Building Embedded Operating System with IMGUI Demo for *Raspberry* π with *Yocto*

Kaloyan Krastev*

Friday 17th November, 2023 †

*Triple Helix Consulting

†revision 2893

table of contents

1 introduction

These instructions[5] follow the configuration and build of a Linux-based *operating system* (OS) for *Raspberry* π[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 *metadata* file types

layer type	contents
base	base *metadata* for the build
machine aka *board support package* (BSP)	*hardware* (HW) support
distribution	policy configuration
SW	additional SW
miscellaneous	do not fall in upper categories

Table 2: *metadata* layer types defined by *OpenEmbedded*[1]

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. See a list of *metadata* file types in Table 1.

Metadata is organized in *layers*. Layers logically separate information of a project. Table 2 presents *OpenEmbedded*[1] *metadata* layer types.

The complete list of *github* SW *metadata* repositories used in this project includes *Yocto* layers, the *Raspberry* π 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 later during the OS build. See Section 4 for details. It means that users decide where to store fetched *metadata*. It is nice to have all layer sub-directories in one system location. In these instructions it is referred as <META-DIR>. The second directory to create is the <BUILD-DIR>. This is where the build and the build configuration live. I suggest that <BUILD-DIR> one is not inside <META-DIR> to not mix *data* and *metadata*.

2.1 automation

I have a shell script to fetch all *metadata* from public *github* repositories. After some major modifications hopefully it may serve other people to build their own OS for *Raspberry* π. The script performs *metadata* fetch, the *bitbake* initialisation and a simple layer verification.

```
#!/bin/env bash
# name: metafetch
# purpose: clone rpi metadata

FETCHER=https://github.com/
GITFETCHER=git@github.com:
BRANCH=kirkstone
DEFMETADIR=$HOME/yocto_$BRANCH/metadata
DEFBUILDIR=$HOME/yocto_$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
    -m <metadir>    \t metadata directory       \t $DEFMETADIR
    -b <buildir>    \t build directory          \t $DEFBUILDIR
"
    erreur
}

confirm() {    # get confirmation or quit
```

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

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

    case $option in

        m ) METADIR=$OPTARG;;
        b ) BUILDIR=$OPTARG;;
        r ) BRANCH=$OPTARG;;
        g ) FETCHER=$GITFETCHER;;
        d ) DRYRUN=yes;;
        h ) usage $0;;
        * ) usage $0;;
    esac
done

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

declare -A REPO
REPO=(    # associative git repository array
    [yoctoproject/poky.git]=$METADIR/poky
    [openembedded/meta-openembedded.git]=$METADIR/oe
    [agherzan/meta-raspberrypi]=$METADIR/rpi/meta-raspberrypi
    [kaloyanski/meta-thc.git]=$METADIR/thc/meta-thc
    [TripleHelixConsulting/rpiconf.git]=$BUILDIR/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 $command
done
[ -n "$DRYRUN" ] && erreur $0 dry run exit

# adjust bibtbake layer configuration
sed -i s#/home/yocto/layer#$METADIR#g $BUILDIR/conf/bblayers.conf || erreur sed $?

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

bitbake-layers show-layers

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

```
echo cd $METADIR/poky
echo . ./$OEINIT $BUILDIR
echo bitbake core-image-x11
echo
```

Download `metafetch` here. It is designed in a way that after a successful run one may start a build with *bitbake*. The script takes <META-DIR> and <BUILD-DIR> from the command-line. You may use next examples to run `metafetch`. The first one is a minimal example. You may specify directories like the second example. Otherwise the script will use default values. The default fetch protocol is *https* but I recommend using *git* if you can because it is an order of magnitude faster. Use the command-line option $-g$ to switch. The default *git* branch is *kirkstone*. See all command-line options with $-h$.

```
./metafetch
./metafetch -m <META-DIR> -b <BUILD-DIR>
./metafetch -g
```

3 configuration

Build configuration is in <BUILD-DIR>`/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. To not mix them, I have HW specific directives in <BUILD-DIR>`/conf/raspberrypi4-64.conf`.

3.1 directives

Many variables are not covered here. Please refer *bitbake*[8] documentation for details. It is not always easy to understand the meaning and their relations. What is more, *bitbake* syntax is pretty complicated. In short, your life can easily become unbearable if the build configuration is too long. See next a list of important build configuration directives.

3.1.1 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 emulate *Raspberry* π 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).

3.1.2 INIT_MANAGER

The OS *init* process could be `sysvinit`, `systemd` or `mdev-busybox`.

config file	INIT_MANAGER	TCLIBC
`poky.conf`	`sysvinit`	`glibc`
`poky-bleeding.conf`	`sysvinit`	`glibc`
`poky-altcfg.conf`	`systemd`	`glibc`
`poky-tiny.conf`	`mdev-busybox`	`musl`

Table 3: reference distribution configuration

3.1.3 TCLIBC

The *GNU is not UNIX* (GNU) standard C library variant to use during the build. Available options `glibc`, `musl`, `newlib` and `baremetal`.

3.1.4 DISTRO

This is the short name of the OS distribution. *Yocto* provides four variants of their reference distribution called *Poky*. See details in <META-DIR>`/poky/meta-poky/conf/distro/poky*.conf`. Some distribution dependent directive values are presented in Table 3.

3.1.5 MACHINE_FEATURES

This directive controls machine features. It is set in the machine configuration file and specifies the hardware features for a given machine.

3.1.6 DISTRO_FEATURES

Distributions can select which features they want to support through the DISTRO_FEATURES variable, which is set in the distribution configuration file.

3.1.7 IMAGE_FEATURES

This directive controls the contents of the OS *image*. Different predefined packages could be added, removed or modified via this variable. Useful examples for *image* features are $allow-empty-password$, $allow-root-login$, $empty-root-password$, $post-install-logging$, *splash*, $package-management$ and $ssh-server-dropbear$.

3.1.8 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*.

3.1.9 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 it.

3.1.10 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.

3.1.11 IMAGE_OVERHEAD_FACTOR

This defines the free storage space on the `root` partition. Overhead factor of 2 means that the free space will be equal to the space already used by the OS. This will double the size of the image. The default value of 1.3 increases *image* size with 30 %.

3.1.12 INHERIT

This is a list of included *bitbake* classes. See Section 3.2.

3.2 classes

3.2.1 rm_work

This is an example of a *bitbake* class found in <META-DIR>`/poky/meta/classes/rm_work.bbclass`. It defines a specific task for each SW package to remove intermediate files generated during the build. This decreases storage space about twice. Those who want to keep the working data and have enough storage space may comment the next line in `local.conf`.

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

4 build

It is very likely that you will need to install *Yocto* requirements[3] to be able to run *bitbake*. The list of *Yocto* sanity checked distributions currently includes *poky*-3.3, *poky*-3.4, *Ubuntu*-18.04, *Ubuntu*-20.04, *Ubuntu*-22.04, *Fedora*-37, $Debian - 11$, *OpenSUSEleap*-15.3 and *AlmaLinux*-8.8. But I use *bitbake* on *Manjaro* - a not officially supported GNU/Linux distribution. That is why I guess that it should not be complicated to satisfy requirements on a GNU/Linux machine. Of course, binary files are not the same on different HW architectures, but the OS has a Linux kernel and standard open-source programs.

4.1 requirements

Ensure that the following packages are installed.

- *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*, *lz4*, *gawk*, *findutils*, *crypt* and *inetutils* packages. As a double check, make sure to have the following command-line tools on your host machine: *chrpath*, *diffstat*, *lz4c*, *rpcgen*, *bash*, *bzip2*, *file*, *grep*, *patch*, *sed*.

Fetched *metadata* requires only 412 MB of free space but the build may need up to 30 GB or even 50 GB if intermediate files are kept. Read for the *bitbake* class *rm_work* in Section 3.

4.2 environment

To initialise build environment navigate to <META-DIR>`/poky` and source the initialization script like the next command.

```
source oe-init-build-env <BUILD-DIR>
```

Alternatively verify and source a dedicated shell script <META-DIR>`/thc/meta-thc/bin/yoctoinit`. The initialization script will change the system path to <BUILD-DIR>. Now you may want run next command to check project layers.

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

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

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

Unless your host machine is a supercomputer, this will take at least two hours. Find a list of tasks performed by *bitbake* for a typical SW package in Table 4.

4.3 flow

The build happens in <BUILD-DIR>. Table 5 presents a list of important <BUILD-DIR> sub-directories.

Source archives are saved in the *download* directory. They are extracted, configured, compiled and installed in the *work* directory.

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_populate_sysroot	copy files to the staging area
do_package	analyse holding area
do_package_write_rpm	deploy SW package in *rpm* format
do_package_qa	quality checks on the package

Table 4: A short list of *bitbake* tasks

name	location	description
configuration	`conf`	build configuration files
download	`downloads`	fetched SW source code archives
work	`tmp/work`	working directory
package	`tmp/deploy/rpm`	final SW packages in *rpm* format
image	`tmp/deploy/images`	boot files, kernels and *images*

Table 5: *bitbake* workflow

SW packages are created and stored in the *package* directory. Finally, following the build configuration packages are unpacked to create the OS *image* found in the *image* directory.

5 install

The OS includes a kernel *ARM*, 64 bit boot executable *image* of 23 MB, a *Raspberry* π configuration of Linux 5.15. This is a *long − term support* (LTS) kernel release. The total size of kernel modules is 21 MB.

Yocto provides multiple package and *image* formats. Different ways exist to install *images* on *SD* card. The OS has two partitions - `/root` and `/boot`. There are no *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 with the next command.

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

- note 1: run this command in <BUILD-DIR>`/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)

Alternatively, there is a dedicated script - <META-DIR>`/thc/meta-thc/bin/burn`. Use the command-line option `-h` for details. The transfer does not take a lot of time. When it is over, replace the card to *Raspberry* π 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, 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[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*

DHCP *dynamic host configuration protocol*

GNU *GNU is not UNIX*

GUI *graphical user interface*

HW *hardware*

IP *internet protocol*

LTS *long − term support*

OS *operating system*

SSH *secure shell*

SW *software*

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 with Yocto. 2023. URL:
https://kaloyanski.github.io/meta-thc/thchowto.html
(visited on 2023).

[6] Kaloyan Krastev.
Embedded Operating System for Raspberry π 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).