

初心者のための RL78 入門コース（第 2 回：ポート出力例その 1）

第 2 回の今回は実際に簡単なプログラムを作成して動作確認を行います。最初は、シミュレータで動作させます。ついでに、E1 を使って BlueBoard-RL78/G13_64pin で動作確認を行います。

今回の内容

- 6. コード生成を利用した実際のプログラム作成 P19
- 7. BlueBoard-RL78/G13_64pin での動作確認 P34

次回（第 3 回）は、以下の内容を予定しています

- 8. コード生成を利用した実際のプログラム作成（その 2） P40
- 9. コード生成を利用したプログラム作成（ポート入力） P47

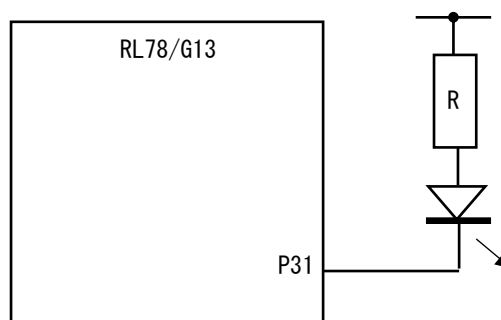
ここまでで使用したプロジェクトは以下のフォルダに格納されています。

| | |
|------------------------------------|-------------------------------|
| 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）↓ |
| ↓ | |

6. コード生成を利用した実際のプログラム作成

プログラムの例として、最初に環境に慣れることもかねて、LED の点滅をやってみようと思います。まずは、P31 に接続した LED を点灯させることから始めます。最初に少しハードウェアについて触れます。

ここで制御対象の LED は右図に示すように接続されているものとします。このように接続するのは、ポートのドライブ能力はロウ側がハイ側に比べて大きいからです。ここで、R は LED に流れる電流を制限するための電流制限抵抗です。この回路構成では、P31 に 0 を設定すれば LED は点灯し、1 を設定すれば消灯します。



ポートで LED 等の負荷をドライブするとき、負荷を動作させるための電流を流すことができれば十分です。ロウでドライブしてもハイでドライブしても LED を目標の明るさで点灯できれば構いません。

RL78/G13 の通常のポートではハイ・レベルで流せる電流は最大 10mA で、ロウ・レベルでは 20mA です。

LED の定格が 20mA 程度と書かれていることがありますが、実際にそんな電流を流すとまぶしくてたまりません。実際はその半分程度でも十分な明るさになることがほとんどです。効率が高い（高輝度タイプ）LED では 1mA でも分かります。使用する LED と目的に合った電流にします。

通常、LED の順方向電圧 (Vf) は 1.5V (赤色 LED) から 3V 程度 (白色系) まであります。そこで、LED をドライブするための電源の電圧から LED の順方向電圧とポートでの電圧降下分を引いた電圧と流したい電流から電流制限抵抗の値を決めます。

RL78/G13 のロウ・レベル出力電圧のスペックを見ると電源電圧 4V 以上で 20mA 流すと 1.3V (MAX) と記載されています。ところが HP で IOL と VOL の特性曲線を見ると 20mA 流しても電源電圧が 5V では 0.4V 程度で、電源電圧 4V 辺りの電圧降下を求めると 0.5V 強となり、スペックの半分もありません。電源電圧が 5V のときに真面目にスペックで計算した抵抗値を使うと、1.5 倍も電流が流れることとなります。

電流の流し過ぎに注意しましょう。

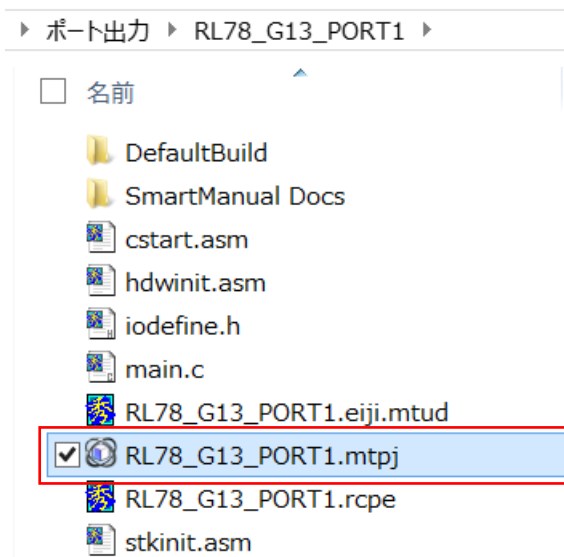
注：RL78/G13 の特性曲線は以下の URL にあります。今回はこの中の RL78/G13:VOL-IOL 特性を参照しました。

<https://www.renesas.com/ja-jp/search/keyword-search.html#genre=document&documenttype=544&productlayer=115082>

ハードウェアについては別途説明する予定です。

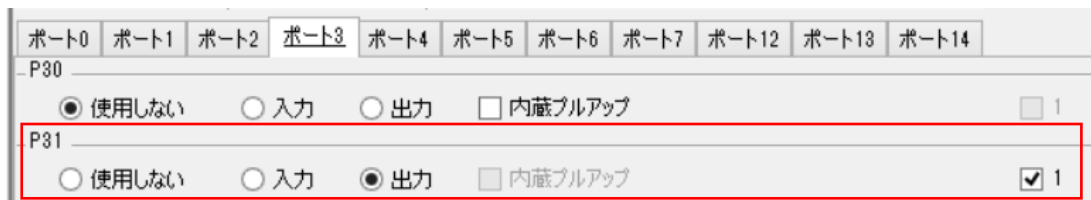
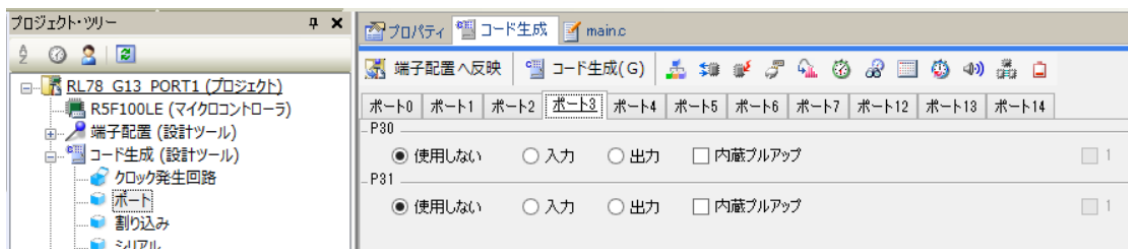
6.1 プロジェクト (RL78_G13_PORT1) を開きます。

「RL78_G13_PORT1」フォルダを開き、「RL78_G13_PORT1..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。



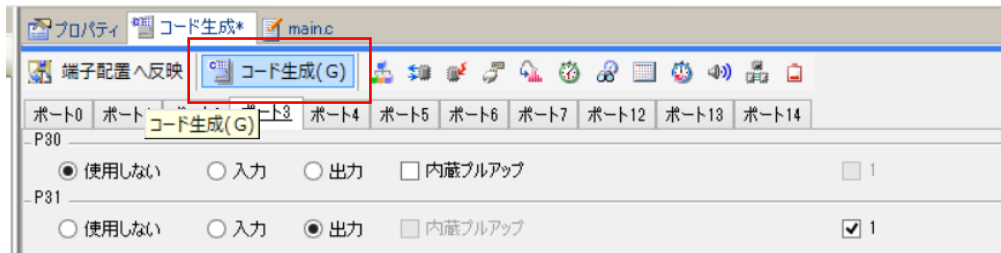
6.2 コード生成でポート 3 を開きます。

初期状態では、全て「使用しない」になっているので、下に示すように P31 を「出力」に変更し、右側の「1」にチェックを入れます。



これは、P31 を出力ポートに設定し、出力するデータの初期値を 1 (初期状態で、LED を消灯) にすることを示します。

今回のコード生成する内容はこれだけです。これで、「コード生成 (G)」をクリックします。



すると、右下の出力ウィンドウに、結果が表示されます。

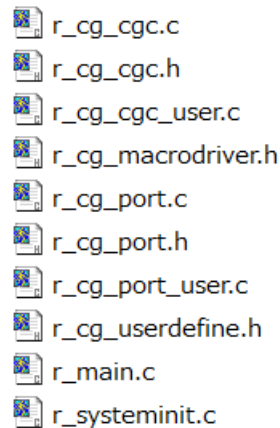
```

M0409001: ファイルを生成します。↓
M0409000: r_main.c を生成しました。↓
M0409000: r_systeminit.c を生成しました。↓
M0409000: r_cg_macrodriver.h を生成しました。↓
M0409000: r_cg_userdefine.h を生成しました。↓
M0409000: r_cg_cgc.c を生成しました。↓
M0409000: r_cg_cgc_user.c を生成しました。↓
M0409000: r_cg_cgc.h を生成しました。↓
M0409000: r_cg_port.c を生成しました。↓
M0409000: r_cg_port_user.c を生成しました。↓
M0409000: r_cg_port.h を生成しました。↓
M0409003: ファイルの生成を完了しました。↓
M0409009: ウォッチドッグ・タイマの設定は'使用しない'です。↓
M0409012: 電圧検出回路の設定は'使用する'です。↓
M0409014: オンチップ・デバッグ機能の設定は'使用する'です。↓
  
```

これで、コード生成が関係するファイルを生成しました。

右に示したのがコード生成で生成したファイルです。

- ・「r_cg_cgc」で始まるのはクロックの設定です。
- ・「r_cg_port」で始まるのはポートの設定です。
- ・「r_main.c」が main 関数を含むファイルです。
- ・「r_systeminit.c」はスタートアップ・ルーチンから呼び出される hdwinit 関数を含んでおり、内蔵周辺機能の初期化を取りまとめています。



6.3 生成された設定の内容

(1) ポートの初期設定

今回は P31 の設定だけなので、ポートの初期設定は以下のように簡単な R_PORT_Create 関数が生成されています。

```

void R_PORT_Create(void)
{
    P3 = _02_Pn1_OUTPUT_1;
    PM3 = _01_PMn0_NOT_USE | _00_PMn1_MODE_OUTPUT | _FC_PM3_DEFAULT;
}
  
```

P31 を出力ポートとして使うための設定は P3 (出カラッチ) レジスタと PM3 (ポート・モード) レジスタだけです。今回使用する 64 ピンの R5F100LE ではポート 3 は P30 と P31 の 2 本しかありません。P3 レジスタに 2 を設定しているのは P31 の初期値を 1 と指定したことに対応しています。PM3 レジスタのビット 1 が 0 になっているのが、P31 を出力ポートに設定したことに対応しています。

(2) r_main.c

このファイルには、main 関数と、R_MAIN_UserInit 関数の (実際には入れ物だけが) 2 つだけです。

R_MAIN_UserInit 関数は初期設定が済んだ内蔵周辺機能を実際に起動 (動作可能に) するために準備された関数で、main 関数は最初にこの R_MAIN_UserInit 関数を呼び出します。実際に生成された R_MAIN_UserInit 関数は以下のようになっています。

ここで、コード生成を使用する際の基本的なルールを示します。コード生成された関数には下で朱書きで表示した 2 つのコメント行があり、プログラムを書く場合には必ずこの間に書いてください。もちろん、この範囲以外にも書くことは可能ですが、この範囲外に記述したプログラムは次にコード生成したときに消されてしまいます。これは、すべてのコード生成で生成されたプログラムやヘッダファイルで共通です。

```
void R_MAIN_UserInit(void)
{
    /* Start user code. Do not edit comment generated here */
    EI();
    /* End user code. Do not edit comment generated here */
}
```

コード生成で生成された main 関数は、以下のようになっています。基本的には while 文の中にプログラムを記述します。

```
void main(void)
{
    R_MAIN_UserInit();
    /* Start user code. Do not edit comment generated here */
    while (1U)
    {
        ;
    }
    /* End user code. Do not edit comment generated here */
}
```

6.4 その他の項目

生成された R_MAIN_UserInit 関数の中に "EI();" という記述がありますが、これは RL78 独自の機能拡張です。

これは、CPU を割り込み（ベクタ割り込み）制御するものです。これ含め、同じような機能拡張として覚えておいて欲しいのが以下の 5 つです。これら 5 つは、組み込み用途向けに追加されたものです。CC-RL で RL78 のプログラムを作成するのに必須だと考えてください。

- ・ **NOP();** RL78 の NOP 命令を指定します。
- ・ **EI();** 割り込み許可（RL78 の EI 命令と同じ）を指定します。
- ・ **DI();** ”EI();”と逆に割り込みの禁止（RL78 の DI 命令と同じ）を指定します。
- ・ **HALT();** CPU を停止させる RL78 の HALT 命令を指定します。
- ・ **STOP();** RL78 のクロック発振を停止させる STOP 命令を指定します。

6.5 ポートの制御プログラム

いよいよ、プログラムを書いてみます。

P31 に接続した LED を点灯させるプログラムです。LED を点灯させるには、P31 を 0 にします。このためには、P3_bit.no1 に 0 を代入します（逆に、LED を消灯させるには、P3_bit.no1 に 1 を代入します。）

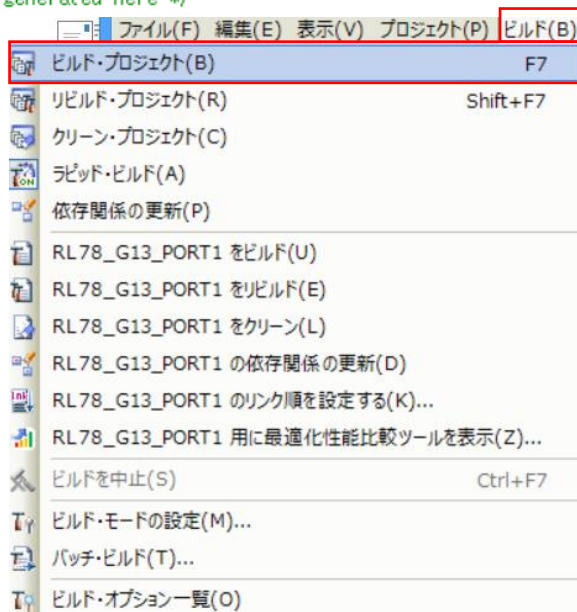
これを、実際に main 関数に書き込みます。下に実際に書き込んだプログラムを示します。

```
void main(void)
{
    R_MAIN_UserInit();
    /* Start user code. Do not edit comment generated here */
    while (1U)
    {
        P3_bit.no1 = 0;           /* P3.1を0（ロウ出力）にする  */
        NOP();
        P3_bit.no1 = 1;           /* P3.1を1（ハイ出力）にする  */
        NOP();
    }
    /* End user code. Do not edit comment generated here */
}
```

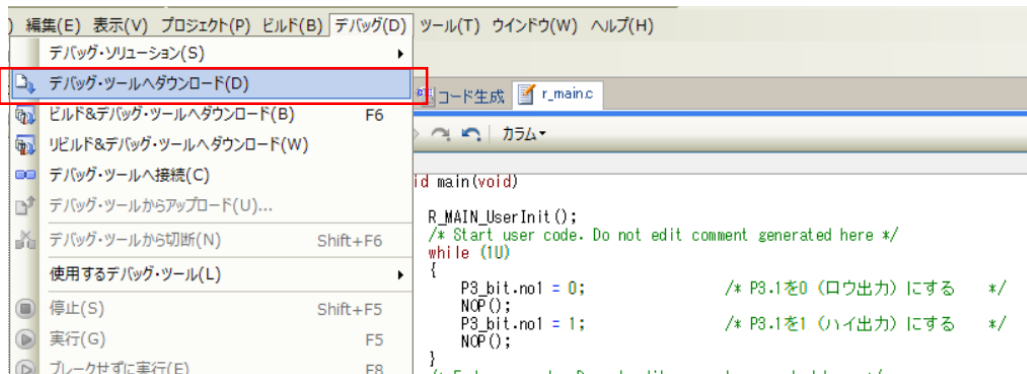
これをビルドしてみます。

ビルドは、右に示すように、メニューバーの「ビルド (B)」をクリックしてメニューを表示させ、「ビルド・プロジェクト (B)」を選択します。

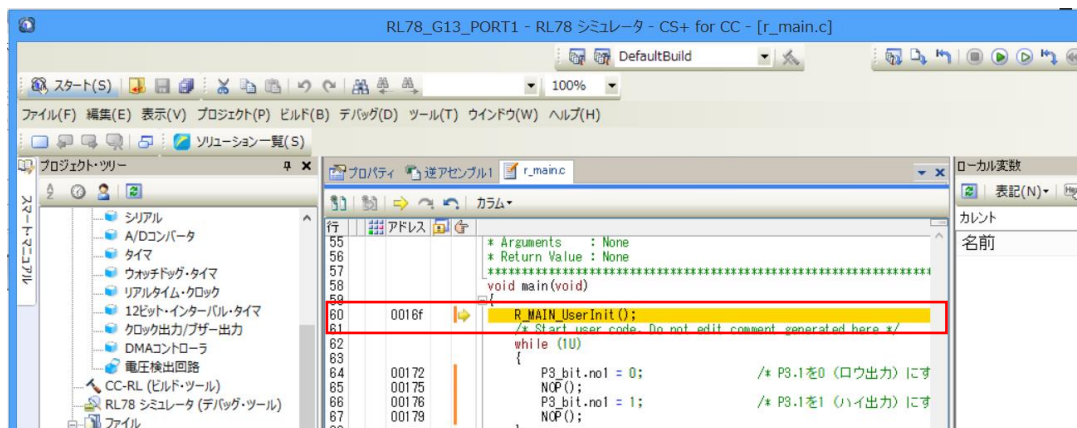
ビルド結果は出力ウィンドウで確認します。




結果を確認するために、デバッグ・ツールに結果をダウンロードします。下に示すようにメニューバーの「デバッグ (D)」をクリックしてメニューを開き、「デバッグ・ツールへダウンロード (D)」を選択してください。

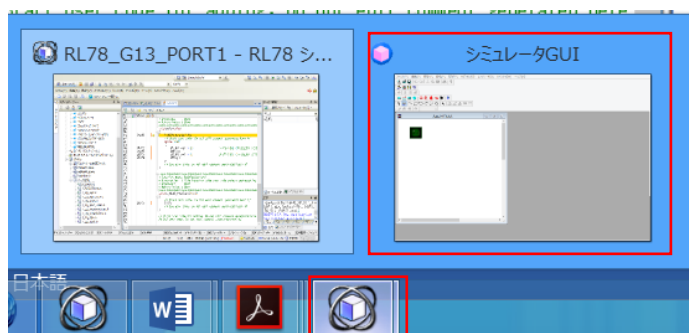


ダウンロードが完了すると、シミュレータの画面となり、下に示すようにダウンロードしたプログラムの main 関数のソースが表示されます。



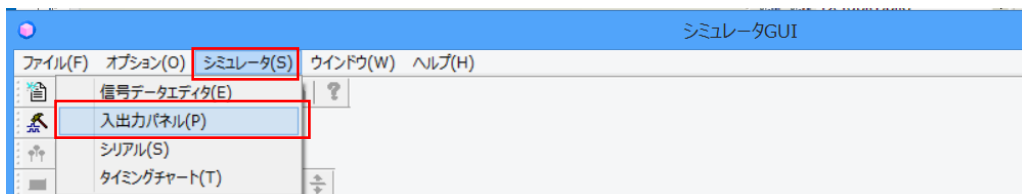
60 行目に黄色の右向きの矢印があり、黄色になったこの行が次に実行する行で、main 関数の最初で、関数 R_MAIN_UserInit を呼び出しているところになります。

ここで、プログラムを実行する前に、シミュレータの設定を行います。このためにはタスクバーにある CS+のタスクを示す  をクリックすると、下のように2つのタスクが表示されるので、右側のシミュレータを選択します。

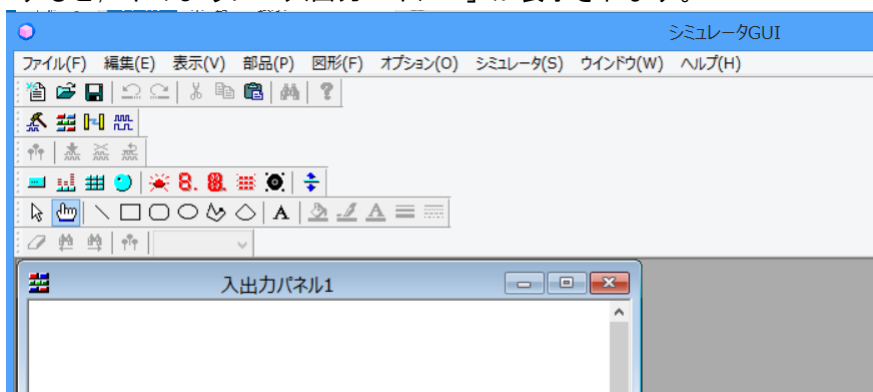


RL78/G13のシミュレータは命令だけでなく、内蔵された周辺機能の一部と外部素子もシミュレートできます。最初にその設定を行います。

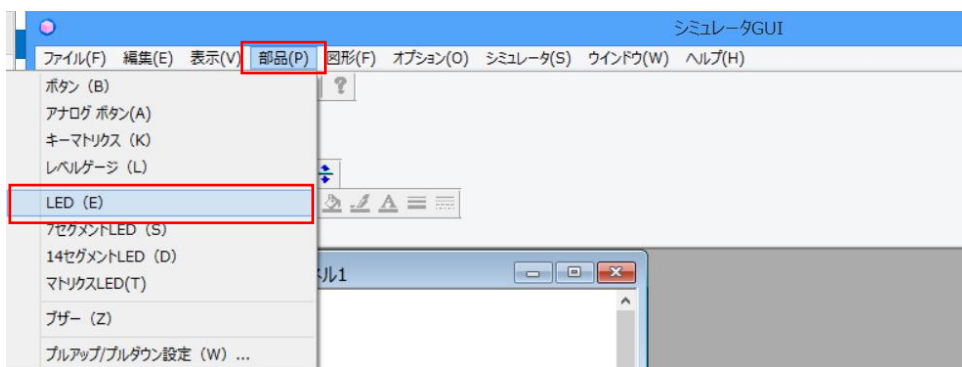
シミュレータ GUIが表示されたら、下に示すようにメニューバーで「シミュレータ (S)」をクリックしてメニューを表示して「入出力パネル (P)」を選択します。



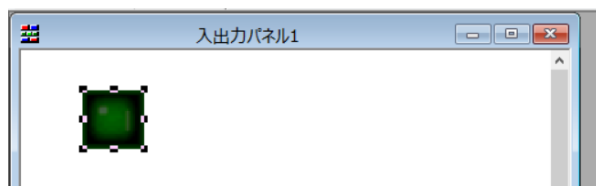
すると、下のように「入出力パネル 1」が表示されます。



次に、メニューバーの「部品 (P)」をクリックして表示されたメニューから「LED (E)」を選択し、「入出力パネル 1」で LED を配置する場所をドラッグします。



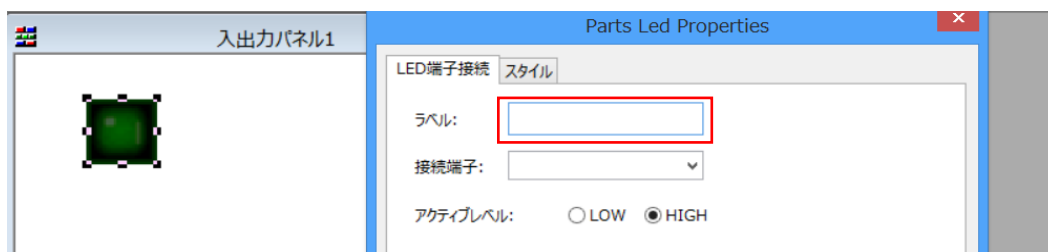
ドラッグした範囲で LED を示す部品が表示されます。



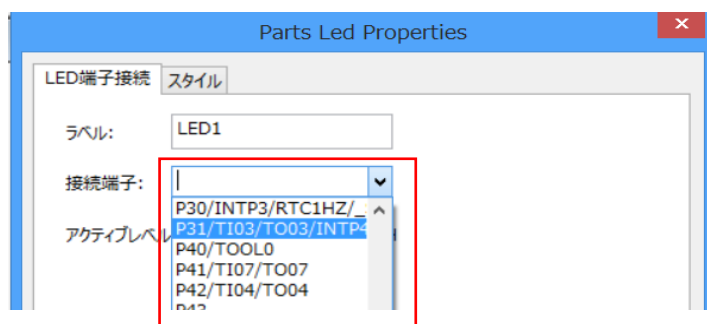
この部品を右クリックしてメニューを表示します。表示されたメニューの一番下にある「プロパティ (R)」を選択します。



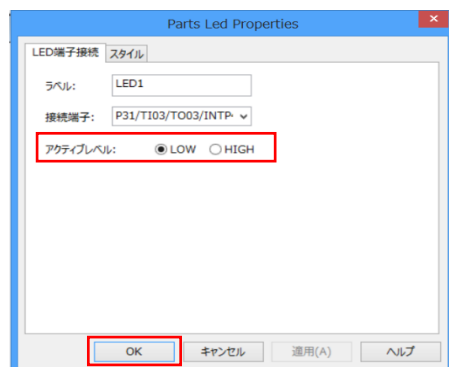
下に示すようなプロパティ画面が表示されるので、「ラベル」に名前を入力します。



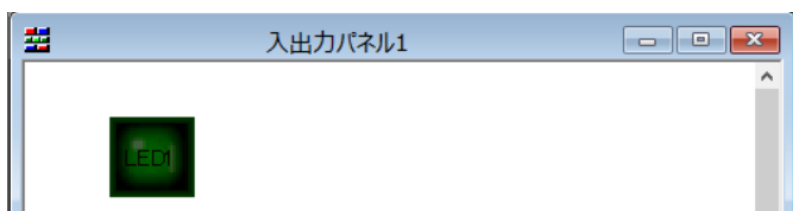
ここでは、「LED1」とでもしておきます。次に、「接続端子」の欄の右側のメニューをクリックしてリストを表示し、接続する端子として「P31/.....」を指定します。



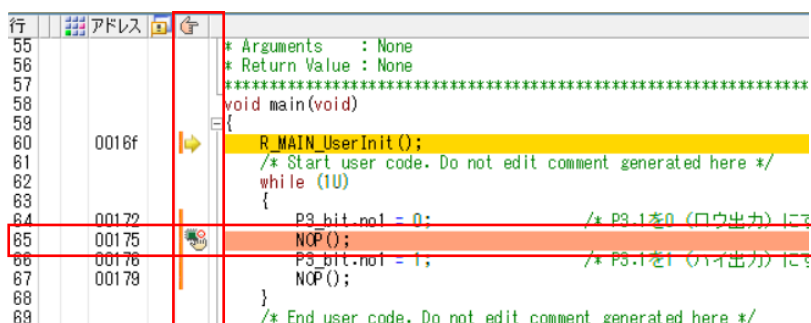
最後に、「アクティブレベル」を「LOW」にして、「OK」をクリックします。




これで、設定が完了し、下のように部品の周りの枠がなくなります。



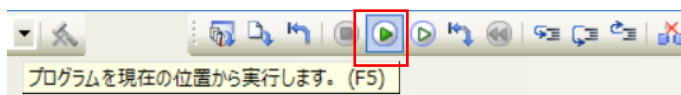
シミュレータ GUI での設定が完了したので、プログラムに戻ります。そこで、下の図の 65 行目の 2 つの赤い枠で囲まれた部分の交わったところをクリックすると、赤っぽい色が NOP(); の行につきます。これはブレーク・ポイントが設定されたことを示し、そこでプログラムの実行を停止させることができます。プログラムに「NOP();」の行を入れたのは、このようにブレーク・ポイントを設定するためです。



同様に、下側の「NOP();」にもブレーク・ポイントを設定します。

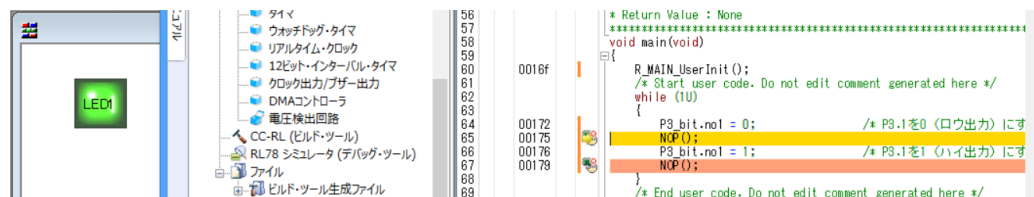
シミュレータ画面の右上に  があります。これがシミュレータのコマンドボタンです。このボタンを使ってプログラムの実行を制御します。

グリーン右向きの三角にマウスを合わせると、説明が表示されます。このボタンをクリックすることで、ブレーク・ポイントを有効にして、プログラムが実行されます。

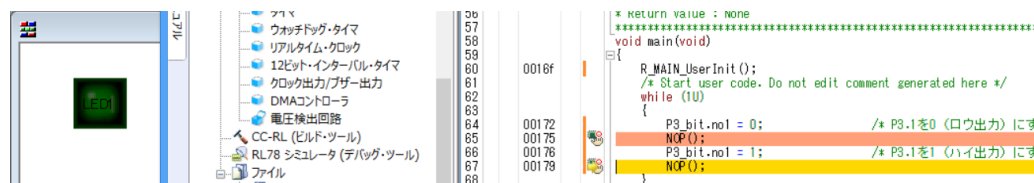


このボタンをクリックして、プログラムを実行させます。

この実行結果が、下に示す画面イメージとなります。ここでは、シミュレータ GUI が右側に見えるようにウィンドウを配置しているの、実行結果とプログラムを同時に見ることができます。左側の GUI 画面で LED が点灯しているのが分かるかと思いますが（LED1 の文字がはっきりと見えています）。プログラムウィンドウでは、上側の「NOP();」の行が黄色になり、そこで実行が停止していることがわかります。つまり、その上の「P3_bit.no1 = 0;」が実行されて LED が点灯したことになります。



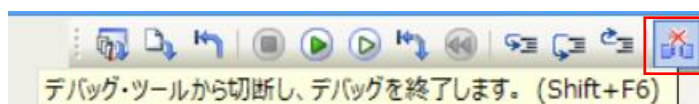
再度、グリーン右向きの三角のボタンをクリックすると、下のように、下側の「NOP();」の行が黄色になり、その上の「P3_bit.no1 = 1;」が実行された結果 LED が消灯したことになります。



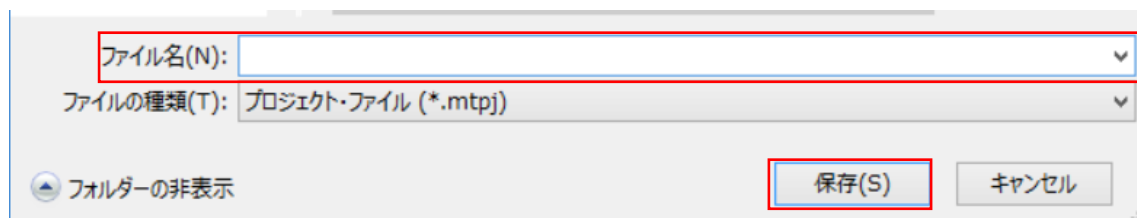
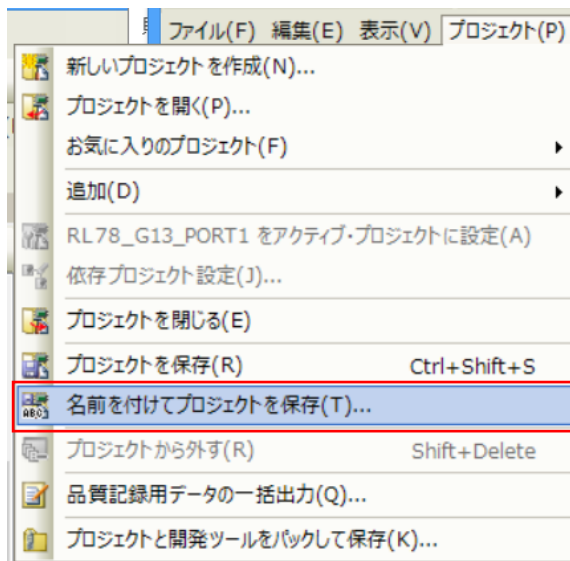
それでは、グリーンで白抜き右向きの三角のボタンをクリックしてみましょう。これはブレーク・ポイントを無効にして実行するボタンです。

すると、シミュレータ GUI ウィンドウが前面に表示され、LED がチカチカ点滅しているのがわかります。これは、決して LED チカチカの状態ではなく、単にシミュレータの実行速度が遅いためにこのように目に見える点滅が発生しているだけです。実際の RL78/G13 で実行すると、点滅は全く目に見えませんが、次は目に見えるように対策します。一応ここまでのプロジェクトを保存しておきます。

プログラムを停止させるには、シミュレータのウィンドウを表示させ、右上の赤い四角が表示されたボタンをクリックします。



シミュレータを終了して、CS+のウィンドウに戻ったら、メニューバーの「プロジェクト (P)」のメニューから「名前を付けてプロジェクトを保存 (T)」を選択し、ファイル名を指定して「保存 (S)」をクリックして保存します。



CS+をクローズします。ここまでの結果を保存しておくために、「RL78_G13_PORT1」フォルダをコピーして、「RL78_G13_PORT1_2」の名前を付けておきます。次は、この「RL78_G13_PORT1_2」フォルダの中のプロジェクト「RL78_G13_PORT1_1.mtpj」を使っていきます。

6.6 LED の点滅プログラム

それでは、目に見える周期で LED を点滅させます。このために最初に考え付くのはソフトウェアでループを作ることです。単純に符号なしの 16 ビットの変数をカウントすることを考えてみます。この場合、1 ループで 10 クロック程度と考えると、32MHz の動作では、約 20ms 程度になります。残念ながらこれでは目に見えません。そこで、32 ビットの変数を使用することになります。RL78 は 16 ビット MCU なので、32 ビットの演算は苦手です。1 ループで倍以上のクロックが必要になります。ここでは、20 クロックと考えると 200ms の時間待ちを考えてみます。200ms のクロック数は 6,400,000 (=32,000,000×0.2) となり、1 ループを 20 クロックと考えると 320,000 回ループさせることになります。これでプログラムを追加してみます。

r_main.c の最後のユーザ領域に次ページに示すような wait_200 関数を作成します。

ここで、32ビットの変数 time を定義し、for ループで 320,000 から減算していき、0 になったら抜けるようにします。

```

/* Start user code for adding. Do not edit comment generated here */↓
/*****
* Function Name: wait_200↓
* Description  : 200ms待つ↓
* Arguments   : None↓
* Return Value : None↓
*****/
void wait_200(void)↓
{↓
    uint32_t time;↓
    for ( time = 320000 ; time > 0 ; time-- )↓
    {↓
        NOP();↓
    }↓
}↓
/* End user code. Do not edit comment generated here */↓
[EOF]

```

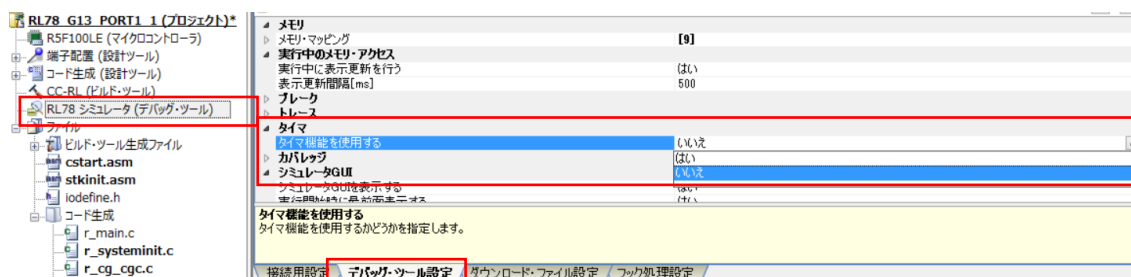
これを呼び出す処理を、main 関数のポートを操作している処理の間に入れます。

```

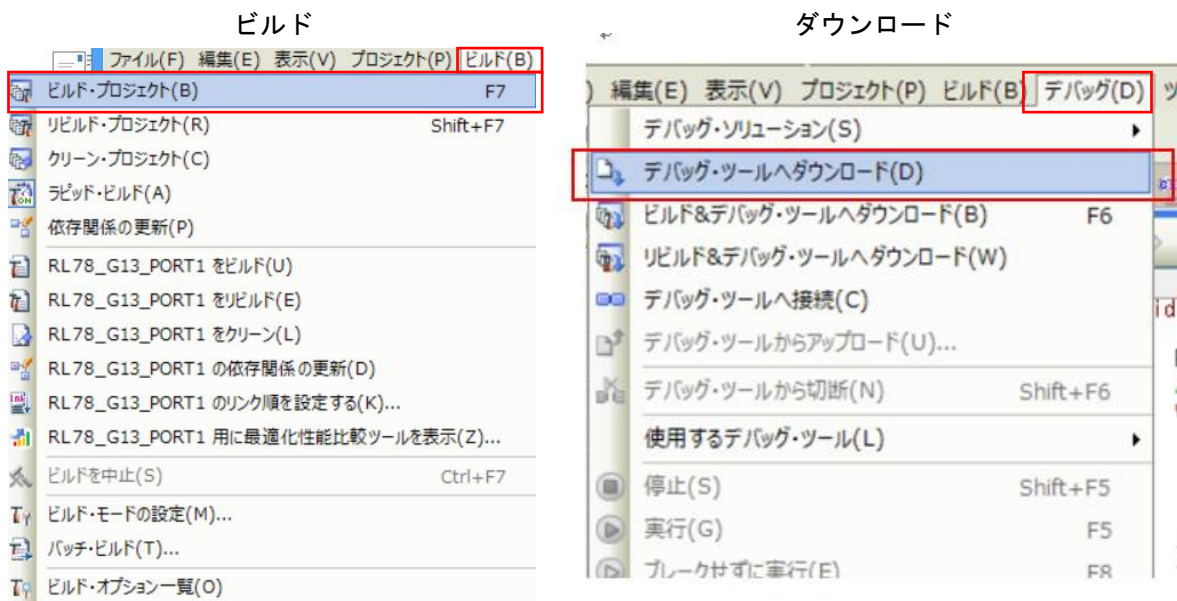
59 void main(void)↓
60 {↓
61     R_MAIN_UserInit();↓
62     /* Start user code. Do not edit comment generated here */↓
63     while (1U)↓
64     {↓
65         P3_bit.no1 = 0; /* P3.1を0（ロウ出力）にする */↓
66         wait_200();↓
67         P3_bit.no1 = 1; /* P3.1を1（ハイ出力）にする */↓
68         wait_200();↓
69  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

```

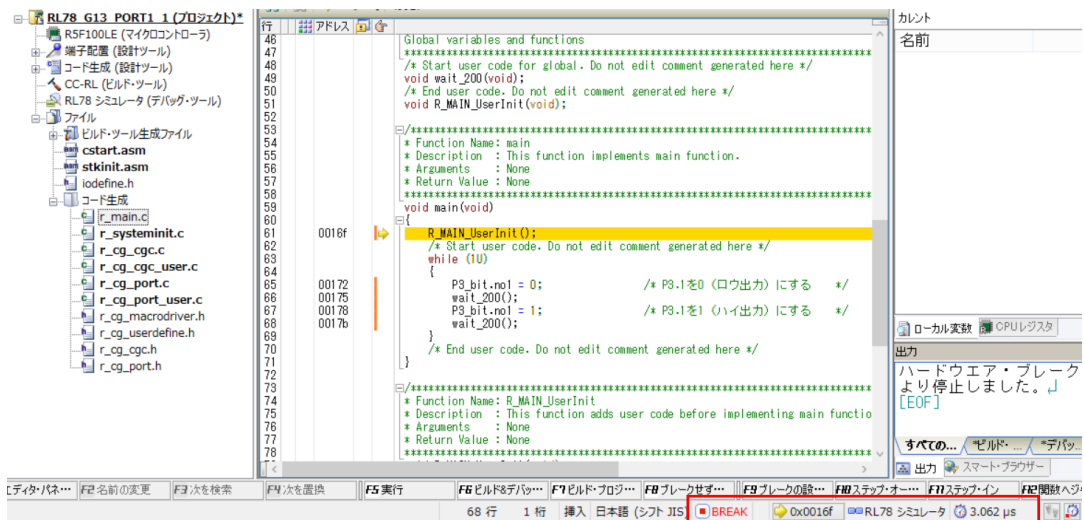
実際に、シミュレータで実行する前に、シミュレータの設定を変更します。「RL78 シミュレータ（デバッグ・ツール）」を右クリックしてメニューから「プロパティ（P）」を選択し、「デバッグ・ツール設定」タブを開きます。「タイマ」の項目をひらき、「タイマ機能を使用する」がデフォルトでは「いいえ」になっているのを「はい」に変更します



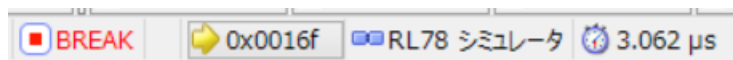
これを行うことで、シミュレータで実行時間を測定することが可能になります。それでは、ビルドして、デバッグ・ツールにダウンロードします。



ダウンロードが完了して、シミュレータが起動した画面を下に示します。ここで、右下の赤く囲んだ部分（ステータス表示部）に注目してください。



この部分の拡大を下に示します。左側が 0x0016f でブレーク中であることを示しています。右端の部分の「3.062 μs」の部分ですが、ここまでの実行時間（実行開始してからブレーク状態になるまでの時間）が 3.062 μs であることを示しています。




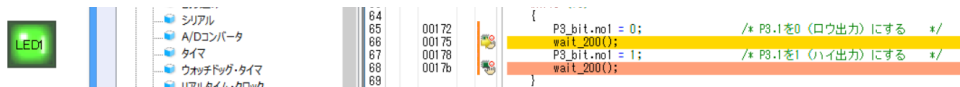
そこで、次ページに示すように 2 つの時間待ちにブレーク・ポイントを設定します。

```

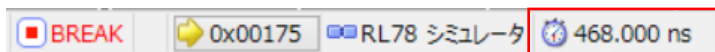
while (1U)
{
00172      P3_bit.no1 = 0; /* P3.1を0 (ロウ出力) にする */
00175      wait_200();
00178      P3_bit.no1 = 1; /* P3.1を1 (ハイ出力) にする */
0017b      wait_200();
}


```

この状態で、 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、以下のようにLEDが点灯して、時間待ちの実行前でブレークします。



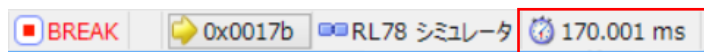
このときの画面右下のステータスを見ると、以下のように468nsの実行時間となっています。これは、R_MAIN_UserInit関数を実行し、P3_bit.no1 = 0の実行が完了するまでの時間です。



再度、 をクリックして、プログラムを実行して、上側の時間待ちと次のP3_bit.no1 = 1を実行します。その結果は以下のようにLEDが消灯した状態になります。



ここでステータスは以下ようになります。



実行時間が170msとなっていることが確認されました。目標の200msよりは15%程度短くなっていますが、ほぼ予想通りの時間です。

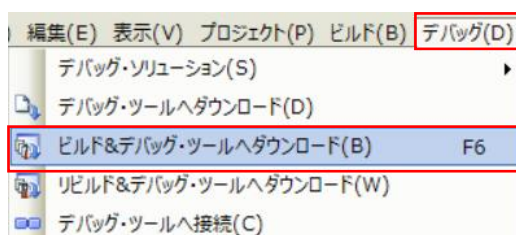
これを、より200msに近づけるためには、wait_200のループ（20クロックと予想）を3クロック分長くすればいいことになります。このためには、以下のようにNOP()を3個追加するのが簡単です。

```

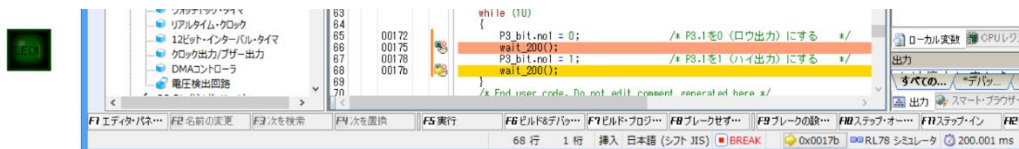
for ( time = 320000 ; time > 0 ; time-- )↓
{↓
  NOP() ;↓
  NOP() ;↓
  NOP() ;↓
  NOP() ;↓
}↓

```

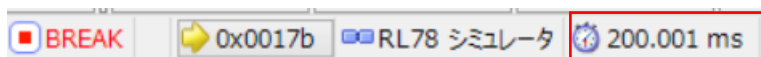
このように変更したプログラムをビルドしてシミュレータにダウンロードします。今回は、右に示すように、一気にビルドからダウンロードまでを実行します。



これを、ブレーク付きで2回実行すると、以下のようになります。



ステータス部分を拡大すると以下のようにほぼ 200ms になりました。



このように、NOP()を使うと、1クロック単位でチューニングすることができます。

これで、何とかほぼ 200ms ごとに LED を点灯／消灯を行わせることができたので、最初の目標は達成できました。プロジェクトを保存しておきましょう

ただし、このプログラムは使用している環境 (CS+CC-RL の V6.00.00) で 200ms が得られたものです。全く同じプログラム (ビットの記述方法が異なるのでそのままではだめですが) を CS+CA78K0R でビルドすると、異なる時間になることが考えられます。

CA-78K0R と CC-RL では、ビットの記述方法が異なります。
CC-RL では iodef.h ファイルでビット・フィールドを定義して対応していましたが、CA-78K0R では初期状態でビットに対応しています。

CC-RL P3_bit.no1
CA78K0R P3.1

そもそも、CA-78K0R では iodef.h ファイルではなく、内部のデバイス・ファイルで SFR を定義しています。そのため、デバイスでの定義に近い表現が可能になっています。

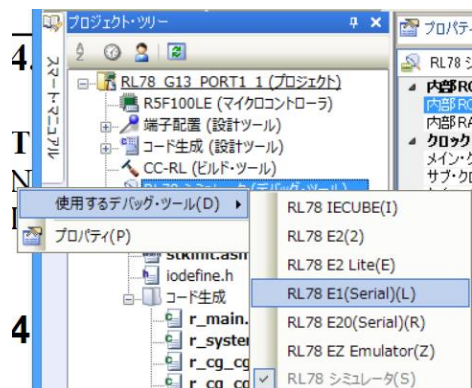
この結果を BlueBoard-RL78/G13_64pin にダウンロードして、実行すると LED が点滅するはずですが。(この時点では、実機確認はまだやっていません。今後、E1 または E2Lite を用いたデバッグの例を示します。)

7. BlueBoard-RL78/G13_64pin での動作確認

E1 を使って、実際に RL78/G13 (BlueBoard-RL78/G13_64pin) で動作させます。

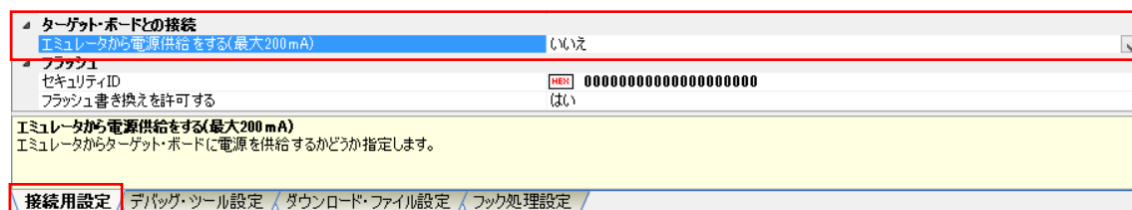
7.1 デバッグ・ツールの変更

使用するデバッグ・ツールを変更するには、プロジェクト・ツリーの RL78 シミュレータ (デバッグ・ツール) を右クリックしてメニューを表示し、「使用するデバッグ・ツール (D)」を選択すると、選択可能なツールが表示されるので、「RL78 E1 (Serial) (L)」を選択します。これでデバッグ・ツールとして E1 が選択されます。

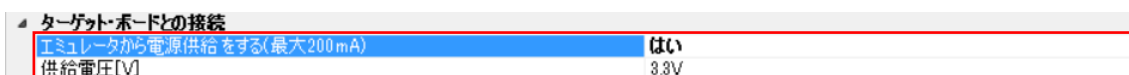


プロジェクト・ツリーの「RL78 E1 (Serial) (デバッグ・ツール)」を右クリックしてメニューを表示して、「プロパティ (P)」を選択します。(E2 Lite を使用する場合は、「RL78 E2 Lite (E)」を選択します。)

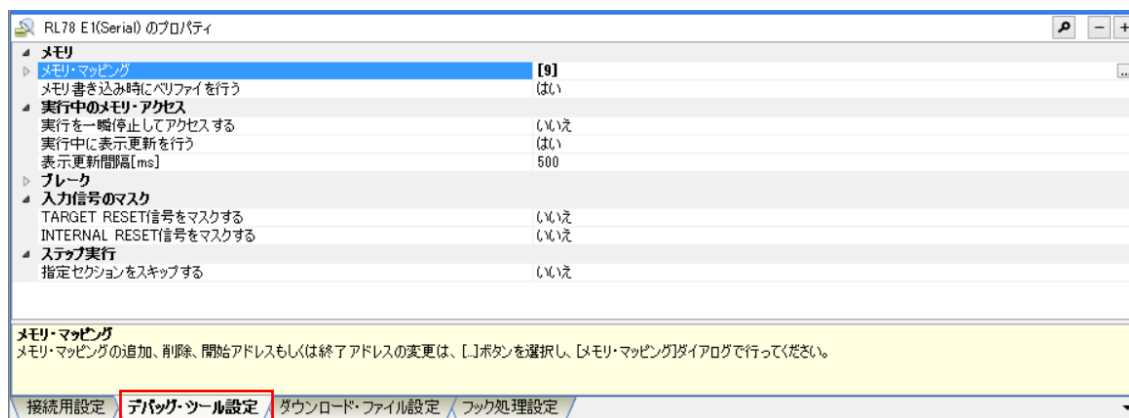
「接続用設定」タグが表示されるので、「ターゲット・ボードとの接続」を開いて「エミュレータから電源供給をする (最大 200mA)」を見ると「いいえ」となっています。



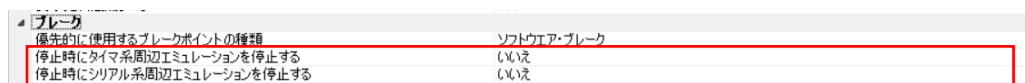
今回はこれを「はい」に変更し、「供給電圧[V]」を 3.3V にします (E2Lite では、3.3V しか供給できません)。



次に、「デバッグ・ツール設定」タグを選択します。



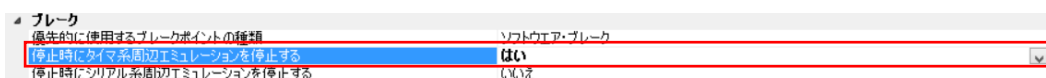
ここで「ブレーク」の項目を開きます。



「優先的に使用するブレーク・ポイントの種類」が「ソフトウェア・ブレーク」となっていますが、これはブレーク・ポイントを設定した命令をブレーク命令に置き換えてしまうことで、プログラムの実行を中断（ブレーク）するものです。このためにデバッグ中には頻繁にフラッシュ・メモリしまいます。このために、デバッグで使用したデバイスは製品に使用しないように制限されています。

ソフトウェア・ブレークの他にも、ハードウェア・ブレークがあります。しかし、設定できるポイントが少ないことと、設定したポイントではなく、そこから少しずれてブレークするので、使いにくいのであまりお勧めできません。

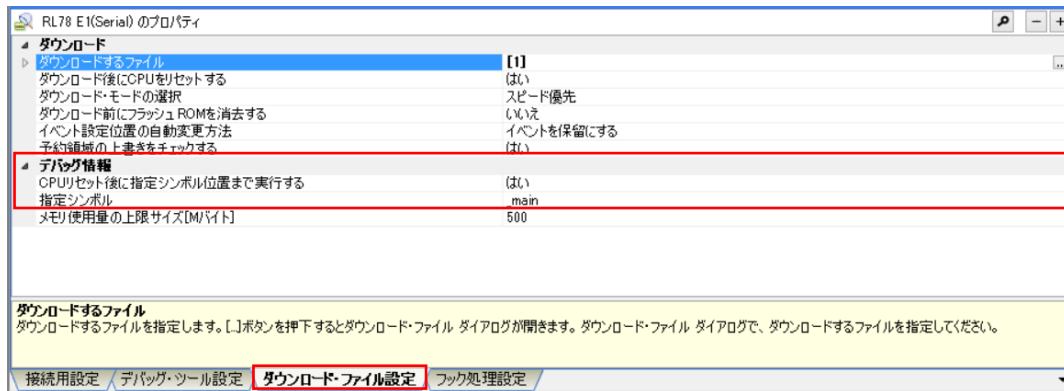
ここで、設定を変更したいのは、ブレーク時にタイマ系を停止させることです。



「停止時にタイマ系周辺エミュレータを停止する」を「はい」に変更します。こうすることで、ブレーク中タイマが停止し、変なタイミングでタイマからの割り込みが発生することがなくなります。（逆に言うと、この設定が「いいえ」になっていると、ブレーク中で CPU が停止している状態でもタイマがカウント動作を継続し、次々と割り込みが発生してしまい、プログラムが正常に実行できなくなる可能性があるからです。

シリアル系は、停止しない方がいいでしょう。これは、通信は必ず通信相手が存在します。できるだけ、相手と合わせることを考えると、動作させておいた方がいいからです。もちろん、これは場合場合で異なるので、どうしても止める必要があり、それで以降の動作に問題が発生しないなら止めるのもあります。

次は、「ダウンロード・ファイル設定」タグを開きます。そこで、「デバッグ情報」の項目を見ると、「CPU リセット後に指定シンボル位置まで実行する」が「はい」で、「指定シンボル」が「_main」となっています。これは、デバッグを起動したときに main 関数の先頭で停止するというものです。つまり、スタートアップ・ルーチンの実行が完了し、内蔵周辺機能の初期化や RAM の初期化が完了しているところからデバッグを開始することを意味しています。



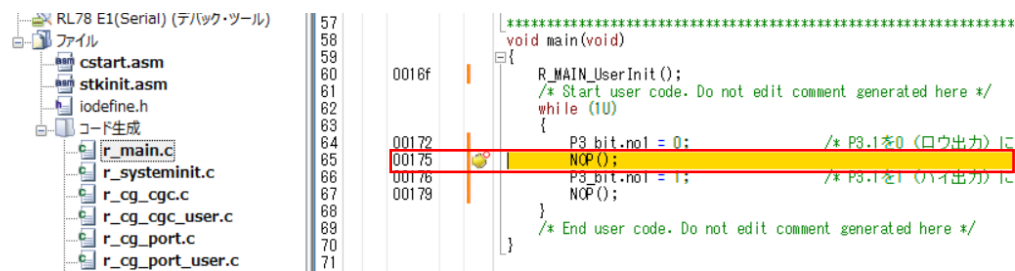
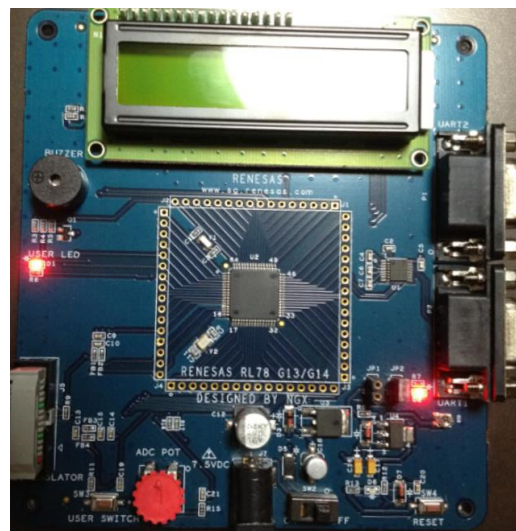
ここで、「main」ではなく、「_main」となっているのは、アセンブラでのシンボル名が使われているので、C表記のシンボルの前に「_」が付いているからです。

7.2 ハードウェアの接続と実行

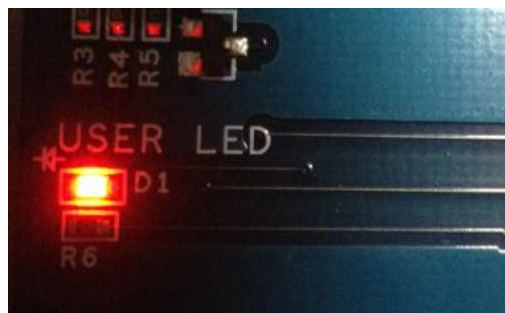
使用するハードウェアは、右に示すボードです。左下の方に E1 を接続するための 14 ピンのコネクタがあり、そこに E1 を接続して、E1 から 3.3V の電源を供給しています。

右に示す写真は、下に示すように、LED を点灯させた直後の NOP(); でブレークをかけた状態です。

このボードには、下側の辺の中央に AC アダプタを接続するコネクタがあります。秋月の HP の説明やドキュメントには、6.5V の AC アダプタを接続するように書かれていますが、これは間違いでしょう。(回路図では 7.5V と書かれていますし、そもそも 6.5V の AC アダプタは存在しません。)

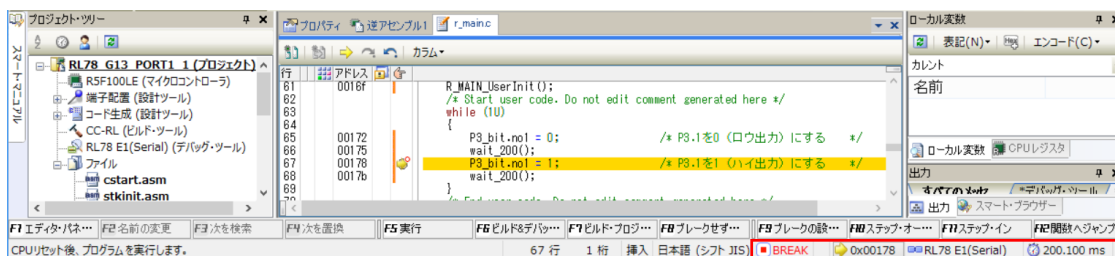


このときのLEDの点灯状態が、下の左側の写真です。右側は、ブレークを掛けず、RUN（実行）させたときの写真です。短い周期で点滅を繰り返していますが、目でも写真でも点滅は見えません。写真では、点灯状態と比べると暗くなっているのが分かる程度です。

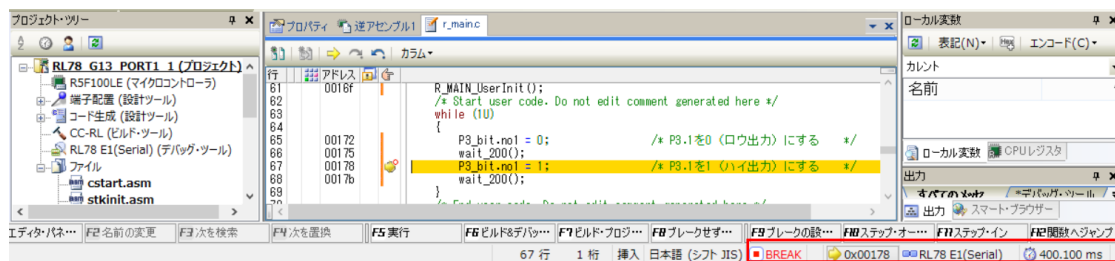


200msのタイマを呼び出す処理を追加したプログラムでは、400ms周期での点滅が確認できました。点滅の様子をビデオでもと思いましたが、ファイルが大きくなり過ぎるのでやめました。実際にハードウェアを動作させて、自分の目で確認してください。

ここでも、実際にプログラムを実行させて、実行時間を計測してみます。LEDを消灯するところにブレーク・ポイントを設定してみました。一回目のブレーク時の測定結果は、200.100msとなっています。なお、シミュレータの場合と異なり、この実行時間はあまり正確ではありませんので、ご注意ください。



更に、ブレーク・ポイント付きで実行させると、下に示すように400.100msと表示されました。



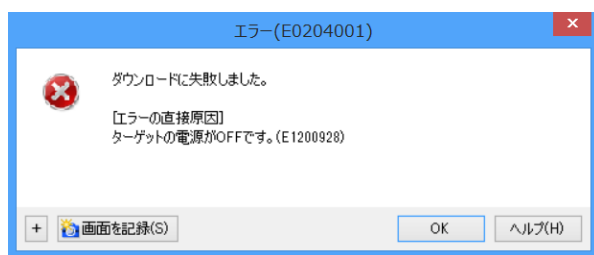
これで、実際のRL78/G13のデバイスでの動作も確認できました。

最後になりましたが、E1 から電源を供給する場合には、他からの電源供給とぶつからないように注意してください。他から電源を供給するときは、E1 からの電源供給はしないように設定してください。

また、E1 から供給できるのは 200mA までです。それ以上流れるような場合には、E1 から供給しないでください。

これらが守られないと、E1 が壊れる可能性があります。

なお、ボードに電源が供給されていない場合には、下のようなメッセージが表示されます。



【おまけ】

AC アダプタから 7.5V と 6V を入力して、CPU_VDD を測定したところ、どちらの場合も 4.4V でした。少なくとも、RL78 は 2.7V 以上の電圧なら 32MHz で動作可能なので、AC アダプタの電圧はあまり気にしないで進めていきます。ちなみに、5V の AC アダプタを使ったところ、LCD がどうも表示できなくなるようです（スイッチや LED は使えそうですが）。

次回は、ポートの入力と出力を組み合わせた制御を行います（単に SW を押せば LED が点灯するだけです）。

以上