How to control Adafruit NeoTrellis 4x4 Led Button Matrix from C code.

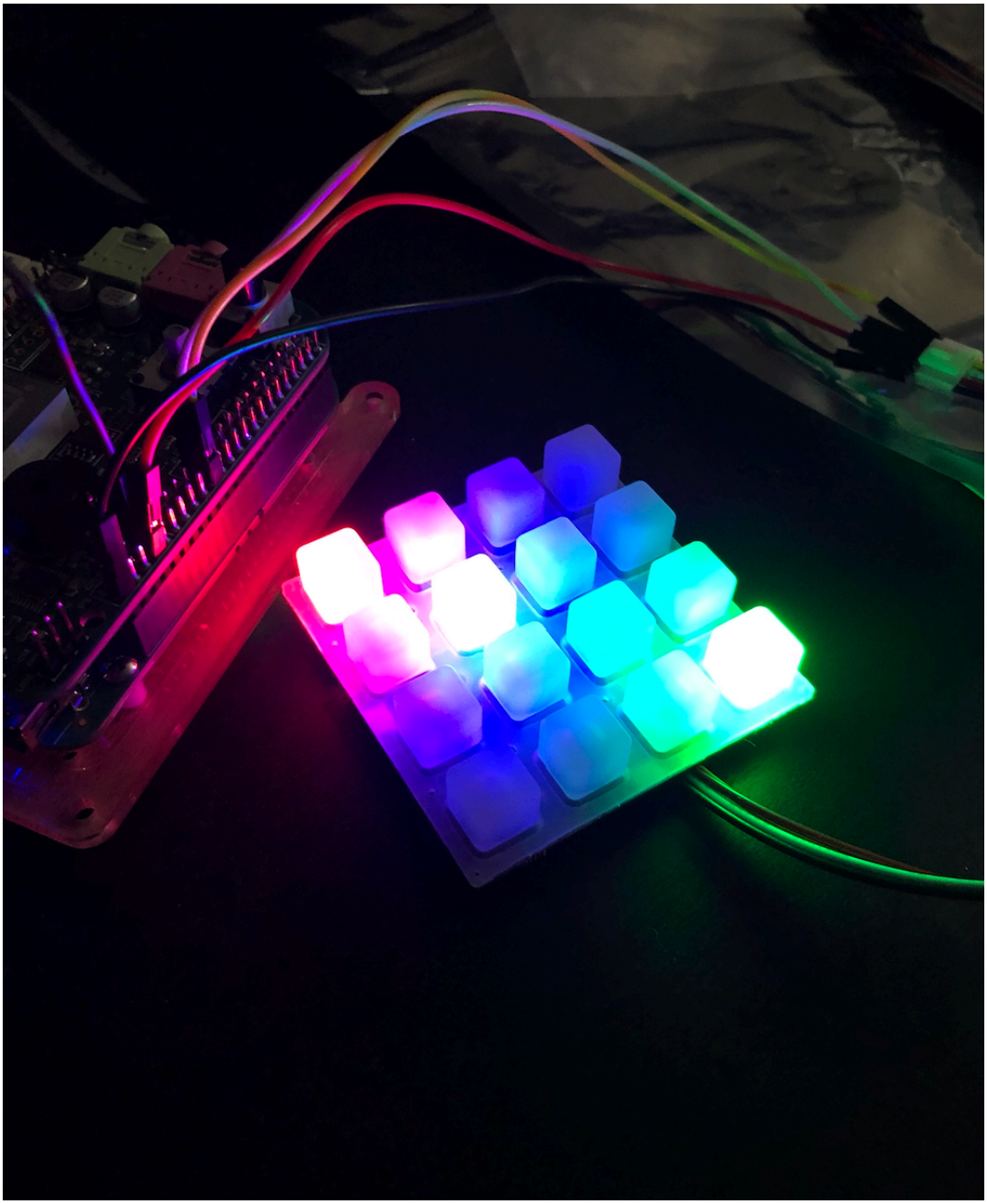


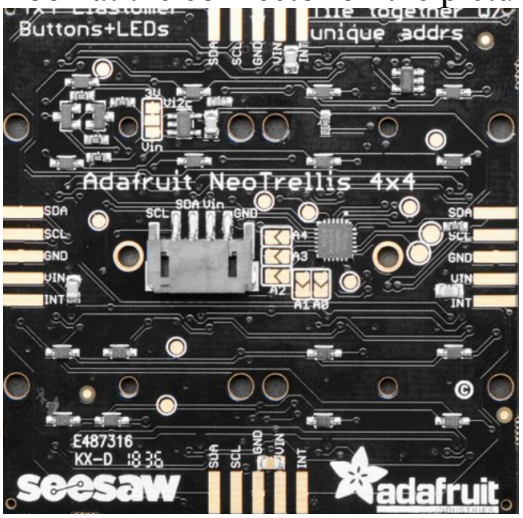
Table of contents

1. Initial Setup

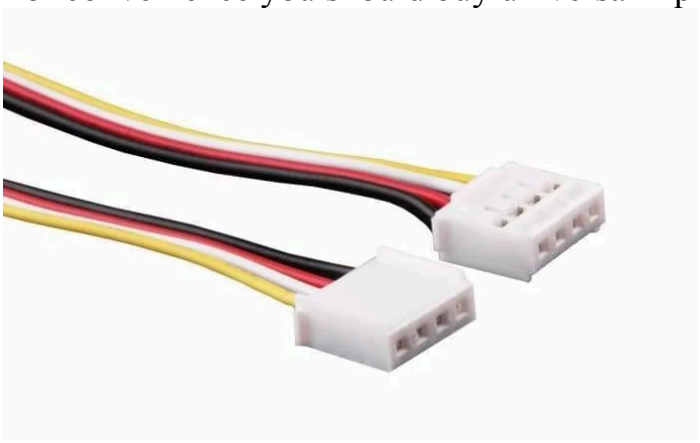
1.1 How to wire up led matrix

To be able to control the matrix we need to wire it up correctly first. Matrix has 4 main pins: ground(**GND**), power(**Vin**), Data Pin(**SDA)**, Clock Pin(**SCL**). We need to connect all those pins correctly for the matrix to function properly.

Look at the connector on the picture to see all the required pins:



For convenience you should buy universal 4 pin female to female cable:



It will simplify the wiring process.

Now we need to identify the required pins on the board. To do so, let's have a look at the beaglebone scheme:

P9

DGND	1	2	DGND
VDD_3V3	3	4	VDD_3V3
VDD_5V	5	6	VDD_5V
SYS_5V	7	8	SYS_5V
PWR_BUT	9	10	SYS_RESETn
GPIO_30	11	12	GPIO_60
GPIO_31	13	14	GPIO_50
GPIO_48	15	16	GPIO_51
GPIO_5	17	18	GPIO_4
I2C2_SCL	19	20	I2C2_SDA
GPIO_3	21	22	GPIO_2
GPIO_49	23	24	GPIO_15
GPIO_117	25	26	GPIO_14
GPIO_115	27	28	GPIO_113
GPIO_111	29	30	GPIO_112
GPIO_110	31	32	VDD_ADC
AIN4	33	34	GNDA_ADC
AIN6	35	36	AIN5
AIN2	37	38	AIN3
AIN0	39	40	AIN1
GPIO_20	41	42	GPIO_7
DGND	43	44	DGND
DGND	45	46	DGND

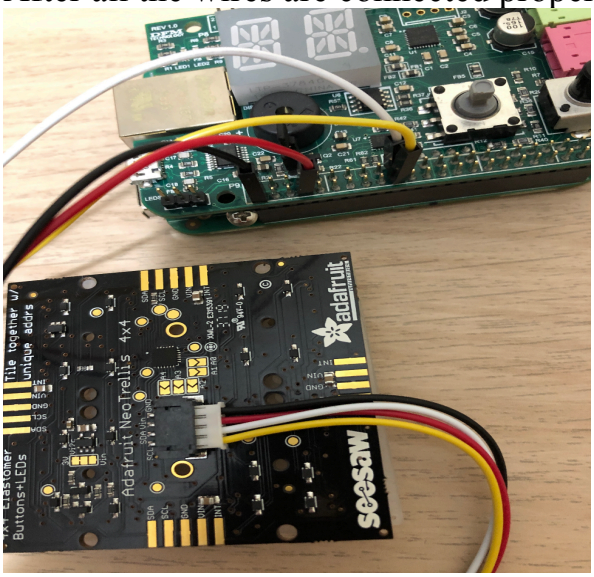
P8

DGND	1	2	DGND
GPIO_38	3	4	GPIO_39
GPIO_34	5	6	GPIO_35
GPIO_66	7	8	GPIO_67
GPIO_69	9	10	GPIO_68
GPIO_45	11	12	GPIO_44
GPIO_23	13	14	GPIO_26
GPIO_47	15	16	GPIO_46
GPIO_27	17	18	GPIO_65
GPIO_22	19	20	GPIO_63
GPIO_62	21	22	GPIO_37
GPIO_36	23	24	GPIO_33
GPIO_32	25	26	GPIO_61
GPIO_86	27	28	GPIO_88
GPIO_87	29	30	GPIO_89
GPIO_10	31	32	GPIO_11
GPIO_9	33	34	GPIO_81
GPIO_8	35	36	GPIO_80
GPIO_78	37	38	GPIO_79
GPIO_76	39	40	GPIO_77
GPIO_74	41	42	GPIO_75
GPIO_72	43	44	GPIO_73
GPIO_70	45	46	GPIO_71

We will use the following pins:

1. **GND(P9-1) or any other DGND pin**
2. **Vin(P9-7) or any other SYS_5V pin**
3. **SDA(P9-19)**
4. **SCL(P9-20)**

After all the wires are connected properly it should look like this:



1.2 How to set the required pins to I2C mode

and verify the connection

After we have correctly wired up the matrix we need to set the **SDA** and **SCL** pins to **I2C** mode.
To do this follow these steps:

1. SSH into the board.
2. **config-pin p9_19 i2c**
   config-pin p9_20 i2c

1.3 Verify the connection

After the pins are configured we can see that the device is connected to the board by using **i2cdetect –y –r 2**. Our device should be at the adress **0x2E**. Example:

```
debian@beagle:~$ config-pin p9_19 i2c
debian@beagle:~$ config-pin p9_20 i2c
debian@beagle:~$ i2cdetect -y -r 2
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- 18 -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- 2e --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- UU UU UU UU -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

We can get the dump of the registers of the device by running **i2cdump –y 2 0x2E:**

```
debian@beagle:~$ i2cdump -y 2 0x2E
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 ff ff ff ff ff ff ff ff ff ff ff ff ff XX    ................X
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
50: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
60: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX    XXXXXXXXXXXXXXXX
```

If terminal says that there are no such commands, you'll need to install i2c-tools package. Just run: **sudo apt-get install i2c-tools**
If it doesn't work, see **3. Troubleshooting**

After that we're all set and ready to write some code.

2. Designing basic API to control the matrix

2.1 High Level Overview

To be able to control the matrix effectively we will need to implement several components:

1) **i2cdevice** – it will contain methods to access connected through i2c devices
2) **smbus** – wrapper for writing and reading from i2c bus
3) **seesaw** – abstraction level to control the chip of the matrix
4) **trellis** – abstraction level to control the LED matrix from user space

2.2 Writing API to access i2cdevice

To access a connected i2c device we need to implement two components:

1) **smbus** (for description see previous section)
2) **i2cdevice** (for description see previous section)

Basic functions of smbus is to write and read from i2cbus. To do that it needs to know two things: **which bus to use** and **which device on that bus to read from/write to**. So, the interface for the **smbus** can look like this:

```
// SMBus protocol tool module header
#ifndef _SMBUS_HEADER_
#define _SBMUS_HEADER_


void initializeSMBus(const char* bus);
void closeSMBus();

int writeTo(unsigned int deviceAddress, void* buf, size_t size);
int readFrom(unsigned int deviceAddress, void* buf, size_t size);


#endif
```

Now let's have a look at each of those functions.

```
#include <linux/i2c-dev.h>

#include "smbus.h"


int i2c_bus;

void initializeSMBus(const char* bus) {
    i2c_bus = open(bus, O_RDWR);

    if (i2c_bus < 0) {
        printf("Error openning i2c-bus: %s\n", bus);
        exit(1);
    }
}
```

initializeSMBus will take path to the "file" used to communicate with the bus and open this file. The path for the bus used in this example is **/dev/i2c-2**.
We will save the file descriptor to be able to later write to this file without opening it again.

After we will be done with using **smbus**, we will need to close the file:

```
void closeSMBus() {
    close(i2c_bus);
}
```

Before any communication is done with the I2C device, we will need to set it to slave mode. For that we will issue a special **sys call** to the I2C bus driver:

```
static void setDeviceToSlaveMode(unsigned int deviceAddress) {
    if (ioctl(i2c_bus, I2C_SLAVE, deviceAddress & 0x7F) != 0) {
        printf("Error setting i2c-bus to slave mode\n");
    }
}
```

I2C_SLAVE is defined in **i2c-dev.h** file and expands to **0x0703**.
Now let's have a look at writing and reading functions:

```
int writeTo(unsigned int deviceAddress, void* buffer, size_t len) {
    setDeviceToSlaveMode(deviceAddress);
    int bytesWritten = write(i2c_bus, buffer, len);
    return bytesWritten;
}

int readFrom(unsigned int deviceAddress, void* buf, size_t len) {
    setDeviceToSlaveMode(deviceAddress);
    int bytesRead = read(i2c_bus, buf, len);

    return bytesRead;
}
```

writeTo and **readFrom** will both take I2C device address (**0x2E** in our case), buffer to write to/read from and length of the buffer. It first sets device to slave mode and then reads from/writes to that device.

Once we have finished **smbus**, we can implement **i2cdevice** so we can communicate with i2c devices easily. The interface can look like this:

```
// I2C Wrapper Tool module header

#ifndef _I2CDEVICE_H
#define _I2CDEVICE_H

#include <unistd.h>


int i2cWrite(void* buffer, size_t length);
int i2cRead(void* buffer, size_t length);

void initializeI2C(const char* bus, unsigned int deviceAddress);
void destroyI2C();


#endif
```

Same as in smbus we have write and read functions, initialization and destruction functions. We will use **smbus** module inside **i2cdevice** to write to and read from the actual device.

The code for it is very straightforward and implementation is provided together with this tutorial. The only point I want to make is automatic pin configuration when i2cdevice is initialized:

```c
#define CONFIG_PIN_COMMAND "config-pin"
#define I2C2_SCL_PIN "P9_19"
#define I2C2_SDA_PIN "P9_20"
#define I2C_MODE "i2c"
#define SPACE " "

unsigned int device;


static void configureI2CPins() {
    system(CONFIG_PIN_COMMAND SPACE I2C2_SDA_PIN SPACE I2C_MODE);
    system(CONFIG_PIN_COMMAND SPACE I2C2_SCL_PIN SPACE I2C_MODE);
    sleepMS(500);
}
```

It simplifies the pin configuration so there's no need to do it manually every time.

2.3 Implementing control mechanism for the LEDs of the matrix

After we have finished implementing **i2cdevice** interface, we can start designing our API to control the matrix. We'll start with implementing **seesaw** module that will allow us to create an abstraction of communication with the matrix chip.

Interface can look like this:

```c
// Low level API to the Seesaw framework based on ATSAMD09 chip

#ifndef _SEESAW_H_
#define _SEESAW_H

#include <stdio.h>

void initializeSeeSawDevice(const char* bus, unsigned int deviceAddress);

// Seesaw write: writes to the chip
void ssWrite(size_t n, ...);
int ssRead(unsigned char base, unsigned char reg, unsigned char* buf, size_t n);
unsigned char ssRead8(unsigned char base, unsigned char reg);
void destroySeeSawDevice();


#endif
```

Same as before we have initialization and destruction of the module, writing and reading from the device.

Seesaw module will be abstraction for writing and reading to the matrix registers. It will use **i2cdevice** for communication with the matrix.
There are several commands for the matrix this module is implementing but the most important one is **resetChip** and **ssRead**.

```
#define SEESAW_CHIP_DEFAULT_ADDRESS 0x2E

#define STATUS_BASE 0x00
#define STATUS_SWRST 0x7F
#define STATUS_RESET 0xff
```

```
// cleans all the registers
// doesn't turn off LEDs though: have to do it manually
static void resetChip()  {
    ssWrite(3, STATUS_BASE, STATUS_SWRST, STATUS_RESET);
    sleepMS(500);
}

int ssRead(unsigned char base, unsigned char reg, unsigned char* buf, size_t n) {
    ssWrite(2, base, reg);
    int bytesRead = i2cRead(buf, n);

    return bytesRead;
}
```

Reset chip function will clear all the registers of the chip, literally resetting it **BUT** it will not write anything to the “activation” registers to light up or turn off the LEDs. This function has to be called after initialization of the device was completed to wipe out all the previous value from the registers (in case there are some).

ssRead function will read from the device file, but do so we first need to write what register we want to read from. We will pass the base register for the component and **offset** indicating the register we want to read from. And only then can we actually read. The rest of the code can be found together with this tutorial.

After we have finished our **seesaw** module to do basic write/read operations, we can get to implementing the **trellis** module which will provide interface for controlling LEDs of the matrix.

To be able to control the LEDs of Adafruit Matrix, we need to know some basic registers we want to write to and initial setup we must do to make it work. Interface of trellis module can look like this:

```
// Trellis platform tools module header
#ifndef _TRELLIS_H_
#define _TRELLIS_H_


#include <stdio.h>
#include "controls.h"
#include "colors.h"

// When setting pixel use GRB

void initializeTrellis(const char* bus, unsigned char deviceAddress);
void setPixelColor(int index, color color);
bool compareColors(color c1, color c2);
// Change naming
void pixelOn();
void pixelOff(int index);
void allPixelsOff();
void blinkAtIndex(int times, int delay, int index, color c);
void destroyTrellis();
#endif
```

Trellis will use **seesaw** for communication with the matrix.

When we initialize trellis module, we need to do some setup on the matrix before we can control it:

1) Initialize **seesaw** device.
2) Set matrix buffer size.
3) Set push event (this is done to register buttons)
4) Turn on the pixels on (it will turn the pixels on with the wiped old colors, so even if any of the pixels were turned on, they all will be off due to resetChip in seesaw).

```
void initializeTrellis(const char* bus, unsigned char deviceAddress) {
    initializeSeeSawDevice(bus, deviceAddress);
    setBufferSize();
    enablePushEventForAll();
    pixelOn();
}
```

Registers that we will need for all the required commands:

```
#define TRELLIS_BASE 0x0E
#define TRELLIS_STATUS 0x00
#define TRELLIS_PIN 0x01
#define TRELLIS_BUF_LEN 0x03
#define TRELLIS_BUF 0x04
#define TRELLIS_SHOW 0x05
#define TRELLIS_EVENT_RISING_EDGE 0x11
#define TRELLIS_EVENT_FALLING_EDGE 0x09
```

You can find the code attached to this tutorial.
After we're done with the setup, we can play with the lights. To turn on specific LED:

```
// index - 1, if index is 1-ordered
void setPixelColor(int index, color c) {
    ssWrite(7, TRELLIS_BASE, TRELLIS_BUF, 0x00, index * 3, c.g, c.r, c.b);
}
```

0x00 + index * 3 (3 is offset for **g, r, b**) is the start address of the **LED** buffer that contains color.
And then we have to call **pixelOn()**, so matrix will display the new color of LED.

2.4 Implementing control mechanisms for the Buttons of the matrix

To be able to identify which LED button the user has pushed, we need to enable push events on the matrix:

```
void setEventForPixel(int index, unsigned char edge, bool enable) {
    unsigned char eventKey = getEventKey(index);
    // Can't really turn it off with OR(|)
    // I think need to toggle last bit of edge ot 0 or 1
    // depending on enable
    ssWrite(4,KEYPAD_BASE, KEYPAD_EVENT, eventKey, edge | enable);
}
```

```
setEventForAll(TRELLIS_EVENT_RISING_EDGE, true);
```

So, we have three options on how to register user's click on the LED button:

1) Button down
2) Button release
3) Button down and button release both

To only detect button down, we have to call **setEventForPixel**, pass LED index and **TRELLIS_EVENT_RISING_EDGE**. Rising edge means that when we push down, we connect the circuit and current starts flowing so we record the push. If we want to register on button release we have to pass **TRELLIS_EVENT_FALLING_EDGE**. And if we want both, we simply register both events one after another.

To be able to read the events from the board, you can use this function:

```
int getNextPushedButtonIndex(int* exit_sig) {
    unsigned char events[DEFAULT_EVENT_NUMBER] = {0x00};
    // PROBABLY WILL HAVE TO MOVE CLEANING PHASE OUTSIDE
    // AND DO IT BEFORE TURN PIXEL IS BLINKING
    // TO AVOID POSSIBLE INFINITE CLEANING WHEN USER
    // SIMPLY KEEPS PRESSING BUTTON INFINITELY
    // but it seems to work super fast
    // so user can't really make it happen
    removePreviousEvents(events);


    while (true && *exit_sig != 1) {
        int count = readEventCount();
        if (count != 0) {
            // + 2 just in case (idk:D)
            // printf("reading board\n");
            int bytesRead = readBoard(events, getMin(count + 2, DEFAULT_EVENT_NUMBER));
            for (int i = 0; i < bytesRead; i++) {
                // Check trellis file in blinka library for reference
                unsigned eventNumber = getSeeSawKey((events[i] >> 2) & 0x3F);
                int edge = events[i] & 0x3;
                if (eventNumber < DEFAULT_TRELLIS_SIZE) {
                    printf("%d event\n", eventNumber);
                    return eventNumber;
                }

            }
        }
        sleepMS(5);
    }

    return -1;
}
```

It will read the event count from the board, and if it's greater then 0, then we will read all the events that happened. After we do it, we will have to convert that data to index of the button that was pushed using **getSeeSawKey:**

```
static int getSeeSawKey(val) {
    return val / 8 * 4 + val % 8;
}
```

I can't say why the formula is this or why we must do (**events >> 2) & 0x3F** to make sense of the data received, but that's the way to do it. You can get more information about the chip and its registers here: https://learn.adafruit.com/adafruit-seesaw-atsamd09-breakout?view=all.

3. Troubleshooting

If after completion of all the steps in 1.1 and 1.2 you don't see the same result as demonstrated in 1.3 you may try to do this:

1) Check all the wiring in 1.1 and commands in 1.2. Make sure that power and ground pins are functioning correctly.
2) If 1) doesn't help, then see if there's any current running in the matrix circuit. Use Ammeter (ampere meter).
3) If circuit is functioning correctly (current is running) and all the pins are set correctly, then there may be something wrong with the **SDA** and **SCL** pins. BeagleBone has another **SDA** and **SCL** pair just above the ones we used. **SDA pin: P9_18** and **SCL pin: P9_17**. These pins are used to communicate over **i2c1** bus. Follow the same steps from 1.2 but use **17** and **18** pins and **i2c1** bus.

4. References

1. Trellis picture in 1.1 - https://cdn-shop.adafruit.com/970x728/3954-05.jpg.
2. Grove cable picture in 1.1 - https://cdn.shopify.com/s/files/1/2311/3697/products/grove-universal-4-pin-to-beaglebone-blue-6-female-jstsh-conversion-cable-10-pack-seeed-cool-components_477_799x599.jpg?v=1537318408
3. Information about the Seesaw chip (ATSAMD09) - https://learn.adafruit.com/adafruit-seesaw-atsamd09-breakout?view=all
4. ATSAMD09 data sheet – http://ww1.microchip.com/downloads/en/devicedoc/atmel-42414-sam-d09_datasheet.pdf