Application Note No. LAN-071e_0
Version: Preliminary
Author: R. Stidronski
Date: 24.10.2016

Historie:

Version	Changes	Date	Author
Preliminary	First draft	24.10.2016	R. Stidronski

phyBOARD®-Segin i.MX 6UL Alpha Kit Getting Started

Content

1 Introduction

This Quickstart describes the tools and provides the know-how to install and work with the *Linux* Board Support Package (BSP) for the phyBOARD-Segin i.MX 6UL Alpha Kit. This Quickstart shows you how to do everything from installing the appropriate tools and sources, to building custom kernels, to deploying the OS in order to operate the software and hardware.

2 Requirements

The following system requirements are necessary to successfully complete this Quickstart. Deviations from these requirements may suffice, or may have other workarounds.

Hardware:

- phyBOARD-Segin (PB-02013) Single Board Computer (SBC)
- phyCORE-i.MX 6UL (PCL-063) already soldered on the phyBOARD-Segin
- Evaluation Adapter Board(PEB-EVAL-01)
- Power Adapter Board (PEB-POW-01)
- Serial cable (RS-232)
- AC adapter supplying 12 V – 24 V DC/ min. 2 A

Software:

- A modern *Linux* Operating host system either natively or via a virtual machine:
  - *Ubuntu 14.04 LTS* 64-bit recommended. Other distributions will likely work, but note that some setup information as well as OS specific commands and paths may differ.
  - If you want to use a virtual machine, VMWare Workstation, VMWare Player, and VirtualBox are possible solutions.
- Root access to your *Linux* host PC. Some commands in the Quickstart will not work if you do not have *sudo* access (e.g. package installation, formatting an SD card).
- At least 40 GB to 50 GB free on the build partition of the host PC.
- An SD card reader operational under *Linux*.
  - If you do not have SD card access under *Linux* on your host PC then formatting, copying the bootloader, and mounting the root file system on an SD card will not be possible.
- An active internet connection.

3 Getting Started with the included SD Card

This section is designed to get the board up and running with the SD Card included in the kit, which is prepared with the pre-built images.

3.1 Connector Interfaces

The following picture shows the phyBOARD-Segin Alpha Kit which is used in this Quickstart and highlights the location of the different interfaces on the board.

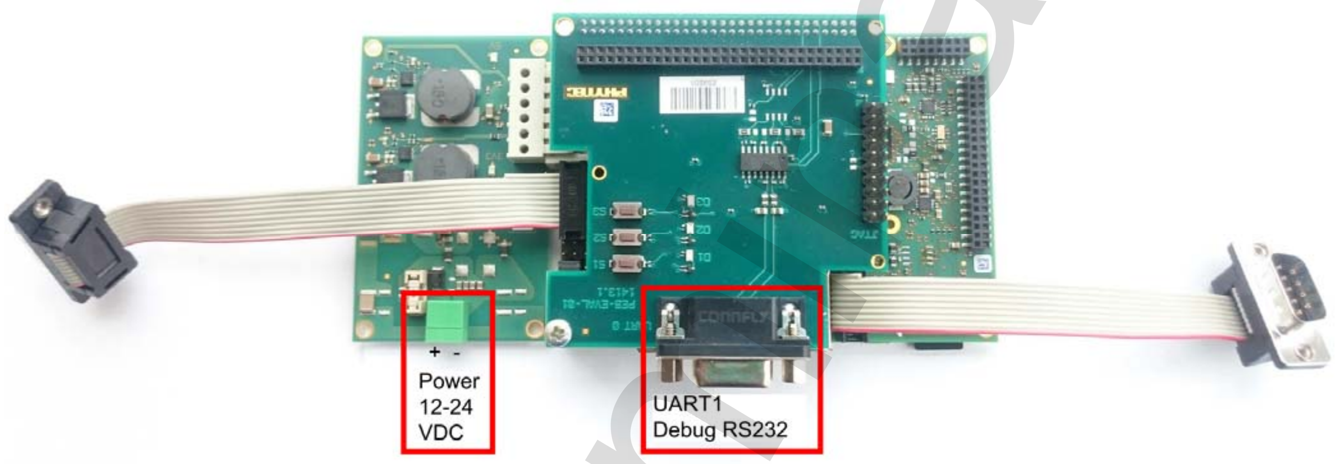


Figure 1: phyBOARD-Segin Alpha Kit

An overview of additional connectors and interfaces of the phyBOARD-Segin can be found in the next picture.

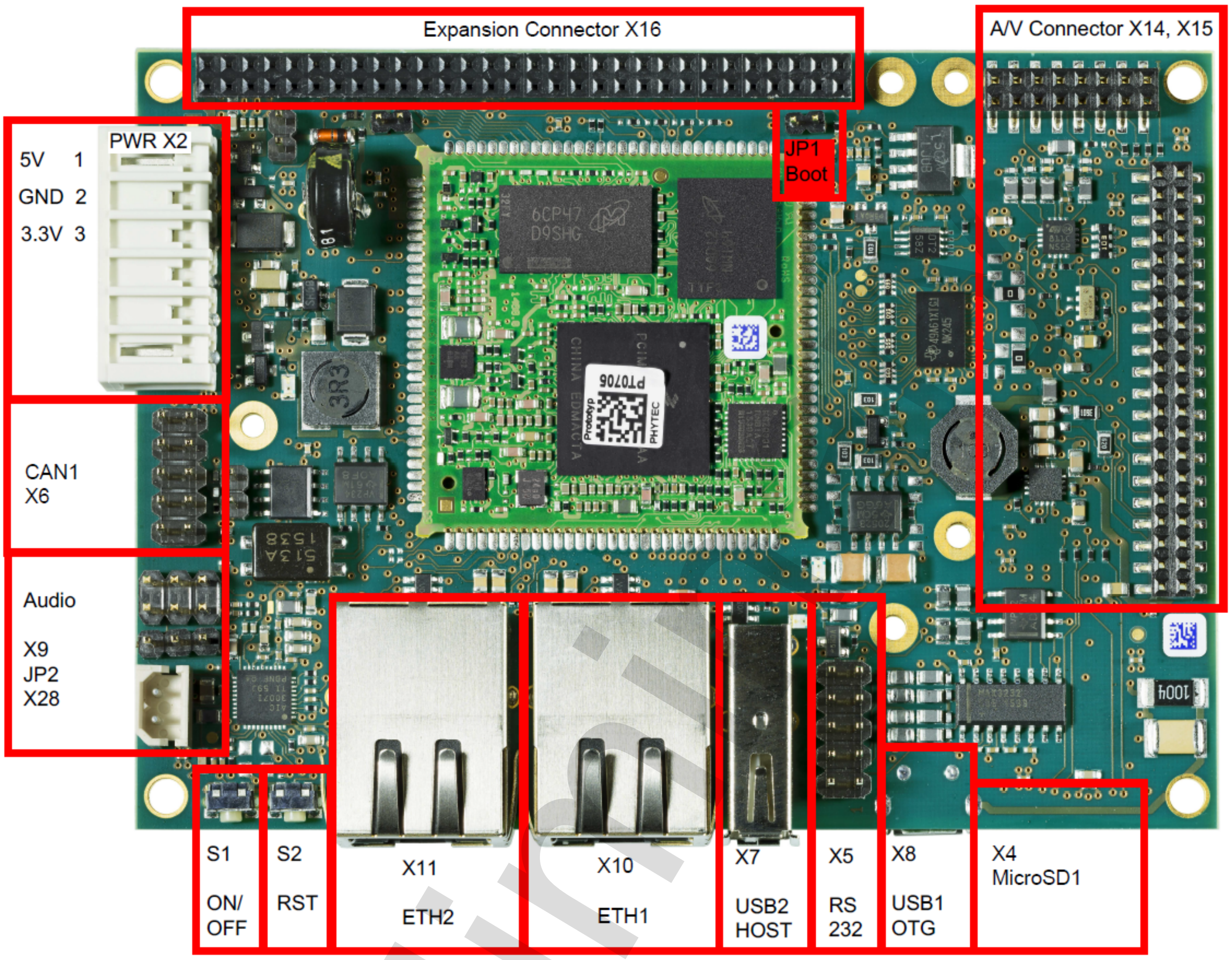


Figure 2: phyBOARD-Segin i.MX 6

3.2 Booting from the SD Card

This section describes how to boot the phyBOARD-Segin with the pre-built images on the included SD card.

- Insert the micro SD card into the SD card slot on the bottom side of board.
- Connect an RS-232 cable to the Evaluation Adapter Board. The UART1 interface (Debug interface) is already at RS-232 level.
- Start your favorite terminal software (e.g. *Minicom*,or *Tera Term*) on your host PC and configure it for 115200 baud, 8 data bits, no parity, and 1 stop bit (8n1) with no handshake.
- Ensure that boot mode jumper JP1 is closed
- Connect the power supply to the Power Adapter Board (please note the polarity shown in *Figure 1*).

The Power LEDs on the Power Adapter Board as well as the Powerinput LED on the phyBOARD-Segin will light up and the board starts booting into *Linux*. The console output can be viewed in your terminal window. If everything was done correctly the login prompt will be shown at the end of the booting process:

```
Yogurt (Phytec Example Distribution) phyBOARD-Segin-i.MX6UL-ALPHA1 phyboard-segin-imx6ul-1 ttymxc0

phyboard-segin-imx6ul-1 login:
```

The default login account is *root* with an empty password.

4 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 *U-Boot* 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 UL kernel is based on the *Linux* kernel. The kernel repository can be found at:
git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.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*

4.1 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
```

4.2 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"

4.3 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 PDphyBOARD-Segin-i.MX6UL-ALPHA1
```

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

4.4 Starting the Build Process

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

4.5 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, *phyboard-segin-imx6ul-1* machine:

- *Barebox*: barebox.bin
- *Barebox* configuration: barebox-defconfig
- Kernel: zImage
- Kernel device tree file: zImage-imx6ul-phytec-phyboard-segin-ff-rdk.dtb
- Kernel configuration: zImage.config
- Root filesystem: phytec-headless-image-phyboard-segin-imx6ul-1.tar.gz,
  phytec-headless-image-phyboard-segin-imx6ul-1.ubifs,
  phytec-headless-image-phyboard-segin-imx6ul-1.ext4
- SD card image: phytec-headless-image-phyboard-segin-imx6ul-1.sdcard

5 Booting the System

The default boot source for the i.MX 6 modules like phyCORE-i.MX 6 UL is the SD card. The easiest way to get started with your freshly created images, is writing them to an SD card and setting the boot configuration accordingly (JP1 closed).

⚠	Please ensure that JP1 is closed, as the Alpha Kit does not support booting from NAND.

5.1 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)

5.1.1 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/BSP-Yocto-i.MX6/*.

⚠	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.

5.1.2 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. *fdisk* will return the new values as acknowledgement. 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
  ```

5.1.3 Booting from USB OTG (Serial Downloader)

The i.MX 6UL ROM code is capable of downloading a bootloader from the USB OTG interface ("Serial Downloader" in the i.MX 6UL 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.

- 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-phycore-
                                                  imx6ul-512mb.img
```

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

5.2 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.

5.2.1 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*.

5.2.2 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*.

5.2.3 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 default
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.

5.3 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*.

6 Updating the Software

In this chapter we explain how to use the *barebox* bootloader to update the images in the NAND Flash.

6.1 Updating from Network

i.MX 6UL 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.

6.1.1 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 create UBI volumes for *Linux* kernel, *oftree* and root filesystem in NAND:

```
bootloader$ ubiformat /dev/nand0.root
bootloader$ ubiattach /dev/nand0.root
bootloader$ ubimkvol -t static /dev/nand0.root.ubi kernel 8M
bootloader$ ubimkvol -t static /dev/nand0.root.ubi oftree 1M
bootloader$ ubimkvol -t dynamic /dev/nand0.root.ubi root 0
```

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

```
bootloader$ ubiupdatevol /dev/nand0.root.ubi.kernel /mnt/tftp/zImage
bootloader$ ubiupdatevol /dev/nand0.root.ubi.oftree /mnt/tftp/oftree
```

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

```
bootloader$ cp –v /mnt/tftp/root.ubifs /dev/nand0.root.ubi.root
```

6.2 Updating from SD Card

To update an i.MX 6UL 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.

6.2.1 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$ ubiformat /dev/nand0.root
bootloader$ ubiattach /dev/nand0.root

bootloader$ ubimkvol -t static /dev/nand0.root.ubi kernel 8M
bootloader$ ubimkvol -t static /dev/nand0.root.ubi oftree 1M
bootloader$ ubimkvol -t dynamic /dev/nand0.root.ubi root 0

bootloader$ ubiupdatevol /dev/nand0.root.ubi.kernel /mnt/mmc/zImage
bootloader$ ubiupdatevol /dev/nand0.root.ubi.oftree /mnt/mmc/oftree
```

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

7 Device Tree (DT)

7.1 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

7.2 Phytec phyBOARD-Segin i.MX 6UL BSP Device Tree Concept

This BSP is in an alpha state. The DT concept is under development.

7.2.1 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.

Here is a simple example on how to enable the *imx6ul-phytec-segin-peb-av-02* (Display Expansion Board) on the phyBOARD-Segin 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/imx6ul-phytec-segin-peb-av-02
```

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

- To enable *imx6ul-phytec-segin-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-Segin:

  ```
  #!/bin/sh
  . /env/expansions/imx6ul-phytec-segin-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.

imx6ul-phytec-segin-peb-av-02:

```
of_fixup_status /soc/aips-bus@02100000/lcdif@021c8000/
of_fixup_status /soc/aips-bus@02100000/lcdif@021c8000/display@di0
of_fixup_status /backlight
of_fixup_status /soc/aips-bus@02100000/i2c@021a0000/edt-ft5x06@38
of_fixup_status /soc/aips-bus@02000000/pwm@02088000/
```

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

7.2.2 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*:

  ```
  # imx6ul-phytec-lcd: 7" display
  #of_display_timings -S /soc/aips-
    bus@02100000/lcdif@021c8000/display@di0/display-timings/ETM0700G0EDH6

  # imx6ul-phytec-lcd: 5.7" display
  #of_display_timings -S /soc/aips-
    bus@02100000/lcdif@021c8000/display@di0/display-timings/ETMV570G2DHU

  # imx6ul-phytec-lcd: 4.3" display
  #of_display_timings -S /soc/aips-
    bus@02100000/lcdif@021c8000/display@di0/display-timings/ETM0430G0DH6

  # imx6ul-phytec-lcd: 3.5" display
  #of_display_timings -S /soc/aips-
    bus@02100000/lcdif@021c8000/display@di0/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 *imx6ul-phytec-segin-peb-av-02-res* board, change the *env/config-expansions* from:

```
#!/bin/sh
. /env/expansions/imx6ul-phytec-segin-peb-av-02
```

to:

```
#!/bin/sh
#. /env/expansions/imx6ul-phytec-segin-peb-av-02
. /env/expansions/imx6ul-phytec-segin-peb-av-02-res
```

8 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 6UL 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).

Please read also *section 7 "Device Tree (DT)"* to get an understanding of our unified i.MX 6UL BSP device tree model.

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

8.1 i.MX 6UL Pin Muxing

The i.MX 6UL 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 6UL 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 NXP i.MX 6UL Reference Manuals for more information about the specific pins and the muxing capabilities:
http://www.nxp.com/products/microcontrollers-and-processors/arm-processors/i.mx-applications-processors/i.mx-6-processors/i.mx6qp/i.mx-6ultralite-processor-low-power-secure-arm-cortex-a7-core:i.MX6UL?fpsp=1&tab=Documentation_Tab

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 UART1 device in *imx6ul-phytec-phycore-som.dtsi*:

```
pinctrl_uart1: uart1grp {
        fsl,pins = <
                MX6UL_PAD_UART1_TX_DATA__UART1_DCE_TX     0x1b0b1
                MX6QUL_PAD_UART1_RX_DATA__UART1_DCE_RX    0x1b0b1
         >;
};
```

The first part of the string `MX6UL_PAD_UART1_TX_DATA__UART1_DCE_TX` names the pad (e.g. *PAD_UART1_TX_DATA*). The second part of the string (here *UART1_DCE_TX*) 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,imx6ul-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/imx6ul-pinfunc.h

8.2 Serial TTYs

The i.MX 6UL SoC provides up to 8 so called UART units. Phytec boards support different numbers of these UART units.

The phyBOARD-Segin uses UART0 (ttymxc0) as standard console output. It is configured as 115200 8N1 (115200 baud, 8 data bits, no parity bit, 1 stop bit). The other UARTs will come up with default settings, which is normally 9600 baud.

- From the command line prompt of *Linux* user space you can easily check the availability of other UART interfaces with:
  ```
  target$ echo "test" > /dev/ttymxc4
  ```
  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 *ttymxc0* (the standard console of the phyBOARD-Segin) to *ttymxc4*.

- First get the current parameters with:
  ```
  target$ stty -F /dev/ttymxc0 -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/ttymxc4
  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
  ```
  or
  ```
  target$ stty -F /dev/ttymxc4 $(stty -g < /dev/ttymxc0)
  ```

8.3 Network

The phyBOARD-Segin features two 10/100 Mbit Ethernet interfaces. The interface offers 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 and eth1 are:

eth0: 192.168.3.11/24
eth1: 192.168.4.11/24

8.4 CAN Bus

The phyBOARD-Segin i.MX 6UL 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 6UL'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.

8.5 MMC/SD Card

The phyBOARD-Segin i.MX 6UL supports a slot for Secure Digital Cards and Multi Media Cards to be used as general purpose block devices. 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.

8.6 NAND Flash

The phyCORE-i.MX 6UL on the phyBOARD-Segin is 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 6UL. 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 *9.1 "Changing MTD Partitions"*.

Adding new partitions can be done by creating a new partition node in the corresponding board device tree. 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.

8.7 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. 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 0 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:

- To get all LEDs available, type:
  ```
  target$ ls /sys/class/leds
  ```
  ```
  mmc0::       phycore:green   user_led_red   user_led_yellow
  ```
- To toogle the LEDs use
  ```
  target$ echo 255 > /sys/class/leds/user_led_red/brightness
  ```
  to turn it ON, and
  ```
  target$ echo 0 > /sys/class/leds/user_led_red/brightness
  ```
  to turn it OFF.

user_led-red and *user_led_yellow* represent the LEDs on the Evaluation Board (PEB-EVAL-01).

8.8 I^{2}C Bus

The i.MX 6UL contains four multimaster fast-mode I^{2}C modules called I2C1, I2C2, I2C3, and I2C4. On the phyBOARD-Segin only I2C1 is supported and connects to different I^{2}C devices on the phyCORE i.MX 6UL as well as on the phyBOARD-Segin. This chapter will describe the basic device usage of some of the I^{2}C devices.

8.8.1 EEPROM

It is possible to read and write to the device directly in
```
target$ /sys/class/i2c-dev/i2c-0/device/0-0052/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/0-0052/eeprom bs=1
                                                   count=1024  | od -x
  ```
- E.g. to fill the whole EEPROM with zeros use:
  ```
  target$ dd if=/dev/zero of=/sys/class/i2c-dev/i2c-0/device/0-
                                           0052/eeprom bs=4096 count=1
  ```

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

8.8.2 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:

```
rv4162
20cc000.snvs:snvs-r
```

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

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*.

8.8.3 Capacitive Touchscreen

The capacitive touchscreen is a part of the display module.

- For a simple test of the basic input handling of the touchscreen use *evtest* after selecting an input device:
  ```
  target$ evtest
  ```
  The raw touch input events will be displayed.

8.9 USB Host Controller

The USB controller of the i.MX 6UL 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 two independent USB controller cores.

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.

8.10 USB OTG

The phyBOARD-Segin provides one 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*.

8.10.1 Using the Board as USB Device

In order to connect the board as USB device to an USB host port (for example a PC), you need to configure the appropriate USB gadget. With USB *configfs* you can define the parameters and functions of the USB gadget. The BSP includes USB *configfs* support as kernel module.

- To load the module execute:

```
target$ modprobe libcomposite
```

Example:

- To define parameters such as the USB vendor and product IDs and to set the information strings for the english (0x409) language execute:

```
target$ cd /sys/kernel/config/usb_gadget/
target$ mkdir g1
target$ cd g1/
target$ echo "0x1d6b" > idVendor
target$ echo "0x0104" > idProduct
target$ mkdir strings/0x409
target$ echo "0123456789" > strings/0x409/serialnumber
target$ echo "Foo Inc." > strings/0x409/manufacturer
target$ echo "Bar Gadget" > strings/0x409/product
```

- Next create a file for the mass storage gadget with:

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

- Now create the functions you want to use:

```
target$ cd /sys/kernel/config/usb_gadget/g1
target$ mkdir functions/acm.GS0
target$ mkdir functions/ecm.usb0
target$ mkdir functions/mass_storage.0
target$ echo /tmp/file.img > functions/mass_storage.0/lun.0/file
```

acm:	Serial gadget, creates a serial interface like */dev/ttyGS0*
ecm:	Ethernet gadget, creates an Ethernet interface, e.g. *usb0*
mass_storage:	The host can partition, format and mount the gadget mass storage the same way as any other USB mass storage

- Bind the newly defined functions to a configuration with:

```
target$ cd /sys/kernel/config/usb_gadget/g1
target$ mkdir configs/c.1
target$ mkdir configs/c.1/strings/0x409
target$ echo "CDC ACM+ECM+MS" >
configs/c.1/strings/0x409/configuration
target$ ln -s functions/acm.GS0 configs/c.1/
target$ ln -s functions/ecm.usb0 configs/c.1/
target$ ln -s functions/mass_storage.0 configs/c.1/
```

- To start the USB gadget first get the correct name of the driver with:

```
target$ cd /sys/kernel/config/usb_gadget/g1
target$ ls /sys/class/udc/
```

which will result in:

```
ci_hdrc.0
```

- Finally start the USB gadget:

```
target$ echo "ci_hdrc.0" >UDC
```

- To stop the USB gadget and unbind the functions used execute:

```
target$ echo "" > /sys/kernel/config/usb_gadget/g1/UDC
```

8.11 Audio

On Phytec boards different audio chips are used. This section is written for the phyBOARD-Segin i.MX 6 (PBA-CD-10) 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:

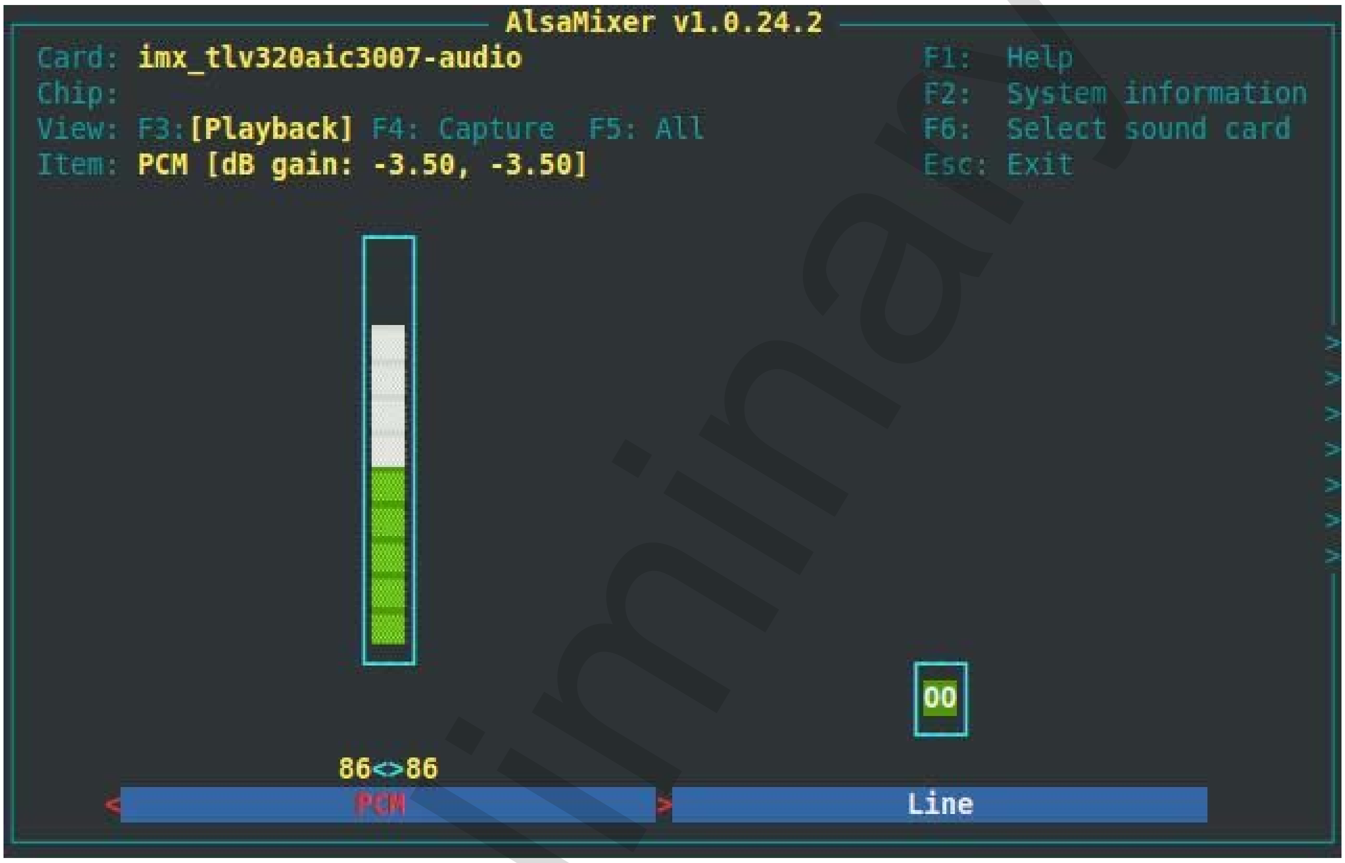
  ```
  target$ alsamixer
  ```

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.

8.11.1 Audio Sources and Sinks

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

                                                  00
           86<>86
         <   PCM   >              Line
```

Figure 3: *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 *alsactl store*. The settings are saved in */var/lib/alsa/asound.state*.

8.11.2 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.

8.11.3 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 *alsamixer*. For example, 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***.

8.12 Framebuffer

This driver gains access to a display connected to the carrier board via device node */dev/fb0*.

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

```
target$ fbtest
```

This will show various pictures on the display.

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

```
target$ fbset
```

which will return:

```
mode "800x480-60"
        # D: 33.265 MHz, H: 31.501 kHz, V: 60.001 Hz
        geometry 800 480 800 480 32
        timings 30062 88 40 33 10 128 2
        accel false
        rgba 8/16,8/8,8/0,0/0
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.

Another useful command allows to query the color depth.

- 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:

```
32
```

8.12.1 Backlight Control

If a display is connected to the phyBOARD-Segin, 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*. Reading the appropriate files, and writing to them allows to control the backlight.

- To get, for example, the maximum brightness level (max_brightness) execute

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

  which will result in:

  ```
  7
  ```

  Valid values for the brightness level are 0 to <max_brightness>.

- To obtain the current brightness level type:

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

  You will get for example:

  ```
  2
  ```

- Write to the file *brightness* to change the brightness. E.g.,

  ```
  target$ echo 0 > /sys/class/backlight/backlight/ brightness
  ```
  turns the backlight off,

  ```
  target$ echo > 6 /sys/class/backlight/backlight/ brightness
  ```
  sets the brightness to the second highest brightness level.

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

8.12.2 Resistive Touchscreens

The phyBOARD-Segin supports connecting a resistive touchscreen which requires *tslib* support in general.

- Calibration of the touchscreen can be done by executing:
  ```
  target$ ts_calibrate
  ```

- Testing the touchscreen can be done with:
  ```
  target$ ts_test
  ```

9 Customizing the BSP

9.1 Changing MTD Partitions

For Memory Technology Devices (MTD) such as NAND Flash or SPI NOR Flash the partitions are usually defined in the device trees, i.e. they are defined in 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
```
can, for example, result in:

```
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.

Please contact our technical support, if you need additional information, or if you have any questions.

Europe (except France):	**France:**
▪ +49 6131 9221-31	▪ +33 2 43 29 22 33
▪ *support@phytec.de*	▪ *support@phytec.fr*
North America:	**India:**
▪ +1 206 780-9047, or +1 800 278-9913	▪ +91-80-4086 7047
▪ *support@phytec.com*	▪ *support@phytec.in*
China:	
▪ +86-755-6180-2110	
▪ support@phytec.cn	