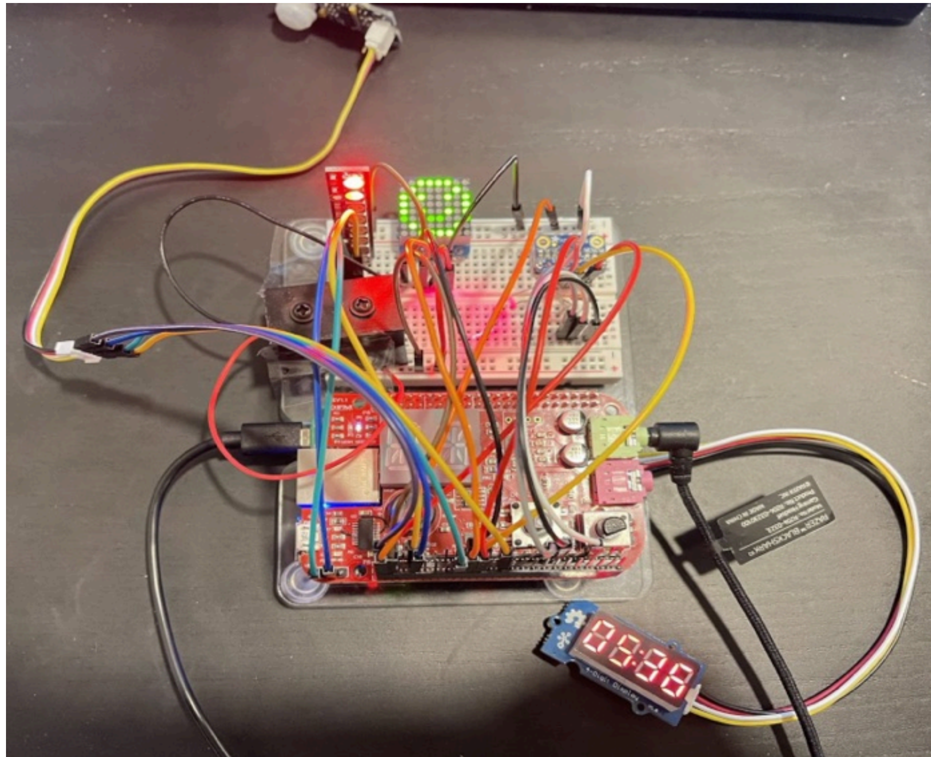
CMPT433 Project: GatekeeperInsights How-To Guide

Introduction: Our project is an embedded system designed for retail stores, featuring advanced sensors to monitor customer traffic and temperature at entrances. This compact, integrated solution provides real-time data analysis, enhancing operational decisions and customer experiences, making it essential for optimizing store performance and environment.



65 possible digital I/Os

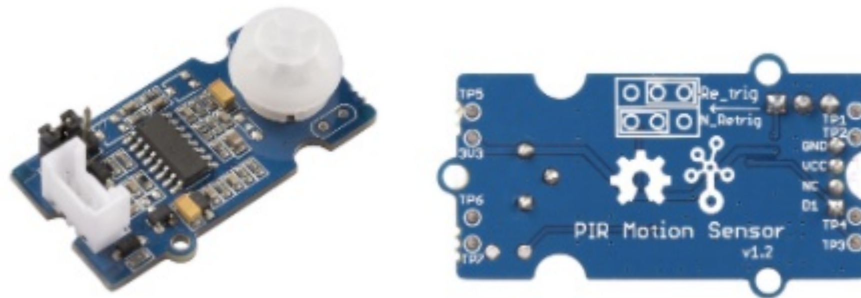
P9				P8			
DGND	1	2	DGND	DGND	1	2	DGND
VDD_3V3	3	4	VDD_3V3	GPIO_38	3	4	GPIO_39
VDD_5V	5	6	VDD_5V	GPIO_34	5	6	GPIO_35
SYS_5V	7	8	SYS_5V	GPIO_66	7	8	GPIO_67
PWR_BUT	9	10	SYS_RESETN	GPIO_69	9	10	GPIO_68
GPIO_30	11	12	GPIO_60	GPIO_45	11	12	GPIO_44
GPIO_31	13	14	GPIO_50	GPIO_23	13	14	GPIO_26
GPIO_48	15	16	GPIO_51	GPIO_47	15	16	GPIO_46
GPIO_5	17	18	GPIO_4	GPIO_27	17	18	GPIO_65
I2C2_SCL	19	20	I2C2_SDA	GPIO_22	19	20	GPIO_63
GPIO_3	21	22	GPIO_2	GPIO_62	21	22	GPIO_37
GPIO_49	23	24	GPIO_15	GPIO_36	23	24	GPIO_33
GPIO_117	25	26	GPIO_14	GPIO_32	25	26	GPIO_61
GPIO_115	27	28	GPIO_113	GPIO_86	27	28	GPIO_88
GPIO_111	29	30	GPIO_112	GPIO_87	29	30	GPIO_89
GPIO_110	31	32	VDD_ADC	GPIO_10	31	32	GPIO_11
AIN4	33	34	GNDA_ADC	GPIO_9	33	34	GPIO_81
AIN6	35	36	AIN5	GPIO_8	35	36	GPIO_80
AIN2	37	38	AIN3	GPIO_78	37	38	GPIO_79
AIN0	39	40	AIN1	GPIO_76	39	40	GPIO_77
GPIO_20	41	42	GPIO_7	GPIO_74	41	42	GPIO_75
DGND	43	44	DGND	GPIO_72	43	44	GPIO_73
DGND	45	46	DGND	GPIO_70	45	46	GPIO_71

Source: http://beagleboard.org/support/bone101

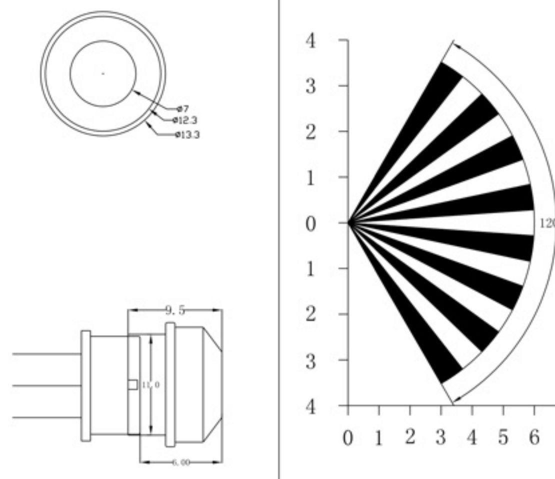
1. Motion Sensor (Traffic IN)

1.1 Motion Sensor Description

PIR Sensors or Passive InfraRed Sensors work by detecting the difference in infrared amounts between two infrared-sensitive slots. When a person or animal passes by, it radiates infrareds, which will be noticed by the first slot. The difference between the new infrared level in slot one and the ambient (environment level) infrared in the second slot is what cause the sensor to detect motion.
Grove PIR Motion Sensor is a digital sensor which will output either 1 (HIGH) or 0 (LOW) to its signal pin.



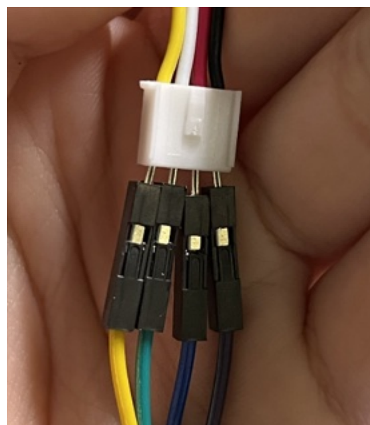
The sensor has a detecting distance of 3 meters and a detecting angle of 120 degrees. Up to the time this guide is written, the sensor does not support BeagleBone and we can't use the Grove I2C connector on BeagleBone Green.



1.2 Motion Sensor Setup

Grove PIR Motion Sensor uses a specific Grove cable which consists of four cables: red, black, white, and yellow cable. The red cable is for power, black cable is for ground, white cable is for secondary digital I/O, and the yellow cable is for primary digital I/O.

The picture above shows the cable you need, if you only have ordinary grove cable on hand, you can try “brute force” the cable setup like we did below.



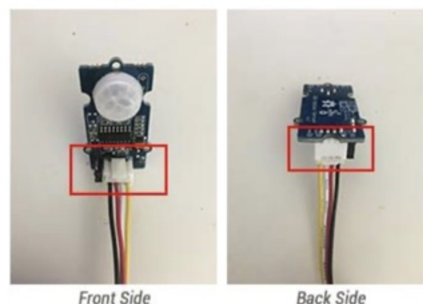
1.3 Motion Sensor Wiring

Now you have all the required parts,

- BeagleBone Green
- 1xGrove 4 pin Female Jumper to Grove 4pin Conversion Cable
- Grove Motion Sensor

Let’s wire them up!

1. Do the following before you start wiring:
   - To ensure safety, turn off your BeagleBone before executing the following steps.
   - To avoid static, wear an antistatic wrist strap or ground yourself by touching an unpainted metal on your computer, such as the USB port.
2. Plug in the Grove 4 pin cable to the Grove connector on the Grove PIR Motion Sensor.



Ensure that the cable is fully plugged into the Grove connector.

3. Wire up the four female jumper cables:
- Black cable to GROUND pin (P9_01 or P9_02)
- Red cable to SYS_5V pin (P9_07 or P9_08)
- White cable to a free GPIO pin (P9_12, P9_15, P9_23, or P9_27)
- Yellow cable to a free GPIO pin (P9_12, P9_15, P9_23, or P9_27)

Our team used P9_01, P9_07, P9_15, and P9_23.
Notes:
- Make sure that each female jumper cable is tightly connected to the pin. Otherwise, it would be easily plugged out from the pins.
- P9_12 corresponds to GPIO #60.
- P9_15 corresponds to GPIO #48.
- P9_23 corresponds to GPIO #49.
- P9_27 corresponds to GPIO #115.

1.4 Motion Sensor Command Line Test

After finishing all the steps above, now we can give it a simple test in the command line to see if the motion sensor is working properly.

1. Connect your BeagleBone to your host VM SSH to your target device:
    - (host)$ ssh root@192.168.7.2
2. Go to sys directory for GPIO:
    - (bbg)$ cd /sys/class/gpio
3. Export the pin which is connected to your primary I/O (yellow cable) in order to tell Linux to handle it as GPIO.
    - (bbg)$ echo 49 > export

    Note if the pin already been export, it will show
    - write error: Device or resource busy

4. Verify if the gpio exported
    - (bbg)$ ls /sys/class/gpio

2. Laser Diode + Photoresistor (Traffic OUT)

2.1 Laser Diode Wiring

The Laser Diode in the above picture works pretty simple, it takes 3.3V or 5V as input voltage, the red cable is VDD, and the black cable is GND. Connecting them to any available pin on the board will light up the laser.

2.2 A2D Basics(Brian Version)

Internally in a computer, values are stored as digital: either on (1) or off (0). In reality, signals are analog, which means they are a voltage level and not just on or off. The potentiometer, for example, is a knob the user can turn and generate a voltage between some min and max levels. The Analog to Digital converter (A2D or ADC) is used to convert an analog signal (such as 1.2V) into a digital number in the computer (such as 2731).
The hardware has a limited range of voltages it can tolerate without being damaged. On the BeagleBone, this range is 0 to 1.8V. Be very careful not to exceed 1.8V on the input, even though there are 3.3V and 5V voltages also on the BeagleBone. On the Zen cape, the potentiometer (POT) has been wired to give a value between 0 and 1.8V.

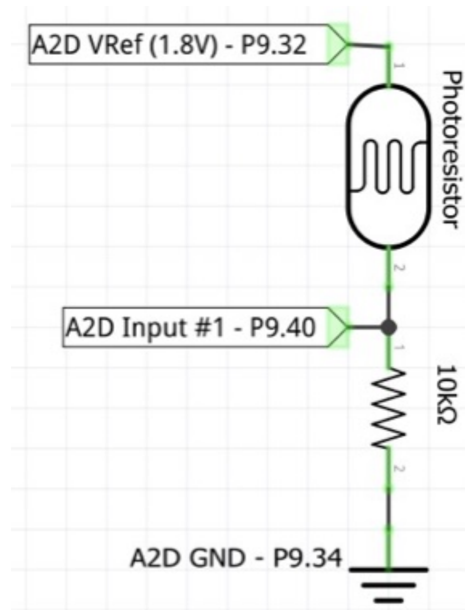
2.3 Photoresistor Introduction

A photoresistor (also called a photocell, or phototransistor) is a discrete component (component we can wire into our breadboard). The resistance of the photoresistor changes depending on how much light it is exposed to. We can use an A2D voltage reading to infer the amount of light.

2.4 Photoresistor Wiring

Next we will introduce how to wire up photoresistor
Critical Tips:

- Do all your wiring and wiring changes with the power to the BeagleBone turned off.
- Try to have some space between the components you place on the breadboard to prevent
- them from unexpectedly shorting (touching the bare wire leads).
- Note that it’s OK for the plastic coated parts of the wires to touch; it’s only the bare metal that can conduct electricity.
- Avoid static! Before working on the components, try grounding yourself by touching some unpainted metal on your computer, such as the USB port, or metal on a light switch.



A photoresistor (also called a photocell, or phototransistor) is a discrete component (component we can wire into our breadboard). The resistance of the photoresistor changes depending on how much light it is exposed to. We can use an A2D voltage reading to infer the amount of light.

The circuit we are creating is shown on the top in a schematic form. It does not matter which direction the resistors and photoresistors are placed: they can be turned around "backwards" without a problem.

- The photoresistor and 10k ohm resistor are connected in series (one into the next). The diagram shows the photoresistor on top (it has the arrow through it), and the 10k ohm resistor on the bottom.
- Connect the "top" of the photoresistor to:
    - P9.32: 1.8V A2D reference voltage
- Connect the "middle" between the components together, and connect this to:
    - P9.40: A2D input pin #1.
- Connect "bottom" of the 10k10 resistor to:
    - P9.34: A2D ground (A2D is a precise device; it has a separated ground from GPIO).

2.5 Enabling the A2D in Linux

1. Change to the sysfs directory for the A2D readings. The sys file system gives access to many Linux devices. The folder is /sys/bus/iio/devices/iio:device0 however the ':' must be escaped for the Linux command line (but not in your C program!)
    - (bbg)$ cd /sys/bus/iio/devices/iio\:device0
2. To read photoresistor
    - (bbg)$ cat in_voltage1_raw
3. To read actual voltage 1

- (bbg)$ cat in_voltage1_raw
- The value will be between 0 and 4095 (4K – 1)
- You can compute the voltage with the formula:
  voltage = (value / max) * reference_voltage
- So, if you just read 3103 it relates to the real world voltage of:
  voltage = (3103 / 4095) * 1.8
  = 1.36V
- Tip: If doing this sort of math in C, make sure you use the correct data types to avoid a rather unhelpful integer division.

Troubleshooting:

- If the value read from the A2D does not change significantly (by at least 500) between direct light and no light, try the following:
- Ensure that the photoresistor and the 10k resistor are wired into the same column on the breadboard so that they make a connection.
- Ensure that the A2D sensing wire is connected to the same column as the connection between the photoresistor and the 10k resistor.
- Try swapping each wire out with a different one (they are sometimes defective)
- If the value read is always very low (<10) it's likely that there is a good connection to ground, but that something is wrong with connecting the photoresistor to 1.8V.
- If the value read is always very high (>4000) it is likely that there is something wrong with the 10k resistor's connection to ground.

3. Audio Play Sound

3.1 Accessing the Cape Manager

Linux must be told what hardware is connected to the CPU. It learns this at boot up using a Device Tree (file is a .DTB for the device tree binary). The boot loader (UBoot) detects what capes are installed (or configured) and sets up the device tree for the kernel to use.

Do the following just once (per board).

1. Backup uEnv.txt

The uEnv.txt file is critical to controlling how UBoot starts the system. We will change it to load the audio setup; however, we must first take a backup copy to recover from some errors.

(bbg)$ cd /boot

(bbg)$ sudo cp uEnv.txt uEnv-BeforeAudio.txt

WARNING: Corrupting the uEnv.txt file may cause the BeagleBone to be unbootable, and hence must be reflashed. Be careful editing this file!

2. Edit uEnv.txt to load the Audio overlays

- Edit uEnv.txt:
  (bbg)$ sudo nano /boot/uEnv.txt
- Find the section titled:

###Additional custom capes

- Change it to be:
  ###Additional custom capes
  uboot_overlay_addr4=/lib/firmware/BB-BONE-AUDI-02-00A0.dtbo
  #uboot_overlay_addr5=<file5>.dtbo
  #uboot_overlay_addr6=<file6>.dtbo
  #uboot_overlay_addr7=<file7>.dtbo

- The first line (BB-BONE-AUDI) loads the audio device tree overlay.
- Ensure you removed the # on the line to uncomment it.

3. Reboot the target
4. Verify it worked:
   - Ensure the audio has loaded (first give it access to the internet):

```
(bbg)$ sudo apt install alsa-utils
...
(bbg)$ aplay -l      # lower-case L
**** List of PLAYBACK Hardware Devices ****
card 0: B [AudioCape Rev B], device 0:
davinci-mcasp.0-tlv320aic3x-hifi
tlv320aic3x-hifi-0 [davinci-mcasp.0-tlv320aic3x-hifi
tlv320aic3x-hifi-0]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```

   - Ensure the I2C for bus 1 has still loaded:

```
(bbg)$ config-pin p9_18 i2c
(bbg)$ config-pin p9_17 i2c
(bbg)$ i2cdetect -l

i2c-1 i2c    OMAP I2C adapter   I2C adapter
i2c-2 i2c    OMAP I2C adapter   I2C adapter
i2c-0 i2c    OMAP I2C adapter   I2C adapter

(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: -- -- -- -- -- -- -- --
```

If this i2cdetect command runs very slowly (> 1s) then you likely have not run config-pin on the two I2C pins.

If you are using a Zen Cape red, then you may not see “1c” but rather “18” in the figure; that is fine.

3.2 Play Audio

1. On the target, install the sound tools and libraries (aplay, alsamixer, lsusb)
(bbg)$ sudo apt install usbutils alsa-utils i2c-tools

2. Plug in speakers or headphones into the audio output on the Zen cape (green 3.5mm socket).
   - WARNING: When testing the audio, do not put headphones in your ear. A very loud sound is possible if there are problems, and this could cause an injury to your ear.
   - Just drape the headphones beside your ears so you can hear it, but not be injured if it goes wrong. Once you know the audio levels are fine, then using headphones normally is fine.
3. Save a WAVE file to your NFS folder on your host (say, sample.wav).
4. On the target, play the file with:
   (bbg)$ aplay sample.wav
5. Change the volume:
   (bbg)$ alsamixer
   - Use the left/right arrows to select different channels to adjust.
   - Use the up/down arrows to change the volume.
   - Press 'M' to mute or unmute channels.
   - Press ESC to exit (and save changes).
   - Change the volume of wave data playback by changing the PCM channel.

3.3 Project Library Requirements

1. On the BeagleBone, install asound
   (bbg)$ sudo apt update
   (bbg)$ sudo apt install libasound2
   - You can check for the necessary files using:
     (bbg)$ ls /usr/lib/arm-linux-gnueabihf/libasound*
     libasound.so.2 libasound.so.2.0.0
2. On the host, install the asound development library (for the header files)
   (host)$ sudo apt update
   (host)$ sudo apt install libasound2-dev
3. The host will need the armhf version of the asound library files in order to cross-compile the application. To do this we will install the armhf library on the host. (Very useful for cross- compiling libraries!)
   - Setup the host to be able to install armhf files and install ALSA:
     (host)$ sudo dpkg --add-architecture armhf
     (host)$ sudo apt update
     (host)$ sudo apt install libasound2-dev:armhf
   - Verify the library installed:

```
(host)$ ls /usr/lib/arm-linux-gnueabihf/libasound.so
        /usr/lib/arm-linux-gnueabihf/libasound.so
```

3.3 CMake Requirements and Music

```
# ALSA support
find_package(ALSA REQUIRED)
target_link_libraries(allin LINK_PRIVATE asound)
```

The Code above should be included in the Cmake file, it will automatically find the armhf library installed in the above steps.

Also, the wave file named enter.wav and out.wav should be copied over under
~/cmpt433/public/myApps/wave-files
The music source file can be changed based on needs, as long as it satisfies the naming requirements.

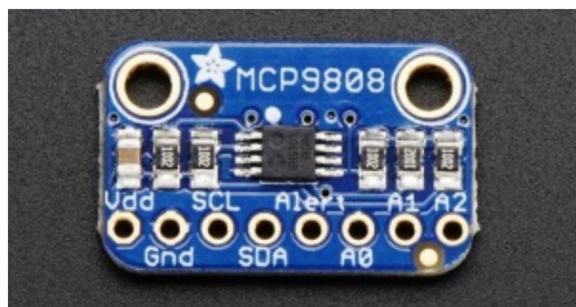
4. Temperature Sensor (MCP9808)

4.0 Solder Your Temperature Sensor

Trim the header strip to the required length. For easier soldering, mount it onto a breadboard with the longer pins facing downwards. Carefully solder each pin to ensure a solid electrical connection. Make sure all pins are thoroughly soldered.

4.1 Wire Temperature Sensor to Beaglebone Green(bbg)

1.There are 8 pins on MCP9808 in total.



Power Pins:
- VDD: This is the positive power and logic level pin. It can be 2.7v - 5.5v.
- GDN: This is the ground power and logic reference pin.

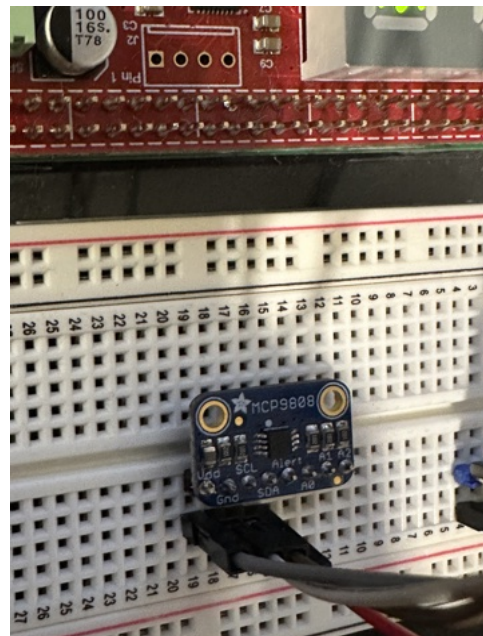
I2C Data Pins:
- SCL: This is the I2C clock pin.
- SDA: This is the I2C data pin.

Optional Pins:
- Alert: This is the interrupt/alert pin from the MCP9808. It can “alert” you when the temperature is going above or below the set of MCP9808.
- A0/A1/A2: There are the address selection pins. The default address of MCP9808 is 0x18, if you want to use another address you can wire those pins to bbg. And the calculation is adding A0(1)/A1(2)/A2(4) to 0x18.

For example, only wire A0 to bbg, the new address is: 0x18 + 1 = 0x19.
only wire A1 to bbg, new address is: 0x18 + 2 = 0x1A.
wire A0 and A2 to bbg, new address is: 0x18 + 1 + 4 = 0x1D.

2.How to wire:(In this guide, we are using I2C-2, you can use other I2C as well)

- If you are going to use the default address(0x18), wire VDD to P9_3 on your bbg (any other PIN between 2.7v - 5.5v will be fine). For example, if you are going to use 0x19 as your address, then only A0 is tied to VDD(P9_3).
- GDN is wired to P9_2 on bbg (any other DGND PIN will be fine).
- SCL is wired to P9_19 on bbg. (In this case, we are going to use I2C-2)
- SDA is wired to P9_20 on bbg.



In the image above, we used VDD pin connected to P9_3 as the positive terminal for the temperature sensor, setting its default address to 0x18. If you plan to use a different address, such as 0x19 described earlier, you simply need to switch the wire from VDD pin to connect to A0 pin instead.

4.2 Configure Pins to I2C mode

- Change the current pin configuration to I2C mode with:
  (bbg)$ config-pin P9_19 i2c
  (bbg)$ config-pin P9_20 i2c
- Check the current pin configuration with:
  (bbg)$ config-pin -q P9_19
  (bbg)$ config-pin -q P9_20
  Now, you should be able to see:
   Current mode for P9_19 is :   i2c
   Current mode for P9_20 is :   i2c

4.3 Check I2C Bus

1. Install the i2c tools:(if you already installed, you can skip this step)
   (bbg)$ sudo apt-get install i2c-tools
2. Display I2C device on the chosen I2C-2 bus: (In the above steps, we chose to wire I2C-2)
   (bbg)$ i2cdetect -y -r 2

   For more information, If you are trying to display I2C-1, you can use:
   (bbg)$ i2cdetect -y -r 1

   If you see the following output, it means your Temperature Sensor is wired correctly and I2C detect successfully on I2C-2:

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:                         -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- UU 19 -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- UU -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

   Explanation: This screenshot is the output from the “i2cdetect -y -r 2” command, which is used to scan the I2C-2 bus for devices. The output displays a grid with rows labeled with numbers from 00 to 70 and columns labeled from 0 to f, which corresponds to hexadecimal addresses that can be used by I2C devices.

   More information about the meaning of symbols in the output:
   --: This indicates that no device was detected at this address.
   UU: This typically means that there is a device at this address, but it is being used by a driver in the system and cannot be probed. It is "Unusable" for direct I2C communication because the system has reserved it for some driver. So, we may not be able to use 0x18 as the address of the Temperature Sensor in this case.
   19: This is a hexadecimal number representing the address of an I2C device detected on the bus that is not reserved by the system (i.e., not marked with UU). This could be the address of your temperature sensor if it is set to use a non-default address and wire A0 pin.

Troubleshooting: If you see “--” on address 0x19 in the above output.

- You can use: (bbg)$ i2cdump -y 2 0x19
- This command line will instruct the system to non-interactively dump the contents of the registers of the I2C device at address 0x19 on bus 2 and display them in a hex dump format.
- If you see the output like the screenshot below:

```
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: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
10: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
20: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
30: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
40: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
```

It means no I2C device at address 0x19 on bus 2.

- Make sure you are using I2C-2 and 0x19 as your address. And double check you correctly wire the Temperature Sensor. Or you can try to wire VDD/A1/A2 instead, and come with a different address 0x18/0x1A/0x1C.

4.4 Reading Data via C

1.Opening the I2C Bus

We need to establish a connection with the I2C bus. This is like opening a communication channel with the devices connected to the bus.

2.Specifying the Device Address

- In the context of I2C communication, every device on the bus is assigned a unique address. This address is used by the I2C master (in our case, the computer or microcontroller that's running our code) to direct communication to the correct device.
- We will use ioctl() on the file descriptor that represents our I2C bus
- C code:

```
if (ioctl(i2cFileDescriptor, I2C_SLAVE, 0x19) < 0) {
    printf("Cannot find the device on the bus.\n");
    //exit(1);
}
```

3.Configuring the Sensor

- Configuring the sensor means setting it up so it knows how we want to receive the temperature data.
- Configuration Register: This register controls various settings of the sensor, like the measurement mode. We're writing 0x00 twice to it to ensure it's set to default settings.

```
char sensorConfig[3] = {0x01, 0x00, 0x00};
write(i2cFileDescriptor, sensorConfig, 3);
```

- Resolution Register: This register controls the precision of the temperature readings. We're setting it to 0x03, which gives us the highest resolution possible with the MCP9808.

```
char resolutionConfig[2] = {0x08, 0x03};
write(i2cFileDescriptor, resolutionConfig, 2);
```

4. Reading Temperature Data

- Read the temperature from the sensor.

```
char tempRegAddr = 0x05;
write(i2cFileDescriptor, &tempRegAddr, 1);
char tempData[2] = {0};
if (read(i2cFileDescriptor, tempData, 2) != 2) {
    printf("Temperature read failed.\n");
    exit(1);
}
```

5.Calculating the Actual Temperature

- The data we read is raw and needs to be converted to human-readable form.

```
int tempRaw = (tempData[0] & 0x1F) << 8 | tempData[1];
if (tempRaw > 4095)
    tempRaw -= 8192;
float temperature = tempRaw * 0.0625;
```

5. 14-Seg Display via I2C

Use the Zen cape's two digit 14-segment display to display the number of people in the room that the program has detected using the Motion Sensor and Laser Diode during the previous time. By controlling this I2C device, we can drive the display's segments. The GPIO extender's 16 bits of output are divided into two 8-bit ports.

5.1. Install the I2C tools:

(bbg)$ sudo apt-get install i2c-tools

5.2. If your device is on hardware bus I2C1 you may first need enable Linux support for the bus (/dev/i2c-1):

Set the pins configuration to i2c:

(bbg)$ config-pin P9_18 i2c

(bbg)$ config-pin P9_17 i2c

```
debian@lingjiel-beagle:~$ config-pin P9_18 i2c
Current mode for P9_18 is:     i2c
debian@lingjiel-beagle:~$ config-pin P9_17 i2c
Current mode for P9_17 is:     i2c
```

5.3. Display the internal memory of an I2C device

(bbg)$ i2cdump -y 1 0x20

```
debian@lingjiel-beagle:~$  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 ff ff 00 00 00 00 ff ff 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    ................
30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
```

5.4. In our setup, each of the two pins capable of digital signal processing is controlled directly by the microcontroller. On the Zen cape, these pins are connected to P8_26 and P8_12, which are recognized within the Linux system as pins 61 and 44. Setting pin 61 to a high state powers the left digit, while doing the same to pin 44 powers the right digit.

- **Configure** both pins on the microprocessor for output through GPIO (see other guide). If GPIO pins not yet exported, then export them (avoid re-exporting pins):
  (bbg)$ echo 61 > /sys/class/gpio/export
  (bbg)$ echo 44 > /sys/class/gpio/export

```
debian@lingjiel-beagle:~$ echo 61 > /sys/class/gpio/export
-bash: echo: write error: Operation not permitted
debian@lingjiel-beagle:~$ echo 61 > /sys/class/gpio/export
debian@lingjiel-beagle:~$ echo 44 > /sys/class/gpio/export
-bash: echo: write error: Operation not permitted
debian@lingjiel-beagle:~$ echo 44 > /sys/class/gpio/export
```

Troubleshooting: If it says "-bash: echo: write error: Operation not permitted" it means that the pin was already exported. However, you'll then need to export the pin a second time to make it work.

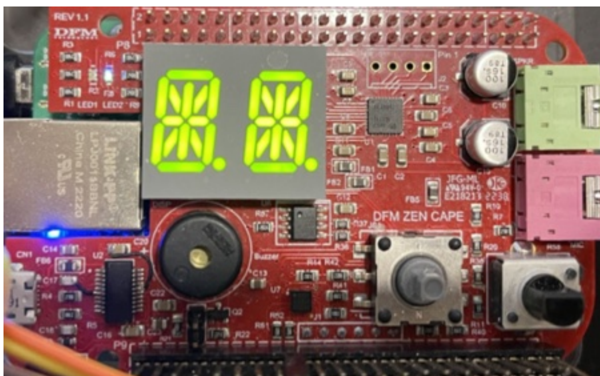
- **Set direction to output**:
  (bbg)$ echo out > /sys/class/gpio/gpio61/direction
  (bbg)$ echo out > /sys/class/gpio/gpio44/direction

Troubleshooting: If it says "-bash: echo: write error: no such file or directory" it means that the pin is not exported; execute the above export command to correct it.

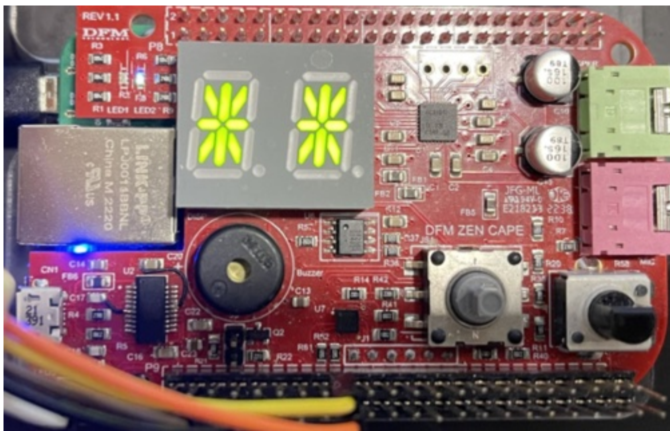
- **Drive a 1 to the GPIO pin to turn on the digit. The following turns on both digits**.
  (bbg)$ echo 1 > /sys/class/gpio/gpio61/value
  (bbg)$ echo 1 > /sys/class/gpio/gpio44/value

Troubleshooting: If it says "-bash: echo: write error: operation not permitted" it means that the pin is not set to output; execute the above direction command to correct it.

- Set direction of both 8-bit ports on I2C GPIO extender to be outputs:
  Zen Cape Red:
  (bbg)$ i2cset -y 1 0x20 0x02 0x00
  (bbg)$ i2cset -y 1 0x20 0x03 0x00

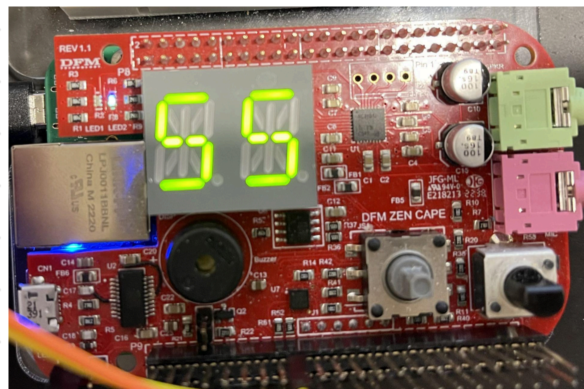


- The following instructions activate specific segments on the display to create a '*' symbol. This two-digit, 14-segment display operates with 15 GPIO pins controlling each segment, including the 14 segments and an additional decimal point. By writing a bit value of 1 into the control registers, a segment is illuminated; conversely, writing a 0 turns the segment off.
  Zen Cape Red (V1.1):
  - **Drive 8-bits in "bank B" with register 0x00**
  (bbg)$ i2cset -y 1 0x20 0x00 0x0f
  - **Drive 8-bits in "bank A" with register 0x01**
  (bbg)$ i2cset -y 1 0x20 0x01 0x5e



5.5. The following are commonly used numerical addresses:

	0x00	0x01
0	0xd0	0xa1
1	0xc0	0x00
2	0x98	0x03
3	0xd8	0x03
4	0xc8	0x22
5	0x58	0x23
6	0x58	0xa3
7	0x02	0x05
8	0xd8	0xa3
9	0xd8	0x23



For example: Number "55"
(bbg)$ i2cset -y 1 0x20 0x00 0x58
(bbg)$ i2cset -y 1 0x20 0x01 0x23

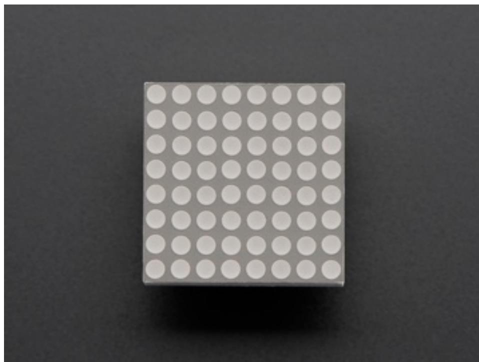
Tips: To ensure a stable display on a two-digit, multi-segment screen, follow these guidelines: Focus the display loop on switching between digits—avoid computations within this loop. Turn off the current digit, update the value, then turn on the next digit, maintaining clarity. Introduce a 5ms pause between activations for visual stability. Avoid external operations like

printing to screen during this loop. Ensuring your system isn't overloaded with other tasks will help achieve a clear and stable digit display without flickering.

5.6. In our project, we employ a 14-segment display to show the number of people detected in the room and the temperature. To ensure accuracy, we've set a maximum limit of 99. This means that regardless of the actual number of people detected or the temperature, the display will not show values exceeding 99.

6. Mini 8x8 Yellow-Green LED Matrix

For an 8x8 LED matrix using the HT16K33 chip, such as this one with a “backpack” board from Adafruit, use the following setup and commands.
Note: This hardware is not included in the hardware kit for this class.



6.1. 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

6.2. Enable /dev/i2c-1 (hardware bus I2C1):

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

6.3. Turn on the LED matrix via its System Setup register:

```
(bbg)$ i2cset -y 1 0x70 0x21 0x00 # On
```

6.4. Set the display to be on, no flashing using its Display Setup register

```
(bbg)$ i2cset -y 1 0x70 0x81 0x00 # On, No flash (required)
```

6.5. Set display bits on the top row (assuming connection pins at top):

```
(bbg)$ i2cset -y 1 0x70 0x00 0x55 # 0x00=top row, 0x55=leds
```

6.6. 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
```

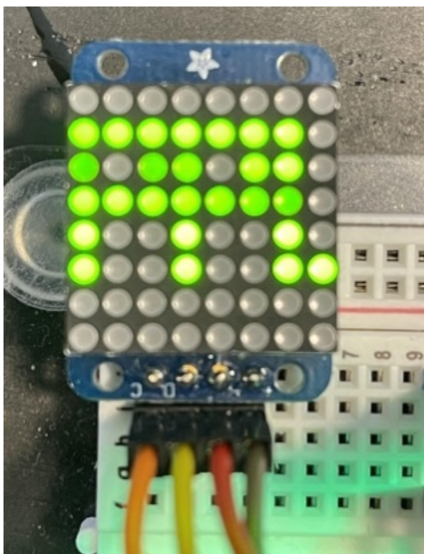
Note: If display is oriented upside down, you’ll need to remap the rows/columns in software

6.7. In a C program, rather than sending a single byte at a time (as the i2cset command does), you have the ability to transmit multiple bytes to the display through a single file operation. Here's how to do it:

- Initiate your write operation at register 0x00, which is the start of the frame memory.
- Design your write command to dispatch 16 bytes in one go, filling all 8 rows of the display, including those rows that are not used.
- While this approach cannot be validated using i2cset, it's feasible with the following method: create a buffer of 17 bytes, with the first byte for the register address (0x00) and the following 16 bytes for the data. Then, write these 17 bytes to the file in a single operation. Remember, you'll be passing an array of 16 bytes for the data, rather than just a single byte.

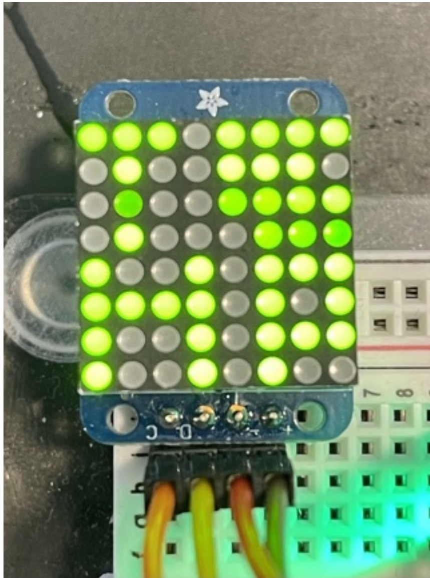
6.8. In our project, we are utilizing a Mini 8x8 Yellow-Green LED Matrix to display the number of people (ppl), the temperature (temp), and a smiley face expression. The design and the required addresses have been provided for reference, allowing for customization based on individual requirements. Users are encouraged to tailor the appearance of the display to meet their specific needs. This flexibility in design ensures that the LED matrix can effectively convey the intended information—whether it's the current temperature, people count, or a simple smiley face to indicate a positive status—all within the constraints of the 8x8 matrix format.

- People Count(PPL)



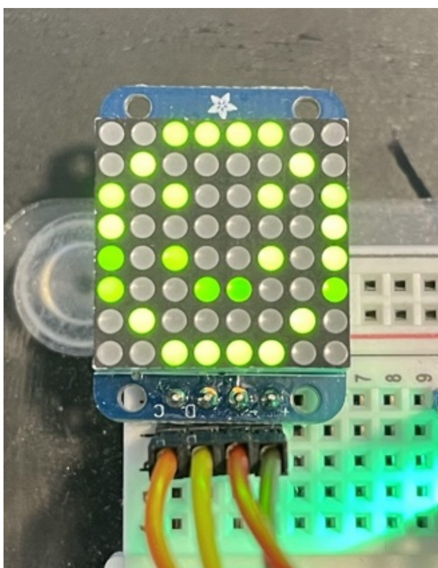
	PPL
Row8	0x00
Row7	0x7f
Row6	0x5b
Row5	0x7f
Row4	0x49
Row3	0xc9
Row2	0x00
Row1	0x00

- Temperature(TEMP)



	TEMP
Row8	0xf7
Row7	0x27
Row6	0xa7
Row5	0xa3
Row4	0xcb
Row3	0xfa
Row2	0xcb
Row1	0x4a

- Smile

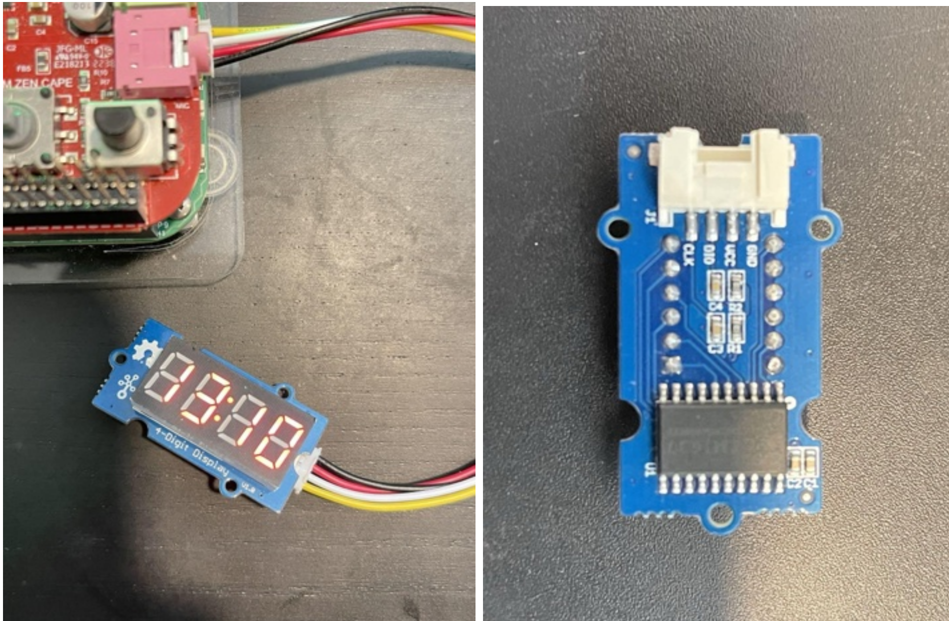


	Smile
Row8	0x1e
Row7	0x21
Row6	0x02
Row5	0xc0
Row4	0xd2
Row3	0xcc
Row2	0x21
Row1	0x1e

7. Grove 4-Digit Display

This tutorial offers a step-by-step process for integrating the Grove 4-digit display with the BeagleBone Green. The Grove system simplifies hardware connections with its uniform 4-pin connector. To follow this guide, you should already possess a basic understanding of working with GPIO in C, as detailed in Brian Fraser's GPIO tutorial. Unlike devices that use standard communication protocols, controlling the 4-digit display requires manual bit manipulation (bit-banging) via the Grove UART port. This involves using one pin for clock signals and another for transmitting data.

Note: Data should only be sent when the clock signal is low. The pulse width of the clock should also be at least 400ns. This hardware is not included in the hardware kit for this class.



7.1. Getting Started with the Display:

The display utilizes the TM1637 chip. Familiarize yourself with the TM1637 Datasheet, as it will be referenced. We'll use the auto increment mode for simplicity, allowing us to control the display with fewer commands.

7.2.GPIO setup:

- On the Grove 4-digit display, we can see that we will need to control the CLK and DIO lines. Since we are using the Grove UART port, these map to RX and TX pins respectively. The Grove UART connector on the BeagleBone is connected to UART2.
- You'll need to control the CLK and DIO lines, mapped to RX and TX pins on the Grove UART connector. Specifically, RX connects to P9_22 (GPIO 2) and TX to P9_21 (GPIO 3) on the BeagleBone Green. Ensure these pins are correctly configured for your project.
- Set the pins configurations to gpio:

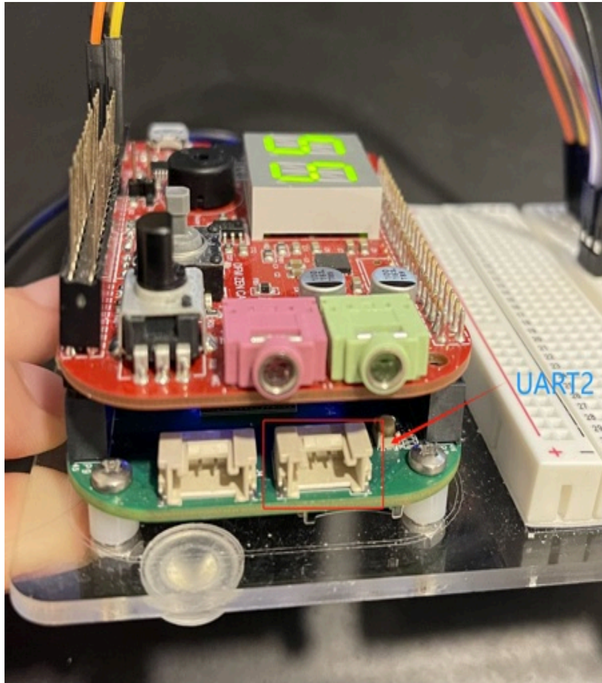
```
(bbg)$ config-pin P9_18 gpio   #CLK
(bbg)$ config-pin P9_17 gpio   #DIO
```

- Configure both pins on the microprocessor for output through GPIO

```
(bbg)$ echo 2 > /sys/class/gpio/export
(bbg)$ echo 3 > /sys/class/gpio/export
```

- Set direction to output:

```
(bbg)$ echo out > /sys/class/gpio/gpio2/direction
(bbg)$ echo out > /sys/class/gpio/gpio3/direction
```



Basic Steps for Display Control:

- **Initialize Communication**: Send a start signal to indicate the beginning of data transmission.
- **Command for Address Incrementing**: Allows automatic progression through display addresses.
- **End Transmission**: Send a stop signal to end the communication session.
- **Repeat Start Signal**: Needed before sending new data.
- **Specify Starting Address**: Indicates where to begin data update on the display.
- **Send Data**: Transmits the actual values to display.
- **Conclude with Stop Signal**: Ends the data sending process.
- **Control Display Data**: Manage what's shown on the display.
- **Final Stop Signal**: Officially ends the communication.

Note:Remember, data transmission occurs when the clock signal is low, with a minimum pulse width of 400ns for reliability.

7.3. Controlling the Display:

Start by sending a command (0x40) to enable auto address incrementing, allowing straightforward data input for the digits. When displaying values, use a starting address followed by the digit data. For brightness control, use the command (0x88) combined with a brightness level from 0-7.
In our project, we're employing the Grove 4-digit display to show the current time. This involves fetching the current time from the system, formatting it, and then displaying it on the Grove module. Below is a guide on how we achieve this without delving into the specifics of the code.

- **Fetching and Formatting Time**

- **Get Current Time**: We first obtain the current time using the time function, which gives us the time in seconds. This is then converted into a more readable format (struct tm) representing local time.
- **Extract Hours and Minutes**: From the struct tm, we extract the hours (tm_hour) and minutes (tm_min).
- **Format Time String**: We format the time into a string that can be displayed. If hours or minutes are less than 10, we prepend a '0' to keep the display consistent (e.g., "08:05" instead of "8:5").
- **Display on Console**: For debugging or monitoring, we also print the formatted time to the console.

- **Displaying Time on the Grove 4-Digit Display**
  - **Continuous Update Thread**: A dedicated thread continuously updates the display with the current time. This thread runs as long as a flag (running_flag) remains true, allowing the display to be stopped gracefully.
  - **Update Process**:
    - **Start**: A start signal is sent to prepare the display for receiving data.
    - **Address and Data**: We then send a command to auto-increment address locations on the display, followed by the actual time data. The data sent corresponds to the formatted time string.
    - **Brightness and Visibility**: The display's brightness is set to maximum for clear visibility.
    - **Pause**: After updating, the thread pauses for one minute (60000 milliseconds) before fetching the current time again, thus keeping the display updated every minute.
- **Implementation Highlights**
  - The use of a separate thread for the display allows the main program to continue other tasks without interruption.
  - Time formatting accounts for single-digit hours and minutes, ensuring the display is always readable.
  - The display automatically refreshes every minute, making it an ideal digital clock.

7.4. Conclusion

This guide outlines the process of using a Grove 4-digit display with the BeagleBone Green to show the current time. By fetching the system time, formatting it appropriately, and updating the display in a dedicated thread, we create a functional and visually appealing time display. This approach showcases the flexibility and utility of the Grove system in embedded projects.

7.5. Troubleshooting Tips:

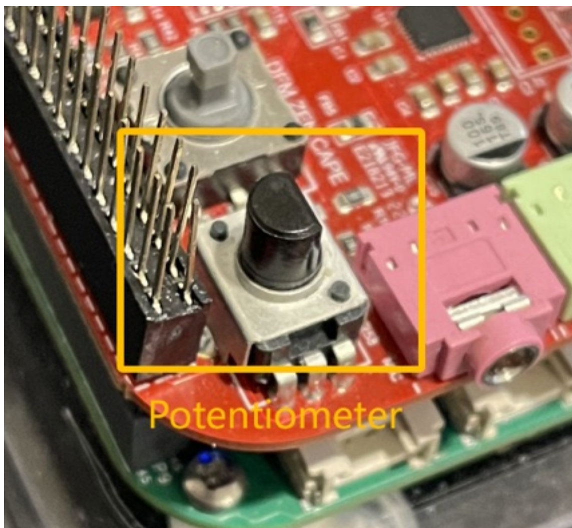
- **No ACK Received:** Ensure the jumper near the buzzer is removed and the GPIO pins are correctly configured.
- **Display Issues:** Verify that you're sending valid digits and the display setting value is correct.

7.6. Detail Guide from:

https://opencoursehub.cs.sfu.ca/bfraser/grav-cms/cmpt433/links/files/2016-student-howtos/Grove4DigitDisplay.pdf

8. Potentiometer - A2D

This guide explains how to control the transition speed of NeoPixels connected to a BeagleBone Green (BBG) by using a potentiometer through analog-to-digital (A2D) conversion in Linux. We'll leverage the Zen cape for an easy connection to the BBG, focusing on adjusting the NeoPixel's transition speed with real-time potentiometer feedback.



8.1. Identify the A2D Channel:

First, determine the analog input channel used by the potentiometer. For the Zen cape, it's connected to AIN0 (analog input 0), accessible via the P9 expansion headers on the BeagleBone.

8.2. Access A2D Readings:

- Navigate to the A2D sysfs directory: /sys/bus/iio/devices/iio:device0. Note that in a Linux command line, the colon (:) in the path needs to be escaped with a backslash (\), like so: iio\:device0.
- Use the cat command to read the voltage value from the potentiometer: (bbg)$ cat in_voltage0_raw
- Note:Replace 0 with the respective channel number if different.

8.3. Reading and Interpretation:

- The output will range between 0 and 4095, representing the potentiometer's position.
- In programming, especially C, ensure the use of appropriate data types to avoid inaccurate calculations due to integer division.

8.4. Adjusting NeoPixel Speed - Calculate Transition Delay:

- Utilize the A2D conversion result to determine the NeoPixel transition speed. The example function getDelayTimeForNeoPixel() reads the potentiometer value and calculates a delay time, which adjusts the speed of NeoPixel transitions.

- The formula int delayTime = 30 + ret / 80; uses the potentiometer's voltage reading (ret) to adjust the delay dynamically. As you turn the potentiometer, ret changes, thus altering delayTime.

8.5. Adjusting NeoPixel Speed - Implementing the Speed Adjustment:
Use the calculated delay time in your NeoPixel control loop to modulate the speed of the LED transitions. As the potentiometer is rotated, the delay time shortens or lengthens, speeding up or slowing down the NeoPixel transitions respectively.

8.6. Conclusion:
By following these steps, you can create an interactive experience with NeoPixels by adjusting their transition speed with a simple turn of a potentiometer. This guide provides a foundation for incorporating analog input devices into your BBG projects, offering a hands-on approach to real-time system control.

8.7. Troubleshooting Tips:

- No Change in Reading: If adjusting the potentiometer doesn't change the in_voltage0_raw value, ensure the Zen cape and BBG are properly connected. Check that none of the P8/P9 pins are visible between the two boards.
- Missing A2D File: If the in_voltage0_raw file is missing, verify that UBoot overlays are enabled (enable_uboot_overlays=1) and the A2D disable command is commented out (#disable_uboot_overlay_adc=1) in /boot/uEnv.txt.

9. Joystick - GPIO

In our project, we are using a joystick attached to the BeagleBone Green (BBG) to control different modes and to safely terminate the program. The joystick inputs are managed through GPIO pins, and we are primarily using the UP and DOWN directions to change modes, while the RIGHT direction is used to exit the program.

9.1. Configure GPIO Pins:
Each direction of the joystick is connected to a specific GPIO pin on the BBG:
UP: P8.14 (GPIO26)
(bbg)$ config-pin p8.14 gpio
DOWN: P8.16 (GPIO46)
(bbg)$ config-pin p8.16 gpio
RIGHT: P8.15 (GPIO47)
(bbg)$ config-pin p8.15 gpio
LEFT: P8.18 (GPIO65)
(bbg)$ config-pin p8.18 gpio
PUSH (IN): P8.17 (GPIO27)
(bbg)$ config-pin p8.17 gpio
Set Pin Directions:

All joystick GPIO pins are set to input mode since they are used to read the joystick's position:
(bbg)$ echo in > /sys/class/gpio/gpio26/direction
(bbg)$ echo in > /sys/class/gpio/gpio47/direction
(bbg)$ echo in > /sys/class/gpio/gpio46/direction
(bbg)$ echo in > /sys/class/gpio/gpio65/direction
(bbg)$ echo in > /sys/class/gpio/gpio27/direction

9.2. Read a GPIO pin value as input: using gpio26 which is the joystick up on Zen cape

- Change to its folder:
  (bbg)$ cd /sys/class/gpio/gpio26
- Read its value:
  (bbg)$ cat value

9.3.Read the input value when Joystick UP Not pressed/pressed

```
debian@lingjiel-beagle:~$ config-pin p8.14 gpio

Current mode for P8_14 is:     gpio

debian@lingjiel-beagle:~$  echo in > /sys/class/gpio/gpio26/direction
debian@lingjiel-beagle:~$ cd /sys/class/gpio/gpio26
debian@lingjiel-beagle:/sys/class/gpio/gpio26$ cat value
1
debian@lingjiel-beagle:/sys/class/gpio/gpio26$ cat value
0
```

9.4. Joystick Thread:

This thread continuously checks the state of the joystick pins and reacts accordingly:

- Mode Change (UP/DOWN): When the UP or DOWN button is pressed, the current mode number is incremented or decremented. This triggers the processMode() function to switch between different display modes such as people count, temperature, and a smiley face.
- Program Termination (RIGHT): Pressing RIGHT stops the program by setting running_flag = 0

9.5. Conclusion:

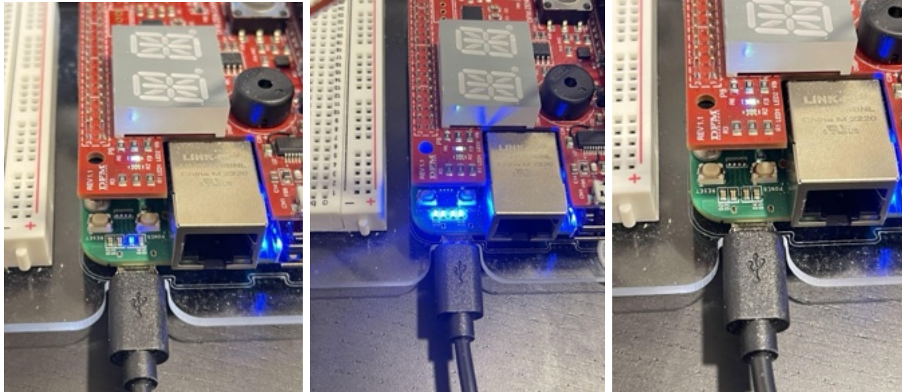
Using a joystick with the BBG offers an intuitive way to interact with your project, allowing for real-time control changes and an easy method to safely terminate the program. By following the steps outlined in this guide, you can implement joystick control for various applications effectively.

9.6. Troubleshooting:

- Debouncing: Implement debouncing either in software or hardware to ensure stable input signals without false triggering.
- Error Handling: Properly handle file opening and reading errors in the isPressed() function to prevent the program from crashing unexpectedly.

10. BBG LEDs blink

In this guide, we will configure the BeagleBone Green (BBG) to make four LEDs blink four times within half a second whenever the joystick is moved up or down. This feature enhances user interaction by providing visual feedback on control inputs.



10.1. Accessing the LED Directory

Enter the /sys/class/leds/ directory to locate controllable LEDs.

    (bbg)$ cd /sys/class/leds

10.2. Viewing and Selecting a Specific LED

List files in a specific LED directory to access control options.
Example: Use ‘ls’ to display files to select a specific LED

    (bbg)$ cd beaglebone\:green\:usr0

10.3. Controlling LED Triggers

View the current trigger settings and select a trigger (e.g., heartbeat, timer, etc.).
View current trigger:

    (bbg)$cat trigger

Change the trigger: echo [trigger] > trigger, for instance echo none > trigger

    (bbg)$ echo none > trigger

10.4. Adjusting LED Brightness

Directly control the LED's on or off state by changing the brightness file.
Turn on LED:

     (bbg)$ echo 1 > brightness

Turn off LED:

     (bbg)$ echo 0 > brightness

10.5.Using C Language to Control LEDs

- Write C programs to open files, write to trigger or brightness values, and properly close files.
- Example: Use fopen() to open a file, fprintf() to write to a file, and fclose() to close the file.

10.6. Alternating Between Command Line and C Language

- For reliable changes in LED brightness, you might need to close and reopen the file each time you want to adjust its brightness.

10.7. Software Configuration

- Initialization: Initialize the GPIO settings for both the joystick and LEDs at the start of your application.
- Joystick Handling: Monitor the joystick's directional movements (UP and DOWN) in a dedicated thread or within the main program loop.

10.8. Blinking Mechanism

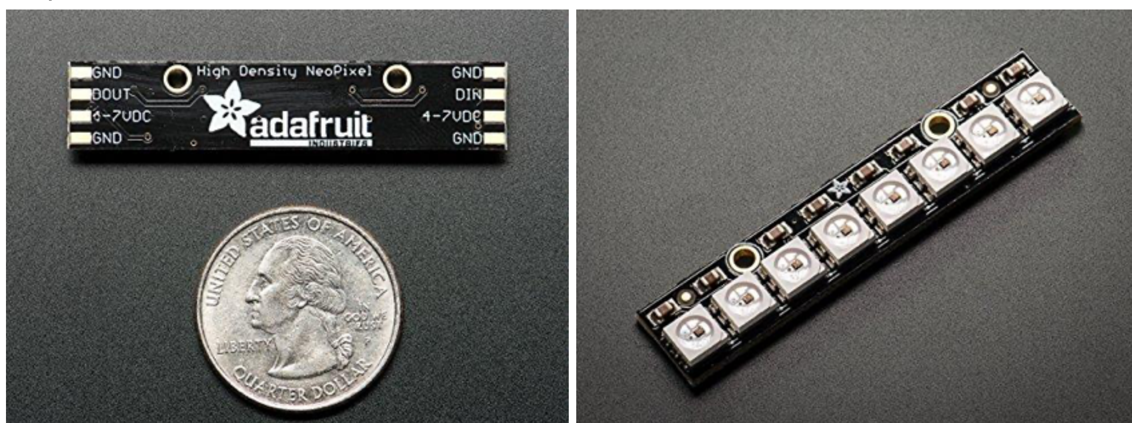
- Function Definition: Create a function blinkLEDs() that:
  - Turn all four LEDs on and off four times.
  - Each cycle (on and off) lasts for 62.5 milliseconds, totaling four blinks in half a second.
- Integration: Call blinkLEDs() whenever an UP or DOWN movement is detected by polling the joystick's GPIO pins.

10.9.Troubleshooting

- LED Non-Responsiveness: Check for correct GPIO pin numbering and ensure that the LEDs are functional.
- Incorrect Blinking Rate: Adjust the values in sleepForMs() if the blinking is too fast or too slow.
- Joystick Sensitivity: Implement debounce logic if the joystick triggers multiple signals for a single press.
- Testing and Validation
  - Functional Test: Move the joystick up and down to validate that the LEDs blink as expected.
  - Performance Monitoring: Observe the timing and consistency of the LED blinking to ensure it meets the project requirements.
- Documentation and References
  - GPIO Control: Refer to the BeagleBone system documentation for detailed instructions on GPIO manipulation.
  - C Code Examples: Review provided C code samples for controlling LEDs and handling GPIO input to model your implementation.

11. NeoPixel - PRU

To create a scrolling effect using the Programmable Real-time Unit (PRU) on a BeagleBone Green to control 8 NeoPixel LEDs, we will need to follow several key steps.



11.1.Build PRU Code and Run it

- On the target, install necessary tools:
  (bbg)$ sudo apt-get install make ti-pru-cgt-v2.3 ti-pru-software-v6.0
- Configure the BBG so the compiler can find pru_cfg.h:
  (bbg)$ cd /usr/lib/ti
  (bbg)$ sudo ln pru-software-support-package-v6.0 pru-software-support-package -s
  This command links the -v6.0 package to the base name.
- Create a folder for the PRU project. Place with your other code, or in this folder:
  (host)$ mkdir -p ~/cmpt433/work/pru/

11.2. Install process

- Put the NeoPixel into a breadboard
- Connect the NeoPixel with 3 wires:
  - Connect NeoPixel "GND" and "DIN" (data in) to the 3-pin "LEDS" header on Zen
    Zen Cape's LEDS header:
    Pin 1: DIN (Data): left most pin; beside USB-micro connection, connects to P8.11
    Pin 2: GND (Ground): middle pin
    Pin 3: Unused (it's "5V external power", which is not powered normally on the BBG)
  - Connect NeoPixel "5VDC" to P9.7 or P9.8
    Suggest using the header-extender to make it easier to make a good connection.
- On Host
  (host)$ make   # on parent folder to copy to NFS
- On Target
  (bbg)$ config-pin P8.11 pruout
  (bbg)$ make
  (bbg)$ make install_PRU0

11.3. Code Review

Explain the purpose of each included file and their roles within the project:

- sharedDataStruct.h: Defines shared data structures.
- neoPixel.h: Declares functions related to NeoPixel control.
- utils.h: Utility functions such as delay.

11.4. Detailed Function Explanations

1. Initialization (neoPixel_init):
   Describe how this function initializes the NeoPixel thread and configures the pin.
2. Memory Mapping (getPruMmapAddr and freePruMmapAddr):
   Explain the process of mapping and unmapping PRU memory to enable direct access from the Linux application.
3. Thread Logic (neoPixelThread):

Discuss the looping structure that checks color_flag and adjusts the NeoPixel colors accordingly.

4. Color Control (setPixelColor):
   Detail how different colors are set based on the color_flag and
peopleCount variables, describing how the LED colors and patterns are adjusted.
5. Utility Functions:
   clearStrip(): Clears all LEDs.
6. flashTeal() and flashBlue(): Utility functions to trigger color changes when detect someone comes in/out.

12. UDP - Network

12.1. Overview of a UDP Server

A UDP (User Datagram Protocol) server is a network application that handles requests and sends responses using the UDP protocol, which is connectionless and often faster than TCP (Transmission Control Protocol) due to its non-reliant nature on handshakes before data transmission. This makes UDP ideal for real-time applications where speed is crucial and occasional data loss is tolerable.

12.2.Components of the UDP Server

- Initialization: The server starts by initializing its environment. This includes setting up the UDP socket, specifying the IP address and port it will listen on, and binding the socket to the address.
-
```
#define PORT 12345 // The port the server will listen on
#define BUFFER_MAX_SIZE 1000
```
-
```
int sockfd;
struct sockaddr_in servaddr, cliaddr;
```
-
```
static void setupUDPSocket(void)
{
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(PORT);

    //create socket
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0)
    {
        perror("Failed to create socket!\n");
        exit(1);
    }

    //bind
    if (bind(sockfd, (const struct sockaddr *)&servaddr, sizeof(servaddr)) < 0)
    {
        perror("Binding failed!\n");
        exit(1);
    }
}
```

- Socket Management: Essential for receiving and sending datagrams (packets). Functions are typically dedicated to opening, closing, and managing the state of the socket.

```
void UDP_cleanup(void)
{
    close(sockfd);
}
```

```
static int receivePacket(char *packet)
{
    unsigned int sin_len = sizeof(cliaddr);
    int bytesRx = recvfrom(sockfd, packet, BUFFER_MAX_SIZE - 1, 0, (struct sockaddr *) &cliaddr, &sin_len);

    if (bytesRx < 0)
    {
        perror("Failed to receive packet.\n");
        exit(1);
    }
    return bytesRx;
}

static int sendPacket(char *packet)
{
    size_t packet_len = strlen(packet);

    unsigned int sin_len = sizeof(cliaddr);
    int byteTx = sendto(sockfd, (const char *)packet, packet_len, 0, (const struct sockaddr *) &cliaddr, sin_len);

    if (byteTx < 0)
    {
        perror("Failed to send packet.\n");
        exit(1);
    }
    return byteTx;
}
```

- Main Server Loop: The core of the server that continuously listens for incoming datagrams, processes them, and decides on appropriate responses or actions based on the content of the datagrams.
- Command Processing: Involves parsing the received data to understand the intent (commands) and executing corresponding actions such as fetching data, modifying system settings, or triggering other functionalities.
- Response Handling: After processing the commands, the server sends responses back to the client. This might include status information, results of a request, or acknowledgments.
- Resource Management: Proper handling of system resources, ensuring that the server efficiently manages memory, handles exceptions, and cleans up resources when shutting down or restarting.

12.3. Operational Logic

- **Server Initialization**: On startup, the server configures its environment, prepares necessary resources, and initializes the network socket.
- **Listening for Requests**: The server enters a loop where it listens for incoming UDP packets. When a packet is received, it's processed to determine the type of request.

- **Command Routing**: Each packet may contain a command that needs to be routed to the appropriate handler function. This function interprets the command and performs the necessary operations.
- **Sending Responses**: For each processed command, a response is generated and sent back to the client, providing feedback or requested data.
- **Repeating Commands**: The server may have functionality to remember the last executed command and repeat it if prompted by the user, enhancing interaction efficiency.
- **Handling Errors and Exceptions**: Includes monitoring for potential errors in network communication or internal processing and managing them appropriately to maintain server stability.
- **Server Shutdown**: Includes properly releasing resources, notifying clients or connected services of the shutdown, and closing the network socket.

13. Node.js Web Interface

To allow web on beaglebone we first need to install nodejs and the corresponding management tools such as npm and nvm, and then we can allow our web application.

13.1 Upgrade Node Version

- First, make sure your own system is up to date. And install node package manager and node.js

```
(bbg)$ sudo apt update
(bbg)$ sudo apt upgrade
(bbg)$ sudo apt install npm # Installs ~500MB on BBG
(bbg)$ sudo npm cache clean -f
```

13.2 Install Node Version Manager (NVM)

```
(bbg)$ curl
-o-https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.1/install.sh | bash
```

13.3 Install Node.js

```
(bbg)$ sudo npm install -g n
(bbg)$ sudo n stable # Updates Node.js to v20.11 (Feb 2024)
```

- Or install Node.js using NVM. You can choose to install the latest version or a specific version of Node.js. To install the latest version, run:

```
(bbg)$ nvm install node
```

- After the installation is complete, check the installed versions of Node.js and npm through the following commands.

```
(bbg)$ node -v
(bbg)$ npm -v
```

13.4 Run our project server on BBG

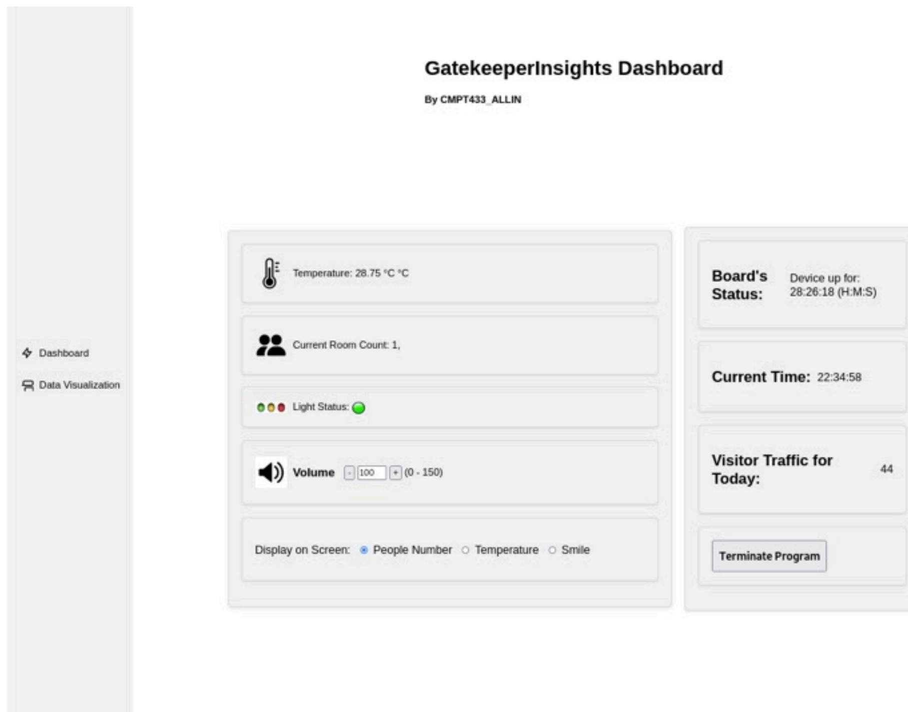
- Next step cd into our project folder and then install the packages with dependencies in package.json

```
(bbg)$ npm install
```

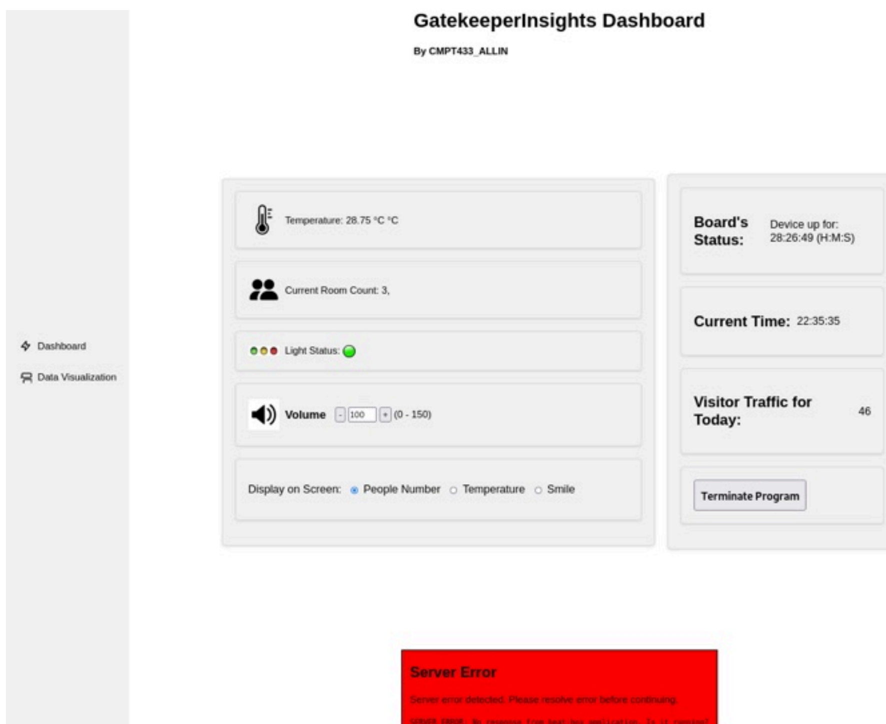
- Finally allow the server in the terminal.
  (bbg)$ node server.js
- Congratulations, you can enter http://192.168.7.2:8080/ in the URL through the firefox browser to view our project web home page.

Successful running:

192.168.7.2:8080/status/index.html



If connection is failed:



We store the head count data in indexedDB for data visualization. Since our test data was insufficient to produce a weekly average chart, we used random data to optimize the display.