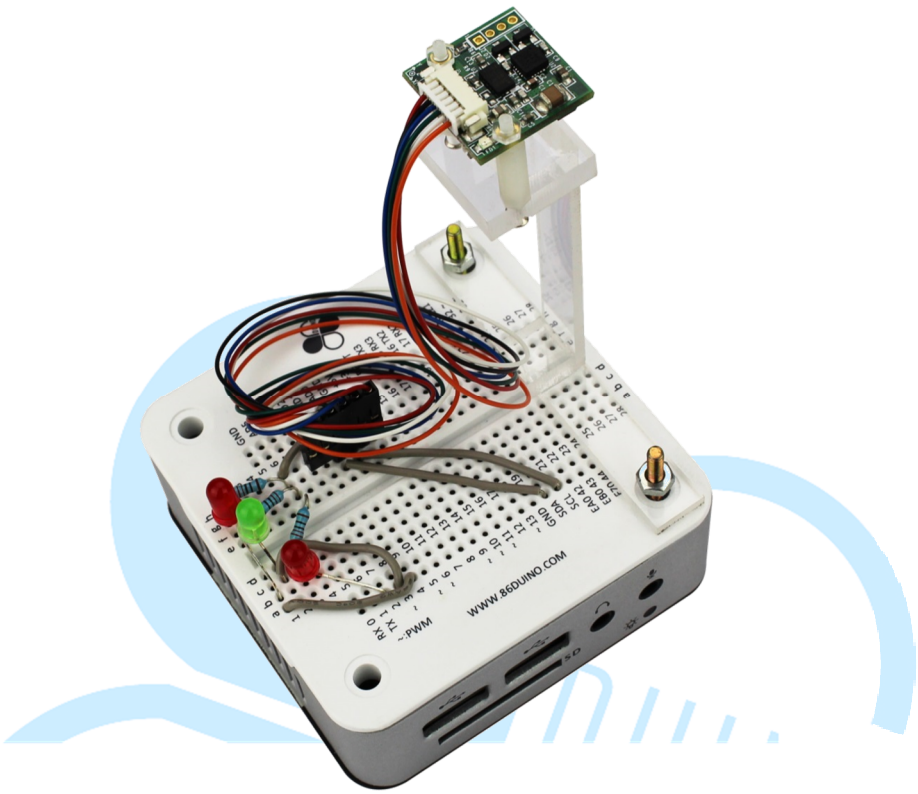
# EduCake: I²C Communication




## 1.Introduction to I²C Communication

In the previous chapter, we talked about the 86Duino EduCake's serial port, where it uses a pair of TX/RX I/O pins to perform serial communication, transmit and receive data serially. I²C, short for Inter-Integrated Circuit, is another form of interface that uses serial communication.

While both I²C and serial port transmit and receive serial data, these are two different type of interface. In term of hardware, serial port transmit data using the TX signal pin and receive data using the RX signal pin where the transmitting node and receiving node must be configured with the same operating parameters, such as Baudrate, Parity, Data-bits, Stop-bit and etc.

Although I²C also uses two signal pins to communication, it's not the same as the serial port. One of the signal pin (SDA) is for transmitting and receiving data, the other pin (SCL) is the clock signal needed to synchronize data transmission.

I²C communication is different from serial port and only support half-duplex mode where one of the node take on the Master role and the other the Slave role. Communication between two I²C nodes involves configuring one of the nodes

as master and the other as slave. Each of these two nodes take turn to read and send data based on the signal from the SCL pin.

Following are some of the communication parameter for I²C interface:

- Transmission Standards:

  Determined by the master node clock speed and processing capability for the slave, I²C can transmit data in low-speed (10kb/s), standard-speed (100kb/s), high-speed (400kb/s) and etc.

- Slave Address:

  Each I²C slave node has its own unique address. When a master node sends a message, it includes the slave address to identify the message is intended for a specific slave with that address. In the initial I²C design, 16-bits is allocated for the address. In practice, 7-bits and 8-bits address are common. You need to check and verify design specification for the device you are using to setup the communication link correctly.

- Slave Register Address:

  There are multiple registers for each I²C slave, where some of these registers are used for device settings, sensor data and etc. Reading data from these registers and writing data to these registers are routine tasks for I²C communication.

  It's common for device maker to define different register addresses for their I²C device (some are fixed address and some are set by hardware design). When using I²C device from different companies, it's necessary to know each device's design specification and associated communication format, in order to correctly interpret data from the device and communicate with the device. There are 2 communication modes for I²C:

  - Master node send data to be written onto a Slave's register

    Configuration parameters to set the slave in certain operating mode.

  - Master node designate register address to read data from slave

    Typical to read sensor or configuration data from the slave.

Figure-1 below demonstrate I²C communication format and sequence:

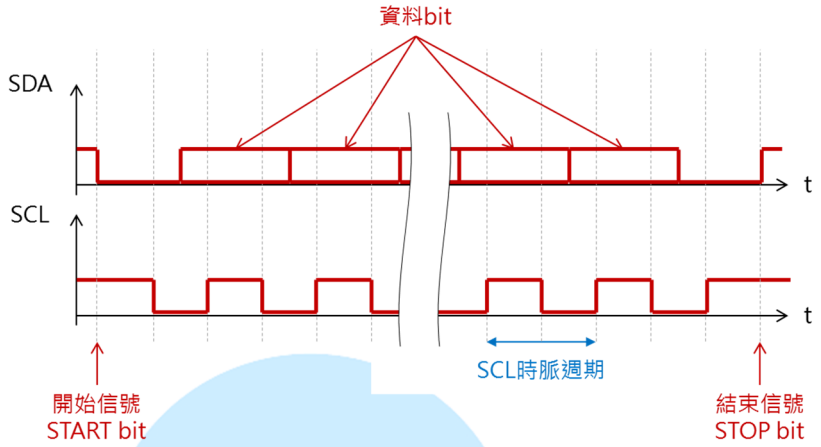



Figure-1: I²C communication format

One of the key advantages for I²C bus is the ability to "daisy chain" and support multiple devices on a single bus, using only two signal lines (SDA and SCL), as shown in Figure-2, which minimize design complexity and PC-board footprint. To communicate with multiple slave devices, communication messages from an I²C master node are sent with slave device-address, which is a unique address for each of the slave on the same bus. While communication messages on the I²C bus are readable by all devices attached to the bus, each device only responses to communication messages with matching device-address. Using simple communication format lacking error correction along with data transmission limited by hardware signal, I²C is not suitable to support longer distance communication such as the RS232 serial port. I²C bus is great connectivity for "chip to chip" and "PC-board to PC-board" communication.

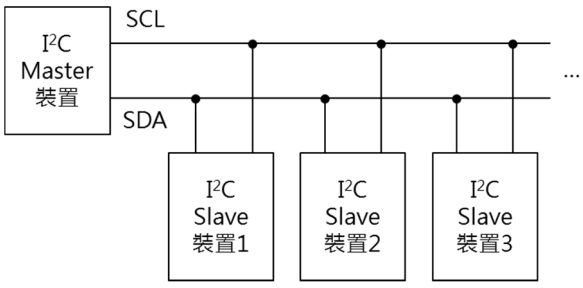


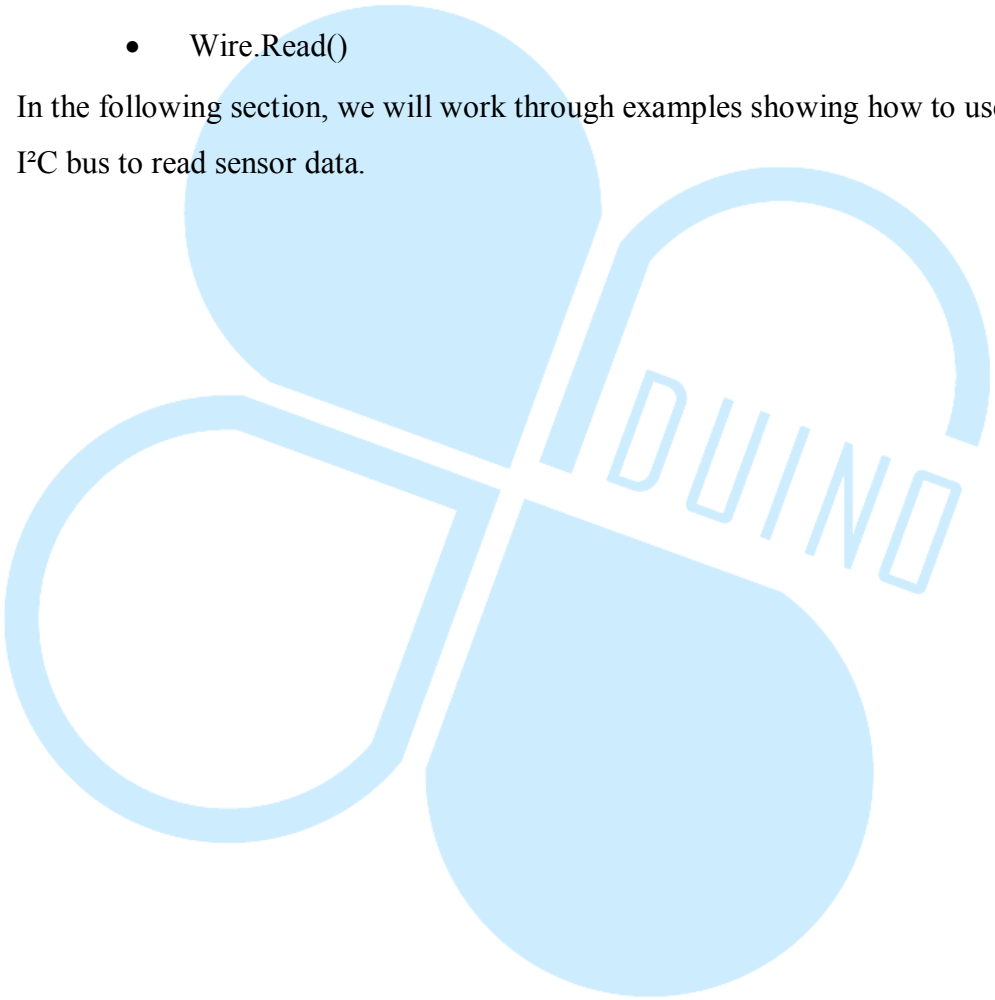

Figure-2: Typical I²C bus implementation.

When working with I²C bus on the 86Duino EduCake platform, it's not necessary to tinker and flip data bits at the bus level. There are prebuilt function to simplify I²C bus usage. By including the Wire.h header file to the project,

you can use the following function to implement I²C communication programmatically:

- Wire.begin()
- Wire.beginTransmission(device-address)
- Wire.write(byte, data)
- Wire.endTransmission()
- Wire.requestFrom(device-address, data-length)
- Wire.Read()

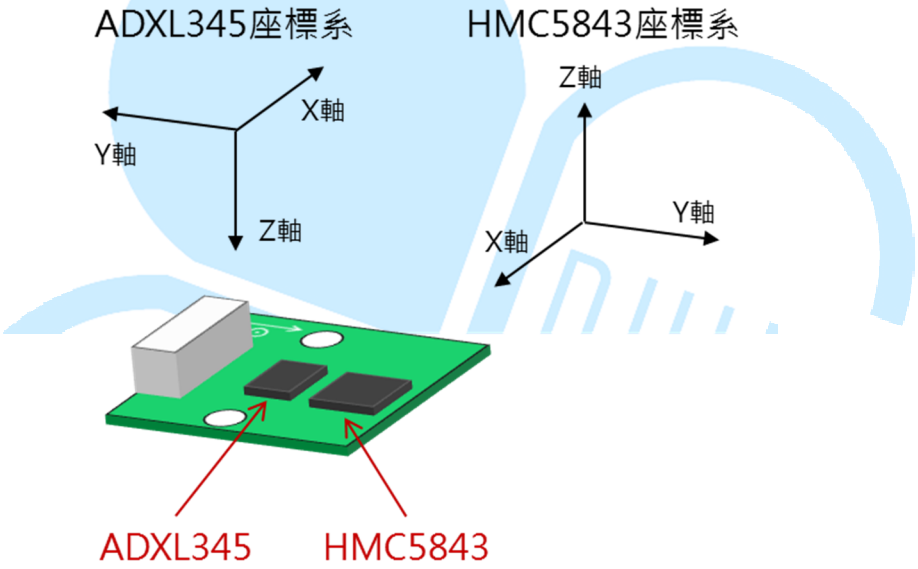
In the following section, we will work through examples showing how to use I²C bus to read sensor data.



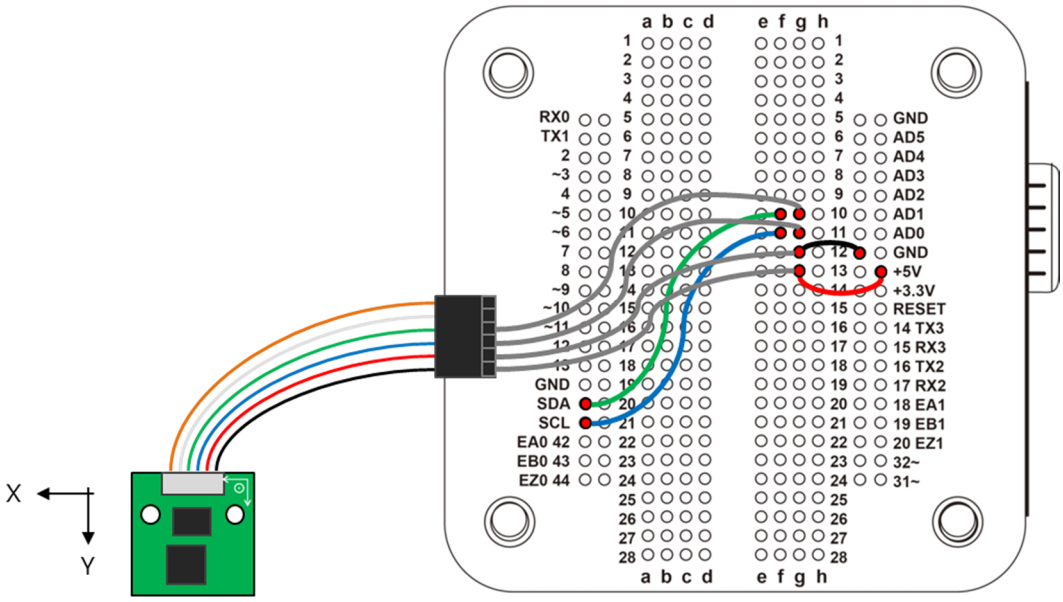
## 2. First Example: Using RM-G144 Accelerometer

In this example, we will look into I²C communication flow between the EduCake and the RM-G144 sensor module, which is built with 3-axes digital compass using HMC5843 chip and 3-axes accelerometer using ADXL345 chip, where both of these two chips are attached to the same I²C bus with different device-address.

First, we need to understand how the RM-G144 sensor module is designed and important to know the sensor coordinates for the digital compass are reversed from the accelerometer, as shown in the figure below:




Attach the RM-G144 module to the EduCake as shown in the figure below:



From the figure above, you can see the connectivity typically needed to connect an I²C device, Vcc (red wire), GND (black wire), SCL (blue wire) and SDA (green wire). The Orange/white wire from the RM-G144 module is not in use. The SCL and SDA signal from the RM-G144 module are connected to the SCL and SDA pin on the EduCake. Vcc is connected to +5V.

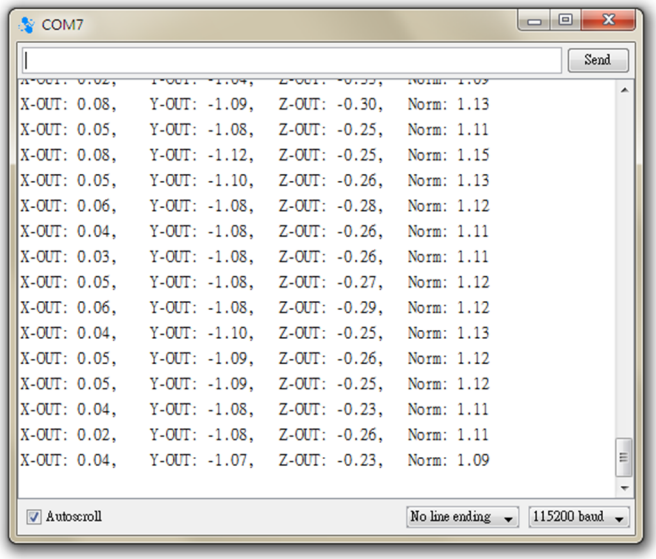
From the 86Duino IDE, enter the following codes:

```
// for RM-G144
// IC : ADXL345
// Digital Accelerometer
#include <Wire.h>

int acc_address = 0x53;// I2C device-dataaddress, 7-bits
unsigned int temp[6];

char *out_title[ ] = {"X-OUT: ","Y-OUT: ","Z-OUT: "};

// array to store sensor reading 0:X-out, 1:Y-out, 2:Z-out
double acc_value[3] = {0,0,0};

void setup( ) {
  Serial.begin(115200); // Initial Serial port

  Wire.begin( ); // Initial I2C device

  G144_Acc_Init( ); // Initialize ADXL345 register
}

void loop( ) {
  G144_Acc_Read( );// Read sensor value

  // Output sensor value
  for(int i=0 ; i<3 ; i++){
    Serial.print(out_title[i]);
    Serial.print(acc_value[i]);
```

```
    Serial.print(",\t");
    }
    Serial.print("Norm: ");
    Serial.print(
sqrt(acc_value[0]*acc_value[0]   +   acc_value[1]*acc_value[1]   +
acc_value[2]*acc_value[2]) );
    Serial.println( );

    delay(100);
}

void    G144_Acc_Init(    ){//    Initialize    ADXL345    register
-------------------------------------------------
    Wire.beginTransmission(acc_address);
      Wire.write(0x2d); //Power_Control register

      // link & measure mode = 0010 1000, Link Bit=1, Measure Bit=1
      Wire.write(0x28); //
    Wire.endTransmission( );

    delayMicroseconds(100);

    Wire.beginTransmission(acc_address);
      Wire.write(0x31); //Data_Format register
      Wire.write(0x08);
      // Set Full_Resolution = 0000 1000
      // FULL_RES Bit=1 (scale factor = 4 mg/LSB), Range Bits=00
(±2g)，10 bits resolution
      // Range bits, bit-0 and bit-1, used to set 4 different
measurement ranges.
      // Resolution can vary from 10 bits ~ 13 bits，
      // under full resolution mode, 4mg/LSB as scale factor
    Wire.endTransmission( );
delayMicroseconds(100);

    Wire.beginTransmission(acc_address);
      Wire.write(0x38); //FIFO_Control register
      Wire.write(0x00); //bypass mode
    Wire.endTransmission( );

    delayMicroseconds(100);
}

void      G144_Acc_Read(      ){//      Read      sensor      value
-------------------------------------------------
Wire.beginTransmission(acc_address);
      Wire.write(0x32); //Read from X register (Address : 0x32)
    Wire.endTransmission( );
```

```
    // Read 6 byte of data from designed address
    Wire.requestFrom(acc_address, 6);

    int count = 0;
    while(Wire.available( ) && count < 6){
       temp[count] = Wire.read( );// read data
       count++;
    }
    /*
    temp[0] : X LSB  temp[1] : X MSB
    temp[2] : Y LSB  temp[3] : Y MSB
    temp[4] : Z LSB  temp[5] : Z MSB
    */
    // Process data
    // ADXL345 return data for each axis as 2 byte binary value
    // data convert to signed integer value
    // mask = 1111 1111 0000 0000

  // 負值符號延伸 = mask | 0000 0000 MSB LSB = 1111 1111 MSB LSB
    // Return value range : -512 ~ 511 ( ± 2 g ) ·
    // when sensor data exceed the range, max value is returned
    int temp_value = 0;
    for(int i=0 ; i<3 ; i++){

      // MSB AND b'10000000'
      // When sign bit for MSB is not 0, it's a negative value
      if((temp[i*2+1] & 0x80) != 0){

        temp_value   =   (((~0)>>16)<<16)   |   ((temp[i*2+1]<<8)   +
temp[i*2]);
      }
      else{
        temp_value = (temp[i*2+1]<<8) + temp[i*2];
      }

      // 4mg/LSB, convert to g
      acc_value[i] = (double)temp_value * 4 / 1000;
    }
  }
```

In this example, sensor data is read from the accelerometer sensor on the RM-G144 module, once every 100ms. Senor data from all three axes are read, converted to decimal value and output to the Serial Monitor, as shown in the figure below.




In the above exercise, the line of code “#include ,Wire.h” at the beginning brings in I²C related function needed by the program. Since the program involves reading data from the ADXL345 register in multiple function, using the int_acc_address variable to store the address for ADXL345 make it easier to make change and adjustment.

The process to get sensor data involves two processes, initialize the device and read the data. Since these functions can be useful elsewhere, the codes related to ADXL345 are encapsulated in the G144_Acc_Init() and G144_Acc_Read() functions, which enable you to easily replicate these function in another application.

Initialization codes for the program are combined in the Setup() function, which include the G144_Acc_Init() function call.

To interact with the salve, an I²C Master send configuration data to the slave’s register which configure the slave to operate in certain mode, using codes such as the following:

```
// trigger the start bit as shown in Figure-1
Wire.beginTransmission(device-address);
Wire.write(register-address);
Wire.write(data);
// trigger the stop bit as shown in Figure-1
Wire.endTransmission( );
```

The slave device respond to the above communication and perform the required actions.

The ADXL345 device initialization involve the following 3 steps:

1. Write the data (“0x28”) to address “0x2d”:

   Configure Power Control register with 0010 1000, Link Bit = 1, Measure Bit = 1.

2. Write the data (“0x08”) to address “0x31”:

   Configure Data Format register with 000 1000, set sensor range to Full_Resolution, FULL_RES Bit=1 indicate the scale factor is 4mg/LSB, set the Range Bits to 00 ( ±2g), in 10 Bits resolution.

   The Range bits, Bit-0 and Bit-1, can set the sensor to 4 different ranges (±2g ~ ±16g)

   The resolution can be set to 10-bits ~ 13-bits, depending on actual usage.

   Under Full_Resolution mode, the scale factor is 4mg/LSB.

3. Write the data (“0x38”) to address “0x00”

   Set FIFO Control register to 0000 1000, in bypass mode.

According to the IC datasheet, after writing data to the device, it’s recommended to delay by 100µs before the next action. The delayMicroseconds(100) function call is added to serve this purpose.

Within the main program loop, loop(), the G144_Acc_Read() function is called to read data from the sensor, which go through the following steps to read the data:

```
Wire.beginTransmission(device-address);
Wire.write(register-address);
Wire.endTransmission( );

// data-length = number of byte to read
Wire.requestFrom(device-address, data-length);

int count = 0;
while(Wire.available( ) && count < data-length){
  temp[count] = Wire.read( );// Read data return from the device
  count++;
}
```

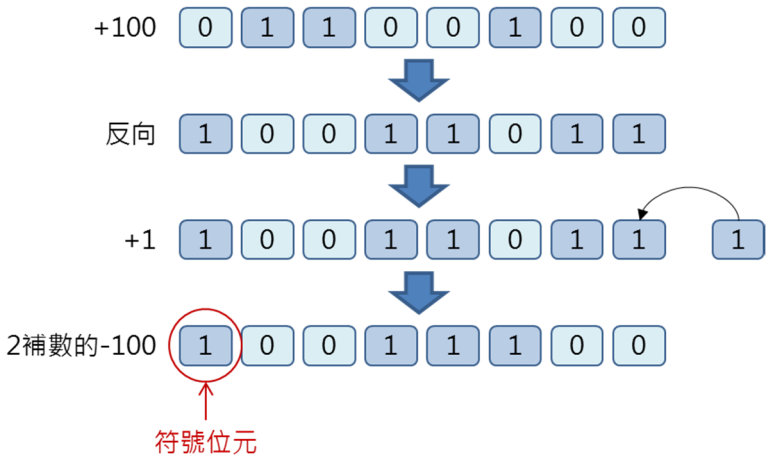
In the above, the first line of code designated a device-address follow by register address to read from. Then, the Wire.requestFrom(device-address, data-length) function sends the request to the slave device and the slave device is expected to return the data in byte, where data-length is the number of byte to return from

the slave. The Wire.available() is used to check whether the slave device returned any data and call the Wire.read() function to read the data. For this exercise, since there are 6 bytes of data from the acceleromter sensor, starting from the address 0x32, the while() loop is set to read 6 bytes of data and place them in the temp[] array.

In addition to getting the data from the sensor, which is in binary form, the data need to be converted to double decimal value. In the beginning of the program, the unsigned int temp[6] array is declared to temporary store sensor value. The double acc_value[3] array is declared to store the converted sensor value. Since there are 6 bytes of sensor data, starting from the address 0x32 for: X LSB, X MSB, Y LSB, Y MSB, Z LSB, Z MSB, one of the last action within the G144_Acc_Read() function is to convert these sensor value. Sensor value for each of the sensor's axis is in binary format presented as 2 bytes, LSB and MSB, which must be combined before the conversion. The sensor value can be negative and uses the following code to identify whether the sensor value is positive or negative:

```
// MSB AND b'10000000'
// When the MSB sign bit is not 0, it indicate a negative value
if((temp[i*2+1] & 0x80) != 0){
   temp_value = (((~0)>>16)<<16) | ((temp[i*2+1]<<8)+temp[i*2]);
}
else{
   temp_value = (temp[i*2+1]<<8) + temp[i*2];
}
acc_value[i] = (double)temp_value * 4 / 1000;
```

Let's take a look at binary value, as shown below:



Using 100 as example, binary representation for -100 is the reverse, where each bit is changed to the opposite, and add 1 to the result. When the above +100 and -100 binary value are added together, it should yield 0 as the result.

Back to the first line of code, the 「temp[i*2+1] & 0x80) != 0」within the if() statement is used to check whether the MSB bit, the left most bit, is equal to 0. When this bit is 0, the value is positive. When this bit is 1, the value is negative. Since the Wire.read() read one byte of data at a time, the MSB value must be move to the left by 8 bit and then add to the LSB. When the data is convert to integer (Interger value type for 86Duino EduCake is in 32 bits). The bits to the left are used to represent whether the value is positive or negative. For negative value, set the bits to the left to 1.

Following are steps to compute binary number with negative value:

1. First, create the following mask:

   11111111 11111111 00000000 00000000

2. Combine the MSB and LSB value

   00000000 00000000 xxxxxxxx 00000000

   00000000 00000000 00000000 yyyyyyyy

   ⇨ 00000000 00000000 xxxxxxxx yyyyyyyy

3. Perform an OR function between the combined value in step-2 and the mask in step-1.

   ⇨ 11111111 11111111 xxxxxxxx yyyyyyyy

In the final step, the value is converted from mg/LSB to g/LSB, where 1000 mg = 1 g. During the initialization phase, Full resolution is selected, where the scale factor is 4 mb/LSB. The following line of code convert the value to g (where g = 9.8m/$s^2$), acceleration due to gravity:

- acc_value[i] = (double)temp_value * 4 / 1000

Note: The value needs to be converted for integer to floating point as part of the above calculation.

For more information about the accelerometer sensor in this exercise, which is part of the RM-G144 sensor module, visit the following URL:

- http://www.roboard.com/G144.html

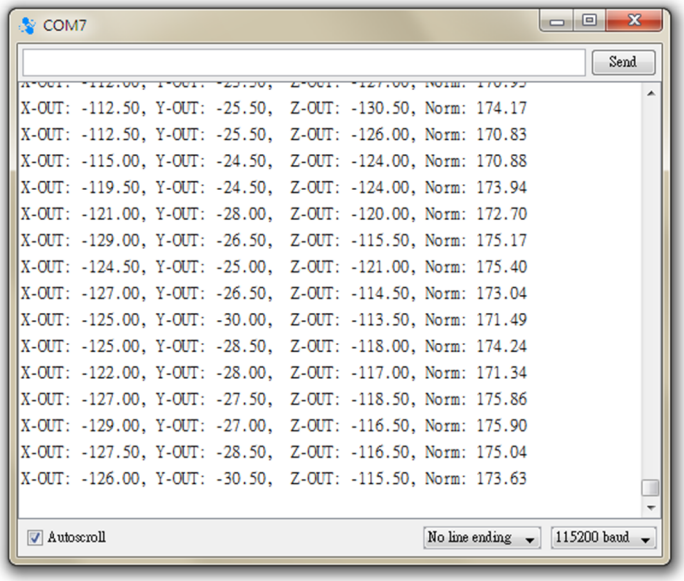
## 3. Second Example: Using RM-G144 Magnetic Compass

In this exercise, we will use the HMC5843 magnetic compass on the RM-G144 sensor module and use the same circuitry from the previous exercise. From the 86Duino IDE, enter the following codes:

```
// for RM-G144
// IC : HMC5843
// Digital Compass
#include <Wire.h>

int mag_address = 0x1e;// I2C 裝置位址，此為 7bit 位址
unsigned int temp[6];

char *out_title[ ] = {"X-OUT: ","Y-OUT: ","Z-OUT: "};
double mag_value[3] = {0,0,0};// 儲存感測器三軸數值的矩陣 0:X-out,
1:Y-out, 2:Z-out

void setup( ) {
  Serial.begin(115200); // Initial Serial port

  Wire.begin( );// Initial I2C device

  G144_Mag_Init( ); // 初始化 HMC5843 的暫存器
}

void loop( ) {

  G144_Mag_Read( );// 讀取感測器並處理數值

  // 印出數值
  for(int i=0 ; i<3 ; i++){
    Serial.print(out_title[i]);
    Serial.print(mag_value[i]);
    Serial.print(",\t");
  }
  Serial.print("Norm: ");
  Serial.print(
   sqrt(mag_value[0]*mag_value[0]  +  mag_value[1]*mag_value[1]  +
mag_value[2]*mag_value[2]) );
  Serial.println( );// 換行

  delay(100);// continue-measureture mode 須有>100us 的間隔時間
}
```

```
void   G144_Mag_Init(   ){//   初 始 化   HMC5843   的 暫 存 器
-------------------------------------------------
  Wire.beginTransmission(mag_address);
     Wire.write(0x02); //mode register
     Wire.write(0x00); //continue-measureture mode
  Wire.endTransmission( );

  delayMicroseconds(100);
}

void   G144_Mag_Read(   ){//   讀 取 感 測 器 並 處 理 數 值
-------------------------------------------------
  Wire.beginTransmission(mag_address);
     Wire.write(0x03); //Read from data register (Address : 0x03)
  Wire.endTransmission( );

  Wire.requestFrom(mag_address, 6); // Read data form Conversion
Result Register, Data : 12bits
  int count = 0;
  while(Wire.available( ) && count < 6){
  temp[count] = Wire.read( );// 接收傳回來的資料
     count++;
  }
  /*
  temp[0] = X MSB  temp[1] = X LSB
  temp[2] = Y MSB  temp[3] = Y LSB
  temp[4] = Z MSB  temp[5] = Z LSB
  */
  // 處理資料
  // HMC5843 回傳數值，每軸向分為兩個 byte
  // 數值以 2 補數法表示，int 變數須做符號延伸
  // 回傳值範圍：-2048~2047，若外在磁力超過表示範圍，將會回傳-4096
  int temp_value = 0;
  for(int i=0 ; i<3 ; i++){
    if((temp[i*2] & 0x80) != 0){// MSB AND b'10000000，如果 MSB 的
sign bit 不為 0，表示為負值
     temp_value   =   (((~0)>>16)<<16)   |   ((temp[i*2]<<8)   +
temp[i*2+1]);
    }
    else{
      temp_value = (temp[i*2]<<8) + temp[i*2+1];
    }
    mag_value[i] = (double)temp_value * 0.5;// 0.5 milli-gauss/LSB，
換算為單位：milli-gauss
  }
}
```

Similar to the previous exercise, the sensor data from the RM-G144 digital compass is read once every 100ms and output to the Serial Monitor via the serial port after the sensor data has been processed, as shown in the figure below.

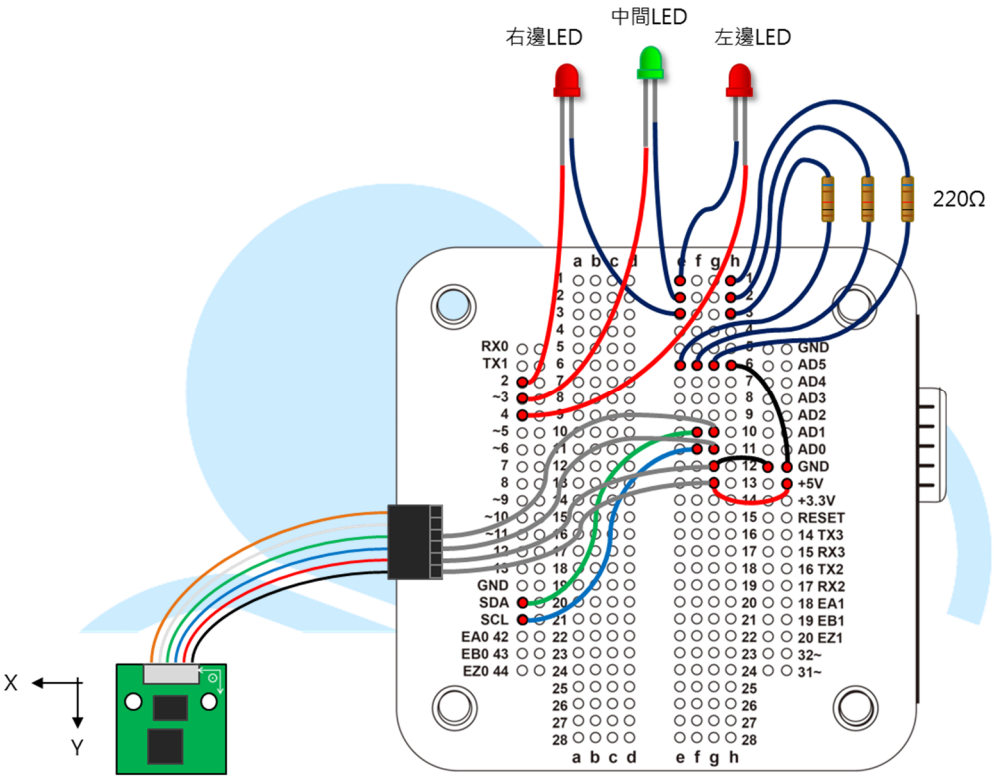



The code's initialization phase is almost identical to the first exercise, except the HMC5843 sensor's device address is at 0x1e, which is also a 7 bit address, and the device's initialization register is different, entering the value 0x00 at the 0x02 register, to set the device to continue-measure mode.

The location to read sensor data is also different from the first exercise, where the data register begin at 0x03, the LSB and MSB data for the 3 axes are in reverse sequence from the ADXL345. The sensor value calculation is also different, from 0.5 milli-gauss/LSB to milli-gauss. On the RM-G144 module, there is a mark that represent the reference coordinate for the HMC5843 digital compass sensor. The sensor measurement is based on the earth due North magnetic field. When the Y-axis is facing the North Pole, it has the highest value and the X-axis value is zero or close to zero. You can try move the sensor in different position to see the different and changing sensor values.

## 4. Third Example: Using sensor to determine orientation

We can create some interesting application with sensor data from the RM-G144 sensor module. For this exercise, add additional circuitry as shown in the figure below.




From the 86Duino IDE, enter the following codes:

```
// for RM-G144
#include <Wire.h>

int acc_address = 0x53;// ADXL345 device-address
int mag_address = 0x1e;// HMC5843 device-address
unsigned int temp[6];

char *out_title[] = {"X-OUT: ","Y-OUT: ","Z-OUT: "};

// array to store ADXL345 sensor value 0:X-out, 1:Y-out, 2:Z-out
double acc_value[3] = {0,0,0};

// array to store HMC5843 sensor value 0:X-out, 1:Y-out, 2:Z-out
double mag_value[3] = {0,0,0};
```

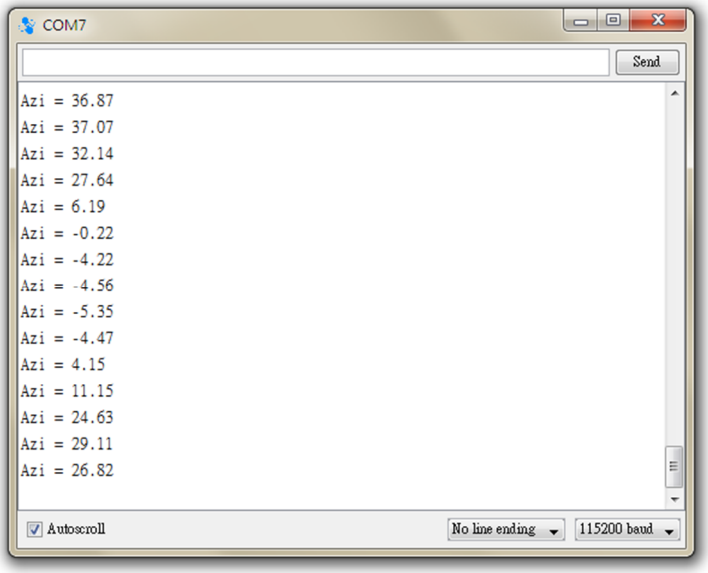
```
int LED_L_pin = 4;// Left LED
int LED_M_pin = 3;// Center LED
int LED_R_pin = 2;// Right LED

void setup() {
  Serial.begin(115200); // Initial Serial port

  Wire.begin( );// Initial I2C device

  G144_Acc_Init( );// initialize ADXL34 register
  G144_Mag_Init( ); // initialize HMC5843 register

  // 初始化 I/O
  pinMode(LED_L_pin,OUTPUT);
  pinMode(LED_M_pin,OUTPUT);
  pinMode(LED_R_pin,OUTPUT);
}

void loop() {
  G144_Acc_Read( );// 讀取感測器並處理數值
  G144_Mag_Read( );// 讀取感測器並處理數值

  float azimuth = GetAzimuth(mag_value);

  Serial.print("Azi = ");
  Serial.println(azimuth);

  // 控制右邊 LED
  if(azimuth<-10){// 羅盤 Y 軸方向偏西北
```

```
digitalWrite(LED_R_pin, HIGH);// 亮起右邊 LED，表示要向順時針方向轉
  }
  else{
    digitalWrite(LED_R_pin, LOW);// LED 熄滅
  }
  // 控制中間 LED
  if(abs(azimuth)<=10){// 羅盤 Y 軸方向在正北+/-10 度內
    digitalWrite(LED_M_pin, HIGH);// 亮起中間 LED，表示目前方向正確
  }
  else{
    digitalWrite(LED_M_pin, LOW);// LED 熄滅
  }
  // 控制左邊 LED
  if(azimuth>10){// 羅盤 Y 軸方向偏東北
    digitalWrite(LED_L_pin, HIGH);// 亮起左邊 LED，表示要向時針方向轉
  }
  else{
    digitalWrite(LED_L_pin, LOW);// LED 熄滅
  }

  delay(100);
}

double GetAzimuth(double *m_val) {//)
  // atan2(y,x) = arccos(y/x);
  double azimuth = atan2(-m_val[0], m_val[1]);// 回傳值為徑度，範圍
為 -PI ~ PI
  /*
  if(azimuth < 0){// 修正 azi 的範圍至 0 到 2*PI 之間
    azimuth = 2 * PI + azimuth; // PI = 3.1415967

}
  */
  azimuth = azimuth * 180 / PI;// 徑度換算成角度
  return azimuth;
    }
```

In this exercise, since these 4 functions are covered in the previous 2 samples, they will not be covered in this exercise:

- G144_Acc_Init()
- G144_Mag_Init()
- G144_Acc_Read()
- G144_Mag_Read()

In this exercise, the sample code read sensor data from the HMC5843 sensor and calculate the current position. When the sensor is moved toward the West direction (more than 10 degree from North), the LED on the right will turn on, as indicator to adjust the direction. Vice-versa, when the sensor position is moved toward the East direction (more than 10 degree from North), the LED on the left will turn on. When the sensor is pointing to the North, within ±10 degree, the LED in the center is turned on to indicate the North pointing direction. After the code is compiled and launched on the 86Duino Educake, the program output data to the Serial Monitor, as shown in the following figure.



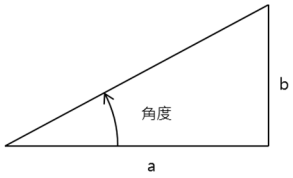

The code that determine sensor direction is in the following function:

- Double GetAzimuth(double *m_val)

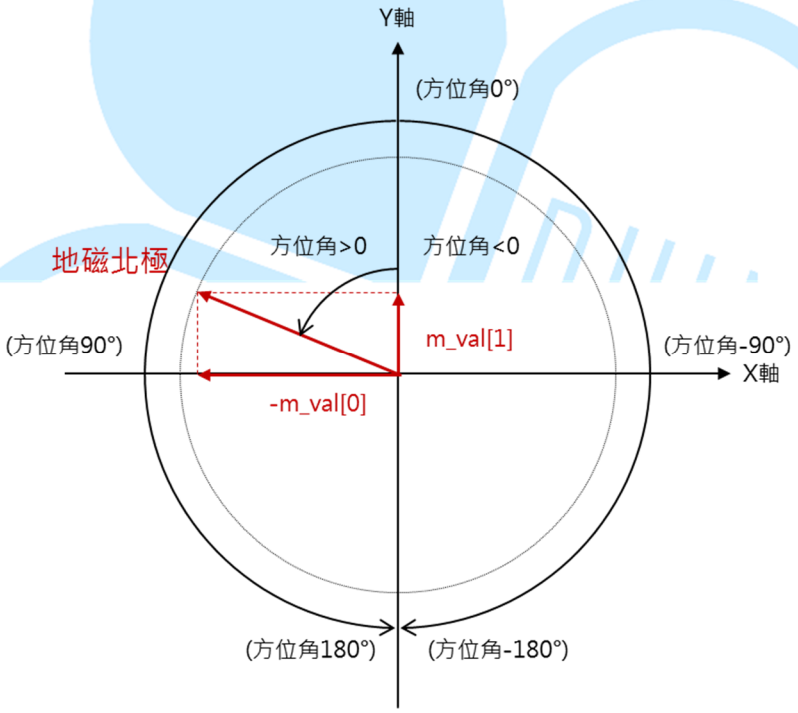
Where the *m_val variable is the pointer to the array that contains sensor data, and is used to calculate the direction based on sensor data in the following line of code:

```
azimuth = atan2(-m_val[0], m_val[1]);
```

The angle is calculated using trigonometric arc tangent, atan2(b,a). As shown in the figure below, the angle and tangent of a triangle can be calculated based on the length of a and b, which is functionally equivalent to $\tan^{-1}(b/a)$, where you can get result in -π~π.




Sensor data from the digital compass' X and Y axes can be used in the above formula, as shown in the following figure.




Since the direction representation in degree is increase (positive) in the counter-clockwise direction, in the atan2 formula, sensor value for the X axis is multiply by -1. Since the value returned from the atan2 function is in radian, the following formula is used to convert the radian value to degree, a more common parameter used to identify direction.

```
azimuth = azimuth * 180 / PI;
```

It's important to know the Earth's North bound geomagnetic field does not point exactly to the North at all region of the Earth. There are different magnetic declination in different region. To calculate accurate heading based on magnetic sensor data, you need to consider the affect caused by magnetic declination. More information about magnetic declination is available on the following URL:

- http://www.ngdc.noaa.gov/geomag-web/#declination

In the program's main loop() function, there are multiple command to turn on LED based on sensor data which function like a simple compass. Using the same sensor data, instead of the LED, you can use an LCD to provide visual representation or a buzzer that produce different audio intensity or frequency to indicate whether the sensor is pointing to the North.

For this exercise, sensor data from the accelerometer is not used. Based on the simple calculation used in this exercise, the result is most accurate when the sensor is level in the horizontal position, where the accelerometer can be used to detect whether the RM-G144 module is tilt or in the horizontal position. Combining sensor data from the accelerometer sensor, with additional code to compensate the angle of inclination, you can get more accurate direction as the sensor is being moved around. However, these calculation involve more complex codes, an area where you can further explore to figure out how to combine these sensors data to gain better result.