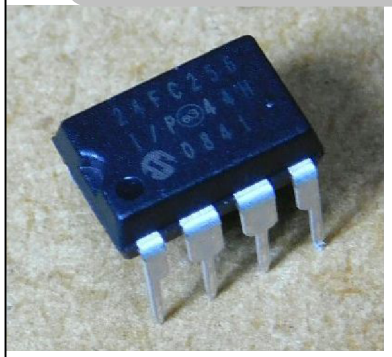
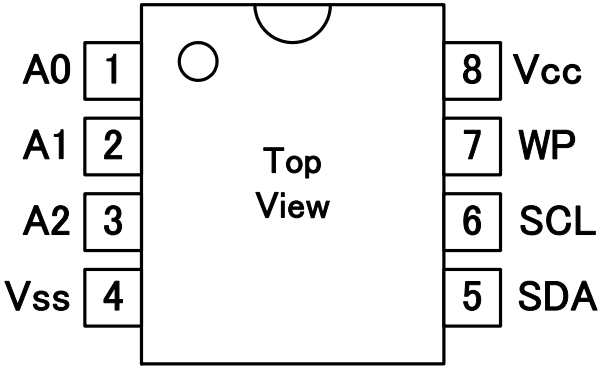
I2C シリアルEEPROM 24FC256



左の画像は マイクロチップ社の I2C シリアルEEPROM 24FC256-IP です。
8ピンの ICです。
256Kbit の 容量なので Byte 換算で 32Kbyteです。
2.5 ～ 5.5Vの範囲内で 400Kbpsの 転送速度で 動作するようです。温度範囲は -40～85℃
ESD保護＞4,000V、データ保存＞200年
100万回の消去／書き込みサイクル



パッケージは、通常の DIP以外に SOPや その他の形状の物もあります。

記憶容量は 大したことないですが、温度範囲が -40～+85℃ 自動車用は、-40～+120℃、ESD（静電気放電）耐圧が 4,000V以上、データ保存 200年以上、消去／書き込みサイクルが 100万回とか、過酷な環境でも、使用する事を想定しているのかな。 とも思いますが、すごいです。 他にもデータシートにいろいろ書いてありますが、使用する上で最低限必要な事だけ書いておきます。

このタイプの シリアルEEPROMには ページというメモリの単位があります。今回の 24FC256は 64byteが 1ページになってます。
このページというのは、書き込みに関わってきます。 読み出し時は関係ないです。 byte単位で アクセス出来るようです。 EEPROMというのは、書き込み時 一連のシーケンスが必要で ミリ秒程度の時間が かかります。

遥か昔の 窓付きEPROMは、マイコン側で書き込みシーケンスのプログラムを 作った事があります。 やや面倒くさい処理だったと思います。

I2Cの シリアルEEPROMは、書き込みシーケンスに関しては EPROM内のマイコンが やってくれるので、面倒な書き込みシーケンスに関して考える必要は ありません。 一つ意識しておいて欲しいのは、先ほども書きましたが、書き込みに関して ミリ秒程度の時間が かかる事です。 で、1byte毎に書き込みシーケンスを行うと やたら遅くなるので、ある程度まとまったバイト数で 書き込みシーケンスを 行います。そのまとまったバイト数というのが 64byteの ページです。その関係で シリアルEEPROM内には、64byteのバッファRAMが あります。

I2Cバスにて 高速に データを バッファRAMに書き込み、シリアルEEPROM内の バッファRAMから EEPROMに 書き込みシーケンスを行います。 で、内部の書き込みシーケンスを 実行している最中は、I2Cバス側から シリアルEEPROMを 見ると busy状態になっており、I2Cの通信シーケンスに反応しません。Write bufferも 64byteしかないので 書き終わるまで待たされる事に なります。 では、書き終わったかどうか確認するのはどうするのか。 というと データシートにACKのポーリングを行う。 と書いてあります。

これは、どういう事かというと、EEPROM側が書き込みシーケンスで 忙しいと、マスタ側で I2Cの コマンドバイトを送ると 受け取った合図のACK(Lowの 1bit)を 返してくれないのです。ACKが 返らない状態は まだEEPROMは Busy状態と判断する事が出来る。 という事です。

具体的に、ACKのポーリングは どのように行うのかというと、C言語にて 関数を 1本用意します。 引数として EEPROMの I2C スレーブアドレス(今回は 50h)を渡し コントロールバイトを出した後の ACK/NAK を 関数値として返す関数です。 関数内の手順は

① スタートコンディション発行
② I2Cコマンドbyteの送信
　(7bit スレーブアドレスと Write bit)
③ スレーブからの ACK or NAK 受信
④ ストップコンディション発行
⑤ そして ACK or NAKを 関数値として返す。

という事になります。

実は、この機能を持つ関数は、既に 作ってあります。 i2c_packet.c 内の

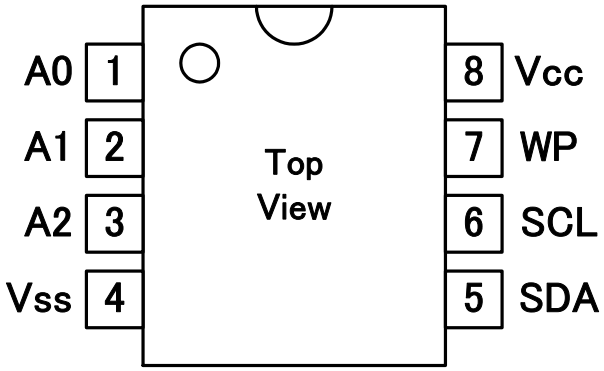
```
_UBYTE  i2c_check_slave( _UBYTE adr );
// 指定アドレスの スレーブ有無の確認
```

が それです。 最初は I2Cバスに接続される全てのスレーブアドレスを サーチするために作成しましたが、今回の用途でも 使えそうです。 以下が 関数の内容です。

```
//*****************************************
//**  I2C 7bitアドレスのスレーブ確認     **
//** ------------------------------------ **
//**  引数  adr :  スレーブアドレス       **
//**  関数値 :  = 1   : Slave 有り        **
//**            = 0   : Slave 無し        **
//*****************************************
_UBYTE  i2c_check_slave( _UBYTE adr )
{
   _UBYTE  sts;

   i2c_start_cond();  // I2C スタートコンディション
   sts = i2c_wr_adr7( adr );
                      // Slave有無確認の 仮アクセス
   i2c_stop_cond();   // I2C ストップコンディション

   return  sts;
}
```



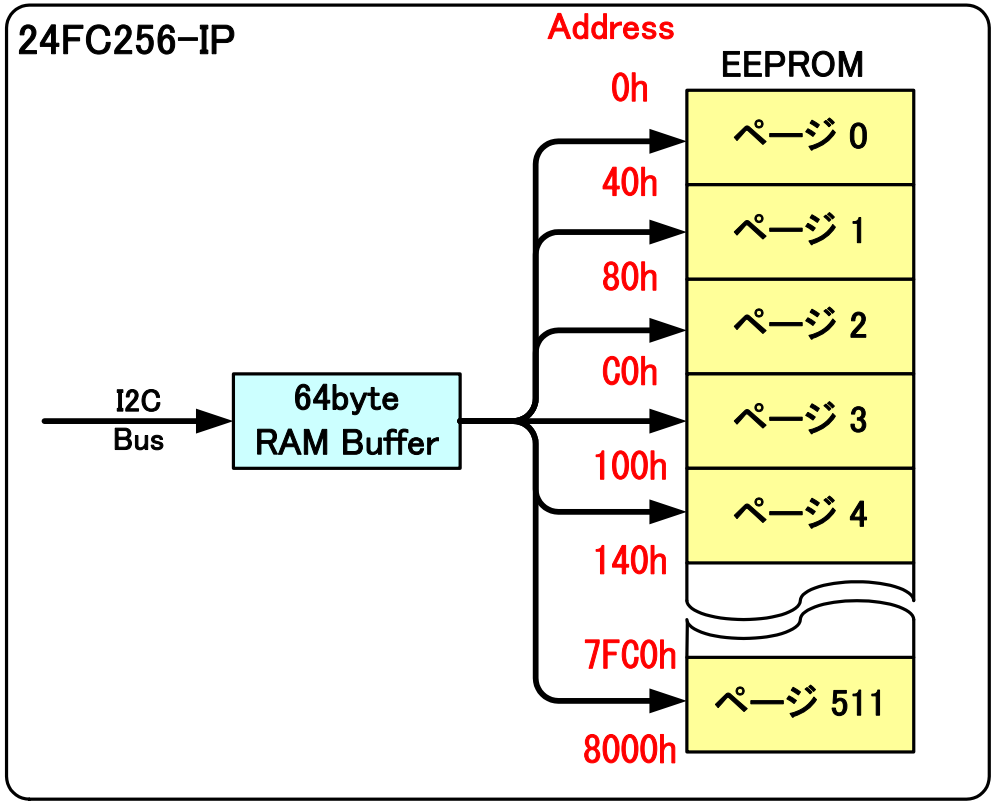
24FC256の 足ピンに関しての説明:

1～3ピンの A0、A1、A2 は、I2Cの デバイスアドレスの 下位 3bitを設定できるという事です。

今回の用途では、1～3ピンを グランドに 接続しているので 下位 3bitは 000 です。

上位 4bitが 1010 固定なので デバイスアドレスは 50hに なります。 仮に A2=0、A1=0、A0=1 の場合は 51hに なります。

あと WP は Write Protect 機能で、Hi に すると 書き込みが 出来なくなります。 常時 読み書き出来る様に するには WPを グランドに接続して下さい。



I2Cによる シリアルEEPROMの読み書きの電文もデータ長は 64byte単位にして転送しようと思います。 64byteに 満たない端数は その長さでも、書き込み出来ます。

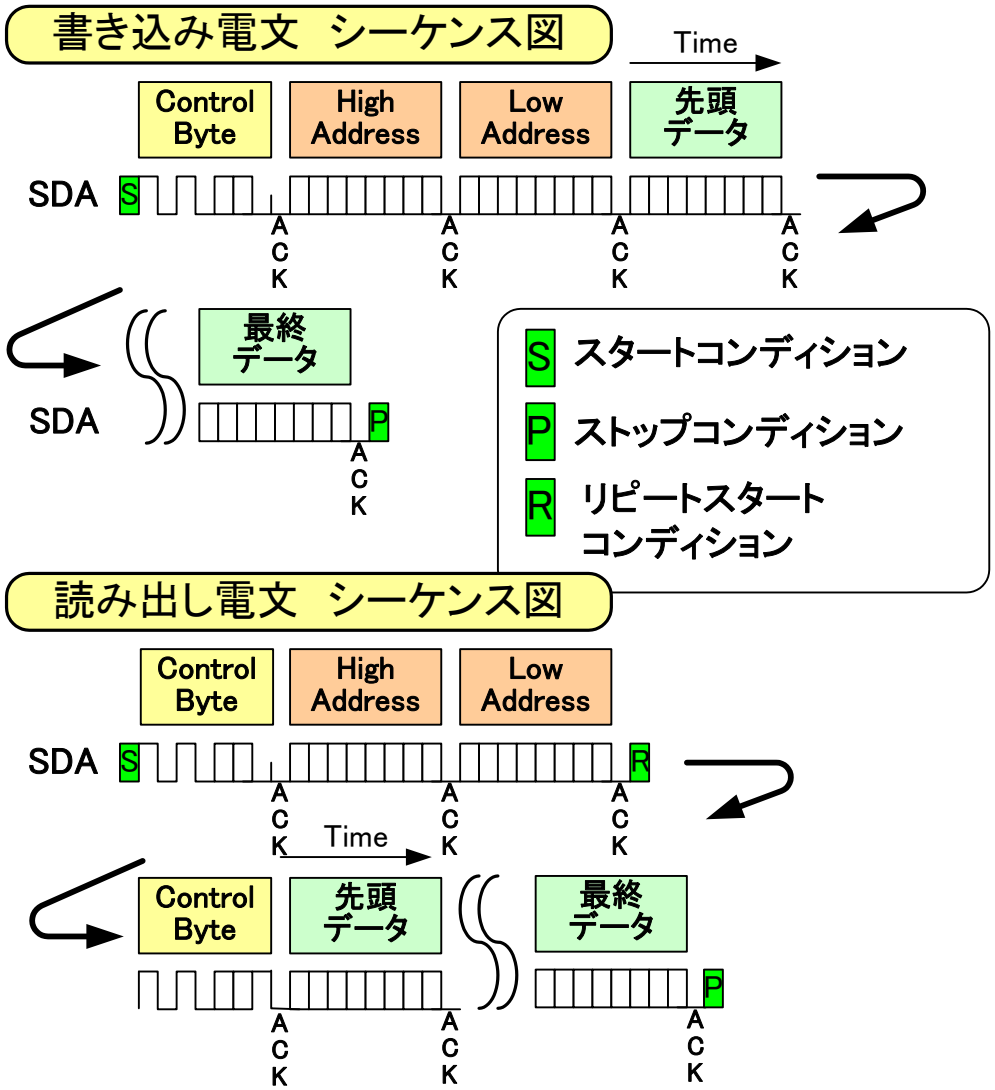
64byteに 満たない端数は その長さでも、書き込み出来ます。と 説明しましたが、物理的には、64byte単位で書き込むので、64byteの buffer RAM内に 残っていたゴミも 一緒に 64byte として書き込みます。

例えば、100byteの データを書き込む場合は 2ページ分で、1回目の電文で 64byte転送して 2回目が 100 - 64 = 36 で 36byte転送して書き込む事になります。 実際は 64byteの単位で 書き込むので 後半の 28byteの エリアに 残っているゴミも 一緒に書き込みます。

1回目書き込み: 64Byte有効データ

2回目書き込み: 36Byte有効 | 28Byteゴミ

ちょっと 細かくて見にくいですが、右に 書き込み、読み出しの I2C電文を 示します。



BME280 温度、湿度、気圧センサーデバイス

センサーデバイスは　何にしようかと思ってましたが、温度、湿度、気圧センサーデバイスのBME280に しました。 ドイツの　ボッシュというメーカーの製品です。　マイコン等のパーツを扱っている通販では、このセンサーデバイスはよく見ます。 Arduinoでも　この BME280の スケッチもある様で　Arduino上であれば　あっさり使う事が 出来るでしょう。

H8マイコンで　この BME280 を 動かすため Arduinoの ソースを　そのまま取り込もうとすると、しこたま コンパイルエラーが 出ます。コンパイルエラーに　ならないエラーも あります。

まずは、Arduinoの BME280ソース内のコーディングで、I2Cから　取り込んだデータを　やたら長い bit長の シフト演算をしている箇所がいくつか あります。

これは、何をやっているのかというと　BME280の整数データは　ビッグエンディアンの並びなのです。よって byte単位の データを bitシフト演算で、上下バイトを差し替えているのです。

2byte整数もありますが、4byte整数もあるのでやたら長い bitシフト演算が あるのです。 それらの演算式は　もう一つ役割が あって、byteデータ上位、下位を連結して　Word データとして取り出したいという事です。

先に　必要無い処理を お伝えします。
整数に関わる　上下バイトの 入れ替えは　必要ありません。 何故なら　H8マイコンも　ビッグエンディアン だからです。

但し、byteデータ上位、下位を連結して　Word データまたは　DWord データ として取り出す処理は　必要です。

で、もう少しスッキリした記述に したかったので、構造体と 共用体を組み合わせたデータでbyteデータを並べ直し、同じアドレスに存在するWord または DWordの変数から、2byteまたは 4byteの整数データを取り出す事に します。

右に 構造体、共用体の 型宣言を示します。一見、難しそうに見えますが、メモリ上のイメージが 掴めると 単純です。2byteの例で示すと、TYP_CHG2　Bw; という変数を 宣言します。Bw.b.h = 上位バイト、Bw.b.l = 下位バイトを代入します。 そして 同じアドレスの　Bw.w にて　2byte 整数データを　取り出せます。

右の 2byte、4byteの構造体は　既に　H8マイコン仕様(ビッグエンディアン)に なってます。
因みに、リトルエンディアンの場合は、並びが下位、上位なので　_UBYTE　l、h; に　なります。

```
typedef struct  {     // 2byte 構造体データ
      _UBYTE     h, l;
} BYTE_2;

typedef union   {     // 2byte 共用体データ
      BYTE_2     b;
      _UWORD     w;
} TYP_CHG2;

typedef struct  {     // 4byte 構造体データ
      _UBYTE     hh, hm, ml, ll;
} BYTE_4;

typedef union   {     // 4byte 共用体データ
      BYTE_4     b;
      _UDWORD    lng;
} TYP_CHG4;
```

H8マイコンで BME280の Arduinoソースを そのまま取り込もうとすると、しこたま コンパイルエラーが 出る。と 言いましたが、どういうところで出るかというと、整数の型宣言で 引っかかります。 Arduinoソースの 大元は ボッシュの BME280の データシートに記載されていた ソースです。

で、整数の型宣言が 特殊で HEWの H8マイコン Cコンパイラでは 全く 受け付けてくれません。 よって、H8マイコン Cコンパイラの プロジェクトに 標準で入っている typedefine.h で、定義されている 特殊な整数データ型に 変更する事にしました。 右側に 示しますが
左が Arduinoソースの整数データ型
右が HEW環境の 整数データ型です。

特殊なと書きましたが、これは 今までが intの 長さというか byte数が CPUにより 異なる長さに なる事を 懸念して 明確に bit数が 分かる 整数宣言名を gccの環境で 試験的に 作成された物と思われます。 私も ハッキリした事は 分かりません。 Microsoftの 見解では 標準では無い。が、一応 使えるようにしておく。との事でした。

```
int8_t    --->  _SBYTE
uint8_t   --->  _UBYTE

int16_t   --->  _SWORD
uint16_t  --->  _UWORD

int32_t   --->  _SDWORD
uint32_t  --->  _UDWORD
```

で、HEWの typedefine.h で、宣言されている右側の整数型の 型名ですが 多分、H8マイコン環境独自の 物と思われます。で、型名の最初に _ が 付いてますが、これも、Microsoftの見解では 標準では無い。 という事を 明示するために 先頭に _ を 付ける という事のようです。

ちょっと思ったのが 右側の H8マイコン環境の型名は アセンブラ的な発想の名称ですね。 私は 好きです。

それと、全体のプログラムの構成を　変えました。 それは　Arduino環境では　メインの　.inoファイル内に　初期化処理の setup関数と、メインループとなる loop関数が 起動処理から読み出されるようになってます。

で、setup関数と　loop関数内に BME280の処理とパソコン側で データ表示するための　シリアル通信が、メインとして　コーディングされているので、メインから　BME280の機能を サブルーチンとして呼び出す形態に なっていないので、BME280の サブルーチンライブラリ的なイメージに　構成をかえました。

が、今回行った移植作業というか、変更作業です。

BME280の　大雑把な処理内容

処理は　大きく初期化処理と、毎回の測定処理になります。

初期化処理は、各種動作モードを 設定する処理と、他のセンサ素子では　あまり見ない処理として、**補正データ読み込み処理**が　あります。

補正データ読み込み処理は、私の推測ですが多分、工場出荷時に　BME280の 各個体デバイスの特性のバラツキを　自動測定して、真値からのズレを 補正するための系数を 各BME280デバイスの　ROMに書き込んである物と思われます。

① 温度補正データは、dig_T1 ～ dig_T3 の3Wordの データです。(Uが 1個、Sが 2個)

② 湿度補正データは、dig_H1 ～ dig_H6 のByte、Word入り混じったデータが　計 6個あります。 (Uが 2個、Sが 4個)

③ 気圧補正データは、dig_P1 ～ dig_P9 の 9Wordの データです。(Uが 1個、Sが 8個)

因みに、Uは 符号なし整数で、Sは符号付き整数です。

次は、毎回行う**測定処理**です。
2回コマンド設定を行い、8byteの測定データを取り込みます。dacという変数名の byte配列の先頭8byteに 測定データが 入ります。
dac[0]が 上位byte、dac[1]が 中位byte、dac[2]が 下位byteの 3byte(24bit) 整数値を右 4bit シフトして adc_P(気圧生データ)に 20bit分解能の値として代入してます。

同様に dac[3]、dac[4]、dac[5] の値に 同様の処理を行い、adc_T (温度生データ)に 20bit分解能の値として代入してます。

次に dac[6]、dac[7] の値を 連結して Wordデータにして、adc_H (湿度生データ)に 16bit分解能の値として代入してます。

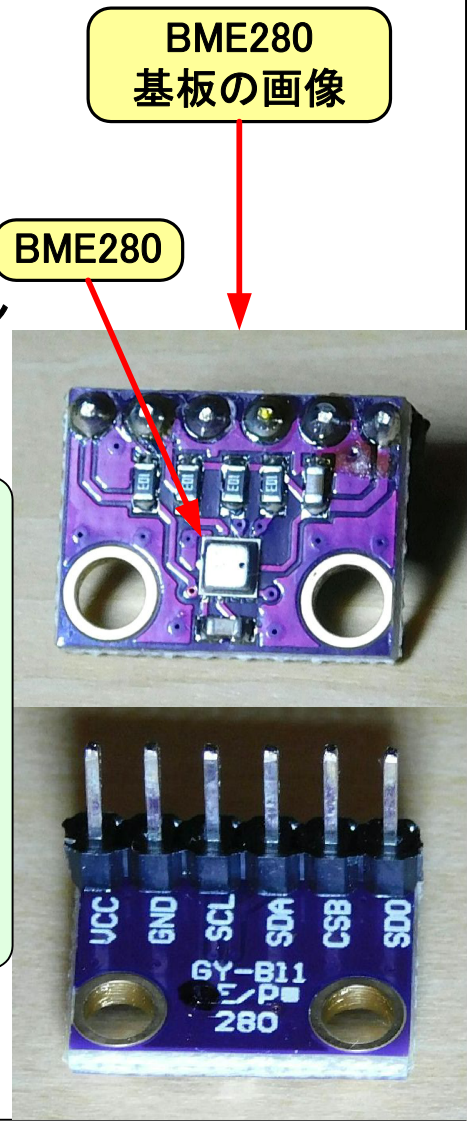
adc_P、adc_T、adc_Hの 3つの 測定生データを用いて、気圧、温度、湿度の 補正計算を行います。 そして pres_cal (気圧補正データ)、temp_cal (温度補正データ)、humi_cal (湿度補正データ) の 3つが 出来ます。
補正計算の関数名は
BME280_compensete_P が 気圧補正計算
BME280_compensete_T が 温度補正計算
BME280_compensete_H が 湿度補正計算
となります。 因みに補正計算の内容は 私には さっぱり分かりません。

最後に 表示上で 適切な値となるように、単位変換の 処理をして、pres(気圧)、temp(温度)、humi(湿度)データに しています。

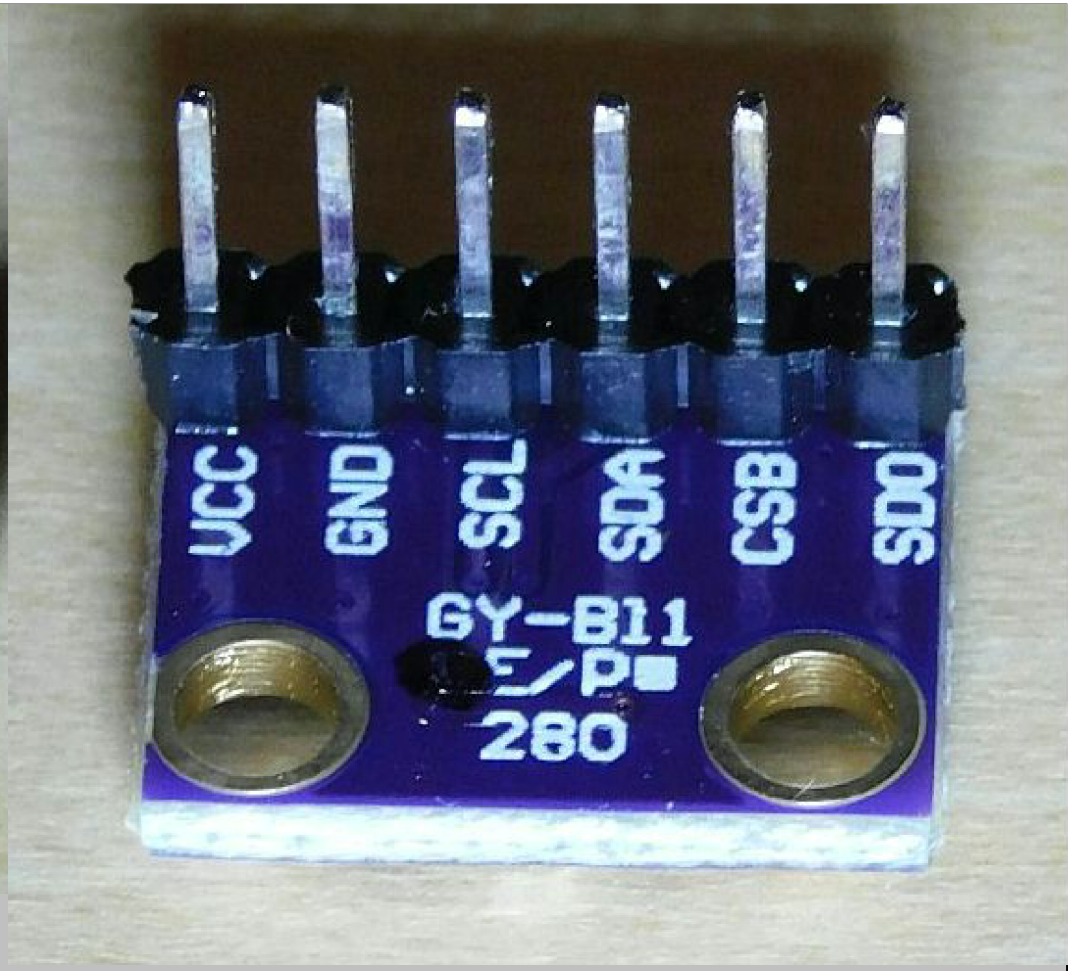
あと、値を メインに返さなければなりませんが、一つの関数値で 3つの値を返せませんので、計算した値を取り出す関数を 3つ用意しました。
下の bme280.h ファイル 関数プロトタイプ宣言の 下3本が 単純に値を取り出す関数です。 値は long になってますが、単位は 温度が 0.1℃単位で、湿度が 0.1%単位、気圧が 0.1Hpa単位に なってます。
文章ばかりで、非常に 分かり難かったと思います。 プログラムソースをダウンロードして、この 静止画 pdfファイルと 見比べると プログラムの内容が 多少見えて来ると 思います。

```
//     BME280 処理関数   関数プロトタイプ宣言 ( bme280.h )
// ------------------------------------------------------
void  init_bme280( void );       // BME280  初期化処理
void  measu_bme280( void );      // BME280 測定実行 ( 1秒以上の周期で行う )

long  get_bme280_temp( void );  // BME280  温度変換データ取り出し
long  get_bme280_humi( void );  // BME280  湿度変換データ取り出し
long  get_bme280_pres( void );  // BME280  気圧変換データ取り出し
```




BME280


VCC
GND
SCL
SDA
CSB
SDO
GY-B11
280