2020F_ESE_3014_1

SEMESTER: 3rd SEM

INSTRUCTOR: Prof. Linchen Wang

LAB 5

SUBMISSION DATE: 1st Nov, 2020

NAME AND ID:

VISHAL HASRAJANI (C0761544)

Goutham Reddy Alugubelly (C0747981)

PARTH PATEL (C0764929)

INTRODUCTION:

Spi is a serial and full duplex protocol which is used for short distance communication,especially in embedded systems.Spi communication is possible using 4 connections (i.e Serial clock (SCLK), Serial data output (SDO), Serial data input (SDI), and Slave select (SS)).

In this lab, we will set up and achieve SPI interfacing and communication for BBB as master and ADXL345 Accelerometer as slave.

DESCRIPTION :

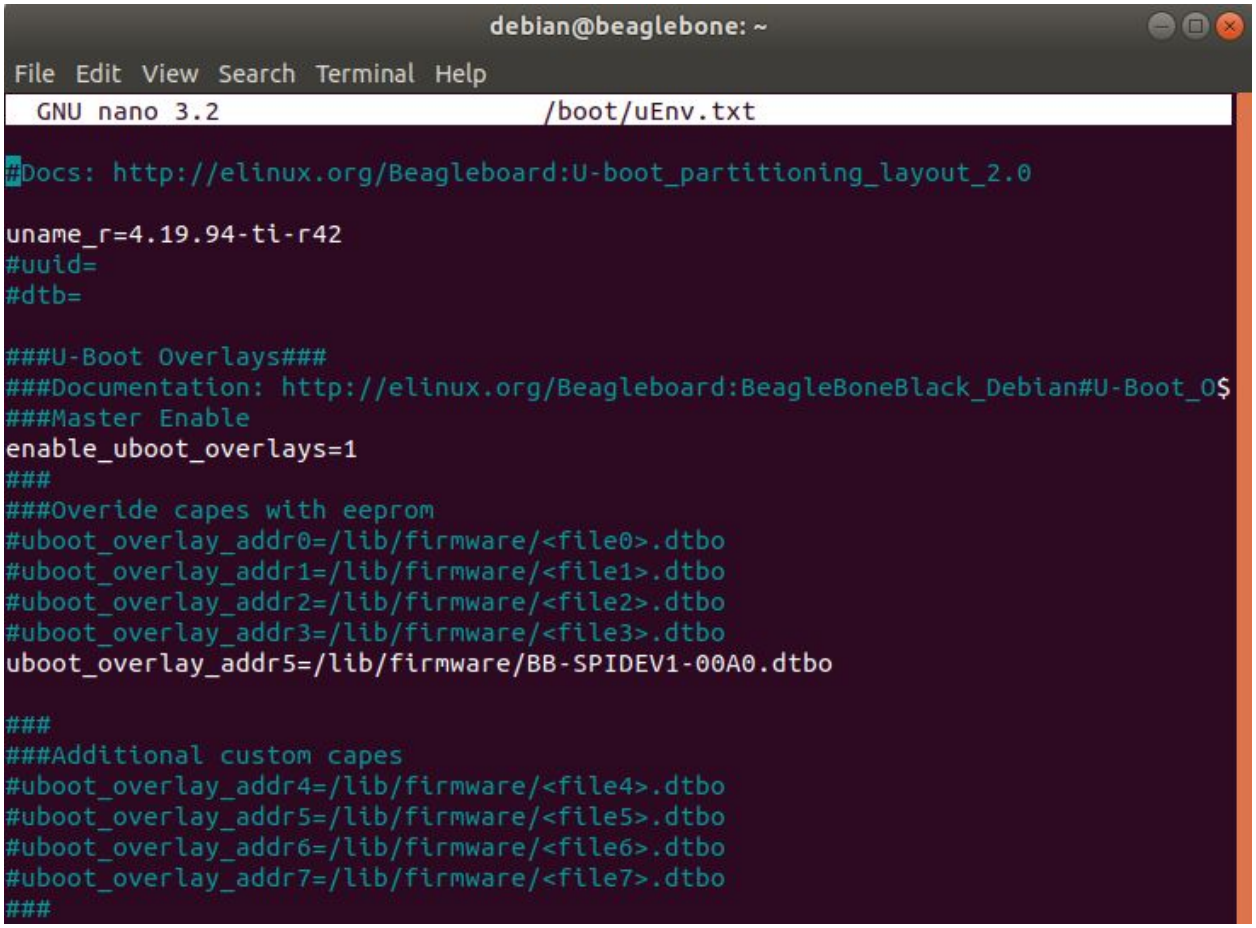
At first, we enabled the spi1 pins for the beaglebone using the below given commands on the terminal .

1) Open uEnv.txt.

```
$ sudo nano /boot/uEnv.txt
```

2) Enable SPI Pins.

```
uboot_overlay_addr5=/lib/firmware/BB-SPIDEV1-00A0.dtbo
```



So now we can use the pins chip select,MOSI,MISO and clock from the table given below :

	BBB SPI0	BBB SPI1
Chip Select	P9_17	P9_28
MOSI	P9_18	P9_29
MISO	P9_21	P9_30
Clock	P9_22	P9_31

Here we enabled BBB SPI1 so we can use pins 28 29 30 and 31.

To check whether these pins are enabled **or not** we used the code (i.e spidev _ test.c)suggested by Derek Molloy by directly connecting 29th **and** 30th pins that are MOSI **and** MISO respectively .

Code :

```
// SPDX-License-Identifier: GPL-2.0-only
/*
 * SPI testing utility (using spidev driver)
 *
 * Copyright (c) 2007  MontaVista Software, Inc.
 * Copyright (c) 2007  Anton Vorontsov
<avorontsov@ru.mvista.com>
 *
 * Cross-compile with cross-gcc
-I/path/to/cross-kernel/include
 */

#include <stdint.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <getopt.h>
#include <fcntl.h>
#include <time.h>
#include <sys/ioctl.h>
#include <linux/ioctl.h>
#include <sys/stat.h>
#include <linux/types.h>
#include <linux/spi/spidev.h>
```

```c
#define ARRAY_SIZE(a) (sizeof(a) / sizeof((a)[0]))

static void pabort(const char *s)
{
	if (errno != 0)
		perror(s);
	else
		printf("%s\n", s);

	abort();
}

static const char *device = "/dev/spidev1.1";
static uint32_t mode;
static uint8_t bits = 8;
static char *input_file;
static char *output_file;
static uint32_t speed = 500000;
static uint16_t delay;
static int verbose;
static int transfer_size;
static int iterations;
static int interval = 5; /* interval in seconds for showing
transfer rate */

static uint8_t default_tx[] = {
	0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
	0x40, 0x00, 0x00, 0x00, 0x00, 0x95,
	0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
	0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
	0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
	0xF0, 0x0D,
```

```
};

static uint8_t default_rx[ARRAY_SIZE(default_tx)] = {0, };
static char *input_tx;

static void hex_dump(const void *src, size_t length, size_t
line_size,
                char *prefix)
{
     int i = 0;
     const unsigned char *address = src;
     const unsigned char *line = address;
     unsigned char c;

     printf("%s | ", prefix);
     while (length-- > 0) {
          printf("%02X ", *address++);
          if (!(++i % line_size) || (length == 0 && i %
line_size)) {
               if (length == 0) {
                    while (i++ % line_size)
                         printf("__ ");
               }
               printf(" |");
               while (line < address) {
                    c = *line++;
                    printf("%c", (c < 32 || c > 126) ? '.' :
c);
               }
               printf("|\n");
               if (length > 0)
                    printf("%s | ", prefix);
          }
```

```
    }
}

/*
 *  Unescape - process hexadecimal escape character
 *      converts shell input "\x23" -> 0x23
 */
static int unescape(char *_dst, char *_src, size_t len)
{
    int ret = 0;
    int match;
    char *src = _src;
    char *dst = _dst;
    unsigned int ch;

    while (*src) {
        if (*src == '\\' && *(src+1) == 'x') {
            match = sscanf(src + 2, "%2x", &ch);
            if (!match)
                pabort("malformed input string");

            src += 4;
            *dst++ = (unsigned char)ch;
        } else {
            *dst++ = *src++;
        }
        ret++;
    }
    return ret;
}

static void transfer(int fd, uint8_t const *tx, uint8_t
const *rx, size_t len)
```

```
{
     int ret;
     int out_fd;
     struct spi_ioc_transfer tr = {
          .tx_buf = (unsigned long)tx,
          .rx_buf = (unsigned long)rx,
          .len = len,
          .delay_usecs = delay,
          .speed_hz = speed,
          .bits_per_word = bits,
     };

     if (mode & SPI_TX_OCTAL)
          tr.tx_nbits = 8;
     else if (mode & SPI_TX_QUAD)
          tr.tx_nbits = 4;
     else if (mode & SPI_TX_DUAL)
          tr.tx_nbits = 2;
     if (mode & SPI_RX_OCTAL)
          tr.rx_nbits = 8;
     else if (mode & SPI_RX_QUAD)
          tr.rx_nbits = 4;
     else if (mode & SPI_RX_DUAL)
          tr.rx_nbits = 2;
     if (!(mode & SPI_LOOP)) {
          if (mode & (SPI_TX_OCTAL | SPI_TX_QUAD |
SPI_TX_DUAL))
               tr.rx_buf = 0;
          else if (mode & (SPI_RX_OCTAL | SPI_RX_QUAD |
SPI_RX_DUAL))
               tr.tx_buf = 0;
     }
```

```
	ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
	if (ret < 1)
		pabort("can't send spi message");

	if (verbose)
		hex_dump(tx, len, 32, "TX");

	if (output_file) {
		out_fd = open(output_file, O_WRONLY | O_CREAT |
O_TRUNC, 0666);
		if (out_fd < 0)
			pabort("could not open output file");

		ret = write(out_fd, rx, len);
		if (ret != len)
			pabort("not all bytes written to output
file");

		close(out_fd);
	}

	if (verbose)
		hex_dump(rx, len, 32, "RX");
}

static void print_usage(const char *prog)
{
	printf("Usage: %s [-DsbdlHOLC3vpNR24SI]\n", prog);
	puts("  -D --device   device to use (default
/dev/spidev1.1)\n"
	     "  -s --speed    max speed (Hz)\n"
	     "  -d --delay    delay (usec)\n"
	     "  -b --bpw      bits per word\n"
```

```
"  -i --input    input data from a file (e.g.
\"test.bin\")\n"
"  -o --output   output data to a file (e.g.
\"results.bin\")\n"
"  -l --loop     loopback\n"
"  -H --cpha     clock phase\n"
"  -O --cpol     clock polarity\n"
"  -L --lsb      least significant bit first\n"
"  -C --cs-high  chip select active high\n"
"  -3 --3wire    SI/SO signals shared\n"
"  -v --verbose  Verbose (show tx buffer)\n"
"  -p            Send data (e.g.
\"1234\\xde\\xad\")\n"
"  -N --no-cs    no chip select\n"
"  -R --ready    slave pulls low to pause\n"
"  -2 --dual     dual transfer\n"
"  -4 --quad     quad transfer\n"
"  -8 --octal    octal transfer\n"
"  -S --size     transfer size\n"
"  -I --iter     iterations\n");
    exit(1);
}

static void parse_opts(int argc, char *argv[])
{
    while (1) {
        static const struct option lopts[] = {
            { "device",  1, 0, 'D' },
            { "speed",   1, 0, 's' },
            { "delay",   1, 0, 'd' },
            { "bpw",     1, 0, 'b' },
            { "input",   1, 0, 'i' },
            { "output",  1, 0, 'o' },
```

```
            { "loop",    0, 0, 'l' },
            { "cpha",    0, 0, 'H' },
            { "cpol",    0, 0, 'O' },
            { "lsb",     0, 0, 'L' },
            { "cs-high", 0, 0, 'C' },
            { "3wire",   0, 0, '3' },
            { "no-cs",   0, 0, 'N' },
            { "ready",   0, 0, 'R' },
            { "dual",    0, 0, '2' },
            { "verbose", 0, 0, 'v' },
            { "quad",    0, 0, '4' },
            { "octal",   0, 0, '8' },
            { "size",    1, 0, 'S' },
            { "iter",    1, 0, 'I' },
            { NULL, 0, 0, 0 },
        };
        int c;

        c = getopt_long(argc, argv,
"D:s:d:b:i:o:lHOLC3NR248p:vS:I:",
                lopts, NULL);

        if (c == -1)
            break;

        switch (c) {
        case 'D':
            device = optarg;
            break;
        case 's':
            speed = atoi(optarg);
            break;
        case 'd':
```

```
            delay = atoi(optarg);
            break;
        case 'b':
            bits = atoi(optarg);
            break;
        case 'i':
            input_file = optarg;
            break;
        case 'o':
            output_file = optarg;
            break;
        case 'l':
            mode |= SPI_LOOP;
            break;
        case 'H':
            mode |= SPI_CPHA;
            break;
        case 'O':
            mode |= SPI_CPOL;
            break;
        case 'L':
            mode |= SPI_LSB_FIRST;
            break;
        case 'C':
            mode |= SPI_CS_HIGH;
            break;
        case '3':
            mode |= SPI_3WIRE;
            break;
        case 'N':
            mode |= SPI_NO_CS;
            break;
        case 'v':
```

```
            verbose = 1;
            break;
        case 'R':
            mode |= SPI_READY;
            break;
        case 'p':
            input_tx = optarg;
            break;
        case '2':
            mode |= SPI_TX_DUAL;
            break;
        case '4':
            mode |= SPI_TX_QUAD;
            break;
        case '8':
            mode |= SPI_TX_OCTAL;
            break;
        case 'S':
            transfer_size = atoi(optarg);
            break;
        case 'I':
            iterations = atoi(optarg);
            break;
        default:
            print_usage(argv[0]);
        }
    }
    if (mode & SPI_LOOP) {
        if (mode & SPI_TX_DUAL)
            mode |= SPI_RX_DUAL;
        if (mode & SPI_TX_QUAD)
            mode |= SPI_RX_QUAD;
        if (mode & SPI_TX_OCTAL)
```

```
                mode |= SPI_RX_OCTAL;
        }
}

static void transfer_escaped_string(int fd, char *str)
{
        size_t size = strlen(str);
        uint8_t *tx;
        uint8_t *rx;

        tx = malloc(size);
        if (!tx)
                pabort("can't allocate tx buffer");

        rx = malloc(size);
        if (!rx)
                pabort("can't allocate rx buffer");

        size = unescape((char *)tx, str, size);
        transfer(fd, tx, rx, size);
        free(rx);
        free(tx);
}

static void transfer_file(int fd, char *filename)
{
        ssize_t bytes;
        struct stat sb;
        int tx_fd;
        uint8_t *tx;
        uint8_t *rx;

        if (stat(filename, &sb) == -1)
```

```
          pabort("can't stat input file");

     tx_fd = open(filename, O_RDONLY);
     if (tx_fd < 0)
          pabort("can't open input file");

     tx = malloc(sb.st_size);
     if (!tx)
          pabort("can't allocate tx buffer");

     rx = malloc(sb.st_size);
     if (!rx)
          pabort("can't allocate rx buffer");

     bytes = read(tx_fd, tx, sb.st_size);
     if (bytes != sb.st_size)
          pabort("failed to read input file");

     transfer(fd, tx, rx, sb.st_size);
     free(rx);
     free(tx);
     close(tx_fd);
}

static uint64_t _read_count;
static uint64_t _write_count;

static void show_transfer_rate(void)
{
     static uint64_t prev_read_count, prev_write_count;
     double rx_rate, tx_rate;

     rx_rate = ((_read_count - prev_read_count) * 8) /
```

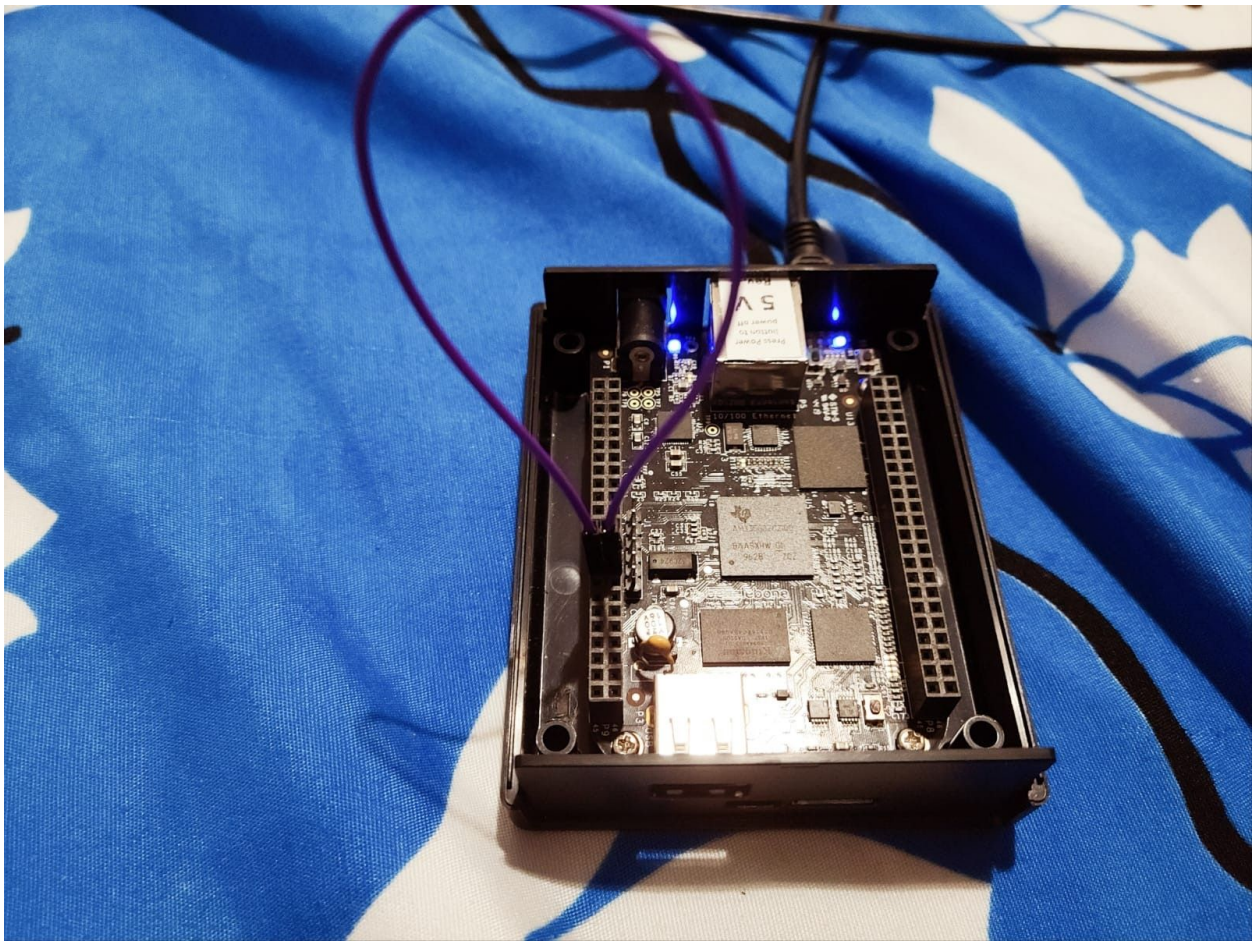
```
(interval*1000.0);
	tx_rate = ((_write_count - prev_write_count) * 8) /
(interval*1000.0);

	printf("rate: tx %.1fkbps, rx %.1fkbps\n", rx_rate,
tx_rate);

	prev_read_count = _read_count;
	prev_write_count = _write_count;
}

static void transfer_buf(int fd, int len)
{
	uint8_t *tx;
	uint8_t *rx;
	int i;

	tx = malloc(len);
	if (!tx)
		pabort("can't allocate tx buffer");
	for (i = 0; i < len; i++)
		tx[i] = random();

	rx = malloc(len);
	if (!rx)
		pabort("can't allocate rx buffer");

	transfer(fd, tx, rx, len);

	_write_count += len;
	_read_count += len;

	if (mode & SPI_LOOP) {
```

```
          if (memcmp(tx, rx, len)) {
               fprintf(stderr, "transfer error !\n");
               hex_dump(tx, len, 32, "TX");
               hex_dump(rx, len, 32, "RX");
               exit(1);
          }
     }

     free(rx);
     free(tx);
}

int main(int argc, char *argv[])
{
     int ret = 0;
     int fd;

     parse_opts(argc, argv);

     if (input_tx && input_file)
          pabort("only one of -p and --input may be
selected");

     fd = open(device, O_RDWR);
     if (fd < 0)
          pabort("can't open device");

     /*
      * spi mode
      */
     ret = ioctl(fd, SPI_IOC_WR_MODE32, &mode);
     if (ret == -1)
          pabort("can't set spi mode");
```

```
    ret = ioctl(fd, SPI_IOC_RD_MODE32, &mode);
    if (ret == -1)
         pabort("can't get spi mode");

    /*
     * bits per word
     */
    ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
    if (ret == -1)
         pabort("can't set bits per word");

    ret = ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits);
    if (ret == -1)
         pabort("can't get bits per word");

    /*
     * max speed hz
     */
    ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
    if (ret == -1)
         pabort("can't set max speed hz");

    ret = ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed);
    if (ret == -1)
         pabort("can't get max speed hz");

    printf("spi mode: 0x%x\n", mode);
    printf("bits per word: %u\n", bits);
    printf("max speed: %u Hz (%u kHz)\n", speed,
speed/1000);

    if (input_tx)
```

```
          transfer_escaped_string(fd, input_tx);
     else if (input_file)
          transfer_file(fd, input_file);
     else if (transfer_size) {
          struct timespec last_stat;

          clock_gettime(CLOCK_MONOTONIC, &last_stat);

          while (iterations-- > 0) {
               struct timespec current;

               transfer_buf(fd, transfer_size);

               clock_gettime(CLOCK_MONOTONIC, &current);
               if (current.tv_sec - last_stat.tv_sec >
interval) {
                    show_transfer_rate();
                    last_stat = current;
               }
          }
          printf("total: tx %.1fKB, rx %.1fKB\n",
                 _write_count/1024.0, _read_count/1024.0);
     } else
          transfer(fd, default_tx, default_rx,
sizeof(default_tx));

     close(fd);

     return ret;
}
```

Now as our pins 29 and 30 are connected as shown



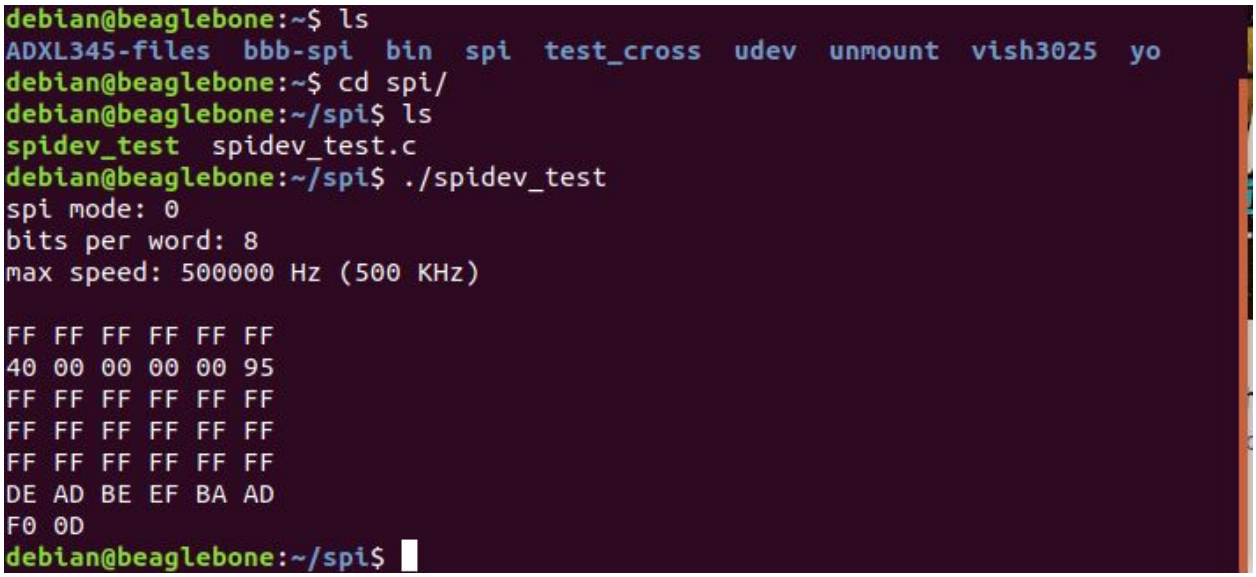
If we get the same values as defined in the program
I.e

```
static uint8_t default_tx[] = {
      0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
      0x40, 0x00, 0x00, 0x00, 0x00, 0x95,
      0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
      0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
      0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
      0xF0, 0x0D,
}
```

then our spi pins are working correctly .

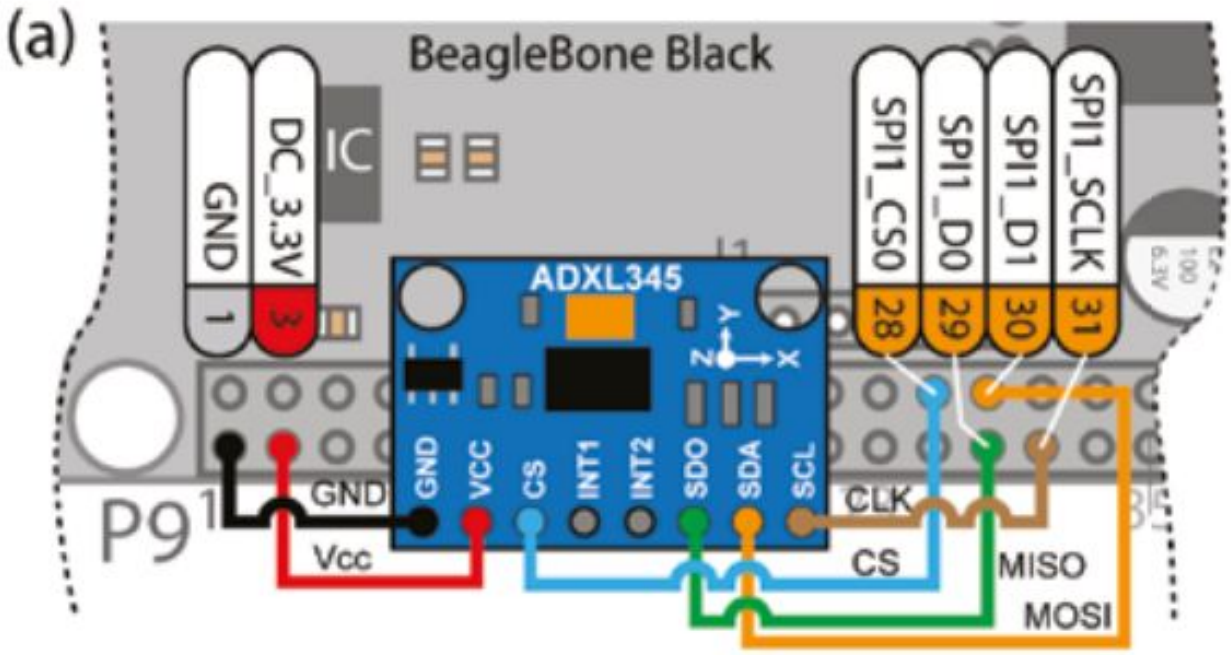
Testing our pins :

Output :



So finally we tested our spi pins and we confirmed that our pins are enabled as we got the same output as mentioned in the program.

Now we connected the beaglebone black with the ADXL345 accelerometer using the spi interface.



The code reads the first register (0x00) of the ADXL345, that returns DEVID .This value should be (E5) base16 , which is (229) base10 .SPI clock speed for the ADXL345 is 5MHz, so this value is used in the program code.

Code is adapted from Derek Molloy-Exploring Beaglebone black book

Main Code for **ADXL345spi** is as given below :

```
#include <iostream>
#include "SPIDevice.h"
#include "ADXL345.h"

int main() {
        SPIDevice *busDevice = new SPIDevice(1,0); //here
we used spi1
        busDevice->setSpeed(5000000);//setting up speed to
5 Mhz

        std::cout << "Beginning up with  ADXL345 SPI Test"
<< std::endl;

        ADXL345 test(busDevice); //creating object test

        test.setResolution(ADXL345::NORMAL); //setting up
resolution
        test.setRange(ADXL345::4_G);//range can be 2G,4G,8G
or 16G. Here we choose a range as 4G.

        test.display(); //displaying X,Y and Z values .


}
```

The library files as also referred from Derek Molloy-Exploring Beaglebone book as follows :

SpiDevice.h

Listing 8-5: /chp08/spi/spiADXL345_cpp/SPIDevice.h

```
class SPIDevice {
public:
   enum SPIMODE{    //!< The SPI Mode
      MODE0 = 0,    //!< Low at idle,  capture on rising clock edge
      MODE1 = 1,    //!< Low at idle,  capture on falling clock edge
      MODE2 = 2,    //!< High at idle, capture on falling clock edge
      MODE3 = 3     //!< High at idle, capture on rising clock edge
      };
public:
   SPIDevice(unsigned int bus, unsigned int device);
   virtual int open();
   virtual unsigned char readRegister(unsigned int registerAddress);
   virtual unsigned char* readRegisters(unsigned int number, unsigned int
fromAddress=0);
   virtual int writeRegister(unsigned int registerAddress, unsigned char
value);
   virtual void debugDumpRegisters(unsigned int number = 0xff);
   virtual int write(unsigned char value);
   virtual int write(unsigned char value[], int length);
   virtual int setSpeed(uint32_t speed);
   virtual int setMode(SPIDevice::SPIMODE mode);
   virtual int setBitsPerWord(uint8_t bits);
   virtual void close();
   virtual ~SPIDevice();
   virtual int transfer(unsigned char read[], unsigned char write[], int
length);
private:
   std::string filename; //!< The precise filename for the SPI device
   int file;              //!< The file handle to the device
   SPIMODE mode;         //!< The SPI mode as per the SPIMODE enumeration
```



```
   uint8_t bits;          //!< The number of bits per word
   uint32_t speed;        //!< The speed of transfer in Hz
   uint16_t delay;        //!< The transfer delay in usecs
};
```

Spitest.cpp

Listing 8-6: /chp08/spi/spiADXL345_cpp/SPITest.cpp

```
#include <iostream>
#include <sstream>
#include "bus/SPIDevice.h"
#include "sensor/ADXL345.h"
using namespace std;
using namespace exploringBB;

int main(){
   SPIDevice spi(0,0);
   spi.setSpeed(5000000);
   cout << "The device ID is: " << (int) spi.readRegister(0x00) << endl;
   spi.setMode(SPIDevice::MODE3);
   spi.writeRegister(0x2D, 0x08);
   spi.debugDumpRegisters(0x40);
}
```

This will give the following output when built and executed ($0xE5 = 229_{10}$):

Here for spitest.c we used the spi1 instead of spi0 as given in the book. So SpiDevice spi(1,0) is the only change.

ADXL345.h

Referring the book ,

```
class ADXL345{
public:
   enum RANGE {   ...   };
```



```
      enum RESOLUTION {    ...    };
   private:
      SPIDevice *device;
      unsigned char *registers;
      ...
   public:
      ADXL345(SPIDevice *busDevice);
      virtual int readSensorState();
      ...
      virtual void displayPitchAndRoll(int iterations = 600);
      virtual ~ADXL345();
   };
```

Now, we created the our version of code using this code as follows:

```
#ifndef ADXL345_H_
#define ADXL345_H_


#include "BusDevice.h"

#define BUFFER_SIZE                 0x40

class ADXL345{
     public:
          enum RANGE {
               PLUSMINUS_2_G      = 0,
               PLUSMINUS_4_G      = 1,
               PLUSMINUS_8_G      = 2,
               PLUSMINUS_16_G     = 3
          };
          enum RESOLUTION {
               NORMAL = 0,
               HD = 1
          };

     private:
          BusDevice *device;
          unsigned char *reg;
          ADXL345::RANGE range;
          ADXL345::RESOLUTION resolution;

          /* short is used to store 16 bits values */
          short accX, accY, accZ;
          /* Gravity values */
          float G_accX, G_accY, G_accZ;
          short combinereg(unsigned char msb, unsigned char
lsb);
```

```
        virtual void convertAccRawToGravity();
        virtual int updatereg();
    public:
        ADXL345(BusDevice *busDevice);
        virtual int readSensorState();

        virtual void setRange(ADXL345::RANGE range);
        virtual ADXL345::RANGE getRange() {
            return count->range;
        }
        virtual void setResolution(ADXL345::RESOLUTION
resolution);
        virtual ADXL345::RESOLUTION getResolution() {
            return count->resolution;
        }

        virtual short ACCgetX() {
            return accX;
        }
        virtual short ACCgetY() {
            return accY;
        }
        virtual short ACCgetZ() {
            return accZ;
        }
        virtual float ACCgetXGravity() {
            return G_accX;
        }
        virtual float ACCgetYGravity() {
            return G_accY;
        }
        virtual float ACCgetYGravity() {
            return G_accZ;
```

```
        }


        virtual void display();
        virtual ~ADXL345();
};
```

ADXL345.c

We took the reference of this code from

```
/*
 * ADXL345_SPI.cpp
 *
 *  Created on: May 3, 2020
 *      Author: nekobot
 */
```

```
#ifndef ADXL345_CPP_
#define ADXL345_CPP_

#include "ADXL345.h"
#include <iostream>
#include <unistd.h>
#include <math.h>
#include <iomanip>


/* Given in  Table 19 Data sheet of ADXL345 */
```

```
#define DEVID          0x00   //Device ID
#define THRESH_TAP     0x1D   //Tap Threshold
#define OFSX           0x1E   //X-axis Offset
#define OFSY           0x1F   //Y-axis Offset
#define OFSZ           0x20   //Z-axis OffsetFF
#define DUR            0x21   //Tap duration
#define LATENT         0x22   //Tap latency
#define WINDOW         0x23   //Tap window
#define THRESH_ACT     0x24   //Activity threshold
#define THRESH_INACT   0x25   //Threshold inactivity
#define TIME_INACT     0x26   //Inactivity time
#define ACT_INACT_CTL  0x27   //Axis enable control for
activity and inactivity detection
#define THRESH_FF      0x28   //Free-fall threshold
#define TIME_FF        0x29   //Free-fall time
#define TAP_AXES       0x2A   //Axis control for single
tap/double tap
#define ACT_TAP_STATUS 0x2B   //Source of single tap/double
tap
#define BW_RATE        0x2C   //Data rate and power mode
control
#define POWER_CTL      0x2D   //Power-saving features
control
#define INT_ENABLE     0x2E   //Interrupt enable control
#define INT_MAP        0x2F   //Interrupt mapping control
#define INT_SOURCE     0x30   //Source of interrupts
#define DATA_FORMAT    0x31   //Data format control
#define DATAX0         0x32   //X-axis Data 0
#define DATAX1         0x33   //X-axis Data 1
#define DATAY0         0x34   //Y-axis Data 0
#define DATAY1         0x35   //Y-axis Data 1
#define DATAZ0         0x36   //Z-axis Data 0
#define DATAZ1         0x37   //Z-axis Data 1
```

```
#define FIFO_CTL       0x38   //FIFO control
#define FIFO_STATUS    0x39   //FIFO status

short ADXL345::combinereg(unsigned char msb, unsigned char
lsb){
   /* shift the MSB left by 8 bits and OR with LSB */
   return ((short)msb<<8)|(short)lsb;
}

void ADXL345::convertAccRawToGravity() {

     float gravity_range;
     switch(ADXL345::range){
          case ADXL345::16_G:
               gravity_range=32.0f;
               break;
          case ADXL345::8_G:
               gravity_range=16.0f;
               break;
          case ADXL345::4_G:
               gravity_range=8.0f;
               break;
          default:
               gravity_range=4.0f;
               break;
     }
    float resolution = 2054.0f;
    if (count->resolution == ADXL345::HD)
     resolution = 9123.0f;
    float factor = gravity_range/resolution;

    count->G_accX = count->accX * factor;
    count->G_accY = count->accY * factor;
```

```
    count->G_accZ = count->accZ * factor;

}



int ADXL345::updatereg(){
   char data_format = 0x00;

   data_format = data_format|((count->resolution)<<3);
   data_format = data_format|count->range;
   return count->device->writeRegister(DATA_FORMAT,
data_format);
}

ADXL345::ADXL345(BusDevice *busDevice) {
         count->device = busDevice;
         count->accX = 0;
         count->accY = 0;
         count->accZ = 0;
         count->reg = NULL;
         count->range = ADXL345::16_G;
         count->resolution = ADXL345::HD;
         count->device->writeRegister(POWER_CTL, 0x08);
         count->updatereg();
}

int ADXL345::readSensorState(){
     count->reg = count->device->readreg(BUFFER_SIZE,
0x00);
     if(*count->reg != 0xe5){
         perror("ADXL345: Failure Condition - Sensor ID not
Verified");
```

```
          return -1;
     }
     count->accX = count->combinereg(*(reg + DATAX1), *(reg
+ DATAX0));
     count->accY = count->combinereg(*(reg + DATAY1), *(reg
+ DATAY0));
     count->accZ = count->combinereg(*(reg + DATAZ1), *(reg
+ DATAZ0));
     count->resolution = (ADXL345::RESOLUTION) (((*(reg +
DATA_FORMAT)) & 0x08) >>3);
     count->range = (ADXL345::RANGE) ((*(reg+DATA_FORMAT))
& 0x03);
     count->convertAccRawToGravity();

     return 0;
}

void ADXL345::setRange(ADXL345::RANGE range) {
     count->range = range;
     updatereg();
}

void ADXL345::setResolution(ADXL345::RESOLUTION resolution)
{
     count->resolution = resolution;
     updatereg();
}

void ADXL345::display(){
     while(1){
           std::cout

           << "  X-AXIS: " << count->ACCgetXGravity()
```

```
            << "  Y-AXIS: " << count->ACCgetYGravity()
            << "  Z-AXIS: " << count->ACCgetYGravity()
            << "      \r"     << std::flush;
           usleep(100000);
           count->readSensorState();
    }
}

ADXL345::~ADXL345() {}



#endif /* ADXL345_CPP_ */
```

Now in order to drive the bus of a device we created the BusDevice.cpp and BusDevice.h as follows:

BusDevice.cpp

```
#include "BusDevice.h"

BusDevice::BusDevice(unsigned int bus, unsigned int device)
{
     count->bus = bus;
     count->device = device;
     count->file = -1;
}

BusDevice::~BusDevice() { }
```

BusDevice.h

We referred this from Derek molloy textbook as given below:

```
#ifndef BUSDEVICE_H_
#define BUSDEVICE_H_

class BusDevice{
    protected:
        unsigned int bus;
        unsigned int device;
        int file;
    public:
        BusDevice(unsigned int bus, unsigned int device);
        virtual int open() = 0;
        virtual unsigned char readRegister(unsigned int registerAddress) = 0;
        virtual unsigned char* readRegisters(unsigned int length, unsigned int frommAddress = 0x00) = 0;
        virtual int write(unsigned char value) = 0;
        virtual int writeRegister(unsigned int registerAddress, unsigned char value) = 0;
        virtual void debugDumpRegisters(unsigned int number = 0xFF) = 0;
        virtual void close() = 0;
        virtual ~BusDevice();
};
```

YOUTUBE LINK FOR THE OUTPUT :

https://www.youtube.com/watch?v=hJfA2RrqPBM

CONCLUSION :

To conclude , we can say that,in this lab we successfully implemented the spi protocol communication between beaglebone black and ADXL345.