Yocto

i.MX 6 BSP Manual

Document No.: **L-814e_2**

Release No.: **i.MX 6 PD15.3.x**

Yocto 1.8.1

Edition: May 2016

	EUROPE	NORTH AMERICA	FRANCE
Address:	PHYTEC Messtechnik GmbH Robert-Koch-Str. 39 D-55129 Mainz GERMANY	PHYTEC America LLC 203 Parfitt Way SW Bainbridge Island, WA 98110 USA	PHYTEC France 17, place Saint-Etienne F-72140 Sillé-le-Guillaume FRANCE
Sales:	+49 6131 9221-32 sales@phytec.de	+1 800 278-9913 sales@phytec.com	+33 2 43 29 22 33 info@phytec.fr
Technical Support:	+49 6131 9221-31 support@phytec.de	+1 206 780-9047 support@phytec.com	support@phytec.fr
Fax:	+49 6131 9221-33	+1 206 780-9135	+33 2 43 29 22 34
Web Site:	http://www.phytec.de http://www.phytec.eu	http://www.phytec.com	http://www.phytec.fr

	INDIA	CHINA
Address:	PHYTEC Embedded Pvt. Ltd. #16/9C, 3rd Main, 3rd Floor, 8th Block, Opp. Police Station Koramangala, Bangalore-560095 INDIA	PHYTEC Information Technology (Shenzhen) Co. Ltd. Suite 2611, Floor 26, Anlian Plaza, 4018 Jin Tian Road Futian District, Shenzhen CHINA 518026
Sales:	+91-80-4086 7046/48 sales@phytec.in	+86-755-3395-5875 sales@phytec.cn
Technical Support:	+91-80-4086 7047 support@phytec.in	support@phytec.cn
Fax:		+86-755-3395-5999
Web Site:	http://www.phytec.in	http://www.phytec.cn

2nd Edition May 2016

List of Figures

Conventions, Abbreviations and Acronyms

This i.MX 6 BSP Manual describes the *Linux* BSP accompanying our hardware products. It is based on The *Yocto* Project, extended with hardware support for our products. We give a brief introduction to *Yocto* in general and the specific changes and additions made by Phytec.

Conventions

The conventions used in this manual are as follows:

- Text in *blue italic* indicates a hyperlink within, or external to the document. Click these links to quickly jump to the applicable URL, part, chapter, table, or figure.
- Text in ***bold italic*** indicates an interaction by the user, which is defined on the screen.
- Text in `Consolas` indicates an input by the user, without a premade text or button to click on.
- Text in *italic* indicates proper names of development tools and corresponding controls (windows, tabs, commands, file paths, etc.) used within the development tool, no interaction takes place.
- White Text on black background shows the result of any user interaction (command, program execution, etc.)

⚠	This is a warning. It helps you to avoid annoying problems.
🦊	You can find useful supplementary information about the topic.

1 Introduction to Yocto

Please read the *Yocto* Reference Manual (L-813e_x) for a better understanding of *Yocto* and this BSP.

2 Introduction to the BSP

2.1 Supported Hardware

For information which boards and modules are supported by the release of Phytec's i.MX 6 unified BSP described herein, visit our web page at *http://www.phytec.de/produkte/software/yocto/phytec-unified-yocto-bsp-releases/*.

Click the corresponding BSP release and look for the ordering number of your module in the column "Hardware Article Number". Now you can find the correct machine name in the corresponding cell under "Machine Name".

3 Building the BSP

This section will guide you through the general build process of the unified i.MX 6 BSP using the phyLinux script. If you want to use our software without phyLinux and the *Repo* tool managed environment instead, you can find all *Git* repositories on

git://git.phytec.de

Used *barebox* repository:

git://git.phytec.de/barebox

Our *barebox* version is based on the *barebox* mainline and adds only a few patches which will be sent upstream in future. Used *Linux* kernel repository:

git://git.phytec.de/linux-mainline

Our i.MX 6 kernel is based on the *Linux* stable kernel. The stable kernel repository can be found at:

git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git

To find out which tag is used for a specific board, have a look at:

meta-phytec/recipes-bsp/barebox/barebox_*.bb
meta-phytec/recipes-kernel/linux/linux-mainline_*.bb

3.1 Get the BSP

- Create a fresh project directory, e.g.

```
host$ mkdir ~/yocto
```

- Download and run the phyLinux script

```
host$ cd ~/yocto
host$ wget ftp://ftp.phytec.de/pub/Software/Linux/Yocto/Tools/phyLinux
host$ chmod  +x  phyLinux
host$./phyLinux init
```

3.2 Basic Set-Up

There are a few important steps which have to be done, before the main build process.

- Setting up the host, see *Yocto* Reference Manual "Setting up the Host"
- Setting up the *Git* configuration, see *Yocto* Reference Manual "*Git* Configuration"

3.3 Finding the right Software Platform

The i.MX 6 BSP is a unified BSP, which means it supports a set of different Phytec carrier boards (CB) with different Systems on Module (SOMs). Sometimes it is not easy to find the right software for your Phytec board. So if you need to figure out the corresponding machine name of your board, have a look at *http://www.phytec.de/produkte/software/yocto/phytec-unified-yocto-bsp-releases/* and click on the corresponding BSP release, or refer to the files in:

meta-phytec/conf/machine/.conf*

where you can find the platform name to the corresponding product IDs. All this information is also displayed by the phyLinux script.

E.g.: *phyflex-imx6-2.conf* machine configuration file:

#@TYPE: Machine
#@NAME: phyflex-imx6-2
#@DESCRIPTION: PFL-A-02-13237E0.A1/PBA-B-01 (i.MX6 Quad, 1GB RAM on one bank, 16MB NOR)

Machine *phyflex-imx6-2.conf* represents the PBA-B-01 (Kit) CB with the PFL-A-02-13237E0.A1 SOM.

3.4 Selecting a Software Platform

- To select the correct SoC, BSP version and platform call:
  ```
  host$ ./phyLinux init
  ```

It is also possible to pass this information directly using command line parameters:

```
host$ ./phyLinux init -p imx6 -r PD15.3.0
```

Please read section "Initialization" in the *Yocto* Reference Manual for more information.

3.5 Starting the Build Process

Refer to *Yocto* Reference Manual "Start the Build".

3.6 BSP Images

All images generated by *Bitbake* are deployed to *yocto/build/deploy/images/<machine>*.

The following list shows for example all files generated for the i.MX 6 SoC, *phyflex-imx6-2* machine:

- *Barebox*: barebox.bin
- *Barebox* configuration: barebox-defconfig
- Kernel: zImage
- Kernel device tree file: zImage-imx6q-phytec-pbab01.dtb
- Kernel configuration: zImage.config
- Root filesystem: phytec-qt5demo-image-phyflex-imx6-2.tar.gz, phytec-qt5demo-image-phyflex-imx6-2.ubifs, phytec-qt5demo-image-phyflex-imx6-2.ext4
- SD card image: phytec-qt5demo-image-phyflex-imx6-2.sdcard

4 Booting the System

The default boot source for the i.MX 6 modules like phyCORE-i.MX 6, phyFLEX-i.MX 6 and phyCARD-i.MX 6is the NAND Flash. The easiest way to get started with your freshly created images, is writing them to an SD card and setting the boot configuration accordingly. For information how to set the correct boot configuration refer to the corresponding hardware manual for your Phytec board.

4.1 Booting from NAND Flash

NAND is the default boot source. To update the software of the NAND Flash see *5 "Updating the Software"*.

4.2 Booting from SD Card

Booting from SD card is useful in several situations, e.g. if the board does not start any more due to a damaged bootloader. To boot from SD card the SD card must be formatted in a special way, because the i.MX 6 does not use file systems. Instead it is hard coded at which sectors of the SD card the i.MX 6 expects the bootloader.

There are two ways to create a bootable SD card. You can either use:

- a single prebuild SD card image, or
- the four individual images (*barebox*-, kernel- and device tree image, and root filesystem)

4.2.1 Using a single, prebuild SD Card Image

The first possibility is to use the SD card image build by *Bitbake*, a tool integrated in *Yocto*. This image has the ending *.sdcard and can be found under *build/deploy/images/<MACHINE>/<IMAGENAME>-<MACHINE>.sdcard*. It contains all BSP files in correctly formatted partitions already and can be easily copied to the SD card using *dd*.

You can also find images on our FTP server *ftp://ftp.phytec.de/pub/Software/Linux/Yocto/*.

⚠	Be very careful when selecting the right drive as all files on the selected device will be erased! Selecting the wrong drive can erase your hard drive!

- Get the correct device name (<your_device>) with `sudo fdisk -l` or from the last outputs of `dmesg` after inserting the SD card.

- Now use the following command to create your bootable SD card

  ```
  host$ sudo dd if=<IMAGENAME>-<MACHINE>.sdcard of=/dev/<your_device>
                                                      bs=1MB conv=fsync
  ```

  where <your_device> could be for example "sde", depending on your system.

The parameter *conv=fsync* forces a sync operation on the device before *dd* returns. This ensures that all blocks are written to the SD card and are not still in memory.

4.2.2 Using four individual Images (*barebox*-, kernel- and device tree image, and root filesystem)

Instead of using the single prebuild SD card image, you can also use the *barebox*-, kernel- and device tree image together with the root filesystem separately to create a bootable SD card manually.

For this method a new card must be setup with 2 partitions and 8 MB of free space at the beginning of the card. Use the following procedure with *fdisk* under Linux:

- Create a new FAT partition with partition id C. When creating the new partition you must leave 8 MB of free space at the beginning of the card. When you go through the process of creating a new partition, *fdisk* lets you specify where the first sector starts. During this process *fdisk* will tell you where the first sector on the disk begins. If, for example, the first sector begins at 1000, and each sector is 512 bytes, then 8 MB / 512 bytes = 16384 sectors, thus your first sector should begin at 17384 to leave 8 MB of free space. The size of the FAT partition needs only be big enough to hold the zImage which is only a few megabytes. To be safe we recommend a size of 64 MB.
- Create a new Linux partition with partition id 83. Make sure you start this partition after the last sector of partition 1! By default *fdisk* will try to use the first partition available on the disk, which in this example is 1000. However, this is our reserved space! You must use the remaining portion of the card for this partition.
- Write the new partition to the SD card and exit *fdisk*.

Example:

- Type:

  ```
  host$ sudo fdisk -l /dev/sdc
  ```

You will receive:

```
Disk /dev/sdc: 4025 MB, 4025483264 bytes
4 heads, 32 sectors/track, 61424 cylinders, total 7862272 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x26edf128

  Device Boot    Start      End     Blocks    Id     System
/dev/sdc1         8192    24575       8192     c     W95 FAT32 (LBA)
/dev/sdc2        24576   655359     315392    83     Linux
```

- Remove and reinsert the card. Otherwise *Linux* will not recognize the new partitions created in the previous step.
- Create a file system on the partitions with (replace "sde" with your device):
  ```
  host$ sudo mkfs.vfat /dev/sde1
  host$ sudo mkfs.ext4 -L "rootfs" /dev/sde2
  ```

Now the images need to be copied to the SD card.

- Write the bootloader in front of the first partition (replace "sde" with your device):
  ```
  host$ sudo dd if=barebox.bin of=/dev/sde bs=512 skip=2 seek=2
                                                        conv=fsync
  ```
- Mount the first partition (*vfat*) and copy the *linuximage* and *oftree* file to it:
  ```
  host$ sudo mount /dev/sd<X>1 /mnt
  ```

Make sure that the images are named as mentioned before, as the bootloader expects them exactly like that.

- In case you want to boot the whole *Linux* from SD card, also mount the *ext4* partition.
- Then untar <IMAGENAME>-<MACHINE>.tar.gz rootfs image to it:
  ```
  host$ sudo mount /dev/sd<X>2 /media
  host$ sudo tar zxf <IMAGENAME>-<MACHINE>.tar.gz -C /media/
  host$ sudo umount /media
  ```

4.3 Booting from USB OTG (Serial Downloader)

The i.MX 6 ROM code is capable of downloading a bootloader from the USB OTG interface ("Serial Downloader" in the i.MX 6 Reference Manual). This is useful for a last resort recovery of a broken bootloader, or for rapid *barebox* development.

First you have to compile the program *imx-usb-loader* in the barebox source directory. You can use any current mainline barebox version and you have to.

- First load the default configuration with:
  ```
  host$ make ARCH=arm imx_v7_defconfig
  ```
- In order to activate *imx-usb-loader* type
  ```
  host$ make ARCH=arm menuconfig
  ```
  and enable *System Type ---> i.MX specific settings ---> compile imx-usb-loader*
- Now, compile the *imx-usb-loader:*
  ```
  host$ make ARCH=arm CROSS_COMPILE=<prefix of your arm cross toolchain>
                                                              scripts/imx/
  ```

Now the tool is in *scripts/imx/imx-usb-loader*.

To load the bootloader to the module execute the following sequence.

- Connect your target to your host PC via USB OTG.
- Check the boot configuration of your board and ensure that the ROM code enters *Serial Downloader* mode (see the corresponding section in the Hardware Manual of your board).
- As the boot configuration is not read during a soft reset, perform a power cycle.
- Finally execute the *imx-usb-loader*, e.g.
  ```
  host$ sudo scripts/imx/imx-usb-loader images/barebox-phytec-pbab01-
                                                           1gib-1bank.img
  ```

After that you should see the *barebox* boot messages on the serial console.

4.4 Booting from SPI NOR Flash

Most of the i.MX 6 modules (e.g. phyCORE-i.MX 6 and phyFLEX-i.MX 6) are optionally equipped with SPI NOR Flash. To boot from SPI Flash, select the correct boot mode as described in the hardware manual of your Phytec board. The SPI Flash is usually quite small so that only the bootloader, the *Linux* kernel and the device tree can be stored. The root filesystem is taken from NAND Flash by default. How to flash the SPI NOR is described in section *5 "Updating the Software"*.

4.5 Booting the Kernel from Network

In this case booting from network means to load the kernel over TFTP and the root filesystem over NFS. The bootloader itself must already be loaded from any other boot device available.

4.5.1 Development Host Preparations

On the development host a TFTP server must be installed and configured. The following tools will be needed to boot the Kernel from Ethernet:

1. a TFTP server and
2. a tool for starting/stopping a service.

- For *Ubuntu* install:

```
host$ sudo apt-get install tftpd-hpa xinetd
```

After the installation of the packages you have to configure the TFTP server.

Set up for the TFTP server:

- Edit */etc/xinetd.d/tftp*.

```
service tftp
{
     protocol = udp
     port = 69
     socket_type = dgram
     wait = yes
     user = root
     server = /usr/sbin/in.tftpd
     server_args = -s /tftpboot
     disable = no
}
```

- Create a directory to store the TFTP files:

```
host$ sudo mkdir /tftpboot
host$ sudo chmod -R 777 /tftpboot
host$ sudo chown -R nobody /tftpboot
```

- Configure a static IP address for the appropriate interface:

```
host$ ifconfig eth1
```

You will receive:

```
eth1      Link encap:Ethernet  HWaddr 00:11:6b:98:e3:47
          inet addr:192.168.3.10  Bcast:192.168.3.255
                                       Mask:255.255.255.0
```

- Restart the services to pick-up the configuration changes:

```
host$ sudo service tftpd-hpa restart
```

- Now connect the first port of the board to your host system, configure the board to network boot and start it.

Usually TFTP servers are using the */tftpboot* directory to fetch files from. If you built your own images, please copy them from the BSP's build directory to there.

We also need a network connection between the embedded board and the TFTP server. The server should be set to IP 192.168.3.10 and netmask 255.255.255.0.

After the installation of the TFTP server, an NFS server needs to be installed, too. The NFS server is not restricted to a certain file system location, so all we have to do on most distributions is to modify the file */etc/exports* and export our root filesystem to the embedded network. In this example file the whole work directory is exported, and the "lab network" address of the development host is 192.168.3.10, so the IP addresses have to be adapted to the local needs:

```
/home/<user>/<rootfspath>
192.168.3.10/255.255.255.0(rw,no_root_squash,sync,no_subtree_check)
```

Where <*user*> must be replaced with your home directory name. The <*rootfspath*> can be set to a folder which contains a rootfs *tar.gz* image extracted with *sudo*.

4.5.2 Preparations on the Embedded Board

- To find out the Ethernet settings in the bootloader of the target, type:

```
bootloader$ ifup eth0
bootloader$ devinfo eth0
```

With your development host set to IP 192.168.3.10 and netmask 255.255.255.0, the target should return:

```
ipaddr=192.168.3.11
netmask=255.255.255.0
gateway=192.168.3.10
serverip=192.168.3.10
```

- If you need to change something, type :

```
bootloader$ edit /env/network/eth0
```

Here you can also change the IP address to DHCP instead of using a static one.

- Just configure: `ip=dhcp`

- Edit the settings if necessary and save them by leaving the editor with ***CTRL+D***.
- Type *saveenv* if you made any changes.
- Set up the paths for TFTP and NFS in the file */env/boot/net*.

4.5.3 Booting the Embedded Board

- To boot from network call

  ```
  bootloader$ boot net
  ```

  or restart the board and press ***m*** to stop autoboot.

  You will get a menu:

  ```
  Main menu
  1: Boot: nand (UBI)
  2: Detect bootsources
  3: Settings
  4: Save environment
  5: Shell
  6: Reset
  ```

- Press ***2*** and then ***Enter*** which opens a second menu:

  ```
  boot
  1: mmc
  2: nand
  3: spi
  4: net
  5: back
  ```

- Press ***4*** and then ***Enter*** in order to boot the board from network.

4.6 Custom Boot Setup

You may have custom boot requirements that are not covered by the four available boot files (*nand*, *net*, *mmc*, *spi*). If this is the case you can create your own custom boot entry specifying the kernel and root filesystem location.

- First create your own boot entry in *barebox*, for example named "custom":

  ```
  bootloader$ edit /env/boot/custom
  ```

- Use the following template to specify the location of the *Linux* kernel and root filesystem.

  ```
  #!/bin/sh

  global.bootm.image="<kernel_loc_bootm.image>"
  global.bootm.oftree="<dts_loc_bootm.oftree>"

  nfsroot="<nfs_root_path>"
  bootargs-ip
  /env/config-expansions

  global.linux.bootargs.dyn.root="<rootfs_loc_dyn.root>"
  ```

Please note that the text in <> such as <*kernel_loc_bootm.image*>, <*rootfs_loc_dyn.root*>, and <*nfs_root_path*> are intended to be replaced with user specific values as described in the following.

- <kernel_loc_bootm.image> specifies the location of the *Linux* kernel image and can be:

```
/dev/nand0.kernel.bb - To boot the Linux kernel from NAND
/mnt/tftp/zImage - To boot the Linux kernel via TFTP
/mnt/mmc/zImage - To boot the Linux kernel from SD/MMC card
```

- <dts_loc_bootm.oftree> specifies the location of the device tree binary and can be:

```
/dev/nand0.oftree.bb - To boot the device tree binary from NAND
/mnt/tftp/oftree - To boot the device tree binary via TFTP
/mnt/mmc/oftree - To boot the device tree binary from SD/MMC card
```

- <rootfs_loc_dyn.root> specifies the location of the root filesystem and can be:

```
root=ubi0:root ubi.mtd=root rootfstype=ubifs - To mount the root
                                               filesystem from NAND
root=/dev/nfs nfsroot=$nfsroot,vers=3,udp rw consoleblank=0
                       - To mount the root filesystem via NFS
root=/dev/mmcblk0p2 rootwait - To mount the root filesystem from
                               SD/MMC card
```

- <nfs_root_path> is only required if mounting the root filesystem from NFS is desired. Replace with the following:

```
nfsroot="/home/${global.user}/nfsroot/${global.hostname}"
```

- After completing the modifications exit the editor using ***CTRL+D*** and save the environment:

```
bootloader$ saveenv
```

- To run your custom boot entry from the *barebox* shell enter:

```
bootloader$ boot custom
```

If you want to configure the bootloader for booting always from "custom", you need to create the */env/nv/boot.default* file. Here you can just insert "custom" and save it. Otherwise the boot source and boot order is defined in */env/init/bootsource*.

5 Updating the Software

In this chapter we explain how to update the images using the *barebox* bootloader into NAND and SPI NOR Flash.

5.1 Updating from Network

i.MX 6 boards that have an Ethernet connector can be updated over network. Be sure to set up the development host correctly. The IP needs to be set to 192.168.3.10, the netmask to 255.255.255.0, and a TFTP server needs to be available.

- Boot the system using any boot device available.
- Press any key to stop autoboot, then type:

  ```
  bootloader$ ifup eth0
  bootloader$ devinfo eth0
  ```

The Ethernet interfaces should be configured like this:

```
ipaddr=192.168.3.11
netmask=255.255.255.0
gateway=192.168.3.10
serverip=192.168.3.10
```

If a DHCP server is available, it is also possible to set:

```
ip=dhcp
```

If you need to change something:

- Type:

  ```
  bootloader$ edit /env/network/eth0
  ```

- Edit the settings, save them by leaving the editor with ***CTRL+D*** and type:

  ```
  bootloader$ saveenv
  ```

- Reboot the board.

5.1.1 Updating NAND Flash from Network

To update the bootloader you may use the *barebox_update* command. This provides a handler which automatically erases and flashes two copies of the *barebox* image into the NAND Flash. This makes the system more robust against ECC issues. If one block is corrupted the ROM loader does use the next block.This handler also creates an FCB table in the NAND. The FCB table is needed from the ROM loader to boot from NAND.

- Type:

  ```
  bootloader$ barebox_update -t nand /mnt/tftp/barebox.bin
  ```

On startup the TFTP server is automatically mounted to */mnt/tftp*. So copying an image from TFTP to flash can be done in one step. Do not get confused when doing an *ls* on the */mnt/tftp* folder. The TFTP protocol does not support anything like *ls* so the folder will appear to be empty.

We recommend to also erase the environment of the old *barebox*. Otherwise the new *barebox* would use the old environment.

- First erase the old environment with:

```
bootloader$ erase /dev/nand0.barebox-environment.bb
```

After erasing the environment, you have to reset your board. Otherwise the *barebox* still uses the old environment.

- To reset your board in order to get the new *barebox* running type:

```
bootloader$ reset
```

- Now get the *Linux* kernel and *oftree* from your TFTP server and store it also into the NAND Flash with:

```
bootloader$ erase /dev/nand0.kernel.bb
bootloader$ cp /mnt/tftp/zImage /dev/nand0.kernel.bb
bootloader$ erase /dev/nand0.oftree.bb
bootloader$ cp /mnt/tftp/oftree /dev/nand0.oftree.bb
```

- For flashing *Linux*'s root filesystem to NAND, please use:

```
bootloader$ ubiformat /dev/nand0.root
bootloader$ ubiattach /dev/nand0.root
bootloader$ ubimkvol /dev/nand0.root.ubi root 0
bootloader$ cp -v /mnt/tftp/root.ubifs /dev/nand0.root.ubi.root
```

You should not flash *Linux*'s root filesystem to NAND with an UBI image the same way as you did with the *Linux* kernel. UBIFS keeps erase counters within the NAND in order to be able to balance write cycles equally over all NAND sectors. So if there is already an UBIFS on your module and you want to replace it by a new one, using *erase* and *cp* will also erase these erase counters which should be avoided.

5.1.2 Updating SPI NOR Flash from Network

- To update the bootloader in the SPI NOR Flash from network do the following:

```
bootloader$ erase /dev/m25p0.barebox
bootloader$ cp /mnt/tftp/barebox.bin /dev/m25p0.barebox
```

We recommend to also erase the environment of the old *barebox*. Otherwise the new *barebox* would use the old environment.

- First erase the old environment with:

```
bootloader$  erase /dev/m25p0.barebox-environment
```

After erasing the environment, you have to reset your board. Otherwise the *barebox* still uses the old environment.

- To reset your board in order to get the new barebox running type:

```
bootloader$ reset
```

- The kernel and *oftree* are then updated with the regular *erase* and *cp* commands:

```
bootloader$ erase /dev/m25p0.kernel
bootloader$ cp /mnt/tftp/zImage /dev/m25p0.kernel
bootloader$ erase /dev/m25p0.oftree
bootloader$ cp /mnt/tftp/oftree /dev/m25p0.oftree
```

The root filesystem is to big to fit into the SPI NOR Flash. So the default configuration when booting from SPI is to take the root filesystem from NAND Flash.

5.1.3 Updating eMMC from Network

From a high level point of view a eMMC device is like an SD card. Therefore it is possible to flash the image <*name*>*.sdcard* from the *Yocto* build system directly to the eMMC. The image contains the bootloader, kernel, device trees and root filesystem.

- To flash the SD card image, use:

```
bootloader$ detect mmc3
bootloader$ cp -v /mnt/tftp/<name>.sdcard /dev/mmc3
```

After flashing the SD card image the root filesystem does not use all available space on the device. To enlarge the file system see section "*Resize the ext4 Root* Filesystem"

- If you only want to update the bootloader on the eMMC, use

```
bootloader$ barebox_update -l
bootloader$ barebox_update -t mmc3 /mnt/tftp/barebox.bin
```

  to find the correct update target

We recommend to also erase the environment of the old *barebox*. Otherwise the new *barebox* would use the old environment.

- Type:

```
bootloader$ cp /dev/zero /dev/mmc3.barebox-environment
```

- You should check the *boot* configuration of the eMMC device with:

```
bootloader$ detect mmc3
bootloader$ devinfo mmc3
```

If you used the *cp* command as described in the previous step the value of *boot* should be *disabled*, or *user*:

```
[...]
Parameters:
 boot: disabled ("disabled", "boot0", "boot1", "user")
 probe: 0
```

If the values are not *disabled*, or *user* the i.MX 6 ROM code cannot load the bootloader from the i.MX 6 device.

Now you have to reset your board. Otherwise the *barebox* still uses the old environment.

- To reset your board in order to get the new *barebox* running type:

```
bootloader$ reset
```

- To update the kernel and device tree, use:

```
bootloader$ ls /mnt/emmc
bootloader$ cp /mnt/tftp/zImage /mnt/emmc/
bootloader$ cp /mnt/tftp/oftree /mnt/emmc/
```

5.2 Updating from SD Card

To update an i.MX 6 board from SD card the SD card used for updating must be mounted after the board is powered and the boot sequence is stopped on the bootloader prompt. If the board is booted from SD card the card is already mounted automatically under */mnt/mmc/*.

The kernel and device tree are already in the first partition of the SD card to allow booting from the SD card. In order to use the SD card also for updating the *barebox* in the NAND, or NOR Flash, you have to copy the *barebox.bin* to the SD card on your host PC, too.

- The actual bootloader on the SD card is not in a partition. It is located before the first partition after the partition table.
- You cannot update the root filesystem, because the first partition is too small for it.

5.2.1 Updating NAND Flash from SD Card

To update the images on the NAND Flash from SD card basically the same commands as updating from TFTP are used with just the path parameters adapted.

- Type:

```
bootloader$ barebox_update -t nand /mnt/mmc/barebox.bin
bootloader$ erase /dev/nand0.barebox-environment.bb
bootloader$ reset
bootloader$ erase /dev/nand0.kernel.bb
bootloader$ cp /mnt/mmc/zImage /dev/nand0.kernel.bb
bootloader$ erase /dev/nand0.oftree.bb
bootloader$ cp /mnt/mmc/oftree /dev/nand0.oftree.bb
```

- Change the boot configuration of your board to NAND boot if necessary, and reset your board.

5.2.2 Updating SPI Flash from SD Card

To update the images on the SPI Flash from SD card basically the same commands as updating from TFTP are used with just the path parameters adapted.

- Type:

```
bootloader$ erase /dev/m25p0.barebox
bootloader$ cp /mnt/mmc/barebox.bin /dev/m25p0.barebox

bootloader$ erase /dev/m25p0.kernel
bootloader$ cp /mnt/mmc/zImage /dev/m25p0.kernel
bootloader$ erase /dev/m25p0.oftree
bootloader$ cp /mnt/mmc/oftree /dev/m25p0.oftree
```

- Optionally, you can erase the environment to use the *barebox* default environment:

```
bootloader$ erase /dev/m25p0.barebox-environment
```

- Change the boot configuration of your board to NOR boot if necessary, and reset your board.

6 Device Tree (DT)

6.1 Introduction

The following text describes briefly the Device Tree and can be found in the *Linux* kernel (*linux/Documentation/devicetree/usage-model.txt*).

"The "Open Firmware Device Tree", or simply Device Tree (DT), is a data structure and language for describing hardware. More specifically, it is a description of hardware that is readable by an operating system so that the operating system doesn't need to hard code details of the machine.
Structurally, the DT is a tree, or acyclic graph with named nodes, and nodes may have an arbitrary number of named properties encapsulating arbitrary data. A mechanism also exists to create arbitrary links from one node to another outside of the natural tree structure.
Conceptually, a common set of usage conventions, called 'bindings', is defined for how data should appear in the tree to describe typical hardware characteristics including data busses, interrupt lines, GPIO connections, and peripheral devices."

The kernel is a really good source for a DT introduction. An overview of the device tree data format can be found on the device tree usage page at devicetree.org:
http://devicetree.org/Device_Tree_Usage

6.2 Phytec i.MX 6 BSP Device Tree Concept

The following sections explain some rules we have defined on how to set up device trees in first place for our i.MX 6 SoC based boards.

The device tree files are divided roughly into three layers:
- the SoC layer
- the module layer, and
- the baseboard layer

This resembles the physical properties of the hardware. E.g., the same phyFLEX-i.MX 6 module can be used on the phyFLEX Carrier Board, the phyBOARD-Alcor i.MX 6 and the phyBOARD-Subra i.MX 6.

In each layer are multiple device tree include files.

Furthermore common functionality within a layer is factored out into an extra device tree include file. E.g., module settings which are identical for Solo/DualLite and Dual/Quad controllers are placed into file *imx6qdl-phytec-pfla02.dtsi*. This file is included together with the specific Solo/DualLite module file *imx6dl-phytec-pfla02.dtsi* and the module file *imx6q-phytec-pfla02.dtsi* specific to the Dual/Quad controller.

An overview of the device tree hierarchy for all Phytec i.MX 6 platforms is presented in the figures below.

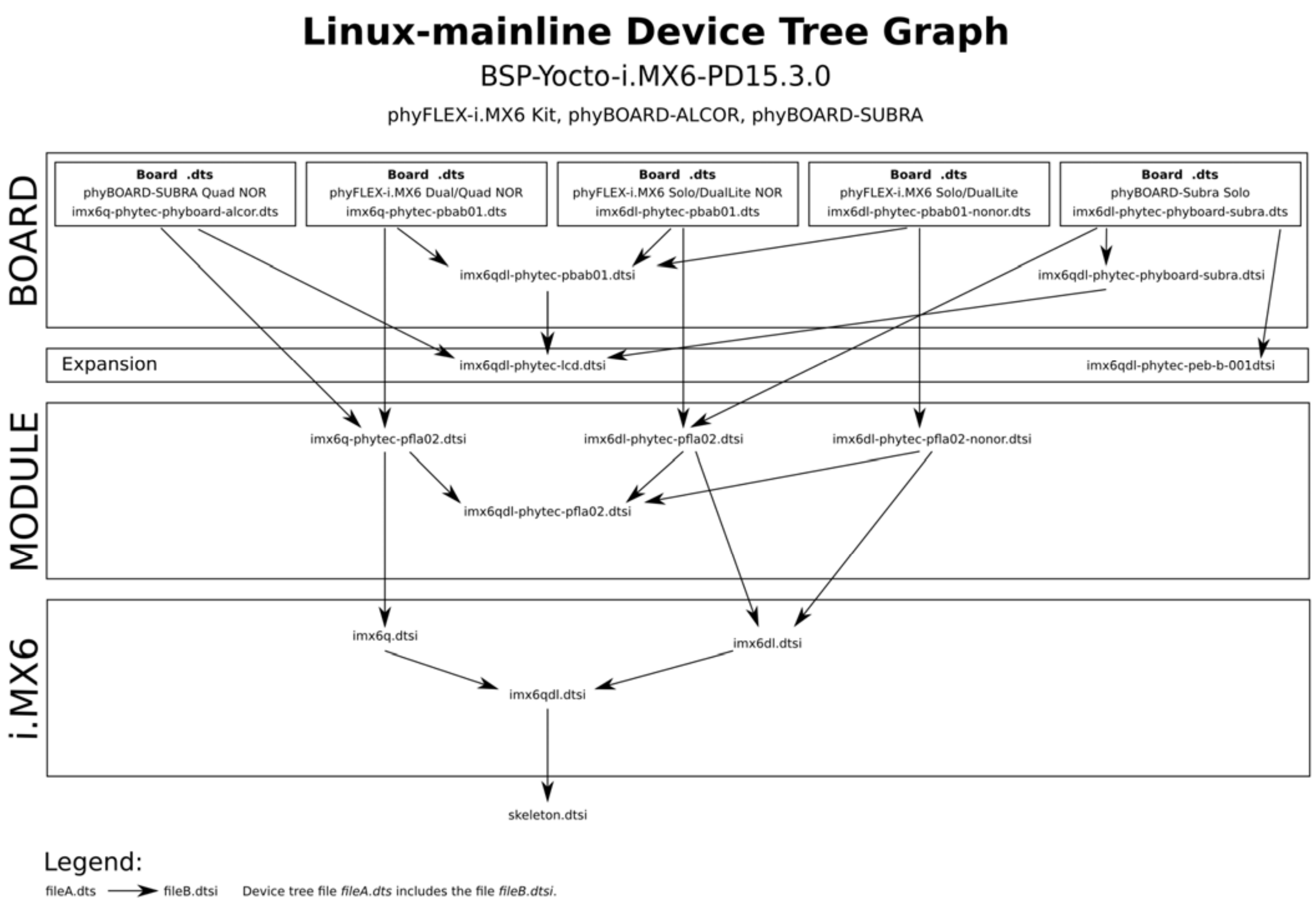

Linux-mainline Device Tree Graph
BSP-Yocto-i.MX6-PD15.3.0
phyFLEX-i.MX6 Kit, phyBOARD-ALCOR, phyBOARD-SUBRA
BOARD
Board .dts
phyBOARD-SUBRA Quad NOR
imx6q-phytec-phyboard-alcor.dts
Board .dts
phyFLEX-i.MX6 Dual/Quad NOR
imx6q-phytec-pbab01.dts
Board .dts
phyFLEX-i.MX6 Solo/DualLite NOR
imx6dl-phytec-pbab01.dts
Board .dts
phyFLEX-i.MX6 Solo/DualLite
imx6dl-phytec-pbab01-nonor.dts
Board .dts
phyBOARD-Subra Solo
imx6dl-phytec-phyboard-subra.dts
imx6qdl-phytec-pbab01.dtsi
imx6qdl-phytec-phyboard-subra.dtsi
Expansion
imx6qdl-phytec-lcd.dtsi
imx6qdl-phytec-peb-b-001dtsi
MODULE
imx6q-phytec-pfla02.dtsi
imx6dl-phytec-pfla02.dtsi
imx6dl-phytec-pfla02-nonor.dtsi
imx6qdl-phytec-pfla02.dtsi
i.MX6
imx6q.dtsi
imx6dl.dtsi
imx6qdl.dtsi
skeleton.dtsi
Legend:
fileA.dts
fileB.dtsi
Device tree file fileA.dts includes the file fileB.dtsi.

Figure 1: Device Tree BSP-Yocto-i.MX6-PD15.3.0 phyFLEX

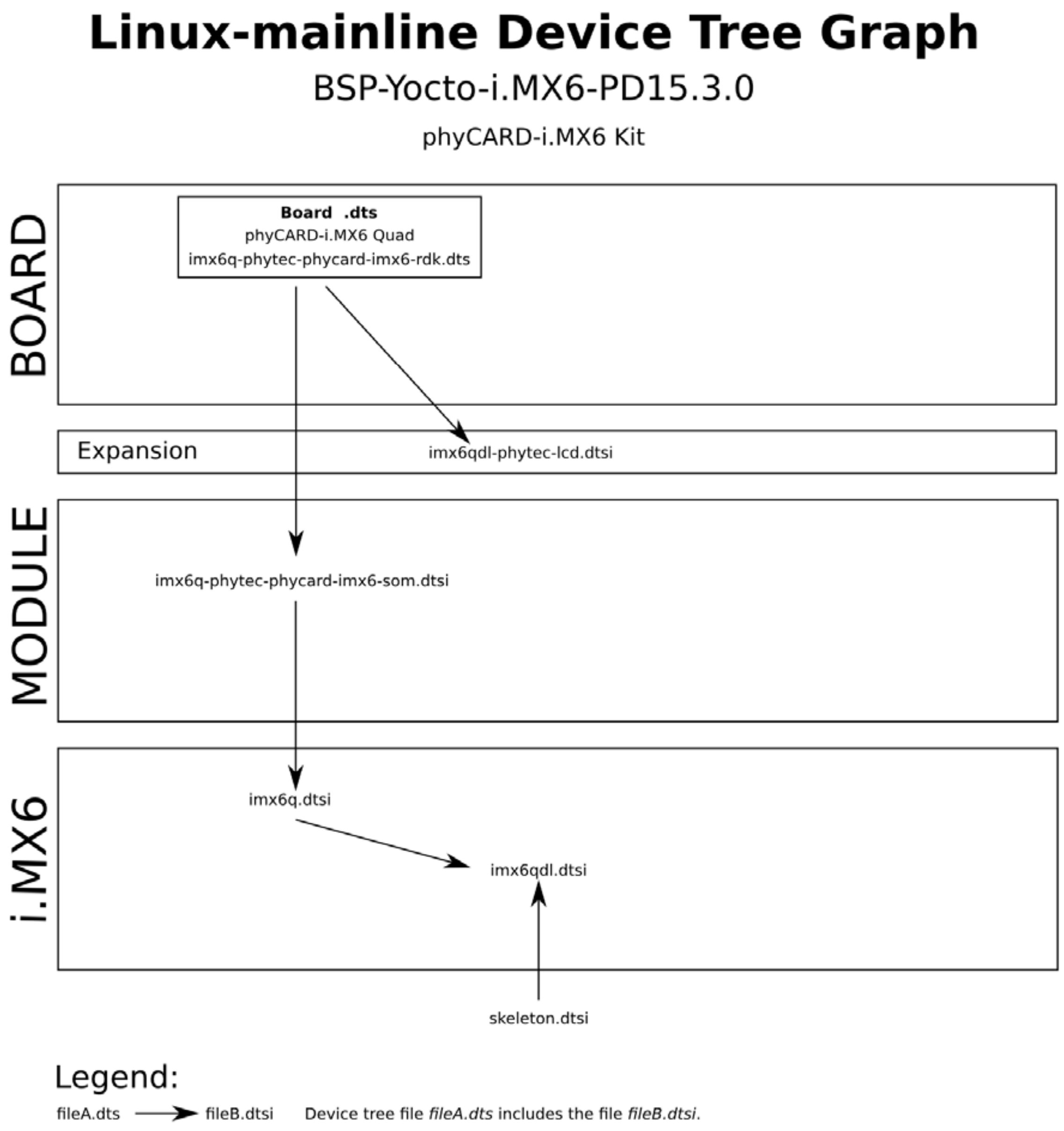


Figure 2: Device Tree BSP-Yocto-i.MX6-PD15.3.0 phyCARD

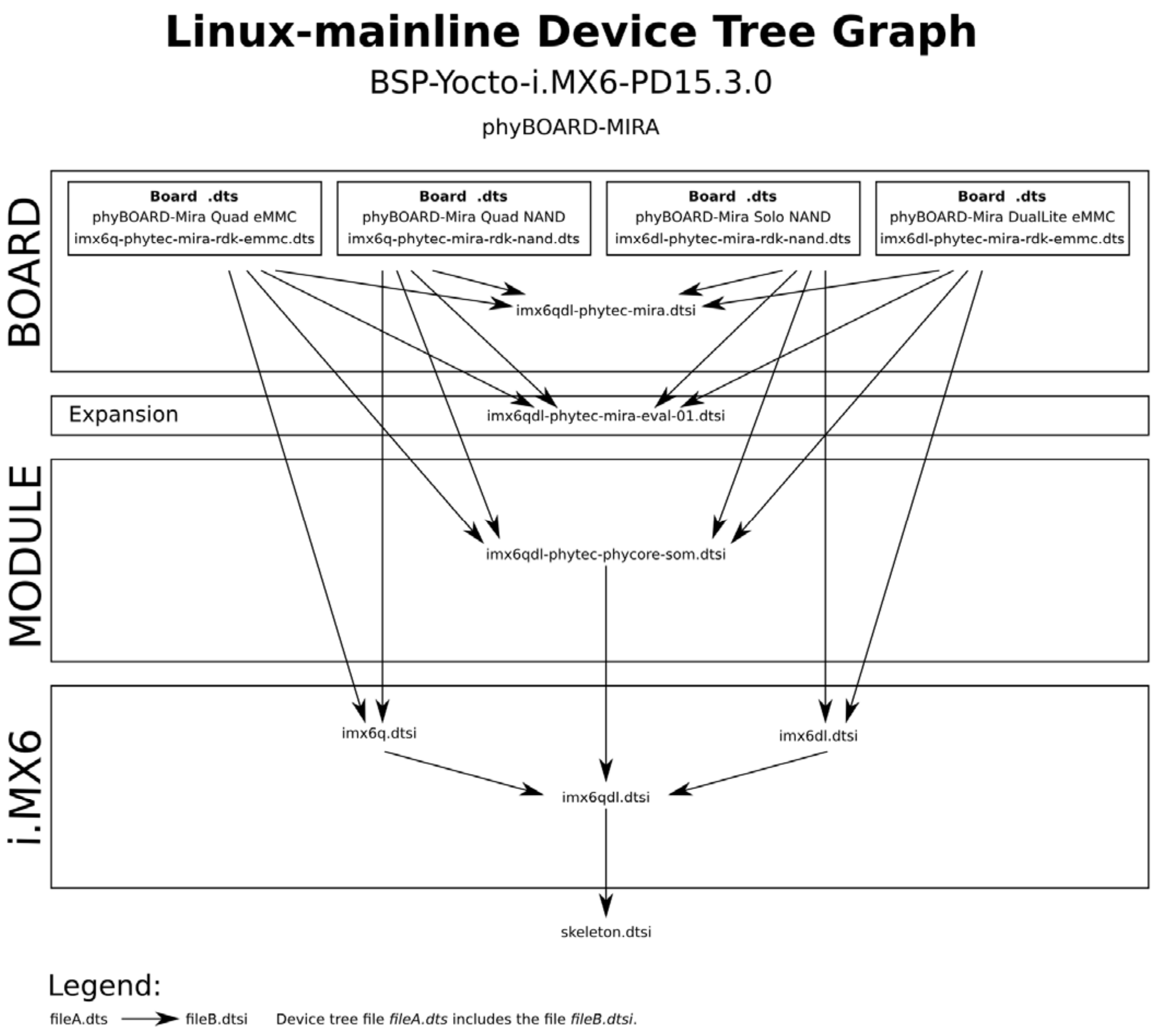

Linux-mainline Device Tree Graph
BSP-Yocto-i.MX6-PD15.3.0
phyBOARD-MIRA
BOARD
Board .dts
phyBOARD-Mira Quad eMMC
imx6q-phytec-mira-rdk-emmc.dts
Board .dts
phyBOARD-Mira Quad NAND
imx6q-phytec-mira-rdk-nand.dts
Board .dts
phyBOARD-Mira Solo NAND
imx6dl-phytec-mira-rdk-nand.dts
Board .dts
phyBOARD-Mira DualLite eMMC
imx6dl-phytec-mira-rdk-emmc.dts
imx6qdl-phytec-mira.dtsi
Expansion
imx6qdl-phytec-mira-eval-01.dtsi
MODULE
imx6qdl-phytec-phycore-som.dtsi
i.MX6
imx6q.dtsi
imx6dl.dtsi
imx6qdl.dtsi
skeleton.dtsi
Legend:
fileA.dts
fileB.dtsi
Device tree file fileA.dts includes the file fileB.dtsi.

Figure 3: Device Tree BSP-Yocto-i.MX6-PD15.3.0 phyCORE

There are some special device tree files which do not fit into the layer scheme. The *imx6qdl-phytec-lcd.dtsi* is an example. It does not belong to a baseboard nor to a module. It contains the configuration of a display and a backlight which can be connected to a phyFLEX, or phyCARD Carrier Board.

6.2.1 Switching Expansion Boards and Displays

Disconnect all power before connecting an expansion board. After you plugged in the board, the software support can be activated in the bootloader without recompiling and flashing the images.

The bootloader will enable all necessary DT nodes for a specific expansion board. We created this mechanism as device tree overlay is not yet supported in the 4.1.18 *Linux* kernel (*https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/devicetree/overlay-notes.txt*).

Here is a simple example on how to enable the *imx6qdl-phytec-lcd-018-peb-av-02* (Display Expansion Board) on the phyBOARD-Mira using the *barebox* bootloader.

The configuration for any expansion board currently selected can be found in *env/config-expansions*. E.g.:

```
#!/bin/sh
. /env/expansions/imx6qdl-phytec-lcd-018-peb-av-02
```

To enable an expansion board the file *config-expansions* in the *barebox* environment must be edited.

- To enable *imx6qdl-phytec-lcd-018-peb-av-02*, modify the *env/config-expansions* file and add, or uncomment the text according to the expansion board used. E.g. for the PEB-AV-02 on the phyBOARD-Mira:

  ```
  #!/bin/sh
  . /env/expansions/imx6qdl-phytec-lcd-018-peb-av-02
  ```

config-expansions is called within each bootsource script (*/env/boot/**) and will be executed before every boot process. This will cause the *barebox* to modify the DT used before the boot process. Information on which DT nodes are necessary for the expansion board can be found in the expansion configuration files.

imx6qdl-phytec-lcd-018-peb-av-02:

```
of_enable_node /display@di0
of_enable_node /backlight
of_enable_node /soc/aips-bus@02100000/i2c@021a0000/edt-ft5x06@38
of_enable_node /soc/aips-bus@02000000/pwm@02080000
```

of_enable_node is a *barebox* command and will enable a given DT node.

6.2.2 Handle the Different Displays

If you have chosen a display as expansion you have to select the appropriate display timings in the device tree.

- Append the following lines to the end of *config-expansions*:

  ```
  # imx6qdl-phytec-lcd: 7" display
  #of_display_timings -S /soc/aips-bus@02000000/ldb@020e0008/lvds-
                         channel@0/display-timings/ETM0700G0DH6

  # imx6qdl-phytec-lcd: 5.7" display
  #of_display_timings -S /soc/aips-bus@02000000/ldb@020e0008/lvds-
                         channel@0/display-timings/ETMV570G2DHU

  # imx6qdl-phytec-lcd: 4.3" display
  #of_display_timings -S /soc/aips-bus@02000000/ldb@020e0008/lvds-
                         channel@0/display-timings/ETM0430G0DH6

  # imx6qdl-phytec-lcd: 3.5" display
  #of_display_timings -S /soc/aips-bus@02000000/ldb@020e0008/lvds-
                         channel@0/display-timings/ETM0350G0DH6
  ```

- Uncomment the *of_display_timings -S* ... command for your screen size and save the file and environment.

Beside the display timings, you have to choose the right touchscreen type. Capacitive touchscreens are the standard touchscreens used with our boards. Thus, if you want to use a resistive touchscreen, you have to choose a modified board file in the *env/config-expansions*. The board files for resistive touchscreens all have the suffix *-res*.

Example: If you have the *imx6qdl-phytec-lcd* board, change the *env/config-expansions* from:

```
#!/bin/sh
. /env/expansions/imx6qdl-phytec-lcd
```

to:

```
#!/bin/sh
. /env/expansions/imx6qdl-phytec-lcd-res
```

6.2.3 Bootloader's DT Modifications

The bootloader loads the device tree blob. Then it will modify the loaded memory node at runtime. Thus, there is no need to handle different RAM sizes in the device tree. However, we added a memory node dummy without RAM size information. Because of that a bootloader is needed which sets the memory node with the correct RAM size.

Snippet from *imx6qdl-phytec-pfla02.dtsi*:

```
memory {
        reg = <0x0 0x0>; /* will be filled by bootloader */
};
```

The memory node is not the only node, which is modified by the bootloader. Display nodes and nodes of expansion boards can be enabled, whereas the partition table in the kernel device tree is modified by the bootloader. Because of that it is not possible to use our device trees without modifications with another bootloader, than the *barebox* bootloader from our unified BSP.

The unified BSP contains device tree (DT) sources for the *barebox* bootloader and for the *Linux* kernel. The bootloader DT holds only the absolutely necessary hardware description for the basic board bring-up and is also using a different DT model. Make sure you are working with the right DT.

- To get a sample *barebox* configuration type:

  ```
  barebox$ cat env/boot/mmc
  ```

The output will show something like:

```
#!/bin/sh

global.bootm.image="/mnt/mmc/zImage"  # kernel image
global.bootm.oftree="/mnt/mmc/oftree" # DTB
```

7 Accessing Peripherals

The following sections provide an overview of the supported hardware components and their corresponding operating system drivers. Further changes can be ported upon customer request.

To find out which boards and modules are supported by the release of Phytec's i.MX 6 unified BSP described herein, visit our web page at *http://www.phytec.de/produkte/software/yocto/phytec-unified-yocto-bsp-releases/* and click the corresponding BSP release. Now you can find all hardware supported in the columns "Hardware Article Number" and the correct machine name in the corresponding cell under "Machine Name".

To achieve maximum software re-use, the *Linux* kernel offers a sophisticated infrastructure, layering software components into board specific parts. The BSP tries to modularize the kit features as far as possible, which means that when a customized baseboard, or even a customer specific module is developed, most of the software support can be re-used without error-prone copy-and-paste. The kernel code corresponding to the boards can be found in device trees (DT) under *linux/arch/arm/boot/dts/*.dts**.

In fact, software re-use is one of the most important features of the *Linux* kernel and especially of the ARM implementation, which always had to fight with an insane number of possibilities of the System-on-Chip CPUs.

The whole board specific hardware is described in DTs and is not part of the kernel image itself. The hardware description is in its own separate binary, called device tree blob (DTB) (*section 6 "Device Tree (DT)"*).

Please read also section *6.2 "Phytec i.MX 6 BSP Device Tree Concept"* to get an understanding of our unified i.MX 6 BSP device tree model.

The following sections provide an overview of the supported hardware components and their operating system drivers on the i.MX 6 platform.

7.1 i.MX 6 Pin Muxing

The i.MX 6 SoC contains many peripheral interfaces. In order to reduce package size and lower overall system cost while maintaining maximum functionality, many of the i.MX 6 terminals can multiplex up to eight signal functions. Although there are many combinations of pin multiplexing that are possible, only a certain number of sets, called IO sets, are valid due to timing limitations. These valid IO sets were carefully chosen to provide many possible application scenarios for the user.

Please refer to the Freescale i.MX 6 Reference Manuals for more information about the specific pins and the muxing capabilities:

- for the Solo and DualLite: *http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=i.MX6DL&tab=Documentation_Tab&link_28*
- for the Dual and Quad core: *http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=i.MX6Q&tab=Documentation_Tab&pspll_35*

The IO set configuration, also called muxing, is done in the Device Tree. The driver *pinctrl-single* reads the DT's node "fsl,pins" and does the appropriate pin muxing.

The following is an example of the pin muxing of the UART4 device in *imx6qdl-phytec-pfla02.dtsi*:

```
pinctrl_uart4: uart4grp {
        fsl,pins = <
                MX6QDL_PAD_KEY_COL0__UART4_TX_DATA     0x1b0b1
                MX6QDL_PAD_KEY_ROW0__UART4_RX_DATA     0x1b0b1
         >;
};
```

The first part of the string `MX6QDL_PAD_KEY_COL0__UART4_TX_DATA` names the pad (e.g. *PAD_KEY_COL0*). The second part of the string (here *UART4_TX_DATA*) is the desired muxing option for this pad.

The pad setting value (hex value on the right) is explained in: *linux/Documentation/devicetree/bindings/pinctrl/fsl,imx6q-pinctrl.txt*

In this example the pad setting value *0x1b0b1* means the pin is configured with: PAD_CTL_HYS, PAD_CTL_SRE_SLOW, PAD_CTL_DSE_40ohm, PAD_CTL_SPEED_MED, PAD_CTL_PUS_100K_UP, PAD_CTL_PUE and PAD_CTL_PKE.

This example is defined in:

- *linux/arch/arm/boot/dts/imx6q-pinfunc.h* for the i.MX 6 Dual/Quad, and
- *linux/arch/arm/boot/dts/imx6dl-pinfunc.h* for the i.MX6 Solo/DualLite.

7.2 Serial TTYs

The i.MX 6 SoC provides up to 5 so called UART units. Phytec boards support different numbers of these UART units.

Phytec i.MX 6 boards use different UART units as standard console output. The following table gives an overview of the UART units used for each i.MX 6 board available.

Board	Standard Console Output
phyFLEX-i.MX 6	ttymxc3
phyCARD-i.MX 6	ttymxc2
phyBOARD-Alcor i.MX 6	ttymxc3
phyBOARD-Subra i.MX 6	ttymxc3
phyBOARD-Mira i.MX 6	ttymxc1

- From the command line prompt of *Linux* user space you can easily check the availability of other UART interfaces with:
  ```
  target$ echo "test" > /dev/ttymxc2
  ```
  Be sure that the baud rate is set correctly on host and target side.

In order to get the currently configured baud rate, you can use the command *stty* on the target. The following example shows how to copy all serial settings from *ttymxc3* (the standard console on most i.MX 6 boards) to *ttymxc2*.

- First get the current parameters with:
  ```
  target$ stty -F /dev/ttymxc3 -g
  ```

```
5500:5:1cb2:a3b:3:1c:7f:15:4:0:1:0:11:13:1a:0:12:f:17:16:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0
```

- Now use the output from the *stty* command above as argument for the next command:
  ```
  target$ stty -F /dev/ttymxc2
  5500:5:1cb2:a3b:3:1c:7f:15:4:0:1:0:11:13:1a:0:12:f:17:16:0:0:0:0:0:0:0
  :0:0:0:0:0:0:0:0:0
  ```

Here is an example from *imx6qdl-phytec-pfla02.dtsi*:

```
&iomuxc {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_hog>;

        imx6q-phytec-pfla02 {
                pinctrl_uart3: uart3grp {
                        fsl,pins = <
                                MX6QDL_PAD_EIM_D24__UART3_TX_DATA 0x1b0b1
                                MX6QDL_PAD_EIM_D25__UART3_RX_DATA 0x1b0b1
                                MX6QDL_PAD_EIM_D30__UART3_RTS_B   0x1b0b1
                                MX6QDL_PAD_EIM_D31__UART3_CTS_B   0x1b0b1
                        >;
                };

                pinctrl_uart4: uart4grp {
                        fsl,pins = <
                               MX6QDL_PAD_KEY_COL0__UART4_TX_DATA 0x1b0b1
                               MX6QDL_PAD_KEY_ROW0__UART4_RX_DATA 0x1b0b1
                        >;
                };
        };
};

&uart3 {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_uart3>;
        fsl,uart-has-rtscts;
        status = "disabled";
};

&uart4 {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_uart4>;
        status = "disabled";
};
```

7.3 Network

The Ethernet features provided by our modules and boards vary (e.g.: 1 x 10/100 Mbit, or 1 x 1 Gbit).

However, all interfaces offer a standard *Linux* network port which can be programmed using the BSD socket interface.

The whole network configuration is handled by the *systemd-networkd* daemon. The relevant configuration files can be found on the target in */lib/systemd/network/* and also in the BSP in *meta-yogurt/recipes-core/systemd/systemd/*.

IP addresses can be configured within **.network* files. The default IP address and netmask for eth0 is:
eth0: 192.168.3.11/24

The DT Ethernet setup might be split into two files depending on your hardware configuration, the module DT and the board specific DT.

Example phyCORE-i.MX 6 mounted on phyBOARD-Mira i.MX 6:

Module DT, *arch/arm/boot/dts/imx6qdl-phytec-phycore-som.dtsi*:
```
&fec {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_enet>;
        phy-handle = <&ethphy>;
        phy-mode = "rgmii";
        phy-supply = <&vdd_eth_io>;
        phy-reset-gpios = <&gpio1 14 GPIO_ACTIVE_LOW>;
        status = "disabled";

        mdio {
                #address-cells = <1>;
                #size-cells = <0>;

                ethphy: ethernet-phy@3 {
                        reg = <3>;
                        txc-skew-ps = <1680>;
                        rxc-skew-ps = <1860>;
                };
        };
};
```

```
&iomuxc {
        imx6qdl-phytec-phycore-som {
                pinctrl_enet: enetgrp {
                        fsl,pins = <
                             MX6QDL_PAD_ENET_MDIO__ENET_MDIO        0x1b0b0
                             MX6QDL_PAD_ENET_MDC__ENET_MDC          0x1b0b0
                             MX6QDL_PAD_RGMII_TXC__RGMII_TXC        0x1b0b0
                             MX6QDL_PAD_RGMII_TD0__RGMII_TD0        0x1b0b0
                             MX6QDL_PAD_RGMII_TD1__RGMII_TD1        0x1b0b0
                             MX6QDL_PAD_RGMII_TD2__RGMII_TD2        0x1b0b0
                             MX6QDL_PAD_RGMII_TD3__RGMII_TD3        0x1b0b0
                             MX6QDL_PAD_RGMII_TX_CTL__RGMII_TX_CTL  0x1b0b0
                             MX6QDL_PAD_ENET_REF_CLK__ENET_TX_CLK   0x1b0b0
                             MX6QDL_PAD_RGMII_RXC__RGMII_RXC        0x1b0b0
                             MX6QDL_PAD_RGMII_RD0__RGMII_RD0        0x1b0b0
                             MX6QDL_PAD_RGMII_RD1__RGMII_RD1        0x1b0b0
                             MX6QDL_PAD_RGMII_RD2__RGMII_RD2        0x1b0b0
                             MX6QDL_PAD_RGMII_RD3__RGMII_RD3        0x1b0b0
                             MX6QDL_PAD_RGMII_RX_CTL__RGMII_RX_CTL  0x1b0b0
                             MX6QDL_PAD_ENET_TX_EN__ENET_TX_EN      0x1b0b0
                             MX6QDL_PAD_SD2_DAT1__GPIO1_IO14     0x80000000
                        >;
                };
        };
};
```

Board specific DT, *arch/arm/boot/dts/imx6dl-phytec-mira-rdk-nand.dts :*

```
&ethphy {
        max-speed = <100>;
};

&fec {
        status = "okay";
};
```

7.4 CAN Bus

The phyCORE-i.MX 6 provides a CAN interface, which is supported by drivers using the proposed *Linux* standard CAN framework *SocketCAN*. Using this framework, CAN interfaces can be programmed with the BSD socket API.

The Controller Area Network (CAN) bus offers a low-bandwidth, prioritized message fieldbus for serial communication between microcontrollers. Unfortunately, CAN was not designed with the ISO/OSI layer model in mind, so most CAN APIs available throughout the industry do not support a clean separation between the different logical protocol layers, as for example known from Ethernet.

The *SocketCAN* framework for *Linux* extends the BSD socket API concept towards CAN bus. Because of that, using this framework, the CAN interfaces can be programmed with the BSD socket API and behaves like an ordinary *Linux* network device, with some additional features special to CAN.

- E.g., use:
  ```
  target$ ip link
  ```
  to see if the interface is up or down, but the given MAC and IP addresses are arbitrary and obsolete.

- To get the information on can0 (which represents i.MX 6's CAN module FLEXCAN1) such as bit rate and error counters type:
  ```
  target$ ip -d -s link show can0
  ```

The information for can0 will look like the following:

```
2: can0: <NOARP,ECHO> mtu 16 qdisc pfifo_fast state DOWN mode DEFAULT group default qlen 10
    link/can  promiscuity 0
    can state STOPPED (berr-counter tx 0 rx 0) restart-ms 0
          bitrate 10000 sample-point 0.866
          tq 6666 prop-seg 6 phase-seg1 6 phase-seg2 2 sjw 1
          flexcan: tseg1 4..16 tseg2 2..8 sjw 1..4 brp 1..256 brp-inc 1
          clock 30000000
          re-started bus-errors arbit-lost error-warn error-pass bus-off
          0          0          0          0          0          0
    RX: bytes  packets  errors  dropped overrun mcast
    0          0        0       0       0       0
    TX: bytes  packets  errors  dropped carrier collsns
    0          0        0       0       0       0
```

The output contains a standard set of parameters also shown for Ethernet interfaces, so not all of these are necessarily relevant for CAN (for example the MAC address).

The following output parameters contain useful information:

Field	Description
can0	Interface Name
NOARP	CAN cannot use ARP protocol
MTU	Maximum Transfer Unit
RX packets	Number of Received Packets
TX packets	Number of Transmitted Packets
RX bytes	Number of Received Bytes
TX bytes	Number of Transmitted Bytes
errors...	Bus Error Statistics

The CAN configuration is done in the *systemd* configuration file */lib/systemd/system/can0.service*.

For a persistent change of e.g. the default bitrates change the configuration in the BSP under *meta-yogurt/recipes-core/systemd/systemd/can0.service* in the root filesystem instead and rebuild the root filesystem.

```
[Unit]
Description=can0 interface setup

[Service]
Type=simple
RemainAfterExit=yes
ExecStart=/sbin/ip link set can0 up type can bitrate 500000
ExecStop=/sbin/ip link set can0 down

[Install]
WantedBy=basic.target
```

The *can0.service* is started after boot by default. You can start and stop it with:

```
target$ systemctl stop can0.service
target$ systemctl start can0.service
```

You can send messages with *cansend* or receive messages with *candump*:

```
target$ cansend can0 123#45.67
target$ candump can0
```

To generate random CAN traffic for testing purpose use *cangen*.

```
target$ cangen
```

See `cansend --help` and `candump --help` messages for further information on options and usage.

The corresponding kernel part can be found within the board specific DT, e.g. *arch/arm/boot/dts/imx6qdl-phytec-pfla02.dtsi*

```
&can1 {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_flexcan1>;
        status = "disabled";
};

&iomuxc {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_hog>;

        imx6q-phytec-pfla02 {
                pinctrl_flexcan1: flexcan1grp {
                        fsl,pins = <
                                MX6QDL_PAD_KEY_ROW2__FLEXCAN1_RX  0x1b0b0
                                MX6QDL_PAD_KEY_COL2__FLEXCAN1_TX  0x1b0b0
                        >;
                };
        };
};
```

7.5 MMC/SD Card

All i.MX 6 kits support a slot for Secure Digital Cards and Multi Media Cards to be used as general purpose block devices. The phyFLEX-i.MX 6 supports even two slots for Secure Digital Cards and Multi Media Cards.

These devices can be used in the same way as any other block device.

⚠	This kind of devices are hot pluggable, nevertheless you must pay attention not to unplug the device while it is still mounted. This may result in data loss.

After inserting an MMC/SD card, the kernel will generate new device nodes in /dev. The full device can be reached via its */dev/mmcblk0* device node, MMC/SD card partitions will show up in the following way:

```
/dev/mmcblk0p<Y>
```

<Y> counts as the partition number starting from 1 to the max. count of partitions on this device.

The partitions can be formatted with any kind of file system and also handled in a standard manner, e.g. the *mount* and *umount* command work as expected.

- These partition device nodes will only be available if the card contains a valid partition table (”hard disk” like handling). If it does not contain one, the whole device can be used as a file system (”floppy” like handling). In this case */dev/mmcblk0* must be used for formatting and mounting.
- The cards are always mounted as being writable. Write-protection of MMC/SD cards is only recognized on the phyCARD and the phyFLEX Carrier Board.

DT configuration for the MMC/SD interface in
arch/arm/boot/dts/imx6qdl-phytec-pfla02.dtsi:

```
&iomuxc {
        imx6q-phytec-pfla02 {

                pinctrl_usdhc2: usdhc2grp {
                        fsl,pins = <
                                MX6QDL_PAD_SD2_CMD__SD2_CMD        0x170f9
                                MX6QDL_PAD_SD2_CLK__SD2_CLK        0x100f9
                                MX6QDL_PAD_SD2_DAT0__SD2_DATA0     0x170f9
                                MX6QDL_PAD_SD2_DAT1__SD2_DATA1     0x170f9
                                MX6QDL_PAD_SD2_DAT2__SD2_DATA2     0x170f9
                                MX6QDL_PAD_SD2_DAT3__SD2_DATA3     0x170f9
                        >;
                };

                pinctrl_usdhc3: usdhc3grp {
                        fsl,pins = <
                                MX6QDL_PAD_SD3_CMD__SD3_CMD        0x17059
                                MX6QDL_PAD_SD3_CLK__SD3_CLK        0x10059
                                MX6QDL_PAD_SD3_DAT0__SD3_DATA0     0x17059
                                MX6QDL_PAD_SD3_DAT1__SD3_DATA1     0x17059
                                MX6QDL_PAD_SD3_DAT2__SD3_DATA2     0x17059
                                MX6QDL_PAD_SD3_DAT3__SD3_DATA3     0x17059
                        >;
                };

                pinctrl_usdhc3_100mhz: usdhc3grp100mhz {
                        fsl,pins = <
                                MX6QDL_PAD_SD3_CMD__SD3_CMD        0x170b9
                                MX6QDL_PAD_SD3_CLK__SD3_CLK        0x100b9
                                MX6QDL_PAD_SD3_DAT0__SD3_DATA0     0x170b9
                                MX6QDL_PAD_SD3_DAT1__SD3_DATA1     0x170b9
                                MX6QDL_PAD_SD3_DAT2__SD3_DATA2     0x170b9
                                MX6QDL_PAD_SD3_DAT3__SD3_DATA3     0x170b9
                        >;
                };
                pinctrl_usdhc3_200mhz: usdhc3grp200mhz {
                        fsl,pins = <
                                MX6QDL_PAD_SD3_CMD__SD3_CMD        0x170f9
                                MX6QDL_PAD_SD3_CLK__SD3_CLK        0x100f9
                                MX6QDL_PAD_SD3_DAT0__SD3_DATA0     0x170f9
                                MX6QDL_PAD_SD3_DAT1__SD3_DATA1     0x170f9
                                MX6QDL_PAD_SD3_DAT2__SD3_DATA2     0x170f9
                                MX6QDL_PAD_SD3_DAT3__SD3_DATA3     0x170f9
                        >;
                };

                pinctrl_usdhc3_cdwp: usdhc3cdwp {
                        fsl,pins = <
                              MX6QDL_PAD_ENET_RXD0__GPIO1_IO27 0x80000000
                              MX6QDL_PAD_ENET_TXD1__GPIO1_IO29 0x80000000
                        >;
                };
        };
};

&usdhc2 {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_usdhc2>;
        cd-gpios = <&gpio1 4 0>;
        wp-gpios = <&gpio1 2 0>;
        status = "disabled";
};

&usdhc3 {
        pinctrl-names = "default", "state_100mhz", "state_200mhz";
        pinctrl-0 = <&pinctrl_usdhc3
                     &pinctrl_usdhc3_cdwp>;
        pinctrl-1 = <&pinctrl_usdhc3_100mhz
                        &pinctrl_usdhc3_cdwp>;
        pinctrl-2 = <&pinctrl_usdhc3_200mhz
                        &pinctrl_usdhc3_cdwp>;
        cd-gpios = <&gpio1 27 0>;
        wp-gpios = <&gpio1 29 0>;
        no-1-8-v;

        vmmc-supply = <&vdd_sd0_reg>;
        status = "disabled";
};
```

7.6 eMMC Devices

Phytec modules, e.g. phyCORE-i.MX 6, can be populated with an eMMC memory chip as main storage instead of the NAND Flash. eMMC devices contain raw MLC memory cells (*section 7.6.3*) combined with a memory controller, that handles ECC and wear leveling. They are connected via an MMC/SD interface to the i.MX 6 and are represented as block devices in the *Linux* kernel like SD cards, flash drives, or hard disks.

The electric and protocol specification is provided by JEDEC (*https://www.jedec.org/standards-documents/technology-focus-areas/flash-memory-ssds-ufs-emmc/e-mmc*). The eMMC manufacturer's datasheet is mostly relatively short and meant to be read together with the supported version of the JEDEC eMMC standard.

Phytec currently utilizes two eMMC chips:

Module article number	eMMC chip	Size	JEDEC version
PCM-058-30242C0X.A0	Micron MTFC4GMDEA-4M IT	4 GB	MMC 4.41
PCM-058-40233C0I.A0	Micron MTFC8GLDEA-4M IT	8 GB	MMC 4.41

7.6.1 Extended CSD Register

eMMC devices have an extensive amount of extra information and settings that is available via the *Extended CSD* registers. For a detailed list of the registers see manufacturer datasheets and the JEDEC standard.

- In the *Linux* user space you can query the registers with:
  ```
  target$ mmc extcsd read /dev/mmcblk3
  ```

You will see:

```
=============================================
Extended CSD rev 1.5 (MMC 4.41)
=============================================

Card Supported  Command  sets [S_CMD_SET:  0x01] [...]
```

- In the bootloader you can use
  ```
  bootloader$ detect mmc3
  ```
  if the device is not probed yet, and then
  ```
  bootloader$ mmc_extcsd /dev/mmc3
  ```

7.6.2 Enable Background Operations (BKOPS)

In contrast to raw NAND Flashs an eMMC device contains a Flash Transfer Layer (FTL) that handles the wear leveling, block management and ECC of the raw MLC cells. This requires some maintenance tasks, e.g. erasing non used blocks, that are performed regularly. These tasks are called *Background Operations* (BKOPS).

In the default case, which depends on the chip, the background operations are not executed periodically which impacts the worst-case read and write latency.

Therefore the JEDEC Standard version v4.41 specifies a method, that the host can issue BKOPS manually. See the JEDEC Standard chapter *"Background Operations"* and the description of registers BKOPS_EN (Reg: 163) and BKOPS_START (Reg: 164) in the eMMC datasheet for more details.

Meaning of Register BKOPS_EN (Reg: 163) Bit MANUAL_EN (Bit 0):

- Value 0: The host does not support manual trigger of BKOPS. Device write performance suffers.
- Value 1: The host does support manual trigger of BKOPS. It will issue BKOPS from time to time when it does not need the device.

The mechanism to issue background operations is already implemented in the *Linux* kernel since *v3.7*. You only have to enable BKOPS_EN on the eMMC device (see below for details).

The JEDEC standard *v5.1* introduces a new automatic BKOPS feature. It frees the host to trigger the background operations regularly, because the device starts BKOPS itself when it is idle (see description of bit AUTO_EN in register BKOPS_EN (Reg: 163)).

eMMC chips deployed by Phytec currently do not support the new standard *v5.1*. The *Linux* kernel and user space tool *mmc* do not support the feature neither.

- To check whether BKOPS_EN is set, execute:

```
target$ mmc extcsd  read /dev/mmcblk3 | grep BKOPS_EN
```

The output will be for example:

```
Enable background operations handshake [BKOPS_EN]:  0x01
```

Here BKOPS_EN is enabled. The host will issue background operations from time to time.

There is also a kernel boot message showing if BKOPS_EN is not set:

```
mmc1:  MAN_BKOPS_EN  bit is not set
```

- To set the BKOPS_EN bit execute:

```
target$ bkops enable /dev/mmcblk3
```

- To ensure that the new setting is taken over and the kernel triggers BKOPS by itself, shut down the system with

  ```
  target$ poweroff
  ```

  and perform a power cycle.

	The BKOPS_EN bit is one-time-programmable only. It cannot be reversed.

7.6.3 Enable pseudo-SLC Mode

eMMC devices use MLC memory cells (*https://en.wikipedia.org/wiki/Multi-level_cell*) to store the data. Compared with SLC memory cells used in NAND Flashs MLC memory cells have a lower reliability and a higher error rate at lower costs.

If you prefer reliability over storage capacity, you can enable the *pseudo-SLC* mode, or *SLC mode*. The method, used here, employs the *enhanced attribute*, describe in the JEDEC standard, which can be set for continuous regions of the device. The JEDEC standard does not specify the implementation details and the guarantees of the *enhanced attribute*. This is left to the chipmaker. For the Micron chips the *enhanced attribute* increases the reliability, but also halves the capacity.

	When enabling the *enhanced attribute* on the device all data will be lost.

The following sequence shows how to enable the *enhanced attribute*.

- First obtain the current size of the eMMC device with:

  ```
  target$ parted -m /dev/mmcblk3 unit B print
  ```

You will receive:

```
BYT;
/dev/mmcblk3:3783262208B:sd/mmc:512:512:unknown:MMC MMC04G:;
```

As you can see this device has 3783262208 Byte = 3608.0 MiB.

- To get the maximum size of the device after pseudo-SLC is enabled use

  ```
  target$ mmc extcsd read /dev/mmcblk3 | grep ENH_SIZE_MULT -A 1
  ```

which shows for example:

```
Max Enhanced Area Size [MAX_ENH_SIZE_MULT]: 0x0001c3 i.e. 1847296 KiB
--
Enhanced User Data Area Size [ENH_SIZE_MULT]: 0x000000 i.e. 0 KiB
```

Here the maximum size is 1847296 KiB = 1804 MiB

- Now, you can set *enhanced attribute* for the whole device, e.g. 1847296 KiB, by typing:
  ```
  target$ mmc enh_area set -y 0 1847296 /dev/mmcblk3
  ```

You will get:

```
Done  setting ENH_USR area on /dev/mmcblk3
setting OTP PARTITION_SETTING_COMPLETED!
Setting OTP PARTITION_SETTING_COMPLETED on /dev/mmcblk3 SUCCESS
Device power cycle needed for settings to take effect.
Confirm that PARTITION_SETTING_COMPLETED bit is set using 'extcsd read' after power cycle
```

- To ensure that the new setting is taken over shut down the system with
  ```
  target$ poweroff
  ```
  and perform a power cycle.

It is recommended to verify the settings, now.

- First check the value of ENH_SIZE_MULT which must be 1847296 KiB:
  ```
  targe$ mmc extcsd read /dev/mmcblk3 | grep ENH_SIZE_MULT -A 1
  ```

You should receive:

```
Max Enhanced Area Size [MAX_ENH_SIZE_MULT]: 0x0001c3
 i.e. 1847296 KiB
--
Enhanced User Data Area Size [ENH_SIZE_MULT]: 0x0001c3
 i.e. 1847296 KiB
```

- Finally check the size of the device, in this example it should be 1891631104 Byte = 1804.0 MiB with:
  ```
  target$ parted -m  /dev/mmcblk3 unit B print
  ```

```
BYT;
/dev/mmcblk3:1891631104B:sd/mmc:512:512:unknown:MMC MMC04G:;
```

- Now you can flash your new image.

Further reference:

https://www.micron.com/support/faqs/products/managed-nand/emmc (chapter "*What are the enhanced technology features mentioned in JEDEC specification, and what are the benefits?*")

7.6.4 eMMC (Boot-)Partitions

An eMMC device contains already four different hardware partitions: *user*, *boot1*, *boot2* and *rpmb*.

The *user* partition is called the *User Data Area* in the JEDEC standard and is the main storage partition. The partitions *boot1* and *boot2* can be used to host the bootloader and are more reliable. Which partition the i.MX 6 uses to load the bootloader, is controlled by the *boot* configuration of the eMMC device. The partition *rpmb* is a small partition and can only be accessed via *trusted mechanism*.

Furthermore the *user* partition can be divided into four user defined *General Purpose Area Partitions*. An explanation of this feature exceeds the scope of this document. For further information see the JEDEC Standard chapter *7.2 Partition Management*.

Do not confuse eMMC partitions with partitions of a DOS MBR, or GPT partition table.

The current Phytec BSP does not use the extra partitioning feature of eMMC devices. The *barebox* is flashed at the beginning of the *user* partition. The barebox environment is placed at a fix location after the *barebox*. An MBR partition table is used to create two partitions, a FAT32 boot and ext4 rootfs partition. They are located right after the *barebox* and the *barebox* environment. The FAT32 boot partition contains kernel and device tree.

There are two ways to flash the bootloader to one of the two boot partitions and to switch the boot device, either via bootloader, or via user space commands, as shown in the following examples.

1. Via bootloader:

- First detect the eMMC if not already done:
  `bootloader$ detect  mmc3`
- Now, query some information about the eMMC device:
  `bootloader$ devinfo mmc3`

The output will look like this:

```
Host information:
[...]
Parameters:
 boot: disabled ("disabled", "boot0", "boot1", "user")
 nt_signature: 85f3edec
 probe: 1
```

Here the boot partition is currently set to *disabled*.

- Next flash the bootloader to the partition *boot0*:
  `bootloader$ cp -v /mnt/tftp/barebox.bin /dev/mmc3.boot0`
- Finally switch the boot partition to *boot0*:
  `bootloader$ mmc3.boot=boot0`

2. Via user space commands:

The partitions *boot0* and *boot1* are read-only by default. To write to them from user space you have to disable *force_ro* in the sysfs.

- Type:
  `target$ echo 0 > /sys/block/mmcblk3boot0/force_ro`
- Now, flash the bootloader to partition *boot0*:
  `target$ dd if=/tmp/barebox.bin of=/dev/mmcblk3boot0`

- After that set the boot partition configuration from user space using the *mmc* tool:
  `target$ mmc bootpart enable 1 0 /dev/mmcblk3`
  (for *boot0*)
  `target$ mmc bootpart enable 2 0 /dev/mmcblk3`
  (for *boot1*)
  `target$ mmc bootpart enable 7 0 /dev/mmcblk3`
  (for *user*)

Currently, the *mmc* tool does not provide a way to show the current boot partition configuration.

7.6.5 Reliable Write

There are two different *Reliable Write* options:

1. *Reliable Write* option for a whole eMMC device/partition
2. *Reliable Write* for single write transactions

Do not confuse eMMC partitions with partitions of a DOS MBR, or GPT partition table (see previous section).

The first *Reliable Write* option can be enable with the *mmc* tool:
`target$ mmc --help`

```
[...]
mmc write_reliability set <-y|-n> <partition> <device>
```

The second *Reliable Write* option is the configuration bit *Reliable Write Request parameter (bit 31)* in command *CMD23*. It is already used in the kernel since *v3.0* by file systems, e.g. *ext4* for the journal, and user space applications such as *fdisk* for the partition table. In the *Linux* kernel source code it's handled via flag *REQ_META*.

Conclusion:

ext4 file system with mount option *data=journal* should be safe against power cuts. The file system check can recover the file system after a power failure, but data that was written just before the power cut maybe lost. At least a consistent state of the file system can be recovered. To ensure data consistency for the files of an application, the system functions *fdatasync*, or *fsync* should be used in the application.

7.6.6 Resize the ext4 Root Filesystem

parted and *resize2fs* can be used to expand the root filesystem. The example works for any block device such as eMMC, SD card, or hard disk.

- Get the current device size:

  ```
  target$ parted -m /dev/mmcblk0 unit B print
  ```

The output looks like:

```
BYT;
/dev/mmcblk0:3991928832B:sd/mmc:512:512:msdos:SD  USD:;
1:4194304B:12582911B:8388608B:::lba;
2:12582912B:138412031B:125829120B:ext4::;
```

- Now use size of device minus one as the new end of the second partition (e.g. 3991928832B):

  ```
  target$ parted /dev/mmcblk0 resizepart 2 3991928831B
  ```

- After expanding the partition size, resize the *ext4* file system in the partition:

  ```
  target$ resize2fs /dev/mmcblk0p2
  ```

The command's output looks like this:

```
resize2fs 1.42.9 (28-Dec-2013)
Filesystem at /dev/mmcblk0p2 is mounted on /; on-line resizing required
old_desc_blocks = 1,  new_desc_blocks = 15
The filesystem on /dev/mmcblk0p2 is now 3886080 blocks long.
```

Increasing the file system size can be done while it is mounted. An *on-line* resizing operation is performed. But you can also boot the board from SD card and then resize the file system on the eMMC partition while it is not mounted.

7.6.7 Erase the Device

It is possible to erase the eMMC device directly rather than overwriting it with zeros. The eMMC block management algorithm will erase the underlying MLC memory cells or mark these blocks as *discard*. The data on the device is lost and will be read back as zeros.

To erase the eMMC device the program *blkdiscard* is required. Currently the tool is not installed on our images by default.

- Hence, first install the program to your image. To do so copy the following line into the file *conf/local.conf* in the BSP source directory.
  ```
  IMAGE_INSTALL_append  = " util-linux"
  ```
- Now rebuild the image.
- After booting from SD card execute:
  ```
  target$ blkdiscard --secure /dev/mmcblk3
  ```

The option *--secure* ensures that the command waits until the eMMC device has erased all blocks.

`dd if=/dev/zero of=/dev/mmcblk3` also destroys all information on the device, but is bad for wear leveling and takes much longer!

For raw NAND flashes the same could be achieved with the command *flash_erase*.

7.6.8 Additional Software in the BSP

In the BSP you will also find the tool *flashbench* which allows to get the page size and erase block size.

- E.g. type:

  ```
  target$ flashbench -a /dev/mmcblk3 --blocksize=1024
  ```

This will, for example, result in:

```
align 1073741824 pre 779µs   on 1.21ms   post 768µs   diff 439µs
align 536870912  pre 855µs   on 1.29ms   post 858µs   diff 433µs
align 268435456  pre 846µs   on 1.29ms   post 821µs   diff 454µs
align 134217728  pre 812µs   on 1.25ms   post 822µs   diff 429µs
align 67108864   pre 846µs   on 1.29ms   post 832µs   diff 452µs
align 33554432   pre 830µs   on 1.24ms   post 807µs   diff 422µs
align 16777216   pre 841µs   on 1.26ms   post 842µs   diff 418µs
align 8388608    pre 842µs   on 1.27ms   post 814µs   diff 446µs
align 4194304    pre 838µs   on 1.28ms   post 842µs   diff 436µs
align 2097152    pre 827µs   on 928µs    post 834µs   diff 97.9µs
align 1048576    pre 826µs   on 921µs    post 827µs   diff 94.5µs
align 524288     pre 828µs   on 924µs    post 841µs   diff 89.6µs
align 262144     pre 835µs   on 903µs    post 841µs   diff 65.1µs
align 131072     pre 842µs   on 949µs    post 853µs   diff 101µs
align 65536 pre 854µs   on 959µs    post 858µs   diff 103µs
align 32768 pre 844µs   on 954µs    post 869µs   diff 97.2µs
align 16384 pre 862µs   on 962µs    post 847µs   diff 108µs
align 8192  pre 849µs   on 946µs    post 847µs   diff 98.5µs
align 4096  pre 858µs   on 953µs    post 855µs   diff 96.9µ2s
align 2048  pre 845µs   on 936µs    post 846µs   diff 90.8µs
```

For an explanation how to interpret the output see
https://git.linaro.org/people/arnd.bergmann/flashbench.git/blob/HEAD:/README#l1

7.7 NAND Flash

Phytec i.MX 6 modules are equipped with raw NAND memory, which is used as media for storing *Linux*, DTB and root filesystem, including applications and their data files.

The NAND Flash is connected to the General Purpose Media Interface (GPMI) of the i.MX 6. The NAND Flash type and size is automatically being detected via the Open NAND Flash Interface (ONFI) during boot.

This type of media is managed by the UBI file system. This file system uses compression and decompression on the fly to increase the quantity of data stored.

From *Linux* user space the NAND Flash partitions start with */dev/mtdblock0*. Only the */dev/mtdblock4* on the Phytec modules has a file system, meaning that the other partitions cannot be mounted to the root filesystem. The only way to access them is by flashing a prepared flash image into the corresponding */dev/mtd* device node.

The partitions of a NAND Flash are defined in all DTs, but the *barebox* bootloader overwrites only the partitions of the kernel device tree. Thus, changing the partitions has to be done either in the *barebox* DT or in the *barebox* environment. How to modify the partitions during runtime in the *barebox* environment is described in section *8.1 "Changing MTD Partitions"*. Adding new partitions can be done by creating a new partition node in the corresponding board device tree (*6.2 "Phytec i.MX 6 BSP Device Tree Concept"*).

The property *label* defines the name of the partition and the *reg* value the offset and size of a partition. Do not forget to update all following partitions when adding a partition, or changing a partition's size.

The partitions are defined in the DT, e.g. *imx6qdl-phytec-pfla02.dtsi* in the *barebox*:

```
&gpmi {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_gpmi_nand>;
        nand-on-flash-bbt;
        status = "okay";

        #address-cells = <1>;
        #size-cells = <1>;

        partition@0 {
                label = "barebox";
                reg = <0x0 0x400000>;
        };

        partition@1 {
                label = "barebox-environment";
                reg = <0x400000 0x100000>;
        };

        partition@2 {
                label = "oftree";
                reg = <0x500000 0x100000>;
        };

        partition@3 {
                label = "kernel";
                reg = <0x600000 0x800000>;
        };

        partition@4 {
                label = "root";
                reg = <0xe00000 0x0>;
        };
};
```

We also kept the partition notes in the *Linux* Kernel DT as fallback.

7.8 GPIOs

Phytec boards have often a set of pins specially dedicated as user I/Os. Those pins are connected directly to i.MX 6 pins and are muxed as GPIOs. They are directly usable in *Linux* user space. The processor has organized its GPIOs into six banks (GPIO1 – GPIO6) of 32 GPIOs each and one bank with 14 GPIOs. *gpiochip0*, *gpiochip32*, *gpiochip64*, *gpiochip96, gpiochip128, gpiochip160* and *gpiochip192* are the sysfs representation of these internal i.MX 6 GPIO banks GPIO1 – GPIO7.

The GPIOs are identified as GPIO<X>_<Y> (e.g. GPIO5_07). <X> identifies the GPIO bank and counts from 1 to 7, while <Y> stands for the GPIO within the bank. <Y> is being counted from 0 to 31 (32 GPIOs on each bank).

By contrast, the *Linux* kernel uses a single integer to enumerate all available GPIOs in the system. The formula to calculate the right number is:

Linux GPIO number = (<X> - 1) * 32 + <Y>

Accessing GPIOs from user space will be done using the sysfs path */sys/class/gpio/*. There you need to register the GPIOs that you want to use by writing their numbers into the file *export, e.g.*:

- `target$ echo 56 > /sys/class/gpio/export`

This will create a new subdirectory *gpio56*. The two files *direction* and *value* in the new subdirectory allow to control the GPIO.

- For example, to use the newly created GPIO 56 as output and to toggle it, execute:

```
target$ echo out > /sys/class/gpio/gpio56/direction
target$ echo 1 > /sys/class/gpio/gpio56/value
target$ echo 0 > /sys/class/gpio/gpio56/value
```

On the carrier board PBA-B-01 for the phyFLEX-i.MX 6, for example, this will toggle LED D105.

- To use, for example, GPIO 135 as input execute the following commands:

```
target$ echo 135 > /sys/class/gpio/export
target$ echo in > /sys/class/gpio/gpio135/direction
target$ cat /sys/class/gpio/gpio135/value
```

On the carrier board PBA-B-01 for the phyFLEX-i.MX 6, for example, this will read pin 3 of GPIO connector X54.

Some of the user IOs are used for special functions on the carrier boards (e.g. GPIO5_8, GPIO1_9 and GPIO7_12 on the PBA-B-01 CB). Before using a user IO refer to the schematic, or the hardware manual of your board to ensure that it is not already in use.

Pinmuxing of some GPIO pins in the device tree *imx6qdl-phytec-pbab01.dtsi*:

```
          pinctrl_hog: hoggrp {
                  fsl,pins = <
                          MX6QDL_PAD_EIM_D23__GPIO3_IO23    0x80000000
                          MX6QDL_PAD_DISP0_DAT3__GPIO4_IO24 0x80000000
                                              /* SPI NOR chipselect */
                          MX6QDL_PAD_SD4_DAT1__GPIO2_IO09   0x80000000
                                                  /* PMIC interrupt */
                          MX6QDL_PAD_ENET_TXD0__GPIO1_IO30  0x80000000
                                                       /* Green LED */
                          MX6QDL_PAD_EIM_EB3__GPIO2_IO31    0x80000000
                                                         /* Red LED */
                          MX6QDL_PAD_GPIO_8__GPIO1_IO08     0x80000000
[...]
```

7.9 LEDs

In case that LEDs are being connected to GPIOs, you have the possibility to access them by a special LED driver interface instead of the general GPIO interface (*section 7.8*). You will then access them using */sys/class/leds/* instead of */sys/class/gpio/*. The maximum brightness of the LEDs can be read from the *max_brightness* file. The *brightness* file will set the brightness of the LED (taking a value from 0 up to *max_brightness*). Most LEDs do not have hardware brightness support and thus will just be turned on by all non-zero brightness settings.

Here is a simple example for the phyFLEX Carrier Board:

- To get all LEDs available, type:
  ```
  target$ ls /sys/class/leds
  ```

```
mmc0::        phycore:green   user_led_red
mmc1::        user_led_green  user_led_yellow
```

- To toogle the LEDs use
  ```
  target$ echo 255 > /sys/class/leds/user_led_green/brightness
  ```
  to turn it ON, and
  ```
  target$ echo 0 > /sys/class/leds/user_led_green/brightness
  ```
  to turn it OFF.

User I/O configuration in device tree file
arch/arm/boot/dts/imx6qdl-phytec-mira-peb-eval-01.dtsi:

```
&iomuxc {
        imx6qdl-phytec-mira-pebeval01 {
                pinctrl_user_leds: user_ledsgrp {
                        fsl,pins = <
                                MX6QDL_PAD_SD3_DAT4__GPIO7_IO01    0x80000000
                                MX6QDL_PAD_SD3_DAT5__GPIO7_IO00    0x80000000
                                MX6QDL_PAD_CSI0_DAT11__GPIO5_IO29  0x80000000
                        >;
                };
        };
};

/ {
        user_leds: user_leds {
                compatible = "gpio-leds";
                pinctrl-names = "default";
                pinctrl-0 = <&pinctrl_user_leds>;
                status = "disabled";
```

```
                user_led_green {
                        gpios = <&gpio5 29 GPIO_ACTIVE_HIGH>;
                        linux,default-trigger = "gpio";
                        default-state = "on";
                };

                user_led_yellow {
                        gpios = <&gpio7 0 GPIO_ACTIVE_HIGH>;
                        linux,default-trigger = "gpio";
                        default-state = "on";
                };

                user_led_red {
                        gpios = <&gpio7 1 GPIO_ACTIVE_HIGH>;
                        linux,default-trigger = "gpio";
                        default-state = "on";
                };
        };
};
```

7.10 SPI Master

Most Phytec boards are equipped with a NOR Flash which connects to the i.MX 6's ECSPI interface. The NOR Flash is suitable for booting (section "*Booting from SPI NOR Flash*").

From *Linux* user space the NOR Flash partitions start with *dev/mtdblock5* (see *mtdinfo -v*). There are currently four partitions: *barebox*, *barebox*-environment, *oftree* and kernel. Please note that there is no root file system partition on the NOR Flash.

The partitions of an SPI Flash are defined in all DTs, but the *barebox* bootloader overwrites only the partitions of the kernel device tree. Thus, changing the partitions has to be done either in the *barebox* DT, or in the *barebox* environment. How to modify the partitions during runtime in the *barebox* environment is described in section *8.1 "Changing MTD Partitions"*.

Adding new partitions can be done, for example in *imx6qdl-phytec-pfla02.dtsi,* by creating a new partition node. The property *label* defines the name of the partition and the *reg* value the offset and size of a partition. Do not forget to update all following partitions when adding a partition, or changing a partition's size.

The *flash* node is defined inside of the SPI master node in the module DTs. The SPI node contains all devices connected to this SPI bus which is in this case only the SPI NOR Flash.

Definition of the SPI master node with the partition layout, e.g. in
imx6qdl-phytec-pfla02.dtsi:
```
&iomuxc {
        imx6q-phytec-pfla02 {
              pinctrl_ecspi3: ecspi3grp {
                     fsl,pins = <
                           MX6QDL_PAD_DISP0_DAT0__ECSPI3_SCLK     0x100b1
                           MX6QDL_PAD_DISP0_DAT2__ECSPI3_MISO     0x100b1
                           MX6QDL_PAD_DISP0_DAT1__ECSPI3_MOSI     0x100b1
                           MX6QDL_PAD_DISP0_DAT3__GPIO4_IO24   0x80000000
                           MX6QDL_PAD_DISP0_DAT4__GPIO4_IO25   0x80000000
                           MX6QDL_PAD_DISP0_DAT5__GPIO4_IO26   0x80000000
                     >;
              };
        };
};

&ecspi3 {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_ecspi3>;
        status = "okay";
        fsl,spi-num-chipselects = <1>;
        cs-gpios = <&gpio4 24 0>;
        flash@0 {
                compatible = "m25p80";
                spi-max-frequency = <20000000>;
                reg = <0>;

                #address-cells = <1>;
                #size-cells = <1>;

                partition@0 {
                        label = "barebox";
                        reg = <0x0 0x100000>;
                };

                partition@1 {
                        label = "barebox-environment";
                        reg = <0x100000 0x20000>;
                };

                partition@2 {
                        label = "oftree";
                        reg = <0x120000 0x20000>;
                };

                partition@3 {
                        label = "kernel";
                        reg = <0x140000 0x0>;
                };
        };
};
```

The *imx6qdl-phytec-pbab01.dtsi* also includes an example for an *spidev* device. This node has been enabled in the phyFLEX-i.MX 6 RDK. *spidev* allows to access an SPI device directly from user space.

```
&ecspi3 {
        spi@1 {
                compatible = "spidev";
                spi-max-frequency = <57600000>;
                reg = <1>;
        };

        spi@2 {
                compatible = "spidev";
                spi-max-frequency = <57600000>;
                reg = <2>;
        };
};
```

The partition layout is also available in the kernel as fallback, e.g. *imx6qdl-phytec-pfla02.dtsi*.

7.11 I²C Bus

The i.MX 6 contains three multimaster fast-mode I^{2}C modules called I2C1, I2C2, and I2C3. Phytec boards provide plenty of different I^{2}C devices connected to the three I^{2}C modules of the i.MX 6. This chapter will describe the basic device usage and its DT representation of some of the I^{2}C devices integrated on our RDK boards.

General I^{2}C bus configuration (e.g. *imx6q-phytec-phycard-imx6-som.dtsi*):

```
&i2c1 {
        status = "okay";
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_i2c1>;
        /* ... */
};

&iomuxc {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_hog>;

        imx6q-phytec-pcaaxl3 {
                pinctrl_i2c1: i2c1grp {
                        fsl,pins = <
                                MX6QDL_PAD_EIM_D21__I2C1_SCL   0x4001b8b1
                                MX6QDL_PAD_EIM_D28__I2C1_SDA   0x4001b8b1
                        >;
                };
        };
};
```

7.11.1 EEPROM

It is possible to read and write directly to the device:

```
target$ /sys/class/i2c-dev/i2c-0/device/0-0050/eeprom
```

- E.g. to read and print the first 1024 bytes of the EEPROM as hex number execute:

  ```
  target$ dd if=/sys/class/i2c-dev/i2c-0/device/0-0050/eeprom bs=1
                                                     count=1024   | od -x
  ```

- To fill the whole EEPROM with zeros use:

  ```
  target$ dd if=/dev/zero of=/sys/class/i2c-dev/i2c-0/device/0-
                              0050/eeprom bs=4096 count=1
  ```

This operation takes some time, because the EEPROM is relatively slow.

DT representation, e.g. in phyFLEX-i.MX 6 file *imx6qdl-phytec-pfla02.dtsi*:

```
        eeprom@50 {
                compatible = "atmel,24c32";
                reg = <0x50>;
        };
```

7.11.2 RTC

RTCs can be accessed via */dev/rtc**. Because Phytec boards have often more than one RTC, there might be more than one RTC device file.

- To find out the name of the RTC device you can read its sysfs entry with:

```
target$ cat /sys/class/rtc/rtc*/name
```

You will get for example:

```
rtc-pcf8563
da9063-rtc
20cc034.snvs-rtc-lp
```

This will list all RTCs including the non-I^2C RTCs. *Linux* assigns RTC devices IDs based on the device tree */aliases* entries if present.

imx6q-phytec-phycard-imx6-som.dtsi:

```
aliases {
        ipu0 = &ipu1;
        ipu1 = &ipu2;
        rtc1 = &pmic;
        rtc2 = &snvs_rtc;
};
```

imx6q-phytec-phycard-imx6-rdk.dts:

```
aliases {
        rtc0 = &i2c_rtc;
};
```

As the time is set according to the value of rtc0 during system boot rtc0 should be always the RTC that is being backed up.

Date and time can be manipulated with the *hwclock* tool, using the *-w* (systohc) and *-s* (hctosys) options.

To set the date first use *date* and then run *hwclock -w -u* to store the new date into the RTC. For more information about this tool refer to the manpage of *hwclock*.

DT representation for I^2C RTCs:
imx6q-phytec-phycard-imx6-rdk.dts:

```
i2c_rtc: rtc@51 {
                compatible = "nxp,rtc8564";
                reg = <0x51>;
};
```

7.11.3 Capacitive Touchscreen

The capacitive touchscreen is a part of the display module.

- For a simple test of this feature start our demo application with:

  ```
  target$ QtDemo
  ```

  This application also includes a *Multitouch Demo*.

- To start another more simple test application type:

  ```
  target$ qt5-opengles2-test
  ```

- To test the basic input handling of the touchscreen use *evtest* after selecting an input device:

  ```
  target$ evtest
  ```

  The raw touch input events will be displayed.

The display DT representation for all displays implemented so far, are summarized in *imx6qdl-phytec-lcd.dtsi*.

DT representation, e.g. in *imx6qdl-phytec-pbab01.dtsi*:

```
&i2c2 {
        polytouch: edt-ft5x06@38 {
                compatible = "edt,edt-ft5406", "edt,edt-ft5x06";
                reg = <0x38>;
                pinctrl-names = "default";
                pinctrl-0 = <&pinctrl_edt_ft5x06>;
                interrupt-parent = <&gpio7>;
                interrupts = <12 0>;
        };
        […]
};

&iomuxc {
        imx6q-phytec-pfla02 {
             pinctrl_edt_ft5x06: edt_ft5x06grp {
                 fsl,pins = <MX6QDL_PAD_GPIO_17__GPIO7_IO12  0x80000000>;
             };
                […]
         };
};
```

7.11.4 LED Dimmer

All LED dimmer controlled LEDs can be accessed through the sysfs. Here are some simple examples:

- Use the *ls* command to list all LEDs:
  ```
  target$ ls /sys/class/leds/
  ```

You will get:

```
green:user4        phyflex:green             red:user1
mmc0::             phyflex:red               yellow:user2
mmc1::             phyflex:user_led_gpio     yellow:user3
```

- To turn on the red user LED1 write 255 to the corresponding file:
  ```
  target$ echo 255 > /sys/class/leds/red\:user1/brightness
  ```

- To turn off the red user LED1 write 0 to the corresponding file:
  ```
  target$ echo 0 > /sys/class/leds/red\:user1/brightness
  ```

Currently, only the GPIO functionality is used with the PCA9533. So you can only switch the LEDs on and off. The dimming feature of the PCA9533 is not yet tested.

DT representation, e.g. in *imx6qdl-phytec-pbab01.dtsi*:

```
leddim: leddimmer@62 {
        compatible = "nxp,pca9533";
        gpio-controller;
        #gpio-cells = <2>;
        nxp,typecodes = <0xFF>;
        nxp,statecodes = <0x55>;
        reg = <0x62>;
};
```

As you can see, the LED dimmer is a gpio-controller node. The PCA9533 controlled LEDs can be used within a gpio-led driver. Please read the related binding documentation in *documentation/devicetree/bindings/* for more information about gpio-controller and gpio-leds.

```
&user_leds_pca9533 {
        led1 {
                label = "red:user1";
                gpios = <&leddim 0 0>;
                linux,default-trigger = "none";
                default-state = "on";
        };

        led2 {
                label = "yellow:user2";
                gpios = <&leddim 1 0>;
                linux,default-trigger = "none";
```

```
                default-state = "on";
        };

        led3 {
                label = "yellow:user3";
                gpios = <&leddim 2 0>;
                linux,default-trigger = "none";
                default-state = "on";
        };

        led4 {
                label = "green:user4";
                gpios = <&leddim 3 0>;
                linux,default-trigger = "none";
                default-state = "on";
        };
};
```

LEDs are exported to the sysfs */sys/class/leds/*.

7.12 USB Host Controller

The USB controller of the i.MX 6 SoC provides a low-cost connectivity solution for numerous consumer portable devices by providing a mechanism for data transfer between USB devices with a line/bus speed up to 480 Mbps. The USB subsystem has four independent USB controller cores. The first core is an On-The-Go (OTG) controller core and capable of acting as an USB peripheral device, or USB host. The second, third and fourth core are host-only controller cores. Only the first and the second core are connected to two USB 2.0 PHY macrocells. The third and fourth core supports only an HSIC interfaces.

The unified BSP includes support for mass storage devices and keyboards. Other USB related device drivers must be enabled in the kernel configuration on demand.

Due to *udev*, all mass storage devices connected get unique IDs and can be found in */dev/disks/by-id*. These IDs can be used in */etc/fstab* to mount the different USB memory devices in different ways.

User USB1 (host) configuration in the kernel device tree *imx6qdl-phytec-pfla02.dtsi*:

```
[…]
                pinctrl_usbh1: usbh1grp {
                        fsl,pins = <
                                MX6QDL_PAD_GPIO_0__GPIO1_IO00  0x80000000
                                MX6QDL_PAD_GPIO_3__USB_H1_OC   0x1b0b0
                        >;
                };

[…]
```

```
&usbh1 {
        vbus-supply = <&reg_usb_h1_vbus>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_usbh1>;
        status = "disabled";
};
```

Kernel device tree *imx6qdl-phytec-pbab01.dtsi*:

```
&usbh1 {
         status = "okay";
 };
```

7.13 USB OTG

Most Phytec boards provide an USB OTG interface. USB OTG ports automatically act as USB device, or USB host. The mode depends on the USB hardware attached to the USB OTG port. If, for example, an USB mass storage device is attached to the USB OTG port, the device will show up as */dev/sda*.

In order to connect the board as USB device to an USB host port (for example a PC), you need to load the appropriate USB gadget, which is a kernel module, with *modprobe*. The BSP includes several USB gadgets: *g_ether*, *g_mass_storage* and *g_ether are activated*.

Examples:

- To start the Ethernet gadget execute:

  ```
  target$ modprobe g_ether
  ```

  on the device. You will get an additional Ethernet interface, e.g. *usb0*.

- To use the mass storage gadget execute:

  ```
  target$ dd if=/dev/zero of=/tmp/file.img bs=1M count=64
  target$ modprobe g_mass_storage file=/tmp/file.img
  ```

  on the device. The host can partition, format and mount the gadget mass storage the same way as any other USB mass storage.

- The *g_serial* gadget is used the same way as *g_ether*, but it creates an additional serial interface like */dev/ttyGS0*.

User USB OTG configuration in the kernel device tree *imx6qdl-phytec-pfla02.dtsi*:

```
&iomuxc {
        imx6q-phytec-pfla02 {
               pinctrl_usbotg: usbotggrp {
                      fsl,pins = <
                             MX6QDL_PAD_GPIO_1__USB_OTG_ID      0x17059
                             MX6QDL_PAD_KEY_COL4__USB_OTG_OC    0x1b0b0
                             MX6QDL_PAD_KEY_ROW4__GPIO4_IO15    0x80000000
                        >;
                };
        };
};

&usbotg {
        vbus-supply = <&reg_usb_otg_vbus>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_usbotg>;
        disable-over-current;
        status = "disabled";
};
```

Kernel device tree *imx6qdl-phytec-pbab01.dtsi*:

```
&usbotg {
        status = "okay";
        dr_mode = "peripheral";
};
```

7.14 SATA

Some boards like the phyFLEX-i.MX 6 Kit feature a *SATA* connecter (Serial ATA) which is used to attach hard disks or optical drives.

As in any other *Linux* system a *SATA* hard disk will show up as standard block device (*/dev/sda* in the following example) after connecting.

- E.g. type:
  `target$ ls -l /dev/sda*`

You will get something like:

```
brw-rw----   1 root   disk      8,  0 May 27 07:56 /dev/sda
brw-rw----   1 root   disk      8,  1 May 27 07:56 /dev/sda1
```

The disk can be partitioned and formatted like any other block device.

To access the partitions you can also use the persistent block device naming in *dev/disk/* (*ArchLinux Wiki-Persistent block device naming*).

7.15 PCIe

Some Phytec boards feature a PCIe- or Mini-PCIe-Slot.

In general PCIe autodetects new devices on the bus. After connecting the device and booting up the system you can use the command *lspci* to see all PCIe devices recognized.

- Type:
  ```
  target$ lspci -v
  ```

You will receive:

```
00:00.0 PCI bridge: Device 16c3:abcd (rev 01) (prog-if 00 [Normal decode])
        Flags: bus master, fast devsel, latency 0
        Memory at 01000000 (32-bit, non-prefetchable) [size=1M]
        Bus: primary=00, secondary=01, subordinate=01, sec-latency=0
        I/O behind bridge: 00001000-00001fff
        Prefetchable memory behind bridge: 01100000-011fffff
        [virtual] Expansion ROM at 01200000 [disabled] [size=64K]
        Capabilities: [40] Power Management version 3
        Capabilities: [50] MSI: Enable- Count=1/1 Maskable+ 64bit+
        Capabilities: [70] Express Root Port (Slot-), MSI 00
        Capabilities: [100] Advanced Error Reporting
        Capabilities: [140] Virtual Channel
        Kernel driver in use: pcieport

01:00.0 Ethernet controller: Realtek Semiconductor Co., Ltd. RTL8111/8168/8411 PCI
Express Gigabit Ethernet Controller (rev 06)
        Subsystem: Realtek Semiconductor Co., Ltd. Device 0123
        Flags: bus master, fast devsel, latency 0, IRQ 308
        I/O ports at 1000 [size=256]
        Memory at 01104000 (64-bit, prefetchable) [size=4K]
        Memory at 01100000 (64-bit, prefetchable) [size=16K]
        Capabilities: [40] Power Management version 3
        Capabilities: [50] MSI: Enable- Count=1/1 Maskable- 64bit+
        Capabilities: [70] Express Endpoint, MSI 01
        Capabilities: [b0] MSI-X: Enable- Count=4 Masked-
        Capabilities: [d0] Vital Product Data
        Capabilities: [100] Advanced Error Reporting
        Capabilities: [140] Virtual Channel
        Capabilities: [160] Device Serial Number ef-ba-00-00-68-4c-e0-00
        Kernel driver in use: r8169
        Kernel modules: r8169
```

In this example the PCIe device is the *RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller*.

For PCIe devices you have to enable the correct driver in the kernel configuration. This Ethernet card, for example, is manufactured by *Realtek Semiconductor Co.* and the corresponding *Kconfig* option for the driver, which must be enabled, is named *CONFIG_R8169* and can be found under *Realtek 8169 gigabit ethernet support* in the kernel configuration.

- In order to activate the driver use:

  ```
  host$ bitbake virtual/kernel -c menuconfig
  ```

For some devices like this Ethernet card additional binary firmware blobs are needed. For example, if you try to bring up the network interface with

```
target$ ip link set up enp1s0
```

you will get the following output on the serial console:

```
[  620.116794] r8169 0000:01:00.0: Direct firmware load for rtl_nic/rtl8168e-3.fw failed
                                                                             with error -2
[     620.125943]   r8169   0000:01:00.0   enp1s0:   unable   to   load   firmware   patch
                                                                  rtl_nic/rtl8168e-3.fw (-2)
[ 620.163733] r8169 0000:01:00.0 enp1s0: link down
[ 620.168703] r8169 0000:01:00.0 enp1s0: link down
[ 620.178436] IPv6: ADDRCONF(NETDEV_UP): enp1s0: link is not ready
```

These firmware blobs have to be placed in */lib/firmware/* before the device can be used.

- E.g. with:

  ```
  host$ scp -r <firmware> root@192.168.3.11:/lib/firmware
  ```

Some PCIe devices, e.g. this Ethernet card, may function properly even if no firmware blob could be loaded from */lib/firmware/*and you received an error message as shown in the first line of the output above. This is because some manufacturers provide the firmware as fall-back on the card itself. In this case the behavior and output depends strongly on the manufacturer's firmware.

7.16 Audio

On Phytec i.MX 6 boards different audio chips are used. This section is written for the phyFLEX Carrier Board (PBA-B-01) with the TI TLV320AIC3007 audio codec, but it should be applicable for other boards and audio chips as well.

Audio support is done via the I^{2}S interface and controlled via I^{2}C.

- To check if your soundcard driver is loaded correctly and what the device is called type:
  ```
  target$ aplay -L
  ```
- Use *scp* to copy a wav file to the board and play it through the sound interface with:
  ```
  target$ aplay -vv file.wav
  ```

Not all wave formats are supported by the sound interface. Use *Audacity* on your host to convert any file into *44100:S16_LE* wave format which should be supported on all platforms.

- Run *speaker-test* to identify channel numbers:
  ```
  target$ speaker-test -c 2 -t wav
  ```
- An external audio source can be connected to the input, in order to record a sound file which can then be played back:
  ```
  target$ arecord -c 2 -r 441000 -f S16_LE test.wav
  target$ aplay test.wav
  ```
- To inspect your soundcards capabilities call:
  ```
  target$ alsamixer
  ```

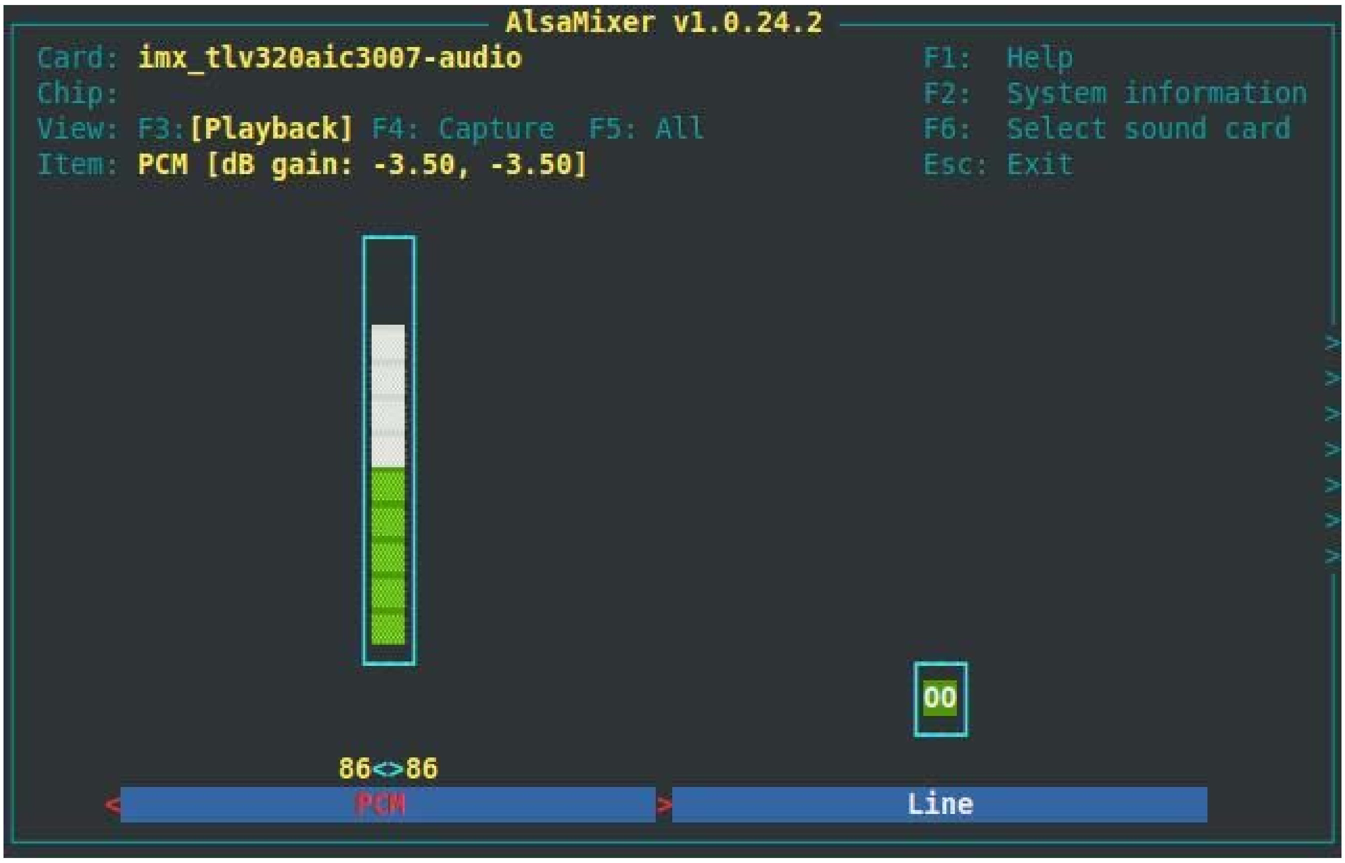
You should see a lot of options as the audio-ICs have many features you can play with. It might be better to open *alsamixer* via *ssh* instead of the serial console, as the console graphical effects could be better. You have either mono or stereo gain controls for all mix points. "*MM*" means the feature is muted, which can be toggled by hitting ***m***.

For more advanced audio usage, you need to have an audio server. This is required e.g. if you want to playback from different applications at the same time, e.g. you a have a notification app which makes a beep every now and then, plus you have a browser running which should be able to play sounds embedded in websites. The notification app has no possibility to open the sound device as long as the browser is running. The notifications will be suppressed. The standard sound server of *Linux* is *pulseaudio*. It is not installed per default at the moment, though.

7.16.1 Audio Sources and Sinks

The baseboard has four jacks for microphone, headset speaker, line in and line out.

Enabling and disabling input and output channels can be done with the *alsamixer* program. ***F3*** selects *Screen Playback* and ***F4*** *Screen Capture*. With the ***Tabulator*** key you can switch between these screens. To enable, or disable switchable controls press ***m*** (mute).



```
AlsaMixer v1.0.24.2
Card: imx_tlv320aic3007-audio                    F1:  Help
Chip:                                            F2:  System information
View: F3:[Playback] F4: Capture  F5: All         F6:  Select sound card
Item: PCM [dB gain: -3.50, -3.50]                Esc: Exit

                                                  OO
                      86<>86
                 <    PCM          >             Line
```

Figure 4: Screenshot of alsamixer

With the keys cursor left and cursor right you can step through the different channels. There are much more channels than fit onto one screen, so they will scroll if your cursor reaches the right or left edge of it. In case you get trouble in the display during scrolling, please use *ssh* instead of *microcom*.

alsamixer can be left by pressing the ***ESC*** key. The settings are saved automatically on shutdown and restored on boot by the systemd service *alsa-restore*. If you want to save the mixer settings manually you can execute *aslactl store*. The settings are saved in */var/lib/alsa/asound.state*.

7.16.2 Playback

To playback simple audio streams, you can use *aplay*. For example:

```
target$ aplay /usr/share/sounds/alsa/Front_Center.wav
```

The file formats *.ogg*, and *.flac* can be played back using *ogg123*. MP3 playback is currently not supported per default, because of licensing issues. If you are going to deliver a product including *.mp3* files, please check the royalty fees.

7.16.3 Capture

arecord is a command line tool for capturing audio streams which uses *Line In* as default input source.

To select a different audio source you can use *alsamixe*r. For examle, switch on *Right PGA Mixer Mic3R* and *Left PGA Mixer Mic3L* in order to capture the audio from the microphone input.

It is a known error that you need to choose *Playback screen* (***F3***) instead of *Capture screen* (***F4***) for accessing these two controls.

The following example will capture the current stereo input source with a sample rate of 48000 Hz and will create an audio file in WAV format (signed 16 bit per channel, 32 bit per sample):

```
target$ arecord -t wav -c 2 -r 48000 -f S16_LE test.wav
```

Capturing can be stopped again using ***CTRL-C***.

7.17 Framebuffer

This driver gains access to displays connected to Phytec carrier boards via device node */dev/fb0*.

The BSP is already prepared for use with the ETM0700G0DH6 (800 x 480) display, e.g. for the phyFLEX-i.MX 6 Kit. Other displays will be added in later BSP versions.

- To run a simple test of the framebuffer feature execute:
  ```
  fbtest
  ```

This will show various pictures on the display.

- Information about the framebuffer's resolution can be obtained with:
  ```
  fbset
  ```

which will return:

```
mode "800x480-0"
        # D: 0.000 MHz, H: 0.000 kHz, V: 0.000 Hz
        geometry 800 480 800 480 16
        timings 0 0 0 0 0 0 0
        accel true
        rgba 5/11,6/5,5/0,0/0
endmode
```

	fbset cannot be used to change display resolution or color depth. Depending on the framebuffer device different kernel commands are mostly needed to do this. Some documentation can be found in the kernel documentation at *https://www.kernel.org/doc/Documentation/fb/modedb.txt*. Please also refer to the manual of your display driver for more details.

In the following you will find some more useful commands.

- To query the color depth of the framebuffer emulation type:
  ```
  target$ cat /sys/class/graphics/fb0/bits_per_pixel
  ```

The result can be, for example:

```
16
```

- To get the name of the video interface execute:
  ```
  target$ ls /sys/class/drm/ -1
  ```

Possible output (e.g. for the phyFLEX-i.MX 6 mounted on the phyFLEX Carrier Board PBA-B-01)):

```
card0
 card0-HDMI-A-1
 card0-LVDS-1
 controlD64
 version
```

The suffixes of "card0-*" are the identifiers which must be used to configure the corresponding interface.

- To set, for example, the HDMI interface to 1024 x 768 pixels, and bits per pixel to 24 bpp, use the following kernel argument in the bootloader:

  ```
  bootloader$ nv linux.bootargs.video="video=HDMI-A-1:1024x768-24"
  bootloader$ saveenv
  ```

Ensure to use the correct *bits per pixel* for your display, e.g. 18, 16, or 24, ... Otherwise the colors will look very strange. Use *fbtest* to test them.

- An additional method to increase only the color depth of the framebuffer emulation with kernel arguments is:

  ```
  bootloader$ nv linux.bootargs.fb="imxdrm.legacyfb_depth=32"
  bootloader$ saveenv
  ```

- If you have connected multiple displays to your board, for example an LVDS and an HDMI display, you can disable one of them by setting:

  ```
  bootloader$ nv linux.bootargs.video="video=LVDS-1:d"
  bootloader$ saveenv
  ```

The display DT representation can be found in *imx6qdl-phytec-lcd.dtsi*. We have split the display configuration into two parts. In a generic display module *imx6qdl-phytec-lcd.dtsi* which includes the display DT representations for all displays implemented so far, and a board specific part, which can be found in all board DTs. The *imx6qdl-phytec-lcd.dtsi* file is included within the board DTs.

Board specific part, e.g. *arch/arm/boot/dts/imx6qdl-phytec-pfla02.dtsi*:

```
&iomuxc {
        imx6q-phytec-pfla02 {
                pinctrl_pwm1: pwm1grp {
                        fsl,pins = <
                                MX6QDL_PAD_DISP0_DAT8__PWM1_OUT   0x1b0b1
                        >;
                };
        };
};

&pwm1 {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_pwm1>;
        status = "okay";
};
```

Board specific part, e.g. *arch/arm/boot/dts/imx6qdl-phytec-pbab01.dtsi*:

```
&backlight {
        pwms = <&pwm1 0 5000000>;
        enable-gpios = <&gpio1 8 GPIO_ACTIVE_LOW>;
        brightness-levels = <0 4 8 16 32 64 128 255>;
};
```

7.17.1 Backlight Control

If a display is connected to the Phytec board, you can control its backlight with the *Linux* kernel sysfs interface. All available backlight devices in the system can be found in the folder */sys/class/backlight*.

- Read the appropriate files, and write to them to control the backlight. E.g. type:

  ```
  target$ cat /sys/class/backlight/backlight/max_brightness
  ```

  to query the max brightness level, e.g.:

  ```
  7
  ```

  Valid brightness values are 0 to <max_brightness>.

  ```
  target$ cat /sys/class/backlight/backlight/brightness
  ```

  will show the current brightness level in the driver, e.g.:

  ```
  2
  ```

  ```
  target$ echo 0 /sys/class/backlight/backlight/ brightness
  ```

  Set brightness to zero. The backlight is turned off.

  ```
  target$ echo 6 /sys/class/backlight/backlight/ brightness
  ```

  Set brightness to the second highest brightness level.

For documentation of all files see
https://www.kernel.org/doc/Documentation/ABI/stable/sysfs-class-backlight.

	The following note applies to the phyFLEX-i.MX 6 Embedded Imaging Kit and the phyFLEX-i.MX 6 Rapid Development Kit with display: If dimming does not work by writing to the file *brightness*, check the DIP switch settings on the bottom side of the display board (LCD-018). The correct setting of DIP switch *S1* is 1=OFF, 2=OFF, **3=ON** and 4=OFF.

7.17.2 Resistive Touchscreens

Most of the Phytec boards support connecting a resistive touchscreen which requires *tslib* support in general. But *tslib* support is also needed for the *Qt* framework for use of a resistive touchscreen, because *tslib* and *Qt5* do not work together out of the box.

- To set the right environment when starting a *Qt* application, start it with our *qtLauncher*:

  ```
  target$ qtLauncher QtDemo [optinal QtDemo parameters]
  ```

The launcher will check, whether the touchscreen is calibrated or not. If not, it will automatically start the calibration program before starting your application.

- Recalibration of the touchscreen can be done after closing the *Qt* application by executing:

  ```
  target$ ts_calibrate
  ```

7.18 CPU Core Frequency Scaling

The CPU of the i.MX 6 SoC is able to scale the clock frequency and the voltage. This is used to save power when the full performance of the CPU is not needed. Scaling the frequency and the voltage is referred to as 'Dynamic Voltage and Frequency Scaling' (DVFS).

The i.MX 6 BSP supports the DVFS feature. The *Linux* kernel provides a DVFS framework that allows each CPU core to have a min/max frequency and a governor that governs it. Depending on the i.MX 6 variant used several different frequencies are supported.

- Type
  ```
  target$ cat
      /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_frequencies
  ```
  to get a complete list.

In case you have for example an i.MX 6 with a maximum of approximately 1 GHz the result will be:
```
 396000 792000 996000
```
.

The voltages are scaled according to the setup of the frequencies.

You can decrease the maximum frequency (e.g. to 792000),
```
target$ echo 792000 >
                   /sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq
```
or increase the minimum frequency (e.g. to 792000)
```
target$ echo 792000 >
                   /sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq
```

- To ask for the current frequency type:
  ```
  target$ cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq
  ```

So called governors are selecting one of this frequencies in accordance to their goals, automatically.

- List all governors available with the following command:
  ```
  target$ cat
       /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_governors
  ```

The result will be:
```
conservative userspace powersave ondemand performance
```

conservative	is much like the *ondemand* governor. It differs in behavior in that it gracefully increases and decreases the CPU speed rather than jumping to max speed the moment there is any load on the CPU.

userspace	allows the user or user space program running as root to set a specific frequency (e.g. to 792000). Type: `target$ echo 792000 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed`
powersave	always selects the lowest possible CPU core frequency.
ondemand	switches between possible CPU core frequencies in reference to the current system load. When the system load increases above a specific limit it increases the CPU core frequency immediately. This is the default governor when the system starts up.
performance	always selects the highest possible CPU core frequency.

- In order to ask for the current governor, type:

```
target$ cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
```

You will normally get:

```
ondemand.
```

- Switching over to another governor (e.g. *userspace*) is done with:

```
target$ echo userspace >
                /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
```

For more detailed information about the governors refer to the *Linux* kernel documentation in: *linux/Documentation/cpu-freq/governors.txt.*

7.19 CPU Core Management

The i.MX 6 SoC can have multiple processor cores on the die. The i.MX 6 Quad, for example, has 4 ARM Cores which can be turned on and off individually at runtime.

- To see all available cores in the system execute:

```
target$ ls /sys/devices/system/cpu  -1
```

This will show for example:

```
cpu0  cpu1  cpu2  cpu3  cpufreq
[...]
```

Here the system has four processor cores.

By default all available cores in the system are enabled to get the maximum performance.

- To switch off a single core execute:

```
target$ echo 0 > /sys/devices/system/cpu/cpu3/online
```

As confirmation you will see:

```
[   69.380888] CPU3: shutdown
```

Now the core is powered down and no more processes are scheduled on this core.

- You can use *htop* to see a graphical overview of the cores and processes:

```
target$ htop
```

- To power up the core up again execute:

```
target$ echo 1 > /sys/devices/system/cpu/cpu3/online
```

Use of the kernel boot parameter *maxcpus* allows to limit the number of cores the system is using at runtime (*https://www.kernel.org/doc/Documentation/kernel-parameters.txt*).

7.20 Thermal Management

The *Linux* kernel has an integrated thermal management which is capable of monitoring SoC temperatures, reducing the CPU frequency, driving fans, advising other drivers to reduce the power consumption of devices and – in the worst case – shutting down the system gracefully (*https://www.kernel.org/doc/Documentation/thermal/sysfs-api.txt*).

This section describes how the thermal management kernel API is used for the i.MX 6 SoC platform.

The i.MX 6 has an internal temperature sensors for the SoC ("Temperature Monitor (TEMPMON)" in the i.MX 6 Reference Manual).

- The current temperature can be read in millicelsius with:

```
target$ cat /sys/class/thermal/thermal_zone0/temp
```

You will get for example:

```
77535
```

There are two trip points registered by the *imx_thermal* kernel driver:

1. *trip_point_0*: 85 °C type: *passive*
2. *trip_point_1*: 105 °C type: *critical*

(see kernel sysfs folder */sys/class/thermal/thermal_zone0/*)

These trip points are used by the kernel thermal management to trigger events and change the cooling behavior.

The following thermal policies (also named *thermal governors*) are available in the kernel: *Step Wise*, *Fair Share*, *Bang Bang* and *User space*.

The default policy used in the BSP is *step_wise*.

If the value of the SoC temperature in the sysfs file *temp* is above *trip_point_0* (greater than 85 °C) the CPU frequency is set to the lowest CPU frequency.
When the SoC temperature drops below *trip_point_0* again, the throttling is released.

If the SoC temperature reaches 105 °C, the thermal management of the kernel shuts down the systems. On the serial console you can see:

```
 kernel[194]: [  895.524255] thermal thermal_zone0: critical temperature reached(105
C),shutting down

 [  OK  ] Stopped target Sound Card.
 [  OK  ] Stopped target System Time Synchronized.
 [  OK  ] Stopped target Network.
          Stopping Autostart Qt 5 Demo...
 [...]
```

See *Linux* mainline patch: *HACK: thermal: imx: force maximum cooling above passive trip point*.

7.21 On-Chip OTP Controller (OCOTP_CTRL) – eFuses

The i.MX 6 provides one-time programmable fuses to store information such as the MAC address, boot configuration and other permanent settings ("On-Chip OTP Controller (OCOTP_CTRL)" in the i.MX 6 References Manual).

The following list is an abstract from the i.MX 6 Solo/DualLite Reference Manual and includes some useful fuse registers in the OCOTP_CTRL (at base address 0x21BC000).

- *OCOTP_CFG4*: addr 0x05, memory offset 0x450 (contains BOOT_CFG[1,2,3,4]. Transferred to register *SRC_SBMR1* 0x20D8004)
- *OCOTP_CFG5*: addr 0x06, memory offset 0x460 (contains BT_FUSE_SEL at the fifths bit. Some bits are transferred to register *SRC_SBMR2* 0x20D801C)
- *OCOTP_MAC0*: addr 0x22, memory offset 0x620 (contains bits 0-31 of MAC_ADDR)
- *OCOTP_MAC1*: addr 0x23, memory offset 0x630 (contains bits 32-47 of MAC_ADDR)

A complete list and a detailed mapping between the fuses in the OCOTP_CTRL and the boot/mac/... configuration is available in section "Fusemap" of the i.MX 6 Reference Manual.

7.21.1 Reading Fuse Values in Barebox

You can read the content of a fuse using memory mapped shadow registers.

To calculate the memory address use the *fuse address* (0x01-0x2f) in the following formula:

memory address = 0x21BC400 + <fuse address> * 0x10

- In the *barebox* use *md* to read the value. Examples:

  ```
  bootloader$ md 0x21BC450+4
  ```
  reads OCOTP_CFG4: addr 0x05
  ```
  bootloader$ md 0x21BC460+4
  ```
  reads OCOTP_CFG5: addr 0x06
  ```
  bootloader$ md 0x21BC620+4
  ```
  reads OCOTP_MAC0: addr 0x22
  ```
  bootloader$ md 0x21BC630+4
  ```
  reads OCOTP_ MAC1: addr 0x23

7.21.2 Burning Boot Configuration Fuses

⚠	Fuses can only be written once! You can brick your board easily, e.g. by burning a wrong boot configuration. I cannot be reversed!

Execute the following commands on the *barebox* prompt to burn the *BOOT_CFG1-4* values and enable reading the boot configuration from fuses. To select a boot device of your choice replace the *BOOT_CFG1-4* value with the appropriate value for your boot device (eMMC, SD, SPI, NAND, etc.).

- First enable write to fuses with:

  ```
  bootloader$ ocotp0.permanent_write_enable=1
  bootloader$ ocotp0.sense_enable=1
  ```
- Then check values:

  ```
  bootloader$ devinfo ocotp0
  ```
- Finally burn the fuses:

  ```
  bootloader$ mw -l -d /dev/imx-ocotp 0x14 0x58003283
  ```
  writes OCOTP_CFG4: BOOT_CFG1-4 Here: NAND 2Gb 64 pages per block (replace with the actual boot config of your board).
  ```
  bootloader$ mw -l -d /dev/imx-ocotp 0x18 0x00000010
  ```
  writes OCOTP_CFG5: Set bit BT_FUSE_SEL

Here is a short overview of different *BOOT_CFG1-4* values for various boards and boot configurations:

SRC_SBMR1 Register (0x20D8004)	Module and Boot configuration
0x58003283	NAND 2Gb 64 pages per block on phyFLEX-i.MX 6/phyCORE-i.MX 6
0x58005863	eMMC usdhc4 on phyCORE-i.MX 6
0x58002243	SD Card usdhc1 on phyCORE-i.MX 6

It is possible to read out a valid boot configuration from a board instead of deriving it from the i.MX 6 Reference Manual. For this purpose boot the board from the boot source you want to configure in the fuses. Press any key to stop *autoboot* and execute:

```
bootloader$ md 0x20D8004+4
```

```
020d8004: 58002243                         C".X
```

The command has read the value of register *SRC_SBMR1* (here 0x58002243) which represents the boot configuration *BOOT_CFG1-4*, and which can be written to the fuse register *OCOTP_CFG4* as described above.

7.21.3 Burning MAC Address

⚠	Fuses can only be written once! You can brick your board easily, e.g. by burning a wrong boot configuration. I cannot be reversed!

To burn the MAC address you do not have to blow the two fuse registers (OCOTP_MAC0 and OCOTP_MAC1) manually. The *barebox* ocotp driver has a special command for that. The following sequence shows how to use it. Replace the dummy MAC address in the following commands with a valid MAC address from your pool of addresses.

- First enable write to fuses with:

```
bootloader$ ocotp0.permanent_write_enable=1
bootloader$ ocotp0.sense_enable=1
```

- Then check values:

```
bootloader$ devinfo ocotp0
```

- Finally burn the MAC address:

```
bootloader$ ocotp0.mac_addr=11:22:33:44:55:66
```

7.22 WLAN Modules

7.22.1 Supported WiFi Modules and Software used

Currently there is only one module supported. The BSP described herein allows to run the TiWi-BLE *Bluetooth* and Wi-Fi combo module in client and Access Point (AP) mode using *WEP*, *WPA* and *WPA2* encryption. More information about the module can be found at *https://www.lsr.com/embedded-wireless-modules/wifi-plus-bluetooth-module/tiwi-ble*

This BSP is using the most current firmware from the linux-firmware repository.
git://git.kernel.org/pub/scm/linux/kernel/git/firmware/linux-firmware.git

The following binaries are used:
wl127x-fw-5-mr.bin (multi role)
wl127x-fw-5-sr.bin (single role)
wl127x-fw-5-plt.bin (used for the calibration process)

The *Linux* kernel driver will select the right firmware.

We are using a self generated *nvs* binary file to allow the TiWi-BLE combo module to operate in compliance with the modular certification for *FCC* and *ETSI*. We have also set the MAC in the *nvs* file to 00:00:00:00:00:00. This causes the driver to use the MAC from the BT BD address that is burned into the wl127x chip.

The *wl1271-nvs.bin* can be generated by using TI's calibrator tool and our *ini* file in */usr/share/wl127x-inis/tiwi-ble-fcc-etsi.ini.*

Please see section *7.22.3 "Calibration"*, or
http://www.lsr.com/downloads/products/330-0078.pdf (chapter 2.2) for more information.

7.22.2 Enable WiFi Expansion Boards

The current BSP supports two expansion boards with the TiWi-BLE *Bluetooth* and Wi-Fi combo module. The supported expansion boards are:

Expansion Board article #	Description
PEB-B-001	Expansion board suitable for phyBOARD-Subra i.MX 6
PEB-WLBT-01	Expansion board suitable for phyBOARD-Mira i.MX 6

To enable the WiFi expansion board on the phyBOARD-Subra, perform the following steps:

- Copy special device tree *zImage-imx6dl-phytec-phyboard-subra-wifi.dtb* (for the i.MX6 DualLite/Solo), or *zImage-imx6q-phytec-phyboard-subra-wifi.dtb* (for the i.MX6 DualLite) into the TFTP server root directory. Then flash the kernel and rootfs as described in *section "Updating NAND Flash from Network"*. After that erase the *oftree* partition with:

  ```
  bootloader$ erase /dev/nand0.oftree.bb
  ```
- Then flash the device tree:

  ```
  bootloader$ cp -v /mnt/tftp/zImage-imx6dl-phytec-phyboard-subra-
                                         wifi.dtb /dev/nand0.oftree.bb
  ```
- Now, boot the board from NAND:

  ```
  bootloader$ boot nand
  ```

⚠	On the phyBOARD-Subra the WiFi expansion board (PEB-B-001) cannot be used with the SD card interface at the same time. You have to boot from NAND Flash. For the correct orientation of the expansion board consult the corresponding hardware manual.

To enable the WiFi expansion board with the phyBOARD-Mira, perform the following steps:

- Start the board and stop in the bootloader shell.
- Then edit the file */env/config-expansion* with:

  ```
  bootloader$ edit /env/config-expansions
  ```
- In the file remove the '#' charachter in the line:

  ```
  . /env/expansions/imx6qdl-phytec-peb-wlbt-01
  ```
- After that press ***CTRL+D*** and save the environment with:

  ```
  bootloader$ saveenv
  ```
- Now, reboot the board:

  ```
  bootloader$ boot
  ```

⚠	To use the expansion board PEB-WLBT-01 on the phyBOARD-Mira hardware modifications are required. These are implemented on the phyBOARD-Mira with article number *PB-01501-1201I*.

After enabling the WiFi module on the expansion board and booting the kernel, you should see an output on your console similar to the following:

```
cfg80211: Calling CRDA to update world regulatory domain
wlcore: loaded
wlcore: firmware booted (Rev 6.3.10.0.133)
```

7.22.3 Calibration

Calibration is a process of determining radio parameters for a specific chip. These configuration parameters are suitable to the specific chip with its unique design. Therefore, the calibration parameters are sent back to the driver to be stored in non-volatile memory for later use. Upon initialization, the wL12xx driver loads an *nvs* file, where it expects to read those calibration parameters to send them to the chip. The *nvs* file includes two main parts: one stores the calibration parameters and the other stores the initialization information required for the wL12xx driver. The calibration is described also in *http://processors.wiki.ti.com/index.php/WL12xx_NLCP_Calibration_Process* and *http://linuxwireless.org/en/users/Drivers/wl12xx/calibrator/#wl12xx_Calibration*.

TI provides a calibration tool which can be found in the 18xx-ti-utils repository:
git://git.ti.com/wilink8-wlan/18xx-ti-utils.git

The calibrator version used is 0.80.

- To start the calibration first generate an *nvs* reference file:

```
target$ calibrator set ref_nvs /usr/share/wl127x-inis/tiwi-ble-fcc-
                                                      etsi.ini
target$ cp new-nvs.bin /lib/firmware/ti-connectivity/wl1271-nvs.bin
target$ ifconfig wlan0 down
target$ rmmod wlcore_sdio
target$ modprobe wlcore_sdio
```

- Now perform the first calibration:

```
target$ calibrator plt calibrate
```

- Copy the newly created file to the proper location:

```
target$ cp new-nvs.bin /lib/firmware/ti-connectivity/wl1271-nvs.bin
```

- Finally set the WLAN MAC address:

```
target$ calibrator set nvs_mac /lib/firmware/ti-connectivity/wl1271-
                                   nvs.bin 00:00:00:00:00:00
```

8 Customizing the BSP

8.1 Changing MTD Partitions

For Memory Technology Devices (MTD) like NAND Flash or SPI NOR Flash the partitions are usually defined in the device trees, i.e. they are defined in the device tree of the MLO, the *barebox* and the kernel. When changing the partition table all those parts need to be touched. Newer *barebox* versions (v2015.07.0 and newer) have a mechanism to overwrite partitions at runtime.

The *barebox* holds an internal list with partitions which is initialized with the partition table out of the *barebox* device tree. This list is used later on to fix up the device tree of the kernel and can be overwritten in the *barebox* shell, or with a script before booting the kernel.

- To print the partitions just type:

  ```
  bootloader$ echo $<mtddevice>.partitions
  ```

Example:

```
bootloader$ echo $nand0.partitions
4M(barebox),1M(barebox-environment),1M(oftree),8M(kernel),1010M(root)
```

- To overwrite the partitions just change the partitions variable:

  ```
  bootloader$ m25p0.partitions=1M(barebox),128k(barebox-
                              environment),128k(oftree),5104k(kernel)
  ```

Adding and deleting partitions by overwriting the partitions variable is possible. But do **not** touch the *barebox* and *bareboxenv* partitions. Those should **not** be changed at runtime.

9 Revision History

Date	Version #	Changes in this manual
17.09.2015	Manual L-814e_1	First edition. Describes the Phytec BSP release PD15.2.0 for i.MX 6 and i.MX 6 products.
17.05.2016	Manual L-814e_2	Second edition. Describes the Phytec BSP release PD15.3.x with Yocto 1.8.1 for i.MX 6 and i.MX 6 products.

Document: **Yocto i.MX 6 BSP Manual**
Document number: **L-814e_2, May 2016**

How would you improve this manual?

Did you find any mistakes in this manual? **page**

Submitted by:

Customer number:	
Name:	
Company:	
Address:	

Return to:

PHYTEC Messtechnik GmbH
Postfach 100403
D-55135 Mainz, Germany
Fax : +49 (6131) 9221-33

Published by

© PHYTEC Messtechnik GmbH 2016

Ordering No. L-814e_2
Printed in Germany