RL78/G22-FPB(048)で遊んでみた(その 2)

RL78/G22-FPB の第 2 弾として、温湿度センサ(HDC1080)を制御するスケッチを RL78/G23-64PFPB から移植してみました。図 1.にシステム構成の写真を示します。

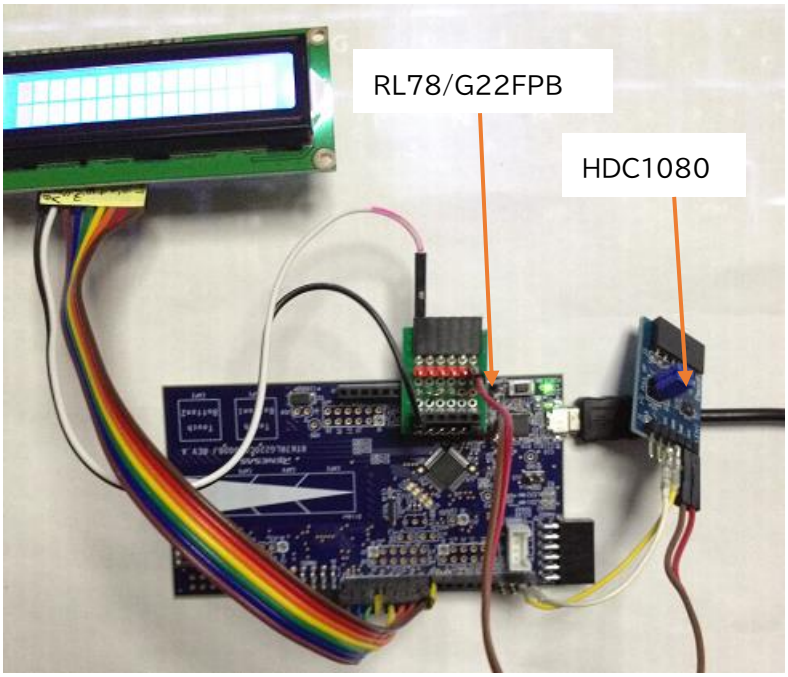

図 1. システム構成

基本的には、LCD とセンサボードをケーブルで接臆するだけですが、電源関係を分岐するための小さなボードも接続しています。

LCD を制御する信号を表 1.に示します。

表 1. LCD 制御信号の配置

Arduino コネクタの pin	HD44780 信号名	備考
GND	GND	
5V	VDD(5V)	
pin 0	RS 信号	
pin 1	R/W 信号	
pin 2	E 信号	
pin 3	–	
pin 4	D4	
Pin 5	D5	
pin 6	D6	
pin7	D7	

Arduino コネクタの 8pin は使っていないので、P30 の問題は影響しません。

測定結果は図 2. に示すように 16 文字×2 行の LCD に表示しています。

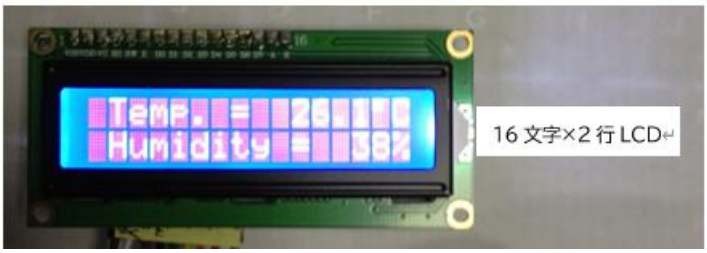

図 2. 測定結果

最初は、Arduino IDE の LiquidCrystal API を使おうとしたのですが、RL78/G22-FPB では LiquidCrystal API が準備されていないために、自前の LCD_LIB 関数を組み込んで対応しました。

対象のファイルを図 3. に示します。LCD_LIB.ino の API 関数プロトタイプ宣言を LCD_LIB.h に独立させ、スケッチでは、このヘッダファイルをインクルードするようにしてあります。

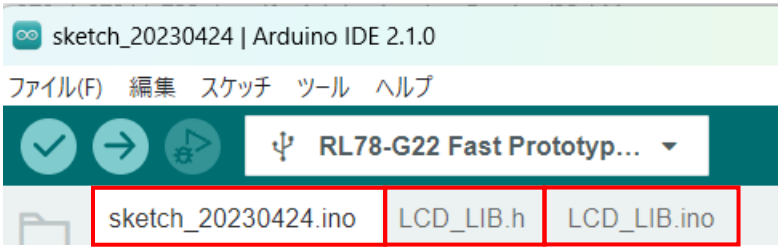

図 3. 対象のファイル

コンパイルした結果を以下に示します。

```
C:\Users\e-bab\AppData\Local\Arduino15\packages\renesas\hardware\rl78g22_fpb\2.0.0\libraries\Wire\src/Wire_IICA
  const unsigned long STATUS_WAIT_TIMEOUT   = 1000;
                      ^
1 warning generated.
最大65536バイトのフラッシュメモリのうち、スケッチが20887バイト（31%）を使っています。
最大4096バイトのRAMのうち、グローバル変数が1792バイト（43%）を使っていて、ローカル変数で2304バイト使うことができます。
```

図 3. コンパイル結果

何かスケッチと関係ないところでワーニングが発生していまが、ここでは無視しておきます。

以下、実際のスケッチの中身です。図 4.が setup 部です。ここでは、スイッチ用のピンの設定、HD1080 センサ制御用の IICA0 の設定、int_LCD 関数で HD44780LCD コントローラを 4 ビット幅に設定、表示情報の初期値(15℃、50%)を表示しています。

```
53  void setup() {
54    // put your setup code here, to run once:
55      pinMode(swPin, INPUT);            // set D18pin to input mode
56
57      Wire.begin();                       // set IICA0 for I2C bus master
58
59      Wire.setClock(400000);              // set fast mode (400kHz clock)
60
61      init_LCD(0,1,2,4,5,6,7);            // initialize LCD display
62
63      disp_line1[14] = 0xDF;              // set degreeC character of LCD
64      print_LCD( (int8_t *)disp_line1, (int8_t *)disp_line2);   // display start data
65
66  }
```

図 4. setup の処理

loop部は、millis()関数を参照して、16 ミリ秒インターバルの基本となるタイミングを作って、そこでスイッチをチェックしています。通常のベアメタル方式のプログラムでは、ハードウェアのタイマを 16 ミリ秒のインターバルタイマとして使用し、インターバル割り込みを使ってこの処理を行うのですが、Arduino のシングルタスクのプログラム(スケッチ)では、経過時間を読み出してソフトウェアで検出することになります。

図 5. にその具体的な処理例を示します。ミリ秒単位の経過時間を読み出し、処理を簡単化するために手抜き処理で、下位の 4 ビットをマスクすることで、16ミリ秒単位に丸めた経過時間を得て、それを前回の値(old_time)と比較して、値が変わっていたら 16ミリ秒経過したと判定しています。

```
76  /*--------------------------------------
77      wait for 16milli seconds interval.
78  --------------------------------------*/
79
80      time_work = ( int )( millis() & 0x07FF0 ); // read milli sec data
81
82      if ( old_time != time_work )        // check 16 milli seconds passed
83      {
84
85  /*--------------------------------------
86      every 16milli seconds timing
87  --------------------------------------*/
88
89          old_time = time_work;           // set current time
90          ++count16ms;                    // count up 16milli seconds counter
```

図 5. 16ms の基本タイミング検出

また、変数 m_time で HDC1080 の制御タイミングを作っています。

16 ミリ秒毎にチェックする m_time の値に応じて以下のような処理を行います。m_time は、通常は 0x00 ですが、1 分経過するか、SW が押されると 0x01 となり、HDC1080 の制御を開始します。

m_time の値に対する処理内容を次に示します。

0x00： 何もしない
0x01： HDC1080 を起動し、16ms 待つ
0x02： さらに、16ms 待つ
0x03： HDC1080からデータ読み出しを起動し、16ms 待つ
0x04： 4 バイトのデータが揃ったら、読み出して、温度と湿度を計算して、結果を LCD に表示する。データが揃っていなかったら、16ms待つ

m_time が 0x04 になったところで、計算から表示変更までの処理を行うので、このときだけはほかの処理を行う時間的な余裕がなくなる可能性がないとは言えません。そこが気になるなら、HDC1080 からの読み出し処理、計算処理、表示処理を分けることが考えられます。例えば、m_time が 0x05 なら温度と湿度の計算、0x06 ならLCDへの表示処理とすることが考えられます。こうすると、32 ミリ秒余分、全体で 100 ミリ秒程度かかりますが、（人間が）気になるようなレベルではないでしょう。

以上