RL78/G23-64PFPB の CS+CC-RL 環境での I2C バスの制御

これまでは、Arduino IDE 環境で遊んできましたが、CS＋CC-RL 環境でのプログラムも作ってみることにします。

Arduino IDE 環境で作ったスケッチを CS+ CC-RL 環境（SC のコードを一部変更）でのマルチタスク（ベアメタル）に置き換えて、作り直してみます。
今回は、Grove コネクタ用のケーブルを入手したので、これを使って RL78/G23-64PFPB の I2C バスでセンサーと LCD 表示器を制御することにします。

（1）　ハードウェアの構成
LCD 表示器としては、秋月電子通商で購入した ACM1602NI-FLW-FBW-M01 を使用
センサーは HDC1080 の Pmod モジュールを使用（LCD ボードの Pmod コネクタに接続）
ケーブルの接続には、図 1 のように必要に応じて両端ロングピンヘッダを使います。

図 1 にシステム構成、図 2 に Pmod コネクタ部、図 3 に Grove コネクタ部 の写真を示します。

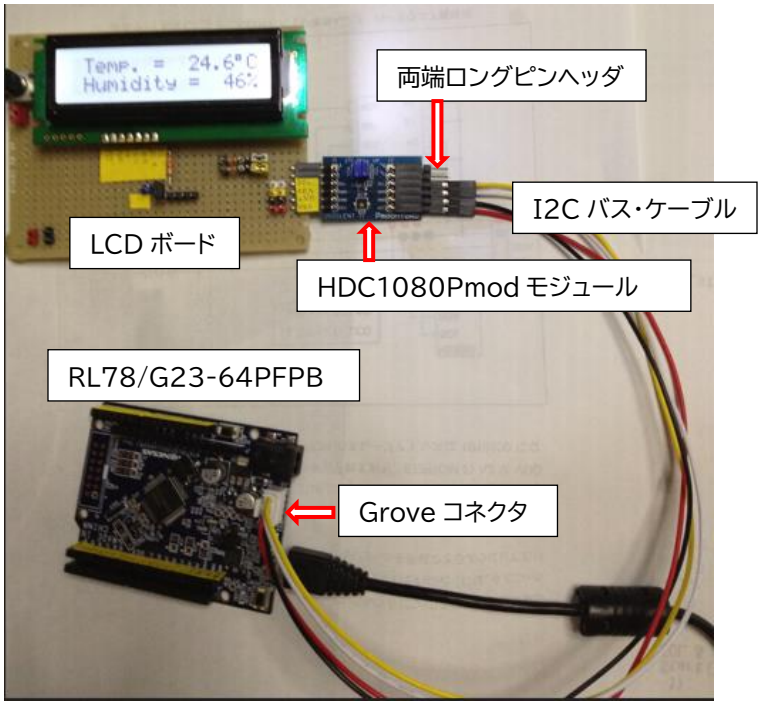

図 1　システム構成

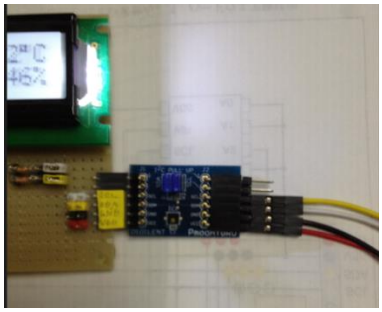

図 2　Pmod コネクタ部

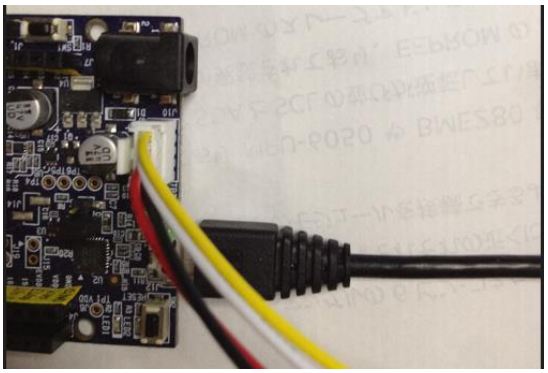

図 3　Grove コネクタ部

（2）　ソフトウェアの構成

タイトルに書いたように、CS+CC-RL 環境でのプロジェクトですが、Arduino IDE 環境のいくつかの API 関数に対応します。今回は、delay 関数、delayMicroseconds 関数、micros 関数、millis 関数のような時間関係の関数について CS+CC-RL 環境で使えるようにしました（ただし millis 関数以外は 32bit 長ではなく、16bit 長に制限しています）。

Arduino IDE 環境では、スケッチ全体の制御は、millis 関数を適宜呼び出すループを使って 16 ミリ秒のインターバルタイマ機能を実現していました。しかし、CS+CC-RL 環境では、TAU の 1 チャネルを使用することで、他の処理に影響されない、インターバルタイマを実現できます。

さらに、センサー（HDC1080 等）の制御については、TM03 をインターバルタイマとして使用して、以下のようにしてシーケンサを実現しています。これで、センサーの制御は、バックグラウンドで行われ、メイン処理では、結果だけを読み出して、それを表示データに変換して LCD に転送するだけになります。

具体的には、起動関数（R_sensor_start()）でセンサーのスタンバイ状態を解除して、TM03 を 40 ミリ秒のインターバルで起動します。その後のインターバル時間として 10 ミリ秒を TM03 に設定しておきます。以降は、INTTM03 をトリガにしたシーケンサで制御を行います。②、③の判別には変数 m_time を使用しています。これにより、細かな処理待ちループが不要となり、プログラムが分かり易くなっていると思います。

① R_sensor_start()でセンサーのスタンバイを解除するため、I2C バスに送信し、シーケンサ用に TM03 を 40 ミリ秒のインターバルで起動し、以降は 10 ミリ秒インターバルとする。（TAU の TDR レジスタの値は起動時に TCR レジスタに転送された後は、変更可能です。）

② 最初の TM03 割り込みで、測定データ読み出し用に I2C バス受信を起動する。

③ 2 番目の TM03 割り込みで、受信処理が完了していたら、受信データから温度、湿度を計算して処理完了はフラグ（sensor_end）をセットして TM03 を停止して処理を完了する。

これにより、R_sensor_start()を引数でセンサーを指定して起動したら、後はフラグ（sensor_end）がセットされるのを待つだけです。その後は得られた結果（humid1 と temp1、または、humid2 と temp2）を表示用データ（ASCII コード）に変換して、LCD に送るだけです。

（3） 使用する内蔵周辺機能
今回、使用した内蔵周辺機能は図 4 に示すようになっています。

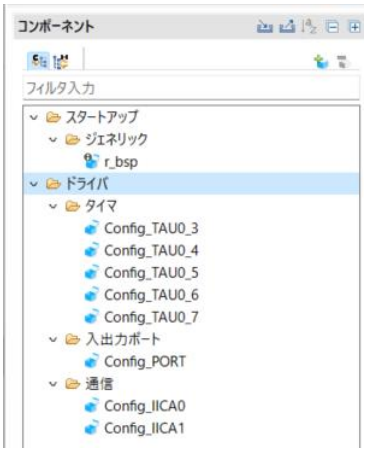

図 4 使用する内蔵周辺機能

タイマーは、一応 TM03～TM07 を準備しています。入出力ポートとしては、オンボードの LED とスイッチ用に準備しています。通信関係では、IICA0 と IICA1 を使用します（今回は Grove コネクタを使うので、IICA1 を用いることになります）。
TM07 は delayMicroseconds 関数（ただし、16bit 長）で使用します。図 5 に SC の初期設定画面を示します。カウントクロックは 32MHzを 32 分周した 1MHz を使用することで、マイクロ秒単位でのカウントを行います。割り込みの優先順位は最高に、インターバル時間はダミーとして 100 マイクロ秒に設定してあります。

図 5 TM07 の設定

図 6 に TM06 の設定を示します。TM06 は、1 ミリ秒のインターバルの割り込みでミリ秒をカウントし、millis 関数と micros 関数で使用します。
図 7 に micros 関数の処理を示します。TM06 ではマイクロ秒をカウントして 1 ミリ秒を生成しています。本来なら、micros 関数は 1 マイクロ秒をカウントする 16bit のカウンタをそのまま使えば、簡単です。それでは、あまり面白くないので、TM06 を millis 関数と共用することにしたのがこのプログラムです。
TM06 のカウント完了（INTTM06 発生）との競合を避けるために、最初に割り込みを禁止しておきます。次に、TM06 のカウント値を読み出して、読み出した値を 1000 から引くことで、マイクロ秒の桁の経過時間を得ます。また、ミリ秒単位の経過時間の下位 16bit を読み出します。（マイクロ秒で 16bit の結果だけなら、ミリ秒の上位 16bit は影響を与えません。）

図 6　TM06 の設定

```
107 uint16_t micros(void)↓
108 {↓
109 ↓
110     uint16_t time1;                     /* 1st read data                */
111     uint16_t time2;                     /* milli second data            */
112 ↓
113 /*  for exclusive control               */↓
114     DI();                               /* prohibit interrupt           */
115     time1  = 1000 - TCR06;              /* get micro second data        */
116     time2  = (uint16_t)(0xFFFF & g_millisecond ); /* get milli second   */
117 ↓
118     if ( ( 1 == TMIF06 )&&( time1 < 2 ) ) /* check if just countup      */
119     {                                   /* when coutup add 1000micro    */
120         time1 += 1000;                  /* adjust micro second          */
121     } ↓
122 ↓
123     EI();                               /* enable interrupt             */
124 /*  end of exclusive control            */↓
125 ↓
126     time2  = (uint16_t)(time2 * 1000);  /* make micro seconds data      */
127     time2  = (uint16_t)( time2 + time1 );  /* add low digit data        */
128 ↓
129     return(time2);                      /* return with elapsed time     */
130 ↓
131 }↓
```

図 7　micros 関数の処理

その後、INTTM06 の割り込み要求とマイクロ秒の桁の値を評価します(118 行目)。ここでは TM06 が INTTM06 発生後 2 マイクロ経過していたかどうかを判断基準としています。

① TM06 が 2 カウントしていない状態(TCR06 が 998 以上)で
② TMIF06 がセットされていた

・この 2 つの条件が満足されたなら、割り込みが禁止された状態で TCR06 を読み出すまでの間に TM06 がそこまでカウントアップされた」と考えられます。この場合には、1 ミリ秒(1000 マイクロ秒)分を加算するようにします。
・TMIF06 がセットされていないなら、単純に、ミリ秒の分と TM06 のカウント数分を加算するだけです。
・TMIF06 がセットされていて、TM06 のカウント値が 2 以上なら、TCR06 レジスタの値を読み出した後から TMIF06 がセットされたと判断して、この場合も単純にミリ秒分と TM06 のカウント数分を加算するだけです。

このような処理を行うことで、TM06 のカウント完了時のカウントの不連続をなくすことができます。
勿論、専用のタイマーを使えば、このような手間はかかりません。ただし、16bit 以上のカウント値が必要な場合には、同じような処理が必要になります。

なお、RL78 の HOCO の精度は 1%なので、あまり気にする必要はないのかもしれませんが。

排他制御が必要なのは、millis 関数の処理も同様です。図 8 に millis 関数の処理を示します。ここで、実行したいのは、ミリ秒をカウントしているグローバル変数 g_millisecond の値を読み出して、戻り値にするだけです。しかし、変数 g_millisecond は 32bit 長の大きさなので、RL78 の 1 命令では読み出すことができません。そこで、読み出しているときには、変数 g_millisecond をカウントアップする INTTM06 の処理を禁止する必要があります。このため、割り込みを禁止して、INTTM06 を受け付けないようにしています。

```
85 uint32_t millis(void)↓
86 {↓
87 ↓
88     uint32_t work;↓
89 ↓
90 /*  for exclusive control                */↓
91     DI();                                 /* prohibit interrupt           */
92     work = g_millisecond;                 /* read elapsed time(milli)     */
93     EI();                                 /* enable interrupt             */
94 /*  end of exclusive control             */↓
95 ↓
96     return(work);                         /* set elapsed time(milli)      */
97 ↓
98 }↓
```

図 8　millis 関数の処理

図 9 に TM05 の設定を示します。TM05 は、1 マイクロ秒のクロックをカウントして、1 ミリ秒のインターバルの割り込みを発生させるために用います。ここでは、インターバルは 100 マイクロ秒に初期設定していますが、delay 関数の頭で、1000 マイクロ秒に設定しています。

図 9　TM05 の設定

図 10 に TM05 を使った delay 関数の処理を示します。ここでは、必要なパラメータを設定し、その後 TM05 を起動して、フラグ（グローバル変数 g_delay_flag）を使って、指定した時間が経過したかを確認しています。指定した時間が経過したら TM05 を停止して、処理を完了しています。ここでは、フラグを参照しているだけなので、排他制御は必要ありません。

```
88  void delay(uint16_t time)↓
89  {↓
90  ↓
91      TDR05 = 1000 - 1;                   /* set micro seconds for milli  */↓
92      g_delaycount = time;                /* set wait time(ms)            */↓
93      g_delay_flag = 0x00;                /* clear time up flag           */↓
94      R_Config_TAU0_5_Start();            /* start TM05                   */↓
95  ↓
96      while ( 0x00 == g_delay_flag )      /* wait for time up flag is set */↓
97      {↓
98          NOP();↓
99      }↓
100 ↓
101     R_Config_TAU0_5_Stop();             /* stop TM05 Operation          */↓
102     g_delay_flag = 0x00;                /* clear time up flag           */↓
103 ↓
104 }↓
```

図 10　delay 関数の処理

図 11 に、対応する TM05 の割り込み処理を示します。グローバル変数 g_delaycount をカウントダウンして 0 になったら、グローバル変数 g_delay_flag を 0x01 に変更するだけです。

```
68  static void __near r_Config_TAU0_5_interrupt(void)↓
69  {↓
70      /* Start user code for r_Config_TAU0_5_interrupt. Do not edit comment generated here */
71  ↓
72      --g_delaycount;                     /* count down wait time         */↓
73      if( 0x0000 == g_delaycount )↓
74      {↓
75          g_delay_flag = 0x01;↓
76      }↓
77  ↓
78      /* End user code. Do not edit comment generated here */↓
79  }↓
```

図 11　TM05 割り込みの処理

ここでは、グローバル変数 g_delaycount は、16bit にしています。65536 ミリ秒以上に対応していないのは、そんなに長い時間 delay 関数から抜けないで他の処理ができない（もちろん割り込み処理は実行可能）のが気になったからです。
長くしたいなら、delay 関数の引数を 32bit に変更し、グローバル変数 g_delaycount も 32bit に変更するだけです。

図 12 に TM04 の設定を示します。TM04 は 20 ミリ秒周期のインターバルタイマとして、スイッチのチェックのタイミングに使用しています。また、1 秒（second_f で通知）や 1 分（minute_f で通知）のインターバルを測定するためにも使用しています。使用する時間間隔が長いので、割り込みの優先順位は最低に設定されています。

図 12　TM04 の設定

スイッチの状態はグローバル変数 sw_work の下位 2bit(変数 sw_data とする)で制御しています。また、押下している時間を測定し、短押し、長押しのような制御も行っています。このために、前回の状態と今回の状態で、図 13 に示すような判断分岐を行っています。400 ミリ秒以下ならば、変数 s_push をカウントアップ、2 秒未満なら変数 m_push をカウントアップ、2 秒以上なら変数 l_push をカウントアップするようになっていて、main 処理ではこれらの変数を確認して、それに応じた処理を行えるようにしています。

このために、スイッチが押されたら(図 13 の 118 ライン)、押下時間をクリアします。押され続けているなら(図 13 の 122 ライン)、押下時間をカウントアップします。放されたら(図 13 の 126 ライン)、押下時間を判断して対応するフラグをカウントアップします。

```
116     switch ( sw_data )↓
117     {↓
118         case 0b10:                  // switch is just pushed↓
119             push_time = 0x0000;     // clear push time↓
120             break;↓
121 ↓
122         case 0b00:                  // keep pushing↓
123             ++push_time;            // count up push time↓
124             break;↓
125 ↓
126         case 0b01:                  // release key↓
127             if ( 20 > push_time )   // less than 400 millis↓
128             {↓
129                 ++s_push;           // set short time push flag↓
130             }↓
131             else if ( 100 > push_time ) // less than 2 second↓
132             {↓
133                 ++m_push;           // set middle time push flag↓
134             }↓
135             else↓
136             {↓
137                 ++l_push;           // set long time push flag ↓
138             }↓
139             break;↓
140     }↓
```

図 13　スイッチの制御

図 14 にはセンサーのシーケンス制御を行うための TM03 の設定を示します。ここで設定しているインターバル時間はダミーです。実際のインターバルは、実際にセンサーを起動するときに設定しています。具体的な動作は、「(2)ソフトウェアの構成」の頭の方に記載されています。

図 14　TM03 の設定

（4） センサーの制御

図 15 にセンサーの動作を起動する R_sensor_start 関数の頭の部分を示します。この関数の引数で、使用するセンサーとして HDC1080 か HS3001 かを指定できるようになっています。今回は手元に有った HDC1080 で評価しています。

```
/***********************************************************************************
* Function Name: R_sensor_start↓
* Description  : This function start sensor↓
* Arguments    : sensor↓
*                0x00 : HDC1080↓
*                0x01 : HS3001↓
* Return Value : status↓
*                0x00(SUCCESS)    : success to trigger sensor↓
*                0x8F(BUS_ERROR)  : I2C bus is busy↓
*                0x80(NO_SLAVE)   : sennsor is not exist↓
***********************************************************************************/
uint8_t R_sensor_start(uint8_t sensor_sel)↓
{↓
↓
    uint8_t status;             // IICAn status↓
↓
    sensor_end = 0x00;          // clear sensor complete flag↓
    senser[0] = 0x00;           // set address data for HDC1080/HS3001↓
↓
/*=================================================================↓
    trigger sensor to start measurement↓
=================================================================*/↓
↓
    if ( 0x00 == sensor_sel )↓
    {↓
/*-----------------------------------------------------------------↓
    trigger HDC1080(start to measure)↓
-----------------------------------------------------------------*/↓
```

図 15 センサーの起動処理

また、図 16 に示すように、IICA0 と IICA1 を変数 sel_IICA で選択できるようにしてあります。

今回は、RL78/G23-64PFPB の Grove コネクタを使うので、IICA1 を使用します。その為、main 関数の最初で、変数 sel_IICA には 1 を設定しています。

```
/*-----------------------------------------------------------------↓
    trigger HDC1080(start to measure)↓
-----------------------------------------------------------------*/↓
        if ( 0 == sel_IICA )    // used IICA is 0↓
        {↓
            status = R_Config_IICA0_Master_Send (↓
                SLADDR_HDC1080*2,   // slave address of HDC1080↓
                (uint8_t *)senser,  // address register value↓
                0x01,               // send data is 1 byte↓
                0xFF);              // timeout is max value↓
        }↓
        else↓
        {↓
            status = R_Config_IICA1_Master_Send (↓
                SLADDR_HDC1080*2,   // slave address of HDC1080↓
                (uint8_t *)senser,  // address register value↓
                0x01,               // send data is 1 byte↓
                0xFF);              // timeout is max value↓
        }↓
↓
            m_time = 0x01;          // set active flag↓
↓
    }↓
```

図 16 IICA の選択処理

センサーの制御完了は、グローバル変数 sensor_end で示され、そのときセンサーから読み出したデータは配列 senser[4]に格納されます。
R_sensor_start 関数がスタートすると、図 16 に示すように、I2C バスを経由してセンサー(ここでは HDC1080)にレジスタアドレス(0x00)を送信します。
また、図 16 に示すように、IICA0 と IICA1 を変数 sel_IICA で選択できるようにしてあります。

図 17 には、図 16 で起動された I2C 送信の通信待ち処理を示します。

```
/*=====================================================================*/
    trigger sensor to start measurement
=======================================================================*/

    if ( MD_OK == status )          // success to send slave address
    {

/*---------------------------------------------------------------------*/
    wait for trigger end
-----------------------------------------------------------------------*/

        if ( 0 == sel_IICA )    // used IICA is 0
        {
            status = R_IICA0_wait_comend(0);// wait for end of command
        }
        else
        {                       // used IICA is 0
            status = R_IICA1_wait_comend(0);// wait for end of command
        }

        if ( SUCCESS == status )        // success to transmit
        {
```

図 17 I2C 通信完了待ち処理

今回は、RL78/G23-64PFPB の Grove コネクタを使うので、IICA1 を使用します。その為、main 関数の最初で、変数 sel_IICA には 1 を設定しています。図 18 に main 処理の先頭で実行する初期化処理部分を示します。

```
void R_MAIN_UserInit(void)
{

    sel_IICA = 1;               // select IICA1 channel

    R_Config_TAU0_4_Start();    // start TM04 as switch check timer
    R_Config_TAU0_6_Start();    // start TM06 as mills seconds timer

    EI();

    do{
        old_time = ( int )( millis() & 0x0FFFF );
    }while ( old_time < 200 );  // mait for 200ms to power up time

    init_LCD();                 // initialize LCD to 2 lines mode

    disp_line1[14] = 0xDF;      // set degreeC character of LCD

    print_LCD(
        (uint8_t *__near)disp_line1,    // display 1st line
        (uint8_t *__near)disp_line2);   // display 2nd line

}
```

図 18 main 処理の先頭で実行する処理

ここでは、20m秒インターバルのTM04と経過時間計測用の1ミリ秒インターバルのTM06を起動しています。その後、システム(主にLCD表示器)の安定化待ちとして起動から200ミリ秒経過するのを待ちます。
その後、LCD表示器関係の初期化を行い、LCD表示の初期化を行い、main 関数に戻ります。

（5） main での処理
図 19 に main 関数の先頭部分を示します。main 関数では、最初に R_sensor_start 関数を呼び出して、センサーから温度と湿度を読み出します。この段階では、シーケンサが起動してセンサーの起動をトリガしただけです。

```
95  void main(void)
96  {
97
98      R_MAIN_UserInit();                  // initialize
99  /*-------------------------------------------------------------------------
100     get new  sensor data
101 -------------------------------------------------------------------------*/
102
103     result = R_sensor_start(sensor_sel);// trigger sensor
104
105     NOP();
106     
107     if ( SUCCESS == result )            // sensor is exist
108     {
109         while ( 0 == sensor_end )       // wait for sensor data
110         {
111             NOP();
112         }
113         sensor_end = 0x00;              // clear sensor ready flag
114
115     }
```

図 19　main 関数の処理

107 行目でトリガに成功したか(センサーが存在したかどうか)を判断し、トリガに成功したら、109 行目の while 文でアクセス完了を待ちます。
その後、図 20 に示す main の無限ループ処理に入ります。ループの最初では、センサーから読み出したデータを表示データに変換して LCD に表示します。

```
121     while (1)
122     {
123         
124         R_conv_LCD(sensor_sel);         // display data
125         
```

図 20　main での無限ループ処理

その後、main 処理では、図 21 に示すように計測間隔である 1 分経過するのを待ちます。この状態で、1 分や 1 秒経過を監視する TM04 や経過時間計測用の TM06 が動作しています。TM04 では、時間経過の監視以外にスイッチの状態のチェックを行っています。
Arduino のスケッチでは、この main ループでいろんな処理を実行しないといけなかったのですが、このプログラムではシンプルな構成になっています。これは、割り込みを使うことで、バックグラウンド処理が行われているためです。特に影響が大きいのは、センサー制御をシーケンサで実現していることだと思います。
逆に、バックグラウンドでどのような処理が行われているかを理解していないと、どのように処理を実現しているかは理解できないかもしれません。とは言え、バックグラウンドで実行しているのは独立性の高い処理なので、main 処理では、あまり細かいことを気にする必要はないかと思います。

```
126 /*-------------------------------------↓
127    wait for next sense timing↓
128 -------------------------------------*/↓
129 ↓
130 ↓
131         while ( 0 == minute_f )          // wait for 1 minute↓
132         {↓
133 ↓
134             if ( 0 != l_push )           // long push of user switch↓
135             {↓
136                 sensor_sel ^=0x01;       // change sensor↓
137             }↓
138             l_push = 0;↓
139 ↓
140             if ( 0 != s_push )           // short push↓
141             {↓
142                 s_push = 0;              // clear short push flag↓
143                 break;↓
144             }↓
145 ↓
146             NOP();↓
147 ↓
148         }↓
149         ↓
150         minute_f = 0x00;                 // clear 1 minute flag↓
```

図 21　main の無限ループ処理

図 21 では、131 行目の while 文で 1 分経過するのを待つ中で、スイッチが押されたかをチェックし、「長押し」(2 秒以上押し続けて放す)ならセンサーを切り替え、「短押し」(400 ミリ秒以下の押下)では、while のループから抜けて次の計測を開始するような制御を行っています。普通は、センサーの切り替えを行うことはないので、「短押し」しか使いません。スイッチを押す際には、できるだけ短く押してください。

図 22 に示すのが while(1)ループの最後の処理で、センサーをトリガして計測データを読み出して、ループの先頭に戻るようになっています。

```
152 /*-----------------------------------------------------------------↓
153     get new  sensor data↓
154 -----------------------------------------------------------------*/
155 ↓
156         result = R_sensor_start(sensor_sel);// trigger sensor↓
157 ↓
158         if ( 0x00 == result )               // sensor is exist↓
159         {↓
160             while ( 0 == sensor_end )       // wait for sensor data↓
161             {↓
162                 NOP();↓
163             }↓
164             sensor_end = 0x00;              // clear sensor ready flag↓
165 ↓
166         }↓
167 ↓
168 ↓
169     }   /* end of while(1) loop */↓
```

図 22　while(1)ループの最後の処理

（6） その他の処理
LCD 制御関係の処理は LCD_LIB.c の中に纏められています。このライブラリはこれまでも使用してきた物を SC の生成するコード（を修正した I2C ライブラリ）に合わせただけなので、説明は省きます。

（7） I2C のライブラリ
ここでは、IICA 関係は、図 23 に示すように初期化処理だけをコード生成し、通常のライブラリ API は、SC が生成するライブラリ API に手を加えてたような API を作成しています。

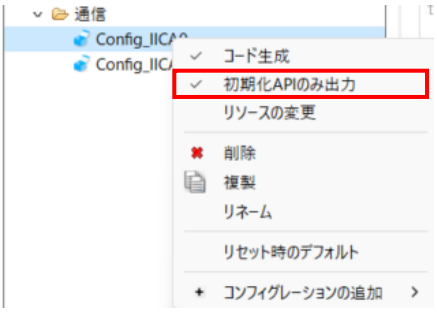

図 23 IICA のコード生成

基本的に SC と同じ名前を使っていますが、中身は変更されています。また、一部名前が変わっている API 関数もあります。当然、新たに追加した API もあります。
図 24 にサポートしている API 関数を示します。93 行目と 94 行目の関数は SC と同じ名前の関数です。引数もそのままにしていますが、中の処理は修正されています。95 行目～99 行目の関数は新たに追加されたものです。

```
90 void R_Config_IICA1_Create(void);↓
91 void R_Config_IICA1_Create_UserInit(void);↓
92 /* Start user code for function. Do not edit comment generated here */↓
93 MD_STATUS R_Config_IICA1_Master_Send(uint8_t adr, uint8_t * const tx_buf, uint16_t tx_num, uint8_t wait);↓
94 MD_STATUS R_Config_IICA1_Master_Receive(uint8_t adr, uint8_t * const rx_buf, uint16_t rx_num, uint8_t wait);
95 uint8_t R_IICA1_StopCondition(void);     /* if possible then issue StopCondition     */↓
96 uint8_t R_IICA1_bus_check(void);         /* check if Bus is enable get Bus           */↓
97 uint8_t R_IICA1_poll(void);             /* check communication status               */↓
98 uint8_t R_IICA1_wait_comend(uint8_t stop_c); /* wait for communication end           */↓
99 uint8_t R_IICA1_check_comstate(void);    /* check communication status               */↓
```

図 24 サポートしている API 関数

IICA 関係のコード生成を初期化処理だけに限定することで、新たにコード生成しても影響を受けないようにはしていますが、確認はしていません。
図 25 に示すような多重割り込み対応の処理を追加していますが、ここでも新たなコード生成に対応するために、図 26 に示すように、”r_cg_userdefine.h”で定義した「USER_MODE」を使って” #ifdef USER_MODE”や” #ifndef USER_MODE”で制御しています。

```
/***********************************************************************************
Pragma directive↓
***********************************************************************************
/* Start user code */↓
#ifndef USER_MODE↓
#pragma interrupt r_Config_TAU0_3_interrupt(vect=INTTM03, bank=RB1)↓
#endif↓
/* End user code */↓
/* Start user code for pragma. Do not edit comment generated here */↓
/* Start user code */↓
#ifdef USER_MODE↓
#pragma interrupt r_Config_TAU0_3_interrupt(vect=INTTM03, bank=RB1, enable=true)
#endif↓
/* End user code */↓
/* End user code. Do not edit comment generated here */↓
```

図 25 多重割り込み対応

```
42 /***********************************************************************
43 Macro definitions↓
44 ***********************************************************************
45 /* Start user code for macro define. Do not edit comment generated here */↓
46 #define LED1             ( P5_bit.no3 )   // USER LED1 for loop↓
47 #define LED2             ( P5_bit.no2 )   // USER LED2 for error↓
48 #define USER_SW          ( P13_bit.no7 ) // on board USER switch↓
49 #define USER_MODE↓
50 /* End user code. Do not edit comment generated here */↓
```

図 26　マクロ定義

◎　SC の生成したライブラリには、以下に示すような基本的な問題点があります。(これ以外にもありますが、省きます。)

・R_Config_IICA0_StopCondition()

図 27 に SC の生成したストップ・コンディション発行関数を示します。単にトリガしているだけで、実際にストップ・コンディションが発行されたかを確認していません。これでは、シリアル EEPROM のように、ストップ・コンディションを検出して転送されたデータをメモリ・セルに書き込みを行うような場合に、正常な書き込みができない可能性があります。

```
/***********************************************************************
* Function Name: R_Config_IICA0_StopCondition↓
* Description  : This function stops the IICA0 condition.
* Arguments    : None↓
* Return Value : None↓
***********************************************************************
void R_Config_IICA0_StopCondition(void)↓
{↓
    SPT0 = 1U;    /* set stop condition flag */↓
}↓
```

図 27　SC の生成したストップ・コンディション発行関数

また、図 28 図 28 にコード生成された R_Config_IICA0_Master_Send 関数の先頭部分を示しますが、I2C バスの状態も何も確認しないで、スタート・コンディションの発行をトリガしています。これも問題がある処理です。この API の戻り値は、全くと言ってもいいほど使い物になりません。そのため、前回の通信が完了していないのに、次の通信を初めてしまうような例が結構ありました。

```
MD_STATUS R_Config_IICA0_Master_Send(uint8_t adr, ui
{↓
    MD_STATUS status = MD_OK;↓
↓
    STT0 = 1U;    /* send IICA0 start condition */↓
    IICAMK0 = 0U;    /* enable INTIICA0 interrupt */
```

図 28　コード生成された R_Config_IICA0_Master_Send 関数の先頭部分

それに対して、この I2C 用の API では図 29 に示すような I2C バスの状態をチェックする R_IICA0_bus_check()関数を呼び出すことで、「前回の通信が完了しているか」や「IICA0 が I2C バスを確保しているか」を確認することができるようになっています。新たに、通信を行えない場合には、エラーで戻るようにしています。

```
uint8_t R_IICA0_bus_check(void)↓
{↓
    uint8_t status;↓
↓
    /* Check I2C bus is used or in waiting state    */↓
↓
    status = BUS_ERROR;                     /* set dummy error flag          */
    if ( 0x00 == ( g_status & 0x10) )       /* check if operating or not     */
    {↓
↓
/*-----------------------------------------------------↓
    IICA0 is not COMMUNICATING and check I2C bus status↓
-----------------------------------------------------*/↓
↓
        if ( ( 1 == MSTS0 ) || ( 0 == IICBSY0 ) )  /* check if I2C is busy */
        {↓
↓
/*-----------------------------------------------------↓
    IICA0 and I2C bus are usable then↓
    issue StartCondition to I2C bus↓
-----------------------------------------------------*/↓
↓
            status = R_IICA0_StartCondition();  /* issue start condition    */
↓
```

図 29　組み込んでいる I2C バスの状態を確認する関数

また、これまでも何回か初心者がつまずいた、通信完了をチェックする(通信完了を待つ)関数も R_IICA0_wait_comend としてサポートしています。図 30 に追加した API 関数の一覧を示します。いずれの関数も結果を確認できるような戻り値をサポートしています。

```
uint8_t R_IICA0_StopCondition(void);     /* if possible then issue StopCondition     */
uint8_t R_IICA0_bus_check(void);         /* check if Bus is enable get Bus           */
uint8_t R_IICA0_poll(void);              /* check communication status               */
uint8_t R_IICA0_wait_comend(uint8_t stop_c); /* wait for communication end           */
uint8_t R_IICA0_check_comstate(void);    /* check communication status               */
```

図 30　追加した API 関数

以上