# introduction to code development and debugging using yocto

## goal

The goal of this Lab is to allow you to set up the development environment using Yocto to write and cross-compile applications for the embedded Linux running on the Raspberry Pi 3 and to perform remote debugging operations.

## prerequisites

To follow this Lab, you need:

1. Raspberry Pi 3 board full;
2. Micro USB cable;
3. 8 GB Micro SD card;
4. USB-to-Serial debug module for Raspberry Pi 3 or USB to TTL adapter;
5. A PC provided with Ubuntu Desktop 14.04 LTS, or a virtual machine hosting Ubuntu Desktop 14.04 LTS;
6. A Micro SD card reader attached to the PC/virtual machine;
7. An Ethernet cable connecting the PC/virtual machine with the Raspberry Pi 3;
8. (**Optional**) Micro HDMI cable.

## workplace setup

We assume that:

* The PC/virtual machine is configured with IP address 192.168.1.1;
* The Raspberry Pi 3 is configured with IP address 192.168.1.2;
* The PC/virtual machine and the Raspberry Pi 3 are connected using Ethernet cable.

Please refer to the first lab for instructions on how to obtain the above configuration.

## Preparing the recipes

To debug application running on the Raspberry Pi 3, a new package shall be added to the Yocto configuration so that debug tools are installed in the embedded Linux distribution. For this purpose, edit the file ***raspberryPi3 /build/conf/local.conf*** to have the following content:

MACHINE ??= 'raspberrypi3'

DISTRO ?= 'poky'

PACKAGE\_CLASSES ?= "package\_rpm"

EXTRA\_IMAGE\_FEATURES = "debug-tweaks tools-debug"

USER\_CLASSES ?= "buildstats image-mklibs"

PATCHRESOLVE = "noop"

BB\_DISKMON\_DIRS = "\

STOPTASKS,${TMPDIR},1G,100K \

STOPTASKS,${DL\_DIR},1G,100K \

STOPTASKS,${SSTATE\_DIR},1G,100K \

STOPTASKS,/tmp,100M,100K \

ABORT,${TMPDIR},100M,1K \

ABORT,${DL\_DIR},100M,1K \

ABORT,${SSTATE\_DIR},100M,1K \

ABORT,/tmp,10M,1K"

PACKAGECONFIG\_append\_pn-qemu-native = " sdl"

PACKAGECONFIG\_append\_pn-nativesdk-qemu = " sdl"

CONF\_VERSION = "1"

ENABLE\_UART = "1"

DL\_DIR ?= "${BSPDIR}/downloads/"

The new configuration adds the tools-debug package that provides the target with debug support.

## Building and deploying the new system

Move to the directory ***raspberryPi3*** and prepare the build environment:

cd ~/raspberryPi3

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

bitbake -c clean rpi-basic-image

bitbake rpi-basic-image

After a while, a new Micro SD card image would be available, which you can deploy in the Micro SD as follows (assuming the Micro SD is available to the PC as /dev/sdN). Alternatively, use a program of your preference to flash the image.

First, run the:

sudo fdisk -l

command to determine which device to flash to (plug in and unplug the SD card to determine which device it is). For this example, the SD card is under the name “sdc” (this may be different in your environment). Next, ensure that the device is unmounted. This can be done using the command:

sudo umount /dev/sdc\*

Once this is done, the following command can be used to copy the image across to the SD card (substitute any folder names and device names to ensure they are relevant to your specific environment).

sudo dd bs=1M if=/home/user/raspberryPi3/rpi-build/tmp/deploy/images/raspberrypi3/rpi-basic-image-raspberrypi3.rpi-sdimg of=/dev/sdc

Note that if not done properly, the image being flashed across to the SD card may cause problems when attempting to turn on the board. If this is the case, it may be worth retrying the process again and ensuring that it is done properly, or use a flash program to automate the process.

Also note that, this time, the most recently built image should be in the “tmp” folder, not “tmp-glibc”.

## Building and deploying the new system

You are now ready to compile the cross-compiler environment that allows building programs compatible with the just compiled embedded Linux system.

If you have restarted since setting up the environment, run:

cd ~/raspberryPi3

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

To build the cross-compiler environment, execute the following command:

bitbake rpi-basic-image –c populate\_sdk

This command will create the ***~/raspberrypi3/rpi-build/tmp/deploy/sdk/poky-glibc-x86\_64-rpi-basic-image-cortexa7hf-neon-vfpv4-toolchain-2.1.3.sh*** file containing the binary files as well as libraries and header files needed to build programs for the embedded Linux distribution previously created.

The cross-compiler environment shall be installed on the development host as follows:

cd ~/raspberryPi3/rpi-build/tmp/deploy/sdk/

./poky-glibc-x86\_64-rpi-basic-image-cortexa7hf-neon-vfpv4-toolchain-2.1.3.sh

The command will produce the following output:

Poky (Yocto Project Reference Distro) SDK installer version 2.1.3 =================================================================

Enter target directory for SDK (default: /opt/poky/2.1.3):

You are about to install the SDK to "/opt/poky/2.1.3". Proceed[Y/n]?

Extracting SDK................................................................done

Setting it up…done

SDK has been successfully set up and is ready to be used.

Each time you wish to use the SDK in a new shell session, you need to source the environment setup script e.g.

$ . /opt/poky/2.1.3/environment-setup-cortexa7hf-neon-vfpv4-poky-linux-gnueabi

The cross-compiler environment is installed into ***/opt/poky/2.1.3/***, and a shell script is provided to set up the variables needed to use it.

Before using the cross-compiler environment, the needed variables shall be set up by running the following command:

sudo chmod +x /opt/poky/2.1.3/environment-setup-cortexa7hf-neon-vfpv4-poky-linux-gnueabi

. /opt/poky/2.1.3/environment-setup-cortexa7hf-neon-vfpv4-poky-linux-gnueabi

## writing a simple test application and DEPLOYING it

Create a new directory and populate it with a simple “Hello World” application stored in the file ***hello.c***, having the following content:

#include <stdio.h>

int main( void )

{

printf( "Hello, World!\n" );

return 0;

}

Now, prepare the file ***Makefile*** with the following content (as before, ensure the lines are indented with tabs, not spaces):

C=arm-poky-linux-gnueabi-gcc --sysroot=/opt/poky/2.1.3/sysroots/cortexa9hf-neon-vfpv4-poky-linux-gnueabi/ -mfloat-abi=hard

LD=arm-poky-linux-gnueabi-ld --sysroot=/opt/poky/2.1.3/sysroots/cortexa9hf-neon-vfpv4-poky-linux-gnueabi/

CFLAGS=-O2 -pipe -g -feliminate-unused-debug-types

CXXFLAGS=-O2 -pipe -g -feliminate-unused-debug-types

all: hello

hello.o: hello.c

hello: hello.o

$(CC) $(CFLAGS) -o hello hello.o

clean:

rm hello hello.o

The file instructs how to run the cross-compiler (arm-poky-linux-gnueacbi-gcc) to generate the executable file to be executed in the Raspberry Pi 3.

To build the program, you can run the following command:

make

In case the cross-compilation is executed correctly, the file ***hello*** will be created.

The executable program can be transferred on the Raspberry Pi 3 using the ***scp*** command as follows:

scp hello root@192.168.1.2:hello

If the operation succeeds, you will find the ***hello*** executable in the ***/home/root*** directory on the Raspberry Pi 3.

It might throw a warning: if this is the case, you can add the key host as suggested in the message.

ssh-keygen -f "/home/user/.ssh/known\_hosts" -R 192.168.1.2

Then, run the “scp” command again, and it should copy the executable across.

## Debugging the test application

We are now ready to remotely debug the application using the command-line debugger that comes with the cross-compilation environment.

On the **Raspberry Pi 3 side**, run the following command:

root@raspberrypi3:~# gdbserver localhost:2000 hello

Process hello created; pid = 1151

Listening on port 2000

The server for the gnu debugger (gdbserver) is executed to debug the hello program, awaiting connection on the localhost at port 2000.

On the **development host**, run the following command:

~/raspberrypi/user-app$ arm-poky-linux-gnueabi-gdb hello

GNU gdb (GDB) 7.9.1

Copyright (C) 2015 Free Software Foundation, Inc.

License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software: you are free to change and redistribute it.

There is NO WARRANTY, to the extent permitted by law. Type "show copying"

and "show warranty" for details.

This GDB was configured as "--host=x86\_64-pokysdk-linux --target=arm-poky-linux-gnueabi".

Type "show configuration" for configuration details.

For bug reporting instructions, please see:

<http://www.gnu.org/software/gdb/bugs/>.

Find the GDB manual and other documentation resources online at:

<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".

Type "apropos word" to search for commands related to "word"...

Reading symbols from hello...done.

**(gdb) target remote 192.168.1.2:2000**

Remote debugging using 192.168.1.2:2000

warning: Unable to find dynamic linker breakpoint function.

GDB will be unable to debug shared library initializers

and track explicitly loaded dynamic code.

0x76fcfac0 in ?? ()

(gdb)

This will load the gnu debugger cross-compiled for the embedded Linux running on the Raspberry Pi 3 (arm-poky-linux-gnueabi-gdb).

Once the debugger is running, we need to connect to gdbserver for performing a remote debugging session. For this purpose, the command ***target remote 192.168.1.2:2000*** is used.

You can now debug the program using the gdb commands. For example, type the following commands:

**(gdb) list**

1 #include <stdio.h>

2

3 int main( void )

4 {

5 printf( "Hello, World!\n" );

6

7 return 0;

8 }

9

10

The ***list*** command shows you the source code of the program.

We can then set up a break point with the break command as follows:

**(gdb) break 5**

Breakpoint 1 at 0x102f4: file hello.c, line 5.

We are not ready to run the program till the break point is reached:

**(gdb) continue**

Continuing.

warning: Could not load shared library symbols for 2 libraries, e.g. /lib/libc.so.6.

Use the "info sharedlibrary" command to see the complete listing.

Do you need "set solib-search-path" or "set sysroot"?

Breakpoint 1, main () at hello.c:5

5 printf( "Hello, World!\n" );

To advance the execution step by step, you can use the ***next*** command. When the printf() instruction is executed, you can see the “Hello, World!” message on the Raspberry Pi 3 terminal.

To terminate the debug session, you can use the ***quit*** command.

In case your program makes use of variables, you can inspect them with the print command (please refer to the online help for details on the syntax, using the command ***help print***).