Building Embedded Operating System with IMGUI Demo for *Raspberry* π - 4 - *model B* with *Yocto*

Kaloyan Krastev*

Monday 6th November, 2023 revision 2893

*Triple Helix Consulting

table of contents

1 introduction

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

The 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.

type	extension	purpose
recipe	*bb*	$software$ (SW) build instructions
recipe	*bbappend*	SW recipe modification
class	*bbclass*	shared instructions
configuration	$conf$	global build definitions

Table 1: A list of *bitbake* file types

2 metadata

$Metadata$ is a set of instructions to build targets. It is organized in *recipe* files with the *bb* extension. There are files with *bbappend* extension to modify *recipes* and *class* files with a suffix *bbclass* for instructions shared between *recipes*. The configuration files have the extension $conf$. These define configuration variables to control the build process. Tgere is a list of *bitbake* file types in Table 1.

$Metadata$ 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
- 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* π - 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*.

2.1 requirements

It is very likely that you will need to install *Yocto* requirements[3] to be able to run *bitbake*. There you find a list of packages to install. *Yocto* sanity checked distributions are *poky*-3.3, *poky*-3.4, *Ubuntu*-18.04, *Ubuntu*-20.04, *Ubuntu*-22.04, *Fedora*-37, *Debian* − 11, *OpenSUSEleap*-15.3, *AlmaLinux*-8.8. I use *bitbake* on a rolling release *Manjaro* Linux. It should not be complicated to satisfy *Yocto* on machines with GNU/Linux operating system. Maybe binaries are not the same on different HW architectures, but the OS is a simplified open-source OS with a Linux kernel with the proper HW configuration.

Install the following packages;

- *git*
- *tar*
- *python*
- *gcc*

- *GNU make*

Find more details in *Yocto* documentation at [3]. You may need to install in addition *diffstat*, *unzip*, *texinfo*, *chrpath*, *wget*, *xterm*, *sdl*, *rpcsvc – proto*, *socat*, *cpio*, *lz*4 and *inetutils* packages. As a double check, make sure to have the following command-line tools on your host machine: *chrpath*, *diffstat*, *lz*4*c*, *rpcgen*. Then have a look at your storage device. Fetched *metadata* requires 412 *MB* of free space. The build may need up to 30 *GB* or 50 *GB* if intermediate files are kept. Read for the *bitbake* class *rm_work* in Section 3.

2.2 automation

I have a shell script to fetch *metadata* from public *github* repositories. This modification may serve people to build their own OS for *Raspberry* π - 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 confirm ||
            erreur $1 interrupted
}

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

    case $option in

        l ) LAYER=$OPTARG;;
        b ) BUILD=$OPTARG;;
        r ) BRANCH=$OPTARG;;
        g ) FETCHER=$GITFETCHER;;
        d ) DRYRUN=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/rpiconf.git]=$BUILD/conf
)

[ -n "$DRYRUN" ] || confirm $0 confirmation
for repo in ${!REPO[@]}; do     # clone repositories

    command="git clone -b $BRANCH $FETCHER$repo ${REPO[$repo]}"
    [ -n "$DRYRUN" ] || $command && echo $command
#         git clone -b $BRANCH $FETCHER$repo ${REPO[$repo]} &&
#             echo git clone -b $BRANCH $FETCHER$repo ${REPO[$repo]}
done
[ -n "$DRYRUN" ] && erreur $0 dry run exit

# adjust bibtbake 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
[ -f $OEINIT ] && . ./$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 *raspberrypi*4-64. 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* π - 4 - *model B* on your host machine with *qemu*, set *MACHINE* to *qemuarm*64. 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* to decrease the built time and added the *wic* format. It is possible to list the partitions on a *wic image* with the *wic* command-line tool. In addition we can copy it to *SD* cards. See Section 5 for details.

3.4 PACKAGE_CLASSES

There are different package formats used in various Linux-based OS's to distribute and manage SW packages. Both *Debian* package format - *deb* and *rpm* from *RedHat* do well, but recently I had issues with *ipk* so I disabled this package format.

3.5 rm_work

This *bitbake* class is in <layer_directory>/poky/meta/classes/rm_work.bbcla
It defines a specific task for each SW package to remove intermediate files generated during the build. This decreases storage space about two times. Those who want to keep the working data and have enough disk space on their storage device may want to comment next line in *local.conf*.

```
INHERIT:append = " rm_work"
```

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 *ipk* package
do_package_qa	quality checks on the package

Table 2: A list of *bitbake* tasks

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 $core - image - x11$[2]. This is a very basic $X11$ image.

There is a script to initialise build environment in <layer_directory>/poky.

```
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
```

The primary build tool of *OpenEmbedded* based projects, such as the *Yocto* project. The next command is going to build the OS *image*.

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

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 2.

name	location	desc
configuration	<build_directory>/conf	buil
download	<build_directory>/downloads	fetc
work	<build_directory>/tmp/work	wor
package	<build_directory>/tmp/deploy/rpm	fina
image	<build_directory>/tmp/deploy/images/raspberrypi4-64	boo

Table 3: *bitbake* workflow

4.1 flow

The build happens in the <build_directory>. Here is a list of the most important sub-directory names by default. These are configurable.

5 install

The OS includes a kernel ARM, 64 bit boot executable $image$ of $23MB$, a $Raspberry$ π - 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 SDcard 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/ras
  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 $Raspberry$ π - 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 ssid
WIFACE=`$IW dev|grep Interface|awk '{print $2}'`
SSID=$(getopt s: $* | awk '{print $2}')

sorry() {
    if [ "$1" = "" -o ! -e "$1" ]; then
        echo "no $2 supplied" 1>&2
        exit 1
    fi
}


sorry $SSID network

[ -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\n" >> $IFCONF
#         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* π - 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* π - 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	*board support package*	4
SSH	*secure shell*	15
GUI	*graphical user interface*	9
SW	*software*	4
HW	*hardware*	4
OS	*operating system*	3
DHCP	*dynamic host configuration protocol*	14
IP	*internet protocol*	15
LTS	*long − term support*	13

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 π - 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 π - 4 - model B with* Yocto. 2023. URL:
https://kaloyanski.github.io/meta-thc/thcport.html
(visited on 2023).

[7] Raspberry π Ltd. *Raspberi π*. 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).