

Yocto

Reference Manual

Document No.: **L-813e_1**

Release No.: **AM335x PD15.1.0**
i.MX 6 PD15.1.0

Edition: August 2015

Copyrighted products are not explicitly indicated in this manual. The absence of the trademark (™, or ®) and copyright (©) symbols does not imply that a product is not protected. Additionally, registered patents and trademarks are similarly not expressly indicated in this manual.

The information in this document has been carefully checked and is considered to be entirely reliable. However, PHYTEC Messtechnik GmbH assumes no responsibility for any inaccuracies. PHYTEC Messtechnik GmbH neither gives any guarantee nor accepts any liability whatsoever for consequential damages resulting from the use of this manual or its associated product. PHYTEC Messtechnik GmbH reserves the right to alter the information contained herein without prior notification and accepts no responsibility for any damages that might result.

Additionally, PHYTEC Messtechnik GmbH offers no guarantee nor accepts any liability for damages arising from the improper usage or improper installation of the hardware or software. PHYTEC Messtechnik GmbH further reserves the right to alter the layout and/or design of the hardware without prior notification and accepts no liability for doing so.

© Copyright 2015 PHYTEC Messtechnik GmbH, D-55129 Mainz.

Rights - including those of translation, reprint, broadcast, photomechanical or similar reproduction and storage or processing in computer systems, in whole or in part - are reserved. No reproduction may occur without the express written consent from PHYTEC Messtechnik GmbH.

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

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

List of Figures	ii
Conventions, Abbreviations and Acronyms	iii
1 The Yocto Project	1
1.1 Introduction	1
1.2 Core Components	1
1.3 Vocabulary	2
1.3.1 Recipes	2
1.3.2 Classes	2
1.3.3 Layers.....	2
1.3.4 Machine	2
1.3.5 Distro	2
1.4 Poky	3
1.4.1 Bitbake	3
1.4.2 Toaster	3
1.5 Official Documentation.....	3
2 Compatible Linux Distributions	4
3 Introduction to the Phytex BSP	5
3.1 BSP Structure	5
3.1.1 BSP Management	5
3.1.1.1 phyLinux.....	5
3.1.1.2 Repo	5
3.1.2 BSP Meta Data	6
3.1.2.1 Poky.....	6
3.1.2.2 meta-openembedded	6
3.1.2.3 meta-qt5.....	6
3.1.2.4 meta-phytec	7
3.1.2.5 meta-phytec/meta-phy<SOC>	7
3.1.2.6 meta-yogurt	7
3.2 Build Configuration	7
4 Installation	8
4.1 Setting up the Host	8
4.2 Git Configuration	8
4.3 site.conf Setup.....	9
5 phyLinux Documentation	10
5.1 Get phyLinux.....	10
5.2 Basic Usage	10
5.3 Initialization	11
5.4 Advanced Usage	12
6 Working with Poky and Bitbake	13
6.1 Start the Build.....	13
6.2 Images	13
6.3 Installing the SDK.....	14
6.4 Accessing Development States between Releases	14
6.5 BSP Features of meta-phytec and meta-yogurt.....	15
6.5.1 Buildinfo.....	15

- 6.6 Customizing the BSP..... 16
 - 6.6.1 Adding existing Community Software or Libraries to the BSP Image..... 16
 - 6.6.2 Creating Your own Layer 17
 - 6.6.3 Adding existing Software Part 2 17
 - 6.6.4 Inspect Your Configuration 18
 - 6.6.5 Kernel and Barebox Configuration and Patching..... 18
 - 6.6.5.1 Option 1: Quick and Dirty - Modify before Configure/Compile 19
 - 6.6.5.2 Option 2: Safe and Clean - Change SRC_URI in local.conf 20
 - 6.6.5.3 Option 3: Use phyexternalsrc..... 21
 - 6.6.6 Configuring the Kernel..... 22
 - 6.6.7 Apply the defconfig/patch to the Kernel 22
- 6.7 Common Tasks..... 23
 - 6.7.1 Debugging a Userspace Application 23
 - 6.7.2 Generating Source Mirrors, Working Offline 24
 - 6.7.3 Compiling on the Target 24
 - 6.7.4 Different Toolchains 25
 - 6.7.5 Working with Kernel Modules 25
- 7 Yocto Documentation..... 26**
- 8 Revision History..... 27**

List of Figures

- Figure 1: BSP-Yocto Repository Graph 6



Conventions, Abbreviations and Acronyms

This hardware manual describes the PB-00802-xxx Single Board Computer (SBC) in the following referred to as phyBOARD-Wega AM335x. The manual specifies the phyBOARD-Wega AM335x's design and function. Precise specifications for the Texas Instruments AM335x microcontrollers can be found in the Texas Instrumenten's AM335x Data Sheet and Technical Reference Manual.

Conventions

The conventions used in this manual are as follows:

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

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

1 The Yocto Project

1.1 Introduction

Yocto is the smallest SI metric system prefix. Like m stands for milli = 10^{-3} , 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 project website <http://www.yoctoproject.org/> 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. It contains the tools to create a *Linux* distribution specially fitted to the product requirements. The most important step to bring order in 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 the versioning scheme.

The release process is similar to the Linux Kernel. *Yocto* increases its version number every six month and gives the release a name. The release list can be found here:

<https://wiki.yoctoproject.org/wiki/Releases>

1.2 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 sub-projects: rt-kernel, vendor patches
 - Infrastructure provided by Wind River
 - Alternative: classic kernel – we use classic and not *Yocto* kernel
- *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*

1.3 Vocabulary

1.3.1 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, 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 bash code.

1.3.2 Classes

Classes combine functionality used inside Recipes into reusable blocks.

1.3.3 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.

<http://layers.openembedded.org/layerindex/branch/master/layers/>

1.3.4 Machine

Machines are configuration variables, which describe the aspects of the target hardware.

1.3.5 Distro

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

1.4 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

1.4.1 Bitbake

Bitbake is the task scheduler. It is written in *Python* and interprets Recipes which contain code in *Bitbake*'s own programming language, *Python* and bash code. The official documentation can be found here.

<http://www.yoctoproject.org/docs/current/bitbake-user-manual/bitbake-user-manual.html>

1.4.2 Toaster

Toaster is a web frontend for *Bitbake* to investigate the build history and statistics. It is planned that it grows into a build management frontend for *Bitbake*. It is not yet complete in its features and under heavy development, but you can already keep an eye on the project.

<http://www.yoctoproject.org/docs/current/toaster-manual/toaster-manual.html#toaster-manual-intro>

1.5 Official Documentation

For more general questions about *Bitbake* and *Poky* consult the mega-manual:

<http://www.yoctoproject.org/docs/latest/mega-manual/mega-manual.html>

2 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 for the specific *Yocto* version. For 1.7 it can be found here:

<http://www.yoctoproject.org/docs/1.7/ref-manual/ref-manual.html#detailed-supported-distros>

3 Introduction to the Phyttec BSP

3.1 BSP Structure

The BSP consists roughly of three parts. BSP management, BSP meta data and BSP content. The management consists of *repo* and *phyLinux*, the meta data depends on the soc and describes how to build the software and the content are Phyttec's *Git* repositories and external sources.

3.1.1 BSP Management

Yocto is an umbrella project. Naturally, this will force the user to base his work on several external repositories. They need to be managed in a deterministic way. The *Repo* tool is one way of managing *Git* repository tasks in a more comfortable way. Phyttec's Yocto BSP is managed with *repo*. This provides us with a method to deliver fixed releases as well as rolling releases.

3.1.1.1 phyLinux

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

3.1.1.2 Repo

Repo is a wrapper around the *Repo* tool set. 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 *Google's Git* server in a specific version to the `.repo/repo` directory. Next time you run *Repo*, all the commands will be available. So be aware of the fact, that the *Repo* version in different build directories can drift apart over the years if you do not run *Repo sync*. Also if you store stuff for the archives, you need to include the complete `.repo` folder.

Repo expects a *Git* repository which will be parsed from the command line. In case of our BSP, we called it *phy²octo*, derived from Phyttec's *Yocto*, *phyyocto*, *phy²octo*. In this repository the whole 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 it states, called "projects". The *Git* repositories can be checked out in different states. The revision field can be a branch, tag or commit id of a repository. So the state of the software is not necessarily unique, e.g. the HEAD of a branch, 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 therefore 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 SOC platform. That is why you

have to choose the SOC you are using. The selected SOC will define the branch of the phy²octo *Git* repository which will be checked out for the manifest selection.

3.1.2 BSP Meta Data

We include several third party layers in our BSP to get a complete *Linux* distribution up and running without the need of integrating external projects at the beginning. All used repositories are shown on the left of *Figure 1* and are describe in the following section.

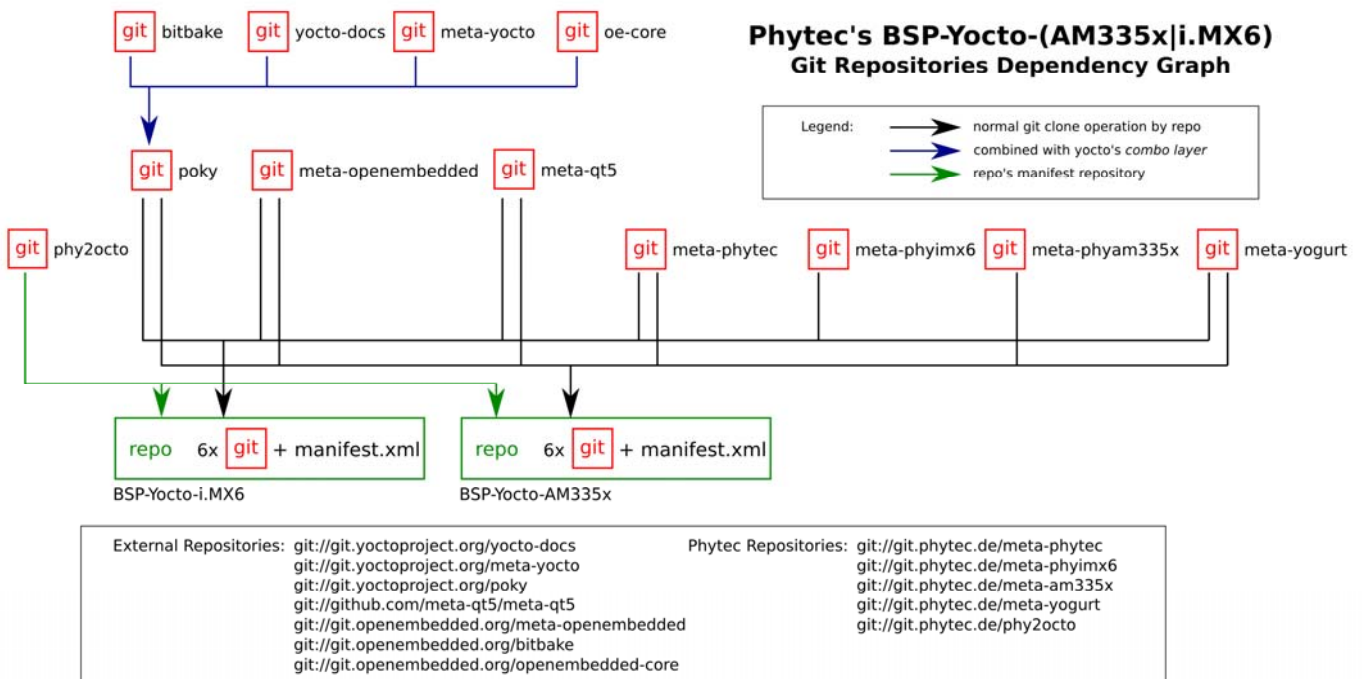


Figure 1: BSP-Yocto Repository Graph

3.1.2.1 Poky

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

3.1.2.2 meta-openembedded

OpenEmbedded is a collection of different layers containing the meta description for a lot of 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.

3.1.2.3 meta-qt5

This layer provides a community supported integration of qt5 in *Poky* based rootfs and is integrated in our BSP.

3.1.2.4 meta-phytec

This layer contains common features for all our BSPs. And is the core of our BSP, together with the SOC layers. Only those two parts are required if you want to integrate our BSP in your existing *Yocto* workflow.

3.1.2.5 meta-phytec/meta-phy<SOC>

Those layers define the barebox, kernel and software configuration for specific boards. The boards get defined in the machine config files in *conf/machine*.

3.1.2.6 meta-yogurt

This is our example distribution. It extends the basic configuration of *Poky* with software projects described by all the other BSP components. It provides a base for some development scenarios. A configuration for *systemd* is provided.

3.2 Build Configuration

The BSP initializes a build folder which will contain all files you create by running bitbake commands. It contains a conf folder which 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 will be preconfigured in *local.conf* when you check out the BSP using phyLinux.

In short: *DISTRO* defines the software configuration, *MACHINE* defines the hardware configuration.

As *DISTRO* we deliver "yogurt" with our BSP. This distribution will be preselected and gives you a starting point for implementing your own configuration.

A *MACHINE* defines a binary image which supports specific hardware combinations of module and baseboard. Have a look at the *machine.conf* file or our webpage for a description of the hardware.

4 Installation

4.1 Setting up the Host

You need to have a running Linux distribution at your hand. It should be running on a powerful machine, as a lot of compiling will be done on it. *Yocto* needs a handful of additional packages on your host. For Ubuntu 14.04 you need:

```
sudo apt-get install gawk wget git-core diffstat unzip texinfo \  
                    gcc-multilib build-essential chrpath socat \  
                    libsdl1.2-dev xterm
```

For the other distributions you can find information in the *Yocto* Quickstart:

<http://www.yoctoproject.org/docs/latest/yocto-project-qs/yocto-project-qs.html>

4.2 Git Configuration

The BSP is heavily based on *Git*. *Git* needs some information from you as a user to be able to identify which changes were done by whom. If you do not have one, create a `~/.gitconfig`. Here is an example:

```
[user]
  name = <Your Name>
  email = <Your Mail>@phytec.de
[core]
  editor = vim
[merge]
  tool = vimdiff
[alias]
  co = checkout
  br = branch
  ci = commit
  st = status
  unstage = reset HEAD --
  last = log -1 HEAD
[push]
  default = current
[color]
  ui = auto
[sendemail]
  smtpserver = idefix.phytec.de
  smtpserverport = 25
```

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

```
git config --global user.email your_email@example.com
git config --global user.name "name surname"
```

4.3 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. It is not a precondition to do this, but strongly recommended, as it will help a lot with build times in the development process.

The download directory is a place where *Yocto* stores all sources fetched from the internet. It can contain tar.gz, *Git* mirror or anything else. It is very useful to set this to a common shared location on the machine. Create this directory with 777 access rights. To be able to share this directory between 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:

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

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

```
sudo find /home/share/ -perm /u=w ! -perm /g=w -exec chmod g+w {} \;
sudo find /home/share/ -perm /u=w ! -perm /g=w -exec chown g+w {} \;
sudo find /home/share/ -perm /u=r ! -perm /g=r -exec chmod g+r {} \;
```

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

Create the two directories on a drive where you have approximately 50 GB of space and assign the following 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, have a look at the documented example settings:

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

5 phyLinux Documentation

Documentation for version: PD15.1.0

The phyLinux script is a basic management tool for Phytec *Yocto* BSP releases written in *Python* by Stefan Müller-Klieser. 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 to 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.

5.1 Get phyLinux

The phyLinux script can be found on the PHYTEC ftp server:

<ftp://ftp.phytec.de/pub/Software/Linux/Yocto/Tools/phyLinux>

5.2 Basic Usage

For the basic usage of phyLinux, type:

```
./phyLinux --help
```

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

This Programs sets up an environment to work with The Yocto Project on Phytects 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

5.3 Initialization

Create a fresh project folder, e.g.:

```
mkdir ~/yocto
```

and run phyLinux from the new folder:

```
./phyLinux init
```

A clean folder is important, because phyLinux will clean its working directory. So all files will be removed after the clean up.

Calling phyLinux not from an empty directory 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 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, Phytex's BSP release number and the hardware you are working on, e.g.:

```
*****
* Please choose one of the available Machines:
*
* 1 :          beagleboneblack-1 : Hardware Revision A5C 2GiB eMMC
* 2 : phyboard-maia-am335x-1 : PB-00702-002
* 3 : phyboard-wega-am335x-1 : PB-00802-0200C PB-00802-0101C (PEB-AV-01)
* 4 : phyboard-wega-am335x-2 : PB-00802-008 PB-00802-010 (PEB-AV-02)
* 5 :          phycore-am335x-1 : PCM-051-12102F0C.A1/KPCM-953 (Kit)
* 6 :          phycore-am335x-2 : 1GiB RAM, 1GiB NAND variant
* 7 :          phyflex-am335x-1 : PFL-A-03-12113F8I.A1/PBA-B-01
```

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

```
./phyLinux info
```

or view the help command for more information:

```
./phyLinux init --help
```

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

optional arguments:

```
-h, --help          show this help message and exit
--verbose
--no-init           dont execute init after fetch
-o REPOREPO        Use repo tool from another url
-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 and also information for the next steps in the build process.

5.4 Advanced Usage

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

```
./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 a different usage. Use `phyLinux` to check out the states:

```
./phyLinux -u <url-of-your-git-repo>
```

6 Working with Poky and Bitbake

6.1 Start the Build

After you downloaded all the meta data 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:

```
source sources/poky/oe-init-build-env
```

The abbreviation for the source command is a single dot.

```
. sources/poky/oe-init-build-env
```

The current working directory of the shell should change to *build/* and you are now ready to build your first image. We suggest to start with our hardware bring-up image to see if everything is working correctly. If you want, you can separate the download process from the compile process to identify problems of your internet connection. With

```
bitbake -c fetchall phytec-hwbringup-image
```

all external source repositories get pulled into the download directory. With

```
du -sh <DL_DIR>
```

you can see what the download volume was. Now start the compile process.

```
bitbake phytec-hwbringup-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.

6.2 Images

If everything went fine, the images can be found under:

```
cd deploy/images/<MACHINE>
```

The easiest way to test your image is to jumper your board for booting from SD card and to flash the build image to the SD card:

```
sudo dd if=phytec-hwbringup-image-<MACHINE>.sdcard  
of=/dev/<your_device> bs=1MB
```

Where <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!

After booting you can login 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 comment out the line

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

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

6.3 Installing the SDK

To install the SDK for a machine and image type, you can create an SDK installer with *Bitbake* in the BSP itself. Ensure that the correct target machine is set. You also need to pass the image type you want to create the SDK for:

```
bitbake <image> -c populate_sdk
```

This takes 1-5 hours depending on the image type and host machine (like building a BSP). After that you may find the installer in your *Yocto* directory under:

```
build/deploy/sdk
```

Install the SDK with (example):

```
~/yocto/build/deploy/sdk$ ./poky-glibc-x86_64-phytec-qt4demo-image-cortexa8t2hf-vfp-neon-toolchain-1.7.sh
```

Usage example with a simple C file named `bumpRTS.c`

```
source /opt/poky/1.7/environment-setup-cortexa8t2hf-vfp-neon-poky-linux-gnueabi make bumpRTS
```

Creates an arm binary:

```
$ file bumpRTS
bumpRTS: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked
(uses shared libs), for GNU/Linux 2.6.32,
BuildID[sha1]=42d4aa389d09ade2023364e4eef9021080f610f9, not stripped
```

6.4 Accessing Development States between Releases

Special release manifests exist to give you access to 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 by the following command line:

```
./phyLinux -r dizzy
```

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

```
repo sync
```

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

6.5 BSP Features of meta-phytec and meta-yogurt

6.5.1 Buildinfo

The buildinfo task is a feature in our recipes which 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

```
$ 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-2014.11.0-phy2), execute

```
$ mkdir -p ~/git
$ cd ~/git
$ git clone ssh://git@git.phytec.de/barebox-dev barebox
$ cd ~/git/barebox
$ git checkout -b v2014.11.0-phy2-local_development
57b87aedbf0b6ae0eb0b858dd0c83411097c777a
```

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 = "v2014.11.0-phy2-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 phyexternalsrc.bbclass. To use it copy and paste the following snippet to your "local.conf":

```
INHERIT += "phyexternalsrc"
EXTERNALSRC_pn-barebox = "${HOME}/git/barebox"
```

```
EXTERNALSRC_BUILD_pn-barebox = "${HOME}/git/barebox"
```

Note: All the compiling is done in the EXTERNALSRC directory. Everytime 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.



Using phyexternalsrc or externalsrc breaks a lot of *Yocto's* internal dependency mechanism. It's not guaranteed that any changes to the source directory is 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 the barebox and redeploy it to `deploy/images/<machine>` execute:

```
bitbake barebox -c compile --force
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

```
bitbake phytec-qt5demo-image -c rootfs --force
```

6.6 Customizing the BSP

To get you started with the BSP we summarize some basic tasks from the *Yocto* official documentation. At this point all common documentation about *Yocto* applies.

6.6.1 Adding existing Community Software or Libraries to the BSP Image

To add another software to the image have a look at the openembedded layer index at:

<http://layers.openembedded.org/layerindex/>

You can search for a software project name and find out in which layer it can be found. If it is in the meta-openembedded or openembedded core layer, the recipe is already in your build tree. Check in `build/conf/bblayers.conf` if the corresponding meta layer gets parsed to *Bitbake*. Than you just add the software to your image by adding the following line into `build/conf/local.conf`

```
IMAGE_INSTALL_append = " <software>"
```



Please note the leading whitespace, which is necessary for the append command.

If you want to add another layer to the build, download it to `sources/` and add it to the `build/conf/bblayers.conf` file.

6.6.2 Creating Your own Layer

Creating your layer should be one of the first tasks when customizing the BSP. You have two basic options. You could either copy and rename our meta-yogurt, or you can create a new layer which will contain your changes. Which way is the better solution, depends on your usecase. meta-yogurt is our example of how to create a custom *Linux* distribution and will be updated in 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 *Yogurt*. If you need to rework a lot of stuff and only need the basic hardware support from Phytex, better copy meta-yogurt, 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 assume that we have an embedded project called "racer" which we will implement using our *Yogurt 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:

```
yocto-layer create racer
```

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

```
mv meta-racer ../sources/
```

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

```
cd ../sources/meta-racer  
git init && git add . && git commit -s
```

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

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

6.6.3 Adding existing Software Part 2

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-yogurt in `meta-yogurt/recipes-images/images/phytec-hwbringup-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-hwbringup-image.bbappend
```

The append will be parsed together with the base recipe, e.g. in the new append you can override 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 to the `IMAGE_INSTALL` variable:

```
IMAGE_INSTALL_append = " rsync"
```

6.6.4 Inspect Your Configuration

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

```
bitbake-layers
```

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

```
bitbake-layers show-append
```

You should now see an entry for the `phytec-hwbringup-image.bb` recipe which has an append in the `meta-racer` layer.

You are now ready to create kernel patches and configurations and add them with appends in your layer.

6.6.5 Kernel and Barebox Configuration and Patching

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

The *Yocto Project* has already some documentation for software developers. You should especially check the 'Kernel Development Manual' for more information about how to configure the kernel.

- [Yocto - Kernel Development Manual](#)
- [Yocto - Development Manual](#)

The following examples will be either for the barebox or the kernel recipe, but they can be adapted to barebox, kernel and any other package in *Yocto*.

6.6.5.1 Option 1: Quick and Dirty - Modify before Configure/Compile

Pro	Contra
No overhead, no extra configuration	Changes are easily overwritten by <i>Yocto</i> (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. You have to start with a clean phytec-hwbringup-image build, so all working directories are on your disk.

Use *Bitbake's devshell* command to jump into the source directory of the recipe. The barebox recipe is:

```
bitbake barebox -c devshell
```

After executing the command, you can alter the source as you wish. When you are finished exit the devshell by typing `exit`.

After leaving the devshell you can recompile the package:

```
bitbake barebox -c compile --force
```

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



Please note that 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.


```
bitbake barbox -c deploy
```

```
# new barebox in e.g. deploy/images/phyflex-imx6-2/barebox.bin
```

```
bitbake phytec-hwbringup-image
```

```
# new sdcard image in e.g. deploy/images/phyflex-imx6-2/
```

```
# phytec-hwbringup-image-phyflex-imx6-2.sdcard
```

	<p>This is the dirty way to alter the source code, because you modify files in <i>Yocto's</i> territory. If you execute a <i>clean</i> e.g. "<code>bitbake barebox -c clean</code>" or <i>Yocto</i> refetches the source code, all your changes are lost!!!</p> <p>To avoid this, you can create a patch and move it to a save location in your layer. For recipes which do not use <i>Git</i> repositories to fetch the source code you may need to use <i>quilt</i> to manage the patch stack. Execute these commands in in the devshell.</p> <pre>git add . # adds all new and modified files to git's staging area git commit -m "dummy commit to create a patch" # creates a commit with a not so useful commit message git format-patch -1 -o ~/ # creates a patch of the last commit and saves it in your home folder</pre>
---	--

Further resources:

<http://www.yoctoproject.org/docs/1.7/dev-manual/dev-manual.html#modifying-temporary-source-code>

6.6.5.2 Option 2: Safe and Clean - Change SRC_URI in local.conf

Pro	Contra
All changes are saved with <i>Git</i>	A lot of working directories in <code>build/tmp-glibc/work/<machine>/<package>/</code>
	You have to commit every change before recompiling
	For each change the toolchain compiles everything from scratch (can be avoided with <i>ccache</i>)

First you need an external checkout of the *Git* repository of barebox or kernel and checkout the current branch. If you do not have one, use the commands:

```
mkdir ~/git
cd ~/git
git clone git://git.phytec.de/barebox
cd barebox
git checkout -b v2015.02.0-phy remotes/origin/v2015.02.0-phy
```

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

```
# Use your own path to the git repository
# NOTE: Branche 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 = "v2015.02.0-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 use the default one (here "v2015.02.0-phy"). Now you should recompile the barebox from your own source:

```
bitbake barebox -c clean
bitbake barebox -c compile
```

Build should be successful because the source wasn't changed yet.

You can alter the source in `~/git/barebox` or for example 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 do that, *Yocto* will not notice that the source code was modified. So execute something like

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

in your repository folder (e.g. `~/git/barebox/`).

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

```
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 the barebox and even create a new SD card image.

```
bitbake barbox -c deploy # new barebox in e.g. deploy/images/phyflex-
                          imx6-2/barebox-phyflex-imx6-2.bin
bitbake phytec-hwbringup-image # new sd-card image in e.g.
deploy/images/phyflex-imx6-2/phytec-hwbringup-image-phyflex-imx6-2.sdcard
```

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

6.6.5.3 Option 3: Use phyexternalsrc

For more information on the phyexternalsrc class see section [6.5.1 "Buildinfo"](#).

6.6.6 Configuring the Kernel

Run menuconfig from bitbake for your kernel. This example assumes the TI kernel, e.g. for AM335x boards:

```
bitbake -c menuconfig linux-ti
```

Configure the kernel and run:

```
bitbake -c savedefconfig linux-ti
```

Now navigate manually into the *tmp/work* directory or run:

```
bitbake -c devshell linux-ti
```

Copy the *defconfig* into your layer. You now need to apply the *defconfig* to your build, to have it permanently stored in your BSP modification.

6.6.7 Apply the defconfig/patch to the Kernel

In your custom meta-racer create a kernel recipe append:

```
./recipes-kernel/linux/linux-ti_%.bbappend
```

This file needs to have an addition to the search path, so that the patches used in this layer can be found by *Bitbake*.

```
FILESEXTRAPATHS_prepend := "${THISDIR}:"
```

The patches and configuration must then be added to the *SRC_URI* variable, e.g.:

```
SRC_URI_append = " \  
    file://defconfig \  
    file://0001-jump-eprom-to-50.patch \"
```

At this point *Bitbake* will search in the folder *meta-racer/recipes-kernel/linux/* and some subfolders for the files listed in *SRC_URI*. The path search order and *OVERRIDE* lookup order determine which *defconfig* will be used for the kernel.

6.7 Common Tasks

6.7.1 Debugging a Userspace Application

The phytec-qt5demo-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 with:

```
bitbake -c populate_sdk phytec-qt5demo-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. Have a 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 *qtcreator* in the same shell. If you do not use *qtcreator* you can directly call the `arm-<..>-gdb` debugger instead, which should be in your path after sourcing the environment script.

If you work with *Qtcreator*, have a look in 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 ignore this stop by adding

```
handle SIGILL nostop
```

to your *GDB* startup script or in the *Qcreator 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 automatic loading of libraries from any location.

If you need to have native debugging you might want 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.

6.7.2 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/ \n
INHERIT += "own-mirrors"
BB_GENERATE_MIRROR_TARBALLS = "1"
```

Now run a

```
bitbake -c fetchall <image>
```

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

```
rm -rf build/download/git2/
etc...
```

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

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

```
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 \n"
INHERIT += "own-mirrors"
BB_NO_NETWORK = "1"
SCONF_VERSION = "1"
```

The BSP directory can now be compressed with

```
tar cfJ <filename>.tar.xz <folder>
```

where *filename* and *folder* should be the full BSP name.

6.7.3 Compiling on the Target

To your *local.conf* add:

```
IMAGE_FEATURES_append = " tools-sdk dev-pkgs"
```

6.7.4 Different Toolchains

There are several ways to create a toolchain installer in *Poky*. First of all you can run:

```
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.

Secondly, you can run:

```
bitbake -c populate_sdk <your_image>
```

This will generate a toolchain installer containing all necessary development packages of the software installed on the rootfs 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 would be to create the ADT (Application Development Toolkit) installer. It will contain the cross-toolchain and additionally some tools to aid the software developers, e.g. an *Eclipse* plugin and a *QEMU* target simulator.

6.7.5 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 buildtime, there are three relevant variables. If you just want to auto load a module which has e.g. no auto load 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 auto loading, you can do it intuitively with:

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

7 Yocto Documentation

The most important piece of the documentation for a BSP user is probably the developer manual.

<http://www.yoctoproject.org/docs/current/dev-manual/dev-manual.html>

Especially the chapter about common tasks is a good starting point.

<http://www.yoctoproject.org/docs/current/dev-manual/dev-manual.html#extendpoky>

The complete documentation is available in one single html page, which is good for searching for a feature or a variable name:

<http://www.yoctoproject.org/docs/current/mega-manual/mega-manual.html>

8 Revision History

Date	Version #	Changes in this manual
14.08.2015	Manual L-813e_1	First edition. Describes the Phytec BSP release PD15.1.0 for i.MX 6 and AM335x products.

Document: Yocto Reference Manual
Document number: L-813e_1, August 2015

How would you improve this manual?

Did you find any mistakes in this manual? _____ page

Submitted by:

Customer number: _____

Name: _____

Company: _____

Address: _____

Return to:

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

