初心者のための RL78 入門コース（第３回：ポート出力例２とポート入力）

第 3 回の今回は，前回作成したプログラムを RL78/G13 のハードウェアを用いて見直しをおこないます。

今回の内容

次回（第 4 回）は，以下の内容を予定しています。これでポートの基本的な使い方はおしまいになる予定です。

ここまでで使用したプロジェクトは以下のフォルダに格納されています。

```
ポート出力フォルダの構成↓
第1回分↓
    +- RL78_G13_PORT共通部  --- プロジェクトを作成しただけ↓
注：ビルドはしていませんが，クイック・ビルドの結果が残っています。↓
↓
第2回分↓
    +- RL78_G13_PORT1       --- ポートでのLED制御のみ↓
    +- RL78_G13_PORT1_E1    --- ポートでのLED制御のみ（E1用）↓
    +- RL78_G13_PORT1_2     --- ソフトタイマでのLEDチカチカ↓
    +- RL78_G13_PORT1_2_E1  --- ソフトタイマでのLEDチカチカ（E1用）↓
↓
第3回分↓
    +- RL78_G13_PORT1_3     --- インターバル・タイマでのLEDチカチカ↓
    +- RL78_G13_PORT1_3_E1  --- インターバル・タイマでのLEDチカチカ（E1用）↓
    +- RL78_G13_PORT1_4     --- タイマの方形波出力によるLEDチカチカ↓
    +- RL78_G13_PORT1_4_E1  --- タイマの方形波出力によるLEDチカチカ（E1用）↓
↓
↓
ポート入力フォルダの構成↓
第3回分↓
    +- RL78_G13_PORT2       --- if文でポート入力を判定しLEDを制御↓
    +- RL78_G13_PORT2_2     --- ＋外部割り込みでのスタンバイ解除↓
    +- RL78_G13_PORT2_2_E1  --- ＋外部割り込みでのスタンバイ解除（E1用）↓
    +- RL78_G13_PORT2_2B    --- ＋外部割り込みでのスタンバイ解除（解除のみ）↓
    +- RL78_G13_PORT2_2B_E1 --- ＋外部割り込みでのスタンバイ解除（解除のみ）（E1用）
    +- RL78_G13_PORT2B      --- データ転送によるポート入力でのLED制御↓
    +- RL78_G13_PORT2B_E1   --- データ転送によるポート入力でのLED制御（E1用）↓
```

８．コード生成を利用した実際のプログラム作成（その２）

前回作成した，下のプログラムで，LED チカチカは実現できましたが，単にポートの設定を組み合わせただけです。これで完成ではありません。更にシェイプ・アップを行います。

実行する前に，「RL78_G13_PORT1_2」フォルダをコピーして「RL78_G13_PORT1_3」の名前に変更し，これを使用します。

```
59 void main(void)↓
60 {↓
61     R_MAIN_UserInit();↓
62     /* Start user code. Do not edit comment generated here */↓
63     while (1U)↓
64     {↓
65         P3_bit.no1 = 0;          /* P3.1を0（ロウ出力）にする    */↓
66         wait_200();↓
67         P3_bit.no1 = 1;          /* P3.1を1（ハイ出力）にする    */↓
68         wait_200();↓
69     }↓
70     /* End user code. Do not edit comment generated here */↓
71 }↓
```

8.1 ポート出力データの演算

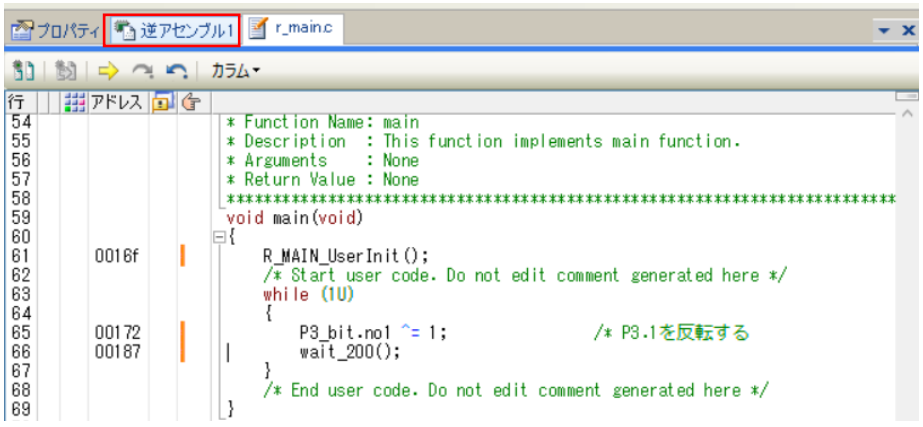
ポートに対して出力したデータ（ビット・データ）に演算を行います。ここで使うのは排他的論理和（XOR）演算です。この新しいプログラムをビルドしてシミュレータにダウンロードします。

下記のプログラムの赤く囲んだ部分がビットに対する XOR 演算部です。

```
59                  void main(void)
60                  {
61      0016f           R_MAIN_UserInit();
62                      /* Start user code. Do not edit comment generated here */
63                      while (1U)
64                      {
65      00172               P3_bit.no1 ^= 1;                /* P3.1を反転する */
66      00187               wait_200();
67                      }
68                      /* End user code. Do not edit comment generated here */
69                  }
```

以下は参考です。

ここで，表示ウィンドウが「r_main.c」となっているのを逆アセンブル表示に切り替えます（上側の「逆アセンブル 1」をクリックします）。



逆アセンブル表示で確認すると，13 個の命令に展開されています。

```
65:         P3_bit.no1 ^= 1;                    /* P3.1を反転する
00172  8d03        MOV     A,P3
00174  73          MOV     B,A
00175  3169        SHL     A,6H
00177  317a        SHR     A,7H
00179  318e        SHRW    AX,8H
0017b  08          XCH     A,X
0017c  7c01        XOR     A,#1H
0017e  08          XCH     A,X
0017f  60          MOV     A,X
00180  718c        MOV1    CY,A.0H
00182  63          MOV     A,B
00183  7199        MOV1    A.1H,CY
00185  9d03        MOV     P3,A
```

このプログラムでも，一応正常に動作はしますが，あまりにコード効率が悪すぎます。これは，最適化をデバッグ優先に設定したためです。

CC-RL の最適化を「既定の最適化を行う」にしておくと，以下のように 1 命令で済みます。

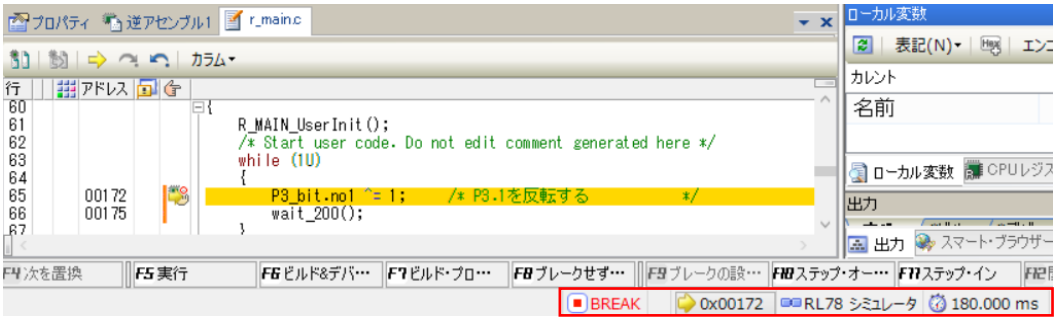
```
65:         P3_bit.no1 ^= 1;      /* P3.1を反転する                */
00172  7a0302      XOR             P3,#2H
```

このように 1 命令（3 バイト）で実現できるのは，ポート 3 は 0xFFF03 番地に配置されていて，そこはショートダイレクト（saddr）領域と呼ばれる領域だからです。saddr 領域は 0xFFE20～0xFFF1F に配置されていて，基本的には RAM ですが，一部の内蔵周辺機能の制御レジスタ（SFR）も入っています。saddr 領域に配置されていると，加減算以外に論理演算も可能です。つまり，RL78 はポートを制御する機能が強化されていることが分かります。

このように，最適化で大きくコード効率が異なります。

ちょっと（？）脇道にそれてしまいましたが，ポートに対する演算を行うことで，LED の ON 時間と OFF 時間を同じにできます。

このようにした理由は，もう一つあります。ここまでは，ソフトウェアでの遅延を使ってきました。これは，同じプログラムでも処理系や最適化により遅延時間が異なります。例えば，「既定の最適化を行う」の状態で実行させると，1 回のループは以下のように 180ms になってしまいます。



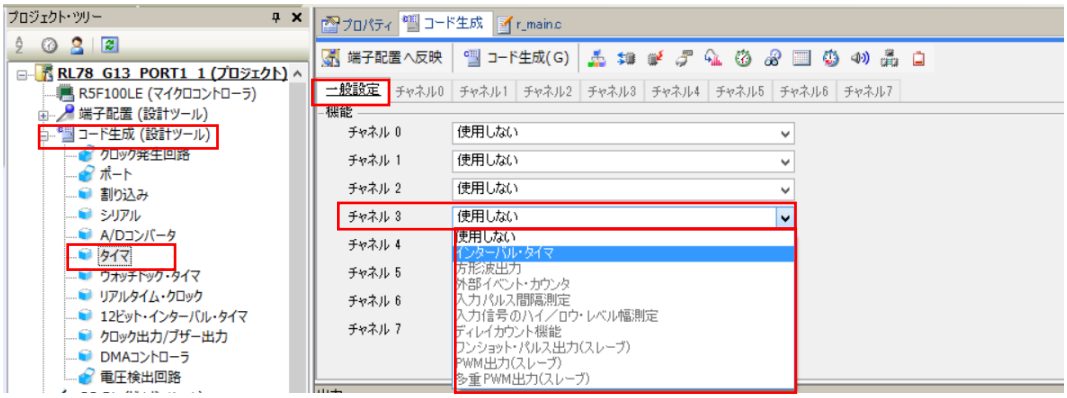
これでは，デバッグ後に最適化レベルを変更するのにはちょっと抵抗があります。そこで，ハードウェアにより時間待ちを行うことにします。ハードウェアで時間測定を行うと，処理系や最適化を変更しても影響は受けなくなります。

LED のドライブで使用している P31 にはタイマ 03（TM03）の入出力端子が割り当てられているので，後のことを考えて，TM03 をインターバル・タイマとして使用することにします。

インターバル・タイマは指定された周期で割り込みを発生させる最も基本的な機能です。インターバル・タイマを使用するには，割り込みを使用するのが必須です。この段階では割り込みの詳しい話は省きます。単に，割り込み処理プログラムで実行するとだけ覚えておいてください。それでは，コード生成の機能を使ったインターバル・タイマの使い方を示します。

8.2 インターバル・タイマの使い方

コード生成で「タイマ」を選択すると，右にタイマの「一般設定」を表示します。機能としては全てのチャネルが「使用しない」となっています。ここで，「チャネル 3」の「使用しない」の右の  をクリックして機能の一覧を表示します。そこで「インターバル・タイマ」を選択します。これで，TM03 はインターバル・タイマで使用されます。



チャネル 3 を「インターバル・タイマ」に設定したら，「チャネル 3」タグをクリックしてチャネル 3 の設定画面を開きます。

ディフォルトではインターバル時間は 100μs になっていますので，これを 200ms に変更します。これ以外の設定はそのままにしておきます。

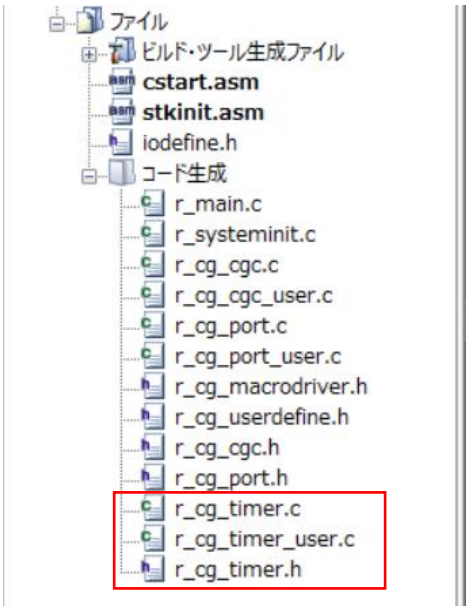
ここまでの設定が完了したら，「コード生成（G)」をクリックしてコード生成を行います。

コード生成を行うと，プロジェクト・ツリーの「ファイル」「コード生成」の下に生成されたファイルが並びます。

下に「r_cg_timer.c」「r_cg_timer_user.c」と「r_cg_timer.h」の 3 つのファイルが生成されています。

「r_cg_timer.c」は，タイマの初期設定を行う R_TAU0_Creat 関数，TM03 を起動するための R_TAU0_Channel3_Start 関数及び停止させるための R_TAU0_Channel3_Stop を含んでいます。

「r_cg_timer_user.c 」には，TM03 の割り込み処理用の r_tau0_channel3_interrupt 関数があります。



r_tau0_channel3_interrupt 関数は，以下のように入れ物だけが生成され，実際の処理はユーザが記述するようになっています。

```
51 /***********************************************************************************
52 * Function Name: r_tau0_channel3_interrupt↓
53 * Description  : This function is INTTM03 interrupt service routine.↓
54 * Arguments    : None↓
55 * Return Value : None↓
56 ***********************************************************************************/
57 static void __near r_tau0_channel3_interrupt(void)↓
58 {↓
59     /* Start user code. Do not edit comment generated here */↓
60     /* End user code. Do not edit comment generated here */↓
61 }↓
```

ここに，割り込みで実行させたいプログラムを記述する

それでは，プログラムを記述していきましょう。

最初に，割り込み処理部分を記述します。インターバル・タイマの割り込み処理では，P31 の出力を反転するだけです。下に示すように，先ほど main 関数に記述した P31 出力を反転させる部分を割り込み処理に記述します。

```
static void __near r_tau0_channel3_interrupt(void)
{
    /* Start user code. Do not edit comment generated here */
    P3_bit.no1 ^= 1;        /* P3.1を反転する                */
    /* End user code. Do not edit comment generated here */
}
```

割り込み処理は，これだけです。

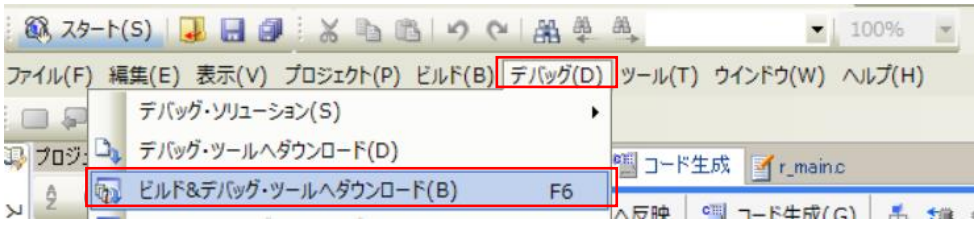
次は，r_main.c ファイルの変更です。ここでは，大きく２つの部分を変更します。まずは，R_MAIN_UserInit 関数に以下のように，TM03 を起動する処理を記述します。

```
void R_MAIN_UserInit(void)
{
    /* Start user code. Do not edit comment generated here */
    R_TAU0_Channel3_Start();        /* タイマ03を起動          */
    EI();
    /* End user code. Do not edit comment generated here */
}
```

次に main 関数を変更します。実際には，main 関数では単にインターバル・タイマ割り込みを待つだけです。下に示すように，main 関数は，while ループ中に NOP();を書くだけです。

```
void main(void)
{
    R_MAIN_UserInit();
    /* Start user code. Do not edit comment generated here */
    while (1U)
    {
        NOP();              /* TM03割り込み待ち            */
    }
    /* End user code. Do not edit comment generated here */
}
```

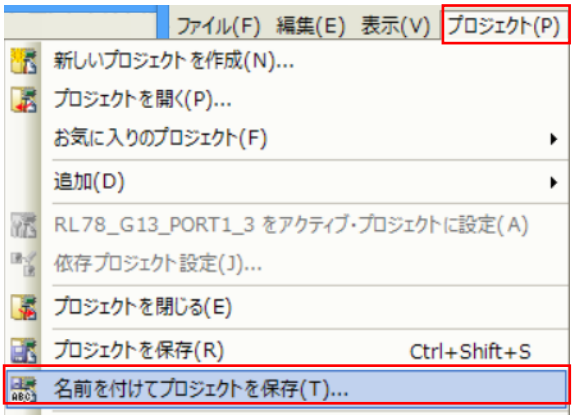
ファイルを変更したら，「デバッグ（D)」－「ビルド&デバッグ・ツールへダウンロード(B)」とクリックして，ビルドとビルド結果をシミュレータにダウンロードします。



これで，ブレーク・ポイントなし（ をクリック）で実行させてみます。

LED は点滅しますが，リアルタイムでの実行はできないので，数秒周期で点滅するだけです。時間を含めて確認するには，やはり E1 で実機を動作させる必要があります。（E1 で実行するプロジェクトは「RL78_G13_PORT1_3_E1」フォルダに入れてあります。）

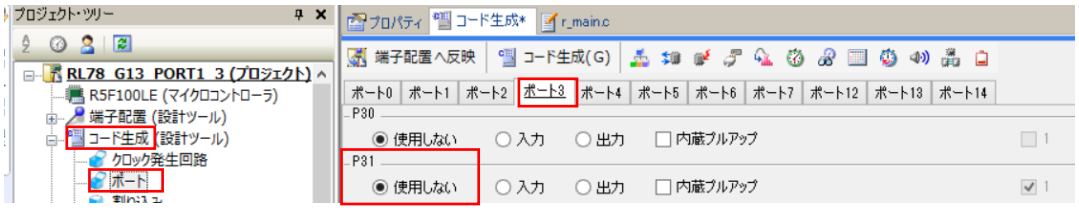
このプロジェクトは，「RL78_G13_PORT1_3」の名前で保存して，終了します。



8.3 方形波出力の使い方

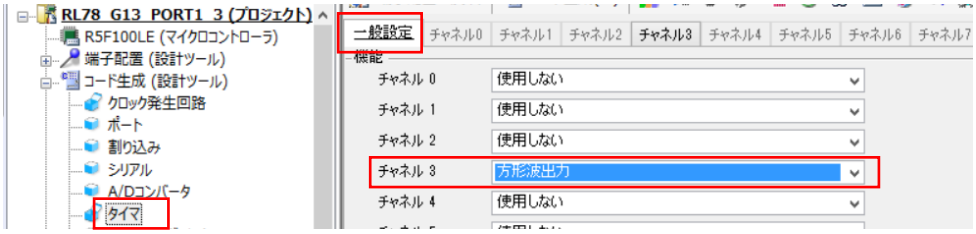
これまでは，インターバル・タイマを使って割り込みで LED を点滅させました。ここでは，更に LED の点滅までハードウェアで処理させてみます。

プロジェクトを開いたら，コード生成の「ポート」－「ポート 3」を選択し，P31 を「使用しない」にします。

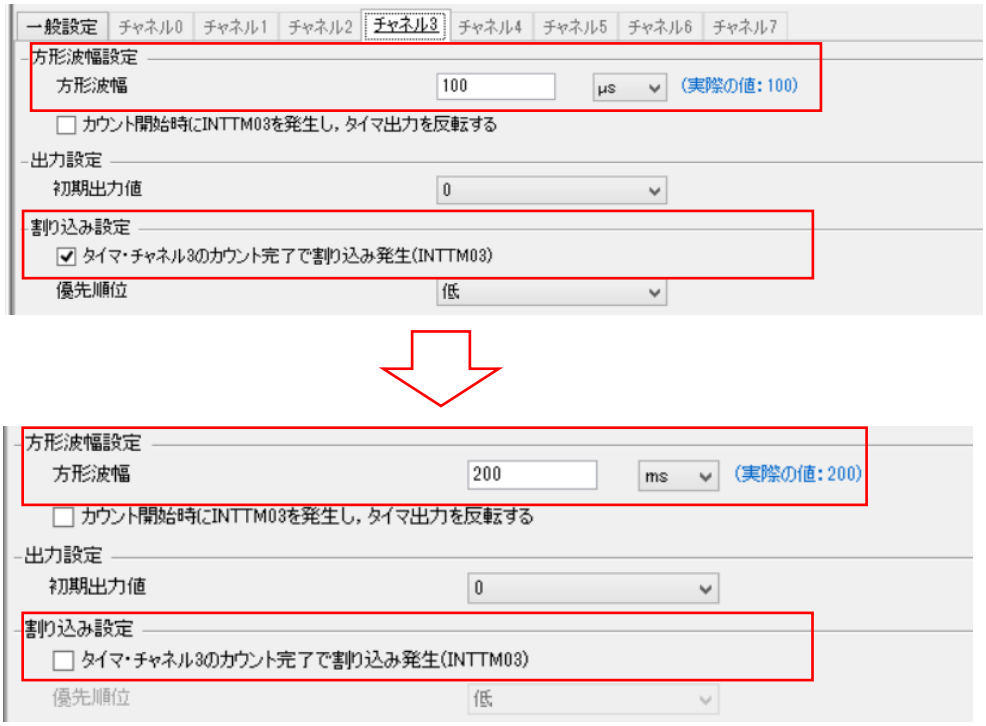


これは，P31 端子を TM03 の出力（TO03）として使用するために，ポートとしての使用を止めるためです。

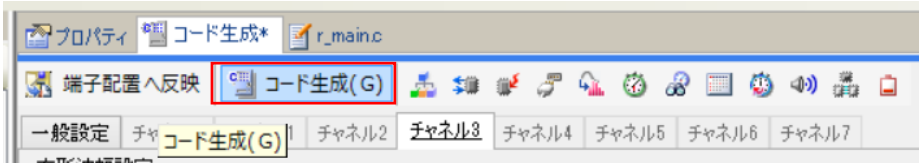
次に，タイマを選択して「一般設定」で「チャネル 3」を「方形波出力」に変更します。



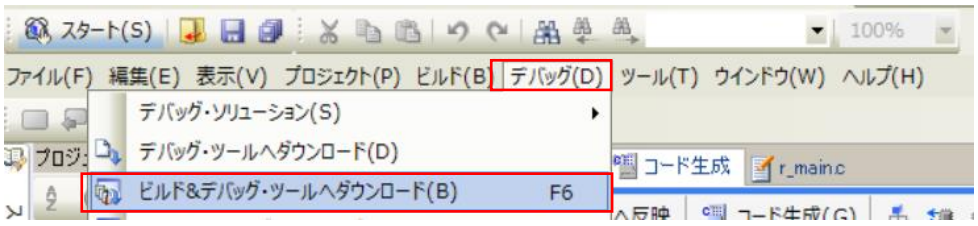
チャネル 3 タグを開くと，下のように 100μs となっているので，これを 200ms に変更します。また，割り込みは使用しないので，チェックを外します。



これで設定変更が完了したので，「コード生成（G)」をクリックして，コード生成を行います。



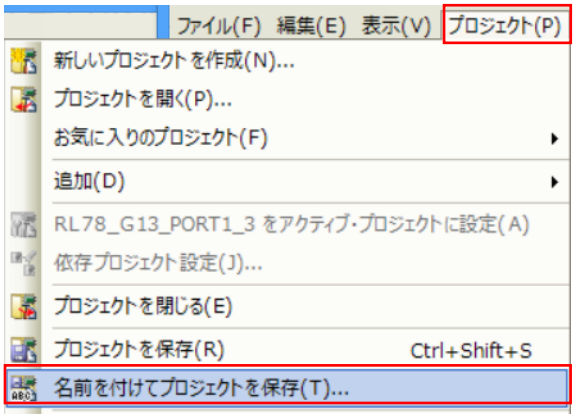
「デバッグ（D)」ー「ビルド&デバッグ・ツールへダウンロード（B)」とクリックして，ビルドとシミュレータにダウンロードします。,

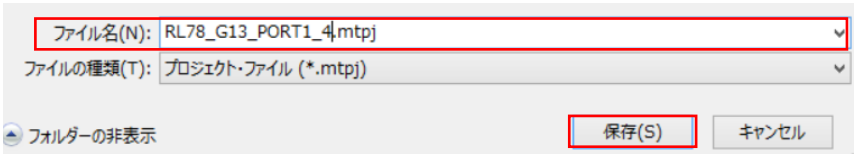


これで，ブレーク・ポイントなし（ をクリック）で実行させてみます。割り込み処理よりは若干速く LED は点滅するようになりました。この方法では，初期設定と起動だけでプログラムのオーバヘッドは一切ありません。これは，CPU は別の処理にかかりっきりにできるということです。

シミュレータ画面を表示して  をクリックして，実行を停止します。その後，右端の  をクリックして，シミュレータを停止し，CS+に戻ります。

このプロジェクトは「RL78_G13_PORT1_4」の名前で保存して，終了します。



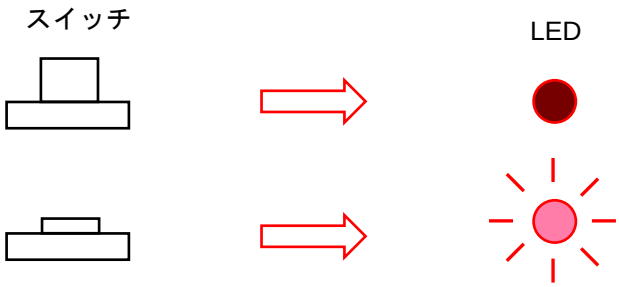


９．コード生成を利用したプログラム作成（ポート入力）

次は，入力ポートによる入力プログラムです。スイッチは，P50/INTP1 端子に接続してあります。通常は P50/INTP1 端子はハイ（1）で，スイッチを押すとロウ（0）になります。このスイッチを使ったポート入力の例を示します。

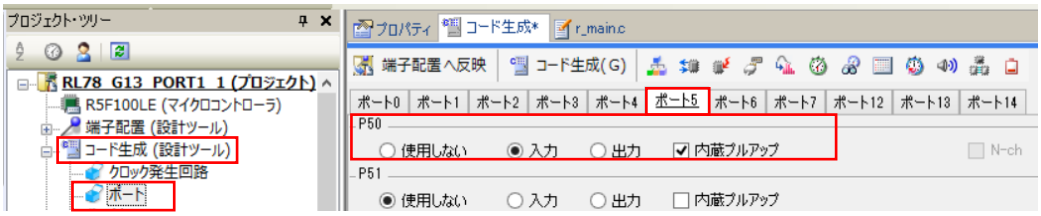
9.1 ポート入力での LED 制御

最初に作成するプログラムは，スイッチが押されたら LED を点灯し，スイッチが押されていなければ，LED は消灯するという単純な制御を行うものです。このために，ポートの入力と出力の２つの機能を使用します。

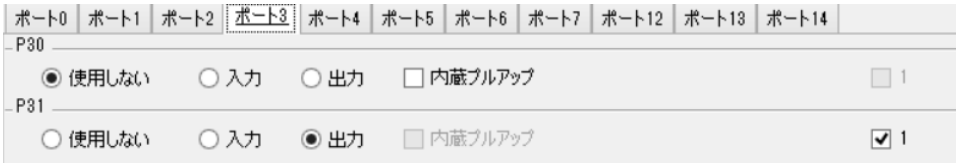


LED チカチカで作った最初のプログラム（RL78_G13_PORT1 ディレクトリ）を基にしてプロジェクトを作成するので，「ポート入力」ディレクトリに RL78_G13_PORT1 ディレクトリをまるごとコピーして RL78_G13_PORT2 と名前を付けておきます。

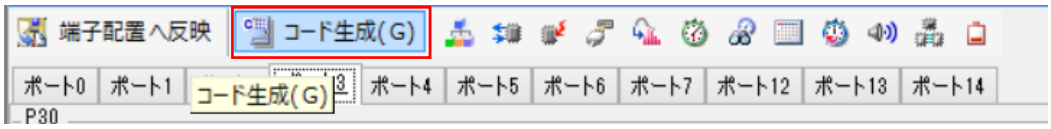
RL78_G13_PORT1_1.mtpj ファイルをダブルクリックしてプロジェクトを開きます。コード生成でポート 5 を入力に設定します。下に示すように，「コード生成（設計ツール）」－「ポート」を選択し，「ポート 5」を選択します。初期状態では，「使用しない」となっているので，「入力」を選択し，「内蔵プルアップ」もチェックしておきます。



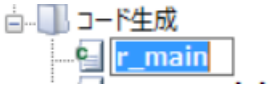
ポート 31 が「出力」になっていることを確認します。



これで，「コード生成（G）」をクリックして，コードを生成します。



コード生成が終わったら，r_main.c ファイルを開きます。



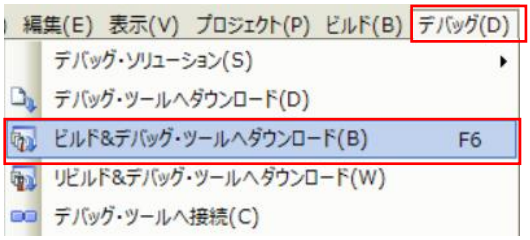
main 関数に以下のようにプログラムを書きます。

```
    while (1U)↓
    {↓
        if ( P5_bit.no0 == 0 )  /* スイッチが押されたかチェック */↓
        {↓
            P3_bit.no1 = 0;     /* P3.1を0（LEDをオン）にする   */↓
        }↓
        else↓
        {↓
            P3_bit.no1 = 1;     /* P3.1を1（LEDをオフ）にする   */↓
        }↓
↓
    }↓
```

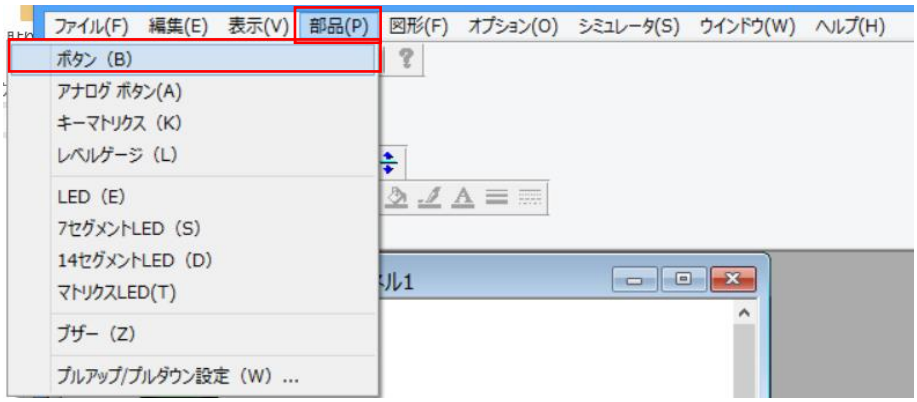
このプログラムでは，スイッチの状態を P50 で読み出し，その値を判定します。その値が 0 なら，P31 を 0 にして LED を点灯し，0 でなければ P31 を 1 にして LED を消灯します。

この処理を while で無限ループしています。入力が完了したら保存します。

これをビルドして，シミュレータにダウンロードします



シミュレータが起動したら，シミュレータ GUI ウィンドウを選択します。GUI では，メニューバーの「部品（P)」を選択して部品メニューから「ボタン（B)」を選択します。

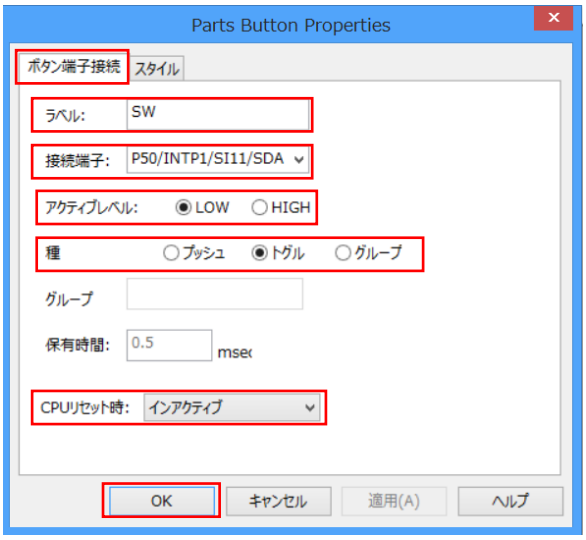


次に，「入出力パネル 1」でボタンを配置する場所をドラッグします。

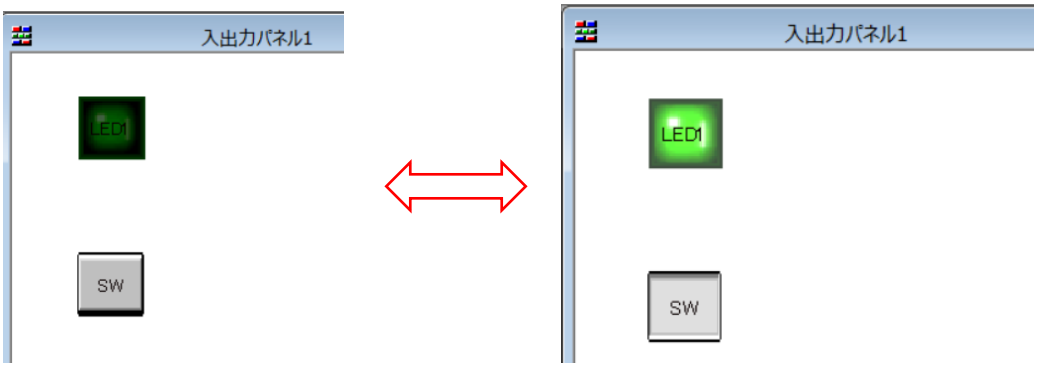
ここでは，左下に示すように，LED の下にボタンを作ります。右下に示すように，作成したボタンを右クリックしてメニューから「プロパティ（R)」を選択します

プロパティが表示されたら，「ボタン端子接続」で，「ラベル」を「SW」に設定し，「接続端子」は「P50/INTP1」を選択します。「アクティレベル」は「LOW」を選択し，「種」は「トグル」，「CPU リセット時」は「インアクティブ」にして，「OK」をプッシュして設定を完了します。

本来は，プッシュなのですが，GUI では，押したままの設定ができないので，動作が分かり易いトグルに設定しています。

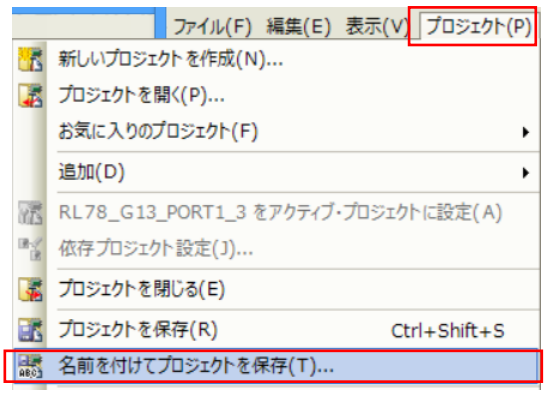


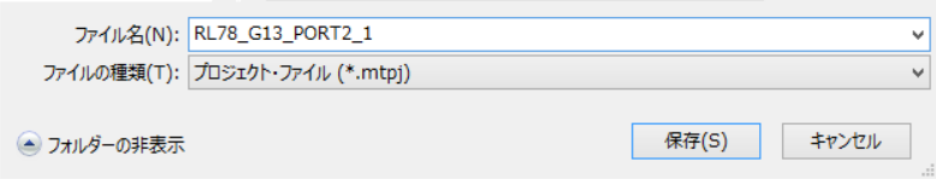
設定が完了したら，シミュレータウィンドウで  をクリックしてプログラムを実行します。実行を開始すると，シミュレータ GUI ウィンドウは左下の状態です。ここで SW をクリックすると右下の状態になります。右下の状態で再度 SW をクリックすると左下の状態になります。



確認ができたら，シミュレータウィンドウの右上の  の  をクリックして実行を停止します。  をクリックして，シミュレータを終了します。

CS+に戻ったら，このプロジェクトは「RL78_G13_PORT2_1」の名前で保存して，終了します。





保存したプロジェクトは，フォルダごとコピーし，「RL78_G13_PORT2_2」のフォルダ名にに変更しておきます。

ポート（スイッチ）の状態でプログラムの処理を変えるという処理内容は満足しました。

なお，上記のプログラムは個人的には好みではありません。このような単純な処理であれば，if 文のような判断分岐処理は必要なく，演算処理と言うか単にビットでの代入だけでも処理できます（このプロジェクトは，「RL78_G13_PORT2B」フォルダに格納しています）。

```
62          while (1U)
63          {
64
65  00172       P3_bit.no1 = P5_bit.no0;     /* LEDをSWに合わせる    */
66
67          }
```

9.2 ポート入力での LED 制御（その 2）

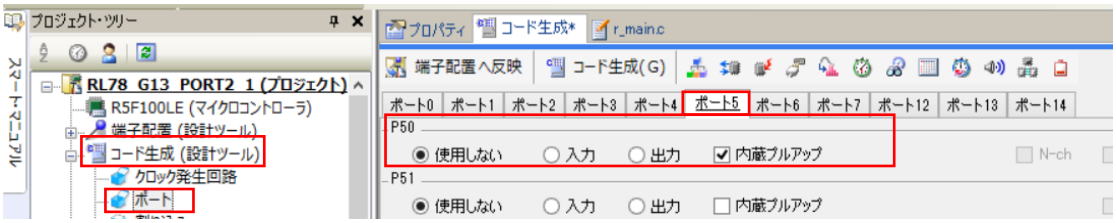
上で作成したプログラムは，単純な処理ですが，常にポートの状態をチェックしているので，省エネに反します。とは言っても，CPU が 32MHz でフルに動作しても数 mA で，LED の点灯電流と殆ど変わりませんが。

RL78/G13 は，複数の消費電力を削減する機能をもっています。動作クロックを低くすることも対策の一つですが，ここではスタンバイ機能（HALT 機能）を使ってみます。

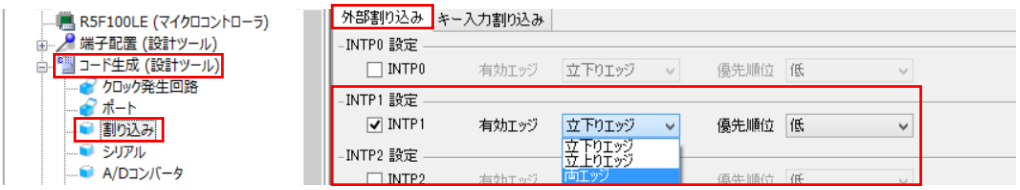
HALT();を実行させることで，HALT 状態に入り，CPU は停止します。HALT 状態から抜けるときには割り込みを使用します。

この方法は「9.1 ポート入力での LED 制御」で説明した方法にちょっと手を加えるだけで実現できます。

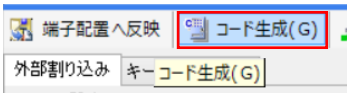
それでは，フォルダ「RL78_G13_PORT2_2」の「RL78_G13_PORT2_1.mtpj」をダブルクリックして CS+を起動します。コード生成で「ポート 5 を使用しない」に設定します。下に示すように，「コード生成（設計ツール）」－「ポート」を選択し，「ポート 5」を選択します。前回「入力」に設定していたものを「使用しないに」に設定します（RL78/G13 としては問題ないのですが，コード生成の制限で端子機能は 1 つにする必要があるので・・・）。



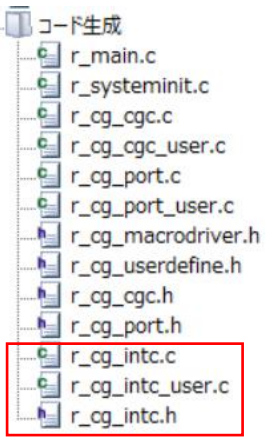
次に，割り込み機能の設定を行います。「コード生成（設計ツール）」－「割り込み」を選択します。下に示すように，「外部割り込み」で「INTP1 設定」をチェックし，「有効エッジ」を「両エッジ」に設定します。



ここまで設定したら，「コード生成（G)」をクリックして，コードを生成します。



コード生成された結果を右に示します。この一番下の３つが，外部割り込み関係のファイルです。r_cg_intc.c は，INTP1 の初期設定を行っている R_INTC_Create 関数（他の INTP は禁止している）と INTP1 の割り込み許可する R_INTC1_Start 関数と禁止する R_INTC1_Stop 関数を含んでいるファイルです。r_cg_intc_user.c は，割り込み処理を行う r_intc1_interrupt 関数の入れ物だけが生成されています。この中身を以下に示します。今回は，入れ物だけで中身は作りません。




```
static void __near r_intc1_interrupt(void)
{
    /* Start user code. Do not edit comment generated here */
    /* End user code. Do not edit comment generated here */
}
```

それでは，r_main.c を開いて，main 関数に以下に示すように while ループの最後に HALT(); を追加します。main 関数はこれだけです。

```
63      while (1U)
64      {
65          if ( P5_bit.no0 == 0 )  /* スイッチが押されたかチェック */
66          {
67              P3_bit.no1 = 0;     /* P3.1を0（LEDをオン）にする   */
68          }
69          else
70          {
71              P3_bit.no1 = 1;     /* P3.1を1（LEDをオフ）にする   */
72          }
73
74          HALT();                 /* CPUを停止する                */
75
76      }
```

次は，R_MAIN_UserInit 関数で R_INTC1_Start 関数を呼び出して INTP1 を許可します。

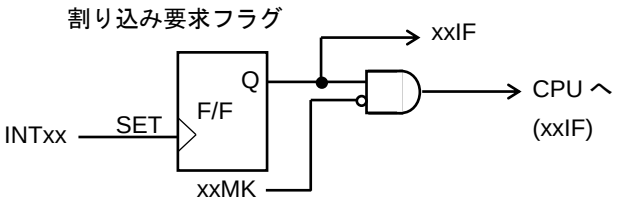
```
/* Start user code. Do not edit comment generated here */
R_INTC1_Start();          /* INTP1の動作を許可する      */
EI();
/* End user code. Do not edit comment generated here */
```

変更が完了したら，ビルドしてシミュレータにダウンロードします。これまでのように実行すると，同じように動作します。

【参考】ここで，割り込みについて若干説明をしておきます。

RL78 では，下図に示すように，割り込み要因（INTxx）は CPU に対する割り込み要求フラグ（xxIF）をセットするために使用されます。割り込み要求フラグ（xxIF）は，レジスタとして CPU から書き込んだり読み出したりすることができます。

xxIF 信号は割り込みのマスク信号（xxMK）でのマスク回路を経由して CPU へ伝えられます。



他の割り込み要求がない場合の各条件でのふるまいを以下の表に示します。

条件			スタンバイ	（ベクタ）割り込み	備考
xxIF	xxMK	IE			
0	x	x	保持	なし	割り込み要求なし
1	1	x	保持	なし	割り込みはマスク
1	0	0	解除	なし	割り込み禁止
1	0	1	解除	受け付け	割り込み処理

割り込み許可（IE=1）の状態で，マスクされていない（xxMK=0）割り込み要求（xxIF=1）でスタンバイは解除され，ベクタ割り込みが処理（受け付け）されます。すると，以下のような動作を行います。

①実行中の PC と PSW はスタックにセーブされ

②受け付けられた割り込み要求はクリアされ（xxIF=0）

③割り込み割り込み禁止（IE=0）となりそれ以上の割り込み受け付けは禁止

④受け付けた割り込みに対応するベクタで示される処理へ分岐します。

ここまでは，全てハードウェアで処理されます。この後，割り込み処理関数として記述されたプログラムが実行されます。

CPU は，割り込み処理を RETI 命令で終了します（割り込み処理関数では，自動的に最後が RETI になります）。

ここで使用した，コード生成された割り込み処理関数は，2 つの部分から構成されます。

①ベクトル定義部

下記のように pragma 指令で，r_intc1_interrupt 関数を INTP1 のベクタとして関係付けしています。

```
/***********************************************************************************
Pragma directive
***********************************************************************************/
#pragma interrupt r_intc1_interrupt(vect=INTP1)
```

②割り込み処理関数部

コード生成されただけでは，以下のように割り込み処理関数の中身はありません。

```
static void __near r_intc1_interrupt(void)
{
    /* Start user code. Do not edit comment generated here */
    /* End user code. Do not edit comment generated here */
}
```

しかし，ビルド結果をシミュレータで逆アセンブル表示すると，r_intc1_interrupt 関数には，RETI 命令が存在します。

```
57:     static void __near r_intc1_interrupt(void)
        _r_intc1_interrupt@1:
0012b       61fc        RETI
```

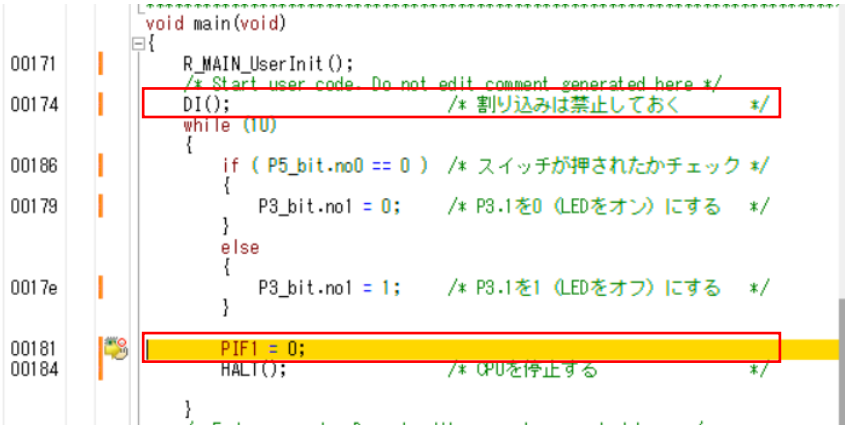
つまり，INTP1（P50/INTP1）端子の信号の立下りと立ち上がりエッジを検出して，割り込み要求が発生すると，スタンバイ（HALT 状態）が解除され，PC と PSW がセーブされ，割り込み要求がクリアされて，r_intc1_interrupt 関数が実行されます。関数には何も記述されてないので，RETI 命令だけが実行され，HALT()の次に戻り，while ループを繰り返すします。そこで端子の状態をポートとして読み出して LED を制御して再度 HALT 状態に入ります。

（このプロジェクトのシミュレータ用は，「RL78_G13_PORT2_2」フォルダにあります。E1 用は，「RL78_G13_PORT2_2_E1」フォルダにあります。

前ページの表を眺めると，割り込み禁止でもスタンバイを解除できます。つまり，同じような動作ができるように思われますが，少し注意が必要です。それは，割り込み要求フラグ（PIF1）の扱いです。割り込み許可状態で実行させた場合には，割り込みを受け付けた段階で PIF1 はクリアされますが，割り込み禁止ではクリアされません。この状態で HALT()を実行しても，直ぐに HALT モードが解除されてしまいます。これに対処するには，HALT()を実行するまでに PIF1 をクリアします。

割り込み禁止状態での処理プログラムを「RL78_G13_PORT2_2B」フォルダに作成しておきます。割り込み許可状態のプログラムとの違いは，main 関数だけです。main 関数の内容を示します。

違いは 2 箇所だけです。while ループの前に割り込みを禁止するための DI();を追加したことと，HALT();の直前に PIF1 = 0;を追加したことです。



思い通りの動作をしているかは，追加した PIF1 = 0;にブレーク・ポイントを設定し，ブレーク・ポイント付きで実行させることで確認可能です。

ダウンロード後に  をクリックして，1 回実行させると，すぐにブレークします（この状態が上の図になります）。再度  をクリックすると，今度は実行状態のままになります。この状態は HALT モードで CPU は停止しています。ここで，シミュレータ GUI ウィンドウで SW をクリックすると LED が点灯して，ブレークが掛かります。このことから，SW の状態が変化するまでは停止していることが分かります。

このプログラムのプロジェクトは「RL78_G13_PORT2_2B」フォルダに保存してあります。（E1 用は「RL78_G13_PORT2_2B_E1」フォルダになります。）

実際の機械式のスイッチには，切り替わるときに数十 ms 程度接触する部分が機械的に振動することで，ON－OFF が切り替わっていくチャタリングがつきものです。

次回は，タイマを用いたチャタリング対策です。チャタリング対策をした上で，いくつか SW を使ったプログラムを作っていく予定です。

以上