My Beaglebone black setup for embedded and robotics development

Sep 13, 2015 • Salah • Crossposted from Salah's blog.

The Beaglebone black is easily my favourite Embedded linux platform. It's cheap, open-source and offers a great amount of functionnalities. Through this article, I would like to document my default setup for the board so I can start developing on it.

What's even greater about this board is it's surprisingly easy to use thanks to some great engineering and a great community of developers. All you need to use this board is the mini USB cable that comes with it: it powers it and emulates an ethernet connection over USB.

Choosing a Linux distribution, the eternal argument

The choice for a Linux distribution is always subject to discussion. The most popular distros on embedded computer boards are Yocto, Angstrom and Debian. The last two are supported by the Beagleboard organisation[1]. But I like Ubuntu better, that's what I run on my laptop, its setup time is short and you will have less problems with drivers.

Fortunately, an official Ubuntu image is supported and maintained by the Beagleboard organisation. So, start by downloading the image from them, uncompress it and then copy the image into your SD card (in my case under `/dev/mmcblk0`):

```
wget https://rcn-ee.com/rootfs/2015-05-08/flasher/BBB-eMMC-flasher-ubuntu-14.04.2-console-armhf-2015-05-08-2gb.img.xz
unxz BBB-eMMC-flasher-ubuntu-14.04.2-console-armhf-2015-05-08-2gb.img.xz
sudo dd if=./BBB-eMMC-flasher-ubuntu-14.04.2-console-armhf-2015-05-08-2gb.img of=/dev/mmcblk0
```

Now remove it from your computer and plug it in the Beaglebone. Press the S2 button while powering it up so it boots from the SD card and starts flashing the image to the eMMC. After a few seconds, the LEDs should start in a K2000/Cylon way. Once the LEDs are off, it's done, you can unplug/replug it and SSH into it.

```
ssh ubuntu@192.168.7.2
```

If network manager is giving you a hard time, switch the LAN interface to manual by modifying your `/etc/network/interfaces` file, mine looks like this:

```
# interfaces(5) file used by ifup(8) and ifdown(8)
auto lo
iface lo inet loopback

# For Beaglebone black
iface eth2 inet manual
iface eth1 inet manual
iface usb0 inet manual
```

Now, if you did this, before you can SSH into the board you will need to configure your IP over the shared network (let's say `eth2`):

```
sudo ifconfig eth2 192.168.7.1
# then
ssh ubuntu@192.168.7.2
```

Linux, RT-PREEMPT or Xenomai, the roboticist's dilemma

If you are starting to play with embedded linux platforms and just want to try to build some cool little thing, skip this part. If you are trying to build more complex systems that impose constraints on the execution time of your tasks, then read this part.

A good comparison between these three solutions is made by this article titled: How fast is fast enough? Choosing between Xenomai and Linux for real-time applications[2] by Dr. Jeremy H.Brown and Brad Martin. The TL;DR of the article is that Linux is OK for soft real-time applications, but for hard real-time requirements you'll either need RT-PREEMPT or Xenomai. The best solution being Xenomai although it requires more effort to install. On most boards you would need to maintain your own kernel with the Xenomai patch, but luckily for us there is a prepackaged kernel available for the Beaglebone in the official repositories. I told you the community was great.

If you want to install Xenomai on your Beaglebone, I suggest you check the section "Xenomai installation: the easy way (3 steps)" of my article Xenomai installation on a Beaglebone black[3]

Slaying Cerberus: IO setup made easy

In the early years of embedded linux, the boards were dark and full of terrors. IO configuration was a hell of task that required kernel recompilation. Luckily, now we have the device tree[4]. They were introduced as a way of decoupling hardware dependencies from the Linux kernel. And they made IO configuration at runtime possible.

Writing a device tree overlay may seem hard but if you're familiar with IO configuration on microcontrollers it's quite similar. But we are not going to write any device tree overlay. Thanks to **cdsteinkuehler**, we have **beaglebone-universal-io**[5], an amazing tool that reduces pin configuration from writing 30 somewhat complicated lines of configuration to a single line in the command line. So we are going to install this on our board:

```
sudo -s
apt-get install device-tree-compiler -y
cd /opt/source
git clone https://github.com/cdsteinkuehler/beaglebone-universal-io
cd beaglebone-universal-io
make install
exit
```

Now we can start messing with the IOs. For example if I want to have a GPIO at pin 20 from header 9, I can just type

```
sudo config-pin P9.20 gpio
```

And that's it. Told you it was going to be easy. To understand more of what you can do with this tool, go read the documentation.

Finding a low-level library to use the hardware peripherals

There are some neat libraries out there that allow you to use the Beaglebone's peripherals with a nice Python API for instance. I can think of the PyBBIO library[6] by graycatlabs or the Adafruit library[7]. But my favourite is the MRAA library[8] developped by the IOT team at Intel.

It was first meant to be a library for the Edison, but it soon became compatible with the Raspberry Pi and the Beaglebone black (and other boards). That way it provides a nice low-level library that abstracts the hardware to some extent. This library is written entirely in C and it provides severals APIs: C, C++, Python and NodeJS. So you can do anything from real-time robotic applications to Web-based/IOT applications.

To install it, do the following:

```
sudo apt-get install libpcre3-dev git cmake python-dev swig -y
cd ~
git clone https://github.com/intel-iot-devkit/mraa.git
mkdir mraa/build && cd $_
cmake .. -DCMAKE_BUILD_TYPE=DEBUG -DBUILDARCH=arm -DBUILDSWIGNODE=OFF
make
sudo make install
cd ~
```

We are almost done:

```
sudo -s
echo "/usr/local/lib/arm-linux-gnueabihf/" >> /etc/ld.so.conf
exit
sudo ldconfig
sudo echo "export PYTHONPATH=$PYTHONPATH:$(dirname $(find /usr/local -name mraa.py))" >> ~/.bashrc
sudo cp mraa/build/examples/mraa-gpio /usr/bin/
sudo chmod +x /usr/bin/mraa-gpio
sudo mraa-gpio list
```

The last command should output the list of all IOs on the board with the possible configurations for each pin. The workflow is the following: you setup the IO function using the `config-pin` command from **beaglebone-universal-io** and then you use **mraa** library to use that peripheral on that pin.

Now you can go write some cool application. Unless that's not enough for you.

We need to go higher: installing ROS

Let's say you want to build some little robot with computer vision for navigation. You can write your PWM driver to control your motors with the **mraa** library, but how can you do the vision part? A few years ago, I would have told you to install **opencv** and work from there. But you would most certainly get stuck communication-wise: how do you interface your vision code with the motor controller part. It turns out there is a very popular framework out there, in the wild forest of open-source projects, that is awesome at interfacing differents chunks of code that perform different tasks: ROS[9].

ROS stands for Robot Operating System. It's not and OS per se, it's more like a virtual machine that runs on top of Linux. In some ways it's similar to **dbus** as it enables inter process communication, but it's safer to use and, I would argue, easier.

Using ROS to build your robotics application also opens the door to use a wide range of nodes written by other people that do can anything from reading a camera to performing SLAM with stereo vision. So you may even be able to reuse and tweak some existing code to complete your little robot with vision-aided navigation.

Enough with the talking, here are the installation guidelines as documented on the ROS wiki[10]

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu trusty main" > /etc/apt/sources.list.d/ros-latest.list'
wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -O - | sudo apt-key add -

sudo apt-get update
sudo apt-get install ros-indigo-ros-base

sudo apt-get install python-rosdep
sudo rosdep init
rosdep update

echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
source ~/.bashrc

echo "export DISTRIB_ID=Ubuntu" >> ~/.bashrc
echo "export DISTRIB_RELEASE=14.04" >> ~/.bashrc
echo "export DISTRIB_CODENAME=trusty" >> ~/.bashrc
echo "export DISTRIB_DESCRIPTION="Ubuntu 14.04"" >> ~/.bashrc

sudo apt-get install python-rosinstall
```

Now we need to setup the ROS workspace as documented yet again in the ROS wiki[11]

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace

cd ~/catkin_ws
catkin_make
source devel/setup.bash
```

This gives you the base install which is lightweight enough to fit on the Beaglebone black's 4GB eMMC. You can go on and search for ROS packages that may be of interest to your application and look up how to install them and use them.

That's it for this guide. You should now have all the tools you need to start developing on your Beaglebone black. Go make some awesome stuff.

References

1. The beagleboard organisation website http://beagleboard.org/ ↩
2. How fast is fast enough? Choosing between Xenomai and Linux for real-time applications by Dr. Jeremy H.Brown and Brad Martin https://www.osadl.org/fileadmin/dam/rtlws/12/Brown.pdf ↩
3. "Xenomai installation on a Beaglebone black" an article I wrote a month ago http://syrianspock.github.io/embedded-linux/2015/08/03/xenomai-installation-on-a-beaglebone-black.html ↩
4. Linux device tree documentation on eLinux website http://elinux.org/Device_Tree ↩
5. beaglebone-universal-io repository on Github https://github.com/cdsteinkuehler/beaglebone-universal-io ↩
6. PyBBIO library for Beaglebone black repository on Github https://github.com/graycatlabs/PyBBIO ↩
7. Adafruit library for Beaglebone black repository on Github https://github.com/adafruit/adafruit-beaglebone-io-python ↩
8. mraa library repository on Github https://github.com/intel-iot-devkit/mraa ↩
9. ROS (Robot Operating System) website http://ros.org/ ↩
10. Installation of ROS on Ubuntu ARM platforms from the ROS official wiki http://wiki.ros.org/indigo/Installation/UbuntuARM ↩
11. Creating a workspace for catkin from the ROS official wiki http://wiki.ros.org/catkin/Tutorials/create_a_workspace ↩