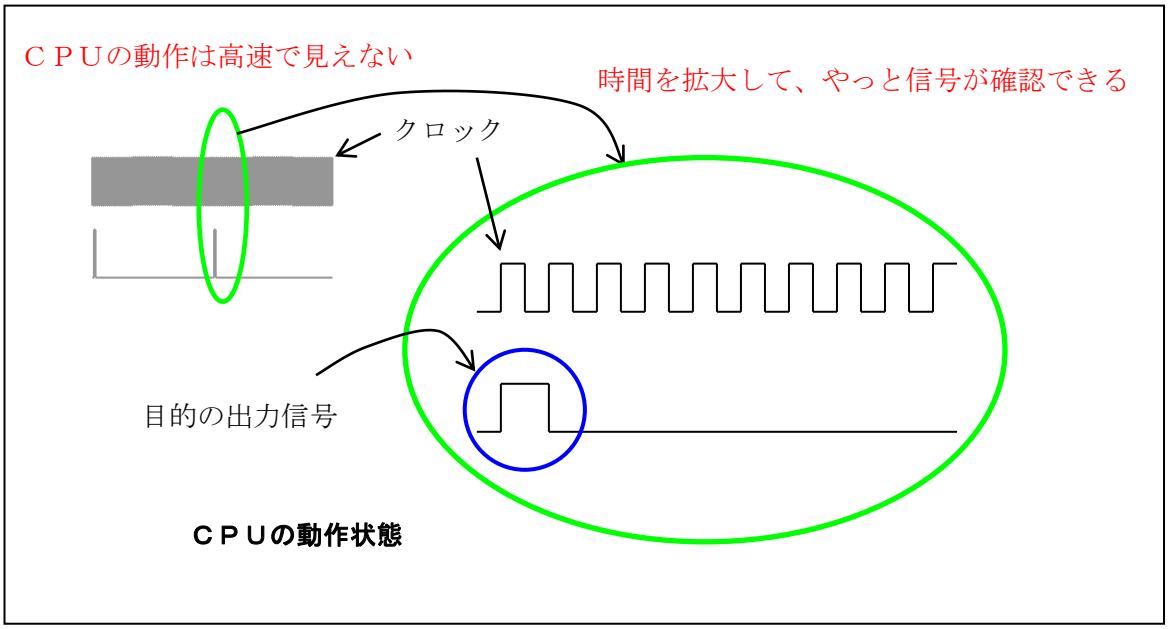
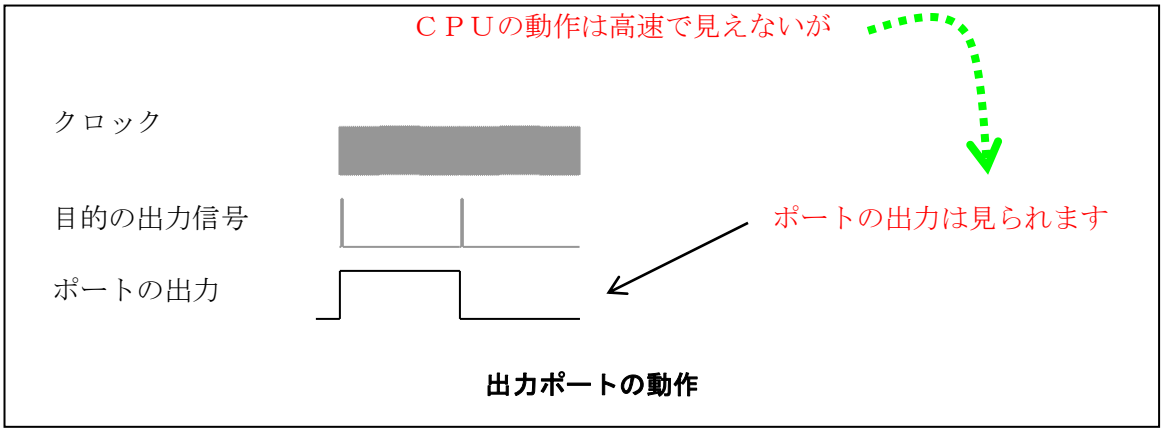
マイコンのポート機能（出力）

マイコンは数十 MHz のクロックで動作しています。そのようなマイコンが
信号（パルス）を出力しても，その幅は数十 ns と非常に短くなってしまい，
いろんな外部機能を動作させることができません。



そこで，外部機能とやり取りを行うための最も基本的な機能がポートです。
出力ポートではデータを保持しておくことができます。そのため，マイコンが
たとえ，数十 MHz で動作していても，1ms や 1s 幅の信号を出力することが可能
になり，外部機能を動作させることが可能になります。

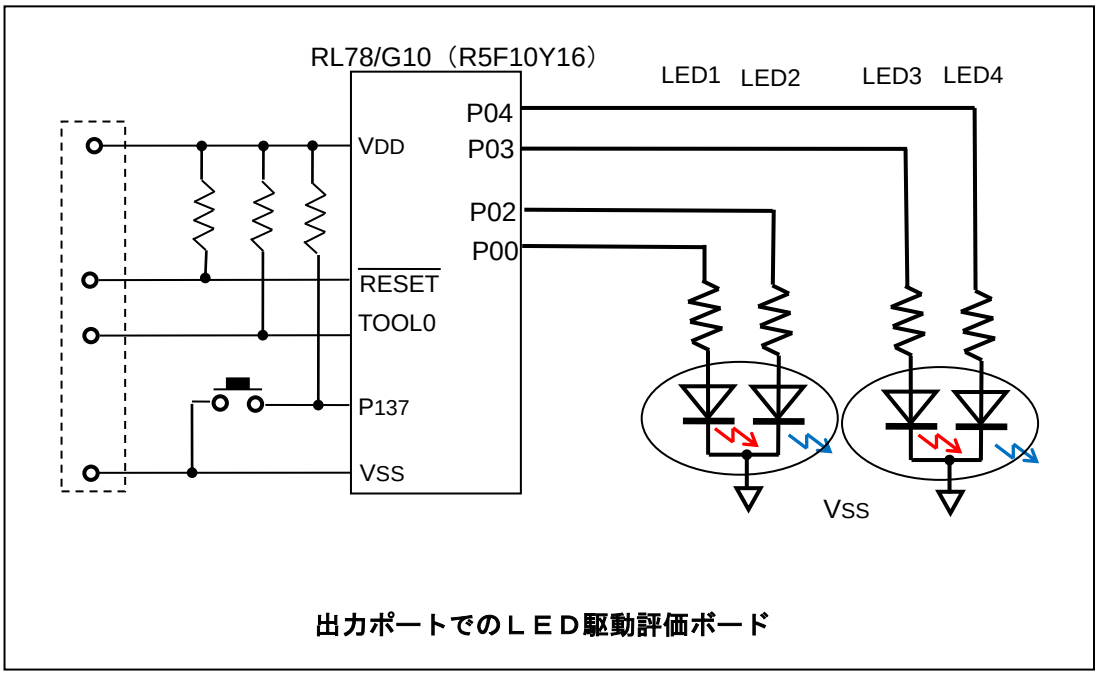


また，プログラムの手間と処理時間を無視すると，ほとんどの機能はポートと
ソフトウェアで実現できると言えます。昔はこのようなことがよく行われて
いました。
ポートとソフトウェアで実現すると，処理に時間がかかり，スピードが遅く
なったり，ソフトウェアが複雑になり過ぎてしまったりすることから，限界が
あり，幾つかの代表的な機能は，ハードウェアとしてマイコンに内蔵される
ようになってきています。

脇道にそれましたが，最も基本的な機能であるポートをどのように使用するか
を実際にみてみましょう。
ここでは，見てすぐに分かるように，LED を点灯させることを考えてみます。

使用するのは，RL78/G10 の 10 ピンの製品である R5F10Y16 です。
P0.0，P0.2，P0.3，P0.4 の 4 つのポートを LED の制御に使用します。

実際の回路は秋月電子の 80mm×50mm のブレッドボード（EIC801）で試作して、
ユニバーサル基板（AZ0526）に落とし込んだもので、下記の回路となっています。
RL78/G10 部分はソケットになっており、評価ボード（QB-R5F10Y16-TB）をセット
したり、北斗電子のマイコンボード（HSB78G10-10）をセットしたりすることが
できます（簡単な評価には DIP タイプが便利です）。
また、E1 を直接接続して書き込んだり、デバッグしたりできるようなピンも立てて
います。E1 のコネクタに北斗電子の E1 接続コネクタ(TOE1)を接続して、そこから
ジャンパで接続します。



出力ポートでのＬＥＤ駆動評価ボード

http://japan.renesas.com/products/tools/introductory_tools/cpu_board/qb_r5f10y16_tb/index.jsp
http://akizukidenshi.com/catalog/g/gP-04303/
http://www.hokutodenshi.co.jp/7/TOE1.htm

使用するプログラムは統合開発環境"CS+"で，"G10_PORT0"の名前を付けて，
プロジェクトを作成してあります。

(1)点灯準備
RL78 でポートを使って LED を制御する場合には、まずポートを出力に設定する必要が
あります。この回路例では，P0 を使用しているので、P0 を出力ポートに設定します。
そのために必要なレジスタは，PMC0 レジスタと PM0 レジスタで、その設定は以下の
ようになります。

```
PMC0 = 0b11100011; /*
            +++--- P0.2～P0.4 をデジタル入出力に設定します。
*/
PM0 = 0b11100010; /*
          +++-+-- P0.0,P0.2～P0.4 を出力ポートに設定します。
*/
```

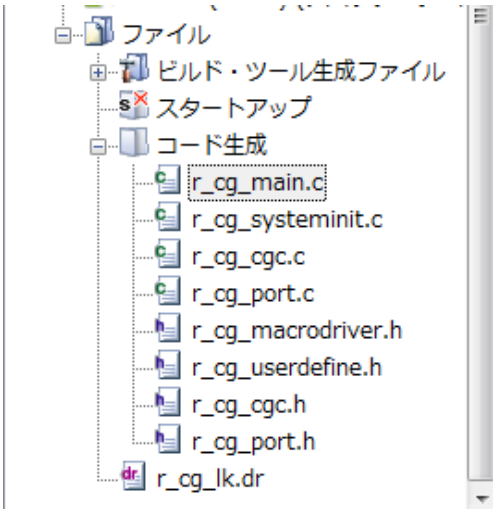
又は，アセンブリ言語では以下のようになります。

```
MOV     PMC0,   #11100011B
MOV     PM0,    #11100010B
```

1 行目の PMC0 レジスタの設定が、ポート 0 をデジタルポートとして使うための設定で，
2 行目の PM0 レジスタの設定がポートを出力ポートとして使うための設定です。
この 2 行は最初に初期設定で実行するだけで済みます。
この部分はプログラムを作成してもいいのですが，統合開発環境 CS+に内蔵されている
コード生成機能を使うと簡単にできます。

コード生成での設定

この設定で、コード生成すると、以下のようなファイルが生成されます。



この中で、”r_cg_port.c”ファイルがポート関係の初期設定プログラムになります。
その中には、“R_PORT_Create”関数が以下の内容で生成されています。

```
void R_PORT_Create(void)
{
    P0 = _00_Pn0_OUTPUT_0 | _00_Pn2_OUTPUT_0 | _00_Pn3_OUTPUT_0 | _00_Pn4_OUTPUT_0;
    PU4 = _01_PUn0_PULLUP_ON;
    PMC0 = _02_PMCn1_NOT_USE | _00_PMCn2_DI_ON | _00_PMCn3_DI_ON | _00_PMCn4_DI_ON | _E1_PMC0_DEFAULT;
    PM0 = _00_PMn0_MODE_OUTPUT | _02_PMn1_NOT_USE | _00_PMn2_MODE_OUTPUT | _00_PMn3_MODE_OUTPUT |
          _00_PMn4_MODE_OUTPUT | _E0_PM0_DEFAULT;
}
```

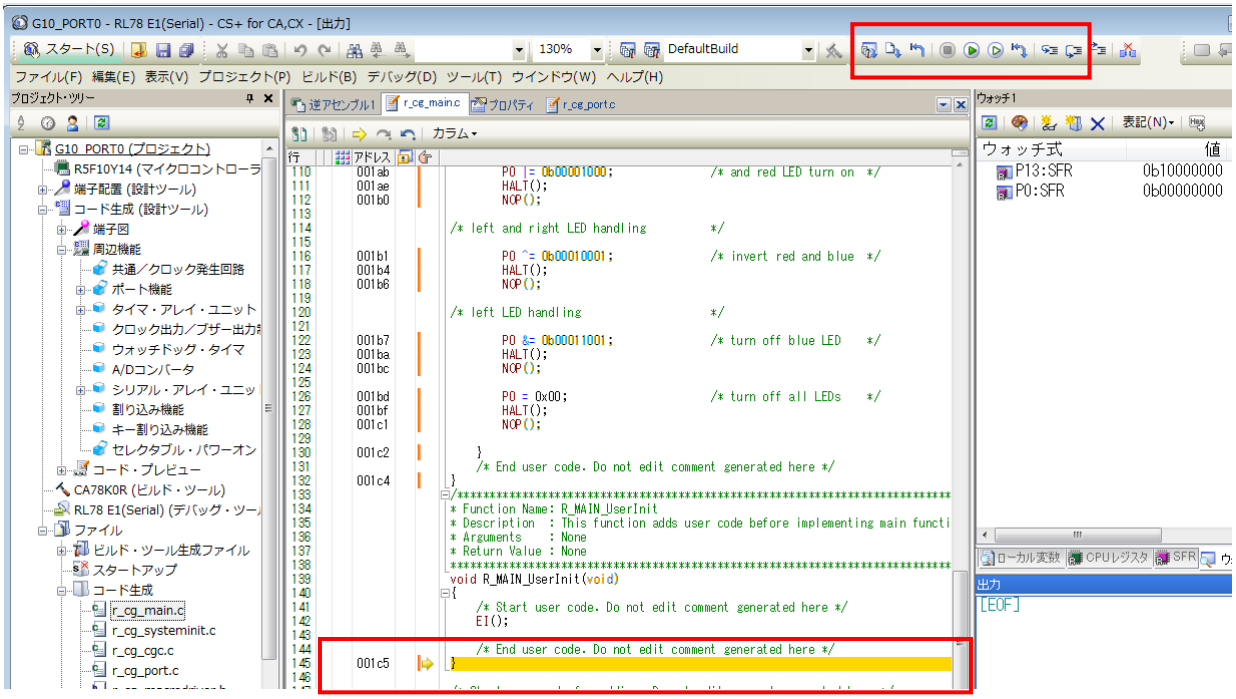
PMC0 レジスタおよび PM0 レジスタの設定がなされているのがわかります。
後は発光させたい LED に応じて P0 に設定する値を変化させるだけです。

RL78/G10 に慣れるまでは、コード生成機能を利用することをおすすめします。

また、”r_cg_main.c””ファイルが main 関数を含むファイルです。ここに、処理したい
内容を追加していきます。
このプロジェクトではすでに、プログラムが作成済みですので、以下は実際に
E1 によるデバッグでの動作（LED の点灯）確認です。

(2)点灯確認
ここでは E1 エミュレータを用いて，オンチップ・デバッグにより確認します。

早速プロジェクトを起動し、ビルドした結果をデバッグ・ツール（E1）に
ダウンロードします。
ダウンロードした直後の画面イメージを示します。



実行の制御は右上の赤く囲んだ部分のボタンをクリックすることで行います。

このイメージの下側の黄色になっているところから実行されることになります。
通常は、周辺機能の初期化が完了して、main 関数の先頭で停止した状態となるの
ですが、RL78/G10 ではソフトウェアでのブレークが使えず、ハードウェアでの
ブレークだけとなります。
このため、main 関数の最初に呼び出されている”R_MAIN_UseInit”関数のところで
停止した状態でスタートしています（ブレーク・ポイントから数命令スリップした
ところでブレークがかかります）。

main 関数には LED を接続したポートを操作する命令と確認し易くするための命令
が並んで記述されています。その最初の部分は以下のようになっています。
最初がポートを操作する命令です。ここでは，P0.0 に対してビット操作で"1"を
代入しています。次の 2 行は点灯状態を確認するために，プログラムの実行を停止
させるためのものです。

```
        P0.0 = 1;                       /* turn on red LED      */
        HALT();
        NOP();
```

 ボタンをクリックして、プログラムを実行すると，P0.0 からハイレベルが出力され，
LED1（左の赤い LED）が点灯します。

その状態で CPU は HALT 命令を実行して、停止します。

そこで、  ボタンをクリックし、実行中のプログラムを停止させ、ブレークさせることで、
デバッガが動作状態になります。このときのデバッガの画面イメージを示します。
ここで、NOP();の行が黄色く表示され、P0.0 = 1;と HALT();が実行され、NOP();の実行
前で停止していることがわかります。

```
58                          void main(void)
59                          {
60      00172                   R_MAIN_UserInit();
61                              /* Start user code. Do not edit comment generated here */
62                              while (1U)
63                              {
64
65                          /* port handling by bit manipulation    */
66                          /* left LED handling                    */
67      00175                       P0.0 = 1;                       /* turn on red LED      */
68      00178                       HALT();
69      0017a                       NOP();
70
71      0017b                       P0.0 = 0;                       /* turn off red LED     */
72      0017e                       HALT();
73      00180                       NOP();
74
75      00181                       P0.2 = 1;                       /* turn on blue LED     */
76      00184                       HALT();
77      00186                       NOP();
78
```

次は，以下のように，P0.0 に 0 を設定してロウレベルを出力するので，LED は消灯します。

```
P0.0 = 0;                       /* turn off red LED     */
HALT();
NOP();
```

参考として、この処理を、アセンブリ言語を用いて記述すると、以下のようになります。

```
CLR1        P0.0                        ; turn off red
HALT
NOP
```

次は LED2（左の青い LED）の点灯と消灯です。

引き続いて，LED3（右の赤い LED），LED4（右の青い LED）を制御します。

```
0019f                       P0.4 = 0;                       /* turn off blue LED    */
001a2                       HALT();
001a4                       NOP();

                        /* port handling by byte data           */
                        /* left LED handling                    */

001a5                       P0 = 0b00000101;                /* turn on red and blue */
001a8                       HALT();
001aa                       NOP();

                        /* right LED handling                   */

001ab                       P0 |= 0b00001000;               /* and red LED turn on  */
001ae                       HALT();
001b0                       NOP();

                        /* left and right LED handling          */

001b1                       P0 ^= 0b00010001;               /* invert red and blue  */
001b4                       HALT();
001b6                       NOP();
```

ここまでは、ポートを１本単位で制御する（ビット操作）場合の処理例です。

引き続いて，ポートをまとめて制御する方法での制御です。

最初に，一番簡単な代入により，左側の LED の赤と青を同時に点灯させます。

```
        P0 = 0b00000101;                /* turn on red and blue */
```

アセンブリ言語では、以下のようになります。

```
        MOV     P0,     #00000101B              ; turn on red and blue
```

次に，ポートに対する論理和演算を行って右 LED の赤を点灯させます。

```
        P0 |= 0b00001000;               /* and red LED turn on  */
```

これも、アセンブリ言語では、以下のようになります。

```
        OR      P0,     #00001000B              ; and red LED turn on
```

次は，排他的論理和の演算を行って，2 つのビットを反転させ，左の LED の
赤を消灯し，同時に右 LED の青を点灯させます。

```
        P0 ^= 0b00010001;               /* invert red and blue  */
```

これも、アセンブリ言語では、以下のようになります。

```
        XOR     P0,     #00010001B              ; invert red and blue
```

次は，論理積演算により左の LED の青を消灯（左の LED は消灯）させます。

```
        P0 &= 0b00011001;               /* turn off blue LED    */
```

これも、アセンブリ言語では、以下のようになります。

```
        AND     P0,     # b00011001B             ; turn off blue LED
```

最後に代入で全ての LED を消灯します。

```
        P0 = 0x00;                      /* tuen off all LEDs    */
```

このように、ポートに対して代入（MOV）や演算処理を行うことで、簡単に LED の
つまりはポートの制御が実現できます。
本来、内蔵周辺機能の制御レジスタ（SFR）にはこのような演算処理は行えないの
ですが、ポートの場合には FFF00～FFF0F のアドレスに配置されていることにより、
ショート・ダイレクト・アドレッシング領域（saddr 領域）として扱うことができ、
saddr 領域としてこのような演算が可能になります。これでプログラム効率を向上
させています。

＜ポートの使い方（出力）＞終了