I2C Guide: 14-Seg Display

by Brian Fraser
Last update: June 8, 2018

This document guides the user through:

1. Understanding I2C
2. Using I2C from Linux command-line to drive the Zen cape's 14-seg display.
3. C code to access I2C and drive the display.

Target kernels v4.x

Table of Contents

1. I2C Basics....2
2. I2C via Linux Command Line....3
   2.1 Removing Conflicting “univ” Cape....3
   2.2 Enable the Bus....4
   2.3 Working with a Device....5
   2.4 Steps for 2-Digit 14-Seg Display....7
3. I2C via C Code....8
   3.1 Initialization....8
   3.2 Writing a Register....8
   3.3 Reading a Register....9
   3.4 Main program....9
4. Driving Individual Digits....10

Formatting:

1. Host (desktop) commands starting with `$` are Linux console commands:
   ```
   $ echo "Hello world"
   ```
2. Target (board) commands start with #:
   ```
   # echo "On embedded board"
   ```
3. Almost all commands are case sensitive.

Revision History:

- Sept 20: Initial version published. Revised mapping of I2C1 to /dev/i2c-1; added section on disabling cape_universal.
- Sept 30: Corrected typo in section 2 about which I2C port the GPIO extender is on.
- June 8, 2018: Corrected `cape_universal` from ‘`enabled`’ to ‘`enable`’ (likewise for disable)

1. I2C Basics

The I^2C (Inter-Integrated-Circuit, pronounced “I squared C”, and often written I2C) protocol is for synchronously communicating between a master and slave devices using two pins: a data (SDA) and a clock (SCL). Often the microprocessor is the master device which controls communication with one or more slave devices on the bus.

On the BeagleBone, the hardware supports three I2C buses, which are numbered 0 through 2.

HW Bus	**Linux Device**	**Default Use**	**Default Status**	**Zen Cape Use**
I2C0	/dev/i2c-0	HDMI (Internal to BeagleBone, not exposed on P8 or P9 headers)	Enabled	
I2C1	/dev/i2c-1[1]		Disabled	14-Seg Display via GPIO extender (address 0x20), Accelerometer (address 0x1C)
I2C2	/dev/i2c-2	Each cape has an EEPROM to identify it to the BeagleBone.	Enabled	EEPROM (address 0x54), Audio codec (address x018)

Each chip connected to an I2C bus has a unique address which is hard-wired into the chip. (Sometimes the hardware designer can select one of a few possible addresses for a chip.) In the simplest case, when the master wants to initiate a read or write to a device, it communicates over the appropriate I2C bus and indicates the address of the device it wishes to interact with.

Each device exposes a set of registers in a small address space. Each register has a special purpose. For example, the register at address 0x14 on the I2C GPIO extender device stores the lower 8-bits it will drive out on its GPIO pins.

Note that there three things one must specify when interacting with a device:

1. Which bus a device is on (hard-wired).
2. What I2C address that device has (hard-wired).
3. What register address to read/write from (from data-sheet).

1 Earlier versions of the BeagleBone image mapped HW I2C1 to /dev/i2c-2, and HW I2C2 to /dev/i2c-1.

2. I2C via Linux Command Line[2]

This guide uses walks through controlling a 2-digit 14-segment display. This display requires 15 GPIO pins (plus two more to control which digit, the left or the right, is on). This is quite a few GPIO pins, and since there are not enough free GPIO pins on the BeagleBone coming directly from the microprocessor, a GPIO-extender chip is connected to the microprocessor's I2C bus (`/dev/i2c-1`). The GPIO-extender used on the Zen cape is a MPC23017 “16-Bit I/O Expander with Serial Interface”. By controlling this I2C device, we can therefore drive the display's segments.

The GPIO extender's 16 bits of output are divided into two 8-bit ports.

2.1 Removing Conflicting “univ” Cape

1. If you have the `univ-emmc` virtual cape loaded then you won’t be able to enable I2C. You can see what capes are loaded as follows (highlighting the offending cape):
   ```
   # cat /sys/devices/platform/bone_capemgr/slots
    0: PF----  -1
    1: PF----  -1
    2: PF----  -1
    3: PF----  -1
    4: P-O-L-   0 Override Board Name,00A0,Override Manuf,univ-emmc
   ```
2. Check if the “universal” cape is enabled:
   ```
   # cat /proc/cmdline
   console=... coherent_pool=1M quiet cape_universal=enable
   ```
   - If it has `cape_universal=enable`, then keep following these steps.
     If it has `cape_universal=disable` (on not listed at all), then skip to the next section.
3. Create a backup copy of the `uEnv.txt` file before editing it.
   ```
   # cp /boot/uEnv.txt /boot/uEnv.bak_univ
   ```
   - Warning: Corrupting the `uEnv.txt` file can cause your BeagleBone to not boot!
4. Deactivate the `cape_universal` option on the `cmdline`:
   - Edit `/boot/uEnv.txt`:
     ```
     # nano /boot/uEnv.txt
     ```
   - Find the `cmdline=` line; change it to:
     ```
     cmdline=coherent_pool=1M quiet cape_universal=disable
     ```
   - Save the file and exit the editor (Ctrl+X).
5. Reboot the BeagleBone:
   ```
   # reboot
   ```
6. Ensure that the `cape_universal` option is disabled:
   ```
   # cat /proc/cmdline
   console=... coherent_pool=1M quiet cape_universal=disable
   ```
7. Double check that the `univ-emmc` virtual cape is no longer being loaded:
   ```
   # cat /sys/devices/platform/bone_capemgr/slots
    0: PF----  -1
    1: PF----  -1
    2: PF----  -1
    3: PF----  -1
   ```

2 Steps referenced from Exploring BeagleBone by Derek Molloy, 2015, chapter 8.

2.2 Enable the Bus

All I2C buses are controlled through the Linux kernel. First we must tell Linux that the hardware I2C bus is going to be used, if not already enabled.

1. Install the I2C tools:
   ```
   # sudo apt-get install i2c-tools
   ```
   - If not already installed, you'll need to ensure your device has internet access.
2. Determine which I2C bus the part is on.
   - Check the hardware schematic (or above tables) to determine which device you are accessing. Note the Linux Device and address.
   - If wiring in a different I2C devices, the BeagleBone's P9 expansion headers allow easy access to two hardware I2C buses: I2C1 and I2C2. (I2C0 is internal to the BeagleBone.)
     - Hardware bus I2C1 has SDA on P9_18, and SCL on P9_17.
     - Hardware bus I2C2 has SDA on P9_20, and SCL on P9_19.
3. Display which I2C buses Linux currently has enabled:
   ```
   # i2cdetect -l
   i2c-0 i2c         OMAP I2C adapter                    I2C adapter
   i2c-2 i2c         OMAP I2C adapter                    I2C adapter
   ```
4. If your device is on hardware bus I2C1 you must first enable Linux support for the bus (`/dev/i2c-1`) by turning on the `BB-I2C1` virtual cape.
   ```
   # echo BB-I2C1 > /sys/devices/platform/bone_capemgr/slots
   ```
   - Re-list the Linux I2C buses:
     ```
     # i2cdetect -l
     i2c-0    i2c         OMAP I2C adapter                    I2C adapter
     i2c-1    i2c         OMAP I2C adapter                    I2C adapter
     i2c-2    i2c         OMAP I2C adapter                    I2C adapter
     ```
5. Display I2C devices on the chosen I2C bus:
   ```
   # i2cdetect -y -r 1
   ```
   - Where `1` refers to the Linux device `/dev/i2c-1`
   - Sample output:
     ```
     # i2cdetect -y -r 1
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
     00:          -- -- -- -- -- -- -- -- -- -- -- -- --
     10: -- -- -- -- -- -- -- -- -- -- -- -- 1c -- -- --
     20: 20 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
     30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
     40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
     50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
     60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
     70: -- -- -- -- -- -- -- --
     ```
     "--" means no device found.
     "##" means device at address ## detected (hex).

     "UU" on `/dev/i2c-2` (HW I2C2) means in use by a driver (cape manager, HDMI, or another kernel device driver).
6. Troubleshooting
   - When trying to load the BB-I2C1 cape, if you get the error:
     ```
     # echo BB-I2C1 > /sys/devices/platform/bone_capemgr/slots
     -bash: echo: write error: File exists
     ```
     And dmesg shows

```
bone_capemgr bone_capemgr: slot #5: 'Override Board Name,00A0,Override Manuf,BB-I2C1'
bone_capemgr bone_capemgr: slot #5: BB-I2C1 conflict P9.18 (#4:BB-I2C1)
bone_capemgr bone_capemgr: slot #5: Failed verification
```

Then you already have the cape loaded; no need to reload it, it should all be working. Trying to load a second time will fail, but will not cause any problems.

- When trying to load the BB-I2C1 cape, if you get the error:
  ```
  # echo BB-I2C1 > /sys/devices/platform/bone_capemgr/slots
  -bash: echo: write error: File exists
  ```
  And `dmesg` shows:
  ```
  bone_capemgr bone_capemgr: slot #5: 'Override Board Name,00A0,Override Manuf,BB-I2C1'
  bone_capemgr bone_capemgr: slot #5: BB-I2C1 conflict P9.18 (#4:univ-emmc)
  bone_capemgr bone_capemgr: slot #5: Failed verification
  ```
  Then it is likely that the `cape_universal` option is enabled on the command line. See the previous section for directions on disabling it.
- If you try to view devices on I2C-1 and it goes very slowly, and possibly missing some devices, then disable the universal cape and manually load the BB-I2C1 cape.
- If you are unable to work with I2C registers, there could be an issue with the initialization. This can be detected by the following command taking a long time to scan each location (should normally be virtually instantaneously):
  ```
  # i2cdetect -y -r 1
  ```
  - This has error has come up when initializing many pieces of hardware in a script. Solution is to ensure that I2C is loaded via $SLOTS before PWM.
- If you are using an older version of the BeagleBone, the cape manager was located at:
  `/sys/devices/bone_capemgr.9/slots`

2.3 Working with a Device

1. Display the internal memory of an I2C device:
   ```
   # i2cdump -y 1 0x20
   No size specified (using byte-data access)
        0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
   00: ff ff 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
   10: 00 00 ff fe 00 00 00 00 00 00 00 00 00 00 00 00    ...?............
   20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
   ```
   <… omitted...>
   - This shows the internal memory for the device at address 0x20 (GPIO extender) on `/dev/i2c-1` (HW I2C1). Output may differ for you.
   - Consult the data sheet for your I2C device to identify what each register address means.
   - You can also read a single byte of memory, if desired:
     ```
     # i2cget -y 1 0x20 0x00
     0xff
     ```
     - `-y 1`: I2C bus `/dev/i2c-1`
     - `0x20`: Address of device on bus
     - `0x00`: Register address to read.
2. Write to the I2C device using `i2cset` command:
   ```
   # i2cset -y 1 0x20 0x00 0x00
   # i2cset -y 1 0x20 0x01 0x00
   ```
   - These commands control device with address 0x20 on `/dev/i2c-1`: to register 0x00 it writes 0x00, and to register 0x01 it writes 0x00.
   - Doing this on the I2C GPIO extender sets the device to be output on all pins. You won't yet

see any output, though; see the next section for details.

3. Display device internal memory:

```
# i2cdump -y 1 0x20
No size specified (using byte-data access)
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
10: 00 00 ff fe 00 00 00 00 00 00 00 00 00 00 00 00    ...?............
20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
```

<… omitted...>

- Output may differ; expect to see the first two values (at 0x00 and 0x01) both set to 0 now.

2.4 Steps for 2-Digit 14-Seg Display

Here are the concise steps to turning on the 14-seg display:

1. Since there are two digits, each with 15 individual segments to activate, this would require 30 separate GPIO pins to individually drive all segments! We save half of the GPIO pins by instead having an enable pin for each of the digits.
   - We individually drive the two digit-enable GPIO pins directly from the microprocessor. On the Zen cape, these are wired to P8_26 and P8_27, corresponding to Linux pin numbers 61 and 44 respectively. Turning number 61 high activates the left digit, 44 activates the right digit.
   - Configure both pins on the microprocessor for output through GPIO (see other guide):
     ```
     echo 61 > /sys/class/gpio/export
     echo 44 > /sys/class/gpio/export
     echo out > /sys/class/gpio/gpio61/direction
     echo out > /sys/class/gpio/gpio44/direction
     ```
   - Drive a 1 to the GPIO pin to turn on the digit. The following turns on both digits.
     ```
     echo 1 > /sys/class/gpio/gpio61/value
     echo 1 > /sys/class/gpio/gpio44/value
     ```
2. Enable `/dev/i2c-1` (hardware bus I2C1):
   ```
   # echo BB-I2C1 > /sys/devices/platform/bone_capemgr/slots
   ```
3. Set direction of both 8-bit ports on I2C GPIO extender to be outputs:
   ```
   # i2cset -y 1 0x20 0x00 0x00
   # i2cset -y 1 0x20 0x01 0x00
   ```
4. Turn on segments of the display to show a '*'. The 2-digit 14-segment display uses 15-GPIO pins to drive each segment of the display (14 segments plus a decimal point). Driving a 1 turns it on the segment, 0 turns it off. Writing to registers 0x14 and 0x15 drive the GPIO pins:
   - Drive lower 8-bits (register 0x14) (0xFF for all segments on, some may be unused)
     ```
     # i2cset -y 1 0x20 0x14 0x1E
     ```
   - Drive upper 8-bits (register 0x15) (0xFF for all segments on, some may be unused)
     ```
     # i2cset -y 1 0x20 0x15 0x78
     ```
   - Exact bit pattern required for any character can be found by determining which of the GPIO extender's pins are connected to which segments of the display.
5. Display GPIO chip's data again to check that you have changed the direction to output and turned on the necessary output pins of output:
   ```
   # i2cdump -y 1 0x20
   No size specified (using byte-data access)
        0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
   00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
   10: 00 00 1e 78 1e 78 00 00 00 00 00 00 00 00 00 00    ..?x?x..........
   20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
   ```
   <… omitted...>
6. If you want to change the pattern displayed on the 14-seg display, adjust what values you write to addresses 0x14 and 0x15. Try out different patterns to see how the display responds.
   - Hint: Look online for images which suggest how to display any letter or number on a 14-seg display. One good one is found here on Wikipedia; but you are welcome to choose you own.

3. I2C via C Code

3.1 Initialization

The following function initializes the I2C device. Note that the BeagleBone virtual I2C cape must already be enabled if trying to access `/dev/i2c-1`

```
#define I2CDRV_LINUX_BUS0 "/dev/i2c-0"
#define I2CDRV_LINUX_BUS1 "/dev/i2c-1"
#define I2CDRV_LINUX_BUS2 "/dev/i2c-2"

static int initI2cBus(char* bus, int address)
{
	int i2cFileDesc = open(bus, O_RDWR);
	if (i2cFileDesc < 0) {
		printf("I2C: Unable to open bus for read/write (%s)\n", bus);
		perror("Error is:");
		exit(1);
	}

	int result = ioctl(i2cFileDesc, I2C_SLAVE, address);
	if (result < 0) {
		perror("I2C: Unable to set I2C device to slave address.");
		exit(1);
	}
	return i2cFileDesc;
}
```

3.2 Writing a Register

The following function allows the program to write to an I2C device's register:

```
static void writeI2cReg(int i2cFileDesc, unsigned char regAddr,
			unsigned char value)
{
	unsigned char buff[2];
	buff[0] = regAddr;
	buff[1] = value;
	int res = write(i2cFileDesc, buff, 2);
	if (res != 2) {
		perror("I2C: Unable to write i2c register.");
		exit(1);
	}
}
```

3.3 Reading a Register

The following function allows the program to read from an I2C device's register:

```
static unsigned char readI2cReg(int i2cFileDesc, unsigned char regAddr)
{
	// To read a register, must first write the address
	int res = write(i2cFileDesc, &regAddr, sizeof(regAddr));
	if (res != sizeof(regAddr)) {
		perror("I2C: Unable to write to i2c register.");
		exit(1);
	}

	// Now read the value and return it
	char value = 0;
	res = read(i2cFileDesc, &value, sizeof(value));
	if (res != sizeof(value)) {
		perror("I2C: Unable to read from i2c register");
		exit(1);
	}
	return value;
}
```

3.4 Main program

The following code drives a pattern to the two 14-seg digits:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/i2c.h>
#include <linux/i2c-dev.h>

#define I2C_DEVICE_ADDRESS 0x20

#define REG_DIRA 0x00
#define REG_DIRB 0x01
#define REG_OUTA 0x14
#define REG_OUTB 0x15

// Insert the above functions here...

int main()
{
	printf("Drive display (assumes GPIO #61 and #44 are output and 1\n");
	int i2cFileDesc = initI2cBus(I2CDRV_LINUX_BUS1, I2C_DEVICE_ADDRESS);

	writeI2cReg(i2cFileDesc, REG_DIRA, 0x00);
	writeI2cReg(i2cFileDesc, REG_DIRB, 0x00);

	// Drive an hour-glass looking character
	// (Like an X with a bar on top & bottom)
	writeI2cReg(i2cFileDesc, REG_OUTA, 0x2A);
	writeI2cReg(i2cFileDesc, REG_OUTB, 0x54);

	// Read a register:
	unsigned char regVal = readI2cReg(i2cFileDesc, REG_OUTA);
	printf("Reg OUT-A = 0x%02x\n", regVal);

	// Cleanup I2C access;
	close(i2cFileDesc);
	return 0;
}
```

4. Driving Individual Digits

To display unique patterns (characters) on each of the digits, one must quickly switch between the two digits. Basically you turn on the left digit and drive the pattern for the left digit. The switch to turning on just the right digit and drive the pattern for the right digit. If this is done fast enough then it seems to the human eye that each digit is on all the time.

Here is the pseudo code for doing this:

1. Spawn a background thread.
2. In background thread, continuously loop through:
    1. Turn off both digits by driving a 0 to GPIO pins (Linux #'s 61 and 44).
    2. Drive I2C GPIO extender to display pattern for left digit.
       Must write pattern to registers 0x14 and 0x15.
    3. Turn on left digit via GPIO 61 set to a 1. Wait for a little time.
       (Waiting for 5ms seems to work well).

    4. Turn off both digits by driving a 0 to GPIO pins (Linux #'s 61 and 44).
    5. Drive I2C GPIO extender to display pattern for right digit.
       Must write pattern to register 0x14 and 0x15.
    6. Turn on right digit via GPIO 44 set to a 1. Wait a little time.
       (Waiting for 5ms seems to work well).

If the display seems to flicker, it either means you are not switching between digits fast enough, or you are not leaving the digits on for long enough for them to glow brightly. Ensure that inside your thread that you are doing only the minimum amount of other computation. Consider pre-computing and caching (in say a local or static global variable) any value which the code might compute on each pass through the loop.

For debugging, you may want to have your background thread wait 1s. This will only show the digits one at a time, but give you time to see what's happening and help you debug.