.ubivol_std	Only create the rootfs volume in the UBI on top of TargetFS
.ubivol_ubi	Create a kernel volume and a rootfs volume in the UBI on top of TargetFS

Table 16: Variables for UBI volume creation

This allows for (at least) the following three Kernel storage strategies:

Use an own partition named Kernel for the Linux kernel. This is the standard layout.

3. Drop the Kernel partition and move the Linux kernel to the UBI on the TargetFS partition, next to the rootfilesystem, but into a separate UBI volume named kernel.

Drop the Kernel partition and move the Linux kernel into the root filesystem itself.

The next three sub-sections will show in detail how to set up each of these 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-leveling information that UBI builds over time, making the whole wear-leveling 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-leveling information. But never erase the underlying TargetFS itself or you will lose all wear-leveling information.

5.2.1 Linux Kernel in MTD partition

To prepare the system in this way, use the following commands. This is how the system is set up by default.

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

To save the downloaded kernel image, use the `nand` command.

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

5.2.2 Linux Kernel In Raw UBI Volume

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

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

To save the downloaded kernel image, 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
```

5.2.3 Linux Kernel In 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
saveenv
```

In this case the kernel is 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 kernel separately within U-Boot. It can only be read with the `ubifsload` command.

5.3 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), and it also has block refresh for regular MTD partitions now. This means that an increasing number of bitflips caused by merely reading this data (read-disturbances) is detected and corrected by automatically rewriting the block.

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, then the original block is erased and finally the data is copied back to the original block. So at any time there is at least one valid copy of the data, either in the original block or in the backup block. If the refresh procedure is interrupted, it will either be continued or repeated from start, depending on the stage where the interruption happened.

Because of this, 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, which is the best compromise between speed and data safety in our view

Note

If you update your system from an older version pre V2.0 to V2.0 or newer, you have to update NBoot, U-Boot and the Linux kernel in one go. Data is passed from one program to the other and if they do not agree about the data format in NAND flash, the next stage can not read the data that the previous stage has written.

5.4 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 in U-Boot to automate these tasks as far as possible. In fact when you look back to the automatic installation from SD card in chaper 3.2 on page 15, you have seen this system in action already.

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

1. It checks at several places for the script file `update.scr`. If the file is found, it is executed. The file is usually used to do a system update.
2. If no `upate.scr` 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 places for the script file `install.scr`. If it is found, it is executed. The file is usually used to install Linux on an empty system.

The places where U-Boot looks for these scripts and also the script names can be configured by environment variables. Possible places are: SD card, USB stick, NAND flash, UBI volume, RAM, TFTP, NFS. It is also possible to switch off this feature completely. By default, `update.scr` and `install.scr` are searched for on SD card and USB stick if supported.

For a detailed description of the variable names that can be used for configuration and also the syntax of how the search places are defined just call

```
help update
```

5.5 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 behavior of U-Boot. When used in 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.

In our examples throughout this document we will also always use the long form `$loadaddr` for two reasons. Firstly a single dot may easily be overlooked and considered “dirt” on the paper and skipped at the input. And secondly 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.

5.6 Allow Wildcards in FAT Filenames

USB sticks or MMC devices (SD cards) are usually 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, it is difficult to load the correct file there as U-Boot can 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).

And it can also work with wildcards (see Table 17).

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 17: Wildcards in FAT names

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

```
armStoneA5 # mmc rescan
armStoneA5 # ls mmc 0
/:
```

```
     524288  uboot-fsvybrid.nb0
    4497696  uImage-fsvybrid
   57266176  rootfs-fsvybrid.ubifs
```

So for example to load the `rootfs-fsvybrid.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.

```
armStoneA5 # ls mmc 0 *fsvybrid*
/:
    4497696  uImage-fsvybrid
   57266176  rootfs-fsvybrid.ubifs
armStoneA5 # load mmc 0 $loadaddr *fsvybrid*
/: Ambiguous matches for "*fsvybrid*"
```

Note

At the moment this only works for FAT, not for EXT2/EXT4 or UBIFS.

6 Using the Standard System and Devices

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 `fsvybrid` platform uses `udev` as it needs some 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.

6.1 The 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 fsvybrid.

```
# ls /sys/class/
backlight/    i2c-adapter/ mmc_host/    regulator/   spidev/
bdi/          i2c-dev/     mtd/         rtc/         tty/
block/        input/       mvf-adc.0/   scsi_device/ ubi/
dma/          leds/        mvf-adc.1/   scsi_disk/   vc/
firmware/     mdio_bus/    mxc_asrc/    scsi_host/   video4linux/
gpio/         mem/         net/         sound/       vtconsole/
graphics/     misc/        pwm/         spi_master/
```

For example access the RTC subsystem

```
# cat /sys/class/rtc/rtc0/date
2016-10-18
```

6.2 Serial

The default speed is 115200 bit/s. On NXP CPUs, the devices are called `/dev/ttymxc<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 redirection to use a serial port from the command line.

Example:

```
echo Hello > /dev/ttymxc2
```

6.3 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
```

Then you can activate the CAN device:

```
ip link set can0 up
```

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`).

6.4 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 eth0
```

6.5 GUI

The default GUI 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 and/or USB keyboard (e.g. by using a USB hub) and then move the mouse cursor and type commands to the terminal window.

The whole system is not very functional and just demonstrates how a GUI could be implemented. We don't want to get a too large default root filesystem and we don't want to influence your decision of what type of GUI to use (QT, GTK, DirectFB, etc).

You can start some X applications on the command line:

```
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.

6.6 Qt support

Qt is a cross-platform application framework that is widely used for developing applications software with a graphical user interface. Qt is *not* included in our standard configuration. If you want to use Qt, you have to add it to the configuration first. The Qt package has a quite large set of configuration parameters of its own. It is beyond a first steps documentation to handle this.

6.7 SPI

This device can be used with the usermode SPI driver (also called spidev).

6.8 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
```

6.9 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
```

6.10 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.

6.11 Touch

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 subdirectory `/etc/X11/xorg.conf.d` and adding a file `99-calibration.conf` that contains the calibration data that was reported by `xinput_calibrator`.

6.12 GPIO

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 COL (armStoneA5: J12#3 / PTD2) as output pin. The pin number on J12 is 3. The “GPIO Reference Card” shows in row `/sys/class/gpio/gpio#` the number 81. To get this pin into sysfs write:

```
echo 81 > /sys/class/gpio/export
```

This creates a new directory `gpio81` in /sys/class/gpio that is used for all further settings of this GPIO.

```
ls /sys/class/gpio/gpio81/
active_low  direction   edge        power       subsystem
uevent      value
```

Set pin as output:

```
echo out > /sys/class/gpio/gpio81/direction
```

Set pin to high level:

```
echo 1 > /sys/class/gpio/gpio81/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/gpio81/value
```

Now pin has a low level. To set a pin as input, write `in` into `direction`.

6.13 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>
```

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
```

Then record the stereo file with high quality with

```
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
```

6.14 Pictures

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
```

6.15 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
```

6.16 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>
```

6.17 SSH

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.

6.18 VNC

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
```

7 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 usually rather slow 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`.

7.1 Install Cross-Compile Toolchain

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-5.2.0-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-5.2.0-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`.

7.2 Installing the mkimage tool

The mkimage tool is used to create a uImage when compiling the kernel. 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 these two environment changes permanent, for example `/etc/profile` or `~/.bashrc`.

7.3 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 and Buildroot next to each other, so that the top directories of their source trees are siblings.

We have prepared a shell script called `install-sources.sh` 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. The build directory must already exist when the script is called.

```
cd <release-dir>
mkdir <build-dir>
./install-sources.sh <build-dir>
```

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.

```
tar xf examples-fus-V<x>.<y>.tar.bz2
tar xf u-boot-2014.07-fsvybrid-V<x>.<y>.tar.bz2
tar xf linux-3.0.15-fsvybrid-V<x>.<y>.tar.bz2
tar xf buildroot-2014.02-fsimx6-V<x>.<y>.tar.bz2
ln -s linux-4.1.15-fsimx6-V<x>.<y> linux-fsimx6
```

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.

7.4 Compiling U-Boot

In your build directory, simply go to the U-Boot source directory, run a configuration command to tell U-Boot to compile for the fsvybrid architecture, and then start the build process. This can be done with the following three commands.

```
cd u-boot-2014.07-fsvybrid-V<x>.<y>
make fsvybrid_config
make
```

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.

7.5 Compiling the Linux Kernel

Go to the Linux source directory and run a configuration command to tell Linux to build for the fsvybrid 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-3.0.15-fsvybrid-V<x>.<y>
make fsvybrid_defconfig
make uImage
```

The final kernel image can be found in `arch/arm/boot/uImage`.

7.6 Compiling 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 two different fsvybrid configurations to Buildroot. A minimal one that just includes Busybox and the GLIBC library. And a standard image that includes gstreamer for multi-media, ALSA for sound support and a small X-Server with the matchbox window manager and a handful of X applications. 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.

Again simply go to the Buildroot directory, issue a configuration command to tell Buildroot to build for the fsvybrid architecure, and then start the build process. The following commands will build the standard fsvybrid configuration.

```
cd buildroot-2016.05-fsvybrid-V<x>.<y>
make fsvybrid_std_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 redone, 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.

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. One of the settings is the path for the toolchain. If you have installed the toolchain under `/usr/local/arm` as recommended, everything will work right out of the box. But if you installed the toolchain at a different place, you have to modify the setting in *Toolchain → Toolchain Path.*

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.

8 Appendix

Listings

List of Figures

List of Tables

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.