DOCUMENTATION

emtrion
embedded systems

emSBC-Argon Debian Manual

Debian Based BSP Manual

Rev. 1 / 11.05.2021
Author: Jan von Wiarda
Last Change: July 9, 2021

Revision: Rev. 1 / 11.05.2021

Rev.	Date/Initial	Changes
1	11.05.2021/Wi	Initial Revision

Contents

1 Terms and definitions 5

2 Introduction 6

3 Requirements for the development PC 7
3.1 Requirement on the development PC for network boot of the target . . . 7

4 Device connection 9
4.1 Serial console preparations . . . 9
4.2 Device boot . . . 9
4.2.1 Default login credentials . . . 9
4.3 Network setup . . . 10
4.4 Using DHCP . . . 10
4.5 Using a fixed IPv4 address . . . 10
4.6 Network configuration check . . . 11

5 Booting Linux 12
5.1 Boot configuration . . . 12
5.2 Stand alone boot . . . 12
5.3 Network remote boot . . . 13

6 Device software components 14
6.1 Bootloader (U-Boot) . . . 14
6.1.1 Basic U-Boot operation . . . 14
6.1.2 Boot architecture . . . 15
6.1.3 Network configuration . . . 15
6.1.3.1 Network setup using DHCP . . . 15
6.1.3.2 Network setup with static addresses . . . 16
6.1.3.3 Network troubleshooting in U-Boot . . . 16
6.1.4 Predefined macros . . . 16
6.1.4.1 Macros intended to update the complete system . . . 17
6.1.5 U-Boot recompilation . . . 17
6.2 emPURS . . . 17
6.2.1 emPURS for production purposes . . . 18
6.3 Restoring the Root Filesystem . . . 18
6.4 Linux kernel . . . 18
6.4.1 Kernel-package recompilation . . . 19
6.5 Debian BSP . . . 19
6.5.1 Board support packages . . . 19
6.5.1.1 emboot . . . 19
6.5.2 Development configuration . . . 19

7 Custom Debian BSP creation **20**
7.1 Embedded Linux build system . . . 20
7.1.1 Setup elbe on the host . . . 20
7.1.2 Virtual build machine preparation . . . 20
7.1.3 Custom Debian BSP creation . . . 21
7.2 Local repository server . . . 21
7.2.1 Apache installation and configuration . . . 21
7.2.2 Basic repository configuration . . . 22
7.2.3 Managing Debian packages . . . 22

8 Common tasks **24**
8.1 Find packages . . . 24
8.2 Add or remove packages . . . 24
8.3 Network configuration . . . 25
8.4 OpenAMP demo . . . 25
8.4.1 Prerequisite . . . 25
8.4.1.1 Adding Debian Package . . . 25
8.4.2 OpenAMP_FreeRTOS_echo . . . 25
8.4.3 OpenAMP_raw . . . 26
8.5 Custom application autostart . . . 27
8.6 Peripherals access . . . 28
8.6.1 GPIOs . . . 28
8.7 Source code acquisition . . . 29

9 Getting help **30**
9.1 Known Issues . . . 30
9.2 Support center . . . 30
9.3 Debian documentation and community . . . 30
9.4 Commercial support . . . 30
9.5 Technical support . . . 31

10 Appendix **32**
10.1 emSTAMP-series . . . 32
10.1.1 emSTAMP-Argon . . . 32
10.1.1.1 on emSBC-Argon . . . 32

1 Terms and definitions

The table below lists some definitions of terms in this manual.

Term	Definition
Target	Synonym for the embedded device
Host	Workstation, Developer PC
Toolchain	Compiler, Linker, etc.
RootFS	Root file system, contains the basic operating system
Console	Text terminal interface for Linux
NFS	Network File System, which can share directories over network
NFS_SHARE	Location that is exported by the NFS for the purpose of updating and booting by using NFS
U-Boot	Bootloader, hardware initialization, updating images, starting OS
IDE	Integrated Development Environment
OS	Operating System
SoC	System on chip
BSP	Board Support Package
SDK	Software Development Kit
DFU	Device Firmware Upgrade mode, allows all devices to be restored from any state
STM32CubeProg	An all-in-one multi-OS software tool for programming STM32 products

Table 1.1: *Terms and definitions*

2 Introduction

Welcome to emtrion's Debian-based Linux board support package. This short manual gives you a startup with our BSP. It describes how to setup your host PC to develop your application for the emtrion developer kit hardware. Furthermore, it gives you a quick introduction to Debian, and explains our motivation behind our decision to release a Debian-based BSP.

It is assumed that users of emtrion Linux developer kits are already familiar with Linux. General Linux and programming knowledge are out of the scope of this document. emtrion gladly assists you in acquiring this knowledge. If you are interested in training courses or getting support, please contact the **emtrion** sales department via sales@emtrion.de.

This guide shows you how to get started with the developer kit. It also explains how to setup a network connection.

The examples in this manual are demonstrated on specific hardware, however, if not mentioned otherwise, they work on all supported emtrion devices.

Attention

Our server may be used in combination with our developer kits only. The servers are not meant to be used from final devices in the field. **emtrion** reserves its right to change the access permissions to the server as well as the support for certain platforms at any time.

3 Requirements for the development PC

For using **emtrion**'s Development Kit you need a PC - or a virtual machine - which you can use as a development PC. On the development PC you can create your application and/or create your root filesystem. Both are running on the target.

As operating system for the development PC, we recommend the current stable release of the Debian distribution which at the time of writing this manual is Debian 10 (Codename Buster). Regardless, you can also use an other Linux distribution. But in this case the commands required on the development PC may differ from the commands listed in this manual.

Info

Please understand that **emtrion** can not guarantee full functionality if another distribution is used. However, **emtrion** can support you as part of the commercial support in the cause search if there are issues comming up. More information how you get commercial support you will find in chapter 9.4.

3.1 Requirement on the development PC for network boot of the target

The target device can boot a root file system from the network when it is made available via NFS. For the development PC to be able to provide the root file system, a NFS server must be installed and configured on it.

The development PC have to fulfill two requirements:

- the NFS server components must be installed
- the directory with your target root filesystem has to be exported

On Debian or Ubuntu the NFS server can be installed using these commands:

```
sudo apt install nfs-kernel-server nfs-common
```

Depending on the bootloader version, the NFS protocol version 2 is required. Newer versions of Debian or Ubuntu distribution have disabled this protocol version by default. To activate, you have to edit the file */etc/default/nfs-kernel-server*. You have to add the part *–nfs-version 2* to the parameter *RPCNFSDCOUNT*:

```
RPCNFSDCOUNT="8 --nfs-version 2"
```

After you have modified the file */etc/default/nfs-kernel-server* you have to restart the NFS service

```
sudo systemctl restart nfs-kernel-server
```

If the NFS server is installed you have to export the directory where the root filesystem for the target is stored. Instead of this directory it's sufficient if you export a parent directory of this directory. The following example allows access to the directory */home/hico/nfs/rootfs* and all subdirectories below.

```
# /etc/exports: the access control list for filesystems which may be exported
#               to NFS clients.  See exports(5).

/home/hico/nfs    *(rw,sync,no_subtree_check,crossmnt,no_root_squash)
```

Listing 3.1: *example /etc/export file*

When you modify the file */etc/exports* you have to restart the NFS service on your development PC:

```
sudo systemctl restart nfs-kernel-server
```

Instead of restarting the NFS service you can use the following command to introduce the NFS service to reload the configuration

```
sudo exportfs -ra
```

4 Device connection

4.1 Serial console preparations

After starting Linux you can log into the console. This is located on the same serial interface as the bootloader console. This serial interface also output the startup messages of Linux. You will find this serial interface on the connector J26 on the Avari base board, on the connector J8 (UART A) on the Bvari base board and on the connector J4 on the emSBC-Argon base board. On your development PC you need a serial terminal application such as picocom. The communication settings are:

baud rate	115200 bps
data bits	8
stop bits	1
parity	none
handshake	none

Table 4.2: *communication settings*

Example using picocom (exchange ttyUSBo with the corresponding UART on your development PC)

```
sudo apt install picocom
sudo picocom -b 115200 /dev/ttyUSB0
```

4.2 Device boot

The devices in our development kits are configured to boot automatically from the integrated mass storage device. Depending on the hardware platform this might be SLC-flash or an eMMC storage. The autostart can be interrupted during the 3 second timeout and then reconfigured as described in chapter 5.

4.2.1 Default login credentials

The default-login credentials for the Debian-based BSP's released by **emtrion** are

- Username: root
- Password: hico

4.3 Network setup

After the System has completed its boot-process, the network interface of the device has to be configured. Basically, two setups are possible, depending on your company's requirements. The following sections show you how to do a temporary network setup. Please refer to the detailed BSP documentation in chapter 8.3 if you require instructions for a permanent configuration.

4.4 Using DHCP

To receive an IP from the DHCP server and to setup the interfaces, run the following command on your device. Depending on the network environment, this can take anything, from a few seconds up to around one minute for auto negotiation and address setup. To check if the setup was successful, please refer to section 4.6

```
root@device~# dhclient eth0
```

Listing 4.1:

4.5 Using a fixed IPv4 address

The fixed IP-address setup requires a few more steps than the auto configuration. Besides the address setup, it is also recommended to configure a nameserver in order to be able to install packages and communicate with the internet.

You require the following information before starting:

1. IP-address to be used on the device
2. Network mask
3. Nameserver address
4. Gateway IP (optional)
5. Broadcast address (optional)

The powerful *ip* utility is used for the network interface setup, which is the successor of the well known *ifconfig*. The netmask has to be transformed into CIDR notation, which uses the number of ones in the netmask, corresponding to its binary representation, appended to the IP-Address. The netmask *255.255.0.0*, for example, results in a */16* suffix. In combination with the IP-address *172.26.1.2* it results in *172.26.1.2/16*.

```
root@device~# ip address add 172.26.1.2/16 dev eth0
```

Listing 4.2:

With optional broadcast address:

```
root@device~# ip address add broadcast 172.26.255.255 dev eth0
```

Listing 4.3:

Adding the default gateway (e.g. 172.26.1.1):

```
root@device~# ip route add default via 172.26.1.1
```

Listing 4.4:

Finally, the nameserver has to be configured to match your environment. Simply write the configuration */etc/resolv.conf* file as shown below (we are using *172.26.1.255* as nameserver in this example):

```
root@device~# echo "nameserver 172.26.1.255" > /etc/resolv.conf
```

Listing 4.5:

4.6 Network configuration check

For this, simply run *ip addr show etho* to see your network interface setup. Depending on the chosen setup, the correct information is displayed.

```
root@device~# ip addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group
    default qlen 1000
link/ether 00:1c:1e:08:e4:45 brd ff:ff:ff:ff:ff:ff
inet 172.26.4.95/16 brd 172.26.255.255 scope global dynamic eth0
valid_lft 34179sec preferred_lft 34179sec
inet6 2003:5a:a012:1:21c:1eff:fe08:e445/64 scope global dynamic mngtmpaddr
valid_lft 2591861sec preferred_lft 604661sec
inet6 fe80::21c:1eff:fe08:e445/64 scope link
valid_lft forever preferred_lft forever
root@device~#
```

Listing 4.6: *Verifying the network configuration*

5 Booting Linux

The preinstalled bootloader offers several possibilities to boot the system. A detailed description of its configuration and features is included in chapter 6.1. There is also a short introduction into the basic usage of U-Boot. The following chapter describes the available boot modes and their configuration.

5.1 Boot configuration

You can configure the U-Boot's boot mode by setting the environment variable `bootcmd` to the corresponding value. There is a default delay of 3 seconds for the command execution, this can be changed by modifying the environment variable `bootdelay`. The default boot mode is already setup and saved into the environment as an example.

Warning

If the bootdelay is set to zero, the prompt cannot be accessed anymore.

```
U-Boot> setenv bootcmd 'run sd_boot'
U-Boot> saveenv
```

Listing 5.1: *Setting your own* `bootcmd` *variable*

5.2 Stand alone boot

The stand-alone boot mode implemented in most emtrion modules is named `flash_boot`. As the name already indicates, the needed system components are loaded from the local flash memory (e.g. eMMC) into RAM, where they get executed. It is also the default bootsource of all emtrion modules with pre-installed Linux BSPs. To use it, please check the base requirements described in chapter 6.1 Bootloader (U-Boot). You can manually execute it by using the following command inside the U-Boot prompt:

```
U-Boot> run flash_boot
```

Listing 5.2: *Running the default boot command* `flash_boot`

5.3 Network remote boot

Also a remote network-based boot mode is available on most systems. It loads the system components from a remote storage into the local RAM and executes them there. The usage of any remote file system has to be supported by the chosen operating system, in this case Linux.

A default network configuration is used in the bootloader to gain the IP-address. It can also be configured to use a static setup. Please refer to chapter 6.1 Bootloader (U-Boot) for detailed instructions.

NFS is the network based boot mode described and supported in this BSP. Some basic configurations are needed before the network boot can be done. First of all, the remote station - the development PC - must have an running and configured NFS server (see 3.1). Then there are some instructions left to execute on the target. For the list below we assume that the NFS server exports the directory `/home/hico/nfs/` and it's subdirectories. We also assume that the development PC uses the IP address `172.26.1.1` on it's network interface. Then we assume that the root filesystem to boot is stored in the directory `/home/hico/nfs/rootfs` on the development PC.

```
1 U-Boot> setenv serverip 172.26.1.1
2 U-Boot> setenv nfsroot /home/hico/nfs/rootfs
3 U-Boot> run net_boot
```

Listing 5.3: *Set* `serverip` *and* `nfsroot` *and boot via NFS*

6 Device software components

6.1 Bootloader (U-Boot)

The basic task of U-Boot is to read the operating system dependencies from bulk memory into RAM and start its kernel. It is also possible to update several components of the system by using the bootloader functionalities. **emtrion** 's U-Boot version contains several useful extensions, which are explained in the following sections.

6.1.1 Basic U-Boot operation

To work with U-Boot, please use a terminal program to connect to the serial line of the board first of all. As soon as the U-Boot prompt appears on the terminal, U-Boot is ready to receive commands. To interrupt the automatic boot mechanism, press any key on your keyboard whilst using a terminal. Here you can find the U-Boot documentation: `http://www.denx.de/wiki/U-Boot/Documentation`

U-Boot has a set of environment variables which are used to store all information needed for booting the operating system. Variables can contain information such as IP addresses, but they can also contain a whole script of actions which are to be performed sequentially. The following commands explain the basic handling of environment variables and the custom emtrion command to restore the default settings: Beside the basic commands, there are a few emtrion-specific

U-Boot command	Explanation
printenv [variable]	This shows the value of the specified variable. If no variable is specified, the whole environment is shown
setenv [variable] [value]	Set a variable to a specific value. If no value is specified, the variable gets deleted.
saveenv	Make your changes permanent, so they remain after power off or reboot.
run [script_variable]	Tries to execute the commands contained inside the script_variable, which in fact is a variable as noted above.

Table 6.1: *Basic U-Boot commands*

environment variables which are not intended to be changed by the end customer. Once deleted, there is no way to restore them automatically.

Environment variable	Explanation
ethaddr	The ethernet MAC address of the system.
hw_serial_nr	The serial number of the CPU-module or SBC.
hw_revision	The hardware revision of the CPU-module.
hw_product_type	Product name of the module as printed on the label above the barcode.

Table 6.3: *Predefined environment variables*

6.1.2 Boot architecture

emtrion uses a boot-architecture, which depends on a file contained in the corresponding rootfilesystem. The intention is to enable the bootloader to load different operating systems, so there is no need to change the bootloader. Furthermore, it is possible to change the system's boot-behavior during in-field modifications without needing to access the bootloaders environment. In default configuration, the U-Boot version delivered by emtrion tries to load the file `uboot_script` from the directory `/boot/` inside the specified partition or network path (corresponding to the selected boot mode). If the local flash is specified as boot source, the partition holding the directory boot can be formatted in any version of the ext-filesystem or fat-filesystem.

Note

Depending on the version of U-Boot on the module it may be mandatory to have the partition formatted without 64Bit support due to recent changes in the default of the extended file system. For this, please call the mkfs.ext4 utility in combination with the following options:
mkfs.ext4 –O^64bit,^metadata_csum

The script, used by the boot system parts contained in U-Boot, has to export at least the following variables, which have to be executable by the `run` command. They follow the format `uboot_script_` + `action_name`. A reference implementation can be found in emtrion's developer kits. A short description of those variables can be found in Table 6.4 Script variables inside uboot_script.

Script variable	purpose
uboot_script_update_rootfs	Load emPURS components and start the update of the rootfilesystem. empurs_cmd update_rootfs has to be set by the script.
uboot_script_net_boot	Extension of net_boot script contained in U-Boot. The script is responsible for loading all needed components like kernel, device tree and initrd. It also has to setup all bootargs for the remote boot.

Table 6.4: *Script variables inside uboot_script*

The script variables denoted in the table above are used in several predefined U-Boot macros, which are described in detail in Chapter 6.1.4 Predefined macros.

6.1.3 Network configuration

The U-Boot network configuration has to be set by using environment variables. There are two options supported by emtrion, we recommend the first one, which is auto configuration by DHCP. The second option is a static address setup. In general, the configuration is handled by the script `configure-ip`, which depends on the variable `ip-method` itself. Two values are allowed for this variable, as described in the following chapter. The first one is `dhcp`, the second one `static`. The `configure-ip` script also creates the correct command line for the Linux kernel network configuration via the `ip=` kernel command line parameter.

6.1.3.1 Network setup using DHCP

In order to configure the usage of the DHCP protocol in U-Boot for the use with predefined scripts like `net_boot`, please set the variable `ip-method` to `dhcp`.

```
U-Boot> setenv ip-method dhcp
U-Boot> run net_boot
```

Listing 6.1: *Set the* `ip-method` *to* dhcp

You can also obtain an IP address manually via the U-Boot command `dhcp`. It is recommended to set the environment variable `autoload` to no, otherwise U-Boot tries to load an optionally provided file from the DHCP server via the TFTP protocol. If this file is not found, the received address is not kept for later usage.

```
U-Boot> setenv autoload no
U-Boot> dhcp
BOOTP broadcast 1
DHCP client bound to address 172.26.1.2 (4 ms)
```

Listing 6.2: *Setting* `autoload` *to* no

6.1.3.2 Network setup with static addresses

Besides auto configuration via the DHCP protocol, it is also possible to adjust the IP address settings manual. As denoted in the section above, the 'configure-ip' script can be used to setup the Linux-kernel parameters. To use scripts or U-Boot network commands like 'nfs' or 'tftp', it is required to set the following environment variables:

Environment Variable	Explanation
ipaddr	The IP address to be used by the device.
serverip	IP address of the server to interact with.
netmask	The network mask of the network, to which the device will be connected.
gatewayip	(Optional) IP address of the gateway to be used. This is only necessary if the device and the server are not in the same network

Table 6.5: *Network configuration variables*

6.1.3.3 Network troubleshooting in U-Boot

If you are facing difficulties when using any of the predefined scripts or your own commands, you can simply check the basic connection and correct address setup by using the ping command in U-Boot. Please note, this ping implementation is very limited and only checks once the availability of the host.

Note

As name resolution is not available in U-Boot, you have to know the IP address of the hosts you want to communicate with.

6.1.4 Predefined macros

The emtrion U-Boot variants have a set of predefined common macros: There are three major variant groups. The following sections give you a brief overview of their usage and the intention behind them. The first group contains general-purpose macros, responsible for booting the system. Secondly, there are macros intended to update or restore

the operating system. The lasts group consists of macros to update the bootloader itself, which should only be used if instructed to do so.

Macro Variable	Explanation
configure_ip	As described in section 6.1.3 Network configuration, this macro configures and validates the network depending on the value of the variable `ip-method`
flash_boot	Starts the operating system contained in the flash memory of the system.
net_boot	Starts the operating system available on remote-storage and exported by NFS. It needs a valid network setup, as described in section 6.1.3 Network configuration, as well as the correct setup of the environment variables `serverip` and `nfsroot`.

Table 6.6: *Predefined macros*

6.1.4.1 Macros intended to update the complete system

Macro Variable	Explanation
update_rootfs	Is used to update the operating system installed in the flash memory.

Table 6.7: *Predefined macros to update the operating system*

6.1.5 U-Boot recompilation

In case you want to recompile the U-Boot binary yourself, follow the instructions. In the example below, the U-Boot version v2020.01 is used.

Recompile the second stage bootloader

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- emsbc-argon_defconfig
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j$(nproc)
```

Listing 6.3: *Cross-compiling on the host*

This creates multiple different binaries. The U-Boot binary used in the trusted environment boot is `u-boot.stm32`.

In chapter "8.7 Source code acquisition" is described, how the source-code for several-packages can be obtained.

6.2 emPURS

The acronym **emPURS** stands for **em**trion **P**roduction **U**pdate and **R**ecovery **S**ystem. It is intended to be used for production, update and recovery tasks. To make use of its features there are several requirements to be met. In this section, the basic operation of emPURS is described. If your hardware is supported by emPURS, you can find specific manuals and packages on the emtrion support pages, describing all detailed steps required. The emPURS ecosystem is a modular, script based, approach to setup and restore onboard flash memory content of your emtrion module. The first component of emPURS is a script executed inside the initial ram disk (initrd). If no empurs_cmd is found in the kernel command line, it proceeds to start the system. If an emPURS command is detected, the emPURS_plat script as well as the steps configured inside get executed. The Debian rootfilesystem wipes the system partition and reinstalls the Debian-BSP exported by the NFS-share.

6.2.1 emPURS for production purposes

For several core modules, emtrion offers support for using emPURS to produce boards with pre-defined software packages. Documentation of how to use this particular emPURS version is available in the hardware section of each supported module on the support pages.

6.3 Restoring the Root Filesystem

If you want to restore the Debian Root Filesystem, you can do so via U-Boot and a NFS server share.

First setup the NFS server on your development PC as described in chapter 3.1. Then extract the `restore.tar.gz` archive to your development PC to folder `/home/hico/nfs/restore`. The directory structure should then look like this:

```
hico@ntb004:~$ ls -la /home/hico/nfs/restore/boot
total 101216
drwxr-xr-x 2 root root     4096 Jul  6 16:46 .
drwxr-xr-x 3 root root     4096 Jul  6 16:13 ..
-rwxr-xr-x 1 root root    18501 Jul  6 16:14 emPURS_plat
-rw-r--r-- 1 root root 91673186 Jul  6 16:14 emsbc-argon-buster.tar.gz
lrwxrwxrwx 1 root root       16 Jul  6 16:46 linux -> linux-5.4.56.itb
-rw-r--r-- 1 root root  5005124 Jul  6 16:14 linux-5.4.56.itb
-rw-r--r-- 1 root root  6920537 Jul  6 16:14 ramdisk-emsbc-argon.rootfs.cpio.gz
-rw-r--r-- 1 root root     1571 Jul  6 16:14 uboot_script
```

Listing 6.4: *Files in NFS server share* `/home/hico/nfs/restore/boot`

Then start the system and hit any key to stop autoboot. You then have a U-Boot shell. First set the `nfsroot` and `serverip` environment variables:

```
STM32MP> setenv nfsroot /home/hico/nfs/restore
STM32MP> setenv serverip [IP address of your development PC]
```

Listing 6.5: *Set environment variables*

Finally, you can start the restore process via `update_rootfs`:

```
STM32MP> run update_rootfs
```

Listing 6.6: *Start restore*

Do not turn of the system while the restore process is running. The system will reboot into Debian after the restore process has finished.

6.4 Linux kernel

The Linux Kernel used in **emtrion** 's recent BSPs is a position independent image. This image depends on the device-tree support, which has been introduced in recent versions of **emtrion** BSPs. The device-tree replaces the outdated boardfile. It is an extra file, describing the hardware platform on which the kernel is running on. New technology makes it possible to have one kernel image running on several different boards, as it might be known from personal computers.

6.4.1 Kernel-package recompilation

During your evaluation it might be necessary to re-build the kernel to include custom drivers or configuration changes. This chapter gives you a short overview on how to create a Debian package. In the following examples, the kernel version 5.4.76 is used. After each Debian package build, the resulting packages get placed in the folder located above the kernel source tree. To install the resulting images, please refer to chapter "8.2 Add or remove packages".

Compile Kernel and create Debian package

```
$ export DEBFULLNAME="Your Name"
$ export DEBEMAIL="your.email@domain.net"
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- emsbc-argon_defconfig
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j$(nproc) FIT_SOURCE_FILE=
    fit_image.its LOCALVERSION= fit-deb-pkg
```

Listing 6.7: *Cross-compiling on the host*

In chapter 8.7 Source code acquisition is described, how the source-code for several-packages can be obtained.

6.5 Debian BSP

6.5.1 Board support packages

Several non-standard packages are included in the emtrion Debian BSP. They provide support for emtrion specific functionalities. The following sections describe these packages, their contents and purpose.

6.5.1.1 emboot

The `emboot` package contains files and configurations necessary to boot the system on emtrion based boards containing the original firmware. A detailed description of the first file, called `uboot_script`, can be found in chapter 6.1.2. The second file provided by this package is a base-configuration of `emPURS`, enabling the restoration of the system with the developer-Kit VM. There are also some hooks in `initramfs-tools`, a tool-suite used to dynamically create an initial Ramdisk. The purpose of these hooks is to install `emPURS` into the initial Ramdisk and to make sure that all of its dependencies are installed. Finally there are some notable scripts to support the system boot with a devicetree. On recent ARM-kernels, the devicetree is used to describe the hardware instead the outdated boardfile. To load `uboot_script` during startup, the devicetree files have to be in the right place with the correct naming. Two scripts are used to achieve this. They are placed inside the kernel installation/removal hooks (`/etc/kernel/postinst.d/`; `/etc/kernel/postrm.d/`; filename `uboot_setup`). Those hooks are executed every time a new kernel is installed or an old one removed. They ensure the availability of the devicetree, supporting the corresponding emtrion hardware, in the current kernel during execution. And if this is not the case, they ensure the system to stay bootable.

6.5.2 Development configuration

The standard Debian BSP is intended to be used as a development system only! Therefore some special adaptations and settings have been made. The default Debian BSP contains a lot of development and documentation packages resulting in a huge amount of used disk space. Optimized images are available through commercial support (ref. chapter 9.4).

7 Custom Debian BSP creation

7.1 Embedded Linux build system

The emtrion Debian image running on your system was created by using the Embedded-Linux-Build-Environment (elbe) developed by Linutronix. Its key benefit is to create reproducible Debian images for embedded systems out of the Debian binary packages, without polluting the host system. It therefore creates its own virtual machine, which is used as a build container. The images used in the Virtual Machine. The official elbe documentation is available under `https://elbe-rfs.org/docs/`.

7.1.1 Setup elbe on the host

As already mentioned, elbe is developed by Linuxtronix and available through their servers. The elbe installation is relatively simple and requires a few steps only. The easiest way is to use pre-built binaries, which are available on the server. To use pre-built binaries, the server address has to be added to the Virtual-Machines apt-settings first. As described in chapter 3.1, it is necessary to have the virtualization extensions available inside the Virtual-Machine.

```
$ echo "deb http://debian.linutronix.de/elbe buster main" > /etc/apt/sources.
    list.d/linutronix
$ echo "deb http://debian.linutronix.de/elbe-common buster main" >> /etc/apt/
    sources.list.d/linutronix
```

It is also recommended to install the linuxtronix gnupg key from their repository:

```
$ wget http://debian.linutronix.de/elbe-common/elbe-repo.pub
$ sudo apt-key add elbe-repo.pub
```

Finally, package lists are updated and elbe packages can be installed:

```
$ sudo apt update
$ sudo apt install elbe elbe-doc
```

7.1.2 Virtual build machine preparation

As described in previous sections, elbe uses a Virtual Machine as a build-container. It has to be created once, prior to the creation of custom root file systems.

Warning

This step requires at least 80 GB of free hard disk space!

```
$ elbe initvm -directory /path/to-place/elbe-vm/ create
```

7.1.3 Custom Debian BSP creation

Once the initial elbe Virtual Machine is created, it can be fed with project configurations in xml-format, as described in the official elbe documentation. However, it is recommended to ensure the virtual machine instance is up and running beforehand.

```
$ cd /path/to-place/elbe-vm/
$ elbe initvm ensure
```

If this is the case, the project XML can be sent into the virtual machine. elbe includes the XML used for its creation in the root-file-system *'image built'* under /etc/elbe_base.xml by default. In addition to the plain project XML, this file does also include information of the package versions used during the initial build. It is recommended to remove this file in a finetuning rule, as it contains information like passwords in plain text. However, it is included in the development Image to enable our customers to recreate their own Images using elbe.

```
$ cd /path/to-get/elbe-xml/
$ elbe initvm -output /path/to-drop/elbe-rfs/ submit emsbc-argon-buster-noX.xml
```

Listing 7.1: *Create Debian image for emSBC-Argon*

7.2 Local repository server

To serve own custom packages, it is necessary to have a local repository server available. The following sections give an overview on how to setup a local package archive with reprepro served by the apache webserver. The following instructions do **not** include security settings, it is highly recommended to restrict access to several components of the repository.

7.2.1 Apache installation and configuration

A local webserver is required to handle the repository creation described in the following sections. Apache2 packages are used as a base for this task.

```
$ sudo apt install apache2 reprepro
```

The apache webserver handles the directory /var/www/ on Debian installations by default. In recent distributions, every site-configuration has its own subfolder inside.

7.2.2 Basic repository configuration

A repository created by *reprepro* has the following structure. The *'conf'* folder, containing configuration files, has to be created to get this set up. The rest of the structure is created automatically during the repository setup phase.

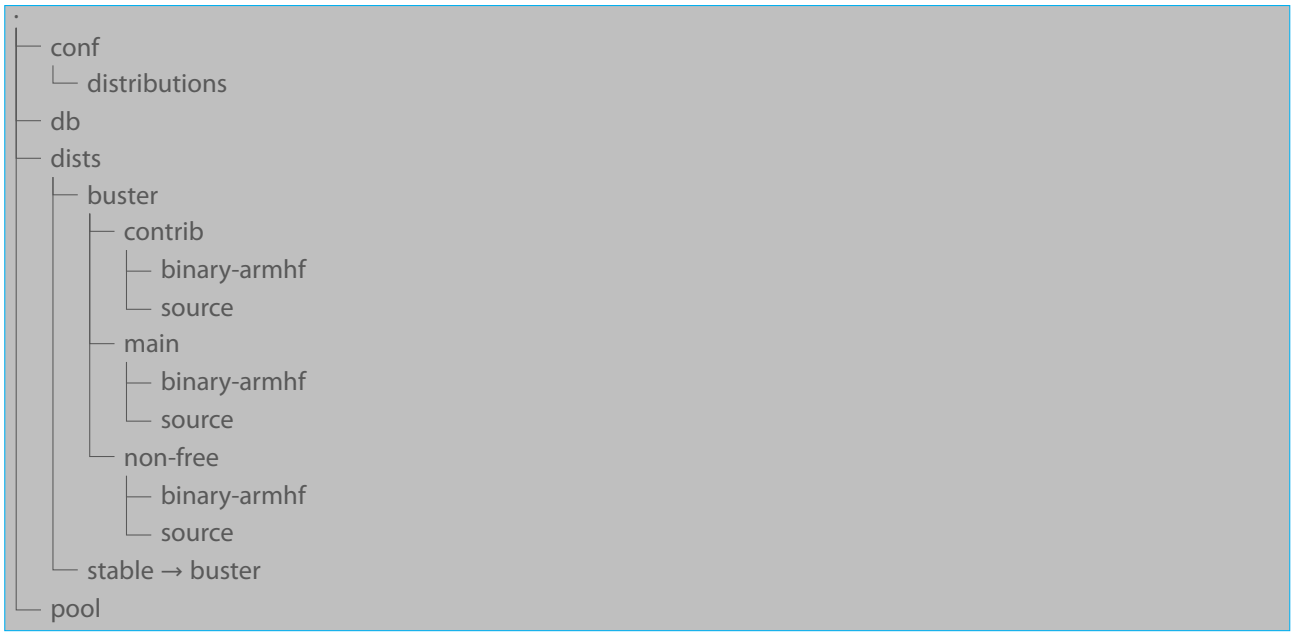

Figure 7.1: *Directory Structure*

The *db* folder is used for internal functions of the repository and is automatically created by *reprepro* during initialization. Inside the *pool* folder, all packages are contained, sorted by their prefixes. Finally, inside *dists* is a directory structure for every distribution, containing file lists with all packages inside the distribution. As it is possible to have packages referenced in multiple distributions, it does not require additional memory to have packages in multiple distributions. To generate the directory structure described above, the file distribution has to be created. The following snippets show how to create the base structure in the exemplary directory */var/www/debian*. Finally, the contents of a minimal *distributions* file are shown.

```
$ mkdir -p /var/www/debian/conf
```

```
Origin: Simple Test Repository
Codename: buster
Suite: stable
Architectures: armhf source
Components: main non-free
```

Listing 7.2: *Example of file 'distributions'*

7.2.3 Managing Debian packages

To add packages to the already created repository, *reprepro* has to be called using one of the following parameters: include, includedeb, includeudeb or includesrc. The following sections briefly describe *include* and *includedeb*. A full documentation is included inside the man-pages of *reprepro*.

Importing binary only packages

The fastest way to add single packages to the repository is to add binary packages without their source information. To include those binary packages, simply call *reprepro* with *includedeb*, as shown in the following example. The repository used in this example is located in */var/www/debian/*

```
$ reprepro -b /var/www/debian/ includedeb buster emboot_1.1-1_armhf.deb
```

Importing package groups with source code

Beside the possibility to import single packages, 'reprepro' offers the possibility to include all packages belonging to one build of a source package, including the package source code (if not already imported). Therefore, a Debian package build process creates the '.changes' file, containing information about created binaries and the source package.

```
$ reprepro -b /var/www/debian/ include buster emboot_1.1-1_armhf.changes
```

This triggers the process of uploading all files referenced inside the *'.changes'* file, adding them to the referenced distribution.

Adding a package to multiple distributions

During the development process it might be useful to add a package to multiple distributions. Normally *reprepro* fails, if the distribution referenced inside the changelog of the package does not match the distribution into which the package should be included to. To overwrite this, the *wrongdistribution* option has to be passed to the *include* parameter as shown in the following snippet.

```
$ reprepro --ignore=wrongdistribution -b /var/www/debian/ include ...
```

Listing packages included in the repositories

To see all packages contained in one distribution, *'reprepro'* provides the *'list'* command. It is very useful to check if everything is setup correctly and if all packages have been included properly.

```
$ reprepro -b /var/www/debian list buster
```

8 Common tasks

8.1 Find packages

There are several ways to find Debian packages. The first one uses the `Debian Packages Search` on the web. The URL is `https://packages.debian.org/index`. There are possiblites to search in package names, descriptions and/or file names which are included in the packages.

The second way uses the package manager 'apt' which is available on a Debian device.

```
root@device# apt search <keyword>
```

Listing 8.1: *Package search using apt*

The keyword is searched as part of the package names / descriptions and the names of the included files.

The command

```
root@device# dpkg -l
```

Listing 8.2: *List installed debian packages*

can be used to find out which packages are installed. Use

```
root@device# dpkg -l | grep -i keyword
```

Listing 8.3: *Search for keyword in installed debian packages*

to find a keyword in the name or description of an installed package.

8.2 Add or remove packages

Mainly two ways exist for adding or removing Debian packages. The first one is the 'apt' way using a remote repository as package source. The second is the offline or local way using 'dpkg'. Both can equally be used to add or remove packages.

```
root@device# apt install examplepackage
root@device# apt remove examplepackage
```

Listing 8.4: *Installation and removal using apt*

```
root@device# dpkg -i examplepackage\_version\_armhf.deb
root@device# dpkg -r examplepackage
```

Listing 8.5: *Installation and removal using dpkg*

8.3 Network configuration

Even if systemd is used, the Debian network configuration can still be done via the well-known `/etc/network/interfaces` file. This results in a permanent network setup. If you want to configure the network temporarily, please refer to chapter 4.3 Network setup. Corresponding to the main network interface, `eth0` is used as an interface in the following examples.

To configure the network with DHCP, place the following information into `/etc/network/interfaces`:

```
allow-hotplug eth0
iface eth0 inet dhcp
```

For a static ip-Setup, the following entries can be used (exemplarily):

```
allow-hotplug eth0
iface eth0 inet static
   address 172.26.1.2
   netmask 255.255.255.0
   gateway 172.26.1.1
```

8.4 OpenAMP demo

The STM32MP157 being an MPU gives the users the flexibility to use the M4 coprocessor in conjunction with the A7 processor. The Debian-based BSP is delivered with OpenAMP demos.

8.4.1 Prerequisite

8.4.1.1 Adding Debian Package

The openamp-demo package is already installed by default. The installed demo files can be found under /opt/openamp-demo/ directory.

If the openamp-demo directory is not available, then follow these instructions to add the demo package on the target. To add the package, update the package-list and then install the package.

```
root@device# apt update
root@device# apt install openamp-demo
```

8.4.2 OpenAMP_FreeRTOS_echo

1. The demo starts the Cortex-M4 processor and initializes OpenAMP Middleware.
2. CM4 creates an rpmsg channel for a virtual UART instance: UARTo.
3. When the FreeRTOS Thread (Idle) is launched, LED_RED Blink periodically while it is waiting for messages from Cortex-A7 processor.
4. When the CA7 receives a message from CM4 through UARTo, the state of LED_GREEN changes.

Steps to start the demo:

```
root@device# cd /opt/openamp-demo/OpenAMP_FreeRTOS_echo
root@device:/opt/openamp-demo/OpenAMP_FreeRTOS_echo# ./fw_cortex_m4.sh start
```

```
[  117.532976] remoteproc remoteproc0: powering up m4
[  117.542347] remoteproc remoteproc0: Booting fw image OpenAMP_FreeRTOS_echo.
    elf, size 2803560
[  117.550681]  remoteproc0#vdev0buffer: assigned reserved memory node vdev0
    buffer@10044000
[  117.558135] virtio_rpmsg_bus virtio0: creating channel rpmsg-tty-channel
    addr 0x0
[  117.563621] virtio_rpmsg_bus virtio0: rpmsg host is online
[  117.565728] rpmsg_tty virtio0.rpmsg-tty-channel.-1.0: new channel: 0x400 ->
    0x0 : ttyRPMSG0
[  117.578310]  remoteproc0#vdev0buffer: registered virtio0 (type 7)
[  117.585366] remoteproc remoteproc0: remote processor m4 is now up
```

Listing 8.6: *Linux Console Messages*

Steps to run the demo:

```
root@device:/opt/openamp-demo/OpenAMP_FreeRTOS_echo# stty -onlcr -echo -F /dev/
    ttyRPMSG0
root@device:/opt/openamp-demo/OpenAMP_FreeRTOS_echo# cat /dev/ttyRPMSG0 &
root@device:/opt/openamp-demo/OpenAMP_FreeRTOS_echo# echo Ping > /dev/ttyRPMSG0
```

Steps to stop the demo:

```
root@device:/opt/openamp-demo/OpenAMP_FreeRTOS_echo# ./fw_cortex_m4.sh stop
```

```
[ 3529.462781] rpmsg_tty virtio0.rpmsg-tty-channel.-1.0: rpmsg tty device 0 is
    removed
[ 3529.973065] remoteproc remoteproc0: warning: remote FW shutdown without ack
[ 3529.978643] remoteproc remoteproc0: stopped remote processor m4
[1]+  Done                    cat /dev/ttyRPMSG0
```

Listing 8.7: *Linux Console Messages*

8.4.3 OpenAMP_raw

1. The demo starts the Cortex-M4 processor and initializes the OpenAMP Middleware.
2. The CM4 creates an rpmsg endpoint and waits for messages from Cortex-A7 Master Core.
3. When the CM4 receives a message on the rpmsg endpoint, it sends the message back to the CA7:
   a) 99 "hello world!" messages are exchanged between the CA7 and CM4
   b) 1 final "goodbye!" message is sent from the CM4 to the CA7.

Steps to start the demo:

```
root@device# cd /opt/openamp-demo/OpenAMP_raw
root@device:/opt/openamp-demo/OpenAMP_raw#./fw_cortex_m4.sh start
```

```
SAMPLE_RPMSG_CLIENT module loaded
[   45.148027] remoteproc remoteproc0: powering up m4
[   45.152045] remoteproc remoteproc0: Booting fw image OpenAMP_raw.elf, size
    216920
[   45.159839]  remoteproc0#vdev0buffer: assigned reserved memory node vdev0
    buffer@10044000
[   45.167646] virtio_rpmsg_bus virtio0: creating channel rpmsg-client-sample
    addr 0x0
[   45.173043] virtio_rpmsg_bus virtio0: rpmsg host is online
[   45.183924]  remoteproc0#vdev0buffer: registered virtio0 (type 7)
[   45.188991] remoteproc remoteproc0: remote processor m4 is now up
[   45.196746] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: new
    channel: 0x400 -> 0x0!
[   45.203906] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming
    msg 1 (src: 0x0)
[   45.215775] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming
    msg 2 (src: 0x0)
........

[   46.026582] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming
    msg 98 (src: 0x0)
[   46.035040] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: incoming
    msg 99 (src: 0x0)
[   46.052006] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: goodbye!
```

Listing 8.8: *Linux Console Messages*

Steps to stop the demo:

```
root@device:/opt/openamp-demo/OpenAMP_raw# ./fw_cortex_m4.sh stop
```

```
Module unloaded
[ 229.057270] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.0: rpmsg
    sample client driver is removed
[  229.574488] remoteproc remoteproc0: warning: remote FW shutdown without ack
[  229.580064] remoteproc remoteproc0: stopped remote processor m4
```

Listing 8.9: *Linux Console Messages*

8.5 Custom application autostart

The Debian based BSP uses `systemd` for managing system tasks, providing nice features for embedded devices, like, for example limiting memory usage for a service, interfacing with kernel Control-groups (CGROUPS). Here we describe shortly how to set-up your own Job and enable it in the system.

For this example we assume that you have stored your application with the name `myapplication` in the folder `/usr/bin`. First create a unit file to define the systemd service. This file has to be stored in the directory `/lib/systemd/system/`. In this example we name it `appname.service`

```
[Unit]
Description=Start myapplication systemd service.

[Service]
Type=simple
ExecStart=/usr/bin/myapplication

[Install]
WantedBy=multi-user.target
```

Listing 8.10: *Unit file for the custom service appname.service*

This defines a simple service. The line `ExecStart` tells the command which is used to start the application. Now copy the unit file to the location `/etc/systemd/system` and give it the correct permissions:

```
root@device# sudo cp -a /lib/systemd/system/appname.service /etc/systemd/system
root@device# sudo chmod 644 /etc/systemd/system/appname.service
```

Now we can test the start by using the commands

```
root@device# sudo systemctl start appname
root@device# sudo systemctl status appname
```

If there is no error reported and the status output tells that the appname.service is active (running) everything is ok. Now we can enable the service so that it runs automatically at boot time.

```
root@device# sudo systemctl enable appname
```

8.6 Peripherals access

One of the most common tasks is to access certain peripherals. The following sections show you how to use the corresponding functions. Please note that information from device specific appendix (page 32ff.) is required.

8.6.1 GPIOs

GPIO access in Linux is possible via the `sysfs` file system if you know the Linux GPIO number.

```
root@device# echo 35 > /sys/class/gpio/export
```

Afterwards it has created it's own subfolder in /sys/class/gpio/, if it is not currently used by any driver. This folder contains the files 'active_low', 'direction', 'edge' and 'value'. The file 'active_low' can be set to 'one' if the values written or read from 'value' ought to be inverted in the manner of a low-active GPIO. But beforehand, the GPIO direction has to be configured via the corresponding 'direction file'. It accepts the values out and in. The following example configures the GPIO as output and sets it to an active state (high).

```
root@device# echo "out" > /sys/class/gpio/gpio35/direction
root@device# echo 1 > /sys/class/gpio/gpio35/value
```

The 'edge' file can be used to configure interrupt-affinity to the GPIO using the poll syscall. However, this document does not cover this in detail.

Info

The gpio access using sysfs is deprecated now. In this case is recommended to use libgpiod when you writting a new application. Further information about libgpiod can be found under `https://git.kernel.org/pub/scm/libs/libgpiod/libgpiod.git/about/`

8.7 Source code acquisition

The source code for U-Boot can be obtained from GitHub:

```
root@device# git clone https://github.com/emtrion/stm32mp1_u-boot.git -b argon_
    v2020.01
```

Listing 8.11: *Get the U-Boot source code*

The source code for the Linux Kernel can also be obtained from GitHub:

```
root@device# git clone https://github.com/emtrion/stm32mp1_linux.git -b argon_v
    5.4
```

Listing 8.12: *Get the Linux Kernel source code*

9 Getting help

9.1 Known Issues

Maybe, there are known issues related to the BSP or Developer Kit. These known issues are documented in our support center (see chapter 9.2)

9.2 Support center

You can access emtrion's support center via the following link: `https://support.emtrion.de`
The support pages have two main sections: hardware and software. On the hardware pages you can find hardware and optionally production related information. On the software pages you can find BSP related information, known issues, frequently asked questions (FAQ) and information regarding new releases.

9.3 Debian documentation and community

Since Debian is a long established Linux distribution you find good and detailed documentation on the internet. To learn more about Debian, please visit the Debian wiki page via `https://wiki.debian.org`

Here you can find an online Debian book containing extensive documentation:

`http://debian-handbook.info/`

9.4 Commercial support

If you require specific support on currently not supported features of emtrion's products, please contact our sales department for an individual quotation.

emtrion GmbH
Am Hasenbiel 6
D-76297 Stutensee
Tel. +49 7244 62694 0
Email (sales department): sales@emtrion.de

9.5 Technical support

If you encounter technical difficulties regarding the officially supported features of our BSP, please send a message to our support team covering the information below. The more detailed your description is, the quicker you receive our feedback, as we can directly forward your request to the correct person internally.

Technical Support: support@emtrion.de

Please include the following Information:

- BSP-variant and version: (e.g. Debian 10, Buster)
- Kernel-version: (e.g. 5.4.76)
- Baseboard and module type (e.g. emSBC-Argon, emSTAMP-Argon)
- Serial number of the affected module (e.g. 10025458)
- Linux distribution on your development PC (e.g. Debian 10)

10 Appendix

10.1 emSTAMP-series

10.1.1 emSTAMP-Argon

10.1.1.1 on emSBC-Argon

The emSBC-Argon baseboard offers several expansion connectors serving PIN-outs from the STM32MP157CAC core-module. The following table gives an overview of their default functions.

Connector	Pin	STM32MP157CAC GPIO	Alternate function
J10	16	GPIO_00	
	15	GPIO_01	ADC1_INP6, ADC1_INN2
	88	GPIO_02	
	8	GPIO_03	TIM8_CH2N
	9	GPIO_04	TIM4_CH2
	10	GPIO_05	TIM5_CH2
	11	GPIO_06	TIM1_CH1
	12	GPIO_07	TIM1_CH1N

Table 10.2: *GPIO translation emSTAMP-Argon on emSBC-Argon*