

# Building Embedded Operating System with IMGUI Demo for *Raspberry $\pi$ - 4 - model B* with *Yocto*

Kaloyan Krastev\*

Tuesday 31<sup>st</sup> October, 2023 03:30

subversion revision 2893

---

\*Triple Helix Consulting

# table of contents

<b>1</b>	<b>introduction</b>	<b>3</b>
<b>2</b>	<b>metadata</b>	<b>4</b>
<b>3</b>	<b>configuration</b>	<b>8</b>
3.1	MACHINE . . . . .	8
3.2	PACKAGE_INSTALL . . . . .	8
3.3	IMAGE_FSTYPES . . . . .	9
<b>4</b>	<b>build</b>	<b>10</b>
<b>5</b>	<b>install</b>	<b>12</b>
<b>6</b>	<b>run</b>	<b>13</b>
<b>7</b>	<b>outlook</b>	<b>15</b>

# 1 introduction

These instructions[5] follow the configuration and build of a Linux-based operating system for *Raspberry  $\pi$  - 4 - model B*[7] with *Yocto*[2]. Find project overview in [6].

The *operating system* (OS) build is done in several steps organized in corresponding sections as follows. Read in Section 2 how to fetch *metadata*. Section 3 shows how to configure the OS build. In Section 4 learn how to build the OS *image* and see how to copy *image* to *SD* card in Section 5. Section 6 is dedicated to post-install issues like the configuration of the WiFi interface from the command line.

## 2 metadata

*Metadata* is a set of instructions to build targets. It is organized in *recipe* files with the *.bb* suffix. Further there are *class* files with the suffix *.bbclass* with information shared between *recipes*. Finally, there are configuration files with the extension *.conf*. These define configuration variables to control the build process. *Metadata* is organized in *layers*. Layers logically separate information of a project. *OpenEmbedded*[1] defines the following layer types.

- base layers contain base *metadata* for the build
- machine aka *board support package* (BSP) layers include *hardware* (HW) support
- distribution layers hold the policy configuration
- *software* (SW) layers are used for additional SW
- miscellaneous layers do not fall in upper categories

The complete list of *github* SW *metadata* repositories used in this project includes *Yocto* layers, the *Raspberry  $\pi$  - 4 - model B* BSP layer, a SW layer with custom recipes, and the build configuration itself. Please refer [6] for details.

In short, users fetch *metadata* in contrast to the *real data* fetched by *bitbake* during OS build. See Section 4 for details. It is an user decision where to put fetched *metadata*. However, it is nice to have all layer sub-directories in one location. In these instructions this location is referred as <layer\_directory>. The second directory to create is the <build\_directory>. This is where the build and build configuration live. I suggest that this one is not inside the <layer\_directory> to not mix *data* and *metadata*.

It is very likely that you will need to install *Yocto* requirements[3] to be able to run *bitbake*. Make sure to have the following command-line tools on your host machine: *chrpath*, *diffstat*, *lz4c*, *rpcgen*; Also have a look at your storage device. Fetched *metadata* requires 400 *MB* of free space. The build needs at least 100 *GB*.

I have a shell script to fetch *metadata* from public *github* repositories. This modification may serve people to build their own OS for *Raspberry  $\pi$  - 4 - model B*. The script performs *metadata* fetch, the *bitbake* initialisation and a simple check of installed layers.

```
#!/bin/bash
# metafetch.sh
# fetch rpi metadata
# release 3.3.2

FETCHER=https://github.com/
GITFETCHER=git@github.com:
BRANCH=kirkstone
DEFLAYER=$HOME/yocto_${BRANCH}/metadata
DEFBUILD=$HOME/yocto_${BRANCH}/rpi4

erreur() { echo $* && exit 0 || kill $$; }

usage() {      # print options and quit

    printf "
usage:
\t $0 <options>
    option          \t purpose          \t default
    -h              \t print this      \t usage
    -d              \t dry run         \t wet run
    -g              \t switch to git protocol \t https protocol
    -r <branch>     \t branch          \t $BRANCH
    -l <layerdir>   \t metadata directory \t $DEFLAYER
    -b <buildir>   \t build directory  \t $DEFBUILD
"
    erreur
}

confirm() {      # get confirmation or quit

    read -p "please confirm (y/n) " choix
    [ "$choix" == "y" ] && echo 1 || echo 0
}

while getopts ":l:b:r:hgd" option; do      # parcer command-line options

    case $option in

        l ) LAYER=$OPTARG;;
        b ) BUILD=$OPTARG;;
        r ) BRANCH=$OPTARG;;
```

```

        g ) FETCHER=$GITFETCHER;;
        d ) DEBUG=yes;;
        h ) usage $0;;
        * ) usage $0;;
    esac
done

# check system path
[ -n "$LAYER" ] || LAYER=$DEFLAYER
[ -n "$BUILD" ] || BUILD=$DEFBUILD
[ -d $LAYER ] || mkdir -p $LAYER || erreur $? cannot create $LAYER
[ -d $BUILD ] || mkdir -p $BUILD || erreur $? cannot create $BUILD
LAYER=$(realpath $LAYER) && printf "\nmetadata:\t $LAYER\n" || erreur $? cannot
find $LAYER
BUILD=$(realpath $BUILD) && printf "build:\t\t $BUILD\n" || erreur $? cannot find
$BUILD
printf "branch:\t\t $BRANCH\nprotocol:\t $FETCHER\n\n"

declare -A REPO
REPO=( # associative array of git repositories
[ yoctoproject/poky.git ]=$LAYER/poky
[ openembedded/meta-openembedded.git ]=$LAYER/oe
[ agherzan/meta-raspberrypi ]=$LAYER/rpi/meta-raspberrypi
[ kaloyanski/meta-thc.git ]=$LAYER/thc/meta-thc
[ TripleHelixConsulting/rpicnf.git ]=$BUILD/conf
)

[ $(confirm) = 0 ] && erreur $? $0 interrupt || echo $0 continue

for repo in ${!REPO[@]}; do # clone repositories
    [ -n "$DEBUG" ] ||
        git clone -b $BRANCH $FETCHER$repo ${REPO[$repo]} &&
        echo git clone -b $BRANCH $FETCHER$repo ${REPO[$repo]}
done

[ -n "$DEBUG" ] && exit 0

# adjust bitbake layer configuration
sed -i s#/home/yocto/layer#$LAYER/g $BUILD/conf/bblayers.conf || erreur sed $?

# bitbake environment
OEINIT=oe-init-build-env
cd $LAYER/poky && pwd || erreur $? cannot find $LAYER/poky
[ -x $OEINIT ] && ./$OEINIT $BUILD || erreur $? cannot find $OEINIT

bitbake-layers show-layers

printf "\n\t how to start a new build\n\n"

echo cd $LAYER/poky
echo ./$OEINIT $BUILD
echo bitbake core-image-x11
echo

```

Download *metafetch.sh* [here](#). It is designed in a way that after a succesful run you may start a build with *bitbake*. The script takes

<layer\_directory> and <build\_directory> from the command-line. You may use next commands to run *metafetch.sh*. The first one is a minimal example. You may specify directories like the first example. Otherwise the script will use default values. The default *Yocto branch* is *kirkstone*. You may want to specify another *branch* with the second command.

```
./metafetch -l <layer_directory> -b <build_directory>  
./metafetch -l <layer_directory> -b <build_directory> -r <branch_name>
```

## 3 configuration

Build configuration is in `<build_directory>/conf`, check files *local.conf* and *bblayers.conf*. *Yocto* layers are specified in *bblayers.conf*. The build directives are in *local.conf*. Variables in this file control the build. Sometimes I call these *directives* to avoid repetitions. Many directives are not covered in these instructions. Please refer *bitbake*[8] documentation for details. It is not always easy to understand the meaning and the relations between different directives. What is more, *bitbake* syntax is pretty complicated. In short, your life can easily become unbearable if the build configuration is too long. Here is a short list.

### 3.1 MACHINE

No doubt, this is the most important directive, set here to *raspberrypi4-64*. You may want to change this value if you build an OS for a different HW. If you want to examine OS built for *Raspberry  $\pi$  - 4 - model B* on your host machine with *qemu*, set *MACHINE* to *qemuarm64*. I confirm that this works although I did not find this approach very useful to test a *graphical user interface* (GUI).

### 3.2 PACKAGE\_INSTALL

This is where to specify additional SW packages. This is useful for packages not included in the *image* by default. In my experience, the default OS has all necessary programs or compact alternatives. However this is the directive used to append *imgui*.



### 3.3 IMAGE\_FSTYPES

This is another important directive. Here I have removed archived *images* that I do not need to decrease the built time and added the *wic* format to have an *image* file ready to be copied to the *SD* card immediately after the build. See [Section 5](#) for details.

## 4 build

*Yocto* provides a list of image types. As I want to have a compact OS and I need a *X* server to run a GUI, I rely on *core-image-x11*<sup>[2]</sup>. This is a very basic *X11* image.

The primary build tool of *OpenEmbedded* based projects, such as the *Yocto* project, *bitbake*, works in the `<build_directory>`. Here is a list of the most important sub-directory names by default. These are configurable but usually there is no need to change their default names.

- `<build_directory>/conf` - build (*local.conf*) and layer (*bblayers.conf*) configuration files
- `<build_directory>/downloads` - fetched source code archives
- `<build_directory>/tmp/work` - working directory where source code is extracted, configured, compiled and installed
- `<build_directory>/tmp/deploy/ipk` - final SW packages in *ipk* format
- `<build_directory>/tmp/deploy/images/raspberrypi4-64` - boot files, compiled kernels and OS *images*.

First, you need to initialize build environment.

```
source <layer_directory>/poky/oe-init-build-env <build_directory>
```

This will change your system path to `<build_directory>`. You may run now next command to check the project layers.

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

If this is fine, the following command is going to build the OS *image*.

task	description
do_fetch	fetch the source code
do_unpack	unpack the source code
do_patch	apply patches to the source
do_configure	source configuration
do_compile	compile the source code
do_install	copy files to the holding area
do_package	analyse holding area
do_package_write_ipk	create <i>ipk</i> package
do_package_qa	quality checks on the package

Table 1: A list of *bitbake* tasks

```
bitbake core-image-x11
```

Be patient because, unless your host machine is a supercomputer, this will take hours. Find a list of tasks performed by *bitbake* for a typical SW package in Table 1.

## 5 install

The OS includes a kernel *ARM*, 64 *bit* boot executable *image* of 23MB, a *Raspberry  $\pi$  - 4 - model B* configuration of Linux 5.15. The total size of kernel modules is 21MB. Happily this kernel release has a *long – term support* (LTS).

*Yocto* provides multiple package and *image* formats. Find *image* files in

< *build\_directory* > /tmp/deploy/images/raspberrypi4 – 64

Further, different ways exist to install *images* on *SD* card. The final result is an OS with two partitions - /*root* and /*boot*. There are not *swap* and *home* partitions. I recommend the classic command-line tool *dd* to copy data. It works fine with different *image* formats like *rpi – sdimg*, *hddimg* and *wic*. The last one is recommended. Find the *SD*card device name, in example /*dev/* < *xxx* >, unmount it with *umount* if mounted, and do copy data from the command line with

```
dd if=core-image-x11-raspberrypi4-64.wic of=/dev/<xxx> status=progress
```

- note 1: run this command in <build\_directory>/tmp/deploy/images/raspberrypi4 – 64
- note 2: run this command with *root* privileges
- note 3: be careful to not specify the device name of your hard drive (see note 2)

The transfer is going to take a while. Once it is over, put the card in you *Raspberry  $\pi$  - 4 - model B* and turn it on. That's it.

## 6 run

Wireless connection is established via classic command-line tools like *ip* and *iw*. I use a *dynamic host configuration protocol* (DHCP) client, *udhcpc*, and *wpa\_supplicant* to store WiFi connection. The shell scripts *wifini.sh* is designed by me and installed in */usr/bin*, as well as a running GUI example to demonstrate the usage of the *Dear ImGui* library.

```
#!/bin/sh
# wifini.sh
# wifi connection requirements:

# wpa_passphrase, wpa_supplicant, ip, iw, grep, awk
# designed by kaloyansen at gmail dot com
# copyleft triplehelix-consulting.com
#####

# files
WPACONF=/etc/wpa_supplicant.conf
WPASOCKET=/run/wpa_supplicant/$WIFACE
UDHCPID=/run/udhcpc.$WIFACE.pid
IFCONF=/etc/network/interfaces

# command-line tools
WPAPASS=/usr/bin/wpa_passphrase
IW=/usr/sbin/iw
WPASUPP=/usr/sbin/wpa_supplicant
DHCP=/sbin/udhcpc
IP=/sbin/ip

erreur() { echo $* && exit 1; }

# get wifi interface and network id
WIFACE=$(IW dev|grep Interface|awk '{print $2}')
SSID=$(getopt s: $* | awk '{print $2}')

[ -n "$SSID" ] &&
    echo $0: $WIFACE $SSID ||
        erreur interface $WIFACE specify network: $0 -s '<SSID>'

[ "$USER" = "root" ] || erreur run $0 as root

# enable interface connexion on boot
if [ -f $IFCONF ]; then
    grep "auto $WIFACE" $IFCONF > /dev/null ||
        printf "auto $WIFACE
        wpa-roam /etc/wpa_supplicant.conf\n" >> $IFCONF
else
    erreur $0: $IFCONF not found;
fi

#grep "auto $WIFACE" $IFCONF > /dev/null ||
```

```

#   printf "auto $WIFACE\n" >> $IFCONF

# verify connexion
$IW dev|grep $SSID > /dev/null &&
    erreur $0 info: $WIFACE $SSID ||
        echo $0 connecting to $SSID

# up interface
$IP link show $WIFACE | grep UP ||
    $IP link set $WIFACE up

# search network
$IW $WIFACE scan|grep $SSID ||
    erreur $0 warning: cannot find network $SSID;

# save network
grep $SSID $WPACONF ||
    $WPAPASS $SSID >> $WPACONF

# load network
[ -S "$WPASOCKET" ] ||
    $WPASUPP -B -D wext -i $WIFACE -c $WPACONF

# start a dhcp client
$DHCP -i $WIFACE ||
    erreur $0 warning: $?

# [ -f "$UDHCPID" ] ||
# ip addr show $WIFACE
# iw $WIFACE link
# ip route show

```

The script is available for download [here](#). Specify network *id* from the command line with a short command-line option *s*. See next example usage.

```
wifini -s <SSID>
```

Once an *internet protocol* (IP) address is assigned to *Raspberry  $\pi$  - 4 - model B*, the *secure shell* (SSH) server by *Dropbear*[4] allows for secure remote login, control and file transfer.

## 7 outlook

This reports the progress in the development of a custom Linux-based OS for *Raspberry  $\pi$  - 4 - model B*[\[7\]](#). The kernel version of this embedded OS is Linux release 5.15. An example GUI application using the *Dear ImGui* library is built as a part of the OS *image*. In addition, an SSH server provides remote connection, data transfer and device control. As the OS is now functional, performance and real-time tests are ongoing.

## acronyms

BSP	<i>board support package</i> . . . . .	4
SSH	<i>secure shell</i> . . . . .	14
GUI	<i>graphical user interface</i> . . . . .	8
SW	<i>software</i> . . . . .	4
HW	<i>hardware</i> . . . . .	4
OS	<i>operating system</i> . . . . .	3
DHCP	<i>dynamic host configuration protocol</i> . . . . .	13
IP	<i>internet protocol</i> . . . . .	14
LTS	<i>long – term support</i> . . . . .	12



# bibliography

- [1] community. *OpenEmbedded*. 2017.  
URL: <https://www.openembedded.org> (visited on 2023).
- [2] Yocto Project community. *Yocto Project*. 2023.  
URL: <https://www.yoctoproject.org> (visited on 2023).
- [3] Yocto Project community. *Yocto Project Quick Build*. 2023. URL:  
<https://docs.yoctoproject.org/brief-yoctoprojectqs/index.htm>  
(visited on 2023).
- [4] Matt Johnston. *Dropbear SSH*. 2023.  
URL: <https://matt.ucc.asn.au/dropbear/dropbear.html>  
(visited on 2023).
- [5] Kaloyan Krastev. *Building Embedded Operating System with IMGUI Demo for Raspberry  $\pi$  - 4 - model B with Yocto*. 2023. URL:  
<https://kaloyanski.github.io/meta-thc/thchowto.html>  
(visited on 2023).
- [6] Kaloyan Krastev. *Embedded Operating System for Raspberry  $\pi$  - 4 - model B with Yocto*. 2023. URL:  
<https://kaloyanski.github.io/meta-thc/thcport.html>  
(visited on 2023).
- [7] Raspberry  $\pi$  Ltd. *Raspberi  $\pi$* . 2023.  
URL: <https://www.raspberrypi.com> (visited on 2023).
- [8] Richard Purdie, Chris Larson, and Phil Blundell.  
*BitBake User Manual*. 2023. URL:  
<https://docs.yoctoproject.org/bitbake> (visited on 2023).