YOCTO

-

an introduction

(getting started with Yocto)

KONTAKTPERSON
Henrik B. Sørensen

ADDRESSE
Teglgårdsvej 637, 2tv
DK-3050 Humlebæk
Denmark

CVR
20461187

info@beech-grove.eu

Preface

This document is meant to give an overview of the Yocto build system. Assuming that the reader has a basic knowledge of Linux, we will cover very basic Git commands. Also, during this document, we will use OpenSuse as a software platform and primarily Open Source tools.

Details on using the realtime version of the Kernel, optimization of the boot time and application development is outside the scope of this document. Hence, these areas will be covered in other documents.

As mentioned, the document will provide an overview. If the reader wishes a more detailed, in-depth, we highly recommend visiting the official documentation at http://www.yoctoproject.org/docs/2.0/yocto-project-qs/yocto-project-qs.html.

About the author

Henrik Sørensen has worked with IT for 15 years developing a variety of software ranging from ATTiny45 based systems and all the way up to grid computing solutions encorporating 120+ connected systems. His software is used in 122 countries and on the International Space Station.

His interest lies within embedded systems and his company does custom software development within this area.

Table of Contents

Why Linux?

There are a number of operating systems out there, so how does one choose which one to use? This document concerns itself with Linux, so let's look at the reasons for selecting Linux as an operating system.

First of all Linux is Open Source, meaning that the entire source code is accessible for review and changes. So why, is this important? Let's say there's a bug in either a driver or any other component of the system, then it's possible for the developer to fix this bug without having to wait for a vendor to submit a patch or a fix. It's also possible to optimize code if needed.

Second, Linux has a far less complicated device driver power model meaning that device driver development is much easier.

Third and maybe the most important reason, Linux can be customized in all ways thought possible. Trimming the Linux Kernel (core) and reducing the number of drivers/modules to load will not only reduce the memory footprint of the system but also speed up the system. Boot time can also be speed up by selecting the load order of various modules and custom applications. An example of a highly optimized boot time is a vehicle application for amongst other things showing a reversing camera, which had a (cold) boot time of below 0.8 seconds before displaying an HD image!

Fourth, wide range of hardware support.

Fifth, since Linux is Open Source, no license fees applies. Nor is there any need for license stickers.

Sixth, some applications require things to happen within a predefined timeframe. Linux is the only real realtime operating system currently on the market. Others have claimed realtime capabilities by defining realtime with the use of very broad terms.

Development Setup

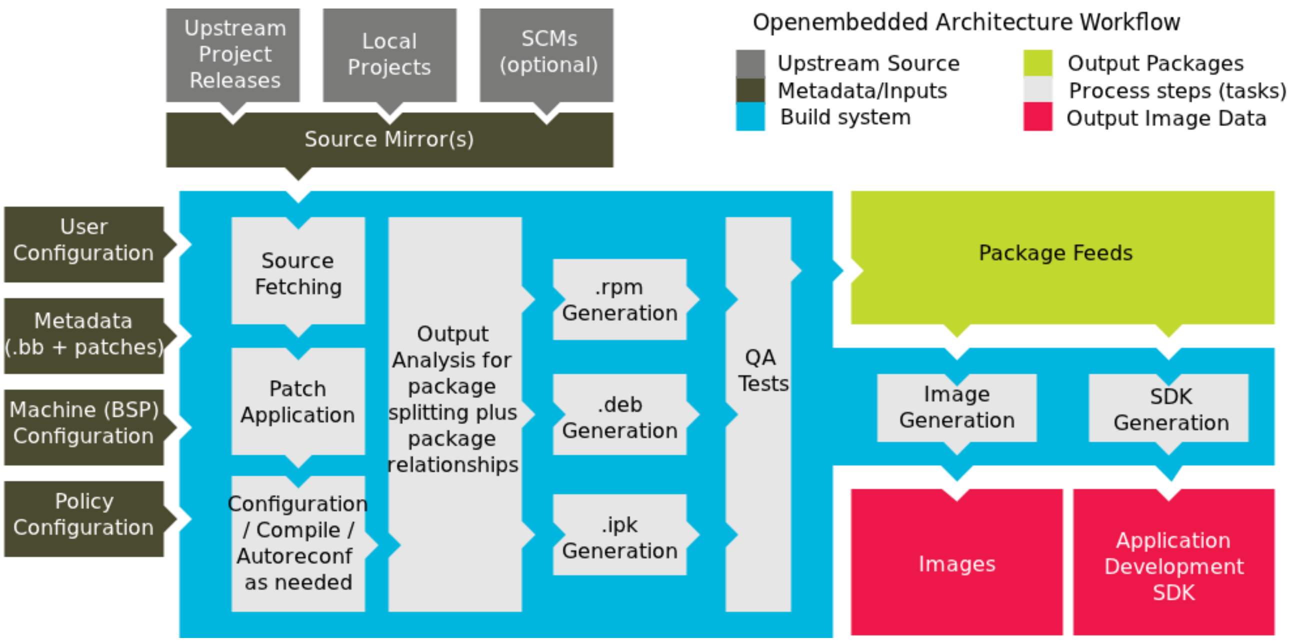
During the course of this document, the examples used will be targetting the Raspberry Pi board. The Raspberry Pi is a small credit card sized embedded system able to run Linux off an SD card. In our setup, a 4.3” LCD screen has been added in order to get a graphical indication of the status of the system. Also, a serial interface and wireless LAN connection can be added in order to get debug information and aid in application development. Also, a number of physical buttons are added for accessing human input.

The reason for selecting the Raspberry Pi is that many people use it for control applications. We had a few laying around from a previous project – development of an automated testing equipment for managing the test of an electro motor. Other candidates might as well be BeagleBone Black or the likes.

At the end of this document, we should be able to boot the system and run a simpel ”Hello World” application. First in simple text mode, and then using a graphic interface.

What is Yocto?

Yocto is a system for building system images for Linux systems based on ARM, x86, MIPS and PowerPC processor architectures. An overview of the system can be found below :



As shown, the system is responsible for fetching source code, applying patches, building the various components and in turn the resulting images and even the Software Development Kits and Application Development Tools. In other words – it provides an automated way of building software and tools necessary for developing embedded systems and applications.

Using BitBake it features a layered approach letting the user add layers to a distribution and reusing other layers generated from other projects/people. This is highly visible when developing, because software is only rebuilt when needed, thus speeding the build process up.
The Yocto system is even able to build the Tool Chains needed for the software development for the particular project. In order to go deep into the Yocto build system, it is necessary to know a set of 5 terms :

Term	Description
BitBake	Build Engine for the Yocto Project Determines dependencies and schedules tasks.
MetaData	Structured organization of BitBake recipes. Layered.
Poky	Reference build system. Based on Yocto and contains tools and projects in order to bootstrap a new distribution.
Recipes	Description of how to fetch, configure, compile and package applications and images.
Layers	A set of recipes

Setting up Yocto

The Yocto build system can either be running on a local machine for development purposes or as recommended : on a build server (which will be the case in the rest of the document). Inside the Yocto system, there's a reference system called Poky, which is used in order to start a new project.

First, we assume the system in question is a Linux system with Git, Python and Tar installed and the system updated. The official recommendation is to use one of the ”bigger” distributions such as Ubuntu, Fedora, openSuse, CentOS or Debian. In our case, Yocto in being installed on a openSuse LEAP 42 running on a Mini-ITX with i7 Quad @ 3.2 GHz, 16 GB RAM.

According to the official documentation, the Yocto requirements can easily be installed using the following (distribution specific) installation command line installations (please note that the \ syntax means that the command line spans multiple lines[1]) :

Ubuntu and Debian

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

Fedora

```
$ sudo dnf install gawk make wget tar bzip2 gzip python unzip perl patch \
diffutils diffstat git cpp gcc gcc-c++ glibc-devel texinfo chrpath \
ccache perl-Data-Dumper perl-Text-ParseWords perl-Thread-Queue socat \
findutils which SDL-devel xterm
```

OpenSUSE

```
$ sudo zypper install python gcc gcc-c++ git chrpath make wget python-xml \
diffstat makeinfo python-curses patch socat libSDL-devel xterm
```

CentOS

```
$ sudo yum install gawk make wget tar bzip2 gzip python unzip perl patch \
diffutils diffstat git cpp gcc gcc-c++ glibc-devel texinfo chrpath socat \
perl-Data-Dumper perl-Text-ParseWords perl-Thread-Queue SDL-devel xterm
```

Moving back to our own build server : In order to get the Yocto and especially Poky, Git is used :

```
git clone git://git.yoctoproject.org/poky -b morty
```

This will take some time depending on the internet connection and system size. On our system it took just above 6 minutes.

When using a standard board (BeagleBoard xM, BeagleBone Black, Intel Editson, Raspberry Pi,

[1] Please consult the documentation for the correct requirements and solve the dependencies as related to the version and type of the operating system.

Variscite Dart etc.), there's a very good chance that someone already has implemented a Board Support Package (BSP). In our case using the Raspberry Pi gives us an advantage, because there's already a Yocto project for the board available at git://git.yoctoproject.org/meta-raspberrypi and the Texas Instruments support at git://git.yoctoproject.org/meta-ti. That means, that it's possible to go to the poky folder and use the git application to fetch the data using :

```
git clone git://git.yoctoproject.org/meta-raspberrypi -b morty

git clone git://git.yoctoproject.org/meta-ti -b morty

git clone git://git.yoctoproject.org/meta-qt5 -b morty
```

The same is done for the various standard boards if needed. Please refer to the Yocto Project website (https://git.yoctoproject.org/) for an overview of all of these boards.

Configuring Yocto

Having setup the Yocto system (with a poky reference system) it is then time to begin using it. Looking at the poky folder, we see a number of folders :

```
bitbake
build
documentation
meta
meta-qt5
meta-raspberrypi
meta-skeleton
meta-ti
meta-yocto
meta-yocto-bsp
oe-init-build-env
scripts
```

Only a few of these are interesting in relation to our use :

Folder name	Description
bitbake	BitBake build engine for building the distribution.
`build`	Main build folder
`documentation`	Documentation
`meta-yocto`	Configuration of the reference distribution
`meta-yocto-*`	BSP[2]s

[2] Board Support Packages – software for utilizing/accessing the hardware on the board. Usually supplied by the manufacturer.

For now, the only folders, we concer ourselves about, are the build and meta-* folders.

Configuration File	Description
`build/conf/bblayers.conf`	Layers are defined/included in this file.
`layer-name/conf/layers.conf`	Contains layer definitions
`build/conf/local.conf`	Local, user defined configuration `BB_NUMBER_THREADS` Maximum number of threads the build system may use `PARALLEL_MAKE` Must match the number of threads. `MACHINE` Which machine configuration to use. `DISTRO` Name of the distrobution configuration to use. `EXTRA_IMAGE_FEATURES` Extra features to add to the images.

As previously mentioned, the BitBake system works using layers, meaning layers of components/software can be added without changing in the current layers. For creating an image or the Raspberry Pi, first we add the BSP to the bblayers.conf file :

```
LCONF_VERSION = "6"

BBPATH = "${TOPDIR}"

BBFILES ?= ""

BBLAYERS ?= " \

  /home/user1/poky/meta \

  /home/user1/poky/meta-yocto \

  /home/user1/poky/meta-yocto-bsp \

  /home/user1/poky/meta-raspberrypi \

  /home/user1/poky/meta-ti \

  "

BBLAYERS_NON_REMOVABLE ?= " \

  /home/user1/poky/meta \

  /home/user1/poky/meta-yocto \
```

As highlighted in the example, we add the BSP to the layer. This includes the Raspberry Pi recipes to the system, allowing the system to build for any of the machines (boards) listed in the recipes.

Now, we turn our attention to the local.conf file. A configuration file containing settings for the local Yocto setup, this file controls such things as which board, the image is built for, which format the image is to be built in and how many threads can be run simultanously.

For our Raspberry Pi image, the following is added/changed in the local.conf file :

```
MACHINE ??= "raspberrypi"

DISTRO ?= "poky"

BB_NUMBER_THREADS = "8"

PARALLEL_MAKE = " -j 8"
```

This means that we use the raspberrypi recipes found in the meta-raspberrypi folder, the ”poky” distrobution image definition and 8 parallel threads meaning 8 components can be compiled at the same time.

At this point, we're ready to invoke the build system. This is done by first initializing the system (please note the syntax) :

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

Parameters (environment variables) are set correctly, and the system is now ready to run. The script will list a number of images, which are available out of the box with the Yocto :

```
Common targets are:

    core-image-minimal

    core-image-sato

    meta-toolchain

    adt-installer

    meta-ide-support
```

However, whenever the user wishes to build another type of image, but the user is not really sure of which images, which can be built, the user can use the following command to locate the desired image configuration :

```
find /home/user1/poky/meta-raspberrypi -name *image*

/home/user1/poky/meta-raspberrypi/classes/sdcard_image-rpi.bbclass

/home/user1/poky/meta-raspberrypi/recipes-core/images

/home/user1/poky/meta-raspberrypi/recipes-core/images/rpi-basic-image.bb

/home/user1/poky/meta-raspberrypi/recipes-core/images/rpi-hwup-image.bb
```

In order to test a standard image, the following command line is called :

```
bitbake rpi-hwup-image
```

This will take a long time depending on the build system, and the result can be found in the

`build/tmp/deploy/images` folder. In this case, the output file can be found using the link rpi-hwup-image-raspberrypi.rpi-sdimg pointing to a 120 MB file ready to be loaded onto an SDCard and inserted into the Raspberry Pi. The image is currently not optimized for speed nor size. We use Win32DiskManager for this. You can also use the dd command.

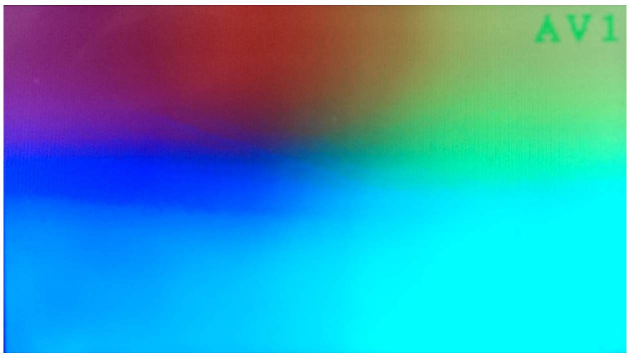
Booting the image

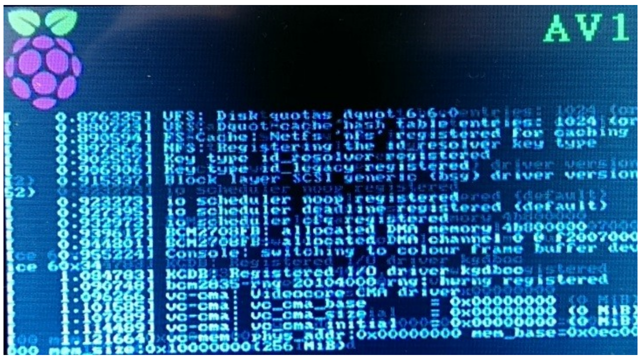
Before booting the image, there are 2 files, which need to be edited in order to make the system boot correctly, cmdline.txt and config.txt. To begin with, we'll start with configuring the system by editing the config.txt.

```
sdtv_mode=0 #for NTSC

sdtv_aspect=3 #for 16:9

framebuffer_width=480

framebuffer_height=272

disable_overscan=1 #non-intuitive but the overscan_scale does the work

overscan_scale=1 #currently undocumented but works

overscan_left=32

overscan_right=32

overscan_top=20

overscan_bottom=10
```

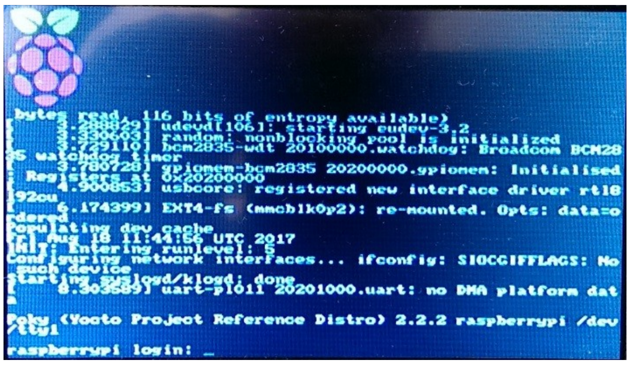
This configuration sets up the screen for the resolution (480x272 pixels[2]) and other display specific settings.

Boot sequence takes around 12 seconds and looks like this:





```
[    0.876335] VFS: Disk quotas dquot_6.6.0
[    0.880724] VFS: Dquot-cache hash table entries: 1024 (or
[    0.890594] FS-Cache: Netfs 'nfs' registered for caching
[    0.897366] NFS: Registering the id_resolver key type
[    0.902557] Key type id_resolver registered
[    0.906906] Key type id_legacy registered
[    0.915337] Block layer SCSI generic (bsg) driver version
[    0.923273] io scheduler noop registered
[    0.927395] io scheduler deadline registered (default)
[    0.932973] io scheduler cfq registered
[    0.939612] BCM2708FB: allocated DMA memory 4b800000
[    0.944801] BCM2708FB: allocated DMA channel 0 @ f2007000
[    0.955224] Console: switching to colour frame buffer dev
ice 60x34
[    1.084783] KGDB: Registered I/O driver kgdboc
[    1.090748] bcm2835-rng 20104000.rng: hwrng registered
[    1.096266] vc-cma: Videocore CMA driver
[    1.101685] vc-cma: vc_cma_base      = 0x00000000
[    1.107689] vc-cma: vc_cma_size      = 0x00000000 (0 MiB)
[    1.114489] vc-cma: vc_cma_initial   = 0x00000000 (0 MiB)
[    1.121664] vc-mem: phys_addr:0x00000000 mem_base=0x0ec00
000 mem_size:0x10000000(256 MiB)
```



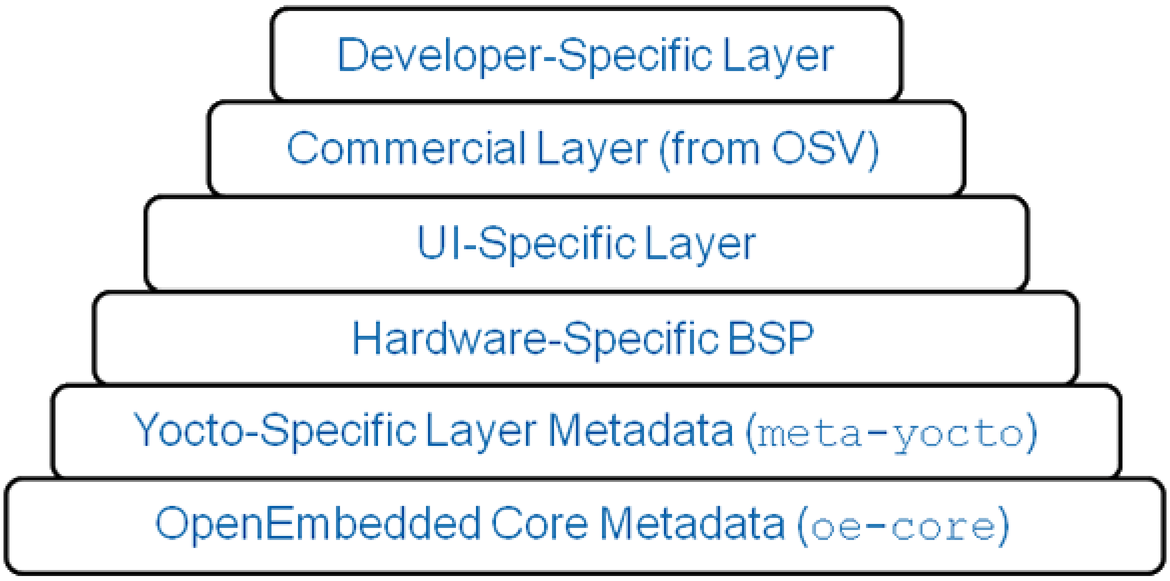
```
bytes read, 116 bits of entropy available)
[    3.288829] udevd[106]: starting eudev-3.2
[    3.330603] random: nonblocking pool is initialized
[    3.729110] bcm2835-wdt 20100000.watchdog: Broadcom BCM28
35 watchdog timer
[    3.780728] gpiomem-bcm2835 20200000.gpiomem: Initialised
: Registers at 0x20200000
[    4.900853] usbcore: registered new interface driver rtl8
192cu
[    6.174399] EXT4-fs (mmcblk0p2): re-mounted. Opts: data=o
rdered
Populating dev cache
Fri Aug 18 11:44:56 UTC 2017
INIT: Entering runlevel: 5
Configuring network interfaces... ifconfig: SIOCGIFFLAGS: No
 such device
Starting syslogd/klogd: done
[    8.303589] uart-pl011 20201000.uart: no DMA platform dat
a

Poky (Yocto Project Reference Distro) 2.2.2 raspberrypi /dev
/tty1

raspberrypi login: _
```

The image contains the basic Linux functionality, such as ls etc.

Customization of the Image

The generalized images are normaly not usable for custom applications. In this case, the system needs to be modified in order to add custom applications, daemons, user interface and/or modules. Thus, we need to take a look at how, BitBake handles building of the modules/components.

Yocto works on layers:


Developer-Specific Layer
Commercial Layer (from OSV)
UI-Specific Layer
Hardware-Specific BSP
Yocto-Specific Layer Metadata (meta-yocto)
OpenEmbedded Core Metadata (oe-core)

This means, that software is added as layers in a layer cake. The image is a physical implementation of a distribution. As seen before, we used a program called BitBake. This is important, so let’s spend a little time on BitBake before proceeding.

BitBake

BitBake uses something called recipes in order to ”bake” the components. These recipes describe how and using what ”ingredients”, the components are made. This should always be run from the build folder. Otherwise, an error message will be given.

Recipes

Recipes are the most common type of meta data in the poky build tree. They describe how to configure and build single packages. These can be extended using the .bbappend files in order to add new files to the recipes. A set of predefined keywords include :

Keyword	Description
SRC_URI	Source Code location
DEPENDS, RDEPENDS	Defines dependencies of other software
EXTRA_OECONFIG, EXTRA_OEMAKE	Option definitions
FILE_*	Defines files to be added to the package.

During the build process, the build engine goes through a number of stages which in then goes through a number of tasks.

Task	Description
do_fetch	Find and download the source code
do_unpack	Unpack the source into a working directory
do_patch	Apply software patches
do_configure	Configure the software before the build
do_compile	Build the software
do_install	Install the build output in the WORKDIR
do_populate_sysroot	Copy build output to a sysroot
do_package_*	Create a binary package for compiled software.

In order to list the tasks for a recipe, the following command can be used :

```
 bitbake -c listtasks recipesname
```

e.g..

```
bitbake -c listtasks core-image-x11
```

and yields (shorted for readability) :

```
core-image-x11-1.0-r0 do_listtasks: do_build                    Default task for a recipe - depends on all
other normal tasks required to 'build' a recipe

core-image-x11-1.0-r0 do_listtasks: do_bundle_initramfs         Combines an initial ramdisk image and kernel
together to form a single image

…...

core-image-x11-1.0-r0 do_listtasks: do_sdk_depends

core-image-x11-1.0-r0 do_listtasks: do_unpack                   Unpacks the source code into a working
directory
```

Also, in order to figure out what packages are available for building a distribution, the following command will display a list of packages and versions.

```
bitbake -s
```

The recipe files have the extension .bb. Since, the BitBake is a layered build model, recipes can be extended using the append files (.bbappend). It is always best to extend existing recipes and layers instead of making copies of the layers and/or recipes.

Now, we will take a look at building an image top-down.

Images

The image is describing how to build a system. Instead of using information regarding the actual machine architecture, it handles information regarding the root file system including which packages to include.

There are 2 ways of generate images :

Either using the oe-init-build-env with a folder specified for the destination and then using the local.conf extending the packages being installed in the root file system.

Or, as we recommend : creating a recipe for building the image. The reason for this recommendation is, that when developing software, one mostly needs 2 variants (development and release versions) and maybe even a third variant for diagnostics. The difference between the variants normally revolves around extra packages. Having the distro definitions outside the build directories, lets the user have several parallel build directories and configurations for a variety of hardware platforms in parallel without having to have multiple copies of the configuration files. In the case one wanted to change the hardware platform, only the build directory should be updated with the new machine. The distributions still work, since they are architecture ignorant.

Image recipes can inherit from other recipes in order to add functionality, thereby being an extension of another image. E.g. a development/troubleshooting image can start out by being the release version and then add software packages needed for troubleshooting. This would be done by creating the release version, and then creating a development image inheriting from this image but also adding the software packages.

In order to get a list of the currently present image recipes, the following command can be used :

```
find meta*/recipes*/images/ -name *.bb
```

Taking a look at one of the built in images (core-image-minimal) reveals :

```
SUMMARY = "A small image just capable of allowing a device to boot."

IMAGE_INSTALL = "packagegroup-core-boot ${ROOTFS_PKGMANAGE_BOOTSTRAP} $
{CORE_IMAGE_EXTRA_INSTALL}"

IMAGE_LINGUAS = " "

LICENSE = "MIT"

inherit core-image

IMAGE_ROOTFS_SIZE ?= "8192"

IMAGE_ROOTFS_EXTRA_SPACE_append = "${@bb.utils.contains("DISTRO_FEATURES", "systemd", "
+ 4096", "" ,d)}"
```

There are a number of keywords with which the image can be customized :

Keyword	Description
DESCRIPTION	Description of the purpose or of the image in general.
IMAGE_BASENAME	Name of the output file
IMAGE_INSTALL	List of packages to install in the image.
IMAGE_ROOTFS_SIZE	Final root file system size
IMAGE_FEATURES	List of features to activate
IMAGE_FSTYPES	Image formats
IMAGE_LINGUAS	Supported languages/locales for the image
IMAGE_PKGTYPE	Type of packages to be built by the system (deb, rpm,

Keyword	Description
	ipk or tar).
IMAGE_POSTPROCESS_COMMAND	Shell command to execute after image build.
LICENSE	Which license to use.

An example of such a distro recipe can be seen below :

```
SUMMARY = "A console-only image that fully supports the target device hardware."

IMAGE_FEATURES += "splash"

LICENSE = "MIT"

inherit core-image
```

This image is the bare minimum, an image can be, and inherits from the most basic image. Besides the most basic functionality, a splash screen is added. Running this basic image will result in this splash screen :

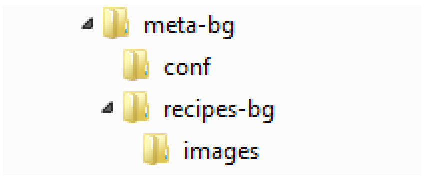
However, getting to build the image, requires that a distribution has been created.

Distributions

A distribution is a combination of the Operating System and a number of additional pieces of software. The lines between Image and Distribution in this case are a bit blurred. We build an image file based on the distribution. So, we'll start by creating an image definition. First, we need to create a folder structure and configuration files:

```
md -p meta-bg/{conf,recipes-bg}/

md -p meta-bg/recipes-bg/images
```

This will create the following structure:



```
meta-bg
    conf
    recipes-bg
        images
```

In the conf folder, a file called layer.conf needs to be created. We base this on existing content from the Raspberry Pi layer:

```
BBPATH .= ":${LAYERDIR}"

BBFILES += "${LAYERDIR}/recipes*/*/*.bb \

            ${LAYERDIR}/recipes*/*/*.bbappend"

BBFILE_COLLECTIONS += "raspberrypi"

BBFILE_PATTERN_raspberrypi := "^${LAYERDIR}/"

BBFILE_PRIORITY_raspberrypi = "9"

LICENSE_PATH += "${LAYERDIR}/files/custom-licenses"
```

And in the images folder, we add a file called rpi-bg-basic-image.bb:

```
# Base this image on rpi-hwup-image
include recipes-core/images/core-image-minimal.bb

# Include modules in rootfs
IMAGE_INSTALL += " \
      kernel-modules \
      "

SPLASH = "psplash-raspberrypi"

IMAGE_FEATURES += "splash"
```

This image can be built and verified as previously done.

In this example, the interesting parts are the ”include”, ”IMAGE_INSTALL” and ”IMAGE_FEATURES” lines.

”include” allows the image to inherit from other images. E.g. a basic boot image can be inherited into a more advanced image. The ”IMAGE_INSTALL” variable allows a number of modules to be installed into the file system, whereas the ”IMAGE_FEATURES” allows a set of modules to be clustered and added as features (groups) instead of having to add a long list of modules.

Using these variables, ”IMAGE_INSTALL” and ”IMAGE_FEATURES”, we can enable and omit software packages from our custom images in order to tailor the image to our exact needs.

If you don’t know, which packages are available, you can run

```
bitbake -s
```

This will yield a list like (limited for readability):

```
Loading cache...done.
Loaded 1540 entries from dependency cache.
Recipe Name                                       Latest Version         Preferred Version
===========                                       ==============         =================

acl                                                   :2.2.52-r0
acl-native                                            :2.2.52-r0
acpid                                                 :2.0.27-r0
adwaita-icon-theme                                      :3.20-r0
alsa-lib                                               :1.1.2-r0
alsa-lib-native                                        :1.1.2-r0
alsa-plugins                                           :1.1.1-r0
alsa-state                                             :0.2.0-r5
alsa-tools                                             :1.1.0-r0
…
…
yasm                                                   :1.3.0-r0
```

```
yasm-native                                         :1.3.0-r0
zip                                                   :3.0-r2
zip-native                                            :3.0-r2
zisofs-tools-native                                 :1.0.8-r0
zlib                                                :1.2.8-r0
zlib-native                                         :1.2.8-r0

Summary: There was 1 WARNING message shown.
```

And now on to the more interesting stuff:

Adding Custom Applications image

The custom image isn't particulary useful, unless it can run some custom code. This code is application specific and can e.g. be a part of a control application or other service. In this example, we'll use an old tried and tested example for beginning programming, the HelloWorld application. In this case written in C++:

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello World!\n");

    return 0;
}
```

When compiled and run, this will provide the following output:

```
Hello World!
```

Not a very useful application but sufficient enough for our purpose.

In order to make the example of creating a an application for adding to the image, we'll create a whole new layer. This layer will be the basis of the experiments for the rest of this document. This is done using the yocto-layer application in the poky folder:

```
yocto-layer create beechgrove
```

Running this tool and replying the questions, it raises, will provide the following framework:

```
meta-beechgrove
 ⊦ conf
 |  ʟ layer.conf
 ⊦ COPYING.MIT
 ʟ README
```

The relevant file so far is the layer.conf file:

```
# We have a conf and classes directory, add to BBPATH
BBPATH .= ":${LAYERDIR}"

# We have recipes-* directories, add to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
      ${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "beechgrove"
BBFILE_PATTERN_beechgrove = "^${LAYERDIR}/"
BBFILE_PRIORITY_beechgrove = "6"
```

The first 2 statements add the current folder and the current recipes to the internal list of folders and files.

BBFILE_COLLECTIONS names the layer.

BBFILE_PATTERN_beechgrove is a regular expression for filtering the BBFILES. It needs to be postfixed with the name of the collection/layer.

BBFILE_PRIORITY_beechgrove is the priority of the layer, if the layer contains recipes also defined in other layers.

Next, we create the image definition. We could use an existing and just add the new package to it, but we'll create a custom image during this introduction, and thus create a whole new image for it.

```
cd poky/meta-beechgrove
md -p recipes-core/images
```

And add a file with the image name (in this case rpi-bg-basic-image) containing:

```
include recipes-core/images/core-image-minimal.bb

# Include modules in rootfs
IMAGE_INSTALL += " \
	kernel-modules \
	"
```

This image is based on the contents of the rpi-basic-image and can be built using

```
cd ../build
bitbake rpi-bg-basic-image
```

Having verified, that the image can build, we now set out to add our helloworld application to our image.

Now, we return to the layer folder and add a recipe for the example package (called dimling version 0.1):

```
cd ~/Development/poky
md -p meta-beechgrove/recipes-examples/examples/dimling-0.1
```

In the examples folder, we add the recipe (dimling_0.1.bb, name_version.bb):

```
SUMMARY = "Simple helloworld application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://$
{COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://helloworld.c"
INSANE_SKIP_${PN} = "ldflags"

S = "${WORKDIR}"

do_compile() {
  ${CC} helloworld.c -o helloworld
}

do_install() {
  -d ${D}${bindir}
  -m 0755 helloworld ${D}${bindir}
}
```

Let's look at the file:

SUMMARY	Description of the package.

`SECTION`	Name of the folder
`LICENSE`	Which license to use
`LIC_FILES_CHKSUM`	Checksum of the license file
`SRC_URI`	List of files, location of the source repository or tar archive.
`INSANE_SKIP_${PN}`	Skip linking flags in order to be able to build.
`do_compile()`	Compile commandline.
`do_install()`	List of commands to run after having compiled[3].

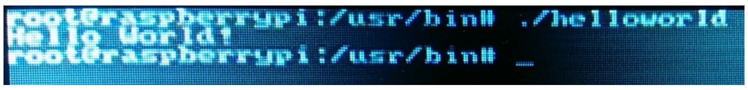
Having placed the helloworld.c file in the dimling-0.1 folder, we're ready to test the building of the package:

```
bitbake dimling
```

The package can build and thus be added to the image by going back to the rpi-bg-basic-image file and add dimling to the image by adding

```
IMAGE_INSTALL_append = " dimling"
```

Now, build the image, transfer it, boot the Raspberry Pi and test run the application.



```
root@raspberrypi:/usr/bin# ./helloworld
Hello World!
root@raspberrypi:/usr/bin# _
```

So now, we have a basic image containing a custom application. This is a good place to start.

[3] Please refer to http://www.yoctoproject.org/docs/current/dev-manual/dev-manual.html for the details.