モールス信号 解読処理1　信号取込み

まず、マイコンの入力ポートから、モールス信号を、定周期で取り出す(モールス信号のサンプリング)を行う事が、最初の処理です。

マイコンで 定周期のタイミングを作りだすには タイマー割り込みを使用します。 今回は、元々R8Cマイコン用に作成していたタイマー処理 R8CM1_IOCS_TIMER.a30 アセンブラソースに インクルードファイルの形でモールス信号取り込みの ソースファイル morse_decipherment.inc を 作成しました。アセンブラの場合、変数領域は、ソースの終りの方に、データセクションという セクション宣言の場所に 変数を宣言します。 セクション宣言とは、プログラムの code を フラッシュROM領域に配置したり、data を RAM領域に配置するためのリンカに渡されるメモリ配置の宣言です。　最初から ちょっと難しい話になってしまい、ごめんなさい。

R8CM1_IOCS_TIMER.a30 内
タイマー割り込み エントリー部分

```
    btst    trbif_trbir    ; 割り込み要求フラグ確認
    jz      q_rb2_exit     ; タイマーRB2割り込みでない
    bclr    trbif_trbir    ; 割り込み要求フラグクリア

    .include  morse_decipherment.inc
                           ; モールス受信解読処理 呼び出し
```

morse_decipherment.inc 内　先頭部分

```
md_phase_1:                ; 解読処理／第１段階
    btst    0, md_swf      ; 停止、開始スイッチ確認
    jz      md_exit        ; 停止時は、即終了

    dec.b   md_ctr         ; md_ctr <= md_ctr - 1
    jnz     md_exit        ; Zeroで なければ Exit
; 5回に１回降りてくる ( 1/5 分周 )
    mov.b    #5, md_ctr    ; ダウンカウント5を再設定
```

R8CM1_IOCS_TIMER.a30 のタイマー割り込みは、1ms周期で回っているので、morse_decipherment.inc 内先頭部分で、1/5分周(5ms 200Hz)に している。

R8CM1_IOCS_TIMER.a30 内　最後の部分

```
    .section    bss_NE, DATA, ALIGN
; ======================================
;     Uses : morse_decipherment.inc  data
; --------------------------------------
md_swf:   .blkb   1   ; 停止、開始スイッチ
md_ctr:   .blkb   1   ; 分周カウンタ 1/5で 5[ms]
md_sf1:   .blkb   1   ; 入力 bit 履歴シフト
md_sf2:   .blkb   1   ; フラグビット： b0=フィルタ
                      ; 処理した端子データ
md_cnt:   .blkb   1   ; bit幅カウンタ 最上位=Sflag
                      ; 下位 7bit カウント値
md_cn2:    .blkb  1   ; 保守用？
```

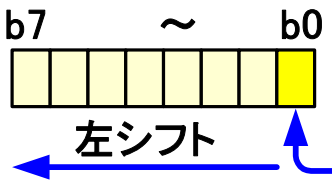
ここでは、あまり細かい部分は、理解する必要は　無いです。 組み込み用マイコンは、プログラムのコードは、ROMに配置して、変数等のデータは、RAMに 置きます。 でも固定的なデータもあります。 例えば、ステッピングモーターの 加減速テーブルは、ROMに置きます。

この場合は codeセクションに　初期値の入ったテーブルを配置します。

C言語では、セクション情報とか、リンカに渡すパラメータは、直接扱う事は出来ませんが、初期値の入ったデータテーブル等は　const という予約語を使えば ROM側に配置してくれるようです。

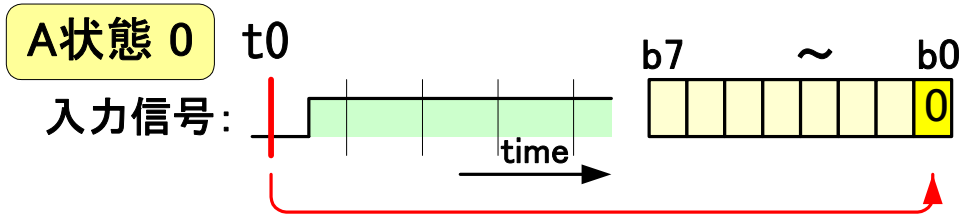
const の詳細は、各C言語のマニュアルを　参照して下さい。 あんまり基本的な話を書いていると、本題のモールス解読に行き付けませんので、このくらいにしておきます。

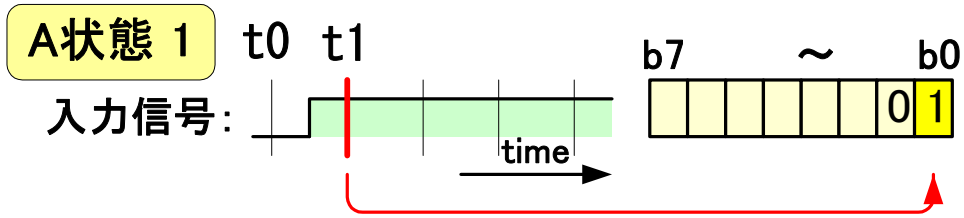
私が　今回　アセンブラの話をするのは、ビット操作処理において　C言語では出来ないような器用な処理が出来るからです。 また、C言語よりコンパクトな マシン語プログラムを生成できて処理速度も高速化出来ます。 実は、今回のモールス解読において、ビット操作処理が非常に 役に立つのです。　モールス信号の瞬時値は、1か 0 の 1bitのデータです。 それを サンプリングデータとして貯め込むと、bitデータの羅列に なります。次に 図にして説明します。

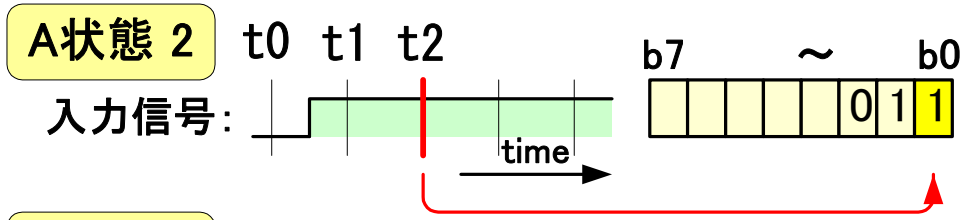


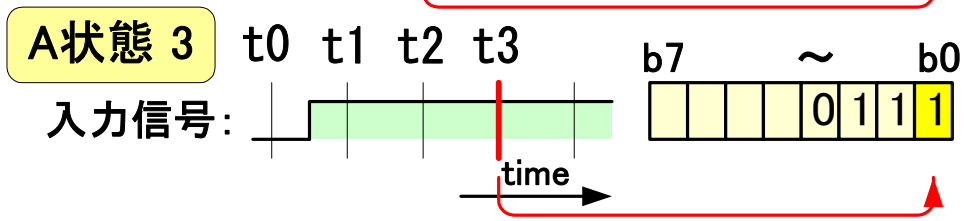
8bit の シフトレジスタを用意して、左シフトして最下位bitに、最新の bit データを入れる事にします。 サンプルレイトは、5msなので、5ms毎に、8bit レジスタを、左シフトして b0に最新の取り込んだデータを入れる事にします。

最初のステージでは、3bit同じデータが、続いたかどうかを確認するデジタルフィルターの用途に使用します。 最初、信号が、0 から 1 に変わる状態を サンプリングしたデータを、順を追って見てみたいと思います。





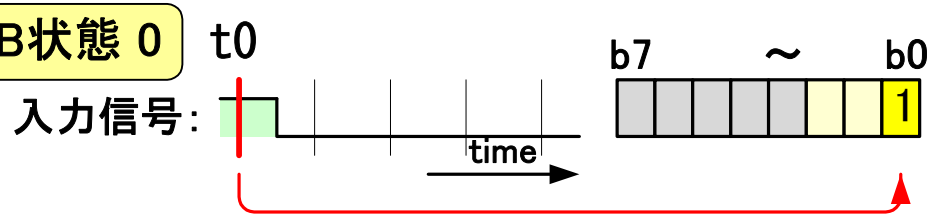




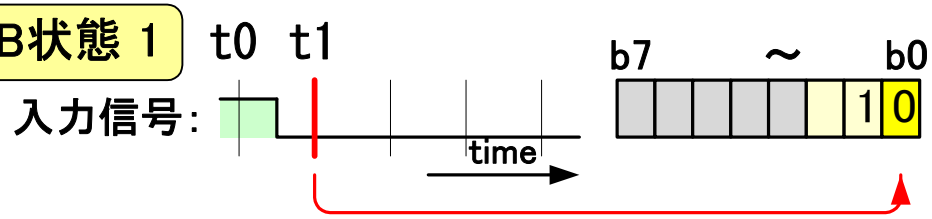
A状態 3 にて、シフトレジスタの下位 3bitが 111 に なりました。 連続した3サンプルで 111 になったので デジタルフィルター出力は 1と 確定します。 具体的な判断の仕方としては、シフトレジスタに 07h で AND を取ります。 その結果が、7で あれば 111ですから デジタルフィルタ出力は、1 とします。

今度は、信号が、1 から 0 に　変わる状態をサンプリングしたデータを、順を追って見てみたいと思います。

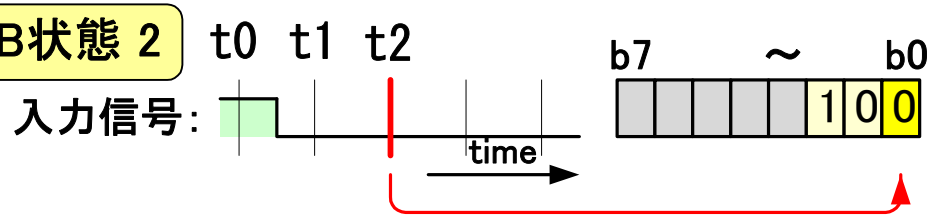
B状態 0



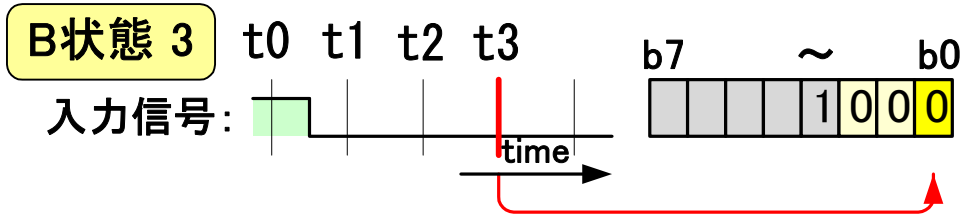
B状態 1



B状態 2



B状態 3

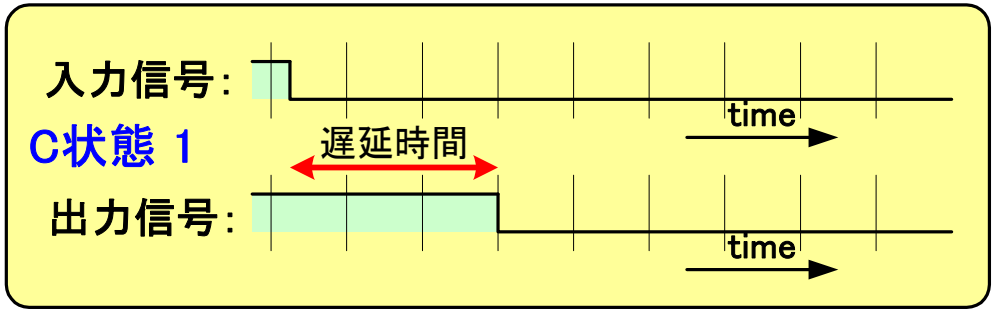


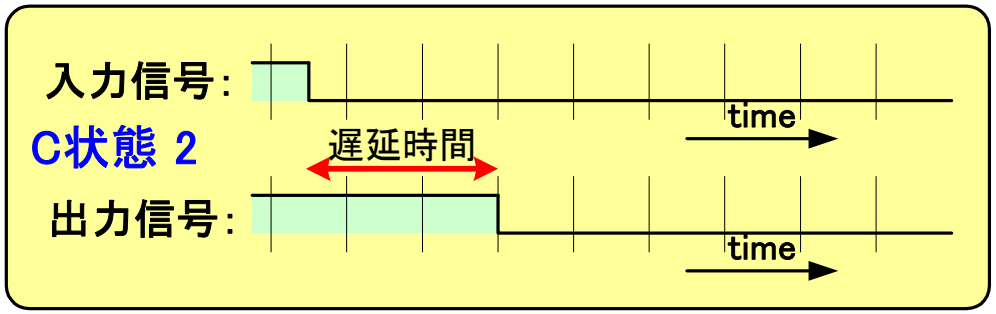
B状態 3 にて、シフトレジスタの下位 3bitが 000 に　なりました。 連続した3サンプルで　000 になったので　デジタルフィルター出力は 0 と 確定します。 具体的な判断の仕方としては、シフトレジスタに　07h で　AND を取ります。 その結果が、0で あれば　000ですから　デジタルフィルタ出力は、0　とします。 シフトレジスタの下位 3bitが 111でも　000 でも無い場合は、どうなるの。

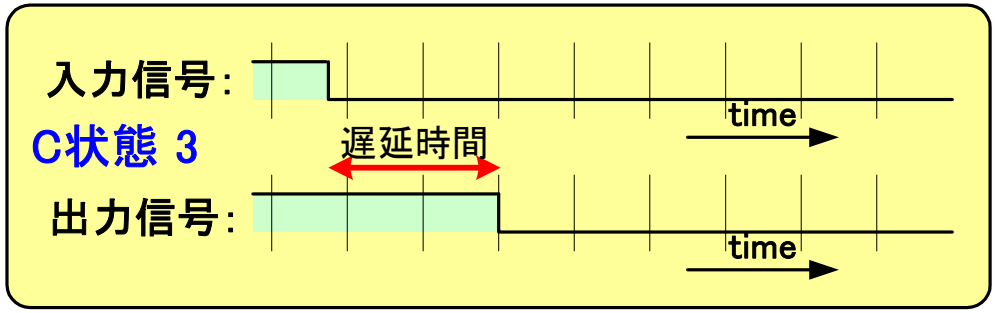
この場合は、前の出力状態を 保持します。 例えば、101 や 010 は、通常は起こり得ないパターンで、信号にノイズが、載った場合などに発生します。 この状態は、111 でも　000 でも無いので　前の出力状態を保持します。 よって細かいバタつきは 抑制するフィルタとして 動作します。

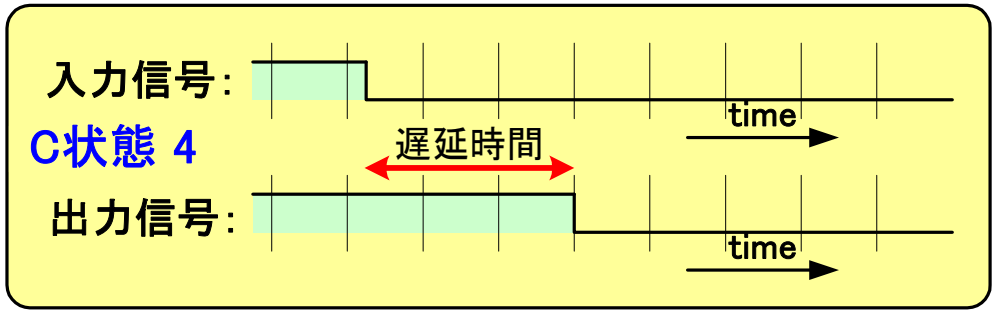
デジタルフィルターの欠点？は、信号の変化が遅延して伝送される事です。 まあ、これはアナログのローパスフィルターでも 遅れますよね。

それとデジタルフィルター特有の現象として遅延時間にジッターが生じます。 このジッターは、送り元の信号の変化時間と サンプリングクロックの 時間差というか 位相差で生じます。

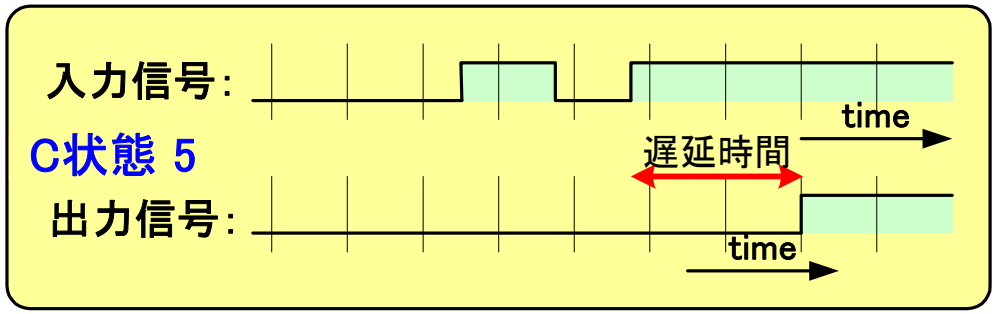








あと、入力が チャタッた場合を、書いてみます。



デジタルフィルターが、どのようなものか理解して頂けましたでしょうか。？　あと、このデジタルフィルターの処理をアセンブラで組んだプログラムをお見せします。

```
    mov.w   md_sf1, r0     ; R0 <= md_sf2 + md_sf1
    btst    p1_1           ; 端子 P1_1 を 確認
    rolc.b  r0l            ; ROLを左シフトして 最下位bitに P1_1の状態が入る
    mov.b   r0l, md_sf1    ; 変数 md_sft に 格納する bit履歴更新
    and.b   #07, r0l       ; 下位 3bitのみ 残す ( 3bit デジタルフィルタ処理 )
    jz      p201           ; Zero なら p201へ行く
    cmp.b   #07, r0l       ; ROL <> 7 か。？
    jnz     p205           ; で、なければ p205へ行く

    bset    1, md_swf      ; ★  Hi状態 確定( md_swf.b1 )
    jmp     p205           ; Phase_2 へ 行く
p201:
    bclr    1, md_swf      ; ★  Low状態 確定( md_swf.b1 )
p205:
```

アセンブラを、やった事が無い方にとっては、暗号のようなコーディングと思われるでしょうが、アセンブラをやった事がある人間にとっては、非常にシンプルな言語なのです。 人によっては原始的と表現される方もいます。一つハッキリ言えることは　使用する CPUの資源を　全て使えます。

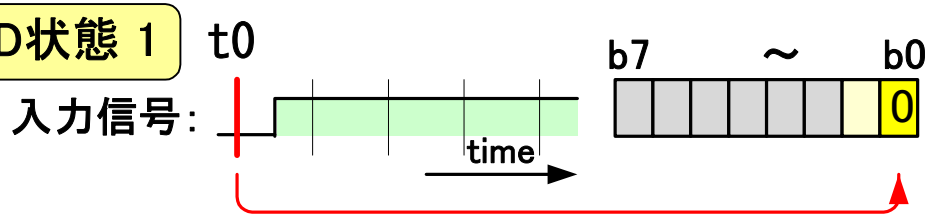
CPU資源を全て使えるという事は、CPUの性能を最大限発揮させる事も出来ます。

パソコンCPUのように超高速で、超大容量メモリを搭載しているマシンであれば、アセンブラを使う必要はありません。 逆に　低価格のローエンドマイコンにとっては、最大限の性能を発揮させる場合はアセンブラを使用する事は、有効な手段です。

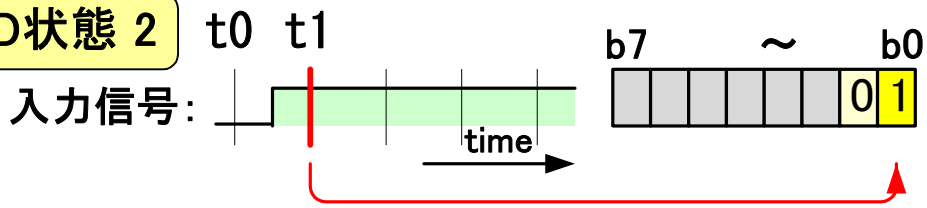
モールスパルスのエッジ検出とパルス幅測定

デジタルフィルターの出力信号を取り込み、Hi から Low または Low から Hiの 切り替えタイミングの検出 これをエッジ検出と呼んでます。 エッジが、分かれば そのエッジから 次のエッジまでの サンプル数を計ると パルス幅が、分かります。
言葉では分かりにくいのでまた、図を示します。 エッジ検出は、先ほどやったデジタルフィルターの シフトレジスタと同様の方法で、ビットの履歴を取ります。 エッジ検出の場合は、下位 2bit(言い方を変えると 最新の 2bit)の状態を、確認すれば出来ます。

D状態 1

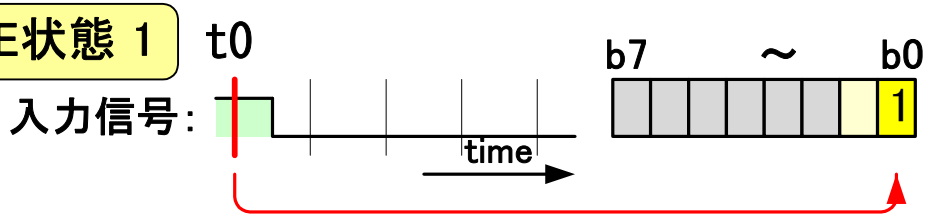


D状態 2

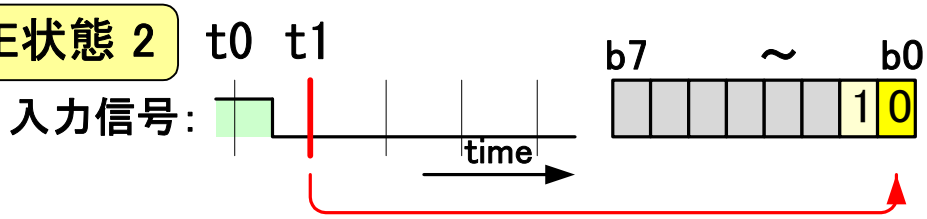


下位 2bitの信号が 01 であれば、Lowから Hi の立ち上がりエッジと 分かります。

E状態 1

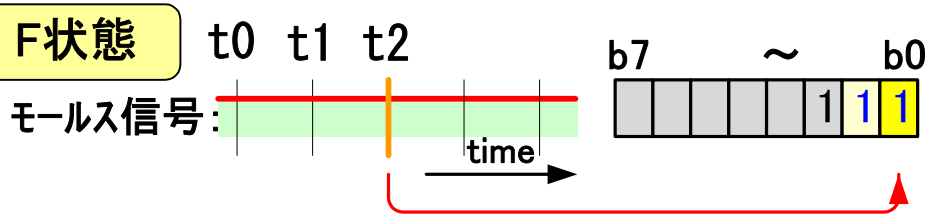


E状態 2

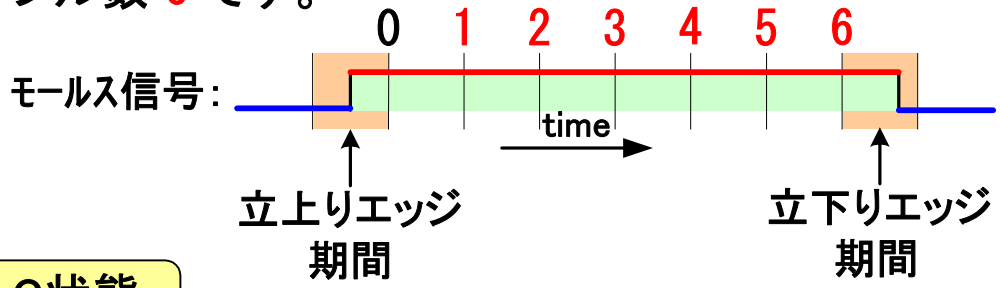


下位 2bitの信号が 10 であれば、Hi から Low の立ち下がりエッジと 分かります。

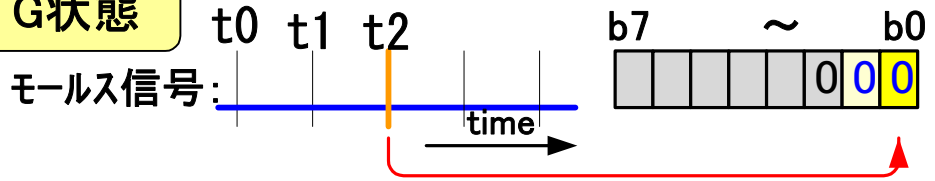
F状態



下位 2bitの信号が 11 であれば、Hi の 状態を保持している状態である事が分かります。
この短い期間は、Hi 側の パルス幅を サンプル数で カウントする期間です。 下の例では サンプル数 6 です。

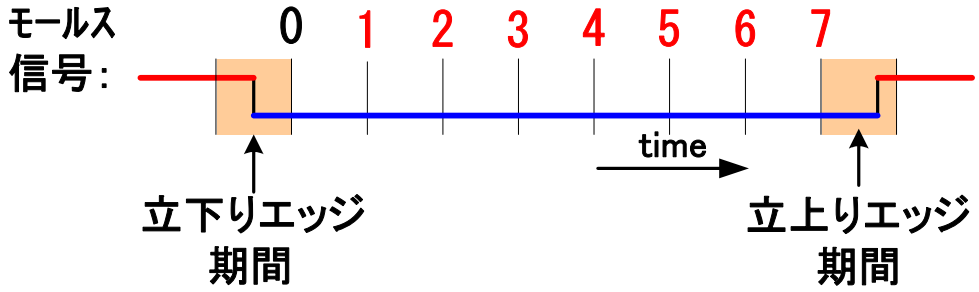


G状態



下位 2bitの信号が 00 であれば、Low の 状態を 保持している状態である事が 分かります。

この短い期間は、Low 側のパルス幅を サンプル数で カウントする期間です。
下の例では、サンプル数 7 です。



サンプル カウンタの クリアは、立ち上がり、又は 立ち下がりの エッジ検出で、カウンタに入っている値を 別変数に 移してから カウンタを ゼロ クリアしています。 因みに カウントは、Hi 側と、Low 側を 区別する必要があるので Hi 側は、初期値ゼロから ＋1して いきます。 Low側は、初期値ゼロから －1して いきます。
ちなみにカウンタ変数は、Byte整数としています。 表せる値は、-128 ～ 127 です。

Byte整数は、マイナスの値を 通常の2の補数表現で表します。 最上位 bit 7 が サインビットとなります。 アセンブラで 負の数を 正の数にする場合は、8bit レジスタの 全ビットを bit 反転して、1を足します。 要は、2の補数です。

この Byte整数が 正の値の場合 Hi 側のパルス幅カウントです。 Byte整数が 負の値の場合 Low 側の パルス幅カウントです。

で、極端に遅いモールス信号が入って来るとカウント値が、大きくなって Byte整数なのでオーバーフローする恐れがあります。

よって、正の値を インクリメントする場合は既に 127 に なっている場合は、インクリメントしないようにしています。 よって 127 で 頭打ちになります。 同様に 負の値を デクリメントする場合は、既に -128 に なっている場合は、デクリメントしないようにしています。 よって負の最大値は、-128 で 頭打ちになります。

今回、テストでやってみて、短点1、長点3の長さでは、まずオーバーする事は無いようです。

長点、短点の間の間隔、音楽で言うなら休符の部分ですが、文字内の 間隔は 1 で、文字間の間隔は、3の長さです。 語間の長さが 7 なのでこの 7 という長さが、超スローなモールスの場合、-128 の頭打ちになる事を確認しています。

-128 になっても問題は、無いようです。

で、ここまでが、タイマー割り込み処理のアセンブラ プログラムで、やっている処理です。

この後は、上記 正または 負の パルス幅 カウント値を、C言語で作成したメインループ処理に渡します。 パルスカウントした後なので、実時間処理的な シビアな要素は、だいぶ緩和されてます。 これからの処理は、モールス パルス幅を用いて まず、長点と、短点を区別する判定値の作成、更新を行います。 判定値は、短点を基準に、生成してます。

このアセンブラのリストは、R0H レジスタを シフトレジスタとして、デジタルフィルター通過後の 最新 bitデータを ビットテスト命令と、キャリーフラグを含む 左ローテート命令で、R0H の最下位 bit に 最新 bitデータを 入れ込み md_sf2変数に格納します。 次に アンド命令で R0Hの 下位 2 bit のみ残し、01の Low から Hi、10の Hi から Low、そして 11 の Hiレベルの保持を 確認します。

```
mov.b   md_sf2, r0h     ; R0H <= md_sf2 フィルター処理後のシフトレジスタ
mov.b   md_cnt, r0l     ; R0L <= md_cnt サンプルカウンタを ロード
btst    1, md_swf       ; Hi,Low確定フラグ(md_swf.b1)確認
rolc.b  r0h             ; R0H <= R0H << 1 + Cflag ( md_sf2 シフトレジスタ )
mov.b   r0h, md_sf2     ; md_sf2 <= R0H / md_sf2 更新 格納

and.b   #03h, r0h       ; 下位 2bitのみ残す
cmp.b   #01h, r0h       ; ( Low -> Hi 変化 ) R0H bitパターン 01 を 確認 Up
jz      p202            ; 一致したら p202へ

cmp.b   #02h, r0h       ; ( Hi -> Low 変化 ) R0H bitパターン 10 を 確認 Down
jz      p203             ; 一致したら p203へ

cmp.b   #03h, r0h        ; R0H bitパターン 11 の確認 ( Hiを 保持 )
jnz     p204              ; パターン11 で 無ければ p204へ
```

前ページの 01＝Up、10＝Down、11＝Hi 保持、00＝Low 保持の判定により、処理を振り分けます。11＝Hi 保持では、サンプル数のインクリメント処理、00＝Low 保持では、サンプル数のデクリメント処理を 行います。01または 10 の場合、p206へ飛びます。Byte整数の サンプル数を md_cn2変数で

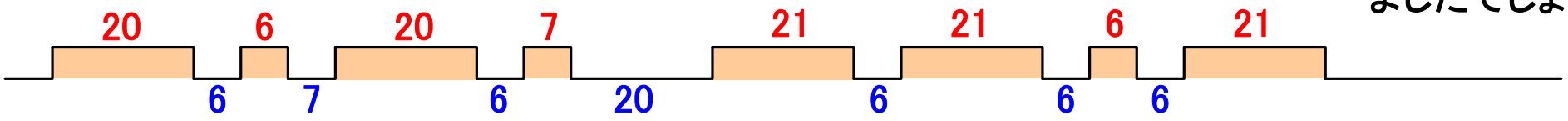
```
                ; ★ サンプルカウンタ　インクリメント処理
    cmp.b   #7Fh, r0l           ; ★　最大値 確認 リミット処理
    jz      md_exit             ; 最大値であれば Phase_2 へ
    inc.b   r0l                 ; R0L <= R0L + 1
    jmp     md_phase_2          ; Phase_2 へ

p204:       ; ★ サンプルカウンタ　デクリメント処理
    cmp.b   #80h, r0l           ; ★　R0L : マイナス最小値 確認 リミット処理
    jz      md_exit             ; マイナス最小値であれば Phase_2へ
    dec.b   r0l                 ; R0L <= R0L - 1
    jmp     md_phase_2          ; Phase_2 へ
p206:
    mov.b   r0l, md_cn2         ; サンプル数の Byte変数を、次の処理へ渡す
    mov.b   #0, r0l              ; カウント値 クリア
md_phase_2:                      ; 解読処理／第２段階
    mov.b   r0l, md_cnt         ; カウント値を md_cntへ格納する
    jmp     md_exit
```

次の処理に渡します。

そしてカウンタ変数 md_cnt をクリアして次のカウントに備えます。

高速 パルス幅 サンプルカウントの ダンプリスト

下の画像は、印刷した紙に フェルトペンで、色を付けたり書き込みを行っていますが、この数値はPが Hiレベル側、Nが Lowレベル側の カウント数の ダンプリストです。 赤線アンダーラインのところを、タイムチャートにしてみます。 何となく タイムチャートのパルス幅と 下の数表の関係が わかりましたでしょうか。



最大速度 パルス幅は 短点が、6～7、長点が、20～21、語間が 49～50 と なります。

モールス信号： 最大速度 短点：6～7，長点：20～21，語間：49～50

N128 P 11 N 6 P 6 N 6 P 6 N 21 P 21 N 6 P 20 N 7 P 20 N 21 P 6 N 6 P 6
N 7 P 6 N 49 P 6 N 7 P 6 N 6 P 6 N 21 P 20 N 7 P 20 N 6 P 21 N 21 P 6
N 6 P 6 N 6 P 7 N 49 P 6 N 6 P 7 N 6 P 6 N 21 P 20 N 6 P 21 N 6 P 21
N 20 P 7 N 6 P 6 N 6 P 6 N 46 P 20 N 6 P 6 N 7 P 20 N 6 P 7 N 20 P 21

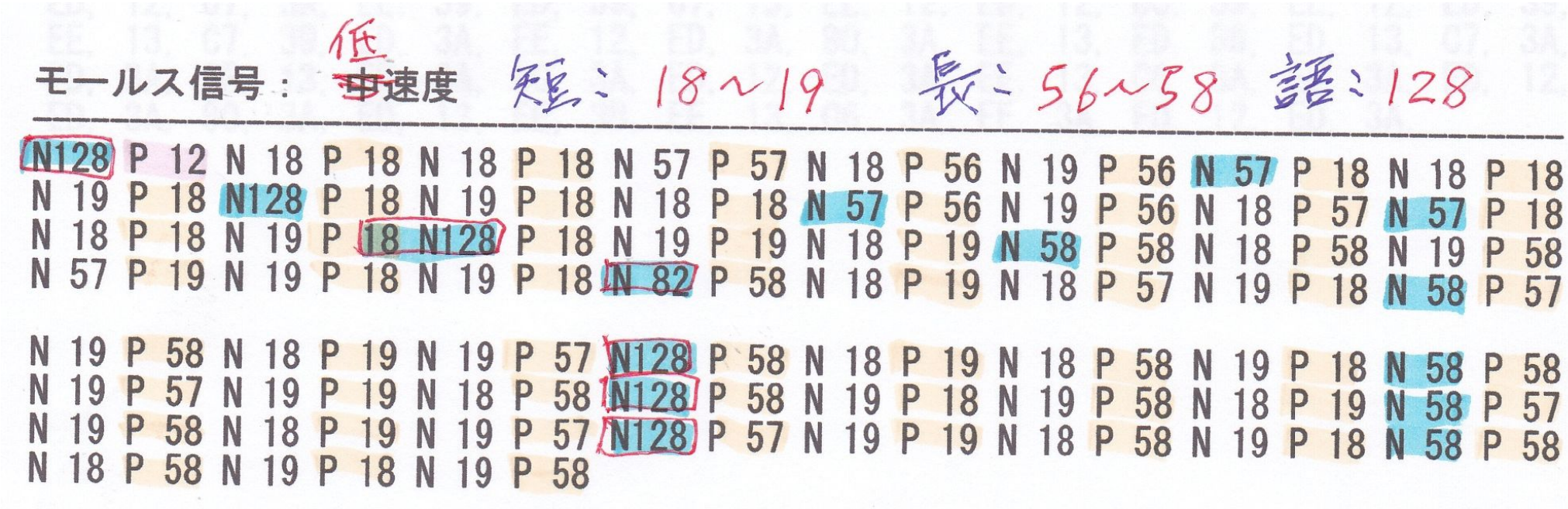
N 6 P 21 N 6 P 6 N 6 P 21 N 49 P 21 N 6 P 6 N 6 P 21 N 6 P 6 N 21 P 21
N 6 P 20 N 7 P 6 N 6 P 21 N 49 P 21 N 6 P 6 N 6 P 21 N 6 P 6 N 21 P 20
N 7 P 20 N 6 P 7 N 6 P 20 N 50 P 20 N 7 P 6 N 6 P 21 N 6 P 6 N 21 P 20
N 6 P 21 N 6 P 6 N 7 P 20

中速 パルス幅 サンプルカウントの ダンプリスト

下の画像は、印刷した紙に フェルトペンで、色を付けたり書き込みを行っていますが、この数値は Pが Hiレベル側、Nが Lowレベル側の カウント数の ダンプリストです。
中速度 パルス幅は 短点が、11～12、長点が、36～37、語間が 87～88 と なります。
(凡そ 高速の 2倍弱伸びた感じです。)

低速 パルス幅 サンプルカウントの ダンプリスト

下の画像は、印刷した紙に フェルトペンで、色を付けたり書き込みを行っていますが、この数値は Pが Hiレベル側、Nが Lowレベル側の カウント数の ダンプリストです。
低速度 パルス幅は 短点が、18～19、長点が、56～58、語間が 128 と なります。
(凡そ 高速の 3倍 伸びた感じです。、語間は 128 で、オーバーフローしているようです。)



```
//*****************************************
//**    短点と判定値の設定                 **
//**  Mpm.p1n :  短点の値                  **
//**  Mpm.d2n :  短点 長点間の 判定値      **
//**  Mpm.d5n :  長点 語間の 判定値        **
//*****************************************
void  set_dot_deci_v( BYTE c )
{
    if( c == 128 )    return;
    if( Mpm.sw == 0 )
    {   Mpm.sw = 1;   mmd_store( 'E' );  }
    else
    {
        if( c < DOT_CNT )
        {
            Mpm.p1n = c;          // 短点のサンプル数
            Mpm.d2n = c << 1;     // Mpm.d2n = C * 2
            Mpm.d5n = c << 2;     // Mpm.d2n = C * 4 ( ４倍に変更 )
            // 語区切りと 文字区切りの判定に 難があるため
            Mpm.dly = (int)c << 1 + 5;
                                  // 入力タイミングによる、遅延時間調整
            if( c > 14 )    Mpm.dly += 20;
        }
    }
}
```

次は、メインループ内で使用される set_dot_deci_v 関数です。 この関数は、短点の サンプル数を 基準にして短点１　長点３　語間7の　判定値を　設定します。

判定値は、短点1と　長点3の ちょうど中間は、2になります。 長点3と　語間7の中間は　5になりますがたまに、うまく行かないため、４ にしました。
入力タイミングによる遅延時間調整というのは、
一番 最後に処理したモールス文字が、出て来ないで内部に残留する現象が発生したため、タイマー監視で一定時間経過したら、文字を吐き出すための処理の遅延タイマー値です。

```
//*****************************************
//**   モールスパルス長 判定処理          **
//** ------------------------------------ **
//**   c :   モールス パルス幅            **
//** ------------------------------------ **
//**   関数値 :  0 = 判定不能             **
//**             1 = 短点                 **
//**             3 = 長点                 **
//**             7 = 語間の隙間           **
//*****************************************
BYTE get_morse_width( BYTE c )
{
    if( Mpm.sw == 0 )  return  0;       // 判定不能
    if( c < Mpm.d2n )  return  1;       // 短点と 判定
    else
    {
        if( c < Mpm.d5n )  return  3;  // 長点と 判定
    }
    return  7;     // 語間の隙間と 判定
}
```

get_morse_width関数です。
モールス信号のパルス幅は、モールスの速度により、パルス幅(サンプル数)が、変わります。 それを一律に短点の場合は 1、長点の場合は 3、語間の場合は 7 と置き換える処理です。

速度	高速	中速	低速	判定結果
短点	6～7	11～12	18～19	1
長点	20～21	36～37	56～58	3
語間	49～50	87～88	128	7

パルス幅のサンプル数を、1, 3、7 に カテゴライズした という事です。
こうすると、後の処理が やりやすくなります。

C側のモールス解読メイン関数の 上半分です。実際のソースから、ちょっと余分な処理は外しています。一番上の morse_pulse_width関数は、アセンブラのタイマー割り込み処理から、Byte整数のパルス幅を、取り出す関数です。
set_dot_deci_v 関数と get_morse_width 関数は、前ページで、説明した関数です。

```
c = morse_pulse_width();    // モールス信号 パルス幅検出
if( (c & 0x80) != 0 )       // 最上位 サイン itが 1 か 判定
{                           // 負の値 ( Low level )
    c = (~c) + 1;            // 負の値の場合： 絶対値を取る
    set_dot_deci_v( c );        // 短点と判定値の設定
    m = get_morse_width( c );   // モールス短点幅の値
    if( m == 3 )             // 長点幅の 隙間
    {
        mmd_send( 0 );        // 文字を パソコンに送信
    }
    if( m == 7 )          // 語間の 隙間
    {
        mmd_send( 1 );  // 文字を PCに送信後に スペース1個を送信
    }
 }
 else
```

このメイン関数前半では、Byte整数が、負の値 (モールスパルスの Low側) の処理です。
m==3 は 長点幅の隙間で、文字の終りを意味します。
M==7 は、語の 終りを意味します。 m が、3 か 7 の場合 カテゴライズした短点、長点文字列から 対応するASCIIコードを 見つけ出し パソコンに ASCII文字コードを 送信します。

送信する関数が、mmd_send関数です。

前ページの続きで C側のモールス解読メイン関数の 下半分です。 前ページは 負の数で パルスの Low側 隙間でしたが、このページの処理は、正の数で、必要となる 短点、長点の データとなります。 ここでは、モールス 1文字を 構成する 長点、短点を '1' 、'3' の 文字で 順番に 連結格納して 行きます。
例えば、モールスの C は、－・－・ で 1文字分連結した 長短文字列は、"3131" となります。
Q は、－－・－ で 1文字分連結した 長短文字列は、"3313" となります。

```
else
{                               // 正の値 ( Hi level )
    set_dot_deci_v( c );          // 短点と判定値の設定
    m = get_morse_width( c );     // モールス短点幅の値
    if( m == 0 )
    {    //  初期段階での判定を 追加
        if( c < DOT_CNT )  m = 1;
        else               m = 3;
     }
    if( m == 1 )  mmd_store( '1' );  // 短点文字 格納
    if( m == 3 )  mmd_store( '3' );  // 長点文字 格納
 }
```

この長短文字を、順次格納していく関数が、mmd_store 関数です。
1文字分 長短文字列を連結したら、mmd_send 関数で送信する事になります。

```
//*********************************************
//**  1文字を構成する 長短文字列をメモリに格納 **
//** ------------------------------------------ **
//**  c : １パルス文字                          **
//**      '1'  :  短点                          **
//**      '3'  :  長点                          **
//**      '+'  :  文字間 ３隙間                 **
//**      ' '  :  語間   ７隙間                 **
//**      'E'  :  Error 無効文字                **
//*********************************************
void mmd_store( char c )
{
     BYTE i, mx;

     mx = PULSE_MAX -1;  // 最大 格納文字数
     i = Mpm.mmc;          // 格納 位置取り出し
     Mpm.mmd[i] = c;       // 文字列配列に格納する
     if( i < mx )    i++;  // 格納位置インクリメント

     Mpm.mmc = i;          // 格納位置 格納
}
```

1文字を構成する 長短文字列をメモリに格納する関数　mmd_store 関数

中身は、構造体変数内の mmd[ ] の 配列に、長短文字を順次格納して行くだけです。例えば、1回目： '3'、2回目： '1'、3回目： '3' 、4回目： '1' で 書き込むと配列内には "3131" ツートツートで、モールスの C の文字を意味します。

```
//*******************************
//**   パルス文字列出力          **
//*******************************
void mmd_send( char sw )
{
    BYTE    i;
    char    c;

    i = Mpm.mmc;
    if( i == 0 )      return;

    Mpm.mmd[i] = 0;  // 終端に Null
    c = decode_ascii( Mpm.mmd );
// モールス短点長点文字列から ASCII文字コードに 変換

    sio_send( c );     // １文字シリアル送信
    if( sw == 1 )     sio_send( ' ' ); // スペース送信

    Mpm.mmc = 0;
}
```

長短文字列を ASCIIコードに変換して、シリアル通信で1文字送信する関数 mmd_send 関数です。

この中で、重要な関数が decode_ascii関数です。

decode_ascii関数の引数 Mpm.mmd[ ]は 文字列です。

前ページの例では C であれば "3131" の文字列が渡されます。で、引数が "3131" であれば、decode_ascii関数の 関数値は、ASCII 文字コード 'C' に なります。

この文字コード 'C' を シリアル通信でパソコンに送ります。

```
char  decode_ascii( char *tx )
{
    if( str_comp( tx, "13" ) == 1 )    return  'A';
    if( str_comp( tx, "3111" ) == 1 )  return  'B';
    if( str_comp( tx, "3131" ) == 1 )  return  'C';
    if( str_comp( tx, "311" ) == 1 )   return  'D';
    if( str_comp( tx, "1" ) == 1 )     return  'E';
    if( str_comp( tx, "1131" ) == 1 )  return  'F';
    if( str_comp( tx, "331" ) == 1 )   return  'G';
    if( str_comp( tx, "1111" ) == 1 )  return  'H';
    if( str_comp( tx, "11" ) == 1 )    return  'I';
    if( str_comp( tx, "1333" ) == 1 )  return  'J';
    if( str_comp( tx, "313" ) == 1 )   return  'K';
    if( str_comp( tx, "1311" ) == 1 )  return  'L';
    if( str_comp( tx, "33" ) == 1 )    return  'M';
    if( str_comp( tx, "31" ) == 1 )    return  'N';
    if( str_comp( tx, "333" ) == 1 )   return  'O';
    if( str_comp( tx, "1331" ) == 1 )  return  'P';
    if( str_comp( tx, "3313" ) == 1 )  return  'Q';

    // 以下　省略
```

モールスの長点短点文字列を、ASCII コードに　変換する　decode_ascii 関数です。　この関数内で 使用されている str_comp 関数は、文字列の比較関数　関数値 が　１ で　文字列一致 です。

ひたすら　長点短点文字列と　一致する文字パターンを　探して行く関数です。　一致すると　関数値が、対応する ASCII コードを　返します。　txが "3131"であれば、'C' が　返されます。　一致するパターンが　無ければ Null を　返します。

一応、ここまでで　モールス受信解読処理の 説明は　終りです。

長々と御視聴して頂き、
誠に　お疲れ様でした。