Accessing the IO ports of the Beaglebone Black with Python

Samuel Bucquet

July 30, 2014

Abstract

The Beaglebone Black is a wonderful little piece of hardware. You could use it to send your next rocket to Mars with just a few line of Python.

With the Beaglebone Black, command all your electronic gear from a few lines of python well sited in your debian system.

Work with serial ports, GPIOs, i2c bus, Analog Input, connect a Digital to Analogic Converter, A Real Time Clock, learn to connect a GPS with PPS to your NTP server.

The Beaglebone black board

Presentation

The **BBB** is a low-cost, low-power, credit card size (3.4“x2.1”) board stuffed with a lot of features. [see BBB_Features.jpg] and costs around 60 bucks.

It sports a **1GHz** Sitara AM335x ARM Cortex-A8 (an ARMv7) processor from Texas Instruments, **512 MB of RAM**, has all the I/O capabilities you'd expect from a typical microcontroller, like access to a CAN bus, SPI interface and i2c, Analog Input, PWM, ... But the board holds also two PRUs, a **µHDMI** video output, a **µSD** card slot, a **100Mb ethernet**. It makes the board **a complete ARM PC** and **fully compatible with Linux**.

Icing on the cake, Beagle fancies an **open hardware philosophy** : all of the chips and designs are available to the public.

Right out of the box you can use is for :

- a great learning platform with an easy acces to the hardware connected. Almost all the functionalities are playable with from the web interface with the *Bonescript* language. Just plug the board via the USB client on a PC, open the page of the board (http://192.168.7.2) and voilà ! See Getting Started[1]

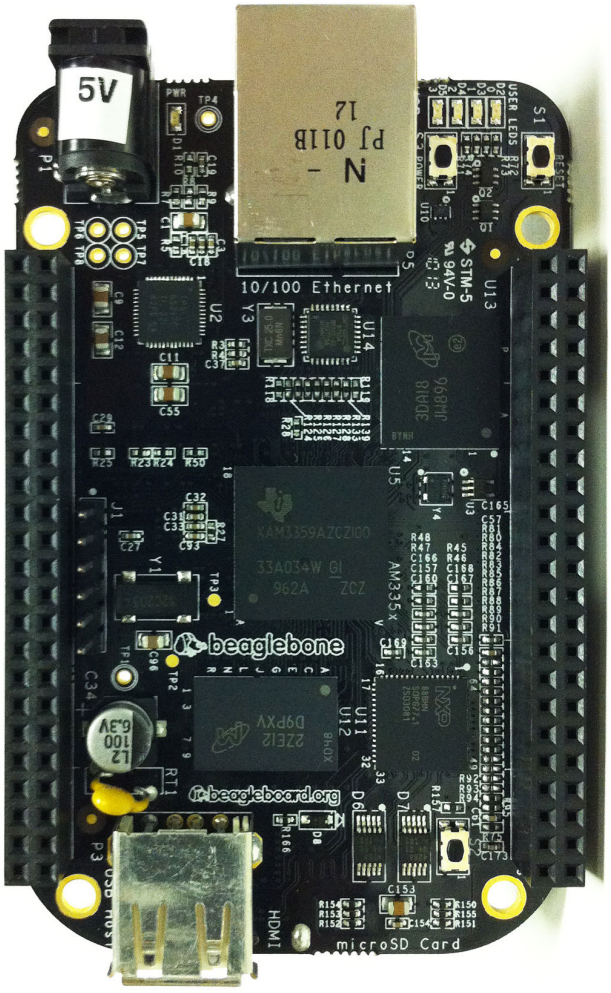


Figure 1: Isn't she beautiful !

[1] http://beagleboard.org/getting-started

	Feature	
Processor	Sitara AM3358BZCZ100 1GHz, 2000 MIPS	
Graphics Engine	SGX530 3D, 20M Polygons/S	
SDRAM Memory	512MB DDR3L 800MHZ	
Onboard Flash	4GB, 8bit Embedded MMC	
PMIC	TPS65217C PMIC regulator and one additional LDO.	
Debug Support	Optional Onboard 20-pin CTI JTAG, Serial Header	
Power Source	miniUSB USB or DC Jack	5VDC External Via Expansion Header
PCB	3.4" x 2.1"	6 layers
Indicators	1-Power, 2-Ethernet, 4-User Controllable LEDs	
HS USB 2.0 Client Port	Access to USB0, Client mode via miniUSB	
HS USB 2.0 Host Port	Access to USB1, Type A Socket, 500mA LS/FS/HS	
Serial Port	UART0 access via 6 pin 3.3V TTL Header. Header is populated	
Ethernet	10/100, RJ45	
SD/MMC Connector	microSD , 3.3V	
User Input	Reset Button Boot Button Power Button	
Video Out	16b HDMI, 1280x1024 (MAX) 1024x768,1280x720,1440x900 ,1920x1080@24Hz w/EDID Support	
Audio	Via HDMI Interface, Stereo	
Expansion Connectors	Power 5V, 3.3V , VDD_ADC(1.8V) 3.3V I/O on all signals McASP0, SPI1, I2C, GPIO(69 max), LCD, GPMC, MMC1, MMC2, 7 AIN(1.8V MAX), 4 Timers, 4 Serial Ports, CAN0, EHRPWM(0,2),XDMA Interrupt, Power button, Expansion Board ID (Up to 4 can be stacked)	
Weight	1.4 oz (39.68 grams)	
Power	Refer to Section 6.1.7	

Figure 2: Beaglebone black features

- a light desktop system if you add a 5VDCC external power supply, a µHDMI cable, a screen, keyboard and Mouse.

This article will focus on working on a BBB from a **Debian** system with **Python** and some mimimalism in mind. But this is still a fully fledged Linux system, not an arduino or a microcontroller.

We will see how to access some of the IO ports :

- the serial ports, to read and write on devices with a RS232 interface like a GPS for example.
- the GPIOs allowing us to trap or send TTL signal, drive a relay, read a button status and in particular, how to add a PPS to Linux.
- the Analog Input voltage for reading voltages coming from a lot of sensors.
- components on the i2c bus : an RTC handled by the system and a DAC driven from our applications.

And we will finish with the BBB as a time server, thanks to our GPS.

What do we use it for

In our project, the BBB boards are embedded in small compartments, only accessible through a network connection. I choose to configure them with the bare minimum, from the eLinux version. There is no X session, no fancy web interface, just the good old command line via ssh and the applications are launched automatically at boot via `/etc/rc.local`.

They are used in a USV (Unmaned Surface Vehicule), the interactions with the operator are done through a hardware control panel via a serial line, through web interfaces and with networked applications. The interactions with the hardware of the USV are done through its many I/O ports of course.

Why Python

Do I really need to tell you ? OK, we have enough processing power, so Python equals less code, more readability of the applications and a lot of useful modules already available. All code and examples are for **Python 2.7**. It would be not very difficult to port it to Python 3.4.

⚠Important usage precautions

There are several precautions that need to be taken when working with the expansion headers to prevent damage to the board :

- **ALL VOLTAGE LEVELS ARE 3.3V MAX**. APPLICATION OF 5V TO ANY I/O PIN WILL DAMAGE THE PROCESSOR AND VOID THE WARRANTY.
- **ANALOG IN VOLTAGES ARE 1.8V MAX**. APPLICATION OF >1.8V TO ANY A/D PIN WILL DAMAGE THE PROCESSOR AND VOID THE WARRANTY.
  - **Do not apply any voltages to any I/O pins when the board is not powered on.**
  - Do not drive any external signals into the I/O pins until after the SYS_RESETn signal is HI (3.3V).
  - Do not apply any voltages that are generated from external sources until SYS_RESETn signal is HI.
  - If voltages are generated from the VDD_5V signal, those supplies must not become active until after the SYS_RESETn signal is HI.
  - If you are applying signals from other boards into the expansion headers, make sure you power the board up after you power up the BeagleBone Black or make the connections after power is applied on both boards.
  - Powering the processor via its I/O pins can cause damage to the processor.

Pimp my BBB

The Debian system

When we received our **BBB** boards in September 2013, they were pre-installed with the *Ångström* distribution. Thanks to the work of others who were also attached to *Debian*, I was quickly able to keep playing with my favorite Linux system on my fresh **BBB**s.

I installed the images provided by elinux[2], but you could also have fetched an image from armhf[3].

Since March 2014, Debian Wheezy (stable) is an official system image available for the Beaglebone Black (rev B and C) latest images[4].

You can choose to upgrade to *testing* (or *sid* if you feel more adventurous) in order to enjoy more recent softwares. [see sidebar 1 - Upgrading from Debian Stable to Debian testing]

The **BBB** accepts booting from the internal memory, the *eMMC*, or from an external µSD. [see sidebar 2 - Booting the BBB from a µSD card]

To test another version of the system, you just have to download and write it on your µSD.

And if you are satisfied with it, you have the option to put it on the eMMC.

As the environment hosting our **BBB**s is subject to strong vibrations, I choose to put my system in the eMMC rather than on a µSD.

Flash the eMMC

In order to flash your new system into your eMMC, download the *flasher* version from eLinux or the official one. Write it to your µSD and boot your **BBB** from the µSD. The process of flashing happens automagically. You have to wait less than 10 minutes before the four blue LEDs become steady, indicating the flashing is over. As the official firmware is much bigger, the flashing will take a lot longer (45 minutes).

⚠*You need to power the board with an external 5VDC power supply when flashing !*

In order to use the armhf version, you will have to partition and format your µSD card following the armhf site instructions[5].

[2] http://elinux.org/BeagleBoardDebian
[3] http://www.armhf.com/boards/beaglebone-black/#wheezy
[4] http://beagleboard.org/latest-images/
[5] http://www.armhf.com/boards/beaglebone-black/bbb-sd-install

You can then download a recent armhf rootfs archive[6] and copy it to your µSD. Then, when booted from the µSD, you can copy your µSD installation to your eMMC likewise.

As the time of writing (July 2014), these Debian images come with a Linux Kernel 3.8.13. This version brought many improvements to the access of the BBB hardware by the kernel via *sysfs*.

Here is a list of packages I recommend for working with the board from the shell and from Python :

```
# apt-get install kbd locales htop vim screen\
rsync build-essential git python-setuptools\
cython python-pip python-virtualenv python-dev\
manpages-{dev,posix{,-dev}} glibc-doc-\
reference python-serial python-smbus python-\
lxml python-psutil i2c-tools
```

For interfacing with a **GPS** with a **PPS**:

```
# apt-get install gpsd python-gps pps-tools \
bison flex git-core
```

If you want to play with **NTPd**:

```
# apt-get install ntp
```

Configure the system

what time is it ?

If a *NTP* server is available to your BBB, good to you, but as the **BBB** lacks a backed battery RTC, it doesn't retain date and time after reboot, you will have to take measures. First, enter manually the date in UTC, **before** anything else.

```
# date -u 072314512014.30
Wed Jul 23 14:51:30 UTC 2014
```

If your BBB must be isolated from a NTP server, one solution is to add a RTC to the board, like a *ds1307*. We will see how to add one on the *i2c bus*.

Finally, if you are isolated and without RTC module, try the *fake-hwclock* package from the Debian repositories. It will allow your clock to restart with the last date saved when the system halted.

[6] http://s3.armhf.com/dist/bone/debian-wheezy-7.5-rootfs-3.14.4.1-bone-armhf.com.tar.xz

to dhcp or not to dhcp

If your network hosts a *dhcp* server then you are good, else you can configure your network card **static** in order to avoid a big dhcp timeout when you boot your BBB with the ethernet cable plugged. [see Sidebar 3 - Configuring the network card static]

a life line (serial debug)

The boards are more often than not in a place where we can't have keyboard and display attached to them. We can work remotely by *ssh*, but if something goes wrong, we need to access the serial debug interface on the board.

Please note that the serial interface available through the USB connection to the board is not ready when you boot with uBoot, you can't see the kernel starting or intervene.

That's why we use the serial debug provided by the **J1** connector on the board, referred to as **ttyO0** by the system.

As a side note, this serial line can be made available via ethernet with a cheap RS232⇔IP converter if remote boot monitoring were needed.

Before connecting our **BBB** on a PC via this serial line, we need a TTL⇔RS232 converter. See some serial debug references[7] on eLinux. You can purchase a PL2303HX USB To RS232 TTL Auto Converter Module, they are very cheap. Just make sure the end of the cable is made of jumper wires and not a fixed connector. [see the picture PL2303HX_USB2RS232TTL_Converter.jpg]

This Code Chief's Space page[8] provides a very thorough step by step guide if needed.

serial login into the system

You know the text consoles with `Login:` you make appear with Ctrl-Alt-F{1..6}, we want the same, but through our serial debug.

We configure a *tty* on the BBB allowing us to connect to it via the serial line when it is booted. Just add this line (if not already present) to the end of */etc/inittab* :

```
T0:23:respawn:/sbin/getty -L ttyO0 115200 vt102
```

And make sure the kernel knows the correct console to output its messages on. In the file */boot/uboot/uEnv.txt*, you have to read :

[7] http://elinux.org/Beagleboard:BeagleBone_Black_Serial

[8] http://codechief.wordpress.com/2013/11/11/beaglebone-black-serial-debug-connection

Figure 3: An essential accessory

```
console=ttyO0,115200n8
#console=tty
```

Finally to connect us to the BBB through our serial debug line from another PC with a USB⇔RS232 adapter :

```
$ screen /dev/ttyUSB0 115200
```

Access the Input/Output ports

Now we want our **BBB** to do some work :

The serial ports

As we previously saw, the serial port must be accessed with a TTL/RS232 adapter. In our project, we reused some old Maxim MAX3232 to do it and it works of course for the other UARTs available on the board.

The **BBB** come with six UARTs but three of them can't be addressed by our applications.

- UART0 is the serial debug on **J1**
- UART3 lacks a RX line
- UART5 can't be used altogether with the HDMI output.

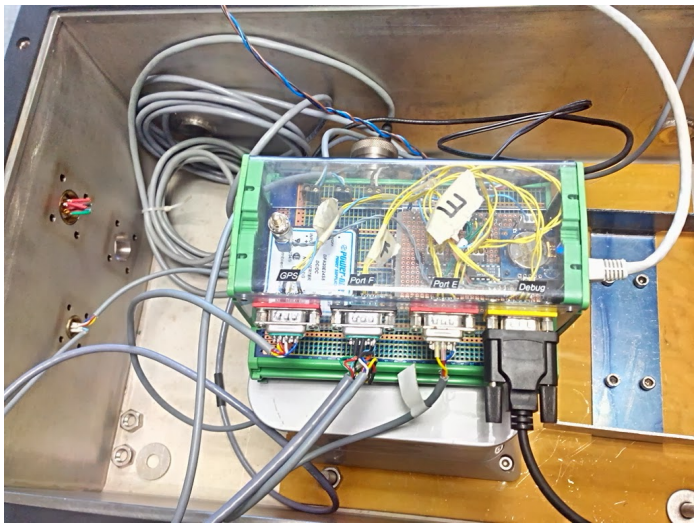


Figure 4: The Board *bbbO2* in situ with three serial ports and the serial debug port, a temperature sensor on AIN0, a water detector on GPIO_48, a PPS input on GPIO_49 and a POWER_RESET button

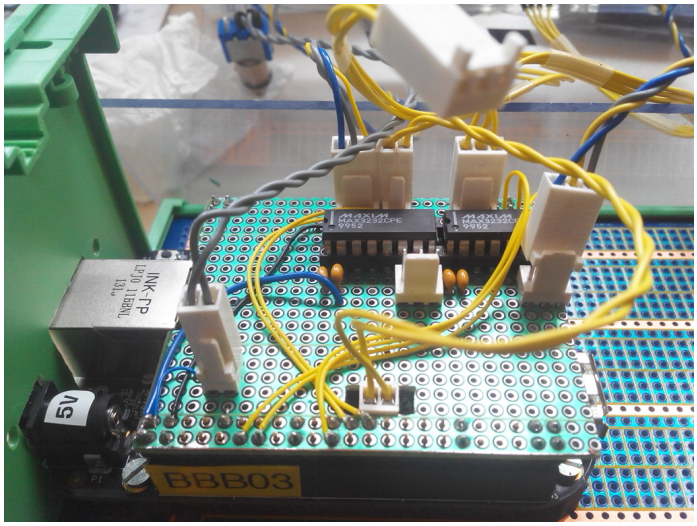


Figure 5: Our CAPE with two MAX3232

So **UART1**, **UART2** and **UART4** are the ones available and they are respectively addressed as `/dev/ttyO1`, `/dev/ttyO2` and `/dev/ttyO4`.

Please note that only UART1 and UART4 have CTS and RTS pin.

Load the CAPE files for each UART :

```
for i in {1,2,4}
do
  echo BB-UART$i > /sys/devices/bone_capemgr.*/\
slots
done
```

The output of `dmesg` should show a correct initialization :

```
# dmesg |grep tty
...
[    1.541819] console [ttyO0] enabled
[  286.489374] 48022000.serial: ttyO1 at MMIO\
0x48022000 (irq = 89) is a OMAP UART1
[  286.627996] 48024000.serial: ttyO2 at MMIO\
0x48024000 (irq = 90) is a OMAP UART2
[  286.768652] 481a8000.serial: ttyO4 at MMIO\
0x481a8000 (irq = 61) is a OMAP UART4
```

Now we can ask the kernel to load them at boot time. Edit `/boot/uboot/uEnv.txt` and modify the line with *optargs* like this :

```
optargs=capemgr.enable_partno=BB-UART1,\
BB-UART2,BB-UART4
```

[see sidebar 5 - passing arguments at boot time]

OK, but does it work ?

To test the serial lines, we just have to connect two of them together [see BBB_testing_UART1et4.jpg]:

UART4_Tx	PIN **13** of P9	⇔	PIN **26** of P9	**UART1_Rx**
UART4_Rx	PIN **11** of P9	⇔	PIN **24** of P9	**UART1_Tx**

Table 1: direct connection of UART1 with UART4

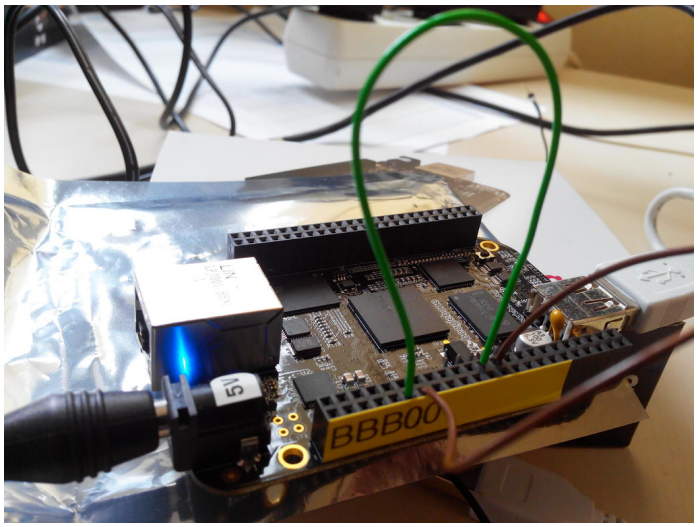


Figure 6: Direct connection between UART1 and UART4

Now launch a screen session on UART1 with '**screen**

/dev/ttyO1 115200', split your window with '**CTRL-A S**', move to the next window with '**CTRL-TAB**', open a screen on UART4 with '**CTRL-A :**' then '**screen /dev/ttyO4 115200**' and check that what you type in one window appears in the other [see screen_testingUARTs.png] :

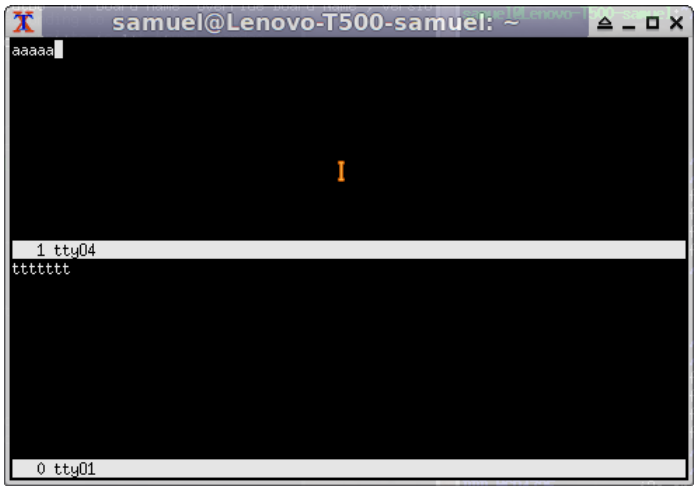


Figure 7: ttyO1 & ttyO4 `screen` session

For now on, we can use the module *python-serial* to read and write on our serial ports like this :

```
import time
from serial import Serial

ctrl_panel = Serial(port=comport, baudrate=9600,
bytesize=8, parity='N', stopbits=1, timeout=0.25 )

# send a commande
ctrl_panel.write(pupitre_commande)

# read an answer
buff = ctrl_panel.read(answer_size)
# as the serial port is configured non-blocking
# with 'timeout=0.25', we make sure all the
# bytes asked for are received.
while len(buff) < answer_size:
  n = len(buff)
  b = ctrl_panel.read(answer_size-n)
  buff += b
  print "------",n,"------"
  time.sleep(0.02)
print "Serial IN: ",
print ' '.join(['%02X' % ord(c) for c in buff])
```

The *python-serial* module is quite efficient. We managed to read two serial ports refreshed at 50Hz plus a third at 5Hz (on the same board and simultaneously) with few glitches between the timestamps and the CPU load stayed below 50%.

The GPIO

The General Purpose Input or Output pins allow the board to receive a signal, read a frequency on input or drive a relay on output.

First we had to understand the mapping between the pin on the P8 and P9 connectors and the GPIO numbers as seen by the kernel. The *BBB System Reference Manual*[9] gives the Expansion Header P9 pinout.

For each GPIO, the name is made of the GPIO Controller number between 0 and 3 and the pin number on the Sitara AM3359AZ. The kernel numbers the GPIO pin with **PIN + (GPIO Controller number * 32)**. If we pick the *GPIO1_16* on the pin 15 of the P9 connector, the kernel will see it as *GPIO_48* (16+(1 * 32)).

[See BBB_GPIO_Pinouts_P8_P9.png] to read the direct mapping for all the GPIOs.

65 possible digital I/Os

P9

DGND	1	2	DGND
VDD_3V3	3	4	VDD_3V3
VDD_5V	5	6	VDD_5V
SYS_5V	7	8	SYS_5V
PWR_BUT	9	10	SYS_RESETN
GPIO_30	11	12	GPIO_60
GPIO_31	13	14	GPIO_40
GPIO_48	15	16	GPIO_51
GPIO_4	17	18	GPIO_5
I2C2_SCL	19	20	I2C2_SDA
GPIO_3	21	22	GPIO_2
GPIO_49	23	24	GPIO_15
GPIO_117	25	26	GPIO_14
GPIO_125	27	28	GPIO_123
GPIO_121	29	30	GPIO_122
GPIO_120	31	32	VDD_ADC
AIN4	33	34	GNDA_ADC
AIN6	35	36	AIN5
AIN2	37	38	AIN3
AIN0	39	40	AIN1
GPIO_20	41	42	GPIO_7
DGND	43	44	DGND
DGND	45	46	DGND

P8

DGND	1	2	DGND
GPIO_38	3	4	GPIO_39
GPIO_34	5	6	GPIO_35
GPIO_66	7	8	GPIO_67
GPIO_69	9	10	GPIO_68
GPIO_45	11	12	GPIO_44
GPIO_23	13	14	GPIO_26
GPIO_47	15	16	GPIO_46
GPIO_27	17	18	GPIO_65
GPIO_22	19	20	GPIO_63
GPIO_62	21	22	GPIO_37
GPIO_36	23	24	GPIO_33
GPIO_32	25	26	GPIO_61
GPIO_86	27	28	GPIO_88
GPIO_87	29	30	GPIO_89
GPIO_10	31	32	GPIO_11
GPIO_9	33	34	GPIO_81
GPIO_8	35	36	GPIO_80
GPIO_78	37	38	GPIO_79
GPIO_76	39	40	GPIO_77
GPIO_74	41	42	GPIO_75
GPIO_72	43	44	GPIO_73
GPIO_70	45	46	GPIO_71

Figure 8: Mapping GPIO Kernel numbers with P8 & P9 pinouts

And you can find a similar table for each kind of IO port of the BBB on this Cape Expansion headers page[10].

operate the GPIO

The file *gpio_sysfs.txt*[11], provided with your kernel documentation, explains how to work with GPIOs with a Linux kernel.

These steps are for the 3.8.13 version of the Linux kernel.

Each GPIO must be enabled independently, so for the GPIO_48 on P9_15 :

[9] http://elinux.org/Beagleboard:BeagleBoneBlack#Hardware_Files
[10] http://elinux.org/Beagleboard:Cape_Expansion_Headers
[11] https://www.kernel.org/doc/Documentation/gpio/sysfs.txt

```
# echo 48 > /sys/class/gpio/export
```

Next we choose the way it will operate :

- If we want to process input signals :

  ```
  # echo in > /sys/class/gpio/gpio48/direction
  ```

  then we choose if we detect 'rising' or 'falling' edge or 'both':

  ```
  # echo both > /sys/class/gpio/gpio48/edge
  ```

- If we want to output signals, we may choose 'out' or one of 'high' or 'low' (telling when the signal is active) :

  ```
  # echo high > /sys/class/gpio/gpio48/direction
  ```

  and to stop emitting the signal :

  ```
  # echo 0 > /sys/class/gpio/gpio48/value
  ```

- At the end, we release the GPIO :

  ```
  # echo 48 > /sys/class/gpio/unexport
  ```

wait for a signal on a GPIO from Python

We connected an optical water detector to one of our BBB inside an enclosure, on the GPIO_48 (P9_15). [see picture BBB02_InSitu.jpg]

The water detector sends a TTL signal if there is water passing through its lens. Here is how we wait for an event describing water presence from Python using the BBB_GPIO class from *bbb_gpio.py* :

```
from bbb_gpio import BBB_GPIO

water = BBB_GPIO(48,gpio_edge='both',\
    active_low=True)
for value in water:
    if value:
        print "Water detected !"
    else:
        print "No more water !"
```

The BBB_GPIO class is a generator. For each iteration, we wait for a change on the GPIO and return in `value` the GPIO status. When the GPIO is waiting for an event, we don't want to be polling aggressively on the system, but to be awaken only when the event occurs. That's what the `poll()` system call does. [see *bbb_gpio.py* line 58]

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# --- bbb_gpio.py ---
# Author  : samuel.bucquet@gmail.com
# Date    : 20.02.2014
# License : GPLv2

from select import epoll, EPOLLPRI, EPOLLERR


class BBB_GPIO(object):

    _sysfs_path = '/sys/class/gpio/'

    def __init__(
        self,
        gpio_number,
        gpio_edge='rising',
        active_low=False,
        ):
        open(self._sysfs_path + 'unexport', 'w'
             ).write(str(gpio_number))
        open(self._sysfs_path + 'export', 'w'
             ).write(str(gpio_number))
        self.path = self._sysfs_path + 'gpio%d/' \
            % gpio_number
        open(self.path + 'direction', 'w'
             ).write('in')
        open(self.path + 'edge', 'w'
             ).write(gpio_edge)
        if active_low:
            open(self.path + 'active_low', 'w'
                 ).write('1')
        self.ep = epoll()
        self.read()

    def read(self):
        self.value = int(open(self.path + 'value'
                              , 'r').read())

    def get(self):
        return self.value

    def next(self):
        self.wait()
        return bool(self.value)

    def __iter__(self):
        return self

    def wait(self):

        # we don't want buffering on this one

        fd = open(self.path + 'value', 'r', 0)
        self.ep.register(fd, EPOLLPRI | EPOLLERR)
        self.ep.poll()
        self.ep.unregister(fd)
        self.value = int(fd.read())
        fd.close()
```

Figure 9: bbb_gpio.py

a special case, the PPS signal

A GPS often delivers a PPS (Pulse Per Second) signal in order to synchronize with good accuracy the timing NMEA sentences like *$GPZDA*.

The PPS signal is a TTL we can connect to a GPIO, we choose GPIO_49 (P9_23). Once wired we check if the signal is present and can be read :

```
# echo 49 > /sys/class/gpio/export
# echo in > /sys/class/gpio/gpio49/direction
# echo rising /sys/class/gpio/gpio49/edge
# cat /sys/class/gpio/gpio49/value
1
```

⚠Be careful to the output voltage of the PPS as the GPIOs of the BBB accepts **a TTL of 3.3V max**.

The PPS signal is also a special input understood by the Linux kernel.

In order to enable our GPIO input as a PPS, we have to compile a *dts* (Device Tree Source file) into a *dtbo* (Device Tree Binary Object file) with the tool `dtc` (Device Tree -you guess- Compiler). [see the file GPS-PPS-P9_23-00A0.dts]

```
# ./dtc -O dtb -o GPS-PPS-P9_23-00A0.dtbo -b 0 \
-@ GPS-PPS-P9_23-00A0.dts
```

⚠The `dtc` program available in Debian Wheezy is not able to write a dynamically loadable *dtbo* file, it may lacks the `-@` option depending of the system you installed. Check the output of `dtc -h` and if the `-@` is not present [see sidebar 4 - fetching a good `dtc`].

The *dtbo* file produced will then be loaded to the kernel :

```
# cp GPS-PPS-P9_23-00A0.dtbo /lib/firmware
# echo GPS-PPS-P9_23 > /sys/devices/\
bone_capemgr.*/slots
```

To verify that the PPS is correctly seen by the Linux kernel, we need the `pps-tools` package installed.

```
# ppstest /dev/pps0
trying PPS source "/dev/pps0"
found PPS source "/dev/pps0"
ok, found 1 source(s), now start fetching data...
source 0 - assert 1391436014.956450656, sequence:\
 202 - clear  0.000000000, sequence: 0
source 0 - assert 1391436015.956485865, sequence:\
 203 - clear  0.000000000, sequence: 0
source 0 - assert 1391436016.956517240, sequence:\
 204 - clear  0.000000000, sequence: 0
source 0 - assert 1391436017.956552407, sequence:\
 205 - clear  0.000000000, sequence: 0
...
(Ctrl-C to end)
```

We see here a signal received at 1 Hz, with a timestamp jitter less than 1 ms.

The I2C bus

Two i2c buses on the BBB are available. The Kernel see them as *i2c-0* for I2C1 on P9_17(SCL) & P9_18(SDA), and *i2c-1* for I2C2 on P9_19(SCL) & P9_20(SDA).

add an RTC to the BBB

The DS1307 or the DS3231 are RTC modules with a battery keeping the clock running when there is no power. The ChronoDot RTC (with a ds3231) is much more accurate and the ds1307 is much less expensive.

wire the RTC on i2c

You can feed the 5VDCC of the board to the RTC module **as long as you clip out the two 2.2k resistors**. [see Adafruit_DS1307_with_resistors_clipped.jpg] The internal resistors of the bbb i2c bus will then be used.

P9	ds1307
pin 1	GND
pin 5	5VCC
pin 19	SDA
pin 20	SCL

Table 2: ds1307 Wiring on the BBB i2c_2 bus

⚠if you power the BBB over USB, use P9_7 (SYS 5V) instead.

enable the new RTC

Declare the new RTC to the kernel :

```
# echo ds1307 0x68 >
/sys/class/i2c-adapter/i2c-1/new_device
[   73.993241] rtc-ds1307 1-0068: rtc core: \
registered ds1307 as rtc1
[   74.007187] rtc-ds1307 1-0068: 56 bytes \
nvram
[   74.018913] i2c i2c-1: new_device: \
Instantiated device ds1307 at 0x68
```

Push the current UTC date to the ds1307 :

```
/*
 * Copyright (C) 2014 samuel.bucquet@gmail.com
 * based on the work of the8thlayerof.net
 *
 * GPS cape for a PPS TTL on P9_23
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */
/dts-v1/;
/plugin/;

/ {
    compatible = "ti,beaglebone", "ti,beaglebone-black";

        /* identification */
        part-number = "GPS-PPS";
        version = "00A0";

        /* state the resources this cape uses */
        exclusive-use =
                /* the pin header uses */
                "P9.23",        /* gpio1_17   */
                /* the hardware ip uses */
                "gpio1_17";

        fragment@0 {
                target = <&am33xx_pinmux>;
                __overlay__ {
                        gps_pps_pins: pinmux_gps_pps_pins {
                                pinctrl-single,pins = <
                                        0x44 0x27 /* P9.23 gpio1_17 pulldown */
                                >;
                        };

                };
        };

        fragment@1 {
                target = <&ocp>;

                __overlay__ {
                    pps {
                        compatible = "pps-gpio";
                        status = "okay";
                        pinctrl-names = "default";
                        pinctrl-0 = <&gps_pps_pins>;
                        gpios = <&gpio2 17 0 >;
                        assert-rising-edge;
                    };
                };
        };
};
```

Figure 10: GPS-PPS-P9-23-00A0.dts

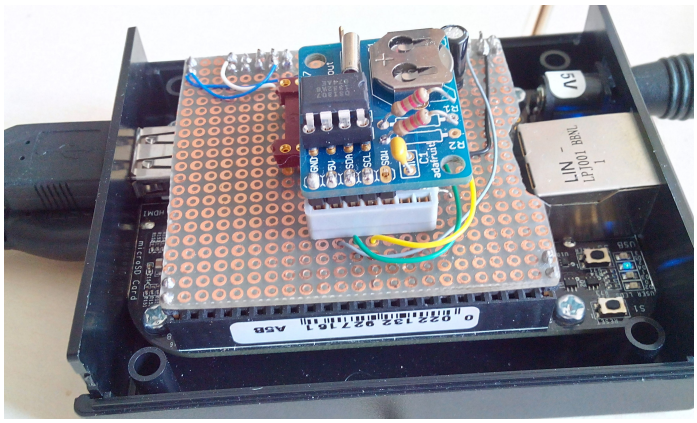


Figure 11: The Adafruit RTC ds1307 module wired on the BBB

```
# hwclock -u -w -f /dev/rtc1
```

Verify the clock

```
# hwclock --debug -r -f /dev/rtc1
hwclock from util-linux 2.20.1
Using /dev interface to clock.
Last drift adjustment done at 1406096765 seconds \
after 1969
Last calibration done at 1406096765 seconds after\
 1969
Hardware clock is on UTC time
Assuming hardware clock is kept in UTC time.
Waiting for clock tick...
...got clock tick
Time read from Hardware Clock: 2014/07/23 15:42:51
Hw clock time : 2014/07/23 15:42:51 = 1406130171 \
seconds since 1969
Wed Jul 23 17:42:51 2014  -0.438131 seconds
```

To permanently benefit from the new RTC as soon as the system boots, modify the file */etc/init.d/hwclock.sh* with a little dirty hack. Add to the end of the file :

```
echo ds1307 0x68 > /sys/class/i2c-adapter/i2c-1/\
new_device
HCTOSYS_DEVICE=rtc1
hwclocksh "$@"
```

⚠I had to comment the udev part of the file, and I'm still trying to figure how to do that part in a cleaner way :

```
#if [ -d /run/udev ] || [ -d /dev/.udev ]; then
#     return 0
#fi
```

if something goes wrong with a component on i2c

If the kernel can't see the ds1307 for example, try to detect it on the i2c bus with :

```
# i2cdetect -y -r 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- UU UU UU UU -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- 68 -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

if **68** is not here, but **UU** is printed, the kernel already keeps hold of the ds1307 with a driver. Recheck your kernel messages and try to unload the driver for the ds1307 with :

```
# echo 0x68 >
/sys/class/i2c-adapter/i2c-1/delete_device
# rmmod rtc_ds1307
```

and retry.

If this is just `--` instead of `68`, recheck your wiring.

add a DAC, the MCP4725, on i2c

The MCP4725 is a 12 bits Digital Analog Converter, allowing us to output a voltage from a numerical value between 0 and 4095 (2^{12} - 1). Its EEPROM allows us to store a value in a register wich become the default value as soon as the DAC is powered on. When you want to drive a motor with it, you can then store a default safe value in EEPROM and you make certain that when the power is restored, your motor doesn't start full speed.

The wiring is almost identical as for the ds1307 module, we take the 3.3V VDD (P9_3) as VREF for the MCP4725.

The MCP4725 address on i2c is **0x60** or **0x61**, you select it with the A0 pin on the MCP4725. Thus you can use two MCP4725 on the same bus, so with two i2c buses you can easily output four independant voltages from your BBB.

write a value to the MCP4725

To set a voltage ouput on the MCP4725, we write to the i2c bus. So for a value of `0x0FFF` (4095) at the address `0x60` on *i2c-1*:

```
$ i2cset -f -y 1 0x60 0x0F 0xFF
```

write to the MCP4725 from Python

The Python module to access the MCP4725 [see file `i2c_mcp4725.py`] is a bit more complex because we try to handle all the functionnalities of the DAC. It depends on the python-smbus package.

Here is how we use it :

```
import time
from smbus import SMBus
from i2c_mcp4725 import MCP4725

dac = MCP4725(SMBus(1), int('0x60', 16))
safe_value = 2047
# write to the DAC and store the value in EEPROM
dac.write_dac_and_eeprom(safevalue, True)

# read the value ouput by the dac and the content
# of its EEPROM
dac.read_and_eeprom()
print dac

# send a mid-ramp 1.69V to 3.38V
for value in range(2048,4096):
  # write to the DAC without storing value
  dac.write_dac_fast(value)
  time.sleep(0.2)
```

The Python subtleties allow us to do simply :

```
# send a new value to output a voltage
dac(new_value)
# check the value currently ouput by the DAC
current_value = dac()
```

How to read and verify the output voltage ?

The BBB has 7 Analog INput ports named AIN{0..6}. They are 12 bits Analog Digital Converter. They accept a max voltage of **1.8V**. To read a voltage from an Analog Input, we wire the GND (P9_1), and the + of the voltage we want to measure.

We reinject the output of our MCP4725 in AIN0. In this case the VDD for the MCP4725 is 3.38V (the 3.3V of the PIN 3&4 of P9). We divide it by 2 with two resistors as it musts not be superior to 1.8V. And we wire the output of the resistors to P9_39 (AIN0).

The vRef for the MCP4725 is VMAX (3.38V) so the voltage we send is :

$V_{out} = out\_value \times \frac{V_{max} \times 0.5}{4096} = out\_value \times \frac{1.69}{4096}$

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# --- i2c_mcp4725.py ---
# Author  : samuel.bucquet@gmail.com
# Date    : 10.09.2013
# License : GPLv2

from math import ceil


class MCP4725(object):

    _WRITE_DAC = 0x40
    _WRITE_DAC_AND_EEPROM = 0x60

    def __init__(
        self,
        bus,
        address=0x60,
        vRef=3.3,
        ):
        self.bus = bus
        self.address = address
        self.vRef = vRef
        self.read_dac_and_eeprom()

    def read_dac_and_eeprom(self):
        buff = \
            self.bus.read_i2c_block_data(
                    self.address, 0x00, 5)
        self.lastread = ' '.join('%02X' % d
                for d in buff)
        self.ready = bool(buff[0x00] & 0x80)
        self.power_on_reset = bool(buff[0x00]
                & 0x40)

        # powerdown : see page 20, table 5-2

        self.powerdown = (buff[0x00] & 0x06) >> 1
        self.dac_register = buff[1] << 4 & 0x0FF0 \
            | buff[2] >> 4 & 0x0F
        self.eeprom_data = buff[3] << 8 & 0x0F00 \
            | buff[4]

    def write_dac_fast(self, value):
        if not 0x00 <= value < 4096:
            raise ValueError
        self.bus.write_byte_data(self.address,
                value >> 8 & 0x0F, value & 0xFF)

    def write_dac_and_eeprom(self, value,
            do_store_value=True):
        if not 0x00 <= value < 4096:
            raise ValueError
        cmd = \
            (self._WRITE_DAC_AND_EEPROM \
            if do_store_value \
            else self._WRITE_DAC)
```

Figure 12: i2c_mcp4725.py

```
60          buff = [value >> 4 & 0xFF, value << 4
61                  & 0xFF]
62          self.bus.write_i2c_block_data(self.address,
63                  cmd, buff)
64
65      def __call__(self, value=None):
66          if value is None:
67              self.read_dac_and_eeprom()
68              return self.dac_register
69          else:
70              self.write_dac_fast(value)
71
72      @property
73      def register(self):
74          return self()
75
76      @register.setter
77      def register(self, value):
78          self(value)
79          self.read_dac_and_eeprom()
80
81      def __str__(self):
82          return """\t%s
83      Ready=%s|PowerOnReset=%s|PowerDown=%d
84      DAC_Register=%d|EEPROM_Data=%d
85  """ \
86              % (
87              self.lastread,
88              str(self.ready),
89              str(self.power_on_reset),
90              self.powerdown,
91              self.dac_register,
92              self.eeprom_data,
93              )
94
95      def voltage2register(self, voltage):
96          return (None if voltage
97                  > self.vRef else int(ceil(voltage
98                  * (4096.0 / self.vRef))))
99
```

Figure 13: i2c_mcp4725.py

The vRef for the AIN is always 1.8V, so in order to convert our 12 bits numerical value into a voltage reading we have :

$V_{in} = in\_value \times \frac{1.8}{4096}$

For a numerical raw value *out*, we must read a numerical raw value *in* such as :

$in\_value = int(out\_value \times \frac{1.69}{1.8})$

read AIN0 from sysfs

If the BBB ADC kernel driver is not loaded, load it now with :

```
# echo BB-ADC /sys/devices/bone_capemgr.8/slots
```

If you need it loaded automatically at boot, do like we did for the BB-UARTs. [see sidebar 5 - passing arguments at boot]

- To read a raw numerical value on AIN0 :

```
$ cat /sys/bus/iio/devices/iio\:device0/\
in_voltage0_raw
3855
$
```

is the input consistent with the output ?

We can see that the value sent to the MCP4725 is correctly reread on AIN0.

OUT raw value	IN theoretical raw value	IN read raw value
255	239	**243**
2047	1927	**1929**
3839	3614	**3613**
4095	3855	**3835**

Table 3: Discrepancies between DAC output and ADC input.

The values match but there is still some inaccuracies. We need to record a table of corresponding values between the MCP4725 and AIN0 in this configuration. Our control loop driving our propellers speed can better handle the reinjection of the output voltage.

read AIN0 from Python

To read a temperature sensor wired on AIN0 giving 1mV for 0.1°C, we use the BBB_AIN class from the file *bbb_ain.py* :

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-

# --- bbb_ain.py ---
# Author  : samuel.bucquet@gmail.com
# Date    : 28.02.2014
# License : GPLv2


class BBB_AIN(object):

    """
    Access to the sysfs interface of
    the AIN{0..6} of the BBB
    """

    _sysfs_path = \
        '/sys/bus/iio/devices/iio:device0/'
    _to_milivolts = 1800.0 / 4096

    def __init__(
        self,
        ain_idx,
        coefficient=_to_milivolts,
        valeurmax=None,
        ):
        if not 0 <= ain_idx < 7:
            raise ValueError
        self.fsname = self._sysfs_path \
            + 'in_voltage%d_raw' % ain_idx
        self.coeff = (coefficient if valeurmax
                      is None else valeurmax
                      / 4096.0)

    def __iter__(self):
        return self

    def next(self):

        # we don't want buffering on this one

        with open(self.fsname, 'r', 0) as \
            ain_file:
            while True:
                try:
                    line = ain_file.read()
                    break
                except IOError, e:
                    if e.errno != 11:
                        return None

                    # IO Error -- Resourcer
                    # temporarily unavailable --
                    # just retry and it will do !

                    continue
                self.raw_value = int(line)
                return self.raw_value * self.coeff
```

Figure 14: bbb_ain.py

```python
import time
from bbb_ain import BBB_AIN

tempe = BBB_AIN(0, valeurmax=180)
for value in tempe:
  print "%.4f" % value
  time.sleep(0.5)
```

Once again we use a generator. At each iteration we read a value on the AIN. There is no interrupt mechanism on the AIN, we read at the frequency of the `for loop`. That's why we have to pause at each iteration.

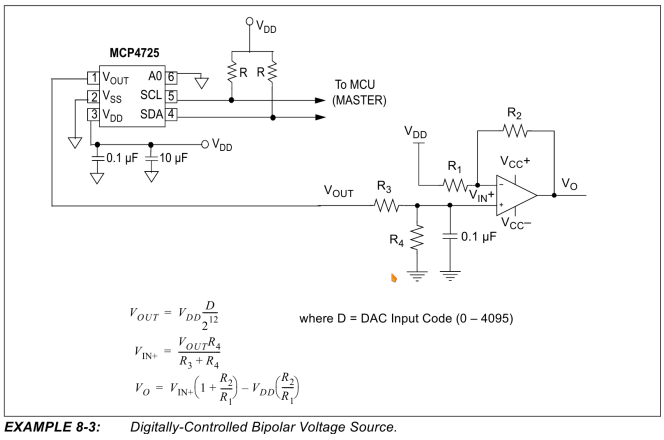
The only catch is the error "Resource Temporarily unavailable" (error=11) which may occur occasionaly.

how we use it

To pilot our propellers, we need to output a voltage between -10 V and 10 V.

We use two MCP4725 on *i2c_1*. One with A0 on V_{SS} (0x60) and the other with A0 on V_{DD} (0x61).

We use a *bipolar operation* type circuit to output -10V/10V from the MCP4725's 0/3.3V output.



EXAMPLE 8-3: *Digitally-Controlled Bipolar Voltage Source.*

Figure 15: MCP4725 - Bipolar Operation Circuit

The MCP4725 outputs are reinjected into AIN0 & AIN1 in order to improve the control loop accuracy. And we read the speed of the propellers back with two frequency/voltage converters, which we feed to AIN2 & AIN3. [see BBB04_InSitu.jpg]

The BBB as time server from a GPS.

What if you put together the interface to your GPS, the NTP server of your Linux machine equipped with an RTC and the accuracy provided with the PPS signal ?

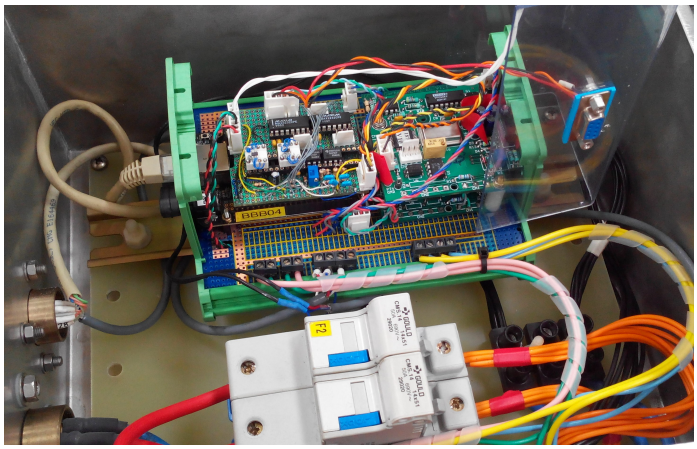


Figure 16: The Box in charge of the propellers

You can build a NTP server for your other CPUs in your network and with good accuracy as soon as the GPS is aligned.

⚠Keep in mind that we need a good RTC wired to the BBB for a real NTP server. So choose one like the Adafruit Chronodot[12] rather than the simplest DS1307.

GPSd

Now that we know how to handle a serial port, we can install the software **GPSd**. GPSd connects to a local GPS, as the one we wired to UART4, and serves GPS data to clients.

```
# apt-get install gpsd python-gps gpsd-clients
```

Edit the `/etc/default/gpsd.conf` file and modify it to connect to your serial port (here `tty04`) and tell GPSd to listen to all network interfaces (`-G`) :

```
START_DAEMON="true"
GPSD_OPTIONS="-G -n"
DEVICES="/dev/tty04"
USBAUTO="false"
GPSD_SOCKET="/var/run/gpsd.sock"
```

Then restart it :

```
# /etc/init.d/gpsd stop
# /etc/init.d/gpsd start
```

By now your GPS is available to all clients on your network, you can try to connect to GPSd with *QGIS* or *OpenCPN* for example, but a rather simple solution is with `gpsmon`. On another machine with the `gpsd-clients` package installed, launch :

[12] http://www.adafruit.com/products/255

```
$ gpsmon tcp://bbb02:2947
```

and you obtain a screen like [see gpsmon_screenshot.png]

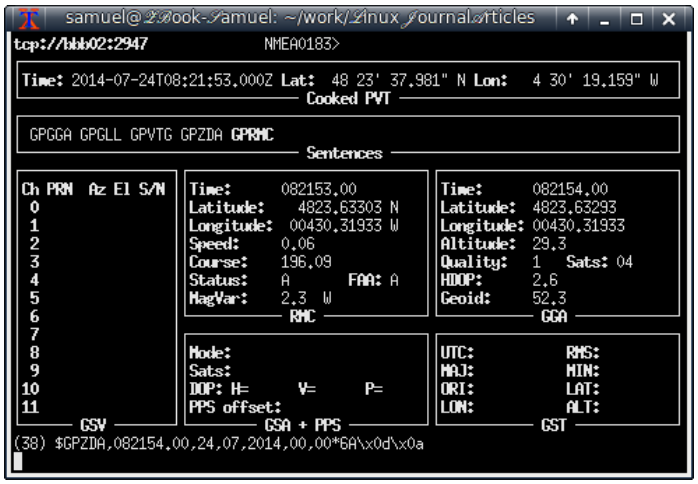


Figure 17: A session with `gpsmon`

connect to GPSd from Python

The package `python-gps` provides what we need, but the dialogue sequence with GPSd is not trivial. I wrote a little Python class `GPSd_client` [see `gpsd_client.py` file] in order to be able to access my GPS like this :

```
$ ipython
Python 2.7.8 (default, Jul 22 2014, 20:56:07)
Type "copyright", "credits" or "license" for more \
information.
...
In [1]: from gpsd_client import GPSd_client

In [2]: gps = GPSd_client('bbb02')

In [3]: for gpsdata in gps:
    print gpsdata
   ...:
GPS(time=u'2014-07-24T08:53:54.000Z', latitude=\
48.3938485,longitude=-4.505373, altitude=\
30.4, sog=0.051, cog=187.71, ept=0.005, mode=3)
GPS(time=u'2014-07-24T08:53:55.000Z', latitude=\
48.393848, longitude=-4.505373167, altitude=\
30.3, sog=0.067, cog=194.8, ept=0.005, mode=3)
GPS(time=u'2014-07-24T08:53:56.000Z', latitude=\
48.393847667, longitude=-4.505373167, altitude=\
30.2, sog=0.062, cog=184.8, ept=0.005, mode=3)
GPS(time=u'2014-07-24T08:53:57.000Z', latitude=\
48.393847333, longitude=-4.505373167, altitude=\
30.2, sog=0.036, cog=189.77, ept=0.005, mode=3)
GPS(time=u'2014-07-24T08:53:58.000Z', latitude=\
```

```
48.393847167, longitude=-4.505373, altitude=\
30.1, sog=0.041, cog=175.46, ept=0.005, mode=3)
^C-----------------------------------------\
----------------------------
```

Again, we use a generator. For each iteration, the `GPSd_client` class opens a session with GPSd, listens for a report with position and time information, closes the session and returns a `namedtuple` with informations wanted.

NTPd

One of these GPSd clients is **NTP**. NTPd process the NMEA GPS timing sentences to set the date and use the PPS signal to be more accurate.

The handling of PPS signal is available only in the development versions of NTPd, not the version in the Debian repositories.

The version we installed is `ntp-dev-4.2.7p416`. Grab it and compile this way :

```
# apt-get install libcap-dev
# wget http://www.eecis.udel.edu/~ntp/ntp_spool/\
ntp4/ntp-dev/ntp-dev-4.2.7p416.tar.gz
# tar xf ntp-dev-4.2.7p416.tar.gz
# cd ntp-dev-4.2.7p416/
# ./configure --enable-all-clocks --enable-\
linuxcaps
# make
```

Modify the `/etc/ntp.conf` file for the connection to GPSd and the PPS signal listening :

```
# Server from shared memory provided by gpsd
server 127.127.28.0 prefer
fudge 127.127.28.0 time1 0.040 refid GPS

# Kernel-mode PPS ref-clock for the precise seconds
server 127.127.22.0
fudge 127.127.22.0  flag2 0 flag3 1  refid PPS

# allow ntpd to serve time if GPS is OFF
tos orphan 5
```

`0.040` might need adjusting relative to your GPS but is a safe bet. `127.127.22.0` is the NTPd reference to `/dev/pps0`. If your GPSd declare a `/dev/pps` to the kernel before you, your real PPS signal might become `/dev/pps1`. Bottom line : try to load your PPS signal before starting GPSd.

Verify that NTPd has started and serves the time with `ntpq` :

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-

# --- gpsd_client.py ---
# Author  : samuel.bucquet@gmail.com
# Date    : 27.03.2014
# License : GPLv2

import gps
from collections import namedtuple


class GPSd_client(object):

    def __init__(self, gpsd_host):
        self.host = gpsd_host
        self.GPS = namedtuple('GPS',
                'time latitude longitude altitude\
                sog cog ept mode'
                )

    def __iter__(self):
        return self

    def next(self):
        session = gps.gps(host=self.host)
        session.stream(gps.WATCH_ENABLE
                       | gps.WATCH_NEWSTYLE)
        for report in session:
            if report['class'] == 'TPV' \
                and report['tag'] == 'RMC':
                trame = dict(report)
                break
        session.close()
        return self.GPS(*[trame[k] for k in (
            'time',
            'lat',
            'lon',
            'alt',
            'speed',
            'track',
            'ept',
            'mode',
            )])

```

Figure 18: gpsd_client.py

```
# ntpq -p
     remote           refid      st t when poll \
     reach   delay   offset  jitter
==============================================\
==============================
*SHM(0)          .GPS.            0 l   13   64 \
  377    0.000  -50.870   0.886
oPPS(0)          .PPS.            0 l   12   64 \
  377    0.000   -1.419   0.128
```

You can see here that `.GPS.` is identified as the system peer by ntpd (`*`) and `.PPS.` is also correctly recognised and valid (`o`).

The PRU, a very hot topic

The two Programmable Realtime Units of the BBB can work independently of the main CPU on IO ports, AINs and PWM. They are sophisticated enough to share memory with CPU up to 300MB, have a rich instruction set and can trigger or receive interrupts.

Recently, some hard workers managed to make more accessible the use of the BBB PRUs. And there is this great promising GSOC 2014 coming : **Beaglelogic**[13].

See also the work of Fabien Le Mentec : "Using the Beaglebone PRU to achieve realtime at low cost[14]"

Conclusion

The Beaglebone Black is a very fun platform to play with. As a Linux sysadmin for nearly twenty years, I'm very comfortable using it to access electronic hardware I'm not so well acquainted with usually.

I have to thank my co-worker, Rodolphe Pellaë, whose skills in electronic were essential to connect all the components to the board. He did our four home made Capes in no time and did it well.

The community orbiting the BBB is big with a lot of good resources on line. We managed to realize some complex stuff with very little time and almost no complexity because we can profit of the work of those people and from the Linux and Python ecosystems.

Bio

Samuel Bucquet is a system developer and a sysadmin on robotic platforms in the french DOD. Married with four kids and living in Brest, France, he is a long time Linux aficionado.

Resources

Python libraries on Github

You will find the source of the Python code used in the article on my github account[15].

PyBBIO[16] is a Python Library for the BBB mimicking the Arduino IO access by Alexander Hiam.

Of course there is also the *Python Adafruit Library*[17], initially for the Raspberry pi, now for the BBB.

[13] https://github.com/abhishek-kakkar/BeagleLogic/wiki
[14] http://www.embeddedrelated.com/showarticle/586.php
[15] https://github.com/samgratte/BeagleboneBlack
[16] https://github.com/alexanderhiam/PyBBIO
[17] https://github.com/adafruit/adafruit-beaglebone-io-python