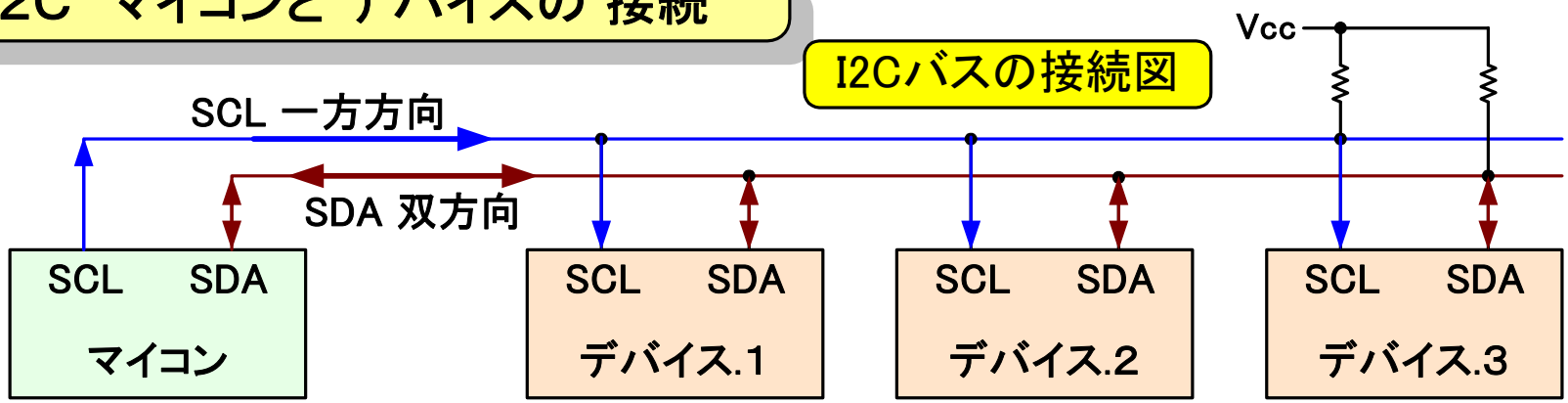
I2C マイコンと デバイスの 接続



I2Cは SCL:シリアルクロック信号、SDA: 双方向データ信号の 2本の信号線で構成されます。 標準的な転送速度は 400Kbpsです。 デバイスによっては、更に早い物もあります。 また、Read/Writeコマンドに 7bitのアドレスも付くので、2線に アドレスの異なる複数のデバイスを接続する事が出来ます。

プルアップ抵抗は必要ですが、デバイスに内蔵されている場合も ありま す。 その場合は、デバイスの複数接続を考慮して やや高めの抵抗値にしてあります。

I ２ C	
長所	① 信号線2本＋GNDでデバイスと通信可能。 ② 複数デバイスを接続する事が可能。 複数デバイスを接続しても、信号線は2本のままで、OK。 但し 各デバイスのアドレスは、重複させてはならない。
短所	① SPI と比べると データ転送速度が遅い。

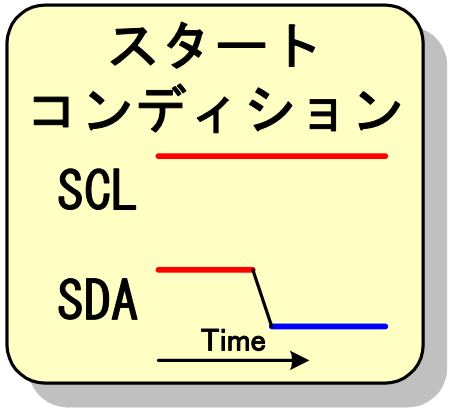
という事で、データ転送速度が あまり問題にならなければ、I2Cは 気軽に使えると思います。

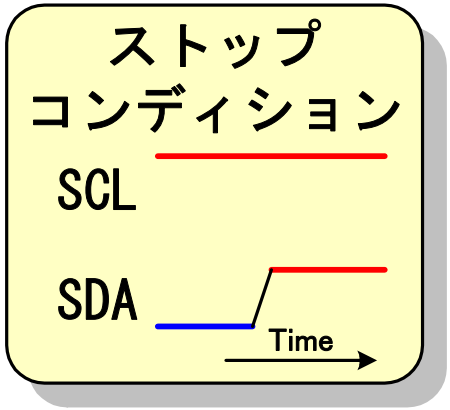
I2C通信シーケンス(1)

I2C通信は、SCLと SDAの２本の信号線を用います。待機中 SCLと SDAは、両方とも Hiレベルです。

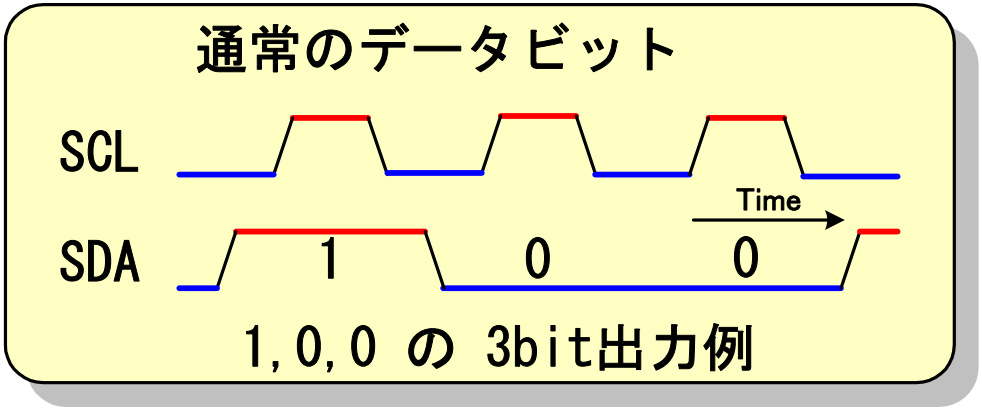
[1] スタートコンディション：
今から通信シーケンスを開始する事をマスタが、スレーブに通知するための信号です。 SCLが、Hiの期間中に SDAを Hiから Lowに変化させます。

[2] ストップコンディション：
マスタが、スレーブに対し通信を終了させる時に出します。 SCLが、Hiの期間中に SDAを Lowから Hiに変化させます。





通常のデータビットでは、SCLが Lowの期間中に、SDAを変化させるので、データビットと、スタート／ストップ コンディションは、区別出来ます。

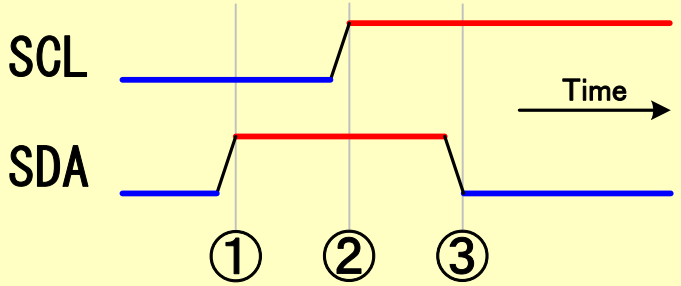


I2C通信シーケンス(2)

[3] リピートスタートコンディション：
　8ピンの EEPROMをアクセスする際に
リピートスタートコンディションを発行
する場合があります。

① SCLが、Lowの期間に一旦、SDAをHi
　にします。
② SCLを Hiにします。
③ SDAを Lowにします。

リピートスタートコンディション



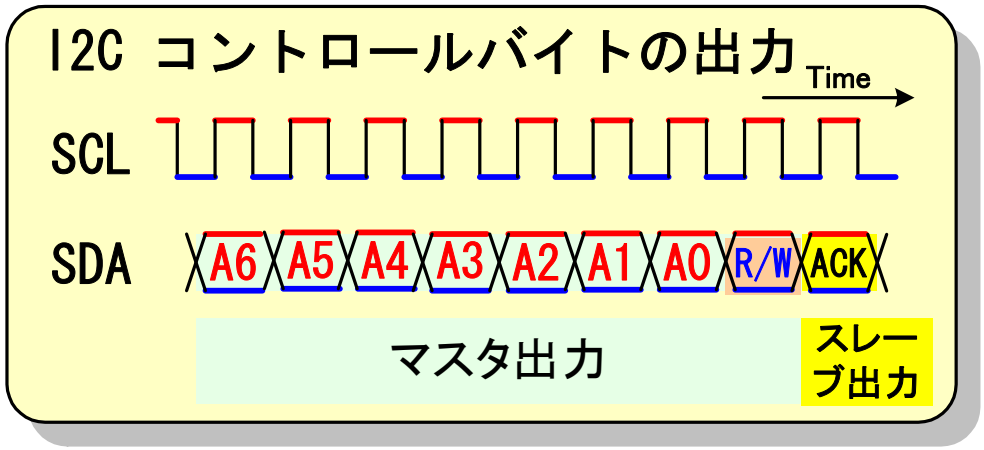
最近は、殆どのマイコンに、データ用フラッシュROMが入っている事もあり
外付けで 8pinのシリアルEEPROMを使う事が、少なくなってきました。
　これにより、リピートスタートコンディションを使う機会も減ったように思います。

　いかし、まだリピートスタートコンディションが 必要なデバイスが、一部存在します。 殆どの場合、コマンドWriteから、データ Readに 連続して切り替える用途で 使われます。

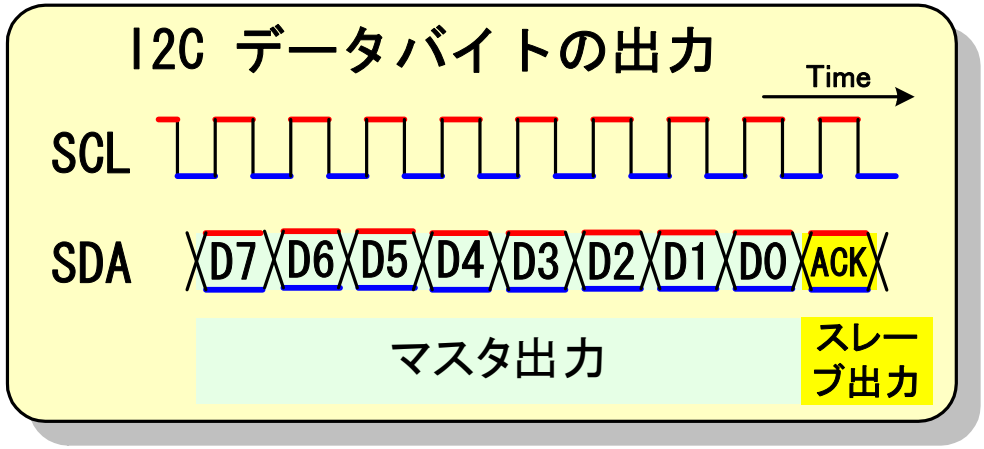
I2C通信シーケンス(3)

[4] I2Cコントロールバイト：
スタートコンディション直後、最初に出力するバイトデータが、コントロールバイトです。今回は、7bitアドレスで説明します。 10bitアドレスも規格上はありますが、私は使った事が無いです。

① 一旦 SCLをLowに降ろします。
② スレーブのI2CアドレスのA6 ～ A0の 7bitを 順次 bit単位でスレーブに書き込みます。
③ 次にデータを書込む際は、Write(SDA=Low)、読出す際は、Read(SDA=Hi)を、1bit 出力します。スレーブからの ACK/NAK(1bit)を受け取ります。

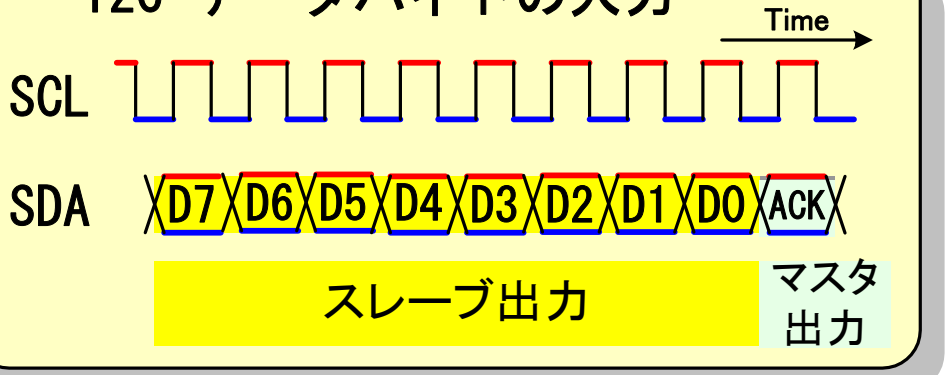


[5] データバイト出力（Write）：
内容(データ)が異なるだけで、コントロールバイト出力と同じです。



I2C通信シーケンス(4)

[6] データバイト入力（Read）：
SDAの出力元が、入れ替わるだけで
シーケンスは、同じです。



[7] 一連の電文シーケンス例：
I2Cスレーブアドレス 3Ch に、
40h、41hのデータ2byteを 書き込む
例です。

① スタートコンディションを実行。
② 7bitAddress = 3CHでコントロール
バイト(Write)を、出力します。
③ データ40hを データバイトとして
出力します。
④ データ41hを データバイトとして
出力します。
⑤ ストップコンディションを実行。

ACK／NAKに関して：
通常、通信制御コードの ACK、NAKは、肯定応答、否定応答の意味で、送り元が、受信側からNAKを受け取った場合は、再送信等のエラーリカバリ処理を行います。が、I2Cはどちらかというと、転送する最終バイト識別の意味合いで用います。

ESP32の I2C出力ピン

ESP32にて、I2Cインタフェースを使用する時は 使用するピンが、決まっています。 今回使用する ESP32は ESP32-WROOM-32の DEV-KIT 30ピンの基板です。

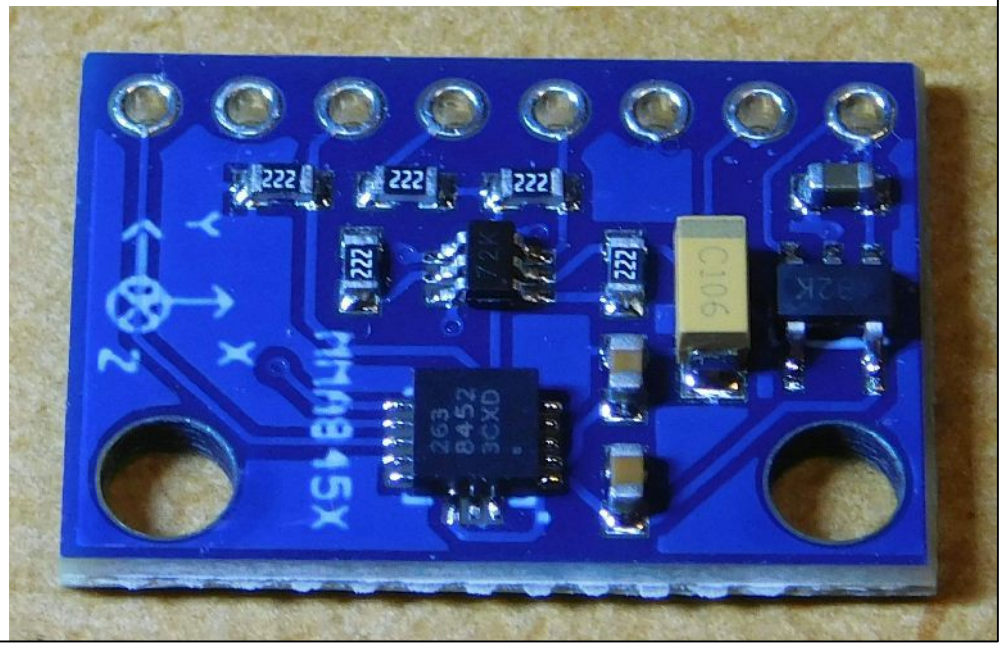
GPIO22 が I2C SCL です。
GPIO21 が I2C SDA です。

ESP32 & Arduino IDE環境で 使用する I2Cの ライブラリは Wire です。
よって インクルードするファイルは
#include <Wire.h> です。

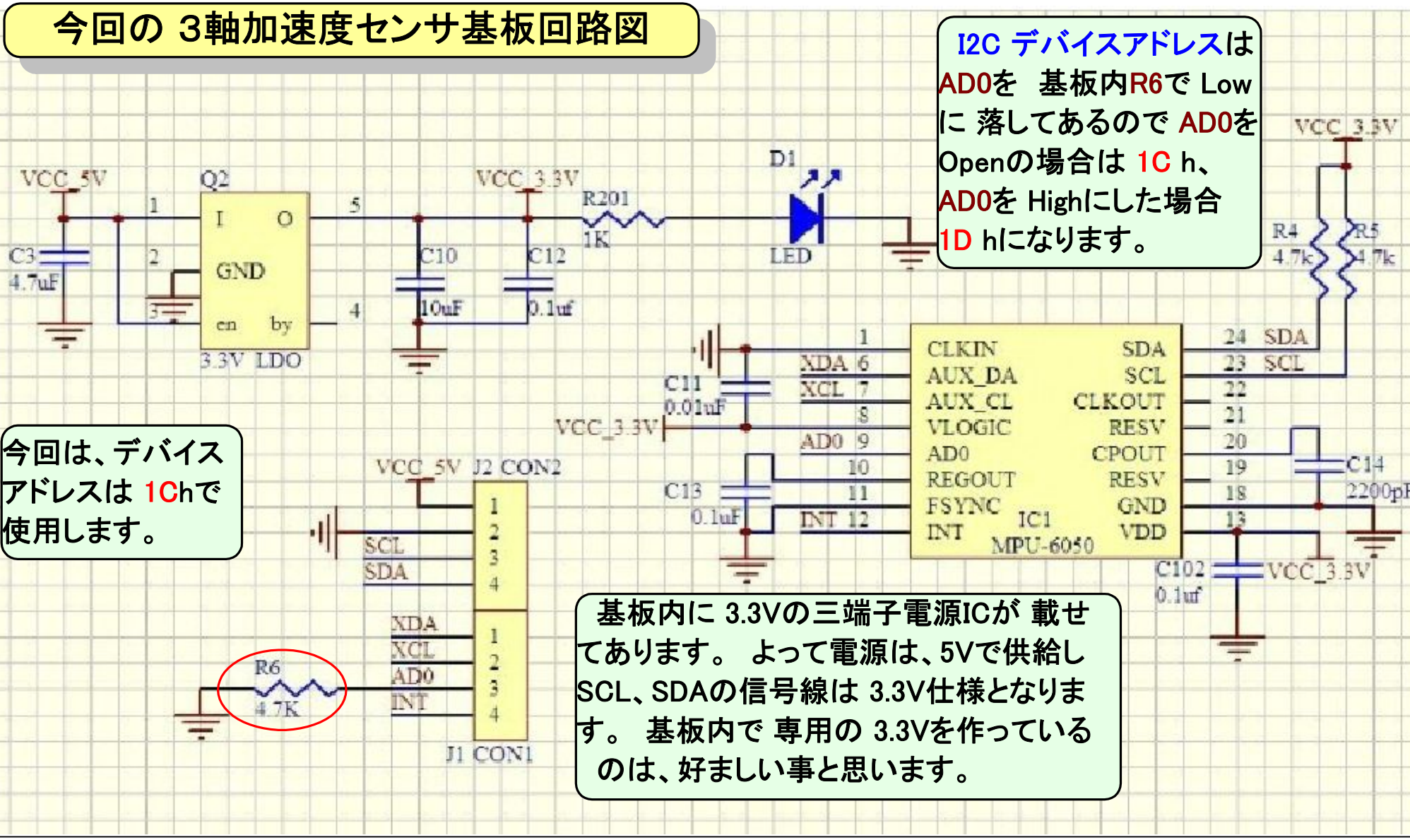
この Wire という ライブラリは ラズベリーパイでも 同様の物が ありました。

使用する I2C 3軸加速度センサ基板

今回、使用する I2C 3軸加速度センサ基板は MM8452Qという ICを付けた_3軸加速度センサ基板です。Amazonで 4個 999円を 買いました。ちゃんと動くかな。？ ちょっと不安。
今回の I2C 3軸加速度センサ基板の、画像です。 コネクタピンの 信号名は、基板裏側に書いてあります。



今回の 3軸加速度センサ基板回路図



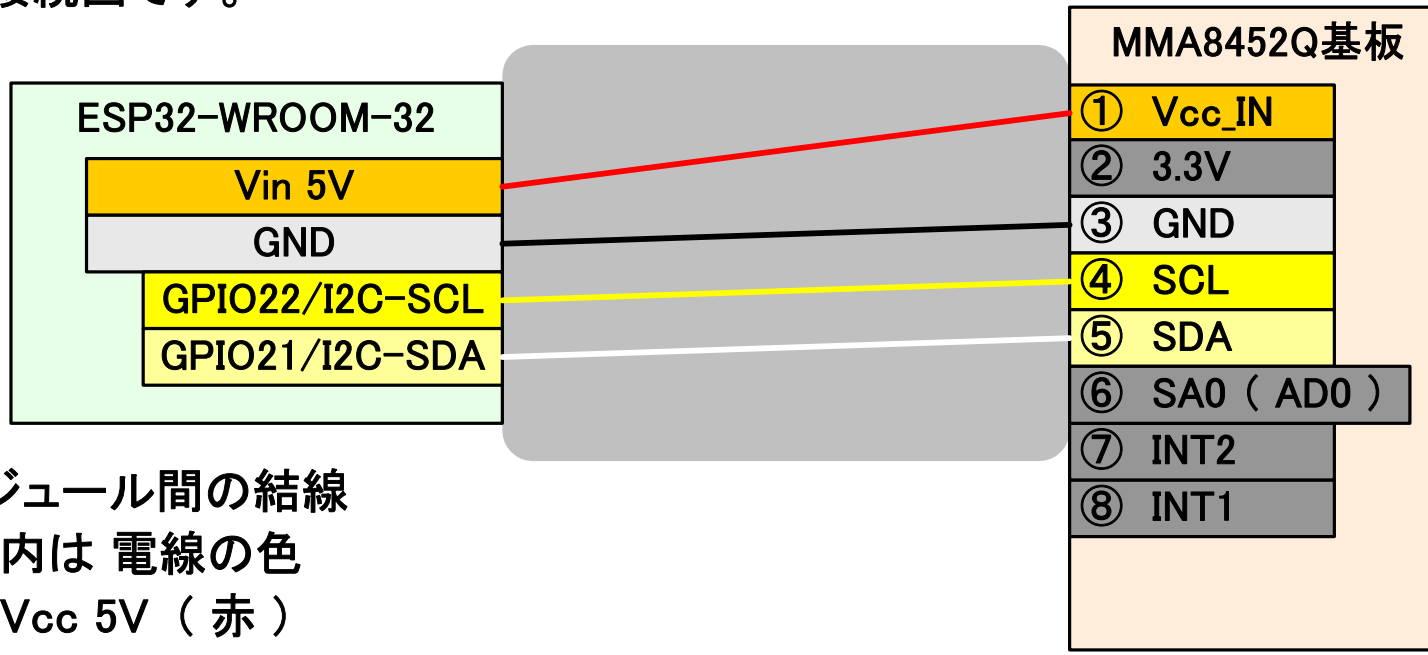
I2C デバイスアドレスは AD0を 基板内R6で Low に 落してあるので AD0を Openの場合は 1C h、AD0を Highにした場合 1D hになります。

今回は、デバイスアドレスは 1Chで使用します。

基板内に 3.3Vの三端子電源ICが 載せてあります。 よって電源は、5Vで供給し SCL、SDAの信号線は 3.3V仕様となります。 基板内で 専用の 3.3Vを作っているのは、好ましい事と思います。

ESP32 と MMA8452Q基板の接続

ESP32と 3軸加速度センサ MMA8452Q基板の接続図です。



センサ基板 裏側

モジュール間の結線
（）内は 電線の色
Vcc 5V （ 赤 ）
GND （ 黒 ）
SCL （ 黄色 ）
SDA （ 白 ）
の 4本です。

今回は、MMA8452Q基板に 必要な抵抗、コンデンサ等が 一通り付いているので、小基板に部品を追加する必要は 無かったです。

MMA8452Qの初期設定と データ読み出し

秋月電子でも、MMA8452Qを使った基板を、過去に販売していて、その商品の簡易 取扱い説明書を サイトで公開していたのでダウンロードしました。 今は 売って無いようです。

その取扱い説明書の裏面に Arduinoを対象としたサンプルプログラムを載せてありました。
そのプログラムをベースに使って MMA8452Q を アクセスしようと思います。

まず、秋月電子の サンプルプログラムをお見せします。

右のプログラムソースで setup() 内にて、Serial.begin(38400); は 115200 に 変更します。 ESP32-WROOM-32の標準的な ボーレイトは、115200 に なっているからです。

```
#include <Wire.h>                    // Source ( 1/5 )

// MMA8452のI2Cスレーブアドレスを設定します。
// 基板ジャンパSJ1が未接続(デフォルト)なら0x1D
// はんだで接続したら 0x1Cです。

#define MMA8452_ADRS                       0x1D
// MMA8452の内部レジスタアクセスと加速度算出
// に使う定数です。
#define  MMA8452_OUT_X_MSB                 0x01
#define  MMA8452_XYZ_DATA_CFG              0x0E
#define  MMA8452_CTRL_REG1                 0x2A
#define  MMA8452_CTRL_REG1_ACTV_BIT        0x01
#define  MMA8452_G_SCALE                   2

void  setup()
{
    byte tmp;

    // UARTのボーレートは、38400bpsに設定します。
    Serial.begin(38400);
    Wire.begin();
```

```
                                        // Source ( 2/5 )

  // MMA8452の内部レジスタを設定します。

  tmp = MMA8452_ReadByte( MMA8452_CTRL_REG1 );

  MMA8452_WriteByte( MMA8452_CTRL_REG1, tmp &
      ~(MMA8452_CTRL_REG1_ACTV_BIT) );

  MMA8452_WriteByte( MMA8452_XYZ_DATA_CFG,
      (MMA8452_G_SCALE >> 2) );

  tmp = MMA8452_ReadByte( MMA8452_CTRL_REG1 );

  MMA8452_WriteByte( MMA8452_CTRL_REG1, tmp |
      MMA8452_CTRL_REG1_ACTV_BIT );

}
```

ここで使用している定数宣言を 表示します。

```
#define  MMA8452_CTRL_REG1              0x2A
#define  MMA8452_CTRL_REG1_ACTV_BIT  0x01
```

上記、定数は、左のソースの 赤で囲んだ中で使用されてます。 頭の MMA8452は 省略しますが、CTRL_REG1 は MMA8452内の制御レジスタ1です。

最初、制御レジスタ1を 読み出して byte tmp に格納しています。 そして CTRL_REG1_ACTV_BIT (01h)の ビット反転を行い tmp と ANDを 取り制御レジスタ1 に 書き込んでいます。 要は、制御レジスタ1の 最下位 bit だけを 0 にしているという事です。 同様に 左下の 赤枠内は、制御レジスタ1の 最下位 bit だけを 1 にしているという事です。

2つの 赤枠の間の行は

```
#define  MMA8452_XYZ_DATA_CFG  0x0E
```

で XYZ_DATA_CFG レジスタに (2 >> 2)で 0 を 書き込んでいる事になります。

```
void  loop()                              // Source ( 3/5 )
{
  byte    buf[6];
  float   g[3];

  // MMA8452の内部レジスタにある測定値を読み込みます。
  // X: g[0], Y: g[1], Z: g[2] に対応します。
```

MMA8452Qからのデータ取り込み

```
MMA8452_ReadByteArray( MMA8452_OUT_X_MSB, 6, buf );
```

X軸データ変換

```
g[0] = -(float((int( (buf[0] << 8) | buf[1]) >> 4))/
        ( (1 << 11) / MMA8452_G_SCALE ) );
```

Y軸データ変換

```
g[1] = -(float((int( (buf[2] << 8) | buf[3]) >> 4))/
       ( (1 << 11) / MMA8452_G_SCALE ) );
```

Z軸データ変換

```
g[2] = -(float((int( (buf[4] << 8) | buf[5]) >> 4))/
        ((1 << 11) / MMA8452_G_SCALE ) );
```

ループ先頭で宣言されている、byte buf[6]; は MMA8452内の データレジスタを読み出すバッファです。 float g[3]; は X,Y,Zの 各軸加速度データに 変換した値を入れる変数です。変数名の g は多分 CGS単位系の 加速度の単位 gal から g とされたのでしょう。 因みに 3軸加速度センサ基板を 水平に設置された場合、Z軸が上下軸に なります。 その状態で加速度センサの値を読み出すと、Z軸にだけ -1が 定常的に出力されます。これは、引力加速度 1G です。 Z軸の 下方向に引っ張られるため、-1G と なります。

MMA8452_ReadByteArray()は 配列データをbyte単位で連続して転送する I2C関数です。第一引数が 送り元の MMA8452Q内の レジスタ値です。 第二引数が 転送 byte数です。 第三引数が、送り先の buf 変数の先頭アドレスです。

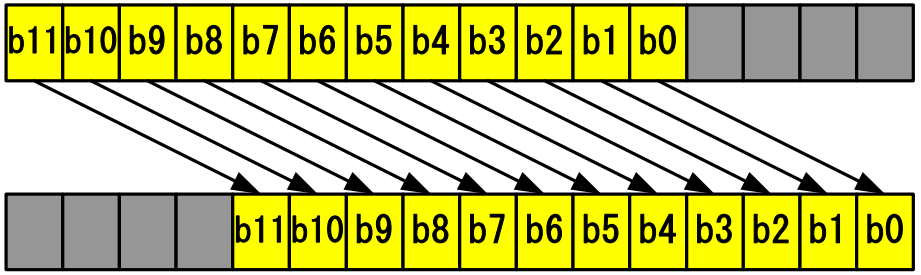
これにより、X,Y,Z 各軸のデータが 2byteで 計6byte buf 配列に 転送されます。 その下のデータ変換計算は 次のページで 説明します。

```
g[0] = -(float((int( (buf[0] << 8) | buf[1]) >> 4))/
        ( (1 << 11) / MMA8452_G_SCALE ) );
```

データ変換の式は X、Y、Z ありますが、入力の配列と 出力の配列の添え字が異なるだけで、同じ計算を行っています。 このややこしい見た目の式は bitの並べ替えと、スケール変換、極性反転の ３ つの事をやっています。

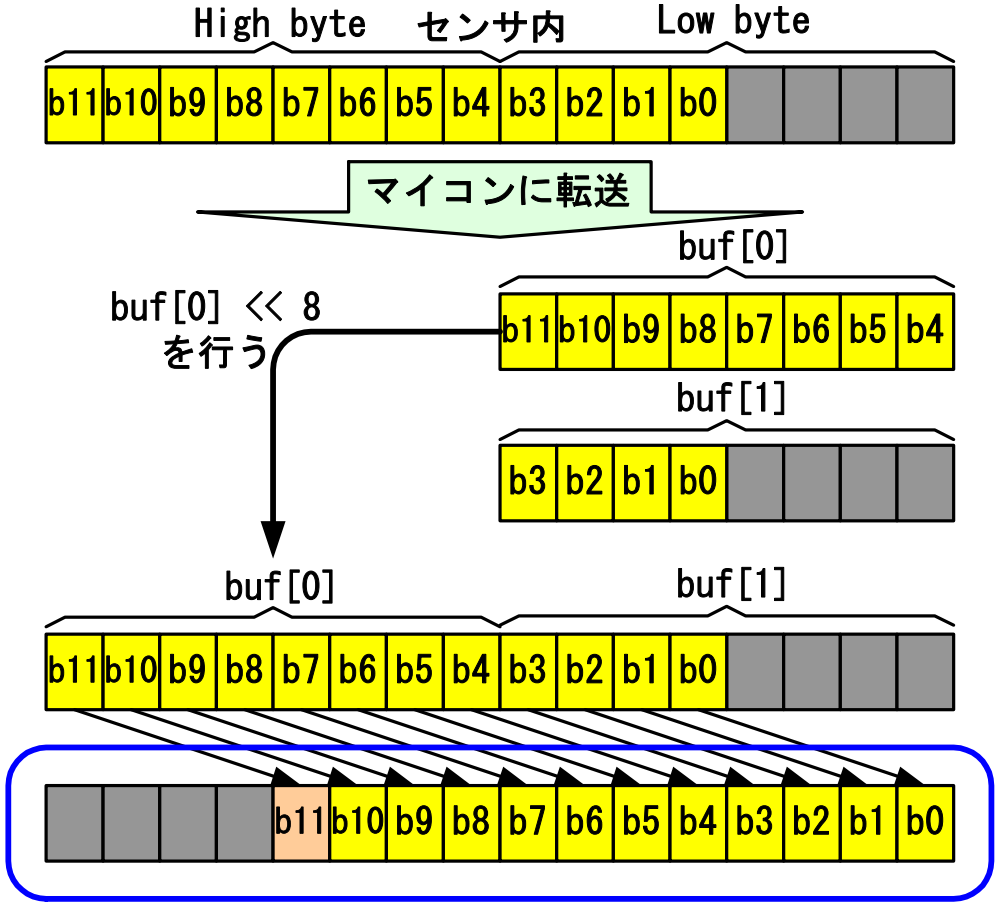
① bitの並べ替え： X,Y,Z の 各要素データは 16bit で読み出していますが、有効なデータは 12bit です。そして 有効なデータ 12bitの 最上位 bit の MSBは、16 bitの最上位bit MSBと 重なっています。 で、パソコンや、マイコンで データを扱う時は、12bit 最下位 bitの LSBと、16bit 変数のLSBを 重ねます。 センサーデバイスの場合 最上位の MSBから 有効データを 詰めて bit を並べてある場合が多いです。 これは、逐次比較型の A/Dコンバータの変換シーケンスを考えるとその方が自然なんです。 どういう事かと 言うと

逐次比較型A/Dコンバータは、電圧を2分探索的に 1/2、1/4、1/8、1/16と 比較判定していくので、大きい重みを持ったデータから、順に値が決まって行きます。 その関係で 最上位から並べる方が都合がいい。という事です。 この並びは デジタルオーディオ用のデバイスも 同じ並びになってます。その MSB を 重ねた並びを LSBを 重ねた並びにしないと パソコン、マイコンでは 扱いにくいという事です。



上の図を見ると 単純に 4bit 右シフトすれば良さそうな気がしますが、そう単純に行かない理由がもう一つあるのです。 この手のセンサーデバイスは、ワードデータ内の バイト並びが ビッグエンディアンなのです。 ESP32は リトルエンディアンです。 その関係で、buf[0]を 左シフト 8 を やってあるのです。

前ページの bit並び図を、エンディアンの変換で上下バイトを 入れ替える処理も含めた形で描くと、下の図のようになります。




```
g[0] = -(float((int( (buf[0] << 8) | buf[1]) >> 4))/
        ( (1 << 11) / MMA8452_G_SCALE ) );
```

最終的に 左下の 16bit整数型の変数の 下位 12bit が 有効なデータです。 で、12bit の MSB b11は、サインbitです。 int()で 囲った中で 4bit 右シフトしているのは、最上位のサインビットを 保持しながらシフトするためです。 このあたりのビット操作は アセンブラと2の補数が、分かる方であればイメージがつかみやすいと思いますけど、高級言語から入ってきた方には、難しいかもしれませんね。 で、続きの (1 << 11) ですが 2^{11} = 2048 の 事です。 MM8452_G_SCALE = 2 なので 上の式の 2行目は 1024 になります。 MM8452_G_SCALE = 2 は レンジ設定で、±2Gの レンジです。 それが 2の歩数の量子化数で -2048 ～ 2047 に なります。 この値を floatに変換して2行目の式の 1024 で割ると 約 -2 ～ +2 G の加速度を 表す事になります。 右辺の 最初に - が あるのは、絶対値は そのままに 極性だけ反転させているという事です。 2行の式の説明が 長くなりましたね。

```
                                    // Source ( 4/5 )
void MMA8452_ReadByteArray( byte  adrs,
     int  datlen, byte  *dest )
{
    Wire.beginTransmission(MMA8452_ADRS);
    Wire.write(adrs);
    Wire.endTransmission(false);
    Wire.requestFrom(MMA8452_ADRS, datlen);
    while( Wire.available() < datlen );
    for( int x = 0; x < datlen ; x++ )
        dest[x] = Wire.read();
}

byte MMA8452_ReadByte( byte  adrs )
{
    Wire.beginTransmission( MMA8452_ADRS );
    Wire.write( adrs );
    Wire.endTransmission( false );
    Wire.requestFrom( MMA8452_ADRS, 1 );
    while( !Wire.available() );

    return( Wire.read() );
}
```

```
                                    // Source ( 5/5 )
void MMA8452_WriteByte( byte  adrs, byte dat )
{
    Wire.beginTransmission( MMA8452_ADRS );
    Wire.write( adrs );
    Wire.write( dat );
    Wire.endTransmission();
}
```

I2C　連続した１ブロックデータの読み出し

```
void MMA8452_ReadByteArray( byte  adrs,
     int  datlen, byte  *dest );
```

i2c 1byte 読み込み

```
byte MMA8452_ReadByte( byte  adrs );
```

I2C　1byte 書き出し

```
void MMA8452_WriteByte( byte  adrs,
     byte dat );
```

上記 ３つの関数は、Wireライブラリ関数を I2C の伝送シーケンスとして 一まとめにした関数です。