I2C Guide: 14-Seg Display & 8x8 LED Matrix

by Brian Fraser
Last update: Oct 14, 2022

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. Using I2C from Linux command-line to drive an 8x8 Matrix (HT16K33 chip).
4. C code to access I2C and drive the display.

Guide has been tested on

BeagleBone (Target):	**Debian 11.4**
PC OS (host):	**Debian 11.5**

Table of Contents

Formatting

1. Commands for the host Linux's console are show as:
   ```
   (host)$ echo "Hello PC world!"
   ```
2. Commands for the target (BeagleBone) Linux's console are shown as:
   ```
   (bbg)$ echo "Hello embedded world!"
   ```
3. Almost all commands are case sensitive.

Revision History

- Sept 17, 2019: Initial version not using Cape Manager
- Oct 3: Updated i2cget argument explanation.
- Oct 16: Added note that `config-pin` may fail if universal cape is not loaded.
- Jan 28, 2021: Changed how prompts are shown
- Oct 14, 2022: Added 8x8 LED matrix

1. I2C Basics

The I^{2}C (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]	SDA pin: P9_18 SCL pin: P9_17	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. SDA pin: P9_20 SCL pin: P9_19	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 walks through controlling a 2-digit 14-segment display. This display device 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's P8/P9 headers, 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 Enable the Bus

Note: If you find any guides that refer to the bone cape manager (`bone_capemgr`) or slots files, these guides no longer apply.[3]

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:
   ```
   (bbg)$ 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 you are wiring in a new 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:
   ```
   (bbg)$ 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
   ```
4. If your device is on hardware bus I2C1 you may first need enable Linux support for the bus (`/dev/i2c-1`):
   - Check if the pins are configured for i2c:
     ```
     (bbg)$ config-pin -q P9_18
     (bbg)$ config-pin -q P9_17
     ```
     If it prints mode "i2c", then it's configured

     Otherwise, when you have no capes loaded, then the universal cape may be active which makes the two pins for I2C-1 available as GPIO. You must change the pin configuration from GPIO to I2C:

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

3 Before kernel 4.9 (perhaps into the early 4.x kernels) there was a bone cape manager. One could use this to at run-time tell the kernel about changes to your hardware. For example, tell the kernel that it should enable support for I2C-1 with a statement like: `echo BB-I2C1 > /sys/devices/platform/bone_capemgr/slots`

However, this has now been removed from the Linux kernel and is now part of UBoot, and configured via `/boot/uEnv.txt`. See the audio guide for more info on loading capes.

```
(bbg)$ config-pin P9_18 i2c
(bbg)$ config-pin P9_17 i2c
```

Note that this may not be necessary if the I2C-1 cape has been loaded at startup (see Audio guide for more). Therefore, if your C program is executing these config-pin commands, you may want to allow it to ignore any failures so your program works regardless of how the I2C-1 bus is setup at the moment.

5. Display I2C devices on the chosen I2C bus:
   ```
   (bbg)$ i2cdetect -y -r 1
   ```
   - Where `1` refers to the Linux device `/dev/i2c-1`
   - Sample output:
     ```
     (bbg)$ 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. You must set the pin configuration each time your BeagleBone reboots.
   - If you are using any of the I2C devices on I2C-1, you likely need to have your program try to configure the P9_18 and P9_17 pins for I2C operation using the `config-pin` commands above.
7. Troubleshooting
   - If you run
     ```
     (bbg)$ i2cdetect -y -r 1
     ```
     and it takes a long time (seconds per address), and does not find anything, then you likely need to change your pins to be using I2C
     - Change the pin configuration to I2C:
       ```
       (bbg)$ config-pin P9_18 i2c
       (bbg)$ config-pin P9_17 i2c
       ```
     - You can see the current pin configuration with:
       ```
       (bbg)$ config-pin -q P9_18
       (bbg)$ config-pin -q P9_17
       ```
     - You can see the available pin modes using:
       ```
       (bbg)$ config-pin -l P9_18
       (bbg)$ config-pin -l P9_17
       ```

2.2 Working with a Device from Command Line

1. Display the internal memory of an I2C device:
   ```
   (bbg)$ 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 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:
     ```
     (bbg)$ i2cget -y 1 0x20 0x00
     0xff
     ```
     Arguments explanation:
     - `-y`: Disable "are you sure" confirmation prompt
     - `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:
   ```
   (bbg)$ i2cset -y 1 0x20 0x00 0x00
   (bbg)$ i2cset -y 1 0x20 0x01 0x00
   ```
   - These commands control device with address 0x20 on `/dev/i2c-1`: into register 0x00 it writes 0x00, and into 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:
   ```
   (bbg)$ 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 00 00 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.

2.3 Steps for Zen Cape’s 2-Digit 14-Seg Display

Here are the concise steps to turning on the Zen Cape’s 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):
     ```
     (bbg)$ echo 61 > /sys/class/gpio/export
     (bbg)$ echo 44 > /sys/class/gpio/export
     (bbg)$ echo out > /sys/class/gpio/gpio61/direction
     (bbg)$ echo out > /sys/class/gpio/gpio44/direction
     ```
     - If it says “-bash: echo: write error: Device or resource busy” it means that it’s already exported; so you may ignore this error at this time.
   - Drive a 1 to the GPIO pin to turn on the digit. The following turns on both digits.
     ```
     (bbg)$ echo 1 > /sys/class/gpio/gpio61/value
     (bbg)$ echo 1 > /sys/class/gpio/gpio44/value
     ```
2. Enable `/dev/i2c-1` (hardware bus I2C1):
   ```
   (bbg)$ config-pin P9_18 i2c
   (bbg)$ config-pin P9_17 i2c
   ```
3. Set direction of both 8-bit ports on I2C GPIO extender to be outputs:
   ```
   (bbg)$ i2cset -y 1 0x20 0x00 0x00
   (bbg)$ 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)
     ```
     (bbg)$ i2cset -y 1 0x20 0x14 0x1E
     ```
   - Drive upper 8-bits (register 0x15) (0xFF for all segments on, some may be unused)
     ```
     (bbg)$ i2cset -y 1 0x20 0x15 0x78
     ```
   - Find the exact bit pattern required for a character by determining which of the GPIO extender's pins are connected to which segments of the display (one bit at a time!).
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:
   ```
   (bbg)$ 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. 8x8 LED Matrix (HT16K33 chip)

For an 8x8 LED matrix using the HT16K33 chip, such as this one with "backpack" board from Adafruit, use the following setup and commands:

1. Wiring:
   - P9.01 (Ground) to "-" on LED matrix
   - P9.03 (3.3V) to "+" on LED matrix
   - P9.17 (I2C1's SCL) to "C" on LED matrix
   - P9.18 (I2C1's SDA) to "D" on LED matrix
2. Enable `/dev/i2c-1` (hardware bus I2C1):
   ```
   (bbg)$ config-pin P9_18 i2c
   (bbg)$ config-pin P9_17 i2c
   ```
3. Turn on the LED matrix via its System Setup register:
   ```
   (bbg)$ i2cset -y 1 0x70 0x21 0x00         # On
   ```
4. Set the display to be on, no flashing using its Display Setup register
   ```
   (bbg)$ i2cset -y 1 0x70 0x81 0x00         # On, No flash (required)
   ```
5. Set display bits on the top row (assuming connection pins at top):
   ```
   (bbg)$ i2cset -y 1 0x70 0x00 0x55         # 0x00=top row, 0x55=leds
   ```
6. Set other rows:
   ```
   (bbg)$ i2cset -y 1 0x70 0x02 0x55         # 0x02=2nd row
   (bbg)$ i2cset -y 1 0x70 0x04 0x55         # 0x04=3rd row
   (bbg)$ i2cset -y 1 0x70 0x0E 0x55         # 0x0E=bottom row
   ```
   Note: If display is oriented upside down, you'll need to remap the rows/columns in software.
7. Fast access to write all bits:

   In your C program, instead of writing one byte (like `i2cset` does) you can send multiple bytes to the display in a single file operation.
   - Make your write command use register 0x00 (start of frame memory)
   - Make your write command write 16 bytes to fill all 8 rows (and the unused rows)
   - This cannot be tested using `i2cset`, but works with the code below by making `buff` 17 bytes (one for the register address 0x00, and 16 for the data), and then writing the 17 bytes to the file. You'll need to pass in an array of 16 bytes instead of a single value too!

3.1 Registers Addresses and Commands for HT16K33

The LED Matrix driver's HT16K33 chip (datasheet here) allows us to use I2C to control the LED matrix. However, the chip's use of I2C registers is non-standard because it merges I2C register addresses and commands into one byte[4]. For example, in the above steps that writing to the System Setup and Display Setup registers write 0x00. It turns out that it does not matter what is written. Simply writing to that address changes the value.

For example, the Display Setup register's address is defined (p30 of datasheet) to be `0b1000XQRS` (in binary), where bits `X`=don't care, `Q`=blink rate bit 1, `R`=blink rate bit 0, `S`=on(1) or off(0). So, writing anything to address 0x81 turns on the display; writing anything to address 0x80 turns off the display; writing anything to 0x83 turns it on and flashes.

4 Yup! That was _fun_ to figure out!

4. I2C via C Code

4.1 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`

```
// Assume pins already configured for I2C:
//   (bbg)$ config-pin P9_18 i2c
//   (bbg)$ config-pin P9_17 i2c

#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);

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

4.2 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);
	}
}
```

4.3 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;
}
```

4.4 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;
}
```

5. Driving Individual Digits to Zen Cape 14-seg Display

On the Zen Cape, 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.