Device Tree Layovers in BBB - Lab 6

Using BeagleBone Black GPIOs Device Tree

Reference: www.armhf.com/index.php/using-beaglebone-black-gpios/

To control digital input / outputs for the BeagleBone Black, you can use the facilities exposed by the kernel in the /sys/class/gpio directory. Note that the BeagleBone White pinouts are different from the BeagleBone Black. Also note that the GPIOs available on the BBW have changed between revisions, so be certain to get the proper technical reference manual for your actual board revision.
After boot, nothing is exported for use, but we do see the four GPIO controllers (gpio0, gpio1, gpio2, and gpio3):

```
# ls -al /sys/class/gpio
total 0
drwxr-xr-x  2 root root    0 May 26 15:40 .
drwxr-xr-x 45 root root    0 Jan  1  2000 ..
--w-------  1 root root 4096 May 26 15:48 export
lrwxrwxrwx  1 root root    0 May 26 15:45 gpiochip0 -> ../../devices/virtual/gpio/gpiochip0
lrwxrwxrwx  1 root root    0 May 26 15:45 gpiochip32 -> ../../devices/virtual/gpio/gpiochip32
lrwxrwxrwx  1 root root    0 May 26 15:45 gpiochip64 -> ../../devices/virtual/gpio/gpiochip64
lrwxrwxrwx  1 root root    0 May 26 15:45 gpiochip96 -> ../../devices/virtual/gpio/gpiochip96
--w-------  1 root root 4096 May 26 15:45 unexport
```

Note that each GPIO controller is offset by 32 from the previous (0, 32, 64, 96) — more on this later.
Not all GPIO pins are available by default. A lot of frustration may come from not knowing this — sometimes when it appears that all should be working, it does not. The microcontroller needs to be reconfigured to enable some of the pin modes. To understand what may be available, you need to study table 10 on page 70 for the P8 header and table 11 on page 72 for the P9 header in the BeagleBone Black System Reference Manual. Pay close attention to which header is P8 and which is P9 (page 68) because they feel backwards to me.
In this example, I will export P9 header GPIO pins 12, 14, 15, 16. Referencing page 72 mode 7 column, it shows pin 12 to be gpio1[28], pin 14 to be gpio1[18], pin 15 to be gpio1[16], pin 16 to be gpio1[19]. All of these pins are on the gpio1 controller (gpiochip32). To get to the GPIO number that should be exported from the kernel, we must add on 32 to each GPIO (64 for gpio2, and 96 for gpio3). Therefore gpio1[28] = 32 + 28 = 60. I will just do all the math here and we will refer to them by the kernel nomenclature and stop with the gpio1[x] business as it gets confusing.

```
pin 12 --> gpio60
pin 14 --> gpio50
pin 15 --> gpio48
pin 16 --> gpio51
```

Now that we have a relationship between the P9 header physical pins and what the kernel is calling them, we can begin configuration. To make this example easy to follow, I am just going to use bash scripts, but all of the same principles apply to the file manipulation APIs in your favorite language.
In this example, we will be configuring all four pins to be digital outputs. To do so, we need to copy the gpio number we want to export into the kernel gpiolib /sys/class/gpio/export file.

```
echo 48 > /sys/class/gpio/export
echo 50 > /sys/class/gpio/export
echo 51 > /sys/class/gpio/export
echo 60 > /sys/class/gpio/export
```

Afterward we can see there are four new subdirectories for each of the gpios:

```
# ls -al /sys/class/gpio
```

```
total 0
drwxr-xr-x 2 root root    0 May 26 15:40 ./
drwxr-xr-x 45 root root   0 Jan  1  2000 ../
--w------- 1 root root 4096 May 26 16:25 export
lrwxrwxrwx 1 root root    0 May 26 16:25 gpio48 -> ../../devices/virtual/gpio/gpio48/
lrwxrwxrwx 1 root root    0 May 26 16:25 gpio50 -> ../../devices/virtual/gpio/gpio50/
lrwxrwxrwx 1 root root    0 May 26 16:25 gpio51 -> ../../devices/virtual/gpio/gpio51/
lrwxrwxrwx 1 root root    0 May 26 16:25 gpio60 -> ../../devices/virtual/gpio/gpio60/
lrwxrwxrwx 1 root root    0 May 26 15:45 gpiochip0 -> ../../devices/virtual/gpio/gpiochip0/
lrwxrwxrwx 1 root root    0 May 26 15:45 gpiochip32 -> ../../devices/virtual/gpio/gpiochip32/
lrwxrwxrwx 1 root root    0 May 26 15:45 gpiochip64 -> ../../devices/virtual/gpio/gpiochip64/
lrwxrwxrwx 1 root root    0 May 26 15:45 gpiochip96 -> ../../devices/virtual/gpio/gpiochip96/
--w------- 1 root root 4096 May 26 15:45 unexport
```

And the contents of one of the gpio subdirectories:

```
# ls -alH /sys/class/gpio/gpio48
total 0
drwxr-xr-x  3 root root    0 May 26 16:25 .
drwxr-xr-x 10 root root    0 May 26 15:45 ..
-rw-r--r--  1 root root 4096 May 26 16:29 active_low
-rw-r--r--  1 root root 4096 May 26 16:29 direction
-rw-r--r--  1 root root 4096 May 26 16:29 edge
drwxr-xr-x  2 root root    0 May 26 16:29 power
lrwxrwxrwx  1 root root    0 May 26 16:25 subsystem -> ../../../../class/gpio
-rw-r--r--  1 root root 4096 May 26 16:25 uevent
-rw-r--r--  1 root root 4096 May 26 16:29 value
```

There are several configuration options available to each gpio. I suggest you read the Linux GPIO Interfaces manual for all of the details. I am only covering the basics here, and the Linux GPIO Interfaces manual is a very important read to understanding Linux gpio control.

Next we must specify if the exported gpio is input or output. It does not make sense to configure anything else ahead of this and as far as I know, the kernel doesn't let you do anything else with it until you set the gpio as input or output.

To set a gpio as output, we set its direction as in or out. For outputs there is an alternative nomenclature where output direction can be set instead as high or low to help with glitch free operation. I'll use this nomenclature:

```
echo high > /sys/class/gpio/gpio48/direction
echo high > /sys/class/gpio/gpio50/direction
echo high > /sys/class/gpio/gpio51/direction
echo high > /sys/class/gpio/gpio60/direction
```

Pins 12, 14, 15, 16 are now configured as outputs and are currently high — these pins will now be reading 3.3v. Do not use 5v TTL logic parts or you will damage the board.

Finally, we will set the pins low which will de-power them:

```
echo 0 > /sys/class/gpio/gpio48/value
echo 0 > /sys/class/gpio/gpio50/value
echo 0 > /sys/class/gpio/gpio51/value
echo 0 > /sys/class/gpio/gpio60/value
```

Putting this all together, this script will configure and run a binary counter that will overflow and start at zero again. It is kind of cool to watch if you hook relays up to the board.

```
#!/bin/bash -e

if [ ! -d /sys/class/gpio/gpio48 ]; then echo 48 > /sys/class/gpio/export; fi
if [ ! -d /sys/class/gpio/gpio50 ]; then echo 50 > /sys/class/gpio/export; fi
if [ ! -d /sys/class/gpio/gpio51 ]; then echo 51 > /sys/class/gpio/export; fi
```

```
if [ ! -d /sys/class/gpio/gpio60 ]; then echo 60 > /sys/class/gpio/export; fi

echo low > /sys/class/gpio/gpio48/direction
echo low > /sys/class/gpio/gpio50/direction
echo low > /sys/class/gpio/gpio51/direction
echo low > /sys/class/gpio/gpio60/direction

for (( i=0 ; ; ++i ))
do
  if (( i > 0x0f )); then
    i=0
    printf '\n[press  + c to stop]\n\n'
  fi

  bit0=$(( (i & 0x01) > 0 ))
  bit1=$(( (i & 0x02) > 0 ))
  bit2=$(( (i & 0x04) > 0 ))
  bit3=$(( (i & 0x08) > 0 ))
  echo $bit3 $bit2 $bit1 $bit0

  echo $bit0 > /sys/class/gpio/gpio60/value
  echo $bit1 > /sys/class/gpio/gpio50/value
  echo $bit2 > /sys/class/gpio/gpio48/value
  echo $bit3 > /sys/class/gpio/gpio51/value

  sleep .2
done
```

Save the above code in text file called GPIO_LEDs. And execute the shell script to see the LEDs on the board blinking:
> ./GPIO_LEDs

REFERENCE: http://learn.adafruit.com/introduction-to-the-beaglebone-black-device-tree/overview

Beaglebone Black Device Tree: UART

The Device Tree (DT), and Device Tree Overlay are a way to describe hardware in a system. An example of this would be to describe how the UART interfaces with the system, which pins, how they should be muxed, the device to enable, and which driver to use.

The original BeagleBone didn't use the DT, but the recently released BeagleBone Black was released with the DT and is now using the 3.8 Linux Kernel.

Below is the device tree overlay for the UART1 device. It tells the kernel everything it needs to know in order to properly enable UART1 on pins P9_24 and P9_26.

BB-UART1-00A0.dts

```
/*
 * Copyright (C) 2013 CircuitCo
 *
 * Virtual cape for UART1 on connector pins P9.24 P9.26
 *
 * 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 = "BB-UART1";
       version = "00A0";

       /* state the resources this cape uses */
       exclusive-use =
              /* the pin header uses */
              "P9.24",        /* uart1_txd */
              "P9.26",        /* uart1_rxd */
              /* the hardware ip uses */
              "uart1";

       fragment@0 {
              target = <&am33xx_pinmux>;
              __overlay__ {
                     bb_uart1_pins: pinmux_bb_uart1_pins {
                            pinctrl-single,pins = <
                                   0x184 0x20 /* P9.24 uart1_txd.uart1_txd MODE0 OUTPUT
(TX) */
                                   0x180 0x20 /* P9.26 uart1_rxd.uart1_rxd MODE0 INPUT (RX)
*/
                            >;
                     };
              };
       };

         fragment@1 {
              target = <&uart2>;        /* really uart1 */
              __overlay__ {
                     status = "okay";
                     pinctrl-names = "default";
```

```
                pinctrl-0 = <&bb_uart1_pins>;
            };
        };
};
```

In this case, it's P9_24 as MODE0, which will be an OUTPUT (TX). As well as P9_26 as MODE0 an INPUT (RX).

Exporting and Unexporting an Overlay

To start with, navigate into /lib/firmware to view all of the device tree overlays available by default.

```
root@beaglebone:~# cd /lib/firmware
root@beaglebone:/lib/firmware# ls -ltr
total 888
...
```

You'll see quite a few overlays available. The source (dts), and compiled (dtbo) files are both in that directory.

Let's use the BB-UART1-00A0.dts that we walked through on the previous page.

We could view the .dts file, but we already know what it contains. Let's take advantage of the fact that there's already a compiled .dtbo file available to us. We'll build and compile one of our own in a future section.

Ok, let's next navigate to where we can view which overlays are enabled by the bone cape manager:

```
root@beaglebone:/lib/firmware# cd /sys/devices/bone_capemgr.*
```

Note above, the * is necessary as we don't know what that number can be. It depends on the boot order. I've seen the path as /sys/devices/bone_capemgr.8/slots, as well as /sys/devices/bone_capemgr.9/slots. You'll need to determine what your particular path is.

Next up, cat the contents of the slots file:

```
root@beaglebone:/sys/devices/bone_capemgr.8# cat slots
```

It should look something like this, assuming you haven't customized your Angstrom installation very much:

```
0: 54:PF---
1: 55:PF---
2: 56:PF---
3: 57:PF---
4: ff:P-O-L Bone-LT-eMMC-2G,00A0,Texas Instrument,BB-BONE-EMMC-2G
5: ff:P-O-L Bone-Black-HDMI,00A0,Texas Instrument,BB-BONELT-HDMI
```

According to the BeagleBone documentation, "the first 3 slots are assigned by EEPROM IDs on the capes". The next two are overlays loaded at boot. Number 4 is the EMMC memory built in that you're mostly likely booting your Angstrom distribution from. The 5th overlay is for enabling the HDMI component.

If you were to now export another overlay, such as our favorite UART1 overlay, you

would see a new option listed as number 6. Let's try that by exporting the UART1 dtbo file:
root@beaglebone:/sys/devices/bone_capemgr.8# root@beaglebone:/sys/devices/bone_capemgr.8# echo BB-UART1 > slots
We're taking the output of echo, "BB-UART1", and writing it to the slots file to enable the drivers and device for UART1 using the overlay. Now, let's check that the overlay loaded properly:
root@beaglebone:/sys/devices/bone_capemgr.8# cat slots
We should now have the UART1 device loaded up, and ready to go:
0: 54:PF---
1: 55:PF---
2: 56:PF---
3: 57:PF---
4: ff:P-O-L Bone-LT-eMMC-2G,00A0,Texas Instrument,BB-BONE-EMMC-2G
5: ff:P-O-L Bone-Black-HDMI,00A0,Texas Instrument,BB-BONELT-HDMI
6: ff:P-O-L Override Board Name,00A0,Override Manuf,BB-UART1
Now, let's say you're done with using the UART1 device, and need those pins for something else. One way to remove the overlay would be to restart your BeagleBone. The other way would be to unexport it.

You can export it by executing the following command:
root@beaglebone:/sys/devices/bone_capemgr.8# echo -6 > slots
We took the 6th listed option, and put a '-' before it, and wrote it to the slots file.

One thing to note. As of the 6-20-2013 release of Angstrom, unloading various overlays can cause a kernel panic, and cause you to lose your ssh session, along with making the capemgr unpredictable. It's recommended to just restart your system to unload overlays until that issue is resolved.

Now that we know that restarting the system will cause the overlays to unload, how do we have them loaded when the system boots up?

This is fairly simple to do. All you need to do is reference them in the uEnv.txt in the small FAT partition on your BeagleBone Black.

The following steps illustrate how to do this for the UART1 overlay:
mkdir /mnt/boot
mount /dev/mmcblk0p1 /mnt/boot
nano /mnt/boot/uEnv.txt
#add this to the end of the single line of uEnv.txt:
capemgr.enable_partno=BB-UART1

To start with we're using the device tree compiler (dtc). Everything required to compile DT overlays are included with the latest Angstrom distribution.

wget -c https://raw.github.com/RobertCNelson/tools/master/pkgs/dtc.sh
chmod +x dtc.sh
./dtc.sh

Next, we'll execute the command to compile this file into the device tree overlay compiled format (.dtbo):

dtc -O dtb -o BB-UART1-00A0.dtbo -b 0 -@ BB-UART1-00A0.dts

Beaglebone Black Device Tree: SPI

SPI0 pins
Chip Select - CS0 - pin 17
Clock - SCLK - pin 22
MOSI - D0 - pin 21
MISO - D1 - pin 18

Navigate to your home directory, and open nano to copy and paste the new file. You'll need to save it exactly as named below as well:
root@beaglebone:/tmp# cd ~
root@beaglebone:~# nano ADAFRUIT-SPI0-00A0.dts

Below is a copy of the overlay for SPI0 that we'll use to compile, and enable:

```
/dts-v1/;
/plugin/;

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

   /* identification */
   part-number = "ADAFRUIT-SPI0";

   /* version */
   version = "00A0";

   fragment@0 {
      target = <&am33xx_pinmux>;
      __overlay__ {
         spi0_pins_s0: spi0_pins_s0 {
```

```
            pinctrl-single,pins = <
              0x150 0x30 /* spi0_sclk, INPUT_PULLUP | MODE0 */
              0x154 0x30 /* spi0_d0, INPUT_PULLUP | MODE0 */
              0x158 0x10 /* spi0_d1, OUTPUT_PULLUP | MODE0 */
              0x15c 0x10 /* spi0_cs0, OUTPUT_PULLUP | MODE0 */
            >;
        };
    };
};

fragment@1 {
    target = <&spi0>;
    __overlay__ {
        #address-cells = <1>;
        #size-cells = <0>;

        status = "okay";
        pinctrl-names = "default";
        pinctrl-0 = <&spi0_pins_s0>;

        spidev@0 {
            spi-max-frequency = <24000000>;
            reg = <0>;
            compatible = "spidev";
        };
        spidev@1 {
            spi-max-frequency = <24000000>;
            reg = <1>;
            compatible = "spidev";
        };
    };
};
};
```

Next, we'll execute the command to compile this file into the device tree overlay compiled format (.dtbo):

```
dtc -O dtb -o ADAFRUIT-SPI0-00A0.dtbo -b 0 -@ ADAFRUIT-SPI0-00A0.dts
```

The compilation should be nearly instant, and you should end up with the newly compiled file:

```
root@beaglebone:~# ls -ltr
...
-rw-r--r-- 1 root root 1255 Jul 29 14:33 ADAFRUIT-SPI0-00A0.dts
-rw-r--r-- 1 root root 1042 Jul 29 14:35 ADAFRUIT-SPI0-00A0.dtbo
```

Then, copy the file into /lib/firmware/:

```
cp ADAFRUIT-SPI0-00A0.dtbo /lib/firmware/
```

Then enable the device tree overlay:

```
echo ADAFRUIT-SPI0 > /sys/devices/bone_capemgr.*/slots
```

Plug in your BBB to a host computer using the mini usb data cable.

Go to My Computer>BeagleBone Getting Started> and open uEnv.txt Copy and paste this command into the .txt file. Make sure to save your changes. (Ctrl+s)

```
optargs=quiet drm.debug=7 capemgr.disable_partno=BB-
BONELT-HDMI,BB-BONELT-HDMIN
capemgr.enable_partno=BB-SPI1-01
```

after you save the changes, reboot your beaglebone black
Make sure it is enabled You should now have two spidev-files in the folder /dev/

```
ls -al /dev/spidev1.*
```

You should also be able to see the pingroups:

```
cat /sys/kernel/debug/pinctrl/44e10800.pinmux/pingroups
```

Now try a test with your new overlay. Create the following file.

```
nano spi_test.c
```

And paste and save the following code. (Ctrl-o to save, Ctrl-x to exit nano). The test code is a slightly modified version of https://www.kernel.org/doc/Documentation/spi/spidev_test.c. Some macros and struct variables were removed because of compilation errors. There is some disparity between the spidev library on the board and the online documentation.

```
/*
 * SPI testing utility (using spidev driver)
 *
 * Copyright (c) 2007 MontaVista Software, Inc.
 * Copyright (c) 2007 Anton Vorontsov
<avorontsov@ru.mvista.com>
 *
```

```
  * This program is free software; you can redistribute it and/
or modify
  * it under the terms of the GNU General Public License as
published by
  * the Free Software Foundation; either version 2 of the
License.
  *
  * Cross-compile with cross-gcc -I/path/to/cross-kernel/include
  */
 #include <stdint.h>
 #include <unistd.h>
 #include <stdio.h>
 #include <stdlib.h>
 #include <getopt.h>
 #include <fcntl.h>
 #include <sys/ioctl.h>
 #include <linux/types.h>
 #include <linux/spi/spidev.h>
 #define ARRAY_SIZE(a) (sizeof(a) / sizeof((a)[0]))
 static void pabort(const char *s)
 {
      perror(s);
      abort();
 }
 static const char *device = "/dev/spidev1.0";
 static uint32_t mode;
 static uint8_t bits = 8;
 static uint32_t speed = 500000;
 static uint16_t delay;
 static void transfer(int fd)
 {
      int ret;
```

```
	uint8_t tx[] = {
		0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
		0x40, 0x00, 0x00, 0x00, 0x00, 0x95,
		0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
		0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
		0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
		0xDE, 0xAD, 0xBE, 0xEF, 0xBA, 0xAD,
		0xF0, 0x0D,
	};
	uint8_t rx[ARRAY_SIZE(tx)] = {0, };
	struct spi_ioc_transfer tr = {
		.tx_buf = (unsigned long)tx,
		.rx_buf = (unsigned long)rx,
		.len = ARRAY_SIZE(tx),
		.delay_usecs = delay,
		.speed_hz = speed,
		.bits_per_word = bits,
	};
	if (mode & SPI_TX_QUAD)
		tr.tx_nbits = 4;
	else if (mode & SPI_TX_DUAL)
		tr.tx_nbits = 2;
	if (mode & SPI_RX_QUAD)
		tr.rx_nbits = 4;
	else if (mode & SPI_RX_DUAL)
		tr.rx_nbits = 2;
	if (!(mode & SPI_LOOP)) {
		if (mode & (SPI_TX_QUAD | SPI_TX_DUAL))
			tr.rx_buf = 0;
		else if (mode & (SPI_RX_QUAD | SPI_RX_DUAL))
			tr.tx_buf = 0;
	}
```

```
     ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
     if (ret < 1)
          pabort("can't send spi message");
     for (ret = 0; ret < ARRAY_SIZE(tx); ret++) {
          if (!(ret % 6))
               puts("");
          printf("%.2X ", rx[ret]);
     }
     puts("");
 }
 static void print_usage(const char *prog)
 {
     printf("Usage: %s [-DsbdlHOLC3]\n", prog);
     puts(" -D --device  device to use (default /dev/
spidev1.1)\n"
          " -s --speed  max speed (Hz)\n"
          " -d --delay  delay (usec)\n"
          " -b --bpw   bits per word \n"
          " -l --loop   loopback\n"
          " -H --cpha   clock phase\n"
          " -O --cpol   clock polarity\n"
          " -L --lsb   least significant bit first\n"
          " -C --cs-high chip select active high\n"
          " -3 --3wire  SI/SO signals shared\n"
          " -N --no-cs  no chip select\n"
          " -R --ready  slave pulls low to pause\n"
          " -2 --dual   dual transfer\n"
          " -4 --quad   quad transfer\n");
     exit(1);
 }
 static void parse_opts(int argc, char *argv[])
 {
```

```
      while (1) {
           static const struct option lopts[] = {
                { "device", 1, 0, 'D' },
                { "speed",  1, 0, 's' },
                { "delay",  1, 0, 'd' },
                { "bpw",    1, 0, 'b' },
                { "loop",   0, 0, 'l' },
                { "cpha",   0, 0, 'H' },
                { "cpol",   0, 0, 'O' },
                { "lsb",    0, 0, 'L' },
                { "cs-high", 0, 0, 'C' },
                { "3wire",  0, 0, '3' },
                { "no-cs",  0, 0, 'N' },
                { "ready",  0, 0, 'R' },
                { "dual",   0, 0, '2' },
                { "quad",   0, 0, '4' },
                { NULL, 0, 0, 0 },
           };
           int c;
           c = getopt_long(argc, argv, "D:s:d:b:lHOLC3NR24",
lopts, NULL);
           if (c == -1)
                break;
           switch (c) {
           case 'D':
                device = optarg;
                break;
           case 's':
                speed = atoi(optarg);
                break;
           case 'd':
                delay = atoi(optarg);
```

```
            break;
        case 'b':
            bits = atoi(optarg);
            break;
        case 'l':
            mode |= SPI_LOOP;
            break;
        case 'H':
            mode |= SPI_CPHA;
            break;
        case 'O':
            mode |= SPI_CPOL;
            break;
        case 'L':
            mode |= SPI_LSB_FIRST;
            break;
        case 'C':
            mode |= SPI_CS_HIGH;
            break;
        case '3':
            mode |= SPI_3WIRE;
            break;
        case 'N':
            mode |= SPI_NO_CS;
            break;
        case 'R':
            mode |= SPI_READY;
            break;
        case '2':
            mode |= SPI_TX_DUAL;
            break;
        case '4':
```

```
                mode |= SPI_TX_QUAD;
                break;
          default:
                print_usage(argv[0]);
                break;
          }
     }
     if (mode & SPI_LOOP) {
          if (mode & SPI_TX_DUAL)
                mode |= SPI_RX_DUAL;
          if (mode & SPI_TX_QUAD)
                mode |= SPI_RX_QUAD;
     }
}
int main(int argc, char *argv[])
{
     int ret = 0;
     int fd;
     parse_opts(argc, argv);
     fd = open(device, O_RDWR);
     if (fd < 0)
          pabort("can't open device");
     /*
      * spi mode
      */
     ret = ioctl(fd, SPI_IOC_WR_MODE32, &mode);
     if (ret == -1)
          pabort("can't set spi mode");
     ret = ioctl(fd, SPI_IOC_RD_MODE32, &mode);
     if (ret == -1)
          pabort("can't get spi mode");
     /*
```

```
	 * bits per word
	 */
	ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
	if (ret == -1)
		pabort("can't set bits per word");
	ret = ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits);
	if (ret == -1)
		pabort("can't get bits per word");
	/*
	 * max speed hz
	 */
	ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
	if (ret == -1)
		pabort("can't set max speed hz");
	ret = ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed);
	if (ret == -1)
		pabort("can't get max speed hz");
	printf("spi mode: 0x%x\n", mode);
	printf("bits per word: %d\n", bits);
	printf("max speed: %d Hz (%d KHz)\n", speed, speed/1000);
	transfer(fd);
	close(fd);
	return ret;
}
```

Connect pin 21 and pin 18 on P9. Compile and run the code.

```
gcc -Wall spi_test.c -o spi_test.o
```

```
./spi_test.o
spi mode: 0x0
bits per word: 8
```

```
max speed: 500000 Hz (500 KHz)
FF FF FF FF FF FF
40 00 00 00 00 95
FF FF FF FF FF FF
FF FF FF FF FF FF
FF FF FF FF FF FF
DE AD BE EF BA AD
F0 0D
```

Beaglebone Black Device Tree: I2C

I2C header pins P9.19 and P9.20 must be appropriately muxed using a Device Tree Overlay.
First, create the file "/lib/firmware/BB-I2C2-00A0.dts" with the following content (credit goes to Stephen Wolfe):

```
/dts-v1/;
/plugin/;

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

        /* identification */
        part-number = "BB-I2C2";
        version = "00A0";

        /* state the resources this cape uses */
        exclusive-use =
                /* the pin header uses */
                "P9.20",        /* i2c2_sda */
                "P9.19",        /* i2c2_scl */
                /* the hardware ip uses */
                "i2c2";

        fragment@0 {
                target = <&am33xx_pinmux>;
                __overlay__ {
```

```
                        bb_i2c2_pins: pinmux_bb_i2c2_pins {
                                pinctrl-single,pins = <
                                        0x178 0x73    //
spi0_d1.i2c2_sda,  SLEWCTRL_SLOW | IMPUT_PULLUP | MODE3
                                        0x17c 0x73    //
spi0_cs0.i2c2_scl, SLEWCTRL_SLOW | INPUT_PULLUP | MODE3
                                >;
                        };
                };
        };

        fragment@1 {
                target = <&i2c2>;    /* i2c2 is numbered
correctly */
                __overlay__ {
                        status = "okay";
                        pinctrl-names = "default";
                        pinctrl-0 = <&bb_i2c2_pins>;

                        /* this is the configuration part
*/
                        clock-frequency = <100000>;

                        #address-cells = <1>;
                        #size-cells = <0>;
                };
        };
};
```

Then compile the overlay using the following command:

```
#  dtc -O dtb -o /lib/firmware/BB-I2C2-00A0.dtbo -b 0 -@ /
lib/firmware/BB-I2C2-00A0.dts
```

Next, export the overlay:

```
# cd /sys/devices/bone_capemgr.*
# echo "BB-I2C2" >slots
```

The new overlay should now be available:

```
# cat slots
 0: 54:PF---
 1: 55:PF---
 2: 56:PF---
 3: 57:PF---
```

```
 4: ff:P-O-L Bone-LT-eMMC-2G,00A0,Texas Instrument,BB-BONE-
EMMC-2G
 5: ff:P-O-L Bone-Black-HDMI,00A0,Texas Instrument,BB-
BONELT-HDMI
 9: ff:P-O-L Override Board Name,00A0,Override Manuf,BB-
I2C2
```

The I2C2 interface can then be accessed as before via the "/dev/i2c-1" device file.