パソコンと
リアルタイム処理の相性

パソコンは、超高速なCPUと、ふんだんにあるメモリ資源、大容量HDD等の補助記憶を使う事が出来、高解像度のグラフィック、LAN接続も出来ます。

何でも出来そうな感じですが、パソコンの苦手な事もあります。 パソコンというよりも、Windows等の OSでハードウェア資源を管理されているので、直接IO命令等の特権命令は使えません。 それと、リアルタイム性（特に早い応答）を、保証出来ません。 Windows等の OSは、多数のプロセスをマルチで起動しています。

そしてタイムスライス的に管理してます。よって自分のプロセスに CPU資源を渡されるまで、そのプロセスは待ち状態になります。 特に起動直後は、いろんな事をやってるようで、一次的に画面描画が止まる、音が途切れる、キーボードの文字がすぐに入らない。 という経験をした方も多いと思います。

よって、OSで管理されるコンピュータは特殊なリアルタイムOSを除き、リアルタイム処理を要求する組込み処理や、計測制御の用途では、使えない と 考えた方がいいと、私は考えます。

リアルタイム処理の救世主
組み込み用ワンチップマイコン

逆に、組込み用ワンチップマイコンは、パソコンに比べ、CPUの処理能力が低い。メモリが少ない。 特に RAMが少ない。

等の、制限がありますが、低価格ゆえ、1つの仕事に専念させる事が出来ます。

マイコンには、タイマー割込みの機能もあるのでやや時間のかかる演算処理をしながら、IOのスキャニングを タイマー割込みで、定周期で行う事が出来ます。

今回使う、百円マイコン R8C/M120Aでも以前作成したSTEPモーターの加速、減速制御に、10KHz（ 1秒間に10,000回 ）のタイマー割込み処理を 使用しました。そういう意味では。組込み用マイコンはリアルタイム処理の救世主といえます。

ただ、パソコンに比べ、メモリ資源が少ないので効率のいいプログラムを工夫して作る必要が あります。

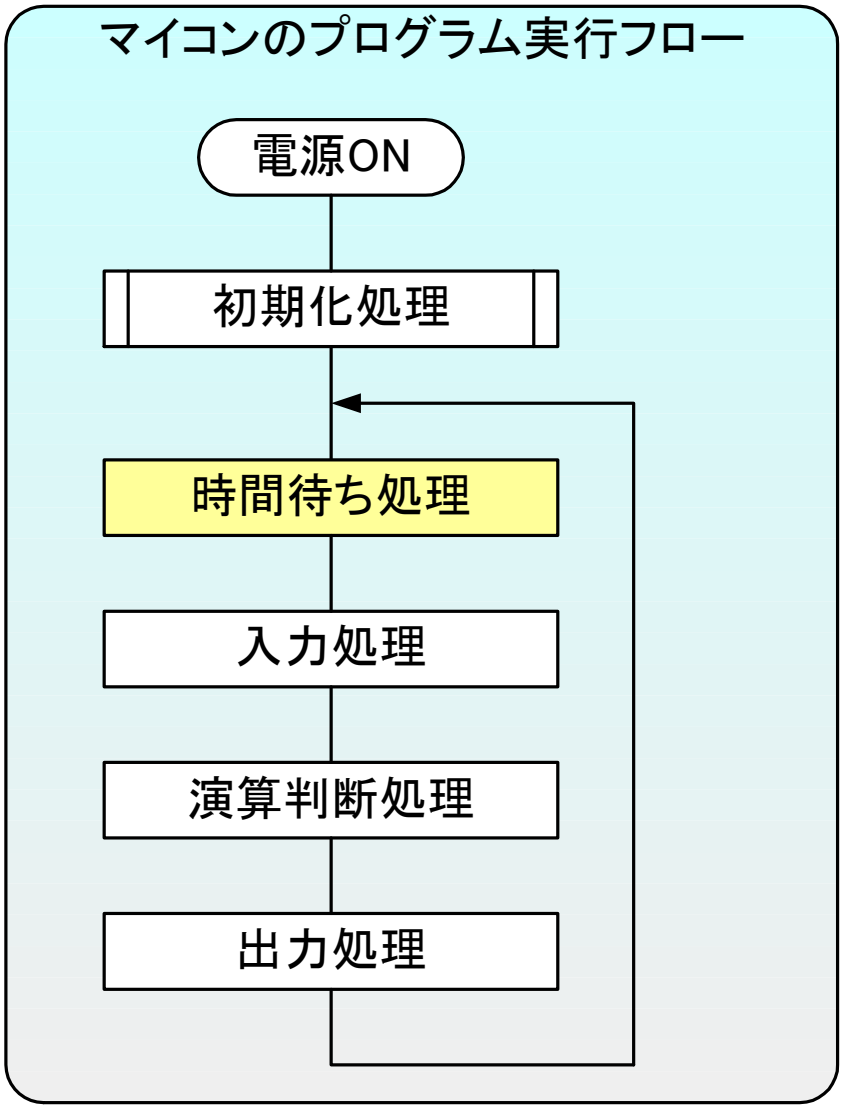
また、用途によっては、USB-シリアルインターフェースを用いてパソコンと、マイコン間で データの やり取りも出来ます。

マイコンのプログラムは終わる事のないループ処理

非常に基本的な話ですが
OSで管理されるプログラムは、用が済んだら終わるのが 当たり前です。

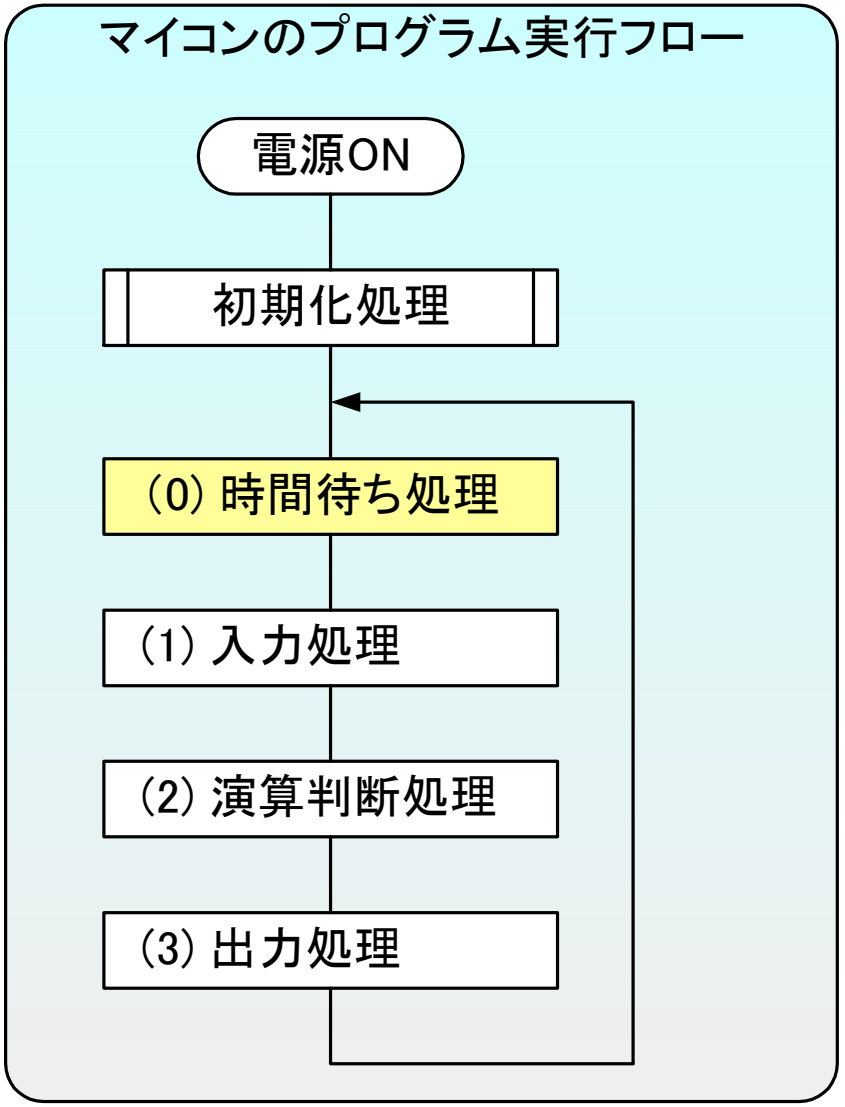
しかし、ワンチップマイコンは、書き込んだプログラムが、全てですので、終わるという概念がありません。　延々メインループを定周期で回り続けます。

しいていえば、電源を切った時が、終わる時です。 よってマイコンのプログラムは右のフローチャートのようになります。



マイコンのプログラム実行フロー

まず電源ON直後に、各周辺回路、変数の初期化を行います。 １回のループ処理は、（1）その瞬間のデータを取込み、そのデータを元に、(2) 演算判断処理を行い、(3) 出力を行います。 よって、１回まわる周期内で出来る事を、極力 短時間で処理します。
時間のかかる処理を、入れてはいけません。
本格的にやる場合、ループの頭にある、
(0) 時間待ち処理は、入れずに
タイマー割込み処理で行います。
今回は、入門者向けという事で、メインループに時間待ち処理を入れて、大雑把に一定時間のタイムインターバルを作り、実験を行います。

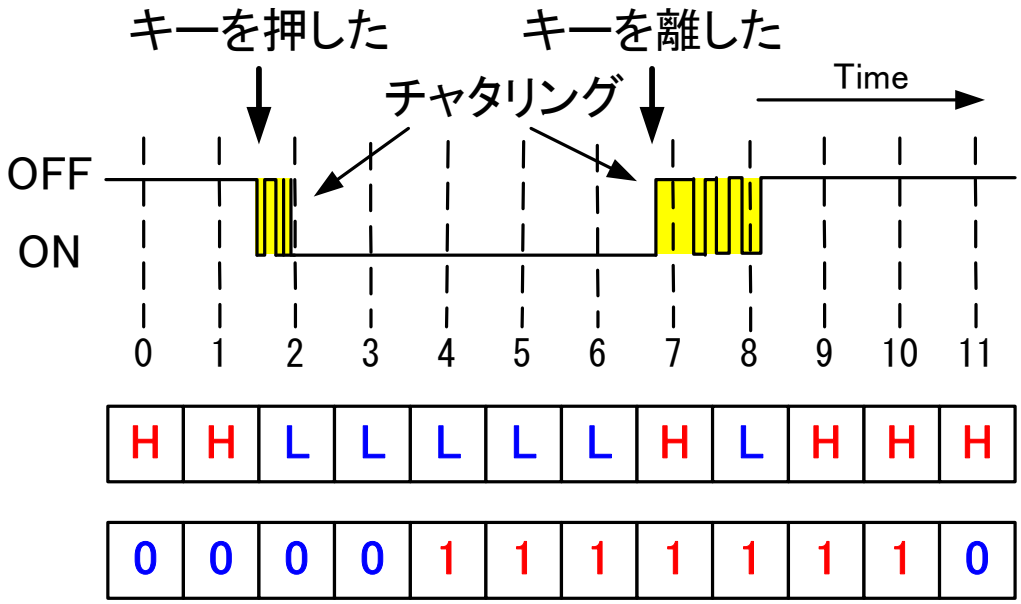


②キー入力の チャタリングキャンセル

キーの接点が、ONしたり、OFFする瞬間は細かいON、OFFが繰り返される場合があります。 これをチャタリングといいます。

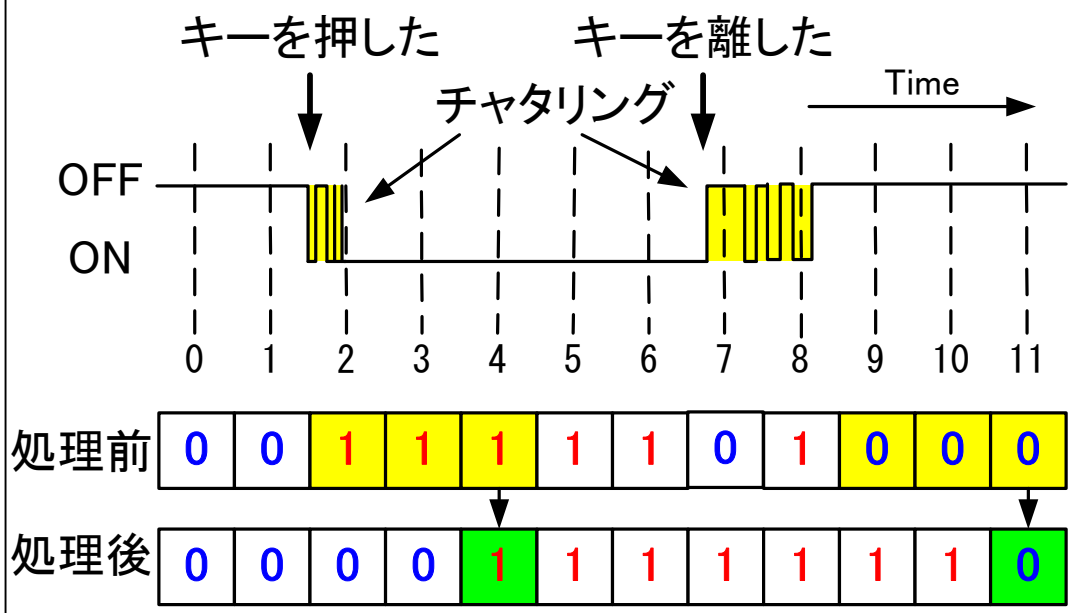
これがあると、キー入力の誤動作になるので、除去する必要があります。

今回は、デジタルフィルタという方法でチャタリングを取り除きます。 0～11までの破線は、接点の状態をサンプリングするタイミングです。サンプリングした結果は、H と L で 表示しています。キーを押したタイミングでは、チャタリングは、サンプル 1 と 2の間に　納まっています。



キーを離したタイミングでは、サンプル 7 と 8 の間でチャタリングが発生して、H、Lの判定もバタついています。 これを除去するために、仮に L が 3回続けば キーが押されている(論理1)、H が 3回続けば キーが離された(論理0) と 判断する事にします。

論理値の表現



今回、キー入力等の外から取り込む信号、LEDなどの外へ出す信号が、負論理になっております。、前ページでは、H と L で、表現しましたが、ここでは 負論理の 1 、0 で表現してます。

しかしマイコンの中では、正論理で、0 と 1 で表現されます。 通常 1 が アクティブです。 処理前のレジスタで、1が 3回続けば、処理後のレジスタが、1になります。 但しこの場合、一つ条件があり、処理後レジスタのそれまでの状態が、0である事が前提です。 処理後レジスタの最新値が1になったら、今度は、0が 3回続く事を監視します。 つまり処理後レジスタの最新値が、0であるか、1であるかで 状態変移の監視する 論理レベルが 変わるのです。

ある意味ヒステリシスを持たせたような状態で、処理後レジスタは、チャタリングも含め ノイズ等で、簡単に変わらないようにしているという事です。

プログラムの説明前に マクロ宣言の説明

私は、符号なし変数宣言時、

```
unsigned char  a;
unsigned int   b;
```

と宣言する代わりに、

```
BYTE  a;
WORD  b;
```

と宣言してます。これを実現するにはソース先頭、またはヘッダファイル内で、マクロ宣言

```
#define BYTE unsigned char
#define WORD unsigned int
```

を、宣言すれば可能です。

今回の用途で、いくつかマクロを用意してます。

```
#define LED_G   p1_2 // 緑LEDポート
#define LED_Y   p1_1 // 黄LEDポート
#define LED_R   p1_0 // 赤LEDポート
#define ON_LED    0 // LED点灯
#define OFF_LED   1 // LED消灯

#define PSW_1   p1_5 // SW_1ポート
#define PSW_2   p1_6 // SW_2ポート
#define PSW_3   p1_7 // SW_3ポート
#define PSW_ON    0 // スイッチON
#define PSW_OFF   1 // スイッチOFF

#define SHOT_COUNT 160 // ショット出力の
                       // 時間 0.8秒
```

チャタリングキャンセルを C言語でどのように実現するか

意外と簡単に実現出来ます。
前回は、状態の移り変わりを見せるため、処理前レジスタと、処理後レジスタを用意しましたが処理前のレジスタは、最新の3bitあればOKです。処理後のレジスタは、1bit フラグがあれば OKです。
BYTE変数が２個あれば出来ます。

私は構造体で宣言してます。

```
typedef struct {
    BYTE   sw_sh;  // スイッチ変移状態
    BYTE   sts;    // ON,OFF 判定値
} sw_param;     // １個のSW関連パラメータ
```

ソースの一部分を、表示します。
内容を分かりやすくするため少し変えています。

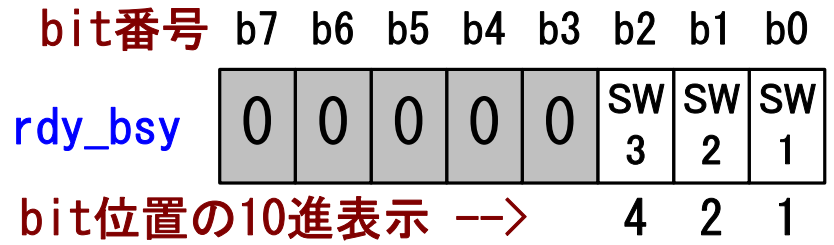
```
sw.sw_sh = sw.sw_sh << 1;
                // スイッチ変移状態 更新
sw.sw_sh &= 0x07;
                // 下位 3bitのみ残す
if( PSW_1==PSW_ON ) sw.sw_sh |= 1;
      // SW=ONの時 最下位bitを 1にする

if( sw.sts == 0 )
{     // 連続３回 SW=ON が 成立か ？
  if( sw.sw_sh == 7 )
    sw.sts = 1;      // SW=ON 確定
}
else
{     // 連続３回 SW=OFF が 成立か ？
   if( sw.sw_sh == 0 )
     sw.sts = 0;     // SW=OFF 確定
 }
```

エッジ検出と、先着優先機能

今回は、缶コーヒー販売機のイメージで作ったので、各押しボタンのチャタリングキャンセル処理は、3つ独立して処理してますが、エッジ検出と、先着優先機能は、2つが、結合した形になり、ちょっと ややこしくなりました。

```
// 制御管理用の構造体データ
typedef struce {
  BYTE  rdy_bsy;  // 待機中 処理中 Flag
  BYTE  sel_sw;   // 先着 SW番号
  BYTE  sel_sw_e; // 先着 SW番号 一つ前
  BYTE  shot_cnt; // Shot出力時間ｶｳﾝﾀ
} ctrl_param;     // 制御パラメータ
```



rdy_bsy変数は、b0～b2が 各 SW1 ～SW3のスイッチONの 時 1 になるフラグです。
各スイッチの押下状態を一度に確認できます。
0であれば、SW1～SW3は、どれも OFF状態です。 SW3だけ ONであれば、10進数で 4
SW1～SW3 全てが、ONであれば、7 になります。 rdy_bsy変数は、各スイッチの チャタリングキャンセル処理にて、ONが確定した時、該当する bit に、1が記録されます。
OFFが、確定した時は、該当する bit に 0が記録されます。

rdy_bsy変数が、設定された後に、sel_sw変数（先着 SW番号）が、設定されます。
　よって、rdy_bsy != 0　&&　sel_sw == 0 のタイミングが、スイッチが押されたエッジを検出した事になります。 エッジ検出のタイミングで、shot_cnt変数に、ワンショットパルス出力の時間長さのカウント数を設定します。

　そして、rdy_bsy変数の、b0、b1、b2 を順に調べて、最初に bit ==1 が、見つかったbitに対応する SW番号を sel_sw に設定し、後の bit は無視します。 この、sel_sw に設定された番号が、先着優先番号となります。

　残りの、sel_sw_e変数は、1サンプル前のsel_swの値を保持します。

sel_sw変数が、0 で sel_sw_e変数が、0 でないタイミングが、Busy状態から、Ready状態に戻ったタイミングで、点灯したLEDを消すタイミングとなります。
Ready状態は、SWを、どれも押して無い待機状態です。
Busy状態は、SWのどれか1つ以上押された状態です。
あるいは、全てのボタンを離しても LED点灯のワンショットパルスの点灯時間が、終わってない場合は、点灯時間が 終わるまで Ready状態に戻るのは待たされます。

```
if( cpm.shot_cnt > 0 )  cpm.shot_cnt--;
if( cpm.rdy_bsy != 0 )
{
  if( cpm.sel_sw == 0 )
  {
    cpm.shot_cnt = SHOT_COUNT;
    if( ( cpm.rdy_bsy & 1 ) != 0 )
        cpm.sel_sw = 1;
    else
    {
      if( ( cpm.rdy_bsy & 2 ) != 0 )
          cpm.sel_sw = 2;
      else
          cpm.sel_sw = 3;
    }
   cpm.sel_sw_e = cpm.sel_sw;
  }
}
else
{
  if( cpm.shot_cnt == 0 )     cpm.sel_sw = 0;
}
```

一番上の行の説明が抜けてました。
メンバー変数 shot_cnt は、LED点灯出力の残り時間となります。0 以上であれば、毎回ループ処理で回ってくる毎に、デクリメントされます。
0 になれば、LEDは 消灯します。

ここでの説明だけでは、分かりにくいと思いますので、今回も、HEWプロジェクトファイルを、ダウンロード出来るようにしておきます。
興味のある方は、ダウンロードしてソースファイル simple_2.c を見て下さい。

ワンショットパルス出力処理

先頭で、BYTE変数 led を宣言してます。
shot_cnt が 0 でなければ、led に ON_LEDを代入します。0 であればled に OFF_LEDを、代入します。
case 0　は、sel_sw が 0 という事は、どれも選択されて無いので、3つの LEDを、全て消灯して、sel_sw_e 変数を 0 に します。
case 1 は、SW_1 が、選択された状態なのでLED_R（ 赤LED ）に led の値を入れる。
case 2 は、SW_2 が、選択された状態なのでLED_Y（ 黄LED ）　に led の値を入れる。
case 3 は、SW_3 が、選択された状態なのでLED_G（ 緑LED ）に、led の値を入れる。

```
void one_shot_out( void )
{
  BYTE  led;

  if( cpm.shot_cnt > 0 )   led = ON_LED;
  else                     led = OFF_LED;

  switch( cpm.sel_sw )      // 出力LEDの選択
   {
     case 0: if( cpm.sel_sw_e != 0 )
               {
                  LED_R = OFF_LED;// 全消去
                  LED_Y = OFF_LED;
                  LED_G = OFF_LED;
                  cpm.sel_sw_e = 0;
               }
              break;
      case 1: LED_R = led; // 赤LEDの 点灯
              break;
      case 2: LED_Y = led; // 黄LEDの 点灯
              break;
      case 3: LED_G = led; // 緑LEDの 点灯
              break;
  }
}
```

構造体を一本化

最後に、main 関数の説明をする前に、説明が、前後して申し訳ありませんが、構造体宣言の、変更をしました。

最終的に ctrl_param 構造体の中に、sw_param 構造体変数を、3個 sw1、sw2、sw3 の名前で 組込みました。

構造体変数 cpm 内に 制御に必要な全ての 静的変数が、組み込こまれています。

```
//  構造体変数 宣言
// ------------------------------------------------
typedef struct {
    BYTE   sw_sh;     // スイッチ変移状態変数
    BYTE   sts;       // フィルタ処理後の ON,OFF 判定値
}   sw_param;  //  １個のスイッチ関連パラメータ

typedef struct {
    sw_param  sw1;    // スイッチ１のパラメータ
    sw_param  sw2;    // スイッチ２のパラメータ
    sw_param  sw3;    // スイッチ３のパラメータ
    BYTE  rdy_bsy;    // 待機中 処理中 判定フラグ
    BYTE  sel_sw;     // 先着スイッチ番号（ 1, 2, 3 ）
    BYTE  sel_sw_e;   // 先着スイッチ番号 消去用
    BYTE  shot_cnt;   // ワンショット出力カウンタ
}  ctrl_param;  // 制御パラメータ

//  静的変数宣言
// ------------------------------------------------
static  ctrl_param  cpm;  // 制御パラメータ実態宣言
```

メイン関数

initproc 関数(初期化処理)を、最初に行います。 その後、無限 while ループ内にて、ms_wait(5); で、5msのタイムインターバルを作ります。

次に、3つのスイッチのフィルタ処理を1本の sw_filter 関数で、変数を独立させて処理を行います。 その関係で、
1回目引数が (&cpm.sw1、1);
2回目引数が (&cpm.sw2、2);
3回目引数が (&cpm.sw3、3);
第1引数が、sw構造体変数のアドレス渡し
第2引数が、IO Port識別用スイッチ番号
と、なります。

```
void main(void)
{
    initproc();         // 初期化処理

    while( 1 )
     {
        ms_wait( 5 );          // 約 5ms 待つ

        sw_filter( &cpm.sw1, 1 );  // SW１のフィルタ処理
        sw_filter( &cpm.sw2, 2 );  // SW２のフィルタ処理
        sw_filter( &cpm.sw3, 3 );  // SW３のフィルタ処理

        first_sw_sel();     // 先着スイッチ番号判定
        one_shot_out();     // ワンショット LED点灯
     }
}
```

おまけ、アドレス渡し ポインター、参照渡し

前ページの説明で、
sw_filter(&cpm.sw1, 1); が、ありました。
第一引数の &cpm.sw1 が、構造体変数 cpm 内のメンバー変数 sw1 の値ではなく、変数がメモリ上に 格納されているアドレス値 を、渡すという事で & で 指定しています。
呼び出される関数のプロトタイプは、
void sw_filter(sw_param *sw, int sel);
で、それを、ポインター変数として受け取ります。
構造体変数ポインターの場合、ちょっと厄介なのは、sw 構造体の メンバー変数 sw_sh をアクセスする場合は、sw->sw_sh と記述します。
(-> は、アロー演算子と呼ぶようです)

ポインターではなくて 構造体変数の実態をアクセスする場合は、sw.sw_sh になります。 いつも、構造体のポインターを使うとき -> は、面倒だなと思います。

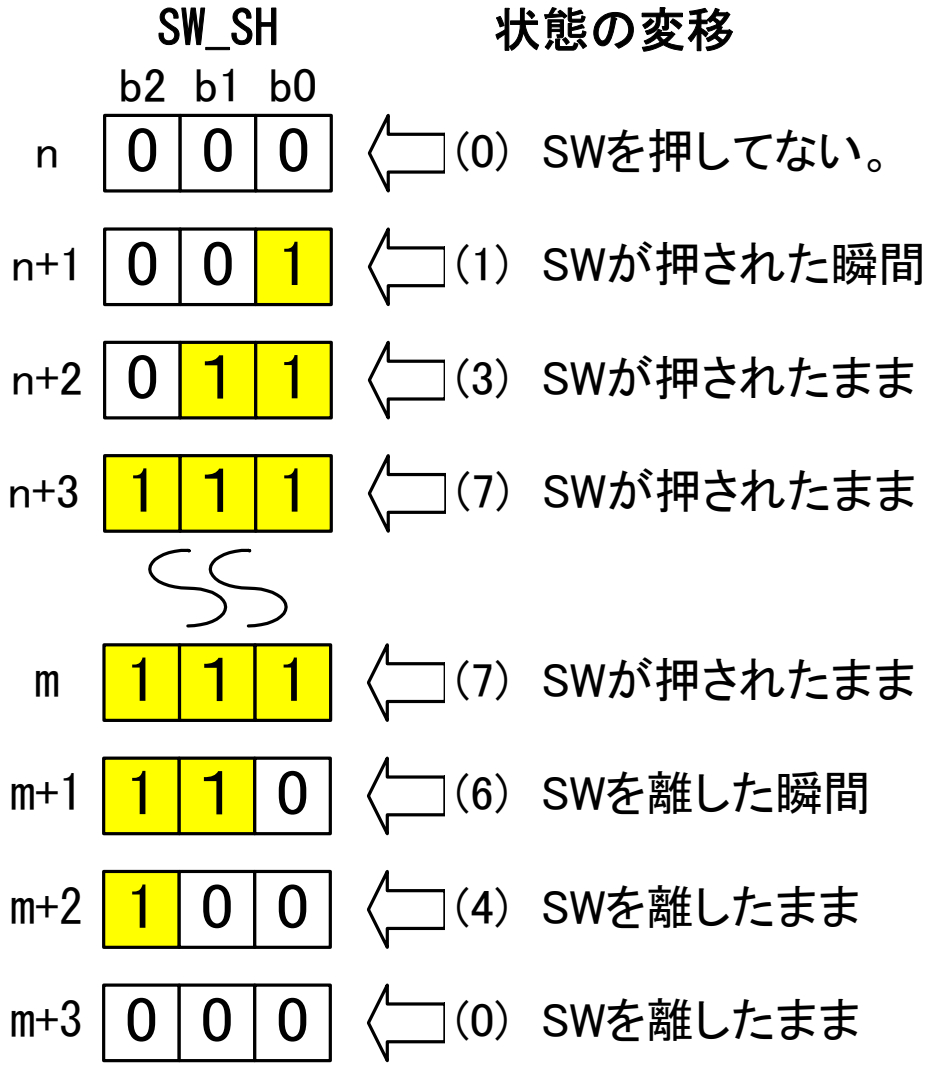
その後、開発された Object志向言語には、参照渡しという 引数の渡し方があります。

一般的に、ポインター渡しより、参照渡しの方が、null を アクセスする心配がないので安全と言われてます。残念ながら、C言語ではこのタイプの 参照渡しは、使えません。

実験による動作確認、その1

① まずは、スイッチの抵抗でPullUpされている箇所にオシロのプローブ（ch.1）を当てチャタリングの状態を観測します。
それと接点の状態を読み込んだ直後の sw_sh変数の最下位 bitの状態を オシロ（ch.2）で観測します。

② サンプルレートを遅くして
sw_sh の 変位状態（ 右の図参照 ）を ３個のLEDで 可視化します。



実験による動作確認、その2

① エッジ検出と、LEDの ワンショット
点灯（ 0.8秒 ）を確認する。

② 瞬間押しでも、長押ししても、0.8秒
しか点灯しない事を確認する。

③ スイッチを複数同時押し、しても
どれか一つしか点灯しない事を
確認する。

④ 全てのスイッチを、一旦離さないと
次の 点灯が出来ない事を確認する。