初心者のための RL78 入門コース（第４回：ポート入力と出力の組み合わせ）

第 4 回の今回は，スイッチ入力をもう少し突き詰めていきます。また，検証のためにシミュレータも新しい機能を利用します。最後に E1 経由でボードに接続してデバッグ（評価）します。

今回の内容

今回もかなりタイマを使用しましたが，次回からは，タイマの本番です。コード生成担当の鈴木さんから 3 桁の 7SEG-LED 表示ボードを頂いたので，インターバル・タイマを用いたダイナミック点灯をやってみることから始める予定です。

ここ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。

```
「ポート入出力」フォルダの構成↓
↓
第4回分↓
    +- RL78_G13_PORT3_1      --- 100msインターバルでのサンプリングでの入力↓
    +- RL78_G13_PORT3_2      --- 20msサンプリングでのノイズとチャタリング対策↓
    +- RL78_G13_PORT3_3      --- 上記のシミュレータでの確認↓
    +- RL78_G13_PORT3_4_E1   --- TAUのパルス間隔測定での対策例（E1用）↓
↓
```

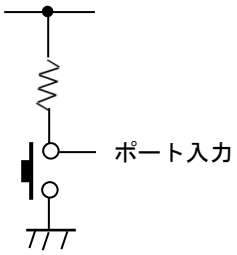
１０．スイッチの入力とチャタリング対策

前回作成したプログラムでは，スイッチの状態で LED の点灯を制御するもので，ある意味えでゃチャタリングがあっても，人は気が付かないというずるいプログラムです。今回はきちんとチャタリングに対応します。

10.1 スイッチのチャタリングとは

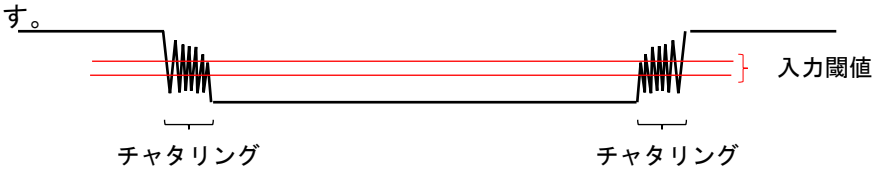
対策を行うには現象を理解する必要があるので，最初に簡単にチャタリングについて見直してみます。

右にスイッチ入力の回路例を示します。

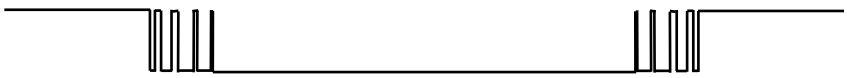


通常のスイッチの入力では，ポート入力は，抵抗を介して電源に接続され，スイッチでグランドに接続されています。RL78 のようにプルアップ抵抗が内蔵されていれば，スイッチだけ付けます。

この回路では，通常はハイ・レベルで，スイッチを押すとポート入力はロウ・レベルになり，放すとハイ・レベルに戻ります。このときの，ポート入力の波形は，下に示すようになると考えられます。

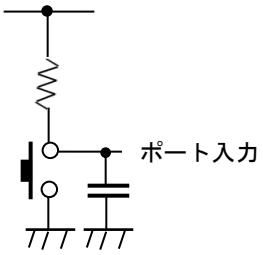


このように，スイッチを押したときと放したときにスイッチの接点が付いたり離れたりすることで信号が振動します。キー・ボード辺りでは，構造としてヒステリシスをもたせたり，接点に水銀を塗ったりしてチャタリングが発生しにくくしたものやもあるようですが，殆どのスイッチではチャタリングは避けられません。これを，ポートを通してみると，以下のように入力にパルス状の信号が出てきます（押したときだけでなく，放したときにも立下りエッジがあることにも注意してください）。



チャタリング対策は，殆どの場合はソフトウェアで対策しますが，ハードウェアで行うこともあります。

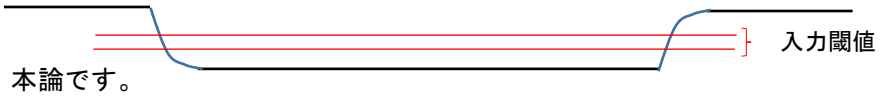
今回，使用した BlueBoard-RL78/G13_64pin では右に示すような回路になっています。抵抗が 10kΩ で，コンデンサは 0.1μF（100nF）なので，時定数は 1ms となっています。



かなり大容量のコンデンサが付いているのは，個人的には気になるところです。電源の瞬断（たとえば，電池駆動で電池交換時）に端子電圧が電源電圧以上になる可能性が考えられます。

もう一つ，スイッチを押したときに瞬間的にですが，スイッチの定格以上の大きな電流が流れ，寿命に影響することも考えられます。

ちょっと，横道にそれましたが，このように信号線にコンデンサを付けることで，信号を鈍らせています。たとえば，以下のような波形にしようとしています。

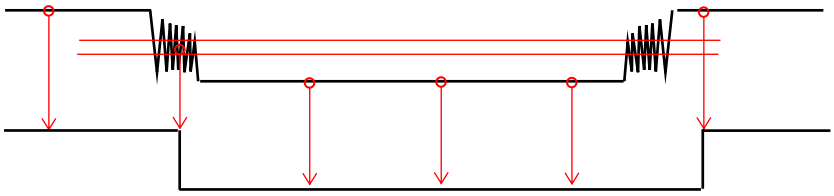


ここからが，本論です。

10.2 ソフトウェアでのチャタリング対策

通常，チャタリングは数十 ms 程度の期間発生します。長時間使用して材質が疲労すると長くなるようですが，ここでは，最大でも 100ms とみておきます。

一番簡単な対策は，100ms のインターバルでサンプリングすることです。下にサンプリングした結果を付け加えてみました。



2 回目のサンプリングがチャタリング期間に掛かっていますが，サンプリング結果は 0 か 1 のどちらかです。ですから，上の 2 回目のチャタリングと重なったところがどちらになるかは不明でも，チャタリングの影響でのポート入力の振動はなくなり，出力が 1 サンプリング時間だけ遅くなるかどうかになります。この方法が一番簡単なのですが，ノイズの影響は避けられません。

10.3 どのような動作例にするか？

LED が 1 個だけなので，点灯／消灯を制御しようとすると，スイッチを押すたびに切り替えるのが簡単です。つまり，押されたエッジをソフトで検出することにします。

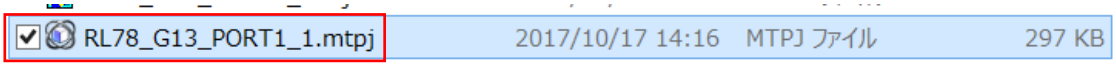
押されたエッジを検出するには，以前の状態を覚えておく必要があります。今回のように，1 個のスイッチの状態をチェックするだけであれば，変数を 1 個準備し，サンプリングした値（1/0）を変数に右からシフトインしていきます。この変数の下位 2 ビットが 10 ならスイッチが押されたことになります（11 なら押されていないし，01 なら放された，00 なら，押され続けているとなります）。

10.4 コード生成を利用したプログラム作成

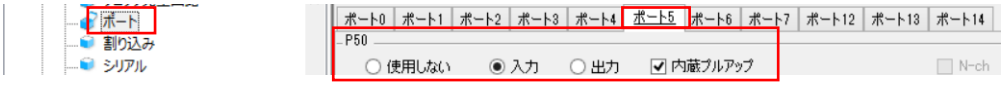
それでは，サンプリングのタイミングを作るために RL78/G13 のタイマ 00 をインターバル・タイマで使用します。

まず，「ポート入出力」という名前のフォルダを作成し，そこに，第 1 回で作成した「RL78_G13_PORT1」フォルダをフォルダごとコピーして，「RL78_G13_PORT3_1」に名前を変更しておきます。

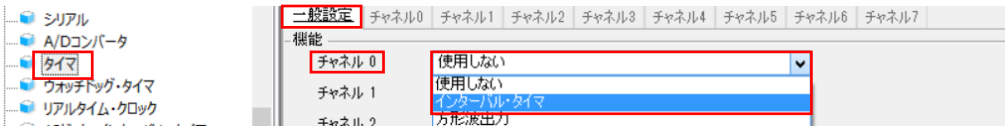
「RL78_G13_PORT3_1」フォルダのプロジェクト・ファイル「RL78_G13_PORT1_1.mtpj」をダブル・クリックして CS+を起動します。



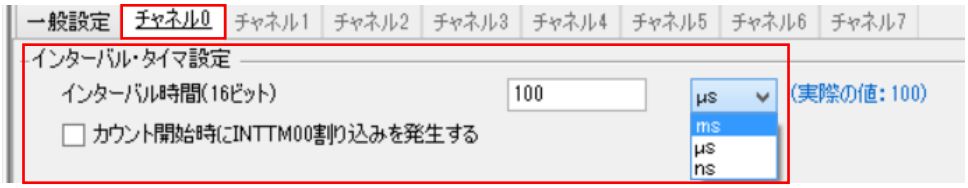
ワンポイント・アドバイスは「OK」をクリックして，CS+を起動します。CS+が起動したら，「コード生成（設計ツール）」の「ポート」を選択し，「ポート 5」を開き，「P50」を「入力に設定します。さらに，「内蔵プルアップ」にチェックしておきます。

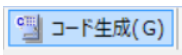


次に，インターバル・タイマの設定です。「タイマ」を選択し，「一般設定」を選びます。「チャネル 0」は「使用しない」となっているので，「使用しない」をクリックして，機能一覧を表示して，「インターバル・タイマ」を選択します。



インターバル・タイマに設定すると，「チャネル 0」のタグが選択可能になるので，選択します。初期状態では，「100μs」となっているので，100ms に設定するために，「μ」をクリックして選択できる単位を表示して，「ms」を選択します。



設定はこれだけなので， 

をクリックしてコードを生成します。

コードが生成されたら，「r_cg_timer_user.c」を開きます。今回はソフトウェアでエッジ検出を行うので，前回の値を保持しておく必要があります。そこで，8 ビットのグローバル変数”g_port”を定義しておきます。タイマ 00 のタイマ割り込みごとに変数”g_port”を左シフトし

て，読み出したスイッチの状態（P50）を LSB に付け加えます。これで，g_port には bit0 が最新のデータ，bit1 が前回のデータ，bit2 がさらにその前のデータと格納されていきます。

変数 g_port の内容

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0		P50
Di-7	Di-6	Di-5	Di-4	Di-3	Di-2	Di-1	Di	⇦	SW

変数 g_port の下位 2 ビットをチェックし，10 になったら，スイッチの押下エッジ検出フラグ（8 ビットのグローバル変数 g_edge）をセットします。

main 関数では，このフラグをチェックして，セットされていたら LED を反転させます。

下に示すように，グローバル変数を定義しておきます。頭の”volatile”は最適化で消されることを防ぐために宣言しておきます。

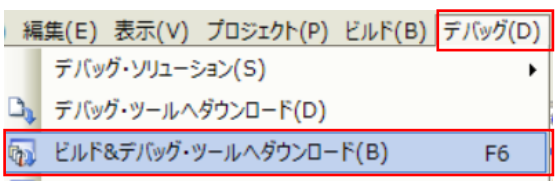
```
/***********************************************************************
Global variables and functions
***********************************************************************
/* Start user code for global. Do not edit comment generated here */
volatile    uint8_t g_port = 0xFF;          /* ポート入力変化       */
volatile    uint8_t g_edge = 0x00;          /* スイッチ押下フラグ   */
/* End user code. Do not edit comment generated here */
```

次は，割り込み関数での処理です。

以下に示すように，3 つの処理から構成されています。最初は，保持データを左シフトしてポートの状態を取り込めるように準備ます。2 つ目で，P50 の状態を変数に取り込みます。最後に，最新のデータをチェックしてフラグに反映します。

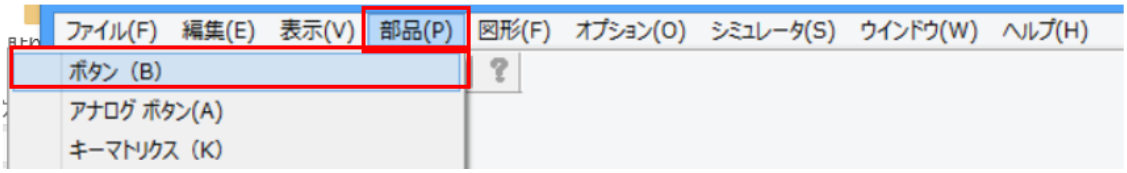
```
static void __near r_tau0_channel0_interrupt(void)
{
    /* Start user code. Do not edit comment generated here */
    g_port  = ( g_port << 1 );          /* データを左シフトする */

    if ( P5_bit.no0 == 1)               /* スイッチをチェック   */
    {                                   /* 1なら変数のLSBを1に  */
        g_port += 1;
    }

    if ( ( g_port & 0b00000011 ) == 0b00000010 )
    {                                   /* 立下りならフラグをセット
        g_edge = 0x01;
    }
    /* End user code. Do not edit comment generated here */
}
```

これをビルドしてシミュレータにダウンロードします。



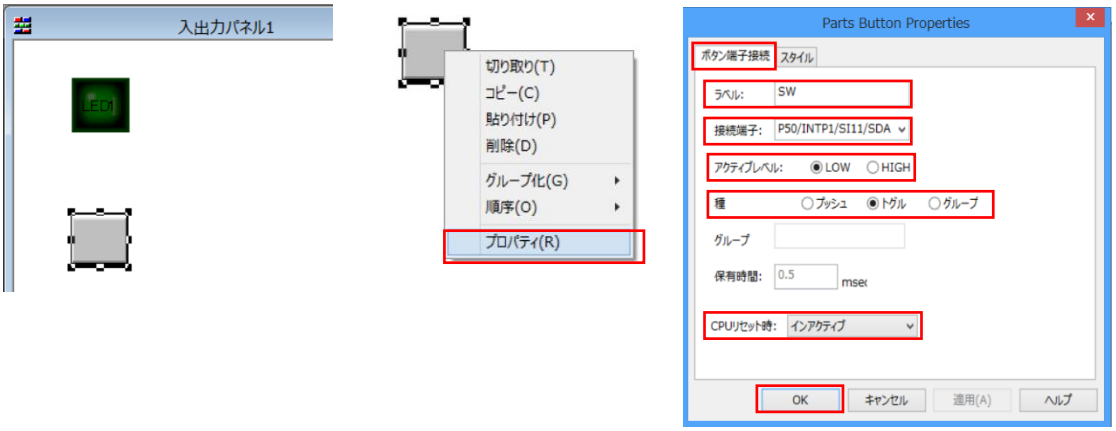
10.5 シミュレータでの動作確認

シミュレータの GUI ウィンドウを見ると，LED しか定義されていないので，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。

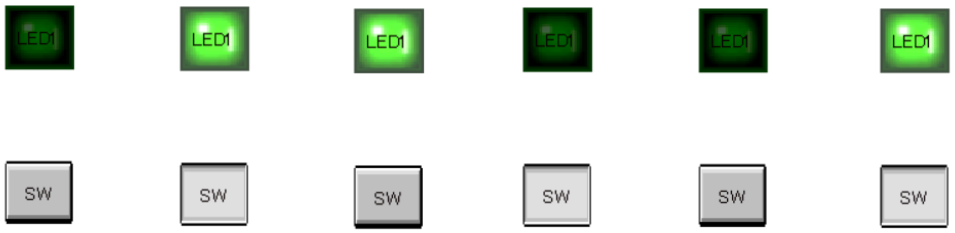


次に，「入出力パネル 1」でボタンを配置する場所をドラッグします。（左下の図）

ここでは，左下に示すように，LED の下にボタンを作ります。下中央に示すように，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。右下に示すように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（P49 で行ったのと同じ手順です)。



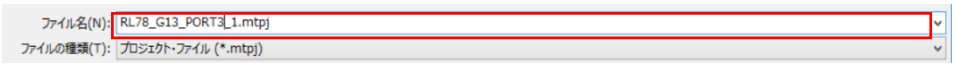
設定が完了したら，シミュレータウィンドウで  をクリックしてプログラムを実行します。実行を開始すると，シミュレータ GUI ウィンドウは下の左端の状態です。ここで SW をクリックすると右隣の状態になります。この状態で再度 SW をクリックすると左端から 3 番目の状態になります。スイッチは放されているのに LED は点灯していることから以前の処理とは変わっていることが分かります。再度，SW をクリックします。これを繰り返していくと下のようになり，スイッチを押すごとに LED の状態が変化していることが分かります。



この動きは目標とした動作と一致しています。

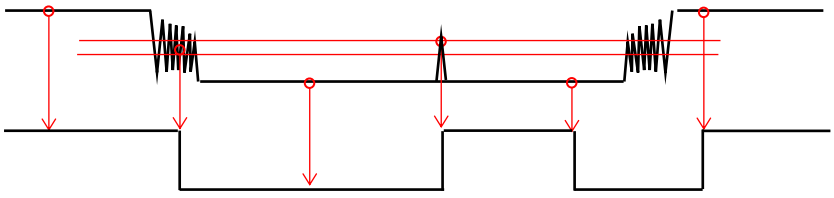
目標とした動作が確認できたので，シミュレータのウィンドウに切り替えの をクリックし実行を停止し， クリックしてシミュレータから切り離しデバッグを終了します。

この状態を新しいプロジェクト名（RL78_G13_PORT3_1.mtpj）で保存して CS+を終了します。



１１．スイッチ入力のチャタリングとノイズ対策

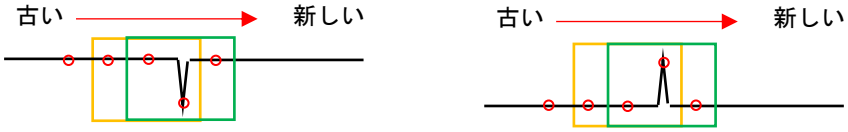
前章でチャタリング対策を示しましたが，ある意味で手抜きの対策です。サンプリング間隔が長いので，応答時間が長くなることと，サンプリングのタイミングでノイズが載ると，それを正規の信号として取り込んでしまい，下に示すような異常なスイッチ押下の検出となります。



11.1 ノイズへの対策とは

サンプリング時のノイズ対策は，複数回同じ状態が続いたときにその状態が正しいと判断し，その回数未満ならそれはノイズとして無視することで実現します。ここでは，2 回以上連続した

ら正常と判断します。それでは，ノイズと判断するケースを考えてみます。ノイズと判断するのは，下に示すように，グリーンで囲んだ 3 回のサンプリング結果が”1”,”0”,”1”と変化する場合と”0”,”1”,”0”と変化する場合だけになります（その左側のオレンジで囲んだ 3 回分のサンプリング結果ではまだノイズとは判断できません）。



この 2 つの場合だけなので，サンプリングしたデータ（変数 g_port）の下位 3 ビットをチェックして”101”の場合には”111”に，”010”の場合には”000”に置き換えることでノイズの影響を除去することにします。これで，ビット 1 より上位のビット（古いデータ）は安定したビットと考えることができます。

（複数のスイッチ入力が対象の場合には，3 回分のサンプリング結果を演算（連続した 2 つのデータの排他的論理和）して，2 回連続して変化（排他的論理和が 1）ならノイズ除去を行いますが，スイッチ 1 つなら場合分けの方が分かり易いので今回の方法を使います。）

11.2 エッジ検出

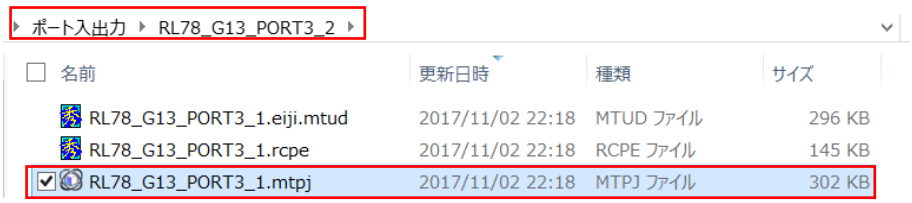
その後に変数 g_port のビット 2 と 1 をチェックすることでスイッチが押された／放された／変化なしを判断します。ビット 0 はまだ値が確定できないのでチェック対象にはしません。

今回のノイズ対策は，ある程度チャタリング対策にもなるので，チャタリングは 60ms 続かないと判断して，サインプリングの間隔を 20ms に短くします。これは，チャタリングでの誤動作は，立下り時に 0011 とサンプリングしたときだけです。つまり，4 サンプリング目（最後の 1 に相当）でハイ・レベルの閾値を超える場合だけです。それ以外の場合は，ノイズ対策で除去されます。このことから 60ms の 1/3 の 20ms のサンプリング時間としています。実際はもっと短くても問題ないはずですが，余裕を見ています。

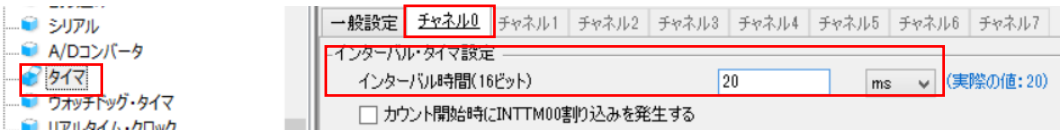
11.3 コード生成での変更

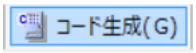
ここでは，「10.スイッチ入力とチャタリング対策」で作成した「RL78_g13_PORT3_1」フォルダを丸ごとコピーして「RL78_g13_PORT3_2」に名前を変更したものを使います。

フォルダ内の RL78_G13_PORT3_1.mtpj をダブル・クリックしてプロジェクトを開きます。



プロジェクトが開いたら，プロジェクト・ツリーの「コード生成（設計ツール）」で「タイマ」を選択し，「チャネル 0」タグを開きます。「インターバル・タイマ設定」の「インターバル時間（16 ビット）」を 20ms に変更します。

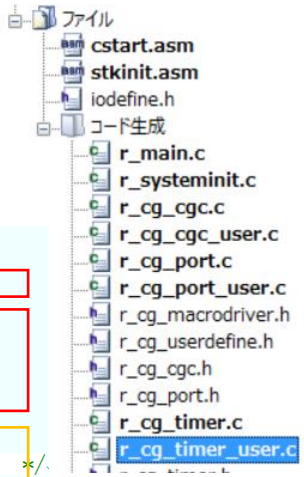




をクリックしてコードを生成します。これで，インターバル時間が 20ms の設定に変更されます。

11.4 プログラムの変更

今回は，生成されたファイルの中で，インターバル・タイマ割り込み関数が含まれる「r_cg_timer_user.c」を「11.1 ノイズへの対策とは」で説明した内容に沿ったプログラムに変更します。下に示すのが前回作った割り込み関数です。




```
static void __near r_tau0_channel0_interrupt(void)
{
    /* Start user code. Do not edit comment generated here */
    g_port  = ( g_port << 1 );              /* データを左シフトする */

    if ( P5_bit.no0 == 1)                   /* スイッチをチェック   */
    {                                       /* 1なら変数のLSBを1に  */
        g_port += 1;
    }

    if ( ( g_port & 0b00000011 ) == 0b00000010 )
    {                                       /* 立下りならフラグをセット    */
        g_edge = 0x01;
    }
    /* End user code. Do not edit comment generated here */
}
```

今回は，上 2 つの赤く囲んだ部分はそのままで，3 つ目のオレンジで囲んだ部分だけを変更します。変更した結果を下に示します。まだまだ，短いですが，やっと，プログラムらしくなってきました。

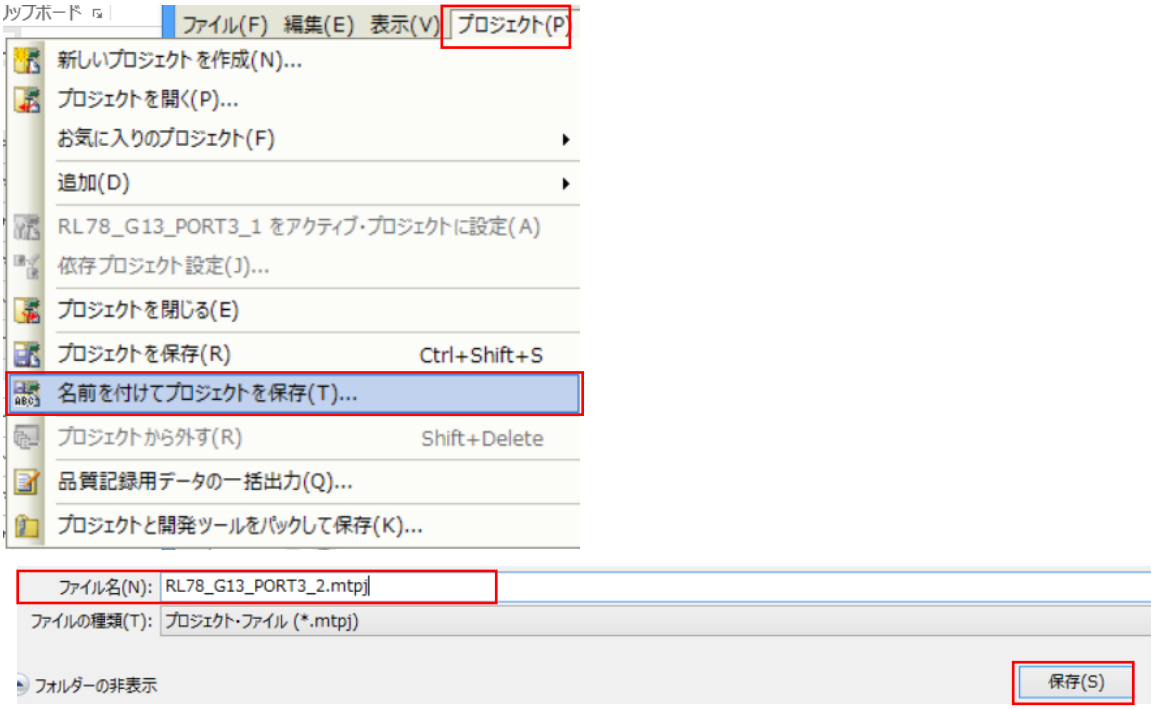
```
/* 最新の3回分のサンプル値を調べ，ノイズ対策を行う  */

    if ( ( g_port & 0b00000111 ) == 0b00000010 )
    {                                       /* ハイのノイズの場合   */
        g_port &= 0b11111101;               /* ビット1をクリア      */
    }
    else
    {
        if ( ( g_port & 0b00000111 ) == 0b00000101 )
        {                                   /* ロウのノイズの場合   */
            g_port |= 0b00000010;           /* ビット1をセット      */
        }
    }

```

```
/* 立下りエッジをチェックし，エッジ検出でフラグをセット */
if ( ( g_port & 0b00000110 ) == 0b00000100 )
{                                   /* 立下りならフラグをセット */
    g_edge = 0x01;
}
```

最初の赤で囲んだ部分が，ロウの状態でハイのノイズが載った場合のノイズ除去です。2 つ目が，ハイの状態でロウのノイズが載った場合のノイズ除去です。3 つ目が立下りエッジ検出部分です。最後のデータ（LSB）は値が確定していないので，その上（前）の 2 ビットを判断に使用します。このように，ノイズ除去では遅延はつきものです。今回は 1 サンプリング分のノイズ除去なので，結果を得るのに，1 サンプリング時間（20ms）だけ検出が遅れます。

一応，動作確認しましたが，現在使用している環境では，チャタリング自体が発生しないので，今回は，結果を示しません。

今回も，新しい名前（RL78_G13_PORT3_2.mtpj）でプロジェクトを保存しておきます。



保存が完了したら，CS+を閉じます。

１２. RL78 でのスイッチ入力とチャタリング及びノイズ対策の確認

スイッチ入力は基本的には入力ポートですが，これまで説明した中にもあるように，押されたことの検出に外部割り込みも使えますし，まだ説明していませんが，「キー・リターン（キー割り込み）」はスイッチ入力に対応した機能で，押したことだけを検出して割り込みを発生させ，後は入力ポートを用いてどのスイッチが押されたかを確認するというものです。スイッチが押されたことを検出する方法を表にまとめてみます。

検出方式	検出方法	備考
外部割り込み	ハードウェアでのエッジ検出	STOP 解除可能
インターバル・タイマ	一定間隔で，ソフトウェアで確認	通常 STOP 解除不可
キー割り込み	ハードウェアで立下り検出	STOP 解除可能
タイマの外部入力	パルス間隔測定でのエッジ検出	ノイズ除去可能

今回（第 4 回）で説明したインターバル・タイマを用いた方法は，タイマとソフトウェアで押されたことを検出する簡単な方法ですが，常にタイマが動作していないと使えません。

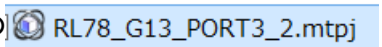
いずれの方法も，ノイズに対してはソフトウェアで対策する必要がありました。ノイズ対策をハードウェアで行う方法もあります。それは，タイマの入力機能を使う方法です。タイマをパルス間隔測定モードに設定すると，入力の指定されたエッジの検出で割り込みを発生しますが，同時にノイズ・フィルタを有効にすることができます。動作クロックを適切に選択すれば，有効なノイズ対策となります。さらに，カウント値からチャタリングの判断も可能です。（13.でプロジェクトを説明します。）

【備考】RL78 で考慮すること

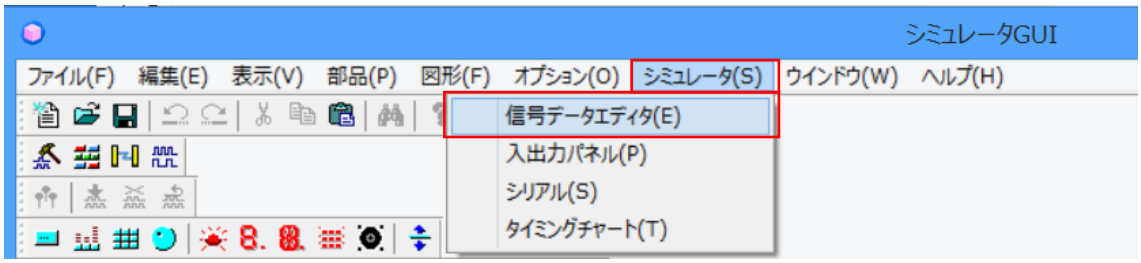
RL78 の大きな特徴は省電力です。最も動作電流が少なくなるのは，STOP モードです。ここまで説明した TAU によるインターバル・タイマはクロック発振が停止してしまう STOP モードでは動作できません。これに対して外部割り込みやキー割り込みは STOP モードでも使用できます。

つまり，外部割り込みやキー割り込みは，STOP モードの解除に使用し，その後のスイッチのチャタリングやノイズ対策では，タイマを使う使い分けが考えられます。ここらは，後日キー・マトリクスの例辺りで触れることになります。

12.1 シミュレータ GUI でのチャタリングの確認準備

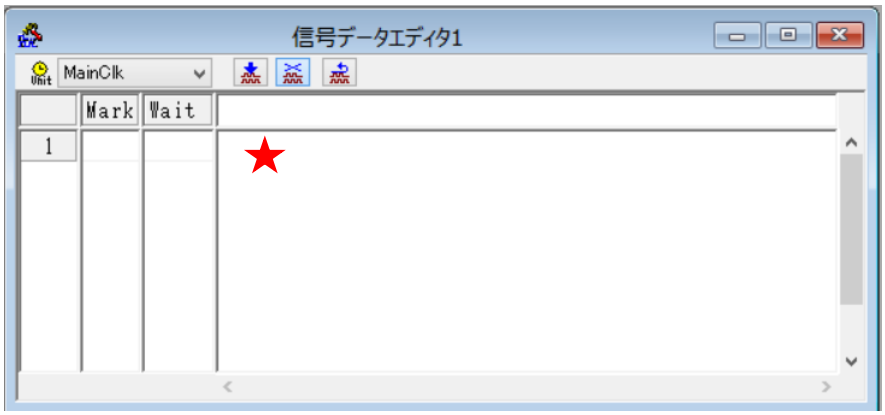
「RL78_G13_PORT3_2」フォルダをコピーして，「RL78_G13_PORT3_3」に変更しておきます。そのフォルダの  をダブル・クリックして，プロジェクトを開きます。

今回はチャタリング対策の効果を確認するために，シミュレータの「信号データエディタ」を使用します。シミュレータ GUI のメニューバーで「シミュレータ(S)」をクリックしてプルダウンメニューを開き，「信号データエディタ(E)」を開きます。

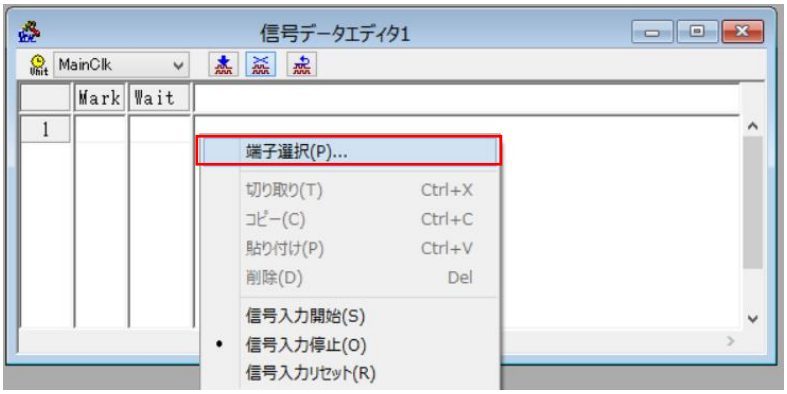


信号データエディタは端子に入力する信号波形を生成するために使用する機能です。ここでは，チャタリングを含む入力信号を作成して，その信号をスイッチの代わりに使用するものです。信号データエディタを指定すると，「信号データエディタ 1」というパネルが開きます。

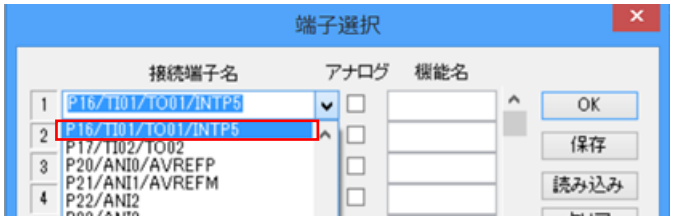
下に示す赤い★の部分で右クリックするとメニューが開きます。



表示されたメニューの「端子選択(P)...」を選択します。

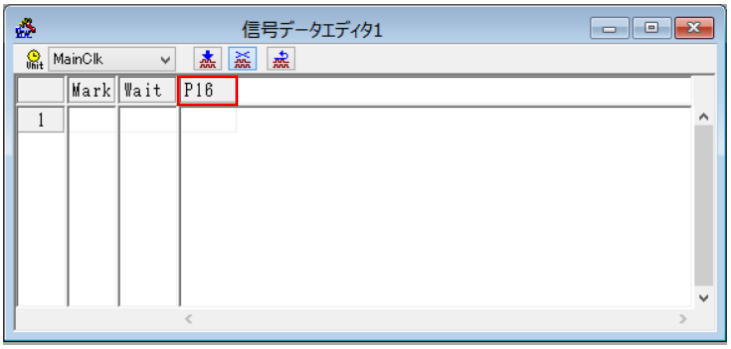


端子選択ウィンドウが表示されたら，「1」の右側の  をクリックして接続する端子名の一覧を開きます。そこで，「P16/・・・・」を選択します。

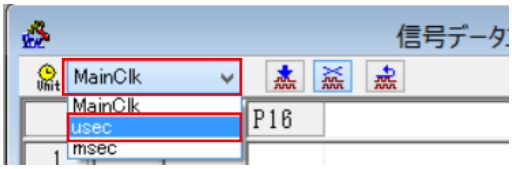


機能名には，P16 と名前を付けて，「OK」をクリックします。

すると，下に示すように，「P16」が表示されます。（2 以降の接続端子名に機能名を設定すると，P16 の右側に追加されていきます。

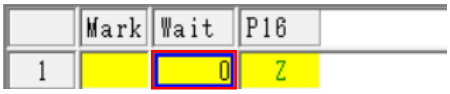


この状態で基準となるクロックが CPU のクロック（Main Clk）になっているので，その右側の をクリックしてメニューを開いて，「usec」を選択します。ここで指定した usec は「Wait」の欄で指定する数値の単位となります。

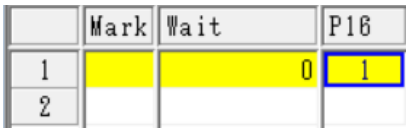


「1」（最初の行）の「Wait」の欄に「0」を入力すると，左下のように，バックがイエローに変わり，P16 の下に「Z」が表示されます。イエローは，この行から信号が入力されることを

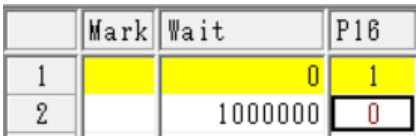
示します。Wait は起動時に右側で示す状態にするまでの待ち時間を示します。ここの「Z」はハイインピーダンスを示します。ここでは右下に示すように「1」に変更します。



このままでは，Wiat の欄の幅が狭いので，「Wait」と「P16」の間に区切りをドラッグして，幅を広げます。



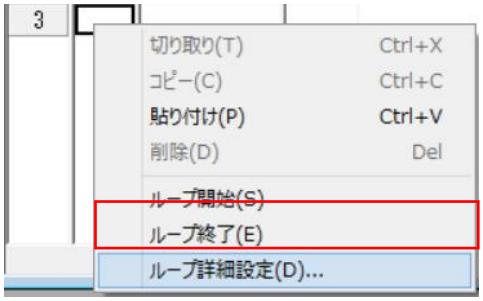
2 行目の「Wait」に「1000000」（1 秒）をセットし，「P16」に「0」をセットします。これで，1 行目で 1 にして，その後 1 秒間は 1 を保持して 0 にします。ここが，スイッチが押されたタイミングになります。



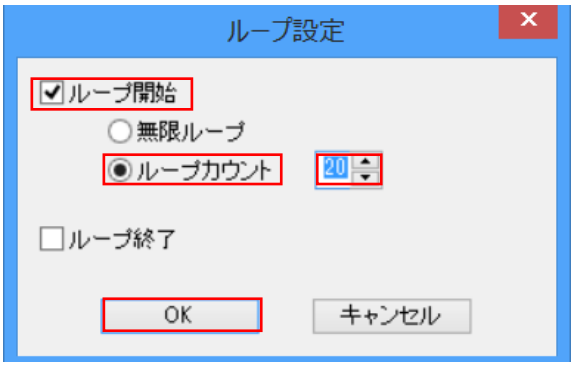
次にチャタリング部分を作ります。チャタリングの最初の部分は 0.5ms の 0 と 0.75ms の 1 の繰り返しとします。

3 行目の「Mark」の欄を右クリックしてメニューを表示し， 

をクリックします。



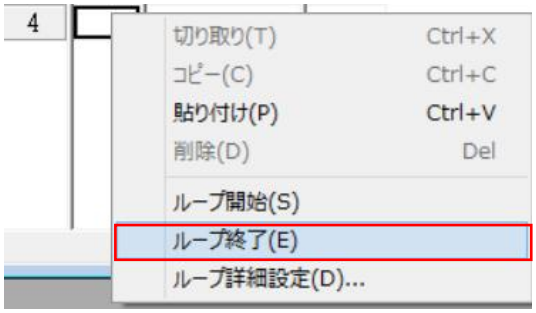
ループの設定パネルが表示されるので，「ループ開始」にチェックし，「ループカウント」を選択し，回数を 20 に設定して「OK」をクリックします。



次に P16 の下の欄に 1 をセットし，0.5ms 待ってから 1 にします。

	Mark	Wait	P16
1		0	1
2		1000000	0
3	20	500	1
4			

次に，4 の行の「Mark」の欄を右クリックして，メニューから ループ終了(E) を選択すると，「Wait」が 0，P16 が 1 となります。



	Mark	Wait	P16
1		0	1
2		1000000	0
3	20	500	1
4		0	1

そこで。「Wait」を 0.75ms，「P16」を 0 にします。これで，3 及び 4 行目で，0 が 0.5ms で 1 が 0.75ms の信号を 20 パルス出力します。

	Mark	Wait	P16
1		0	1
2		1000000	0
3	20	500	1
4		750	0

次に，0 と 1 が 0.5ms の部分を 15 回分作成します。ここまでと同じやり方でループ回数と 1 の時間が異なるだけで，以下のようになります。

	Mark	Wait	P16
1		0	1
2		1000000	0
3	20	500	1
4		750	0
5	15	500	1
6		500	0

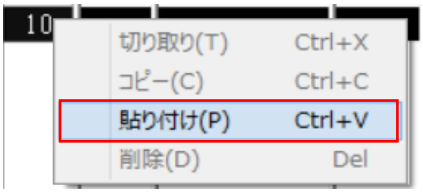
チャタリング部分の最後として，1 の時間を 0.25ms にしたものを 30 回分作成します。

	Mark	Wait	P16
1		0	1
2		1000000	0
3	20	500	1
4		750	0
5	15	500	1
6		500	0
7	30	500	1
8		250	0

これで，チャタリング部は完了したので，次はスイッチが押されたことを示す 1 秒間 0 を保持して，スイッチを放したことに対応する 1 への立ち上がりです。

	Mark	Wait	P16
1		0	1
2		1000000	0
3	20	500	1
4		750	0
5	15	500	1
6		500	0
7	30	500	1
8		250	0
9		1000000	1

立ち上がりでも，チャタリングは発生するので，立下りと逆の順番で作成しておきます。そこで，左の 7 と 8 をドラッグして選択し，右クリックでメニューを表示してコピー(C)をクリックします。10 を選択して，右クリックしてメニューを表示して貼り付け(P)をクリックします。

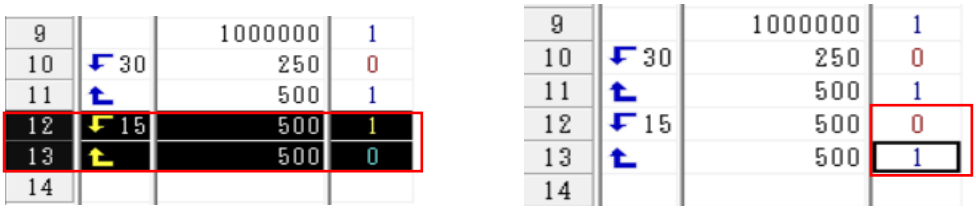


これで，10，11 行目に 7，8 行目の設定がコピーされます。「Wait」と「P16」の値を変更します。

7	30	500	1
8		250	0
9		1000000	1
10	30	500	1
11		250	0
12			

7	30	500	1
8		250	0
9		1000000	1
10	30	250	0
11		500	1
12			

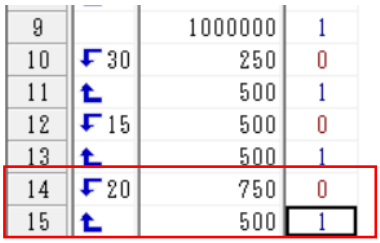
同様に 5，6 行目を 12 行目にコピーします。今度は「P16」のデータだけを変更します。



9		1000000	1
10	30	250	0
11		500	1
12	15	500	1
13		500	0
14			

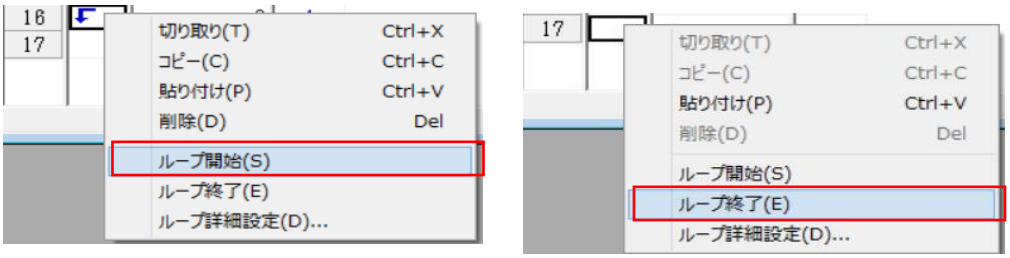
9		1000000	1
10	30	250	0
11		500	1
12	15	500	0
13		500	1
14			

最後に 2，3 行目を 14 行目にコピーして，以下のように変更します。

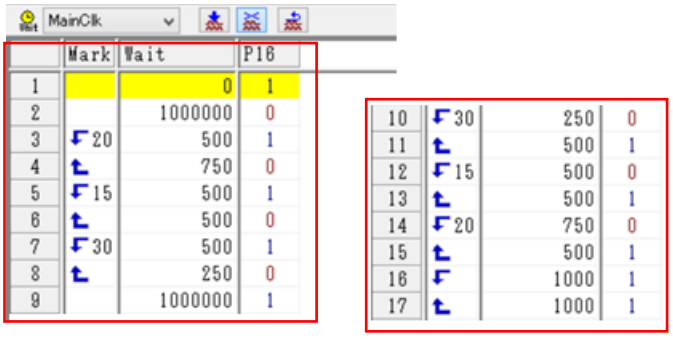


9		1000000	1
10	30	250	0
11		500	1
12	15	500	0
13		500	1
14	20	750	0
15		500	1

最後に，スイッチを放した状態で無限ループさせておきます。16 行の「Mark」で右クリックしてメニューを表示したら，ループ開始(S) を選択します。その上で，「Wait」を適当に 1000 位にして P16 は 1 にしておきます。17 行の「Mark」で右クリックしてメニューを表示したら，ループ終了(E) を選択します。ここも「Wait」を 1000 にしておきます。



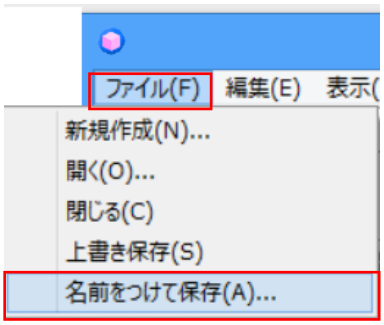
最終的には，以下のようになります（実際にシミュレータで実行させるときに 2 行目と 9 行目の 1 秒では時間がかかり過ぎるので，100m 秒（100000）に変更しておきます）。

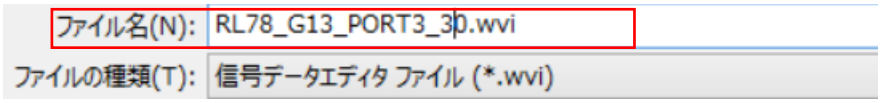


	Mark	Wait	P16
1		0	1
2		1000000	0
3	20	500	1
4		750	0
5	15	500	1
6		500	0
7	30	500	1
8		250	0
9		1000000	1

10	30	250	0
11		500	1
12	15	500	0
13		500	1
14	20	750	0
15		500	1
16		1000	1
17		1000	1

信号データエディタのウィンドウを選択した状態で，，シミュレータ GUI のメニューバーの ファイル(F) をクリックして，「名前を付けて保存(A)...」をクリックします。ファイル名（ここでは，「RL78_G13_PORY3_30.wvi」として保存しておきます。





このファイルは他でも使う可能性があるので，別のところにコピーしておいた方がいいでしょう（ここで保存したファイルはそのままでは，プロジェクトを保存終了した段階で消えてなくなるようです）。

最後に，チャタリングがきちんと除去されたかどうかを確認するために，r_main.c のグローバル変数の定義部に以下のように変数 g_count を定義しておきます。

```
/* Start user code for global. Do not edit comment generated here */
extern  volatile    uint8_t g_edge;
volatile    uint8_t g_count = 0;
/* End user code. Do not edit comment generated here */
```

この変数を，main 関数のエッジ検出処理のところでカウントアップしておきます。

変更した結果をビルドして，シミュレータにダウンロードしておきます。この追加したところにブレークポイントを設定しておきます。

```
    if ( g_edge )
    {
        P3 ^= 0x02;          /* スイッチ押下フラグが1なら    */
        g_edge = 0x00;       /* P3.1(LED)を反転する          */
        g_count++;           /* 押下フラグをクリアする       */
    }
```

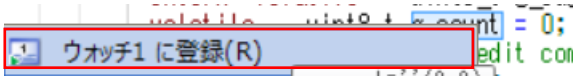
これで，準備ができたので，  をクリックしてブレークポイントを有効にして実行します。

すると，何もしないのにブレークが掛かります。このときの変数 g_port の値をウォッチで確認してみます。

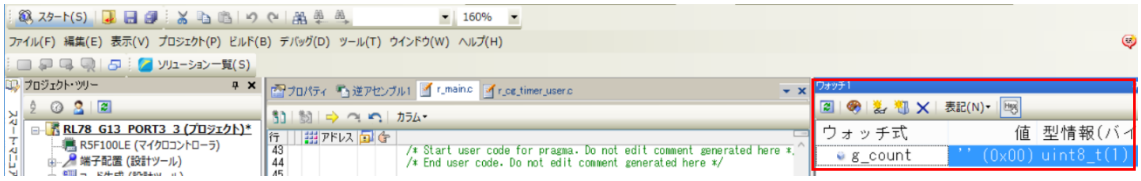
12.2 チャタリングの確認準備（デバッグ機能準備）

ここまでは，シミュレータの GUI 機能を使ってプログラムの確認を行ってきました。これからは，本来のデバッグ機能を使ってデバッグします。具体的には変数の値を確認することにします。どちらかと言うと，こちらが本来のやり方です。

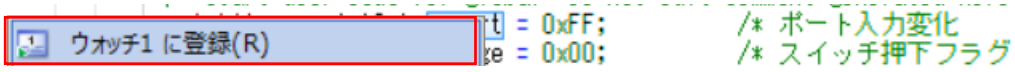
r_main.c で新しく定義した変数 g_count を選択（g_count の文字をドラッグ）して右クリックします。すると，メニューが表示されるので，下のように，「ウォッチ 1 に登録(R)」をクリックします。



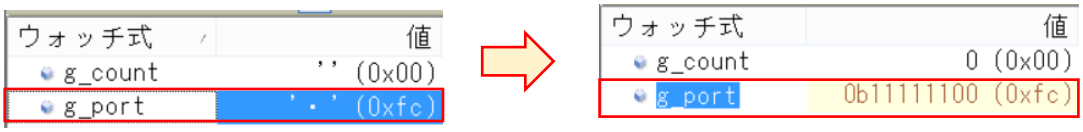
すると，シミュレータウィンドウの右に「ウォッチ 1」パネルが表示され，そこに，指定した変数 g_count が 00 であることが表示されています。



同様に，r_cg_timer_user.c で変数 g_port をウォッチ 1 に登録します。



これで，ウォッチ 1 に各変数の値が下図のように表示されています。左端が変数名で，次が変数の値です。ここで，表記は（ ）の中が 16 進数で，その前が自動での表示です。どのような形式で表記するかは変数を選択して，右クリックしてメニューを表示することで，メニューに表示された形式（2 進，8 進，16 進，符号付 10 進，符号なし 10 進，ASCII など）に変更できます。また，下記のように 16 進数を併記するかどうかも選択できるので，分かり易い表記を選択してください。右下は，変数 g_count を 10 進数，変数 g_port は 2 進数にした例です。



変数 g_port は初期値 0xFF だったので，実行後に 2 回分 0 が検出されたためにエッジを検出し，ブレークが掛かったと考えられます。この段階では，まだ信号データエディタで作成したデータは入力されていません。つまり，何もしていない状態では，端子入力は 0 だということが分かりました。この誤検出を避けるもっとも簡単な方法は実行開始前に信号データエディタからの信号入力を開始します。

【備考】

なお，ウォッチ 1 パネルはメニューバーからも表示させることができます。どちらかと言うと，こちらの方がウォッチ 1 以外に多くの情報を表示させることができます。

シミュレータのメニューバーの 表示(V) をクリックしてプルダウンメニューを表示します。

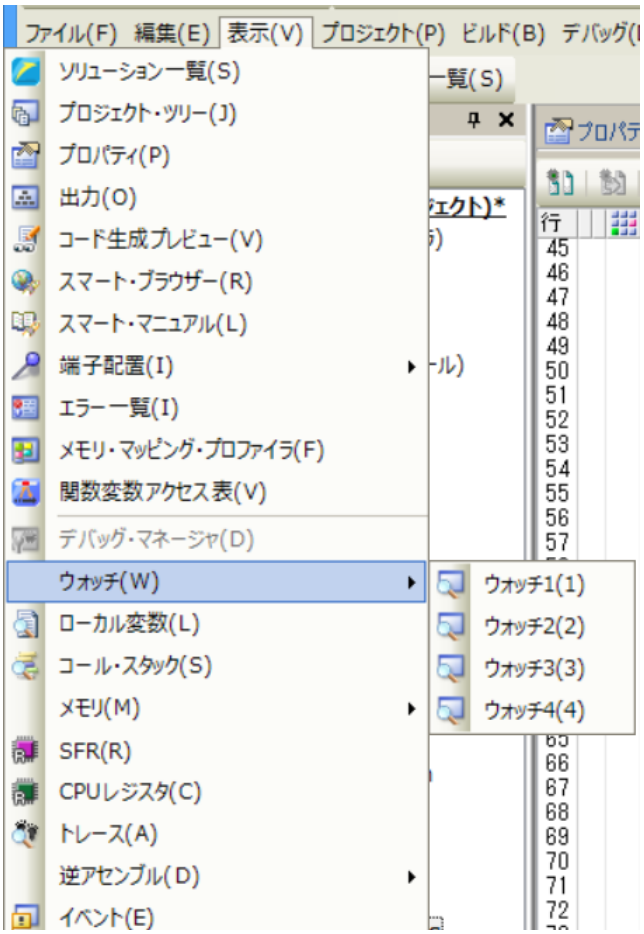
メニューから ウォッチ(W) にマウスをもっていくと，右に示すように，ウォッチ 1～ウォッチ 4 があることが分かります。

ウォッチ以外にも，ローカル変数(L)，メモリ(M)，SFR(R)，CPU レジスタ(C)，イベント(E)が選択できることが分かります。

また，メモリや逆アセンブルは ▸ が付いていることから複数選択可能だということが分かります。

ここで，できるのは，表示領域で選択できるようにすることで，そこで表示する内容は別途指定する必要があります。

ここらは，ウォッチ 1 だけが特別なことが分かります。

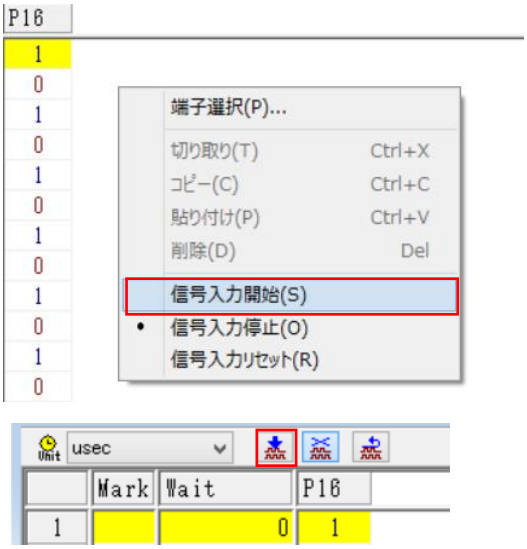


12.3 チャタリング除去の確認

シミュレータ GUI ウィンドウの信号データエディタを選択しておき，右側の部分（P16 の下）で右クリックすると，メニューが表示されるので，信号入力開始(S)をクリックすることで信号が P16 に入力されます。

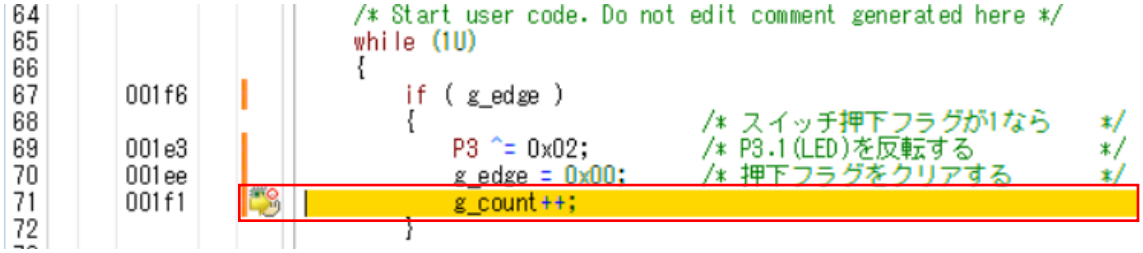
右下に示す  をクリックしても信号の入力を開始できます。

ただし，実行停止状態で，信号入力を開始しても，直ぐには開始しません。プログラムの実行を開始から信号入力が開始されます。

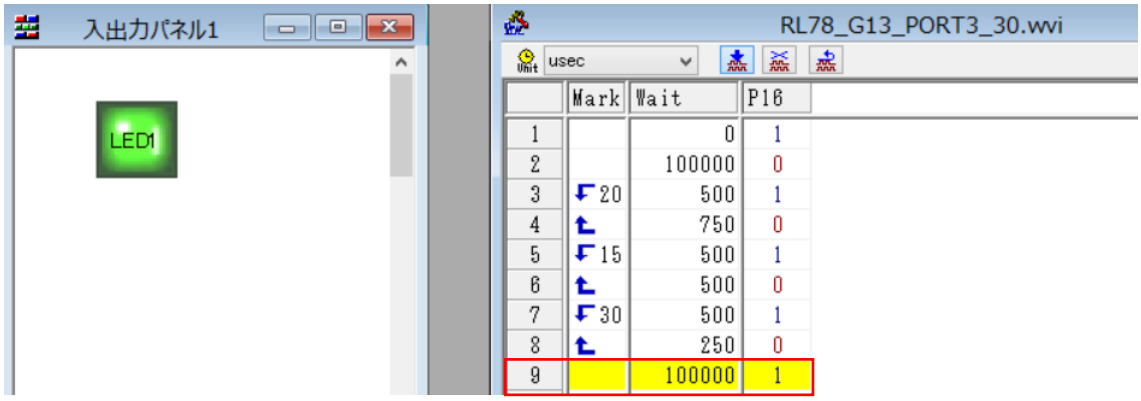


再度ブレークポイントを有効にして実行  します。

暫く，実行状態が続いた後，下に示すようにブレーク状態になります。

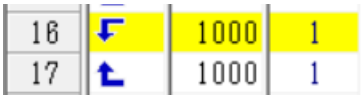


ここで，シミュレータ GUI ウィンドウを表示すると，以下のように LED が点灯し，信号データエディタは 9 行目になっています。つまり，3 行目～8 行目のチャタリング部分では誤検出は起こっていないことが分かります。



この状態で，再度  をクリックして実行を再開します。今度は，チャタリング以外の立下りエッジが存在しないので，ブレークはかからず，実行状態のままになります。

そこで，  をクリックして実行を停止します。その状態で，シミュレータ GUI ウィンドウの信号データエディタを見ると 16 行目まで進んでいて，10 行目～15 行目のチャタリング部分では（立下り）エッジ検出していないことが確認できました。



これで，チャタリングへの対策ができていることが分かります。

１３. TAU の入力パルス間隔計測機能の利用（おまけ）

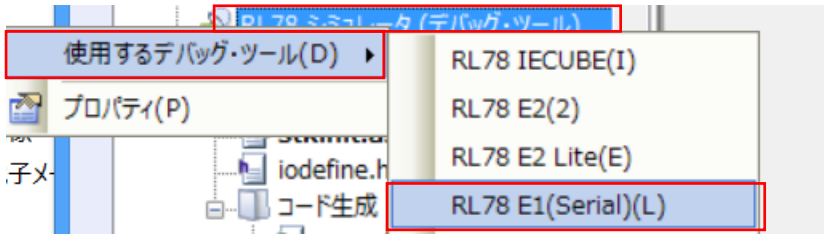
最後に，TAU のパルス間隔計測機能を用いたスイッチ入力のチャタリングとノイズ対策を考えてみます。

本当は，このチェックはシミュレータでやる予定でした。しかし，確認していく中で，タイマの動作がおかしいことが判明したので，シミュレータの使用をあきらめ，E1 を用いた実機での確認に切り替えました。

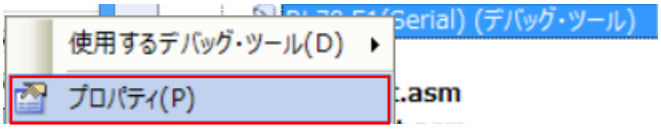
13.1 TAU の入力パルス間隔計測機能の使用準備

「RL78_G13_PORT3_2」フォルダをコピーして，「RL78_G13_PORT3_4_E1」に変更しておきます。そのフォルダの RL78_G13_PORT3_2.mtpj をダブル・クリックして，プロジェクトを開きます。

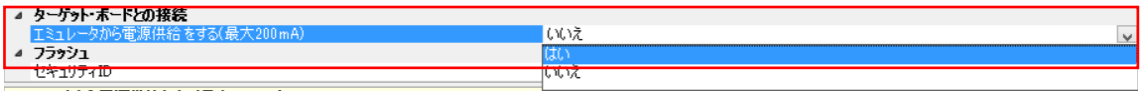
最初に，プロジェクト・ツリーの「RL78 シミュレータ（デバッグ・ツール）」を右クリックしてポップアップ・メニューから「使用するデバッグ・ツール（D）」を選択してツールの一覧を表示して，「RL78 E1 (Serial)(L)」を選択します。



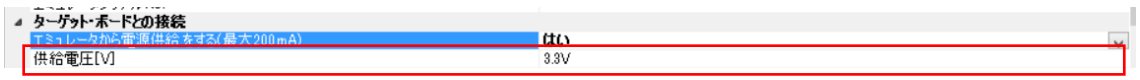
次に，「RL78 E1 (Serial)(デバッグ・ツール)」を右クリックし，ポップアップ・メニューから「プロパティ」を選択します。



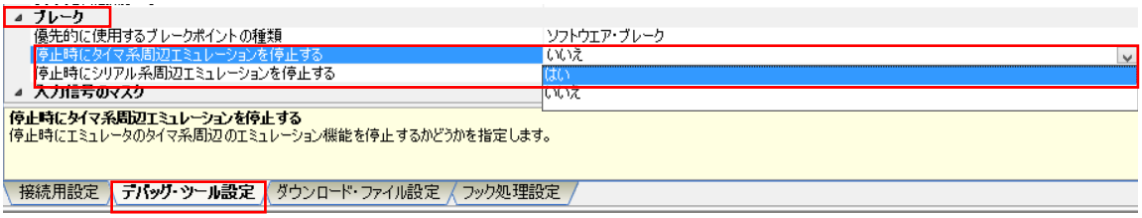
E1 のプロパティが表示されるので，「接続用設定」タグで，「ターゲット・ボードとの接続」の「エミュレータから電圧供給をする（最大 200mA）」を「はい」に変更します。



また，供給電圧は「3.3V」にしておきます。



次に「デバッグ・ツール設定」タグで ブレーク の をクリックして項目を表示します。表示された 2 項目目の「停止時にタイマ系周辺エミュレーションを停止する」を「はい」に変更します。



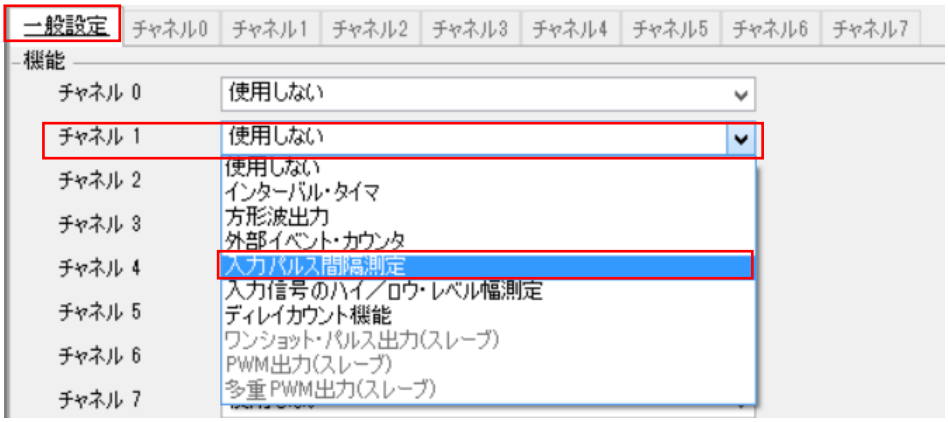
環境の E1 への変更はここまでです。

13.2 TAU の入力パルス間隔計測の設定

コード生成を使った，使用する周辺機能を設定していきます。

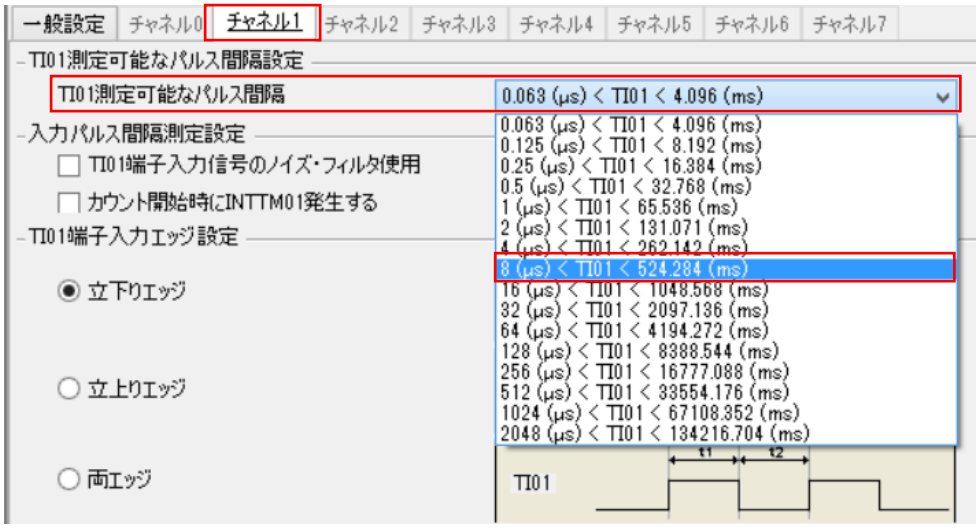
プロジェクト・ツリーの「コード生成（設計ツール）」の「ポート」で「ポート 1」を表示します。P16/TI01 端子にスイッチを接続するので，内蔵プルアップ抵抗を有効にしておきます。

次に「コード生成（設計ツール）」のタイマを選択し，「一般設定」で TM01（チャネル 1）を「入力パルス間隔測定」に設定します。

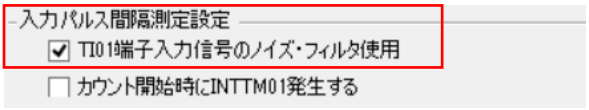


続けて，「チャネル 1」タグを選択します。そこで，一番上の「TI01 測定可能なパルス間隔設定」の「TI01 測定可能なパルス間隔」の項目で「(8μs)・・・・」辺りを選択します。これは 32MHz のシステム・クロック（f_{CLK}）を 128 分周した 250kHz を TM01 の動作クロックに設定するものです。動作クロックが 250kHz（4μs）なので，1 クロック未満の幅では検出できないことがあります。2 クロック間隔（8μs）以上の幅であれば**確実に検出可能**になることからこのような表記になっています。しかし，これは 8μs 未満のパルス幅は**確実に無視する訳ではない**ので注意が必要です。

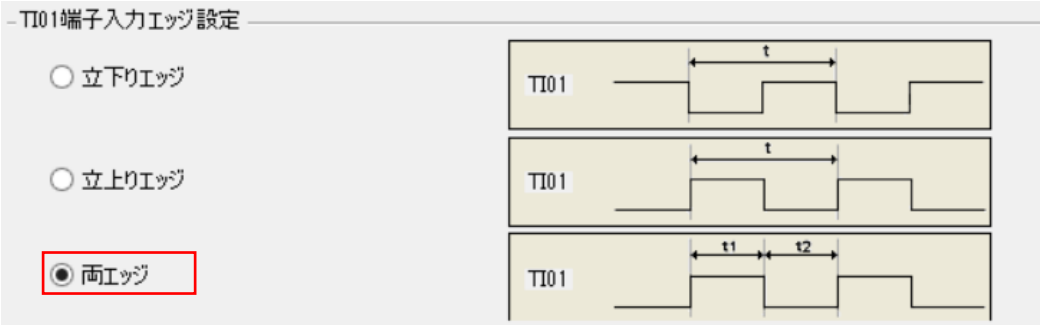
「ノイズ・フィルタ」を使用すると，最低 2 回のサンプリングで一致しないとノイズとして無視されるので，4μs 以下のノイズは確実に無視されます。



「入力パルス間隔測定設定」の「TI01 端子入力信号のノイズ・フィルタ使用」をチェックして，ノイズ・フィルタを ON します。



「TI01 端子入力エッジ設定」は「両エッジ」を選択します。



これで，設定完了なので，コード生成(G) をクリックしてコードを生成します。

13.3 プログラムの変更

次は，プログラムの作成ですが，r_cg_timer_user.c の INTTM01 割り込みに処理に対応した r_tau0_channel1_interrupt 関数を修正します。

タイマ割り込みの中では，入力パルス間隔測定の割り込み処理関数中にコードが生成されています。今回の使い方では生成されたコードを使った方が手間はかかりません。

生成されたコードを示します。キャプチャした値に＋1 したものが実際の間隔になります。

さらに，TSR01 レジスタのビット 0 が 16 ビットからのオーバーフローで 0x10000 を加算していますが，これは問題だと思います。なぜなら，オーバーフローは 1 回とは限らないからです（スイッチでは 1 秒以上はあり得ます）。

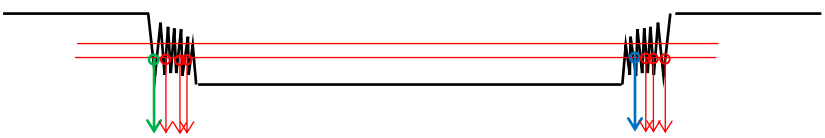
```
static void __near r_tau0_channel1_interrupt(void)
{
    if ((TSR01 & _0001_TAU_OVERFLOW_OCCURS) == 1U)     /* overflow occurs */
    {
        g_tau0_ch1_width = (uint32_t)(TDR01 + 1UL) + 0x10000UL;
    }
    else
    {
        g_tau0_ch1_width = (uint32_t)(TDR01 + 1UL);
    }

    /* Start user code. Do not edit comment generated here */
    /* End user code. Do not edit comment generated here */
}
```

しかし，高々16 ビットからのオーバーフローだけのために 32 ビットの変数を定義して 32 ビットでの演算を行うのは，好みに合いません。そこで，サイズを小さくするために，今回は測定結果を 16 ビットの値とオーバーフローに分けて処理します。

この場合に注意しないといけないのは，上の赤く囲んだ部分以外の部分は，コード生成を行うたびに再生されるので，その都度削除する必要があります。逆にコード生成時に削除されないようにするには，赤く囲んだ部分にコードを記述しておく必要があります。

まず，制御の考え方を示します。チャタリングは，下の図に示すように，スイッチを押したときと放したときの両方で発生します。押した時の最初の立下りエッジ（グリーンで示す）ではキャプチャ値が大きくなるので，正規の立下りエッジと判断します。その後の立下りエッジ（赤で示す）はキャプチャした値が小さいのでチャタリングと判断して無視します。この方法では，最初のエッジで検出できるメリットもあります。さて，スイッチを放したときの信号を見ると，最初の立下りエッジ（ブルーで示す）ではキャプチャした値も大きくなります。これでは，どちらで発生した立下りエッジで発生したものかが判断できません。

そこで，両者を区別するに，その前の状態が「スイッチが押された状態（ロウ・レベル）」か「スイッチが押されていない（ハイ・レベル）」かで判断します。



この方法では，最初にキャプチャ値を確認して，チャタリング部分を無視します。チャタリング部分でなければ，以前の状態を反転させ，今回の状態とします。これにより押されたことと放

されたことが分かります。それではなぜ，検出エッジが立下りではなく，両エッジになっているのでしょうか。

それは，ノイズ除去の影響により，立ち上がりでチャタリングが無視されてしまった場合への対策のためです。

検出エッジを両エッジに固定にするのではなく，ダイナミックに切り替える方法も考えられます（RL78/G13 のハードウェア・マニュアルには，入力パルス間隔測定動作中に検出エッジを指定する CIS ビットを変更できることが記述されています）。この方法が，スマートに思えますが，割り込み要求が 1 本しか使えないので，殆ど変わらないと考えられるので今回は両エッジに固定しておきます。

寄り道からプログラムに戻ります。割り込み処理では最初にオーバーフロー・フラグとキャプチャ値をチェックします。これが，ノイズ・フィルタを通過した信号からのチャタリング防止になります。カウント・クロックが 4μs 周期なので，キャプチャ値を 2500 に設定すると 10ms 以下の振動はチャタリングとして無視されるようになります（通常，チャタリングが数十 ms と言うのは振動が収まるまでの時間で，個々の振動は接点部分の大きさや材質で変化しますが，ms 以下と考えていいでしょう）。

使用する変数と定数は以下のようにしておきます。

```
/* Start user code for global. Do not edit comment generated here */
volatile    uint8_t g_port = 0xFF;          /* ポートの前回の値          */
volatile    uint8_t g_edge = 0x00;          /* スイッチ押下フラグ        */
#define     FILTER  ( 2500 )                /* チャタリング用パラメータ  */
/* End user code. Do not edit comment generated here */
```

その上で，割り込み処理は，コード生成されていたプログラムは削除し，下の赤枠内に示すようにユーザコード部にプログラムを記述します。

```
static void __near r_tau0_channel1_interrupt(void)
{
    /* Start user code. Do not edit comment generated here */
    if ( ( ( TSR01 & _0001_TAU_OVERFLOW_OCCURS ) == 1 ) || ( TDR01 > FILTER ) )
    {
        if ( g_port != 0x00 )
        {
            g_edge = 0x01;
            g_port = 0x00;
        }
        else
        {
            g_port = 0xFF;
        }
    }
    /* End user code. Do not edit comment generated here */
}
```

割り込み処理部は以上です。最後に，r_main.c の R_MAIN_userInit 関数の処理部分を書き換えます。

```
void R_MAIN_UserInit(void)↓
{↓
    /* Start user code. Do not edit comment generated here */↓
    R_TAU0_Channel1_Start();          /* 入力パルス間隔測定起動    */↓
    if ( ( P1 & 0b01000000 ) == 0 )↓
    {↓
        g_port = 0x00;↓
    }↓
    else↓
    {↓
        g_port = 0xFF;↓
    }↓
    EI();↓
    /* End user code. Do not edit comment generated here */↓
```

上側で赤く囲んだように，最初に TM01 を起動します。その後，P16 の状態を読み出して，入力の初期状態とします。これは，このプログラムがエッジ検出処理で入力の状態を反転する処理なので，変数 g_port の初期状態を設定しておく必要があるためです。

13.4 デバッグのための準備

RL78 を使ったプログラムの通常のデバッグでは，変数や内蔵周辺機能を制御するレジスタ（SFR）の内容を確認したり，変更したりしながら進めていきます。デバッグのために p72 で定義したのと同じ変数をここでも r_main.c のグローバル変数として，定義しておきます。

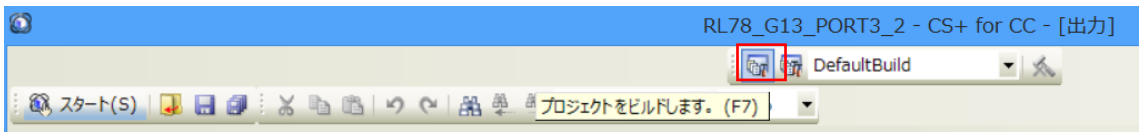
R_MAIN_User_Init 関数で参照した変数 g_port も extern 宣言しておきます。

```
/* Start user code for global. Do not edit comment generated here */↓
extern  volatile    uint8_t g_port;↓
extern  volatile    uint8_t g_edge;↓
volatile    uint8_t g_count = 0;↓
/* End user code. Do not edit comment generated here */↓
```

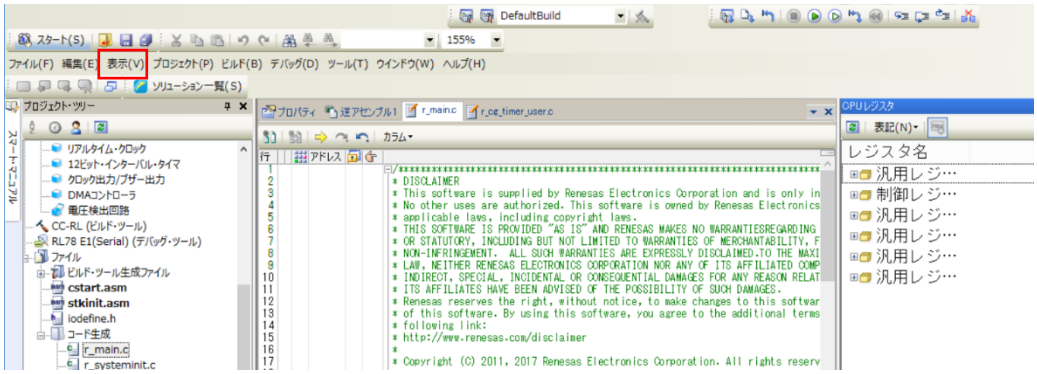
変数 g_count の使い方も p72 と同じで，立下りエッジ検出で回数をカウントします。

```
if ( g_edge )↓
{                               /* スイッチ押下フラグが1なら      */↓
    P3 ^= 0x02;                 /* P3.1(LED)を反転する            */↓
    g_edge = 0x00;              /* 押下フラグをクリアする         */↓
    g_count++;↓
}↓
```

プログラムを保存したら，下に示す  をクリックしてビルドして，E1 を経由して RL78/G13 にダウンロードします。

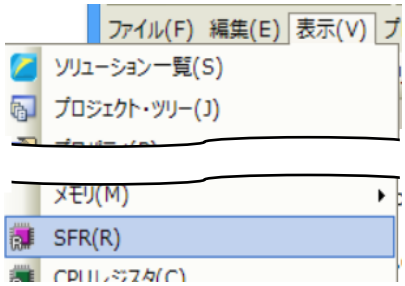


以下のエミュレータ画面が表示されるので，デバッグのために 表示(V) をクリックします。

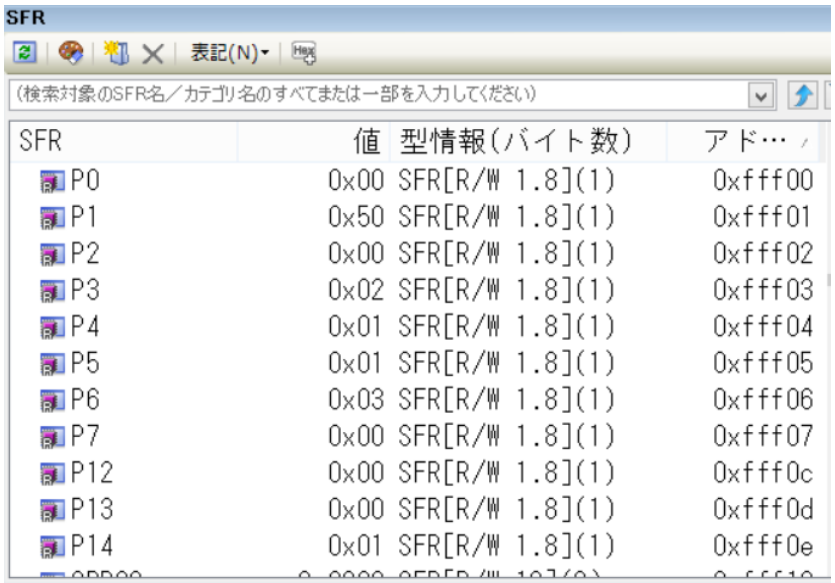


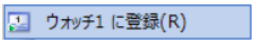
右に示すプルダウンメニューが表示されるので，SFR(R)をクリックします。

とりあえずは，ここで，表示させるのは SFR だけにしておきます。

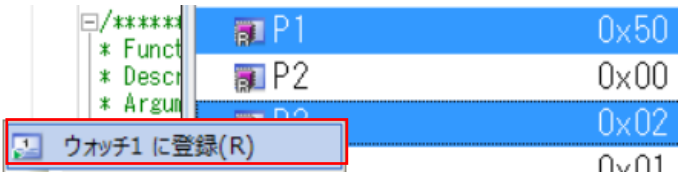


表示された SFR パネルには，SFR 名称（最初は P0 からアドレス順），値，型情報（バイト数），アドレスの順で情報が表示されています。SFR に関してこれらの情報を全て表示するのは画面スペースが無駄なので，通常は SFR とその値だけで十分でしょう。

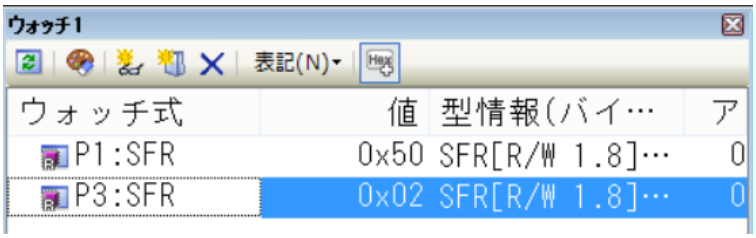


今回のプログラムでは，P16（追加したスイッチ）と P31（LED）だけが確認する対象なので，P1 と P3 を選択（P1 をクリックし，CTRL を押した状態で P3 をクリック）し，P1 と P3 を反転表示させ，右クリックしてメニューを表示し 

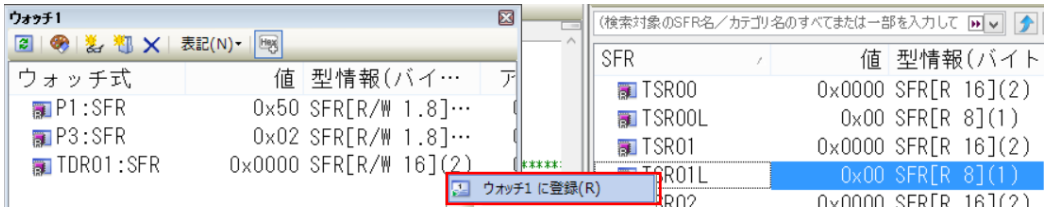
をクリックします。



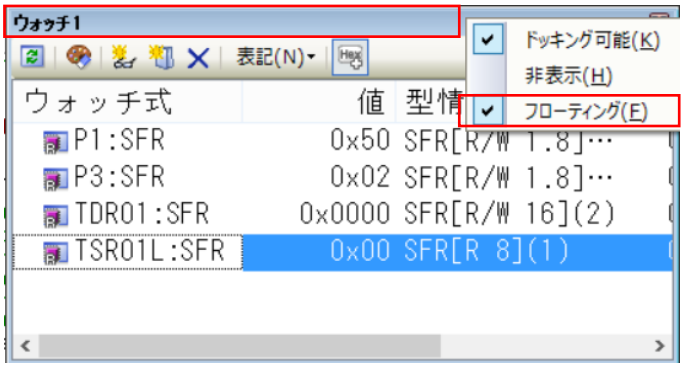
これで，ウォッチ 1 パネルに P1 と P3 が表示されます。



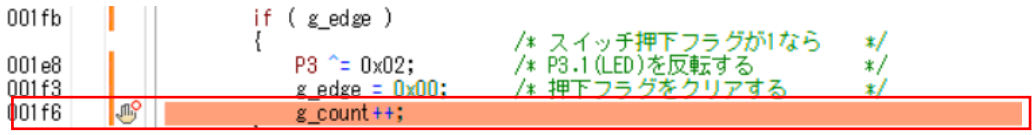
次に TM01 関係のレジスタを確認するのですが，アドレス順では分かりにくいので SFR をクリックして SFR の名前で並び替えておきます（名前で検索を掛けてもいいのですが）。TDR01 を右クリックして，メニューから ウォッチ1 に登録(R) をクリックします。



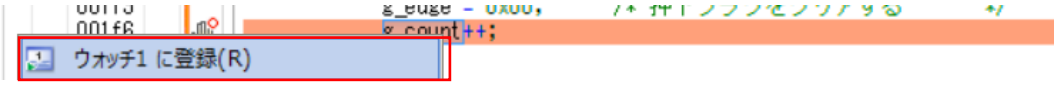
同様に，TSR01L レジスタもウォッチ 1 に登録します。上の画面イメージはウォッチ 1 パネルをフローティング表示して SFR パネルの横に並べた状態でキャプチャしたものです。下に示すように，ウォッチ1 で右クリックしてメニューを表示すると，フローティングになっているのが分かります。そこのチェックを外すと元の位置に戻ります。



デバッグの準備ができたので，先ほど追加した行にブレークポイントを設定します。

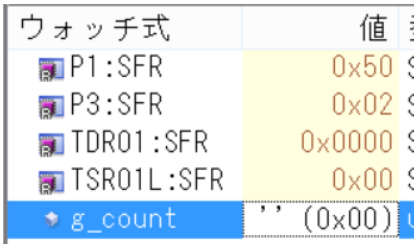


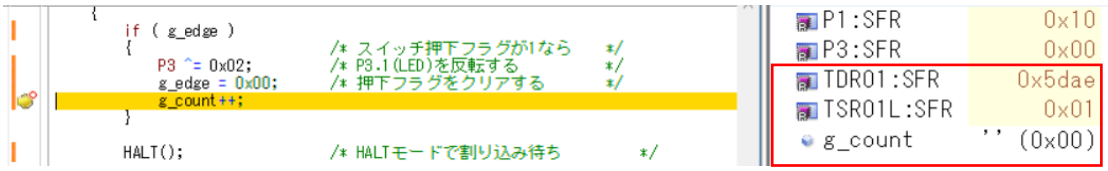
さらに，変数 g_count をウォッチ 1 に登録します。



これで，  をクリックして，ブレークポイントを有効にして実行開始します。

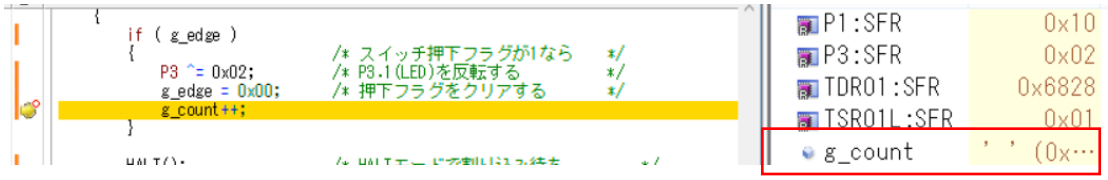
そのままでは実行を継続するだけなので，追加したスイッチを押すことで，ブレークが掛かるります。次ページに示すように，ウォッチ 1 で TDR01 と TSR01L を見ると，0x5DAE と 0x01（オーバフロー発生）になっています。





変数 g_count をチェックすると，0x00 のままです。ブレークは実行前ブレークなので，こうなっています。

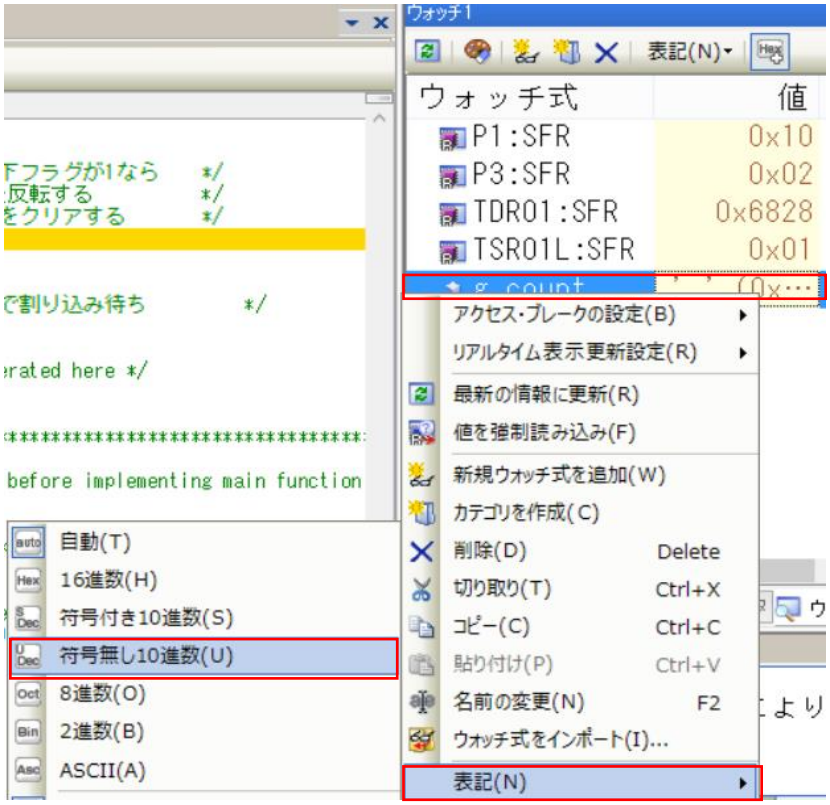
スイッチを押したままの状態で，  をクリックして，実行を再開します。その状態で，スイッチを放してもブレークはかからないことを確認します。再度スイッチを押すと，ブレークして以下の状態になります。



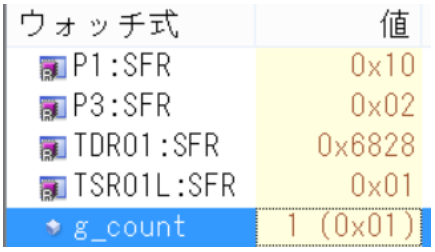
これでは変数 g_count の値が分からないので，表記方法を変更します。g_count を右クリックして，メニューを表示し， 

をクリックし，さらに， 

をクリックして表示を符号なし 10 進数に変更します。

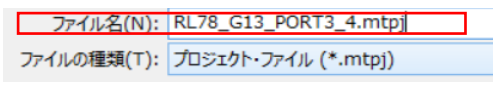


その結果を右に示します。変数 g_count の値は 0x01 になっていることが分かります。



確認が完了したら，  をクリックしてデバッガを終了し，プロジェクトの名前を変更して保存します。

ここでは，”RL78_G13_PORT3_4”にしておきます。



これで，CS+を終了します。

今回は，ポート編の最後として，タイマの機能を使った例を示しましたが，次回からはタイマが主になります。そのためには，最初にハードウェアの拡張を行います。

ポートのちょっと高度な使い方については，別途説明する予定です。

以上