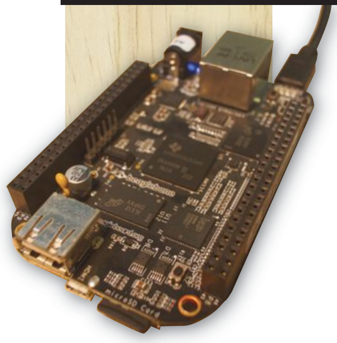
INTRODUCTION TO Hardware Hacking on FreeBSD

by Tom Jones

FreeBSD is the perfect platform for maker and hardware hacking projects because we manage to offer such a solid and uniform support across a wide range of boards.


In this article we use the BeagleBone Black Single Board Computer.

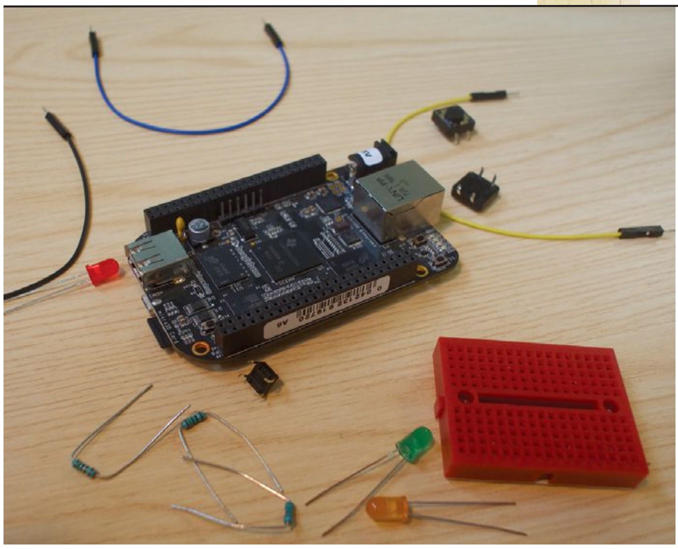
FreeBSD has excellent support for many different ARM Single Board Computers (SBCs). Linux ships with most SBCs, and each SBC seems to come with its own Linux distro.

Normally the Linux distro with the best support for your board is the one from the board manufacturer that leaves you with a fractured set of distributions and varying support for packages and inconsistent releases. FreeBSD does a lot better—we don't always have support for boards right away, but the boards we do support tend to give the same great experience. We have a single, consistent system that runs the same on all the boards we support. If it runs FreeBSD, it runs the same FreeBSD as on a desktop system.

This article will show you how to use FreeBSD's excellent hardware support to control real devices. We will do this by diving into how to use GPIO on FreeBSD on an ARM SBC.

General Purpose Input/Output (GPIO) devices give us a window from the computer to cause a change in the real world. This is normally implemented by writing to a special part of memory or using processor register to control something in the real world. On ARM SoCs, this is normally implemented using a mapped piece of memory that, when written to, allows us to control a voltage on an exposed pin. FreeBSD provides device drivers to allow processes in userspace to interact with GPIO with the same interface across all supported hardware platforms.

GPIO devices are not available on all systems, but they are present on a surprising number. Normally, exposed GPIO can be spotted as pin headers on the main pcb of the system. If you have ever set up a pc engines board, you may have noticed a pin header labelled GPIO next to the serial connector. A large number of SBCs are now being specifically designed to export GPIO for use in hardware projects. There are Intel boards such as the LattePanda that run the familiar 64-bit (x86) platform. By and large, most of the SBCs designed for hardware projects run either 32-bit or 64-bit ARM architecture, such as the BeagleBone and Raspberry Pi families of computers.


Above: The BeagleBone Black and components we use in this article.

In this article, I am going to use the ARMv7 (32-bit) BeagleBone Black board. The GPIO and bus interface concepts are common to all platforms supported by FreeBSD and will transfer to different hardware quite easily. The BeagleBone Black is a nice board because it has very mature FreeBSD support and has a large number of IO ports exposed for experimentation.

To follow along with this article you will need:

- a BeagleBone Black SBC (a different board would work, but all the GPIO names will be different)
- a mini USB cable
- an ethernet cable to connect the BeagleBone Black to the Internet
- a small breadboard
- some jumper wires
- 3x resistors (something between 200 ohm and 400 ohm will be fine)
- 3x LEDS (1 red, 1 green, and 1 orange)

Setting up the BeagleBone Black

The latest FreeBSD release image (12 at the time of writing) for the BeagleBone Black can be downloaded from the project mirror. This must be decompressed and written to an SD card with dd:

```
$ fetch ftp://ftp.freebsd.org/pub/FreeBSD/releases/arm/armv7/ISO-IMAGES/\
      12.0/FreeBSD-12.0-RELEASE-arm-armv7-BEAGLEBONE.img.xz
$ xz -d FreeBSD-12.0-RELEASE-arm-armv7-BEAGLEBONE.img.xz
# dd if=FreeBSD-12.0-RELEASE-arm-armv7-BEAGLEBONE.img of=/dev/da0 bs=1m
```

The location your sd card mounts will vary; be careful with the dd command as it will overwrite the disc at which you point it without any protection.

Once you have prepared the SD card, you should connect the mini USB port of the BeagleBone Black to a USB port on your computer. This cable provides both power and the interface we are going to use to connect to the board. The

BeagleBone Black acts as a USB serial gadget, and through this interface the BeagleBone Black creates a virtual serial port. After connecting the board, look in dmesg for a new umodem device or in /dev for a /dev/ttyUX device (on the Mac OS, this device will probably be /dev/tty.usbmodemFreeBSD11). You should see a USB serial device, which can establish a connection from a FreeBSD system using the cu serial terminal in the base at a baud rate of 115200:

```
# cu -l /dev/ttyU0 -s 115200
```

To exit from cu, hit enter, type ~. (a tilde, then a full stop). Once connected over serial, you can get into the system with the default username and password (root/root or freebsd/freebsd).

GPIO with FreeBSD

Once in, you can have a look around and see that this is a normal FreeBSD install. If you search in dmesg, you will see that four GPIO controllers have connected and a GPIO bus (gpiobus) and control (gpioc) interface have been created for each.

```
gpio0: <TI AM335x General Purpose I/O (GPIO)> mem 0x44e07000-0x44e07fff irq
       7 on simplebus0
gpiobus0: <OFW GPIO bus> on gpio0
gpioc0: <GPIO controller> on gpio0
```

There can be multiple GPIO controllers on a system. FreeBSD exposes each of these controllers as a separate /dev/gpiocX device. On the BeagleBone Black, there are four buses.

Unlike Linux, where GPIO devices are exposed through /sys/class as files, FreeBSD has a command line tool (and a C library) for interacting with GPIO pins called gpioctl. Let's try using gpioctl to list some of the available pins on the BeagleBone Black.

```
# gpioctl -f /dev/gpioc1 -lv
...snip...
pin 19: 0    gpio_19<IN,PD>, caps:<IN,OUT,PU,PD,UNKNOWN,UNKNOWN,UNKNOWN,UNKNOWN,UNKNOWN>
pin 20: 0    gpio_20<IN,PD>, caps:<IN,OUT,PU,PD,UNKNOWN,UNKNOWN,UNKNOWN,UNKNOWN,UNKNOWN>
pin 21: 0    gpio_21<OUT>, caps:<IN,OUT,PU,PD,UNKNOWN,UNKNOWN,UNKNOWN,UNKNOWN,UNKNOWN>
pin 22: 0    gpio_22<OUT>, caps:<IN,OUT,PU,PD,UNKNOWN,UNKNOWN,UNKNOWN,UNKNOWN,UNKNOWN>
...snip...
```

gpioctl gives a list of all the pins the controller knows about. Each of the four GPIO controllers on the BeagleBone Black has 32 pins connected. gpioctl tells us several things about each pin. First, it tells us the number it uses to refer to the pin. Next, the current logical level of the pin. And the final column in the output has the hardware name of the pin, i.e., gpio_22 and the pins' flags. Some controllers have more descriptive names for pins taken from the SoCs datasheet.

gpioctl(8) lists the possible flags for a pin:

```
IN        Input pin
OUT       Output pin
OD        Open drain pin
PP        Push pull pin
TS        Tristate pin
PU        Pull-up pin
PD        Pull-down pin
II        Inverted input pin
IO        Inverted output pin
```

From our gpioctl output, we can see that pin 19 is configured as an input by default and it has a pull down configured, while pin 21 is configured as an output, but could be configured as either an input or an output and could have either a pull-up or pull-down resistor. A pull down is a resistor added between the pin and ground to keep the logical level of the pin output low unless it is driven upwards. This will help us later.

Pin 21 is configured as an output. If we look at the BeagleBone Black System Reference manual section 6.5, we see that the BeagleBone Black has four user-controllable LEDS called USR0-USR3 and they are connected to GPIO1 21, GPIO2 22, GPIO2 23, and GPIO2 24. LED USR0 is available for us to use, while the others are set aside for other tasks.

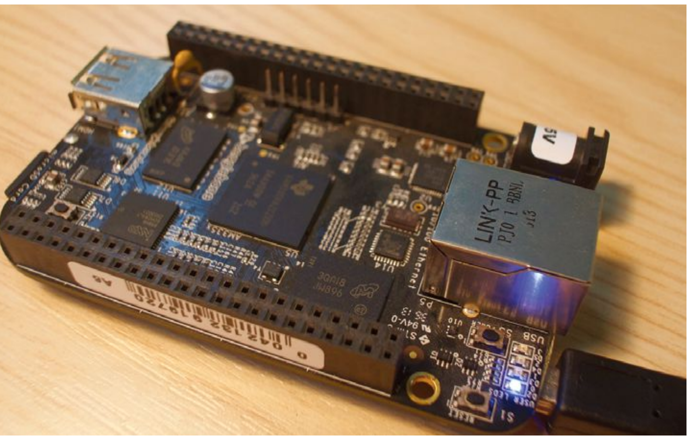
https://cdn-shop.adafruit.com/datasheets/BBB_SRM.pdf

LED	GPIO SIGNAL	PROC PIN
USR0	GPIO1_21	V15
USR1	GPIO2_22	U15
USR2	GPIO2_23	T15
USR3	GPIO2_24	V16

User LED Control Signals (Table 7 from the BBB SRM, CC-SA).
Table by Gerald Coley of BeagleBoard.org.
For more information, see http://creativecommons.org/license/results-one?license_code=by-sa.

In the gpioctl output, we can see that all four pins have a logical level of 0; the easiest way to find the LEDs on the board is to turn them on.

Let's do that.



The BeagleBone Black has four USR LEDs. In this case, USR0 is illuminated.

gpioctl can set the LED to a digital value of 0 or 1. Let's configure and turn on USR0:

```
# gpioctl -f /dev/gpioc1 21 OUT
# gpioctl -f /dev/gpioc1 21 1
```

Bask in the illumination of our glorious LED.

Further we can toggle an LED by using the -t flag to gpioctl. This can be helpful when you just want to blink the LED.

```
# gpioctl -f /dev/gpioc1 -t 21
# gpioctl -f /dev/gpioc1 -t 21
# gpioctl -f /dev/gpioc1 -t 21
```

Should cause the LED to turn on and off.

Using the built-in LEDs is handy for showing status information from the board and for experimenting with tools, but the GPIOs that back these LEDs are not broken out to the pin headers on the BeagleBone Black. Let's do the same thing again with external LEDs.

The BeagleBone Black has two sets of pin headers called P8 and P9. With the ethernet jack on the right, P8 is the bottom header and P9 is the top header. Most of P8 is assigned to LED pins (which can be reconfigured) and there are still several available GPIOs we can use.

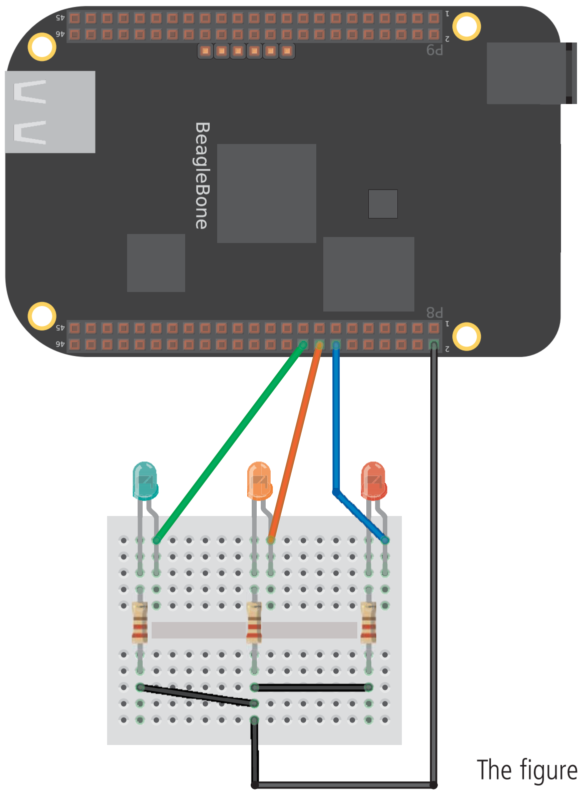
We are going to connect LEDs to pins 14, 16, and 18 on P8. From the System Reference Manual, we can find out the actual chip GPIO they are connected to:

PIN	PROC	NAME
14	T11	GPIO0_26
16	V13	GPIO1_14
18	V12	GPIO2_1

P8 connector pin out (derived from Table 10 of the BBB SRM, CC-SA).

We need to connect the pins to the long leg on the LED and we also need to use a resistor to limit the current through the LED. A resistor from around 200 ohm to 470 ohm should be suitable for this.

With the LEDs connected, we need to configure their GPIOs as outputs and then we can play with them:



The figure shows how we have connected these three LEDs to the BeagleBone Black.

```
# gpioctl -f /dev/gpioc0  -c 26 OUT      #  red
# gpioctl -f /dev/gpioc1  -c 14 OUT      #  orange
# gpioctl -f /dev/gpioc2  -c 1 OUT       #  green

# gpioctl -f /dev/gpioc0  26 1
# gpioctl -f /dev/gpioc1  14 1
# gpioctl -f /dev/gpioc2  1 1
```

You should now have all three LEDs turned on. If you used three colors as I did, then you should have a nice red, orange, and green ready to do something interesting.

A Hardware Project with FreeBSD

FreeBSD has continuous integration (CI) servers that build FreeBSD for different architectures to test commits as they are made to the tree. FreeBSD uses Jenkins for CI, and Jenkins offers a handy status API that tells us how a build is doing and whether it succeeded or failed.

Let's put together a build status indicator that tells us how the FreeBSD ARMv7 (the architecture of the BeagleBone Black) is doing. Jenkins exposes the build status through a json api, and textproc/jq provides a simple way to query json objects. Jenkins returns two fields concerning the in-progress build—'building' and 'result'. If 'building' is true, then there will not be a valid 'result'. 'result' can be 'SUCCESS' 'FAILURE' 'UNSTABLE' or 'none' depending on the 'building' status and the 'result' of the most recent build.

The scripts require the jq tool to parse the json from the CI server and we need a base set of ssl certificates to authenticate with the server. The BeagleBone Black doesn't have a real time clock—you will need to have accurate time to do ssl to the CI server.

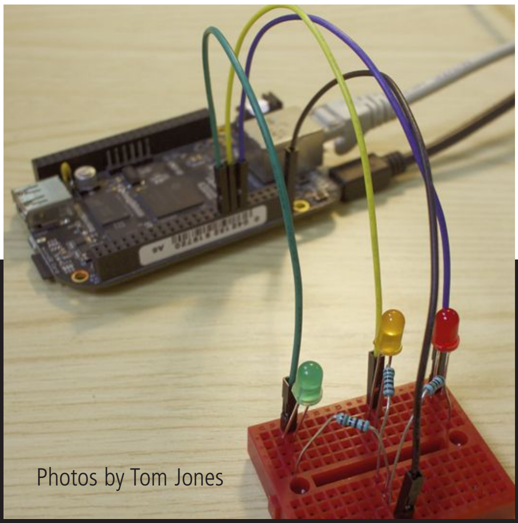
```
# pkg install jq ca_root_nss
# ntpdate -s pool.ntp.org
```

```
#!/bin/sh

project=FreeBSD-head-armv7-build

# configure the leds as outputs
gpioctl -f /dev/gpioc0 -c 26 OUT     # red
gpioctl -f /dev/gpioc1 -c 14 OUT     # orange
gpioctl -f /dev/gpioc2 -c 1 OUT      # green

# flash all the leds at start
gpioctl -f /dev/gpioc0 26 1
gpioctl -f /dev/gpioc1 14 1
gpioctl -f /dev/gpioc2 1 1

sleep 2

# check the build status once a minute
while true; do
    echo fetching $project
    output=`fetch -o - https://ci.freebsd.org/job/$project/
    lastBuild/api/json 2>/dev/null`

    building=`echo $output | jq ".building"`
    result=`echo $output | jq ".result"`

    # clear all leds
    gpioctl -f /dev/gpioc0 26 0
    gpioctl -f /dev/gpioc1 14 0
    gpioctl -f /dev/gpioc2 1 0

    if [ "$building" = "true" ]; then
      echo $project is building
      gpioctl -f /dev/gpioc1 14 1 # orange led

      for foo in `jot 60 1`;
      do
        gpioctl -f /dev/gpioc1 -t 14 # orange led
        sleep 0.5
      done
    else
      echo $project done building
      if [ "$result" = '"SUCCESS"' ]; then
        echo build suceeded for $project
        gpioctl -f /dev/gpioc2 1 1 #green led
      else
        echo build failed for $project
        gpioctl -f /dev/gpioc0 26 1 #red led
      fi
    fi

    sleep 60  # 1 minute
done
```

We can connect three LEDs to GPIO on the BeagleBone Black—a green, a red, and an orange one. The listing blinks the orange LED when the build is running, the green LED blinks when the build is passing, and the red LED blinks when the build fails.

Using LEDs is just an example. We could equally connect a relay to control a siren to the GPIO enabling the red LED to wake us up when the build starts to fail.•

BeagleBone Black connected to our three status LEDs.


Photos by Tom Jones

Tom Jones is a founder and director of a Hackerspace in Aberdeen, Scotland (57northhacklab.org.uk). He started hardware hacking on FreeBSD trying to port a project from Linux and got sucked into the world of kernel hacking.