

初心者のための RL78 入門コース（第 5 回：インターバル・タイマ）

今回から、タイマ機能の説明を行います。RL78 に搭載されているタイマとしては、共通的に搭載されているタイマ・アレイ・ユニット（TAU）以外にサブクロック（XTCLK）や低速内蔵クロック（LOCO）で動作するインターバル・タイマや RTC があります。RL78/G13 には搭載されていませんが、RD78/G14 等に搭載されたタイマ Rx（TMRx）、特殊な用途で使うタイマ Kx などもありますが、ここでは TAU に絞って説明します。TAU はある程度機能を簡単にしたタイマを数多く使うようにしたもので、ユニット当たり最大 8 チャンネル、2 ユニットで最大 16 チャンネルが使用できます。

今回は、タイマの説明を進める前に、ハードウェアの拡張を行います。

今回の内容

- 14. 基板の拡張 P2-2
- 15. TAU の特徴 P2-4
- 16. インターバル・タイマとしての使い方（その 1）. P2-6
- 17. インターバル・タイマとしての使い方（その 2）. P2-10
- 18. インターバル・タイマとしての使い方（その 3）. P2-15

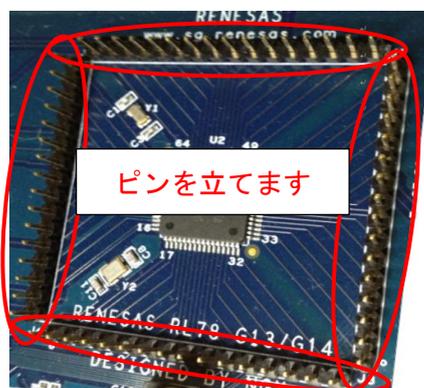
ここで使用したプロジェクトは以下のフォルダに格納されています。

「インターバル・タイマ」フォルダの構成↓	
↓	
第5回分↓	
+ RL78_G13_INTTM1_1	--- ダイナミック点灯の考え方例↓
+ RL78_G13_INTTM1_1_E1	--- ダイナミック点灯の考え方例（E1用）↓
+ RL78_G13_INTTM1_1B_E1	--- ダイナミック点灯での漏れ表示対策（E1用）↓
+ RL78_G13_INTTM1_2_E1	--- ダイナミック点灯でのカウント時間表示（E1用）↓
+ RL78_G13_INTTM1_2B_E1	--- ダイナミック点灯でのカウント時間表示の修正版（E1用）↓
+ RL78_G13_INTTM1_3_E1	--- ダイナミック点灯での60秒タイマ機能（E1用）↓

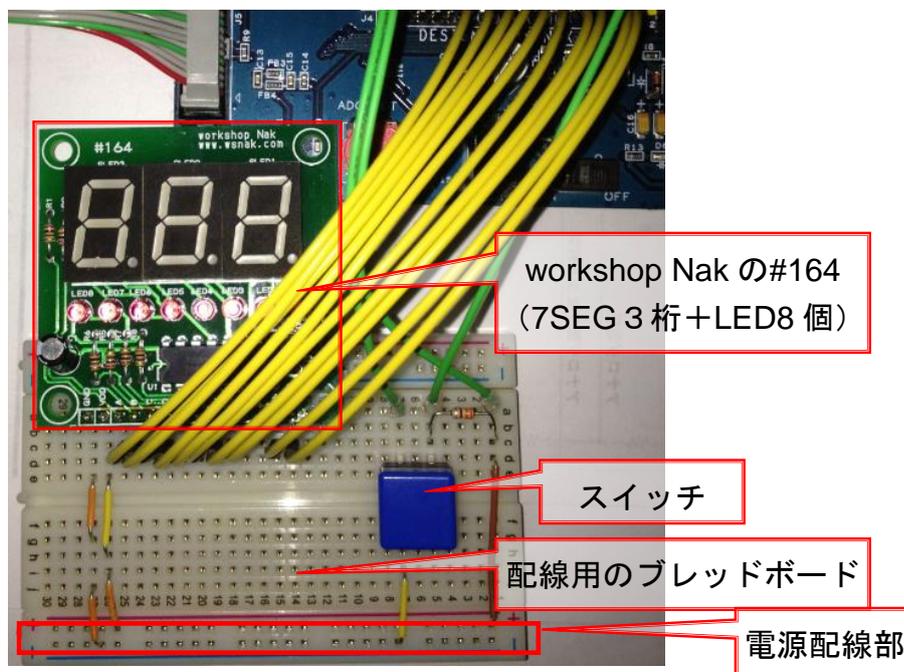
1 4. 基板の拡張

まず、RL78/G13の端子信号を引き出すために、ピンヘッダを立てます（半田付けします）。使用するピンヘッダは秋月電子の「ピンヘッダ 1×40（40P）」[PH-1x40SG/2211S-40G]を2個準備します。

ボードには、下図に示すように、2.54mmピッチで16個のスルーホールが四つ（16ピン×4=64ピン）並んでいます。ここに部品面からピンヘッダを挿入して半田付けします。



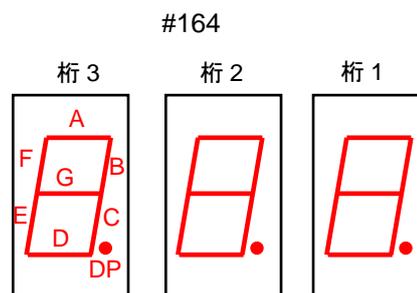
RL78/G13に接続するLED表示ボード（workshop Nakの#164ボード）は下の図に示すように、ブレッドボード（秋月電子の「ブレッドボード EIC-801」([165-40-4-8010]))差し込みます（下図参照）。この状態で、ブレッドボードの通常配線部分は上下方向が接続されているので、接続したい信号の下側の穴に、同じく秋月電子の「ブレッドボード・ジャンパーワイヤ（オスメス）15cm」のピン（オス）側を差し込みます。差し込むときに結構力が必要で、ピンが折れないように注意してください。ジャンパーワイヤのメス側を先ほど付けたピンの必要な信号と接続します。



また、電源とグラウンドは、一番下の電源配線部で配線します。

#164 ボードの 7SEG-LED の各セグメントは P7 に接続します。P70 をセグメント A, P71 をセグメント B, P72 をセグメント C, …P77 をセグメント DP と接続しておきます。

また、各桁は P10 を桁 1 の選択信号に接続し、P11 を桁 2 の選択信号、P12 を桁 3 の選択信号に接続します。



#164 ボードにはこれ以外に 8 個の LED がありますが、これらは P17 に接続することになります。

これをまとめると、以下の表のようになります。P10～P12 及び P17 のどれか 1 つを 1 にすることで、対応した桁（または LED1～LED8）の P70～P77 を 1 にしたセグメント（または LED）が点灯します。回路図を読んでもいいですが、この表を参考にポートを制御するのが簡単です。例えば、桁 3 で”1”を表示させるには、P71 と P72 を 1 にして P12 だけを 1 にします。同様に桁 1 で”8”を表示させるには、P70～P76 を 1 にして P10 だけを 1 にします。

ポート	LED 側
P70	1 でセグメント A/LED1 を点灯
P71	1 でセグメント B/LED2 を点灯
P72	1 でセグメント C/LED3 を点灯
P73	1 でセグメント D/LED4 を点灯
P74	1 でセグメント E/LED5 を点灯
P75	1 でセグメント F/LED6 を点灯
P76	1 でセグメント G/LED7 を点灯
P77	1 でセグメント DP/LED8 を点灯
P10	1 で桁 1 を点灯
P11	1 で桁 2 を点灯
P12	1 で桁 3 を点灯
P17	8 ビット LED (LED1～LED8) を点灯

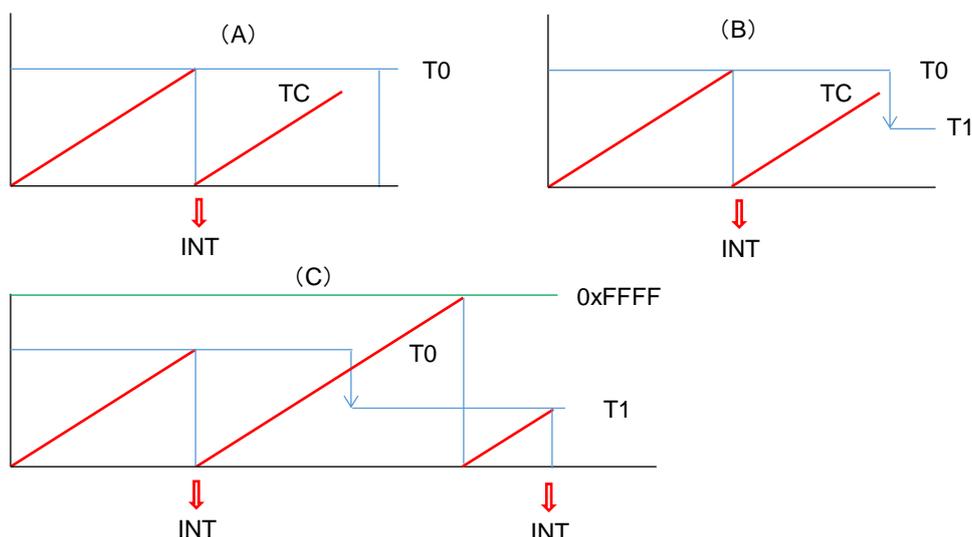
P70～P77 は直接 LED をドライブするので、同時には 1 桁だけしか点灯できません。

1 5. TAU の特徴

TAU は、比較的機能を絞ったタイマをできるだけ多く使えるようにしたものです。このために、カウンタそのものはよく見かける非同期のカウンタではなく、同期式カウンタを使用しています。また、決まった時間のカウントはカウントダウンで処理しています。このため、カウントする時間は、次の周期から反映されるようになります。これは、いつ (TDR レジスタの) 設定値を書き換えても問題がないことになります。

インターバル・タイマのように決まった時間をカウントする場合に、カウンタがカウントアップするタイマもあります。このようなタイマでは、書き換えるタイミングと書き換えるデータの組み合わせによっては時間がおかしくなるものがあります。

具体的には、現在のカウンタ値を TC、(比較する) 設定値を T0 ($TC < T0$) とします。TC が T0 と一致すると INT が発生し、カウンタ値はクリアされ、0 からカウントします (A)。設定値を T0 → T1 に変更する場合があります。このとき、新しい設定値 T1 (現在のカウンタ値より小さい) に変更する場合 (B) に、すでに、新しい設定値のカウントは過ぎているために 0xFFFF までの期間で一致が発生しなくなり、次の一致はオーバーフローして再度 0 からカウントアップし、カウンタ値が T2 と等しくなったときになります (C)。



このような問題を避けるには設定値レジスタにバッファレジスタを追加して、設定値は INT のタイミングで実際の比較レジスタに転送するにすればよく、V850 等のタイマではそのようになっているものがあるようですが、ミドルレンジから下ではこの問題が発生するようです。

RL78 では、当然ながら、ハードウェアを少なくするため、新たなハードウェアを追加することなく、カウントダウンを行うことで対応しています。

カウントダウンでは、周期を変更する場合に、1 周期遅くなることから、TAU のインターバル・タイマでは、起動時にも割り込みを発生できるようになっています。これで、例えば、20 回毎にインターバルを変更する場合には、起動時の割り込みを許可することで、20 回カウントしてインターバルを書き換えればよいことになります (RL78/G13 のインターバル・タイマの AN ではこのことを無視しているので、周期を変更した 1 回目だけ P10 に接続した LED の点灯 / 消灯時間が狂います)。

TAU の特徴である同期式は、動作するためのクロックは全て CPU の動作クロックと同じです。カウントするかしないかは、イネーブル信号で制御しているようです。

このようにすべてが同じクロックで動作することで、非同期でのアクセスがなくなり、非同期問題は無くなりました（非同期ではないので非同期の問題が無いのは当たり前ですが）。

TAUはプリスケータと16ビットのカウンタを組み合わせることで、幅広いタイミングに対応しています。しかし、プリスケータは、チャンネル数分は準備されていないということに注意しておく必要があります。16ビットのカウンタ用には2組のプリスケータが使用できるだけです。これは、RL78はロウエンドのシリーズなので、ハードをできるだけ少なくしているためと思われます。

TAUの特徴の一つが、単体の機能を単純化したことですが、タイマの機能としてPWM出力は当たり前になってきています。TAUでは、マスタ・チャンネルとスレーブ・チャンネルを組み合わせることで、単純な機能でPWM出力を実現しています。

16ビットのチャンネルを組み合わせると32ビットで使用することはできませんが、逆にいくつかのチャンネルは2個の8ビットのカウンタとして使うことができるようにはなっています。

TAU（タイマ・アレイ・ユニット）と呼ばれていますが、ハードウェアとしてはカウンタです。カウンタの中で、時間のベースとなるクロックをカウントして時間を計測するのが本当はタイマです。それでも、カウンタをタイマと呼ぶことが多いようです。ここでは、一般的な呼び方に倣ってタイマと呼んでおきます。

細々したことは、この程度にしてプログラム例に入りたいと思います。

16. インターバル・タイマとしての使い方（その1）

タイマの基本的な使い方は定周期のインターバル・タイマでしょう。インターバル・タイマの例として、ダイナミック点灯を行ってみます。

16.1 定周期タイマとしての使い方

定周期の割り込みの用途としては、「第4回：ポート入力と出力の組み合わせ」で説明したスイッチのチャタリング対策と同じくらいよく用いられるのが、表示のダイナミック点灯です。ここでは、先ほどハードウェアを拡張した7SEG-LEDと8ビットのLEDの表示器を4時分割でドライブしてみます。この場合に注意する必要があるのが、時分割の時間です。あまり遅いと、ちらつきが目に見えるようになります。また、蛍光灯の下では、電源の周波数との関係でもちらつきます。あまり、時間を短くすると、処理が重くなってしまいます。

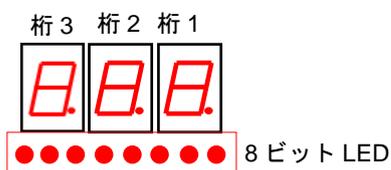
そこで、インターバル・タイマの周期を4ms（表示の周期では4倍の16ms）に設定しておきます。問題があれば、自由に変えてもらって構いません。

ダイナミック点灯

組み込みシステムでは、多桁の表示を行うことがあります。7SEGのLEDを制御するのに7個のポートが必要（ドットを含めると8個必要）です。これを4個使おうとすると、単純計算では32個のポートが必要、8個では64個のポートが必要になります。この方法で表示するのをスタティック表示と呼びます。

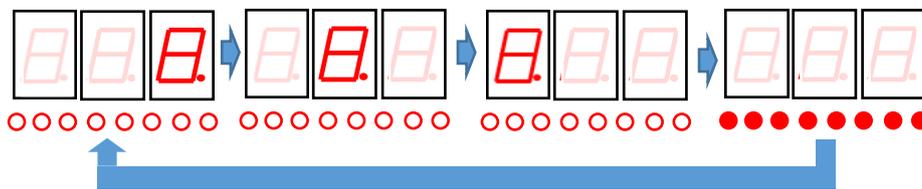
スタティック表示は単純な方法ですが、これだけのポートを準備するのは大変です。そこで使用されるのがダイナミック点灯と呼ばれる制御方法です。

下に、今回使おうとしている、#164の7SEG-LED3桁と8ビットのLEDの組み合わせを示します。



3つの7SEG-LEDと8ビットのLEDは、あるタイミングではどれか一つだけしか点灯できません。そこで、桁1を点灯 → 桁2を点灯 → 桁3を点灯 → 8個のLEDを点灯を繰り返します。これを人間の目では認識できない程度に高速に切り替えると、全てが点灯しているように見えます。何回に分けて点灯させるかを時分割と呼び、この例は4時分割になります。

時分割数を増やすと、多くの内容を表せますが、その分、個々の桁が点灯している時間は短くなり、その分暗くなってしまいます。そのため、時分割数をやみくもに増やすことはできません。



この切り替えのタイミングは、インターバル・タイマで生成します。タイマの設定値だけで簡単に変更できるので、見やすいように調整するのは簡単です。

16.1 コード生成の設定

前回作成した「RL78_G13_PORT3_4_E1」ディレクトリを「インターバル・タイマ」ディレクトリの下にコピーしておき、それを使用します。

プロジェクトを起動し、コード生成で、P1のP10~P12とP17を出力に設定します。



同じく、P7は全ビットを出力に設定します。その他のポートは使用しないに設定しておきます。

タイマは、チャンネル7をインターバル・タイマ、インターバル時間を4msに設定します。



今回はこれだけなので、 **コード生成(G)** をクリックしてコードを生成します。

16.2 r_main.c のプログラム

r_main.c では、まず、表示するデータ（バッファ）を以下のように定義しておきます。

```
/* Start user code for global. Do not edit comment generated here */↓
volatile uint8_t g_disp_data[4] = /* 表示データバッファ */↓
{↓
    0x55,↓
    0xAA,↓
    0x55,↓
    0xAA↓
};↓
/* End user code. Do not edit comment generated here */↓
```

ここで、ビット反転するデータを使用したのは、隣り合う桁のデータが桁ドライバの遅延で漏れて薄く光らないかを確認するためです（そのような場合の対策は後で示します）。

main 関数では、単純に HALT(); を無限ループするだけにしておきます。

```
void main(void)↓
{↓
    R_MAIN_UserInit();↓
    /* Start user code. Do not edit comment generated here */↓
    while (1)↓
    {↓
        HALT();           | /* HALTモードで割り込み待ち */↓
    }↓
    /* End user code. Do not edit comment generated here */↓
}↓
```

R_MAIN_UserInit 関数では、4ms のインターバル・タイマを起動するだけです。

```
void R_MAIN_UserInit(void)↓
{↓
    /* Start user code. Do not edit comment generated here */↓
    R_TAUO_Channel7_Start(); /* 4msタイマ起動 */↓
    ET();↓
    /* End user code. Do not edit comment generated here */↓
}↓
/* Start user code for adding. Do not edit comment generated here */↓
```

16.3 r_cg_timer_user.c のプログラム

最初に、使用する変数と定数を以下のように定義します。

```
/* Start user code for global. Do not edit comment generated here */↓
volatile uint8_t g_sel = 0x00; /* スキャンカウンタ */↓
extern volatile uint8_t g_disp_data[]; /* 表示データ・バッファ */↓
const uint8_t DIG1_SEL[] = /* 表示桁選択データ */↓
{↓
    0x01, /* 桁1選択 */↓
    0x02, /* 桁2選択 */↓
    0x04, /* 桁3選択 */↓
    0x80 /* LED選択 */↓
};↓
/* End user code. Do not edit comment generated here */↓
```

ここで定義した変数 `g_sel` はどの桁を選択するかを指定するためのもので、`0x00~0x03` の 4 つの値をとります。この値で、表示データを配列 `g_disp_data` から表示するデータを指定して読み出します。また、定数の配列 `DIGI_SEL` からどのような桁選択信号を出力するかを指定します。

定数配列 `DIGI_SEL` は各タイミングで出力する桁選択信号を定義したもので、ハードウェアの構成を定義しています。ここに定義された値を桁選択の P1 に出力することで、桁選択の順番が自由に変更できます。

これらを用いて、点灯制御を行うのが、下に示す `INTTM07` の割り込み処理部分です。

```
static void __near r_tau0_channel7_interrupt(void)↓
{↓
    /* Start user code. Do not edit comment generated here */↓
    ↓
    P1 &= 0b01111000; /* クリア・ディスプレイ */↓
    ↓
    P7 = g_disp_data[g_sel]; /* 表示データを設定 */↓
    P1 |= ( DIGI_SEL[g_sel] ); /* 表示桁選択 */↓
    ↓
    g_sel++; /* 表示桁の更新 */↓
    g_sel &= 0x03;↓
    ↓
    /* End user code. Do not edit comment generated here */↓
}↓
```

この処理では、最初に全ての桁を消灯するために、`P17`, `P12~P10` をクリアしています。その状態で、表示データ（バッファ）から点灯させたい桁のデータを読み出して、`P7` に設定します。その後、点灯させたい桁の選択信号を `P1` から出力します。これで、点灯処理そのものはお終いです。

最後に次の桁の準備を行って割り込み処理を完了します。

これらは `TM07` を用いたインターバル・タイマの割り込みで処理するので、上位（`main`）は点灯させたいデータを準備するだけで済みます。

これをビルドし、`E1` を介してハードウェアにダウンロードすることで、動作を確認できます。今回は、ダイナミック点灯の確認を行うだけなので、意味のあるデータではありません。（点灯状態が隣の桁に漏れていないかの確認する意味はありますが。）

最後に、プロジェクトを「`RL78_G13_INTTM1_1_E1`」と言う名前で保存して終了します（手順は省略します）。

16.4 おまけ

動作確認をシミュレータでできないかもやってみました。「インターバル・タイマ」ディレクトリの下に「`RL78_G13_INTTM1_1`」ディレクトリがそのためのプロジェクトです。確認のためにシミュレータ GUI で 7SEG-LED と 8 ビットの LED を定義し、それらを `P1` と `P7` に接続して、動作を確認しましたが、単純に実行させてもシミュレータでの GUI 処理で時間がかかり過ぎて、まともな確認はできませんでした。

16.5 おまけ 2

今回は問題ありませんでしたが、桁選択のドライバの OFF 時間が遅くて、表示が薄く漏れてしまうケースも考えられます（これは、ドライバ IC ではなく単体のトランジスタを用いた場合に起こる可能性が高くなります）。

この対策としては、P2-9 に示す INTTM07 割り込み処理で、桁の選択信号を切ってから次の桁のデータを出力するまでの時間を長くします。このためには、INTTM07 では選択信号を切るだけにして、別のインターバル・タイマを起動し、その割り込み処理の中で新しい桁の点灯処理を行います。処理が終わったら、そのタイマは停止しておきます。

具体的なプロジェクトを「RL78_G13_INTTM1_1B_E1」ディレクトリに作成しておきます。このプロジェクトでは、TM06 を 100us のインターバル・タイマに設定しておき、INTTM07 割り込み処理の中で桁選択と点灯データをクリアしてから TM06 を起動しています。INTTM06 割り込み処理では、残りの処理を行っています。参考にしてください。

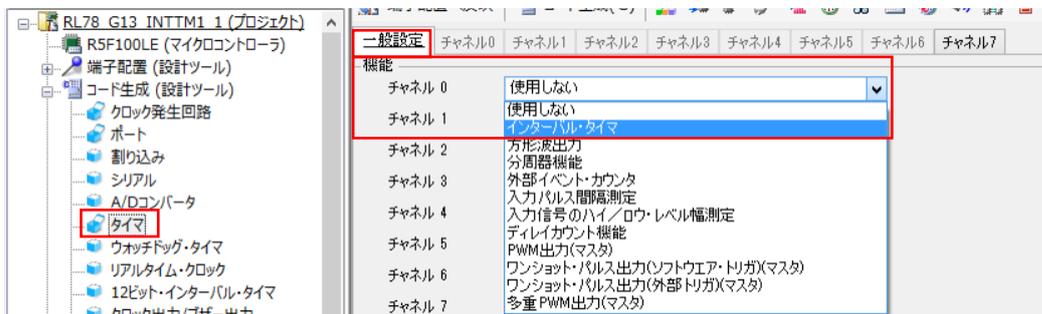
17. インターバル・タイマとしての使い方（その 2）

ダイナミック点灯をより実用的なものに変更します。具体的には、TM00 を 100ms インターバル・タイマに設定し、その中で割り込み回数をカウントアップし、カウント値（時間経過）を 7SEG-LED に表示します。

まず、「16. TAU の使い方（その 1）」で作成した「RL78_G13_INTTM1_1_E1」フォルダをコピーし、フォルダ名を「RL78_G13_INTTM1_2_E1」に変更しておきます。

17.1 定周期タイマの追加

コピーしたフォルダの「RL78_G13_INTTM1_1.mtpj」をダブルクリックして、プロジェクトを開きます。コード生成で TAU のチャンネル 0 をインターバル・タイマに設定し、インターバル時間を 100ms に設定してコード生成します。



17.2 定周期タイマのカウント

100ms の INTTM00 割り込みでは、3桁のカウントを行うことにします。処理を簡単にするために、0~9 をカウントするだけの 100ms の桁、0~99 をカウントする秒の桁に分けておきます。

```
volatile uint8_t g_t_100ms = 0x00; /* 100ms カウンタ */↓
volatile uint8_t g_t_sec = 0x00; /* 秒カウンタ */↓
```

カウントしている値は INTTM00 割り込みで表示データに変換します。変換して表示用のバッファに格納された表示データは INTTM07 割り込み処理で時分割表示を行います。

処理の前半部分を以下に示します。

```
static void __near r_tau0_channel0_interrupt(void)↓
{↓
  /* Start user code. Do not edit comment generated here */↓
  ↓
  uint8_t work;↓
  ↓
  if ( ++g_t_100ms >= 10 ) /* 1秒経過したか? */↓
  {↓
    g_t_100ms = 0x00; /* 100msをクリア */↓
    if ( ++g_t_sec >= 100 ) /* オーバーフローか */↓
    {↓
      g_t_sec = 0x00; /* 秒をクリア */↓
    }↓
  }↓
```

最初に 100ms のカウンタ (変数 g_t_100ms) をカウントアップして 1 秒経過したら、変数 g_t_100ms をクリアし、秒の桁 (変数 g_t_sec) カウントし、100 秒になったら変数 g_t_sec をクリアします。

ここまでで、カウントが完了したので、表示データを作成します。この処理を以下に示します。

```
/* 1秒の桁の表示データ設定 */↓
↓
g_disp_data[1] = CONV_7SEG[( g_t_sec % 10 )];↓
↓
/* 10秒の桁の表示データ設定 (0サプレス処理) */↓
↓
work = g_t_sec / 10; /* 10秒の桁を抽出 */↓
if ( work != 0 )↓
{ /* 10秒の桁を表示 */↓
  g_disp_data[2] = CONV_7SEG[work];↓
}↓
else↓
{ /* 0を表示しない */↓
  g_disp_data[2] = 0x00;↓
}↓
↓
}↓
```

ここで、秒の桁が変化しているので、まず、秒の桁の表示データを作成します。このために変数 `g_t_sec` を 10 で割った剰余を求め、その値に対する 7SEG-LED への出力値を表示領域（配列 `g_disp_data` の 2 番目の要素）に格納します。

次に 10 秒の桁の表示データを作成するために、変数 `work` に 10 秒の桁のデータを設定しておき、10 秒の桁が 0 なら、表示データをクリアし、0 でなければ 7SEG-LED への出力値を設定します。

最後に、100ms の桁の表示データを設定して割り込み処理を終了しています。秒が変化しない限り、秒の桁のデータは変更しないようになっています。

```
/* 100m秒の桁の表示データ設定 */↓  
↓  
g_disp_data[0] = CONV_7SEG[g_t_100ms]; /* 100ms表示*/↓  
↓
```

ここで、使用している配列 `g_disp_data` や変換テーブル `CONV_7SEG` は `r_main.c` で定義しているので、ここでは `extern` で宣言してあります。`CONV_7SEG` が今回追加された部分です。

```
extern volatile uint8_t g_disp_data[]; /* 表示データ・バッファ */↓  
extern uint8_t const CONV_7SEG[11]; /* ASCIIを7SEG変換 */↓
```

17.3 `r_main.c` での変更

`r_main.c` では、データ（0x00~0x09）を 7SEG-LED の表示データに変換するためのテーブルを `CONV_7SEG[11]` として定義しています。

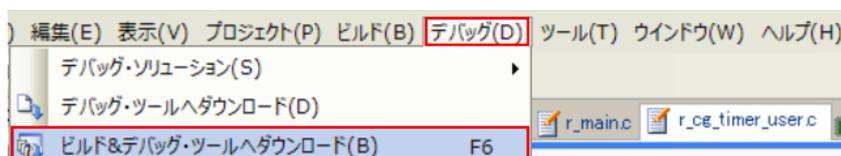
```
uint8_t const CONV_7SEG[11] =↓  
{ ↓  
  0b00111111, /* "0" */↓  
  0b00000110, /* "1" */↓  
  0b01011011, /* "2" */↓  
  0b01001111, /* "3" */↓  
  0b01100110, /* "4" */↓  
  0b01101101, /* "5" */↓  
  0b01111101, /* "6" */↓  
  0b00000111, /* "7" */↓  
  0b01111111, /* "8" */↓  
  0b01100111, /* "9" */↓  
  0b00000000 /* " " */↓  
};↓
```

基本的には、10 個のメンバで十分なのですが、11 番目のメンバとして表示を消すためのデータを追加しています。場合によっては、この後に DP を点灯するデータを組み合わせたパターンを追加しても面白いかもしれません。

プログラムとしては、R_MAIN_UserInit 関数に 8 ビット LED を点灯するための処理と 100ms のインターバル・タイマ用の TM00 の起動処理が追加されています。

```
void R_MAIN_UserInit(void)↓  
{↓  
  /* Start user code. Do not edit comment generated here */↓  
  g_disp_data[3] = 0x00;          /* 8ビットLEDは消灯 */↓  
  R_TAU0_Channel0_Start();        /* 100msタイマ起動 */↓  
  R_TAU0_Channel7_Start();        /* 4msタイマ起動 */↓  
  EI();↓  
  /* End user code. Do not edit comment generated here */↓  
}↓
```

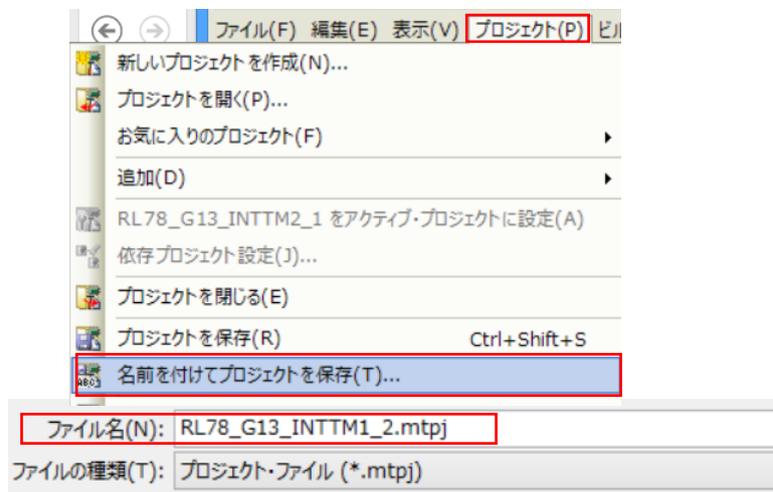
修正したファイルを保存して、ビルドし、E1 経由でダウンロードします。



 をクリックして実行すると、100ms 単位の時間表示ができます。ただ、いくつか気になる点があります。

 をクリックして実行を停止し、 をクリックしてデバッグを終了します。

、プロジェクトを「RL78_G13_INTTM1_2.mtpj」の名前で保存して、終了します。



このプログラムは、起動時に秒の表示が 0 とならず、表示なしから 1 となります。さらに、秒の桁の DP がないので不自然に感じます。そこで、これの対応を考えます。

先ほど作業した RL78_G13_INTTM1_2_E1 ディレクトリをディレクトリごとコピーして RL78_G13_INTTM1_2B_E1 に名前を変更しておきます。

17.4 表示の見直し

上で説明したプログラムは、100ms インターバル・タイマの使い方の基本的なところは動作しているのですが、表示がいま一歩でした。そこで、RL78_G13_INTTM1_2B_E1 ディレクトリを使ってブラッシュアップを行います。

起動時の秒表示の問題簡単な対策としては2つ考えられます。一つは、「秒の桁以上も毎回表示し直す」です。もう一つは「g_disp_data[1]の初期値を0x00（表示せず）から0xBF（0.を表示）に変更する」で、これは、R_MAIN_UserInit 関数で実現可能です。

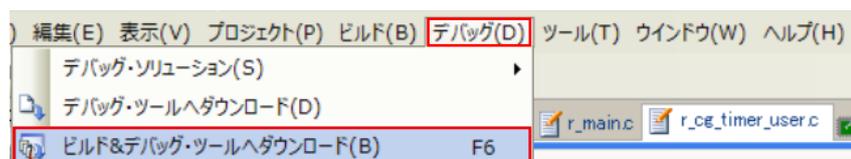
ここでは、無駄な処理が増えますが、秒の桁は毎回表示を更新するようにします。ついでに、ビット7をセットすることで、DP も表示します。

プロジェクト・ファイル「RL78_G13_INTTM1_2.mtpj」をダブルクリックしてプロジェクトを開きます。そこで、r_cg_timer_user.c ファイルを修正することにします。

具体的な修正点は、1秒の桁の表示データ設定を一番外側のif文の外に動かします。修正結果を以下に示します。右辺の最後がDPを点灯させるための処理です。

```
↓
/* 1秒の桁の表示データ設定                                     */↓
g_disp_data[1] = ( CONV_7SEG[ ( g_t_sec % 10 ) ] | 0x80 );
↓
/* 100m秒の桁の表示データ設定                                 */↓
g_disp_data[0] = CONV_7SEG[g_t_100ms]; /* 100ms表示*/↓
/* End user code. Do not edit comment generated here */↓
}↓
```

修正したファイルを保存して、ビルドし、E1 経由でダウンロードします。



 をクリックして実行します。これだけの変更で、表示はすっきりしました。

 をクリックして実行を停止し、 をクリックしてデバッグを終了

これを、RL78_G13_INTTM1_2B のプロジェクト名で保存して、終了します。

18. インターバル・タイマとしての使い方（その3）

インターバル・タイマもそろそろネタ切れになってきたので、これまでの結果を踏まえて、もう少し実用的な応用を考えてみましょう。

17では、割り込みでの変数のカウントアップだったので、今度は割り込みで変数をカウントダウンさせることで、1分のタイマを作ってみます。

RL78_G13_INTTM1_2B_E1 ディレクトリをコピーして、RL78_G13_INTTM1_3_E1 に名前を変更して、これを使用します。

18.1 ターゲットの仕様

今回のタイマの仕様を以下に示します。

・SW を押すことで、1分のタイマがスタート／カウントの一時停止／カウントを再開する。

- ①初期状態での7SEG-LEDは"60.0"を表示する。
- ②カウントをスタートすると、"60.0"から100ms毎にカウントダウンする。
- ③カウント中にSWを押すと、その時点でカウントを停止し、時間表示の点灯と消灯を0.5秒ごとに繰り返す。

④カウント停止中にSWを押すとカウントを再開する。

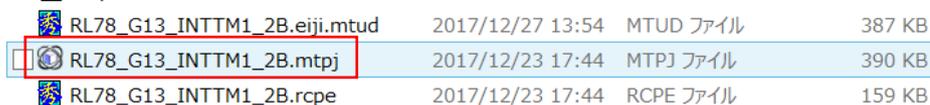
⑤1分経過したら、"00.0"を点灯と消灯を0.5秒ごとに繰り返す。

⑥この状態で、SWを押すと点滅を終了し、①に戻り、7SEG-LEDに"60.0"を表示する。

SWの入力制御には、第4回の「11. スイッチ 入力のチャタリング とノイズ」で作成したプログラムを使用します。ただし、使用するタイマのチャンネル（チャンネル0→チャンネル1）とスイッチの入力ポート（P50→P16）を変更して使用します。

18.2 コード生成

上でコピーして、名前を変更しておいた「RL78_G13_INTTM1_3_E1」フォルダの中のプロジェクト・ファイル（RL78_G13_INTTM1_2B.mtpj）をダブルクリックしてプロジェクトを開きます。

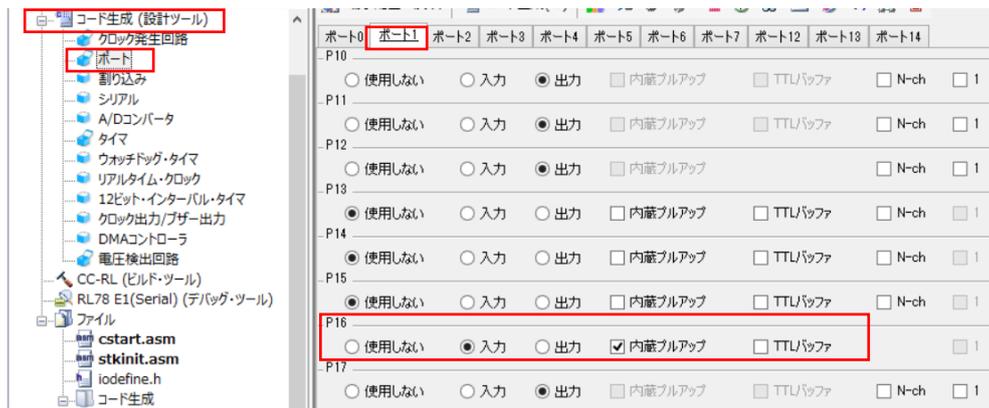


RL78_G13_INTTM1_2B.eiji.mtud	2017/12/27 13:54	MTUD ファイル	387 KB
RL78_G13_INTTM1_2B.mtpj	2017/12/23 17:44	MTPJ ファイル	390 KB
RL78_G13_INTTM1_2B.rcpe	2017/12/23 17:44	RCPE ファイル	159 KB

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「ポート」を選択（ダブルクリック）して「P1」タグを開きます。

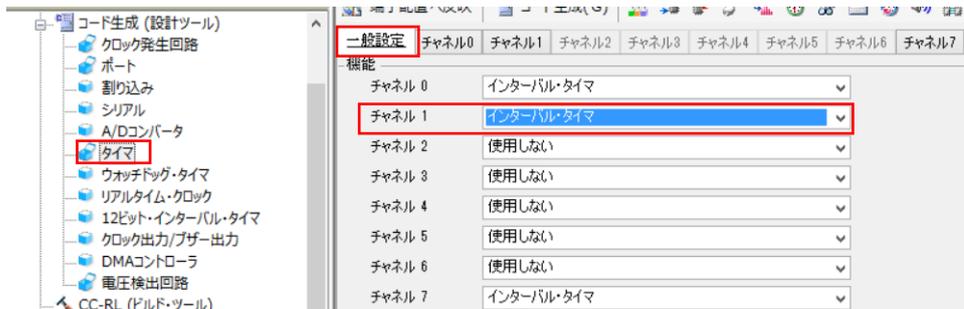
開いた状態では、P10～P12とP17が「出力」で「1」出力に設定されているので、ここで、P16を「入力」に変更し、「内蔵プルアップ抵抗」をチェックしておきます。

この状態を次の図に示します。

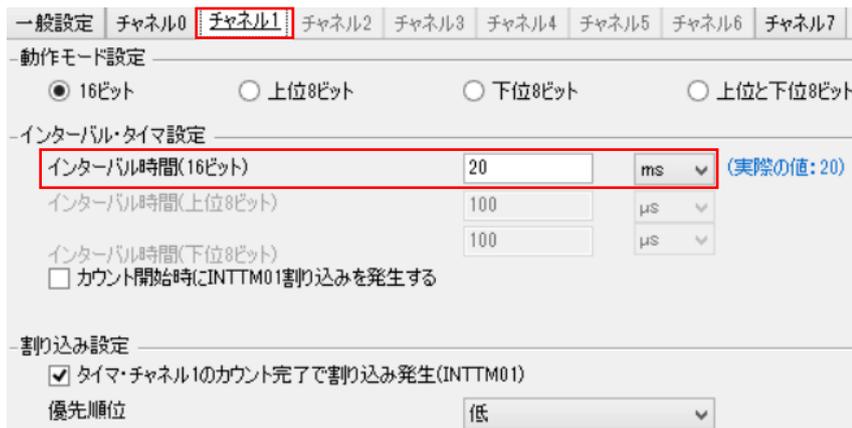


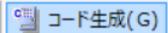
続いて、「タイマ」を選択（ダブルクリック）します。一般設定タグを選択して、「チャンネル1」を「使用しない」から「インターバル・タイマ」に変更します。

変更した状態を以下に示します。



次に、「チャンネル1」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。



コード生成での変更点は以上なので、 をクリックしてコードを生成します。

18.3 スイッチ入力プログラムの移植

まずは、スイッチ入力部分の追加を行います。最初は変数の定義の追加です。表示や時間カウント用の変数等の後ろにスイッチ入力用の変数「g_port」と「g_edge」を下図に示すように、「RL78_G13_PORT3_2」フォルダの「r_cg_timer_user.c」ファイルのグローバル変数の定義内容をコピーしておきます。

```
/******  
Global variables and functions  
*****  
/* Start user code for global. Do not edit comment generated here */  
volatile uint8_t g_sel = 0x00; /* スキャンカウンタ */  
extern volatile uint8_t g_disp_data[]; /* 表示データ・バッファ */  
extern uint8_t const CONV_7SEG[11]; /* ASCIIを7SEG変換 */  
const uint8_t DIGI_SEL[] = /* 表示桁選択データ */  
{  
    0x01, /* 桁1選択 */  
    0x02, /* 桁2選択 */  
    0x04, /* 桁3選択 */  
    0x80, /* LED選択 */  
};  
  
volatile uint8_t g_t_100ms = 0x00; /* 100ms カウンタ */  
volatile uint8_t g_t_sec = 0x00; /* 秒カウンタ */  
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 */
```

次は、下に示す「r_tau0_channel0_interrupt関数」の処理内容をコピーしてスイッチの入力のポート（赤で囲んだ部分）を変更します。

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

実際に「r_cg_tau0_channel1_intrrupt」関数に張り付けて、スイッチ入力用のポートを変更した部分を以下に示します。

```

static void __near r_tau0_channel1_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 ( P1_bit.no6 == 1 )             /* スイッチをチェック */
    {                                   /* 1なら変数のLSBを1に */
        g_port += 1;
    }

    /* 最新の3回分のサンプル値を調べ、ノイズ対策を行う */
}

```

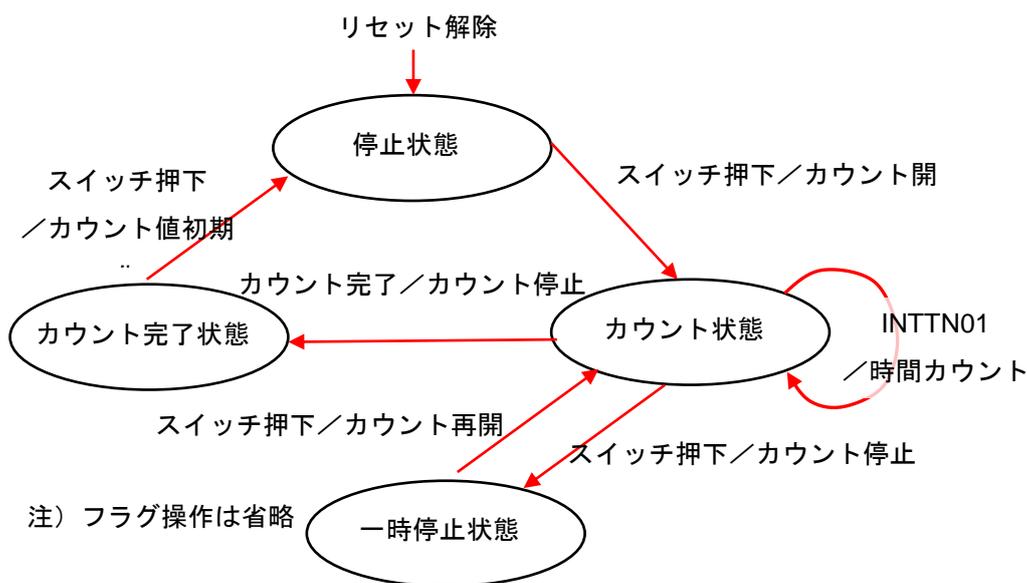
以上が、スイッチ入力の追加部分です。

18.4 時間カウント部の変更

続いて、時間カウント部分の変更を行います。今回は、常時カウントするのではなく、状態に応じてカウントを制御する必要があるため、カウンタの状態を示すフラグを準備して、フラグ「g_count_flag」の値で動作を制御することにします。

フラグの値と、状態の関係を以下のようにしておきます。また、状態遷移図を示します。

g_count_flag	状態	動作
0x00	停止状態	初期状態で起動指示待ち
0x01	カウント状態	カウント完了、停止指示待ち
0x02	一時停止状態	カウント再開指示待ち
0x03	カウント完了状態	表示を点滅させ、停止指示待ち



(このように、非同期の割り込みで処理するプログラムでは、フローチャートよりは状態遷移図の方が分かり易いと思います。)

このような制御を行うために2つのグローバル変数を定義します。

```
volatile uint8_t g_t_100ms = 0x00; /* 100ms カウンタ */
volatile uint8_t g_t_sec = 0x00; /* 秒カウンタ */
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 */
volatile uint8_t g_count_flag = 0x00; /* カウント状態フラグ
0x00 : 停止状態
0x01 : カウント状態
0x02 : 一時停止状態
0x03 : カウント完了状態
*/
volatile uint8_t g_count_flash = 0x00; /* 表示点滅用カウンタ */
/* End user code. Do not edit comment generated here */
```

変数 `g_count_flag` は上で説明した目的で使用します。変数 `g_count_flash` はカウント完了時に表示を点滅させるためのカウンタです。

それでは、100ms インターバル割り込みでのカウント動作ですが、変数 `g_count_flag` の値で処理を分けるために `switch-case` 文を使うのが簡単です。また、カウント動作はこれまでのようなカウントアップを行い、表示時に補数にするのが、一番変更が少なく済むので、手を抜いてしまいました。

時間カウント処理の先頭部分を以下に示します。ここで、ローカル変数 `rest` を定義しています。これは、カウントダウンの代わりにカウントアップしていることから、残り秒数を求めるために使用しています。

最初に変数 `g_count_flag` の値で判断分岐しています。スタート待ち状態では、変数 `g_edge` が1 (立下りエッジ検出) になるのを待ちます。スイッチが押下され、変数 `g_edge` が1になったら、変数 `g_count_flag` を `0x01` に変更し、カウント値を初期化、`g_edge` をクリアすることで、カウントを開始します。

```
static void __near r_tau0_channel0_interrupt(void)
{
    /* Start user code. Do not edit comment generated here */

    uint8_t work;
    uint8_t rest;

    switch ( g_count_flag )
    {
        case ( 0x00 ):
            /* スタート待ち状態 */
            if ( g_edge ) /* スイッチ押下? */
            {
                g_count_flag = 0x01; /* モード変更 */
                g_edge = 0x00; /* 押下フラグクリア */
                g_t_sec = 0x00; /* 秒カウンタクリア */
                g_t_100ms = 0x00; /* 100msをクリア */
            }
            break;
    }
}
```

変数 g_count_flag が 0x01 の状態で 100ms インターバルの割り込みが入ったときの処理部分を次に示します。

最初にローカル変数 rest に初期値として、59 からカウント値 (g_t_sec) を引いた値をセットしておきます。カウントが 60 秒になったなら、変数 g_count_flag を 0x03 (カウント完了) にして、0.0 を点滅させるためのカウント値に設定しています。その後、残り時間の表示を行います。

最後に、変数 g_edge でスイッチ押下をチェックして、一時停止なら、変数 g_count_flag を 0x02 します。

```
        case ( 0x01 ):
/* 時間カウント動作状態 */
    rest = 59 - g_t_sec; /* 残り秒数算出 */
    if ( ++g_t_100ms >= 10 ) /* 1秒経過したか? */
    {
        g_t_100ms = 0x00; /* 100msをクリア */
        if ( ++g_t_sec >= 60 ) /* カウント完了? */
        {
/* 時間カウント完了状態 */
            g_count_flag = 0x03;
            g_t_100ms = 9; /* 100ms表示初期化 */
            g_t_sec = 59; /* 秒表示初期化 */
        }
/* 10秒の桁の表示データ設定 (0サブレス処理) */
        rest = 59 - g_t_sec; /* 残り秒数算出 */
        work = rest / 10; /* 10秒の桁を抽出 */
        if ( work != 0 )
        { /* 10秒の桁を表示 */
            g_disp_data[2] = CONW_7SEG[work];
        }
        else
        { /* 0を表示しない */
            g_disp_data[2] = 0x00;
        }
    }
/* 1秒の桁の表示データ設定 */
    work = rest % 10;
    g_disp_data[1] = ( CONW_7SEG[ work ] | 0x80 );
/* 100m秒の桁の表示データ設定 */
    g_disp_data[0] = CONW_7SEG[ ( 9 - g_t_100ms ) ];
    if ( g_edge ) /* スイッチ押下? */
    {
        g_count_flag = 0x02; /* モード変更 */
        g_edge = 0x00; /* 押下フラグクリア */
    }
    break;
```

変数 g_count_flag が 0x02 の状態で 100ms インターバルの割り込みが入ったときの処理部分を次に示します。

ここは、単純に変数 g_edge でスイッチ押下をチェックしているだけです。スイッチが押下されていたら、変数 g_count_flag を 0x01（カウント状態）に変更します。

```

        case ( 0x02 ):
/* 時間カウント一時停止状態 */
        if ( g_edge ) /* スイッチ押下? */
        {
            g_count_flag = 0x01; /* モード変更 */
            g_edge = 0x00; /* 押下フラグクリア */
        }
        break;

```

最後に、変数 g_count_flag が 0x03 の状態で 100ms インターバルの割り込みが入ったときの処理部分を次に示します。ここでは、スイッチ押下を待ちますが、同時に変数 g_count_flash をカウントし、4 カウント（400ms）毎に、PM1 レジスタで表示を消したり、表示したりできるようにしています。こうすることで、表示制御とは独立して制御できます。

```

        default:
/* カウント完了状態 */
        if ( g_edge ) /* スイッチ押下? */
        {
            g_count_flag = 0x00; /* モード変更 */
            g_edge = 0x00; /* 押下フラグクリア */
            g_count_flash = 0x00;

            g_disp_data[2] = CONV_7SEG[6];
            g_disp_data[1] = CONV_7SEG[0] | 0x80;
            g_disp_data[0] = CONV_7SEG[0];
        }
        else
        {
            g_count_flash++; /* 点滅フラグ更新 */
        }

        if ( g_count_flash & 0x04 )
        {
            PM1 |= 0x87; /* ドライブ禁止 */
        }
        else
        {
            PM1 &= 0x78; /* ドライブ許可 */
        }
    }

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

```

処理部分は以上です。最後に、「r_main.c」ファイルの R_MAIN_UserInit 関数に、赤く囲んだ初期化処理を追加します。

```

.....
void R_MAIN_UserInit(void)
{
    /* Start user code. Do not edit comment generated here */
    g_disp_data[3] = 0x00; /* 8ビットLEDは消灯 */
    g_disp_data[2] = CONV_7SEG[6]; /* 残り秒数の初期化 */
    g_disp_data[1] = CONV_7SEG[0] | 0x80;
    g_disp_data[0] = CONV_7SEG[0];
    R_TAUO_Channel10_Start(); /* 100msタイマ起動 */
    R_TAUO_Channel11_Start(); /* 10msタイマ起動 */
    R_TAUO_Channel17_Start(); /* 4msタイマ起動 */
    EI();
    /* End user code. Do not edit comment generated here */
}

```

ここでは、初期状態で「60.0」を表示する処理と、スイッチのチャタリング防止と押下検出用のタイマの起動処理を追加しています。

ビルド後、実際にE1経路でダウンロードして、動作確認したところ、動作としては期待通りの動作になっています。

ただし、残り時間が40秒を切った辺りから、表示がちらつきだすことが安定的に発生しています。これは、おそらく表示制御用の割り込みが待たされることが原因ではないかと想像しています。

処理のロジック的にはこれ以外の問題は、今のところ確認していません。CS+が不安定になったので、ここまですておきます。

デバッグを終了し、「RL78_G13_INTTM1_3.mtpj」のプロジェクト名で保存して、CS+を終了します。

今回は、一気にプログラムの変更量が多くなってしまいました。今後は、今回のプログラムをベースにして、少しずつ手を加えていくつもりです。