Yocto Reference Manual Mickledore

PHYTEC Messtechnik GmbH

Nov 18, 2024

CONTENTS

Yocto Reference Manual	
Document Title	Yocto Reference Manual Mickledore
Document Type	Yocto Manual
Release Date	XXXX/XX/XX
Is Branch of	Yocto Reference Manual

Compatible BSPs	BSP Release Type	BSP Release Date	BSP Status
BSP-Yocto-NXP-i.MX93-PD24.1.0	Major	05.02.2024	released
BSP-Yocto-NXP-i.MX93-PD24.1.1	Minor	08.05.2024	released

This manual applies to all Mickledore based PHYTEC releases.

CHAPTER

ONE

PHYTEC DOCUMENTATION

PHYTEC will provide a variety of hardware and software documentation for all of our products. This includes any or all of the following:

- **QS Guide**: A short guide on how to set up and boot a phyCORE board along with brief information on building a BSP, the device tree, and accessing peripherals.
- **Hardware Manual**: A detailed description of the System on Module and accompanying carrier board.
- **Yocto Guide**: A comprehensive guide for the Yocto version the phyCORE uses. This guide contains an overview of Yocto; introducing, installing, and customizing the PHYTEC BSP; how to work with programs like Poky and Bitbake; and much more.
- **BSP Manual**: A manual specific to the BSP version of the phyCORE. Information such as how to build the BSP, booting, updating software, device tree, and accessing peripherals can be found here.
- **Development Environment Guide**: This guide shows how to work with the Virtual Machine (VM) Host PHYTEC has developed and prepared to run various Development Environments. There are detailed step-by-step instructions for Eclipse and Qt Creator, which are included in the VM. There are instructions for running demo projects for these programs on a phyCORE product as well. Information on how to build a Linux host PC yourself is also a part of this guide.
- **Pin Muxing Table**: phyCORE SOMs have an accompanying pin table (in Excel format). This table will show the complete default signal path, from processor to carrier board. The default device tree muxing option will also be included. This gives a developer all the information needed in one location to make muxing changes and design options when developing a specialized carrier board or adapting a PHYTEC phyCORE SOM to an application.

On top of these standard manuals and guides, PHYTEC will also provide Product Change Notifications, Application Notes, and Technical Notes. These will be done on a case-by-case basis. Most of the documentation can be found in the applicable download page of our products.

CHAPTER

TWO

YOCTO INTRODUCTION

Yocto is the smallest SI metric system prefix. Like milli equates to `m = 10^-3`, and so is yocto `y = 10^-24`. Yocto is also a project working group of the Linux Foundation and therefore backed up by several major companies in the field. On the Yocto Project website you can read the official introduction:

> The Yocto Project is an open-source collaboration project that provides templates, tools, and methods to help you create custom Linux-based systems for embedded products regardless of the hardware architecture. It was founded in 2010 as a collaboration among many hardware manufacturers, open-source operating systems vendors, and electronics companies to bring some order to the chaos of embedded Linux development.

As said, the project wants to provide toolsets for embedded developers. It builds on top of the long-lasting OpenEmbedded project. It is not a Linux distribution. But it contains the tools to create a Linux distribution specially fitted to the product requirements. The most important step in bringing order to the set of tools is to define a common versioning scheme and a reference system. All subprojects have then to comply with the reference system and have to comply with the versioning scheme.

The release process is similar to the Linux kernel. Yocto increases its version number every six months and gives the release a codename. The release list can be found here: https://wiki.yoctoproject.org/wiki/Releases

CHAPTER

THREE

CORE COMPONENTS

The most important tools or subprojects of the *Yocto* Project are:

- Bitbake: build engine, a task scheduler like make, interprets metadata
- OpenEmbedded-Core, a set of base layers, containing metadata of software, no sources
- Yocto kernel
  - Optimized for embedded devices
  - Includes many subprojects: rt-kernel, vendor patches
  - The infrastructure provided by Wind River
  - Alternative: classic kernel build → we use it to integrate our kernel into *Yocto*
- *Yocto* Reference BSP: *beagleboneblack*, *minnow max*
- *Poky*, the reference system, a collection of projects and tools, used to bootstrap a new distribution based on *Yocto*

CHAPTER

FOUR

VOCABULARY

4.1 Recipes

Recipes contain information about the software project (author, homepage, and license). A recipe is versioned, defines dependencies, contains the URL of the source code, and describes how to fetch, configure, and compile the sources. It describes how to package the software, e.g. into different .deb packages, which then contain the installation path. Recipes are basically written in *Bitbake's* own programming language, which has a simple syntax. However, a recipe can contain *Python* as well as a bash code.

4.2 Classes

Classes combine functionality used inside recipes into reusable blocks.

4.3 Layers

A layer is a collection of recipes, classes, and configuration metadata. A layer can depend on other layers and can be included or excluded one by one. It encapsulates a specific functionality and fulfills a specific purpose. Each layer falls into a specific category:

- Base
- Machine (BSP)
- Software
- Distribution
- Miscellaneous

Yocto's versioning scheme is reflected in every layer as version branches. For each *Yocto* version, every layer has a named branch in its *Git* repository. You can add one or many layers of each category in your build.

A collection of OpenEmbedded layers can be found here. The search function is very helpful to see if a software package can be retrieved and integrated easily: https://layers.openembedded.org/layerindex/branch/mickledore/layers/

4.4 Machine

Machines are configuration variables that describe the aspects of the target hardware.

4.5 Distribution (Distro)

Distribution describes the software configuration and comes with a set of software features.

CHAPTER

FIVE

POKY

Poky is the reference system to define *Yocto* Project compatibility. It combines several subprojects into releases:

- *Bitbake*
- *Toaster*
- OpenEmbedded Core
- *Yocto* Documentation
- *Yocto* Reference BSP

5.1 Bitbake

Bitbake is the task scheduler. It is written in *Python* and interprets recipes that contain code in *Bitbake's* own programming language, *Python*, and bash code. The official documentation can be found here: https://docs.yoctoproject.org/bitbake/2.4/index.html

5.2 Toaster

Toaster is a web frontend for *Bitbake* to start and investigate builds. It provides information about the build history and statistics on created images. There are several use cases where the installation and maintenance of a *Toaster* instance are beneficial. PHYTEC did not add or remove any features to the upstream *Toaster*, provided by *Poky*. The best source for more information is the official documentation: https://docs.yoctoproject.org/4.2.4/toaster-manual/index.html

CHAPTER

SIX

OFFICIAL DOCUMENTATION

For more general questions about *Bitbake* and *Poky* consult the mega-manual: https://docs.yoctoproject.org/4.2.4/singleindex.html

CHAPTER

SEVEN

COMPATIBLE LINUX DISTRIBUTIONS

To build *Yocto* you need a compatible *Linux* host development machine. The list of supported distributions can be found in the reference manual: https://docs.yoctoproject.org/4.2.4/ref-manual/system-requirements.html#supported-linux-distributions

CHAPTER

EIGHT

PHYTEC BSP INTRODUCTION

8.1 BSP Structure

The BSP consists roughly of three parts. BSP management, BSP metadata, and BSP content. The management consists of *Repo* and phyLinux while the metadata depends on the SOC, which describes how to build the software. The content comprises PHYTEC's *Git* repositories and external sources.

8.1.1 BSP Management

Yocto is an umbrella project. Naturally, this will force the user to base their work on several external repositories. They need to be managed in a deterministic way. We use manifest files, which contain an XML data structure, to describe all git repositories with pinned-down versions. The *Repo* tool and our phyLinux wrapper script are used to manage the manifests and set up the BSP, as described in the manifest file.

phyLinux

phyLinux is a wrapper for *Repo* to handle downloading and setting up the BSP with an "out of the box" experience.

Repo

Repo is a wrapper around the *Repo* toolset. The phyLinux script will install the wrapper in a global path. This is only a wrapper, though. Whenever you run `repo init -u <url>`, you first download the *Repo* tools from *Googles* Git server in a specific version to the `.repo/repo` directory. The next time you run *Repo*, all the commands will be available. Be aware that the *Repo* version in different build directories can differ over the years if you do not run *Repo sync*. Also if you store information for your archives, you need to include the complete `.repo` folder.

Repo expects a *Git* repository which will be parsed from the command line. In the PHYTEC BSP, it is called phy^{2}octo. In this repository, all information about a software BSP release is stored in the form of a *Repo* XML manifest. This data structure defines URLs of *Git* servers (called "remotes") and *Git* repositories and their states (called "projects"). The *Git* repositories can be seen in different states. The revision field can be a branch, tag, or commit id of a repository. This means the state of the software is not necessarily unique and can change over time. That is the reason we use only tags or commit ids for our releases. The state of the working directory is then unique and does not change.

The manifests for the releases have the same name as the release itself. It is a unique identifier for the complete BSP. The releases are sorted by the SoC platform. The selected SoC will define the branch of the phy^{2}octo *Git* repository which will be used for the manifest selection.

8.1.2 BSP Metadata

We include several third-party layers in our BSP to get a complete *Linux* distribution up and running without the need to integrate external projects. All used repositories are described in the following section.

Poky

The PHYTEC BSP is built on top of *Poky*. It comes with a specific version, defined in the *Repo* manifest. *Poky* comes with a specific version of *Bitbake*. The OpenEmbedded-core layer "meta" is used as a base for our *Linux* system.

meta-openembedded

OpenEmbedded is a collection of different layers containing the meta description for many open-source software projects. We ship all OpenEmbedded layers with our BSP, but not all of them are activated. Our example images pull several software packages generated from OpenEmbedded recipes.

meta-qt6

This layer provides an integration of *Qt6* in the *Poky*-based root filesystem and is integrated into our BSP.

meta-nodejs

This is an application layer to add recent Node.js versions.

meta-gstreamer1.0

This is an application layer to add recent GStreamer versions.

meta-rauc

This layer contains the tools required to build an updated infrastructure with RAUC. A comparison with other update systems can be found here: Yocto update tools.

meta-phytec

This layer contains all machines and common features for all our BSPs. It is PHYTEC's Yocto Board Support Package for all supported hardware (since *fido*) and is designed to be standalone with *Poky*. Only these two parts are required if you want to integrate the PHYTEC's hardware into your existing *Yocto* workflow. The features are:

- Bootloaders in `recipes-bsp/barebox/` and `recipes-bsp/u-boot/`
- Kernels in `recipes-kernel/linux/` and `dynamic-layers/fsl-bsp-release/recipes-kernel/linux/`
- Many machines in `conf/machine/`
- Proprietary *OpenGL ES/EGL* user space libraries for AM335x and i.MX 6 platforms
- Proprietary *OpenCL* libraries for i.MX 6 platforms

meta-ampliphy

This is our example distribution and BSP layer. It extends the basic configuration of *Poky* with software projects described by all the other BSP components. It provides a base for your specific development scenarios. The current features are:

- systemd init system
- Images: `phytec-headless-image` for non-graphics applications

- Camera integration with OpenCV and GStreamer examples for the i.MX 6 platform bundled in a `phytec-vision-image`
- RAUC integration: we set up basic support for an A/B system image update, which is possible locally and over-the-air

meta-qt6-phytec

This is our layer for Qt6 board integration and examples. The features are:

- Qt6 with eglfs backend for PHYTEC's AM335x, i.MX 6 and RK3288 platforms
- Images: `phytec-qt6demo-image` for *Qt6* and video applications
- A *Qt6* demo application demonstrating how to create a *Qt6* project using *QML* widgets and a *Bitbake* recipe for the *Yocto* and *systemd* integration. It can be found in `sources/meta-qt6-phytec/recipes-qt/examples/phytec-qtdemo_git.bb`

meta-virtualization

- This layer provides support for building Xen, KVM, Libvirt, and associated packages necessary for constructing OE-based virtualized solutions.

meta-security

- This layer provides security tools, hardening tools for Linux kernels, and libraries for implementing security mechanisms.

meta-selinux

- This layer's purpose is to enable SE Linux support. The majority of this layer's work is accomplished in *bbappend* files, used to enable SE Linux support in existing recipes.

meta-browser

- This is an application layer to add recent web browsers (Chromium, Firefox, etc.).

meta-rust

- Includes the Rust compiler and the Cargo package manager for Rust.

meta-timesys

- Timesys layer for Vigiles Yocto CVE monitoring, security notifications, and image manifest generation.

meta-freescale

- This layer provides support for the i.MX, Layerscape, and QorIQ product lines.

meta-freescale-3rdparty

- Provides support for boards from various vendors.

meta-freescale-distro

- This layer provides support for Freescale's Demonstration images for use with OpenEmbedded and/or Yocto Freescale's BSP layer.

base (fsl-community-bsp-base)

- This layer provides BSP base files of NXP.

meta-fsl-bsp-release

- This is the i.MX Yocto Project Release Layer.

8.1.3 BSP Content

The BSP content gets pulled from different online sources when you first start using *Bitbake*. All files will be downloaded and cloned in a local directory configured as `DL_DIR` in *Yocto*. If you backup your BSP with the complete content, those sources have to be backed up, too. How you can do this will be explained in the chapter *Generating Source Mirrors, working Offline*.

8.2 Build Configuration

The BSP initializes a build folder that will contain all files you create by running *Bitbake* commands. It contains a `conf` folder that handles build input variables.

- `bblayers.conf` defines activated meta-layers,
- `local.conf` defines build input variables specific to your build
- `site.conf` defines build input variables specific to the development host

The two topmost build input variables are `DISTRO` and `MACHINE`. They are preconfigured `local.conf` when you check out the BSP using phyLinux.

We use "*Ampliphy*" as `DISTRO` with our BSP. This distribution will be preselected and give you a starting point for implementing your own configuration.

A `MACHINE` defines a binary image that supports specific hardware combinations of module and baseboard. Check the `machine.conf` file or our webpage for a description of the hardware.

CHAPTER

NINE

PRE-BUILT IMAGES

For each BSP we provide pre-built target images that can be downloaded from the PHYTEC FTP server: https://download.phytec.de/Software/Linux/

These images are also used for the BSP tests, which are flashed to the boards during production. You can use the provided `.wic` images to create a bootable SD card at any time. Identify your hardware and flash the downloaded image file to an empty SD card using `dd`. Please see section Images for information about the correct usage of the command.

CHAPTER

TEN

BSP WORKSPACE INSTALLATION

10.1 Setting Up the Host

You can set up the host or use one of our build-container to run a Yocto build. You need to have a running *Linux* distribution. It should be running on a powerful machine since a lot of compiling will need to be done.

If you want to use a build-container, you only need to install following packages on your host

```
host:~$ sudo apt install wget git
```

Continue with the next step *Git Configuration* after that. The documentation for using build-container can be found in this manual after *Advanced Usage* of phyLinux.

Else *Yocto* needs a handful of additional packages on your host. For *Ubuntu* you need

```
host:~$ sudo apt install gawk wget git diffstat unzip texinfo \
      gcc build-essential chrpath socat cpio python3 python3-pip \
      python3-pexpect xz-utils debianutils iputils-ping python3-git \
      python3-jinja2 libegl1-mesa libsdl1.2-dev \
      python3-subunit mesa-common-dev zstd liblz4-tool file locales
```

For other distributions you can find information in the *Yocto* Quick Build: https://docs.yoctoproject.org/4.2.4/brief-yoctoprojectqs/index.html

10.2 Git Configuration

The BSP heavily utilizes *Git*. *Git* needs some information from you as a user to identify who made changes. Create a `~/.gitconfig` with the following content, if you do not have one

```
[user]
    name = <Your Name>
    email = <Your Mail>
[core]
    editor = vim
[merge]
    tool = vimdiff
[alias]
    co = checkout
    br = branch
    ci = commit
    st = status
    unstage = reset HEAD --
```

(continues on next page)

(continued from previous page)

```
    last = log -1 HEAD
[push]
    default = current
[color]
    ui = auto
```

You should set `name` and `email` in your *Git* configuration, otherwise, *Bitbake* will complain during the first build. You can use the two commands to set them directly without editing `~/.gitconfig` manually

```
host:~$ git config --global user.email "your_email@example.com"
host:~$ git config --global user.name "name surname"
```

10.3 site.conf Setup

Before starting the *Yocto* build, it is advisable to configure the development setup. Two things are most important: the download directory and the cache directory. PHYTEC strongly recommends configuring the setup as it will reduce the compile time of consequent builds.

A download directory is a place where *Yocto* stores all sources fetched from the internet. It can contain tar.gz, *Git* mirror, etc. It is very useful to set this to a common shared location on the machine. Create this directory with 777 access rights. To share this directory with different users, all files need to have group write access. This will most probably be in conflict with default *umask* settings. One possible solution would be to use ACLs for this directory

```
host:~$ sudo apt-get install acl
host:~$ sudo setfacl -R -d -m g::rwx <dl_dir>
```

If you have already created a download directory and want to fix the permissions afterward, you can do so with

```
host:~$ sudo find /home/share/ -perm /u=r ! -perm /g=r -exec chmod g+r \{\} \;
host:~$ sudo find /home/share/ -perm /u=w ! -perm /g=w -exec chmod g+w \{\} \;
host:~$ sudo find /home/share/ -perm /u=x ! -perm /g=x -exec chmod g+x \{\} \;
```

The cache directory stores all stages of the build process. *Poky* has quite an involved caching infrastructure. It is advisable to create a shared directory, as all builds can access this cache directory, called the shared state cache.

Create the two directories on a drive where you have approximately 50 GB of space and assign the two variables in your `build/conf/local.conf`:

```
DL_DIR ?= "<your_directory>/yocto_downloads"
SSTATE_DIR ?= "<your_directory>/yocto_sstate"
```

If you want to know more about configuring your build, see the documented example settings

```
sources/poky/meta-yocto/conf/local.conf.sample
sources/poky/meta-yocto/conf/local.conf.sample.extended
```

CHAPTER

ELEVEN

PHYLINUX DOCUMENTATION

The phyLinux script is a basic management tool for PHYTEC *Yocto* BSP releases written in *Python*. It is mainly a helper to get started with the BSP structure. You can get all the BSP sources without the need of interacting with *Repo* or *Git*.

The phyLinux script has only one real dependency. It requires the *wget* tool installed on your host. It will also install the Repo tool in a global path (/usr/local/bin) on your host PC. You can install it in a different location manually. *Repo* will be automatically detected by phyLinux if it is found in the PATH. The *Repo* tool will be used to manage the different *Git* repositories of the *Yocto* BSP.

11.1 Get phyLinux

The phyLinux script can be found on the PHYTEC download server: https://download.phytec.de/Software/Linux/Yocto/Tools/phyLinux

11.2 Basic Usage

For the basic usage of phyLinux, type

```
host:~$ ./phyLinux --help
```

which will result in

```
usage: phyLinux [-h] [-v] [--verbose] {init,info,clean} ...

This Programs sets up an environment to work with The Yocto Project on Phytecs
Development Kits. Use phyLinx <command> -h to display the help text for the
available commands.

positional arguments:
  {init,info,clean}  commands
    init             init the phytec bsp in the current directory
    info             print info about the phytec bsp in the current directory
    clean            Clean up the current working directory

optional arguments:
  -h, --help         show this help message and exit
  -v, --version      show program's version number and exit
  --verbose
```

11.3 Initialization

Create a fresh project folder

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

Calling phyLinux will use the default Python version. Starting with Ubuntu 20.04 it will be Python3. If you want to initiate a BSP, which is not compatible with Python3, you need to set Python2 as default (temporarily) before running phyLinux

```
host:~$ ln -s \`which python2\` python && export PATH=`pwd`:$PATH
```

Now run phyLinux from the new folder

```
host:~$ ./phyLinux init
```

A clean folder is important because phyLinux will clean its working directory. Calling phyLinux from a directory that isn't empty will result in the following **warning**:

```
This current directory is not empty. It could lead to errors in the BSP configuration
process if you continue from here. At the very least, you have to check your build directory
for settings in bblayers.conf and local.conf, which will not be handled correctly in
all cases. It is advisable to start from an empty directory of call:
$ ./phyLinux clean
Do you really want to continue from here?
[yes/no]:
```

On the first initialization, the phyLinux script will ask you to install the *Repo* tool in your */usr/local/bin* directory. During the execution of the *init* command, you need to choose your processor platform (SoC), PHYTEC's BSP release number, and the hardware you are working on

```
***************************************************
* Please choose one of the available SoC Platforms:
*
*   1: am335x
*   2: am57x
*   3: am62ax
*   4: am62x
*   5: am64x
*   6: am68x
*   7: imx6
*   8: imx6ul
*   9: imx7
*   10: imx8
*   11: imx8m
*   12: imx8mm
*   13: imx8mp
*   14: imx8x
*   15: imx93
*   16: nightly
*   17: rk3288
*   18: stm32mp13x
*   19: stm32mp15x
*   20: topic
```

(continues on next page)

(continued from previous page)

```
# Exemplary output for chosen imx93
***************************************************
* Please choose one of the available Releases:
*
*   1: BSP-Yocto-NXP-i.MX93-ALPHA1
*   2: BSP-Yocto-NXP-i.MX93-PD24.1-rc1
*   3: BSP-Yocto-NXP-i.MX93-PD24.1.0
*   4: BSP-Yocto-NXP-i.MX93-PD24.1.1-rc1
*   5: BSP-Yocto-NXP-i.MX93-PD24.1.1-rc2
*   6: BSP-Yocto-NXP-i.MX93-PD24.1.1-rc3
*   7: BSP-Yocto-NXP-i.MX93-PD24.1.1

# Exemplary output for chosen BSP-Yocto-NXP-i.MX93-PD24.1.1
*********************************************************************
* Please choose one of the available builds:
*
no:                 machine: description and article number
                              distro: supported yocto distribution
                              target: supported build target

1: phyboard-nash-imx93-1:  PHYTEC phyBOARD-Nash i.MX93
                           2 GB RAM, eMMC
                           PB-04729-001, PCL-077-23231211I
                           distro: ampliphy-vendor
                           target: phytec-headless-image
2: phyboard-nash-imx93-1:  PHYTEC phyBOARD-Nash i.MX93
                           2 GB RAM, eMMC
                           PB-04729-001, PCL-077-23231211I
                           distro: ampliphy-vendor-rauc
                           target: phytec-headless-bundle
3: phyboard-nash-imx93-1:  PHYTEC phyBOARD-Nash i.MX93
                           2 GB RAM, eMMC
                           PB-04729-001, PCL-077-23231211I
                           distro: ampliphy-vendor-wayland
                           target: -c populate_sdk phytec-qt6demo-image
                           target: phytec-qt6demo-image
4: phyboard-segin-imx93-2: PHYTEC phyBOARD-Segin i.MX93
                           1 GB RAM, eMMC, silicon revision A1
                           PB-02029-001, PCL-077-11231010I
                           distro: ampliphy-vendor
                           target: phytec-headless-image
5: phyboard-segin-imx93-2: PHYTEC phyBOARD-Segin i.MX93
                           1 GB RAM, eMMC, silicon revision A1
                           PB-02029-001, PCL-077-11231010I
                           distro: ampliphy-vendor-rauc
                           target: phytec-headless-bundle
6: phyboard-segin-imx93-2: PHYTEC phyBOARD-Segin i.MX93
                           1 GB RAM, eMMC, silicon revision A1
                           PB-02029-001, PCL-077-11231010I
                           distro: ampliphy-vendor-wayland
                           target: phytec-qt6demo-image
```

If you cannot identify your board with the information given in the selector, have a look at the invoice for the product. After the configuration is done, you can always run

```
host:~$ ./phyLinux info

# Exemplary output
**********************************************
* The current BSP configuration is:
*
* SoC:  refs/heads/imx93
* Release:  BSP-Yocto-NXP-i.MX93-PD24.1.1
* Machine:  phyboard-segin-imx93-2
*
**********************************************
```

to see which SoC and Release are selected in the current workspace. If you do not want to use the selector, phyLinux also supports command-line arguments for several settings

```
host:~$ MACHINE=phyboard-segin-imx93-2 ./phyLinux init -p imx93 -r BSP-Yocto-NXP-i.MX93-PD24.1.1
```

or view the help command for more information

```
host:~$ ./phyLinux  init --help

usage: phyLinux init [-h] [--verbose] [--no-init] [-o REPOREPO] [-b REPOREPO_BRANCH] [-x XML] [-
↪u URL] [-p PLATFORM] [-r RELEASE]

options:
  -h, --help          show this help message and exit
  --verbose
  --no-init           dont execute init after fetch
  -o REPOREPO         Use repo tool from another url
  -b REPOREPO_BRANCH  Checkout different branch of repo tool
  -x XML              Use a local XML manifest
  -u URL              Manifest git url
  -p PLATFORM         Processor platform
  -r RELEASE          Release version
```

After the execution of the *init* command, phyLinux will print a few important notes as well as information for the next steps in the build process.

11.4 Advanced Usage

phyLinux can be used to transport software states over any medium. The state of the software is uniquely identified by *manifest.xml*. You can create a manifest, send it to another place and recover the software state with

```
host:~$ ./phyLinux init -x manifest.xml
```

You can also create a *Git* repository containing your software states. The *Git* repository needs to have branches other than master, as we reserved the master branch for different usage. Use phyLinux to check out the states

```
host:~$ ./phyLinux -u <url-of-your-git-repo>
```

CHAPTER

TWELVE

USING BUILD-CONTAINER

Warning

Currently, it is not possible to run the phyLinux script inside of a container. After a complete init with the phyLinux script on your host machine, you can use a container for the build. If you do not have phyLinux script running on your machine, please see phyLinux Documentation.

There are various possibilities to run a build-container. Commonly used is docker and podman, though we prefer podman as it does not need root privileges to run.

12.1 Installation

How to install podman: https://podman.io How to install docker: https://docs.docker.com/engine/install/

12.2 Available container

Right now we provide 4 different container based on Ubuntu LTS versions: https://hub.docker.com/u/phybuilder

- yocto-ubuntu-16.04
- yocto-ubuntu-18.04
- yocto-ubuntu-20.04
- yocto-ubuntu-22.04

These containers can be run with podman or docker. With Yocto Project branch Mickledore the container "yocto-ubuntu-20.04" is preferred.

12.3 Download/Pull container

```
host:~$ podman pull docker.io/phybuilder/yocto-ubuntu-20.04

OR

host:~$ docker pull docker.io/phybuilder/yocto-ubuntu-20.04
```

By adding a tag at the end separated by a colon, you can also pull or run a special tagged container.

podman pull docker.io/phybuilder/yocto-ubuntu-20.04:phy2

You can find all available tags in our duckerhub space:

- https://hub.docker.com/r/phybuilder/yocto-ubuntu-16.04/tags
- https://hub.docker.com/r/phybuilder/yocto-ubuntu-18.04/tags
- https://hub.docker.com/r/phybuilder/yocto-ubuntu-20.04/tags
- https://hub.docker.com/r/phybuilder/yocto-ubuntu-22.04/tags

If you try to run a container, which is not pulled/downloaded, it will be pulled/downloaded automatically.

You can have a look at all downloaded/pulled container with:

```
$USERNAME@$HOSTNAME:~$ podman images
REPOSITORY                                TAG         IMAGE ID      CREATED       SIZE
docker.io/phybuilder/yocto-ubuntu-22.04  latest      d626178e448d  4 months ago  935 MB
docker.io/phybuilder/yocto-ubuntu-22.04  phy2        d626178e448d  4 months ago  935 MB
docker.io/phybuilder/yocto-ubuntu-20.04  phy2        e29a88b7172a  4 months ago  900 MB
docker.io/phybuilder/yocto-ubuntu-20.04  latest      e29a88b7172a  4 months ago  900 MB
docker.io/phybuilder/yocto-ubuntu-18.04  phy1        14c9c3e477d4  7 months ago  567 MB
docker.io/phybuilder/yocto-ubuntu-18.04  latest      14c9c3e477d4  7 months ago  567 MB
docker.io/phybuilder/yocto-ubuntu-16.04  phy1        28c73e13ab4f  7 months ago  599 MB
docker.io/phybuilder/yocto-ubuntu-16.04  latest      28c73e13ab4f  7 months ago  599 MB
docker.io/phybuilder/yocto-ubuntu-22.04  phy1        5a0ef4b41935  8 months ago  627 MB
docker.io/phybuilder/yocto-ubuntu-20.04  phy1        b5a26a86c39f  8 months ago  680 MB
```

12.4 Run container

To run and use container for a Yocto build, first enter to your folder, where you run phyLinux init before. Then start the container

```
host:~$ podman run --rm=true -v /home:/home --userns=keep-id --workdir=$PWD -it docker.io/
↪phybuilder/yocto-ubuntu-20.04 bash
```

Note

To run and use a container with docker, it is not that simple like with podman. Therefore the container-user has to be defined and configured. Furthermore forwarding of credentials is not given per default and has to be configured as well.

Now your commandline should look something like that (where $USERNAME is the user, who called "podman run" and the char/number code diffs every time a container is started)

```
$USERNAME@6593e2c7b8f6:~$
```

Warning

If the given username is "root" you will not be able to run bitbake at all. Please be sure, you run the container with your own user.

Now you are ready to go on and starting the build. To stop/close the container, just call

```
$USERNAME@6593e2c7b8f6:~$ exit
```

CHAPTER

THIRTEEN

WORKING WITH POKY AND BITBAKE

13.1 Start the Build

After you download all the metadata with phyLinux init, you have to set up the shell environment variables. This needs to be done every time you open a new shell for starting builds. We use the shell script provided by *Poky* in its default configuration. From the root of your project directory type

```
host:~$ source sources/poky/oe-init-build-env
```

The abbreviation for the source command is a single dot

```
host:~$ . sources/poky/oe-init-build-env
```

The current working directory of the shell should change to *build/*. Before building for the first time, you should take a look at the main configuration file

```
host:~$ vim conf/local.conf
```

Your local modifications for the current build are stored here. Depending on the SoC, you might need to accept license agreements. For example, to build the image for Freescale/NXP processors you need to accept the GPU and VPU binary license agreements. You have to uncomment the corresponding line

```
# Uncomment to accept NXP EULA # EULA can be found under
../sources/meta-freescale/EULA ACCEPT_FSL_EULA = "1"
```

Now you are ready to build your first image. We suggest starting with our smaller non-graphical image *phytec-headless-image* to see if everything is working correctly

```
host:~$ bitbake phytec-headless-image
```

The first compile process takes about 40 minutes on a modern Intel Core i7. All subsequent builds will use the filled caches and should take about 3 minutes.

13.2 Images images

If everything worked, the images can be found under

```
host:~$ cd deploy/images/<MACHINE>
```

The easiest way to test your image is to configure your board for SD card boot and to flash the build image to the SD card

```
host:~$ sudo dd if=phytec-headless-image-<MACHINE>.wic of=/dev/<your_device> bs=1M conv=fsync
```

Here <your_device> could be "sde", for example, depending on your system. Be very careful when selecting the right drive! Selecting the wrong drive can erase your hard drive! The parameter conv=fsync forces a data buffer to write to the device before dd returns.

After booting you can log in using a serial cable or over *ssh*. There is no root password. That is because of the debug settings in *conf/local.conf*. If you uncomment the line

```
#EXTRA_IMAGE_FEATURES = "debug-tweaks"
```

the debug settings, like setting an empty root password, will not be applied.

13.3 Accessing the Development States between Releases

Special release manifests exist to give you access to the current development states of the *Yocto* BSP. They will not be displayed in the phyLinux selection menu but need to be selected manually. This can be done using the following command line

```
host:~$ ./phyLinux init -p master -r mickledore
```

This will initialize a BSP that will track the latest development state. From now on running

```
host:~$ repo sync
```

this folder will pull all the latest changes from our Git repositories.

13.4 Inspect your Build Configuration

Poky includes several tools to inspect your build layout. You can inspect the commands of the layer tool

```
host:~$ bitbake-layers
```

It can, for example, be used to view in which layer a specific recipe gets modified

```
host:~$ bitbake-layers show-appends
```

Before running a build you can also launch *Toaster* to be able to inspect the build details with the Toaster web GUI

```
host:~$ source toaster start
```

Maybe you need to install some requirements, first

```
host:~$ pip3 install -r
../sources/poky/bitbake/toaster-requirements.txt
```

You can then point your browser to *http://0.0.0.0:8000/* and continue working with *Bitbake*. All build activity can be monitored and analyzed from this web server. If you want to learn more about *Toaster*, look at https://docs.yoctoproject.org/4.2.4/toaster-manual/index.html. To shut down the *Toaster* web GUI again, execute

```
host:~$ source toaster stop
```

13.5 BSP Features of meta-phytec and meta-ampliphy

13.5.1 *Buildinfo*

The *buildinfo* task is a feature in our recipes that prints instructions to fetch the source code from the public repositories. So you do not have to look into the recipes yourself. To see the instructions, e.g. for the *barebox* package, execute

```
host:~$ bitbake barebox -c buildinfo
```

in your shell. This will print something like

```
(mini) HOWTO: Use a local git repository to build barebox:

To get source code for this package and version (barebox-2022.02.0-phy1), execute

$ mkdir -p ~/git
$ cd ~/git
$ git clone git://git.phytec.de/barebox barebox
$ cd ~/git/barebox
$ git switch --create v2022.02.0-phy1-local-development 7fe12e65d770f7e657e683849681f339a996418b

You now have two possible workflows for your changes:

1. Work inside the git repository:
Copy and paste the following snippet to your "local.conf":

SRC_URI:pn-barebox = "git://${HOME}/git/barebox;branch=${BRANCH}"
SRCREV:pn-barebox = "${AUTOREV}"
BRANCH:pn-barebox = "v2022.02.0-phy1-local-development"

After that you can recompile and deploy the package with

$ bitbake barebox -c compile
$ bitbake barebox -c deploy

Note: You have to commit all your changes. Otherwise yocto doesn't pick them up!

2. Work and compile from the local working directory
To work and compile in an external source directory we provide the
externalsrc.bbclass. To use it, copy and paste the following snippet to your
"local.conf":

INHERIT += "externalsrc"
EXTERNALSRC:pn-barebox = "${HOME}/git/barebox"
EXTERNALSRC_BUILD:pn-barebox = "${HOME}/git/barebox"

Note: All the compiling is done in the EXTERNALSRC directory. Every time
you build an Image, the package will be recompiled and build.

NOTE: Tasks Summary: Attempted 1 tasks of which 0 didn't need to be rerun and all succeeded.
NOTE: Writing buildhistory
```

As you can see, everything is explained in the output.

Warning

Using *externalsrc* breaks a lot of *Yocto's* internal dependency mechanisms. It is not guaranteed that any changes to the source directory are automatically picked up by the build process and incorporated into the root filesystem or SD card image. You have to always use *–force*. E.g. to compile *barebox* and redeploy it to *deploy/images/<machine>* execute

```
host:~$ bitbake barebox -c compile --force
host:~$ bitbake barebox -c deploy
```

To update the SD card image with a new kernel or image first force the compilation of it and then force a rebuild of the root filesystem. Use

```
host:~$ bitbake phytec-qt6demo-image -c rootfs --force
```

Note that the build system is not modifying the external source directory. If you want to apply all patches the *Yocto* recipe is carrying to the external source directory, run the line

```
SRCTREECOVEREDTASKS="" BB_ENV_PASSTHROUGH_ADDITIONS="$BB_ENV_PASSTHROUGH_ADDITIONS␣
↪SRCTREECOVEREDTASKS" bitbake <recipe> -c patch
```

13.6 BSP Customization

To get you started with the BSP, we have summarized some basic tasks from the *Yocto* official documentation. It describes how to add additional software to the image, change the kernel and bootloader configuration, and integrate patches for the kernel and bootloader.

Minor modifications, such as adding software, are done in the file *build/conf/local.conf*. There you can overwrite global configuration variables and make small modifications to recipes.

There are 2 ways to make major changes:

1. Either create your own layer and use *bbappend* files.
2. Add everything to PHYTEC's Distro layer *meta-ampliphy*.

Creating your own layer is described in the section Create your own Layer.

13.6.1 Disable Qt Demo

By default, the BSP image *phytec-qt6demo-image* starts a Qt6 Demo application on the attached display or monitor. If you want to stop the demo and use the *Linux* framebuffer console behind it, connect to the target via serial cable or *ssh* and execute the shell command

```
target:~$ systemctl stop phytec-qtdemo.service
```

This command stops the demo temporarily. To start it again, reboot the board or execute

```
target:~$ systemctl start phytec-qtdemo.service
```

You can disable the service permanently, so it does not start on boot

```
target:~$ systemctl disable phytec-qtdemo.service
```

Tip

The last command only disables the service. It does not *stop* immediately. To see the current status execute

```
target:~$ systemctl status phytec-qtdemo.service
```

If you want to disable the service by default, edit the file *build/conf/local.conf* and add the following line

```
# file build/conf/local.conf
SYSTEMD_AUTO_ENABLE:pn-phytec-qtdemo = "disable"
```

After that, rebuild the image

```
host:~$ bitbake phytec-qt6demo-image
```

13.6.2 Framebuffer Console

On boards with a display interface, the framebuffer console is enabled per default. You can attach a USB keyboard and log in. To change the keyboard layout from the English default to German, type

```
target:~$ loadkeys /usr/share/keymaps/i386/qwertz/de-latin1.map.gz
```

To detach the framebuffer console, run

```
target:~$ echo 0 > sys/class/vtconsole/vtcon1/bind
```

To completely deactivate the framebuffer console, disable the following kernel configuration option

```
Device Drivers->Graphics Support->Support for framebuffer devices->Framebuffer Console Support
```

More information can be found at: https://www.kernel.org/doc/Documentation/fb/fbcon.txt

13.6.3 Tools Provided in the Prebuild Image

RAM Benchmark

Performing RAM and cache performance tests can best be done by using *pmbw* (Parallel Memory Bandwidth Benchmark/Measurement Tool). *Pmbw* runs several assembly routines which all use different access patterns to the caches and RAM of the SoC. Before running the test, make sure that you have about 2 MiB of space left on the device for the log files. We also lower the level of the benchmark to ask the kernel more aggressively for resources. The benchmark test will take several hours.

To start the test type

```
target:~$ nice -n -2 pmbw
```

Upon completion of the test run, the log file can be converted to a *gnuplot* script with

```
target:~$ stats2gnuplot stats.txt > run1.gnuplot
```

Now you can transfer the file to the host machine and install any version of *gnuplot*

```
host:~$ sudo apt-get install gnuplot host:~$ gnuplot run1.gnuplot
```

The generated *plots-<machine>.pdf* file contains all plots. To render single plots as *png* files for any web output you can use *Ghostscript*

```
host:~$ sudo apt-get install ghostscript
host:~$ gs -dNOPAUSE -dBATCH -sDEVICE=png16m -r150 -sOutputFile='page-%00d.png' plots-phyboard-
↪wega-am335x-1.pdf
```

13.6.4 Add Additional Software for the BSP Image

To add additional software to the image, look at the OpenEmbedded layer index: https://layers.openembedded.org/layerindex/branch/mickledore/layers/

First, select the *Yocto* version of the BSP you have from the drop-down list in the top left corner and click **Recipes**. Now you can search for a software project name and find which layer it is in. In some cases, the program is in *meta-openembedded*, *openembedded-core*, or *Poky* which means that the recipe is already in your build tree. This section describes how to add additional software when this is the case. If the package is in another layer, see the next section.

You can also search the list of available recipes

```
host:~$ bitbake -s | grep <program name> # fill in program name, like in
host:~$ bitbake -s | grep lsof
```

When the recipe for the program is already in the *Yocto* build, you can simply add it by appending a configuration option to your file *build/conf/local.conf*. The general syntax to add additional software to an image is

```
# file build/conf/local.conf
IMAGE_INSTALL:append = " <package1> <package2>"
```

For example, the line

```
# file build/conf/local.conf
IMAGE_INSTALL:append = " ldd strace file lsof"
```

installs some helper programs on the target image.

Warning

The leading whitespace is essential for the append command.

All configuration options in local.conf apply to all images. Consequently, the tools are now included in both images phytec-headless-image and phytec-qt6demo-image.

Notes about Packages and Recipes

You are adding packages to the IMAGE_INSTALL variable. Those are not necessarily equivalent to the recipes in your meta-layers. A recipe defines per default a package with the same name. But a recipe can set the PACKAGES variable to something different and is able to generate packages with arbitrary names. Whenever you look for software, you have to search for the package name and, strictly speaking, not for the recipe. In the worst case, you have to look at all PACKAGES variables. A tool such as *Toaster* can be helpful in some cases.

If you can not find your software in the layers provided in the folder *sources*, see the next section to include another layer into the *Yocto* build.

References: Yocto 4.2.4 Documentation - Customizing Yocto builds

13.6.5 Add an Additional Layer

This is a step-by-step guide on how to add another layer to your *Yocto* build and install additional software from it. As an example, we include the network security scanner *nmap* in the layer *meta-security*. First, you must locate the layer on which the software is hosted. Check out the OpenEmbedded MetaData Index and guess a little bit. The network scanner *nmap* is in the *meta-security* layer. See meta-security on layers.openembedded.org. To integrate it into the *Yocto* build, you have to check out the repository and then switch to the correct stable branch. Since the BSP is based on the *Yocto* 'sumo' build, you should try to use the 'sumo' branch in the layer, too.

```
host:~$ cd sources
host:~$ git clone git://git.yoctoproject.org/meta-security
host:~$ cd meta-security
host:~$ git branch -r
```

All available remote branches will show up. Usually there should be 'fido', 'jethro', 'krogoth', 'master', …

```
host:~$ git checkout mickledore
```

Now we add the directory of the layer to the file *build/conf/bblayers.conf* by appending the line

```
# file build/conf/bblayers.conf
BBLAYERS += "${BSPDIR}/sources/meta-security"
```

to the end of the file. After that, you can check if the layer is available in the build configuration by executing

```
host:~$ bitbake-layers show-layers
```

If there is an error like

```
ERROR: Layer 'security' depends on layer 'perl-layer', but this layer is not enabled in your␣
↪configuration
```

the layer that you want to add (here *meta-security*), depends on another layer, which you need to enable first. E.g. the dependency required here is a layer in *meta-openembedded* (in the PHYTEC BSP it is in the path *sources/meta-openembedded/meta-perl/*). To enable it, add the following line to *build/conf/bblayers.conf*

```
# file build/conf/bblayers.conf
BBLAYERS += "${BSPDIR}/sources/meta-openembedded/meta-perl"
```

Now the command *bitbake-layers show-layers* should print a list of all layers enabled including *meta-security* and *meta-perl*. After the layer is included, you can install additional software from it as already described above. The easiest way is to add the following line (here is the package *nmap*)

```
# file build/conf/local.conf
IMAGE_INSTALL:append = " nmap"
```

to your *build/conf/local.conf*. Do not forget to rebuild the image

```
host:~$ bitbake phytec-qt6demo-image
```

13.6.6 Create your own layer

Creating your layer should be one of the first tasks when customizing the BSP. You have two basic options. You can either copy and rename our *meta-ampliphy*, or you can create a new layer that will contain your changes. The better option depends on your use case. *meta-ampliphy* is our example of how to create a custom *Linux* distribution that will be updated in the future. If you want to benefit from those changes and are, in general, satisfied with the userspace configuration, it could be the best solution to create your own layer on top of *Ampliphy*. If you need to rework a lot of information and only need the basic hardware support from PHYTEC, it would be better to copy *meta-ampliphy*, rename it, and adapt it to your needs. You can also have a look at the OpenEmbedded layer index to find different distribution layers. If you just need to add your own application to the image, create your own layer.

In the following chapter, we have an embedded project called "racer" which we will implement using our *Ampliphy Linux* distribution. First, we need to create a new layer.

Yocto provides a script for that. If you set up the BSP and the shell is ready, type

```
host:~$ bitbake-layers create-layer meta-racer
```

Default options are fine for now. Move the layer to the source directory

```
host:~$ mv meta-racer ../sources/
```

Create a *Git* repository in this layer to track your changes

```
host:~$ cd ../sources/meta-racer
host:~$ git init && git add . && git commit -s
```

Now you can add the layer directly to your build/conf/bblayers.conf

```
BBLAYERS += "${BSPDIR}/sources/meta-racer"
```

or with a script provided by *Yocto*

```
host:~$ bitbake-layers add-layer meta-racer
```

13.6.7 Kernel and Bootloader Recipe and Version

First, you need to know which kernel and version are used for your target machine. PHYTEC provides multiple kernel recipes *linux-mainline*, *linux-ti* and *linux-imx*. The first one provides support for PHYTEC's i.MX 6 and AM335x modules and is based on the *Linux* kernel stable releases from kernel.org. The *Git* repositories URLs are:

- *linux-mainline*: git://git.phytec.de/linux-mainline
- *linux-ti*: git://git.phytec.de/linux-ti
- *linux-imx:* git://git.phytec.de/linux-imx
- *barebox*: git://git.phytec.de/barebox
- *u-boot-imx*: git://git.phytec.de/u-boot-imx

To find your kernel provider, execute the following command

```
host:~$ bitbake virtual/kernel -e | grep "PREFERRED_PROVIDER_virtual/kernel"
```

The command prints the value of the variable *PREFERRED_PROVIDER_virtual/kernel*. The variable is used in the internal *Yocto* build process to select the kernel recipe to use. The following lines are different outputs you might see

```
PREFERRED_PROVIDER_virtual/kernel="linux-mainline"
PREFERRED_PROVIDER_virtual/kernel="linux-ti"
PREFERRED_PROVIDER_virtual/kernel="linux-imx"
```

To see which version is used, execute *bitbake -s*. For example

```
host:~$ bitbake -s | egrep -e "linux-mainline|linux-ti|linux-imx|barebox|u-boot-imx"
```

The parameter *-s* prints the version of all recipes. The output contains the recipe name on the left and the version on the right

```
barebox                      :2022.02.0-phy1-r7.0
barebox-hosttools-native     :2022.02.0-phy1-r7.0
barebox-targettools          :2022.02.0-phy1-r7.0
linux-mainline               :5.15.102-phy1-r0.0
```

As you can see, the recipe *linux-mainline* has version *5.15.102-phy1*. In the PHYTEC's *linux-mainline Git* repository, you will find a corresponding tag *v5.15.102-phy1*. The version of the *barebox* recipe is 2022.02.0-phy1. On i.MX8M* modules the output will contain *linux-imx* and *u-boot-imx*.

13.6.8 Kernel and Bootloader Configuration

The bootloader used by PHYTEC, *barebox*, uses the same build system as the *Linux* kernel. Therefore, all commands in this section can be used to configure the kernel and bootloader. To configure the kernel or bootloader, execute one of the following commands

```
host:~$ bitbake -c menuconfig virtual/kernel  # Using the virtual provider name
host:~$ bitbake -c menuconfig linux-ti        # Or use the recipe name directly
host:~$ bitbake -c menuconfig linux-mainline  # Or use the recipe name directly (If you use an i.
↪MX 6 or RK3288 Module)
host:~$ bitbake -c menuconfig linux-imx       # Or use the recipe name directly (If you use an i.
↪MX 8M*)
host:~$ bitbake -c menuconfig barebox         # Or change the configuration of the bootloader
host:~$ bitbake -c menuconfig u-boot-imx      # Or change the configuration of the bootloader␣
↪(If you use an i.MX 8M*)
```

After that, you can recompile and redeploy the kernel or bootloader

```
host:~$ bitbake virtual/kernel -c compile  # Or 'barebox' for the bootloader
host:~$ bitbake virtual/kernel -c deploy   # Or 'barebox' for the bootloader
```

Instead, you can also just rebuild the complete build output with

```
host:~$ bitbake phytec-headless-image  # To update the kernel/bootloader, modules and the images
```

In the last command, you can replace the image name with the name of an image of your choice. The new images and binaries are in *build/deploy/images/<machine>/*.

Warning

The build configuration is not permanent yet. Executing *bitbake virtual/kernel -c clean* will remove everything.

To make your changes permanent in the build system, you have to integrate your configuration modifications into a layer. For the configuration you have two options:

- Include only a configuration fragment (a minimal *diff* between the old and new configuration)
- Complete default configuration (*defconfig*) after your modifications.

Having a set of configuration fragments makes what was changed at which stage more transparent. You can turn on and off the changes, you can manage configurations for different situations and it helps when porting changes to new kernel versions. You can also group changes together to reflect specific use cases. A fully assembled kernel configuration will be deployed in the directory *build/deploy/images/<machine>*. If you do not have any of those requirements, it might be simpler to just manage a separate *defconfig* file.

Add a Configuration Fragment to a Recipe

The following steps can be used for both kernel and bootloader. Just replace the recipe name *linux-mainline* in the commands with *linux-ti*, or *barebox* for the bootloader. If you did not already take care of this, start with a clean build. Otherwise, the diff of the configuration may be wrong

```
host:~$ bitbake linux-mainline -c clean
host:~$ bitbake linux-mainline -c menuconfig
```

Make your configuration changes in the menu and generate a config fragment

```
host:~$ bitbake linux-mainline -c diffconfig
```

which prints the path of the written file

```
Config fragment has been dumped into:
/home/<path>/build/tmp/work/phyboard_mira_imx6_11-phytec-linux-gnueabi/linux-mainline/4.19.100-
↪phy1-r0.0/fragment.cfg
```

All config changes are in the file *fragment.cfg* which should consist of only some lines. The following example shows how to create a *bbappend* file and how to add the necessary lines for the config fragment. You just have to adjust the directories and names for the specific recipe: *linux-mainline*, *linux-ti*, linux-imx, u-boot-imx, or *barebox*.

```
sources/<layer>/recipes-kernel/linux/linux-mainline_%.bbappend     # For the recipe linux-
↪mainline
sources/<layer>/recipes-kernel/linux/linux-ti_%.bbappend           # For the recipe linux-ti
sources/<layer>/recipes-kernel/linux/linux-imx_%.bbappend          # For the recipe linux-imx
sources/<layer>/recipes-bsp/barebox/barebox_%.bbappend             # For the recipe barebox
sources/<layer>/recipes-bsp/u-boot/u-boot-imx_%.bbappend           # For the recipe u-boot-imx
```

Replace the string *layer* with your own layer created as shown above (e.g. *meta-racer*), or just use *meta-ampliphy*. To use *meta-ampliphy*, first, create the directory for the config fragment and give it a new name (here *enable-r8169.cfg*) and move the fragment to the layer.

```
host:~$ mkdir -p sources/meta-ampliphy/recipes-kernel/linux/features
# copy the path from the output of *diffconfig*
host:~$ cp /home/<path>/build/tmp/work/phyboard_mira_imx6_11-phytec-linux-gnueabi/linux-mainline/
↪4.19.100-phy1-r0.0/fragment.cfg \
    sources/meta-ampliphy/recipes-kernel/linux/features/enable-r8169.cfg
```

Then open the *bbappend* file (in this case *sources/meta-ampliphy/recipes-kernel/linux/linux-mainline_%.bbappend*) with your favorite editor and add the following lines

```
# contents of the file linux-mainline_%.bbappend
FILESEXTRAPATHS:prepend := "${THISDIR}/features:"
SRC_URI:append = " \
    file://enable-r8169.cfg \
"
```

Warning

Do not forget to use the correct *bbappend* filenames: *linux-ti_%.bbappend* for the linux-ti recipe and *barebox_%.bbappend* for the bootloader in the folder *recipes-bsp/barebox/* !

After saving the *bbappend* file, you have to rebuild the image. *Yocto* should pick up the recipe changes automatically and generate a new image

```
host:~$ bitbake phytec-headless-image # Or another image name
```

Add a Complete Default Configuration (*defconfig*) to a Recipe

This approach is similar to the one above, but instead of adding a fragment, a *defconfig* is used. First, create the necessary folders in the layer you want to use, either your own layer or *meta-ampliphy*

```
host:~$ mkdir -p sources/meta-ampliphy/recipes-kernel/linux/features/ # For both linux-mainline␣
↪and linux-ti
host:~$ mkdir -p sources/meta-ampliphy/recipes-bsp/barebox/features/ # Or for the bootloader
```

Then you have to create a suitable *defconfig* file. Make your configuration changes using *menuconfig* and then save the *defconfig* file to the layer

```
host:~$ bitbake linux-mainline -c menuconfig # Or use recipe name linux-ti or barebox
host:~$ bitbake linux-mainline -c savedefconfig # Create file 'defconfig.temp' in the work␣
↪directory
```

This will print the path to the generated file

```
Saving defconfig to ..../defconfig.temp
```

Then, as above, copy the generated file to your layer, rename it to *defconfig*, and add the following lines to the *bbappend* file (here *sources/meta-ampliphy/recipes-kernel/linux/linux-mainline_%.bbappend*)

```
# contents of the file linux-mainline_%.bbappend
FILESEXTRAPATHS:prepend := "${THISDIR}/features:"
SRC_URI:append = " \
    file://defconfig \
"
```

Tip

Do not forget to use the correct bbappend filenames: *linux-ti_%.bbappend* for the linux-ti recipe and *barebox_%.bbappend* for the bootloader in the folder *recipes-bsp/barebox/* !

After that, rebuild your image as the changes are picked up automatically

```
host:~$ bitbake phytec-headless-image # Or another image name
```

13.6.9 Patch the Kernel or Bootloader with *devtool*

Apart from using the standard versions of kernel and bootloader which are provided in the recipes, you can modify the source code or use our own repositories to build your customized kernel.

PRO	CON
Standard workflow of the official *Yocto* documentation	Uses additional hard drive space as the sources get duplicated
Toolchain does not have to recompile everything	No optimal cache usage, build overhead

Devtool is a set of helper scripts to enhance the user workflow of *Yocto*. It was integrated with version 1.8. It is available as soon as you set up your shell environment. *Devtool* can be used to:

- modify existing sources
- integrate software projects into your build setup
- build software and deploy software modifications to your target

Here we will use *devtool* to patch the kernel. We use *linux-mainline* as an example for the AM335x Kernel. The first command we use is *devtool modify - x <recipe> <directory>*

```
host:~$ devtool modify -x linux-mainline linux-mainline
```

Devtool will create a layer in *build/workspace* where you can see all modifications done by *devtool* . It will extract the sources corresponding to the recipe to the specified directory. A *bbappend* will be created in the workspace directing the SRC_URI to this directory. Building an image with *Bitbake* will now use the sources in this directory. Now you can modify lines in the kernel

```
host:~$ vim linux-mainline/arch/arm/boot/dts/am335x-phycore-som.dtsi
      -> make a change
host:~$ bitbake phytec-qt6demo-image
```

Your changes will now be recompiled and added to the image. If you want to store your changes permanently, it is advisable to create a patch from the changes, then store and backup only the patch. You can go into the *linux-mainline* directory and create a patch using *Git*. How to create a patch is described in *Patch the Kernel or Bootloader using the "Temporary Method"* and is the same for all methods.

If you want to learn more about *devtool*, visit:

Yocto 4.2.4 - Devtool or Devtool Quick Reference

13.6.10 Patch the Kernel or Bootloader using the "Temporary Method"

PRO	CON
No overhead, no extra configuration	Changes are easily overwritten by *Yocto* (Everything is lost!!).
Toolchain does not have to recompile everything	

It is possible to alter the source code before *Bitbake* configures and compiles the recipe. Use *Bitbake'* s *devshell* command to jump into the source directory of the recipe. Here is the *barebox* recipe

```
host:~$ bitbake barebox -c devshell # or linux-mainline, linux-ti, linux-imx, u-boot-imx
```

After executing the command, a shell window opens. The current working directory of the shell will be changed to the source directory of the recipe inside the *tmp* folder. Here you can use your favorite editor, e.g. *vim*, *emacs*, or any other graphical editor, to alter the source code. When you are finished, exit the *devshell* by typing *exit* or hitting **CTRL-D**.

After leaving the *devshell* you can recompile the package

```
host:~$ bitbake barebox -c compile --force # or linux-mainline, linux-ti, linux-imx, u-boot-imx
```

The extra argument '–force' is important because *Yocto* does not recognize that the source code was changed.

Tip

You cannot execute the *bitbake* command in the *devshell* . You have to leave it first.

If the build fails, execute the devshell command again and fix it. If the build is successful, you can deploy the package and create a new SD card image

```
host:~$ bitbake barebox -c deploy # new barebox in e.g. deploy/images/phyflex-imx6-2/barebox.bin
host:~$ bitbake phytec-headless-image # new WIC image in e.g. deploy/images/phyflex-imx6-2/
↪phytec-headless-image-phyflex-imx6-2.wic
```

Warning

If you execute a clean e.g *bitbake barebox -c clean* , or if *Yocto* fetches the source code again, all your changes are lost!!!

To avoid this, you can create a patch and add it to a *bbappend* file. It is the same workflow as described in the section about changing the configuration.

You have to create the patch in the *devshell* if you use the temporary method and in the subdirectory created by *devtool* if you used *devtool*.

```
host:~$ bitbake barebox -c devshell            # Or linux-mainline, linux-ti
host(devshell):~$ git status                   # Show changes files
host(devshell):~$ git add <file>               # Add a special file to the staging area
host(devshell):~$ git commit -m "important modification"   # Creates a commit with a not so␣
↪useful commit message
host(devshell):~$ git format-patch -1 -o ~/    # Creates a patch of the last commit and saves it␣
↪in your home folder
/home/<user>/0001-important-modification.patch  # Git prints the path of the written patch file
host(devshell):~$ exit
```

After you have created the patch, you must create a *bbappend* file for it. The locations for the three different recipes - *linux-mainline* , *linux-ti* , and *barebox* - are

```
sources/<layer>/recipes-kernel/linux/linux-mainline_%.bbappend     # For the recipe linux-
↪mainline
sources/<layer>/recipes-kernel/linux/linux-ti_%.bbappend           # For the recipe linux-ti
sources/<layer>/recipes-kernel/linux/linux-imx_%.bbappend        # For the recipe linux-imx
```

(continues on next page)

(continued from previous page)

```
sources/<layer>/recipes-bsp/barebox/barebox_%.bbappend             # For the recipe barebox
sources/<layer>/recipes-bsp/u-boot/u-boot-imx_%.bbappend           # For the recipe u-boot-imx
```

The following example is for the recipe *barebox*. You have to adjust the paths. First, create the folders and move the patch into them. Then create the *bbappend* file

```
host:~$ mkdir -p sources/meta-ampliphy/recipes-bsp/barebox/features   # Or use your own layer␣
↪instead of *meta-ampliphy*
host:~$ cp ~/0001-important-modification.patch sources/meta-ampliphy/recipes-bsp/barebox/
↪features  # copy patch
host:~$ touch sources/meta-ampliphy/recipes-bsp/barebox/barebox_%.bbappend
```

Tip

Pay attention to your current work directory. You have to execute the commands in the BSP top-level directory. Not in the *build* directory!

After that use your favorite editor to add the following snipped into the *bbappend* file (here *sources/meta-ampliphy/recipes-bsp/barebox/barebox_%.bbappend*)

```
# contents of the file barebox_%.bbappend
FILESEXTRAPATHS:prepend := "${THISDIR}/features:"
SRC_URI:append = " \
    file://0001-important-modification.patch \
"
```

Save the file and rebuild the *barebox* recipe with

```
host:~$ bitbake barebox -c clean # Or linux-ti, linux-mainline, linux-imx, u-boot-imx
host:~$ bitbake barebox
```

If the build is successful, you can rebuild the final image with

```
host:~$ bitbake phytec-headless-image # Or another image name
```

Further Resources:

The *Yocto* Project has some documentation for software developers. Check the 'Kernel Development Manual' for more information about how to configure the kernel. Please note that not all of the information from the *Yocto* manual can be applied to the PHYTEC BSP as we use the classic kernel approach of *Yocto* and most of the documentation assumes the *Yocto* kernel approach.

- Yocto - Kernel Development Manual
- Yocto - Development Manual

13.6.11 Working with the Kernel and Bootloader using SRC_URI in *local.conf*

Here we present a third option to make kernel and bootloader changes. You have external checkouts of the linux-mainline, linux-ti, or barebox Git repositories. You will overwrite the URL of the source code fetcher, the variable SRC_URI, to point to your local checkout instead of the remote repositories.

PRO	CON
All changes are saved with *Git*	Many working directories in *build/tmp-glibc/work/<machine>/<package>/*
	You have to commit every change before recompiling
	For each change, the toolchain compiles everything from scratch (avoidable with *ccache*)

First, you need a local clone of the *Git* repository *barebox* or kernel. If you do not have one, use the commands

```
host:~$ mkdir ~/git
host:~$ cd ~/git
host:~$ git clone git://git.phytec.de/barebox
host:~$ cd barebox
host:~$ git switch --create v2022.02.0-phy remotes/origin/v2022.02.0-phy
```

Add the following snippet to the file build/conf/local.conf

```
# Use your own path to the git repository
# NOTE: Branch name in variable "BRANCH_pn-barebox" should be the same as the
# branch which is used in the repository folder. Otherwise your commits won't be recognized␣
↪later.
BRANCH:pn-barebox = "v2022.02.0-phy"
SRC_URI:pn-barebox = "git:///${HOME}/git/barebox;branch=${BRANCH}"
SRCREV:pn-barebox = "${AUTOREV}"
```

You also have to set the correct BRANCH name in the file. Either you create your own branch in the *Git* repository, or you use the default (here "v2015.02.0-phy"). Now you should recompile *barebox* from your own source

```
host:~$ bitbake barebox -c clean
host:~$ bitbake barebox -c compile
```

The build should be successful because the source was not changed yet.

You can alter the source in *~/git/barebox* or the default *defconfig* (e.g. *~/git/barebox/arch/arm/configs/imx_v7_defconfig*). After you are satisfied with your changes, you have to make a dummy commit for *Yocto*. If you do not, *Yocto* will not notice that the source code was modified in your repository folder (e.g. ~/git/barebox/)

```
host:~$ git status  # show modified files
host:~$ git diff    # show changed lines
host:~$ git commit -a -m "dummy commit for yocto"   # This command is important!
```

Try to compile your new changes. *Yocto* will automatically notice that the source code was changed and fetches and configures everything from scratch.

```
host:~$ bitbake barebox -c compile
```

If the build fails, go back to the source directory, fix the problem, and recommit your changes. If the build was successful, you can deploy *barebox* and even create a new SD card image.

```
host:~$ bitbake barebox -c deploy # new barebox in e.g. deploy/images/phyflex-imx6-2/barebox-
↪phyflex-imx6-2.bin
```

(continues on next page)

(continued from previous page)

```
host:~$ bitbake phytec-headless-image # new sd-card image in e.g. deploy/images/phyflex-imx6-2/
↪phytec-headless-image-phyflex-imx6-2.wic
```

If you want to make additional changes, just make another commit in the repository and rebuild *barebox* again.

13.6.12 Add Existing Software with "Sustainable Method"

Now that you have created your own layer, you have a second option to add existing software to existing image definitions. Our standard image is defined in meta-ampliphy

```
meta-ampliphy/recipes-images/images/phytec-headless-image.bb
```

In your layer, you can now modify the recipe with a *bbappend* without modifying any BSP code

```
meta-racer/recipes-images/images/phytec-headless-image.bbappend
```

The append will be parsed together with the base recipe. As a result, you can easily overwrite all variables set in the base recipe, which is not always what you want. If we want to include additional software, we need to append it to the IMAGE_INSTALL variable

```
IMAGE_INSTALL:append = " rsync"
```

13.6.13 Add Linux Firmware Files to the Root Filesystem

It is a common task to add an extra firmware file to your root filesystem into */lib/firmware/*. For example, WiFi adapters or PCIe Ethernet cards might need proprietary firmware. As a solution, we use a *bbappend* in our layer. To create the necessary folders, *bbappend* and copy the firmware file type

```
host:~$ cd meta-racer   # go into your layer
host:~$ mkdir -p recipes-kernel/linux-firmware/linux-firmware/
host:~$ touch recipes-kernel/linux-firmware/linux-firmware_%.bbappend
host:~$ cp ~/example-firmware.bin recipes-kernel/linux-firmware/linux-firmware/    # adapt␣
↪filename
```

Then add the following content to the *bbappend* file and replace every occurrence of *example-firmware.bin* with your firmware file name.

```
# file recipes-kernel/linux-firmware/linux-firmware_%.bbappend

FILESEXTRAPATHS:prepend := "${THISDIR}/linux-firmware:"
SRC_URI += "file://example-firmware.bin"

do_install:append () {
        install -m 0644 ${WORKDIR}/example-firmware.bin ${D}/lib/firmware/example-firmware.bin
}

# NOTE: Use "=+" instead of "+=". Otherwise file is placed into the linux-firmware package.
PACKAGES =+ "${PN}-example"
FILES:${PN}-example = "/lib/firmware/example-firmware.bin"
```

Now try to build the linux-firmware recipe

```
host:~$ . sources/poky/oe-init-build-env
host:~$ bitbake linux-firmware
```

This should generate a new package *deploy/ipk/all/linux-firmware-example*.

As the final step, you have to install the firmware package to your image. You can do that in your *local.conf* or image recipe via

```
# file local.conf or image recipe
IMAGE_INSTALL += "linux-firmware-example"
```

Warning

Ensure that you have adapted the package name *linux-firmware-example* with the name you assigned in *linux-firmware_%.bbappend*.

13.6.14 Change the *u-boot* Environment via *bbappend* Files

All i.MX8M* products use the u-boot bootloader. The u-boot environment can be modified using the Temporary Method. In the *u-boot-imx* sources modify the header file corresponding to the processor located in *include/configs/phycore_imx8m**. New environment variables should be added at the end of *CONFIG_EXTRA_ENV_SETTINGS*

```
#define CONFIG_EXTRA_ENV_SETTINGS \
[...]
PHYCORE_FITIMAGE_ENV_BOOTLOGIC \
"newvariable=1\0"
```

Commit the changes and and create the file *u-boot-imx_%.bbappend* in your layer at *<layer>/recipes-bsp/u-boot/u-boot-imx_%.bbappend*

```
# contents of the file u-boot-imx_%.bbappend
FILESEXTRAPATHS:prepend := "${THISDIR}/features:"
SRC_URI:append = " \
    file://0001-environment-addition.patch \
"
```

13.6.15 Change the *barebox* Environment via *bbappend* Files

Since *BSP-Yocto-AM335x-16.2.0* and *BSP-Yocto-i.MX6-PD16.1.0*, the *barebox* environment handling in *meta-phytec* has changed. Now it is possible to add, change, and remove files in the *barebox* environment via the *Python* bitbake task *do_env*. There are two *Python* functions to change the environment. Their signatures are:

- *env_add(d, ***filename as string*, ***file content as string*)*: to add a new file or overwrite an existing file
- *env_rm(d, ***filename as string*)*: to remove a file

The first example of a *bbappend* file in the custom layer *meta-racer* shows how to add a new non-volatile variable *linux.bootargs.fb* in the *barebox* environment folder */env/nv/*

```
# file meta-racer/recipes-bsp/barebox/barebox_2022.02.0-phy1.bbappend
python do_env:append() {
    env_add(d, "nv/linux.bootargs.fb", "imxdrm.legacyfb_depth=32\n")
}
```

The next example shows how to replace the network configuration file */env/network/eth0*

```
# file meta-racer/recipes-bsp/barebox/barebox_2022.02.0-phy1.bbappend
python do_env:append() {
    env_add(d, "network/eth0",
"""#!/bin/sh

# ip setting (static/dhcp)
ip=static
global.dhcp.vendor_id=barebox-${global.hostname}

# static setup used if ip=static
ipaddr=192.168.178.5
netmask=255.255.255.0
gateway=192.168.178.1
serverip=192.168.178.1
""")
}
```

In the above example, the *Python* multiline string syntax **""" text """** is used to avoid adding multiple newline characters *\n* into the recipe *Python* code. The *Python* function *env_add* can add and overwrite environment files.

The next example shows how to remove an already added environment file, for example , */env/boot/mmc*

```
# file meta-racer/recipes-bsp/barebox/barebox_2022.02.0-phy1.bbappend
python do_env:append() {
    env_rm(d, "boot/mmc")
}
```

Debugging the Environment

If you want to see all environment files that are added in the build process, you can enable a debug flag in the *local.conf*

```
# file local.conf
ENV_VERBOSE = "1"
```

After that, you have to rebuild the *barebox* recipe to see the debugging output

```
host:~$ bitbake barebox -c clean
host:~$ bitbake barebox -c configure
```

The output of the last command looks like this

```
[...]
WARNING: barebox-2022.02.0-phy1-r7.0 do_env_write: File 'nv/allow_color' content "false"
WARNING: barebox-2022.02.0-phy1-r7.0 do_env_write: File 'nv/linux.bootargs.base' content
↪"consoleblank=0"
```

(continues on next page)

(continued from previous page)

```
WARNING: barebox-2022.02.0-phy1-r7.0 do_env_write: File 'nv/linux.bootargs.fb' content "imxdrm.
↪legacyfb_depth=32"
WARNING: barebox-2022.02.0-phy1-r7.0 do_env_write: File 'nv/linux.bootargs.rootfs' content
↪"rootwait ro fsck.repair=yes"
```

Changing the Environment (depending on Machines)

If you need to apply some *barebox* environment modifications only to a single or only a few machines, you can use *Bitbake'* s machine overwrite syntax. For the machine overwrite syntax, you append a machine name or SoC name (such as *mx6* , *ti33x,* or *rk3288*) with an underscore to a variable or task

```
DEPENDS:remove:mx6 = "virtual/libgl" or
python do_env_append_phyboard-mira-imx6-4().
```

The next example adds the environment variables only if the MACHINE is set to *phyboard-mira-imx6-4*

```
# file meta-phytec/recipes-bsp/barebox/barebox_2022.02.0-phy1.bbappend
python do_env:append:phyboard-mira-imx6-4() {
    env_add(d, "nv/linux.bootargs.cma", "cma=64M\n")
}
```

Bitbake's override syntax for variables is explained in more detail at: https://docs.yoctoproject.org/bitbake/2.4/bitbake-user-manual/bitbake-user-manual-metadata.html#conditional-metadata

Upgrading the *barebox* Environment from Previous BSP Releases

Prior to BSP version *BSP-Yocto-AM335x-16.2.0* and *BSP-Yocto-i.MX6-PD16.1.0* , *barebox* environment changes via *bbappend* file were done differently. For example, the directory structure in your meta layer (here *meta-skeleton*) may have looked like this

```
host:~$ tree -a sources/meta-skeleton/recipes-bsp/barebox/
sources/meta-skeleton/recipes-bsp/barebox
├── barebox
│   └── phyboard-wega-am335x-3
│       ├── boardenv
│       │   └── .gitignore
│       └── machineenv
│           └── nv
│               └── linux.bootargs.cma
└── barebox_%.bbappend
```

and the file *barebox_%.bbappend* contained

```
# file sources/meta-skeleton/recipes-bsp/barebox/barebox_%.bbappend
FILESEXTRAPATHS:prepend := "${THISDIR}/barebox:"
```

In this example, all environment changes from the directory *boardenv* in the layer *meta-phytec* are ignored and the file *nv/linux.bootargs.cma* is added. For the new handling of the *barebox* environment, you use the *Python* functions *env_add* and *env_rm* in the *Python* task *do_env*. Now the above example translates to a single *Python* function in the file *barebox_%.bbappend* that looks like

```
# file sources/meta-skeleton/recipes-bsp/barebox/barebox_%.bbappend
FILESEXTRAPATHS:prepend := "${THISDIR}/barebox:"
```

(continues on next page)

(continued from previous page)

```
python do_env:append() {
    # Removing files (previously boardenv)
    env_rm(d, "config-expansions")
    # Adding new files (previously machineenv)
    env_add(d, "nv/linux.bootargs.cma", "cma=64M\n")
}
```

13.6.16 Changing the Network Configuration

To tweak IP addresses, routes, and gateways at runtime you can use the tools *ifconfig* and *ip* . Some examples

```
target:~$ ip addr                                       # Show all network interfaces
target:~$ ip route                                      # Show all routes
target:~$ ip addr add 192.168.178.11/24 dev eth0        # Add static ip and route to interface␣
↪eth0
target:~$ ip route add default via 192.168.178.1 dev eth0 # Add default gateway 192.168.178.1
target:~$ ip addr del 192.168.178.11/24 dev eth0        # Remove static ip address from␣
↪interface eth0
```

The network configuration is managed by *systemd-networkd* . To query the current status use

```
target:~$ networkctl status
target:~$ networkctl list
```

The network daemon reads its configuration from the directories */etc/systemd/network/* , */run/systemd/network/* , and */lib/systemd/network/* (from higher to lower priority). A sample configuration in */lib/systemd/network/10-eth0.network* looks like this

```
# file /lib/systemd/network/10-eth0.network
[Match]
Name=eth0

[Network]
Address=192.168.3.11/24
Gateway=192.168.3.10
```

These files **.network* replace */etc/network/interfaces* from other distributions. You can either edit the file *10-eth0.network* in-place or copy it to */etc/systemd/network/* and make your changes there. After changing a file you must restart the daemon to apply your changes

```
target:~$ systemctl restart systemd-networkd
```

To see the syslog message of the network daemon, use

```
target:~$ journalctl --unit=systemd-networkd.service
```

To modify the network configuration at build time, look at the recipe *sources/meta-ampliphy/recipes-core/systemd/systemd-machine-units.bb* and the interface files in the folder *meta-ampliphy/recipes-core/systemd/systemd-machine-units/* where the static IP address configuration for *eth0* (and optionally *eth1*) is done.

For more information, see https://wiki.archlinux.org/title/Systemd-networkd and https://www.freedesktop.org/software/systemd/man/latest/systemd.network.html.

13.6.17 Changing the Wireless Network Configuration

Connecting to a WLAN Network

- First set the correct regulatory domain for your country

```
target:~$ iw reg set DE
target:~$ iw reg get
```

You will see

```
country DE: DFS-ETSI
   (2400 - 2483 @ 40), (N/A, 20), (N/A)
   (5150 - 5250 @ 80), (N/A, 20), (N/A), NO-OUTDOOR
   (5250 - 5350 @ 80), (N/A, 20), (0 ms), NO-OUTDOOR, DFS
   (5470 - 5725 @ 160), (N/A, 26), (0 ms), DFS
   (57000 - 66000 @ 2160), (N/A, 40), (N/A)
```

- Set up the wireless interface

```
target:~$ ip link    # list all interfaces. Search for wlan*
target:~$ ip link set up dev wlan0
```

- Now you can scan for available networks

```
target:~$ iw wlan0 scan | grep SSID
```

You can use a cross-platform supplicant with support for *WEP*, *WPA*, and *WPA2* called *wpa_supplicant* for an encrypted connection.

- To do so, add the network credentials to the file */etc/wpa_supplicant.conf*

```
Confluence country=DE network={ ssid="<SSID>" proto=WPA2 psk="<KEY>" }
```

- Now a connection can be established

```
target:~$ wpa_supplicant -Dnl80211 -c/etc/wpa_supplicant.conf -iwlan0 -B
```

This should result in the following output

```
ENT-CONNECTED - Connection to 88:33:14:5d:db:b1 completed [id=0 id_str=]
```

To finish the configuration you can configure DHCP to receive an IP address (supported by most WLAN access points). For other possible IP configurations, see the section *Changing the Network Configuration*.

- First, create the directory

```
target:~$ mkdir -p /etc/systemd/network/
```

- Then add the following configuration snippet in */etc/systemd/network/10-wlan0.network*

```
# file /etc/systemd/network/10-wlan0.network
[Match]
Name=wlan0

[Network]
DHCP=yes
```

- Now, restart the network daemon so that the configuration takes effect

```
target:~$ systemctl restart systemd-networkd
```

Creating a WLAN Access Point

This section provides a basic access point (AP) configuration for a secured *WPA2* network.

Find the name of the WLAN interface with

```
target:~$ ip link
```

Edit the configuration in */etc/hostapd.conf*. It is strongly dependent on the use case. The following shows an example

```
# file /etc/hostapd.conf
interface=wlan0
driver=nl80211
ieee80211d=1
country_code=DE
hw_mode=g
ieee80211n=1
ssid=Test-Wifi
channel=2
wpa=2
wpa_passphrase=12345678
wpa_key_mgmt=WPA-PSK
wpa_pairwise=CCMP
```

Set up and start the DHCP server for the network interface *wlan0* via *systemd-networkd*

```
target:~$ mkdir -p /etc/systemd/network/
target:~$ vi /etc/systemd/network/10-wlan0.network
```

Insert the following text into the file

```
[Match]
Name=wlan0

[Network]
Address=192.168.0.1/24
DHCPServer=yes

[DHCPServer]
EmitDNS=yes
target:~$ systemctl restart systemd-networkd
target:~$ systemctl status  systemd-networkd -l   # check status and see errors
```

Start the userspace daemon *hostapd*

```
target:~$ systemctl start hostapd
target:~$ systemctl status hostapd -l # check for errors
```

Now, you should see the WLAN network *Test-Wifi* on your terminal device (laptop, smartphone, etc.).

If there are problems with the access point, you can either check the log messages with

```
target:~$ journalctl --unit=hostapd
```

or start the daemon in debugging mode from the command line

```
target:~$ systemctl stop hostapd
target:~$ hostapd -d /etc/hostapd.conf -P /var/run/hostapd.pid
```

You should see

```
...
wlan0: interface state UNINITIALIZED->ENABLED
wlan0: AP-ENABLED
```

Further information about AP settings and the userspace daemon *hostapd* can be found at

```
https://wireless.wiki.kernel.org/en/users/documentation/hostapd
https://w1.fi/hostapd/
```

phyCORE-i.MX 6UL/ULL Bluetooth

Special consideration must be paid when working with any Bluetooth on a phyCORE-i.MX 6UL/ULL. For further information, please check L-844e.A5 i.MX 6UL/ULL BSP Manual - Bluetooth.

13.6.18 Add OpenCV Libraries and Examples

OpenCV (Opensource Computer Vision https://opencv.org/) is an open-source library for computer vision applications.

To install the libraries and examples edit the file *conf/local.conf* in the *Yocto* build system and add

```
# file conf/local.conf
# Installing OpenCV libraries and examples
LICENSE_FLAGS_ACCEPTED += "commercial_libav"
LICENSE_FLAGS_ACCEPTED += "commercial_x264"
IMAGE_INSTALL:append = " \
    opencv \
    opencv-samples \
    libopencv-calib3d2.4 \
    libopencv-contrib2.4 \
    libopencv-core2.4 \
    libopencv-flann2.4 \
    libopencv-gpu2.4 \
    libopencv-highgui2.4 \
    libopencv-imgproc2.4 \
    libopencv-legacy2.4 \
    libopencv-ml2.4 \
    libopencv-nonfree2.4 \
    libopencv-objdetect2.4 \
    libopencv-ocl2.4 \
    libopencv-photo2.4 \
    libopencv-stitching2.4 \
    libopencv-superres2.4 \
    libopencv-video2.4 \
```

(continues on next page)

(continued from previous page)

```
    libopencv-videostab2.4 \
"
```

Then rebuild your image

```
host:~$ bitbake phytec-qt6demo-image
```

Tip

Most examples do not work out of the box, because they depend on the *GTK* graphics library. The BSP only supports *Qt6* .

13.6.19 Add Minimal PHP web runtime with *lightpd*

This is one example of how to add a small runtime for PHP applications and a web server on your target. Lighttpd can be used together with the PHP command line tool over cgi. This solution weights only 5.5 MiB of disk storage. It is already preconfigured in meta-ampliphy. Just modify the build configuration to install it on the image

```
# file conf/local.conf
# install lighttpd with php cgi module
IMAGE_INSTALL:append = " lighttpd"
```

After booting the image, you should find the example web content in */www/pages* . For testing php, you can delete the *index.html* and replace it with a *index.php* file

```
<html>
  <head>
    <title>PHP-Test</title>
  </head>
  <body>
    <?php phpinfo(); ?>
  </body>
</html>
```

On your host, you can point your browser to the board's IP, (e.g. 192.168.3.11) and the phpinfo should show up.

13.7 Common Tasks

13.7.1 Debugging a User Space Application

The phytec-qt6demo-image can be cross-debugged without any change. For cross-debugging, you just have to match the host sysroot with the image in use. So you need to create a toolchain for your image

```
host:~$ bitbake -c populate_sdk phytec-qt6demo-image
```

Additionally, if you want to have full debug and backtrace capabilities for all programs and libraries in the image, you could add

```
DEBUG_BUILD = "1"
```

to the `conf/local.conf`. This is not necessary in all cases. The compiler options will then be switched from FULL_OPTIMIZATION to DEBUG_OPTIMIZATION. Look at the *Poky* source code for the default assignment of DEBUG_OPTIMIZATION.

To start a cross debug session, install the SDK as mentioned previously, source the SDK environment, and run *Qt Creator* in the same shell. If you do not use *Qt Creator*, you can directly call the arm-<..>-gdb debugger instead which should be in your path after sourcing the environment script.

If you work with *Qt Creator*, have a look at the appropriate documentation delivered with your product (either QuickStart or Application Guide) for information on how to set up the toolchain.

When starting the debugger with your userspace application you will get a SIGILL, an illegal instruction from the *libcrypto*. *Openssl* probes for the system capabilities by trapping illegal instructions, which will trigger *GDB*. You can ignore this and hit **Continue** (c command). You can permanently ignore this stop by adding

```
handle SIGILL nostop
```

to your *GDB* startup script or in the *Qt Creator GDB* configuration panel. Secondly, you might need to disable a security feature by adding

```
set auto-load safe-path /
```

to the same startup script, which will enable the automatic loading of libraries from any location.

If you need to have native debugging, you might want to install the debug symbols on the target. You can do this by adding the following line to your *conf/local.conf*

```
EXTRA_IMAGE_FEATURES += "dbg-pkgs"
```

For cross-debugging, this is not required as the debug symbols will be loaded from the host side and the dbg-pkgs are included in the SDK of your image anyway.

13.7.2 Generating Source Mirrors, working Offline

Modify your *site.conf* (or *local.conf* if you do not use a *site.conf*) as follows

```
#DL_DIR ?= "" don't set it! It will default to a directory inside /build
SOURCE_MIRROR_URL = "file:///home/share/yocto_downloads/"
INHERIT += "own-mirrors"
BB_GENERATE_MIRROR_TARBALLS = "1"
```

Now run

```
host:~$ bitbake --runall=fetch <image>
```

for all images and for all machines you want to provide sources for. This will create all the necessary *tar* archives. We can remove all SCM subfolders, as they are duplicated with the tarballs

```
host:~$ rm -rf build/download/git2/
etc...
```

Please consider that we used a local source mirror for generating the dl_dir. Because of that, some archives will be linked locally.

First, we need to copy all files, resolving symbolic links into the new mirror directory

```
host:~$ rsync -vaL <dl_dir> ${TOPDIR}/../src_mirror/
```

Now we clean the */build* directory by deleting everything except */build/conf/* but including */build/conf/sanity*. We change *site.conf* as follows

```
SOURCE_MIRROR_URL = "file://${TOPDIR}/../src_mirror"
INHERIT += "own-mirrors"
BB_NO_NETWORK = "1"
SCONF_VERSION = "1"
```

The BSP directory can now be compressed with

```
host:~$ tar cfJ <filename>.tar.xz <folder>
```

where filename and folder should be the full BSP Name.

13.7.3 Compiling on the Target

To your *local.conf* add

```
IMAGE_FEATURES:append = " tools-sdk dev-pkgs"
```

13.7.4 Different Toolchains

There are several ways to create a toolchain installer in *Poky*. One option is to run

```
host:~$ bitbake meta-toolchain
```

This will generate a toolchain installer in *build/deploy/sdk* which can be used for cross-compiling of target applications. However, the installer does not include libraries added to your image, so it is a bare *GCC* compiler only. This is suited for bootloader and kernel development.

Another you can run is

```
host:~$ bitbake -c populate_sdk <your_image>
```

This will generate a toolchain installer containing all necessary development packages of the software installed on the root filesystem of the target. This installer can be handed over to the user space application development team and includes all necessary parts to develop an application. If the image contains the *QT* libraries, all of those will be available in the installer too.

The third option is to create the ADT (Application Development Toolkit) installer. It will contain the cross-toolchain and some tools to aid the software developers, for example, an *Eclipse* plugin and a *QEMU* target simulator.

```
host:~$ bitbake adt-installer
```

The ADT is untested for our BSP at the moment.

Using the SDK

After generating the SDK with

```
host:~$ source sources/poky/oe-init-build-env
host:~$ bitbake -c populate_sdk phytec-qt6demo-image # or another image
```

run the generated binary with

```
host:~$ deploy/sdk/ampliphy-glibc-x86_64-phytec-qt6demo-image-cortexa9hf-vfp-neon-toolchain-i.
↪MX6-PD15.3-rc.sh
Enter target directory for SDK (default: /opt/ampliphy/i.MX6-PD15.3-rc):
You are about to install the SDK to "/opt/ampliphy/i.MX6-PD15.3-rc". Proceed[Y/n]?
Extracting SDK...done
Setting it up...done
SDK has been successfully set up and is ready to be used.
```

You can activate the toolchain for your shell by sourcing the file *environment-setup* in the toolchain directory

```
host:~$ source /opt/ampliphy/i.MX6-PD15.3-rc/environment-setup-cortexa9hf-vfp-neon-phytec-linux-
↪gnueabi
```

Then the necessary tools like the cross compiler and linker are in your PATH. To compile a simple *C* program, use

```
host:~$ $CC main.c -o main
```

The environment variable $CC contains the path to the ARM cross compiler and other compiler arguments needed like *-march* , *-sysroot* and *–mfloat-abi*.

Tip

You cannot compile programs only with the compiler name like

```
host:~$ arm-phytec-linux-gnueabi-gcc main.c -o main
```

It will fail in many cases. Always use *CC*, CFLAGS, LDFLAGS, and so on.

For convenience, the *environment-setup* exports other environment variables like CXX, LD, SDKTARGET-SYSROOT.

A simple makefile compiling a *C* and *C++* program may look like this

```
# Makefile
TARGETS=c-program cpp-program

all: $(TARGETS)

c-program: c-program.c
    $(CC) $(CFLAGS) $(LDFLAGS) $< -o $@

cpp-program: cpp-program.cpp
    $(CXX) $(CXXFLAGS) $(LDFLAGS) $< -o $@

.PHONY: clean
clean:
    rm -f $(TARGETS)
```

To compile for the target, just source the toolchain in your shell before executing make

```
host:~$ make     # Compiling with host CC, CXX for host architecture
host:~$ source /opt/ampliphy/i.MX6-PD15.3-rc/environment-setup-cortexa9hf-vfp-neon-phytec-linux-
↪gnueabi
host:~$ make     # Compiling with target CC, CXX for target architecture
```

If you need to specify additionally included directories in the sysroot of the toolchain, you can use an '=' sign in the *-I* argument like

```
-I=/usr/include/SDL
```

GCC replaces it by the sysroot path (here */opt/ampliphy/i.MX6-PD15.3-rc/sysroots/cortexa9hf-vfp-neon-phytec-linux-gnueabi/*). See the main page of *GCC* for more information.

Tip

The variables $CFLAGS and $CXXFLAGS contain the compiler debug flag '-g' by default. This includes debugging information in the binary and making it bigger. Those should be removed from the production image. If you create a *Bitbake* recipe, the default behavior is to turn on '-g' too. The debugging symbols are used in the SDK rootfs to be able to get debugging information when invoking *GDB* from the host. Before installing the package to the target rootfs, *Bitbake* will invoke *strip* on the program which removes the debugging symbols. By default, they are not found nor required on the target root filesystem

Using the SDK with GNU Autotools

Yocto SDK is a straightforward tool for a project that uses the *GNU Autotools*. The traditional compile steps for the host are usually

```
host:~$ ./autogen.sh # maybe not needed
host:~$ ./configure
host:~$ make
host:~$ make install DESTDIR=$PWD/build/
```

The commands to compile for the target machine with the *Yocto* SDK are quite similar. The following commands assume that the SDK was unpacked to the directory */opt/phytec-ampliphy/i.MX6-PD15.3.0/* (adapt the path as needed)

```
host:~$ source /opt/phytec-ampliphy/i.MX6-PD15.3.0/environment-setup-cortexa9hf-vfp-neon-phytec-
↪linux-gnueabi
host:~$ ./autogen.sh  # maybe not needed
host:~$ ./configure ${CONFIGURE_FLAGS}
host:~$ make
host:~$ make install DESTDIR=$PWD/build/
```

Refer to the official *Yocto* documentation for more information: https://docs.yoctoproject.org/4.2.4/singleindex.html#autotools-based-projects

13.7.5 Working with Kernel Modules

You will come to the point where you either need to set some options for a kernel module or you want to blacklist a module. Those things are handled by *udev* and go into **.conf* files in

```
/etc/modprobe.d/\*.conf.
```

If you want to specify an option at build time, there are three relevant variables. If you just want to autoload a module that has no autoload capabilities, add it to

```
KERNEL_MODULE_AUTOLOAD += "your-module"
```

either in the kernel recipe or in the global variable scope. If you need to specify options for a module, you can do so with

```
KERNEL_MODULE_AUTOLOAD += "your-module"
KERNEL_MODULE_PROBECONF += "your-module"
module_conf_your-module = "options your-module parametername=parametervalue"
```

if you want to blacklist a module from autoloading, you can do it intuitively with

```
KERNEL_MODULE_AUTOLOAD += "your-module"
KERNEL_MODULE_PROBECONF += "your-module"
module_conf_your-module = "blacklist your-module"
```

13.7.6 Working with *udev*

Udev (Linux dynamic device management) is a system daemon that handles dynamic device management in /dev. It is controlled by *udev* rules that are located in */etc/udev/rules.d* (sysadmin configuration space) and */lib/udev/rules.d/* (vendor-provided). Here is an example of an *udev* rule file

```
# file /etc/udev/rules.d/touchscreen.rules
# Create a symlink to any touchscreen input device
SUBSYSTEM=="input", KERNEL=="event[0-9]*", ATTRS{modalias}=="input:*-e0*,3,*a0,1,*18,*",␣
↪SYMLINK+="input/touchscreen0"
SUBSYSTEM=="input", KERNEL=="event[0-9]*", ATTRS{modalias}=="ads7846", SYMLINK+="input/
↪touchscreen0"
```

See https://www.freedesktop.org/software/systemd/man/latest/udev.html for more details about the syntax and usage. To get the list of attributes for a specific device that can be used in an *udev* rule you can use the *udevadm info* tool. It prints all existing attributes of the device node and its parents. The key-value pairs from the output can be copied and pasted into a rule file. Some examples

```
target:~$ udevadm info -a /dev/mmcblk0
target:~$ udevadm info -a /dev/v4l-subdev25
target:~$ udevadm info -a -p /sys/class/net/eth0
```

After changing an *udev* rule, you have to notify the daemon. Otherwise, your changes are not reflected. Use the following command

```
target:~$ udevadm control --reload-rules
```

While developing *udev* rules you should monitor the events in order to see when devices are attached or unattached to the system. Use

```
target:~$ udevadm monitor
```

Furthermore, it is very useful to monitor the system log in another shell, especially if the rule executes external scripts. Execute

```
target:~$ journalctl -f
```

Tip

You cannot start daemons or heavy scripts in a *RUN* attribute. See https://www.freedesktop.org/software/systemd/man/latest/udev.html#RUN%7Btype%7D .

This can only be used for very short-running foreground tasks. Running an event process for a long period of time may block all further events for this or a dependent device. Starting daemons or other long-running processes is not appropriate for *udev*; the forked processes, detached or not, will be unconditionally killed after the event handling has finished. You can use the special attribute *ENV{SYSTEMD_WANTS}="service-name.service"* and a *systemd*service instead.

See https://unix.stackexchange.com/questions/63232/what-is-the-correct-way-to-write-a-udev-rule-to-stop-a-service-unde

CHAPTER FOURTEEN

TROUBLESHOOTING

14.1 Setscene Task Warning

This warning occurs when the Yocto cache is in a dirty state.

```
WARNING: Setscene task X ([...]) failed with exit code '1' - real task
```

You should avoid canceling the build process or if you have to, press Ctrl-C once and wait until the build process has stopped. To remove all these warnings just clean the sstate cache and remove the build folders.

```
host:~$ bitbake phytec-headless-image -c cleansstate && rm -rf tmp deploy/ipk
```

CHAPTER

FIFTEEN

YOCTO DOCUMENTATION

The most important piece of documentation for a BSP user is probably the developer manual. https://docs.yoctoproject.org/4.2.4/dev-manual/index.html

The chapter about common tasks is a good starting point. https://docs.yoctoproject.org/4.2.4/dev-manual/layers.html#understanding-and-creating-layers

The complete documentation is available on one single HTML page, which is good for searching for a feature or a variable name. https://docs.yoctoproject.org/4.2.4/singleindex.html