7 スイッチの利用

7.1 タクトスイッチ

7.1.1 タクトスイッチとは

スイッチとは、回路の ON と OFF を切り替えるための装置です。代表的なスイッチには以下のようなものがあります。

- タクタイルスイッチ
- ディップスイッチ
- 押しボタンスイッチ
- ロッカースイッチ
- トグルスイッチ

タクタイルスイッチ (tactile switch)

タクタイルスイッチ (tactile switch) はタクトスイッチとも呼ばれます。「感触のあるスイッチ」という意味で、操作部 (プランジャ) を押し込んだときのクリック感が特徴です。操作部を押すと ON に、離すと OFF になります。この動作は「モーメンタリ」と呼ばれます。

ディップスイッチ (DIP switch)

ディップスイッチは DIP(デュアル・インライン・パッケージ) と同じ端子間距離 (ピッチ) 持つスイッチです。操作方法として、スライドタイプ、ピアノタイプ (押し下げる)、ロータリタイプがあります。ディップスイッチは機器の設定などに使い、頻繁には操作されないことが普通です。

押しボタンスイッチ (push-button switch)

押しボタンスイッチもしくは押しボタン (push-button) とは、押すことでスイッチを開閉する電子部品です。スイッチ本体はパネルに固定します。

押している間だけスイッチがオンになる自動復帰型スイッチと、押すたびにオンとオフが反転する位置保持型スイッチがあります。

ロッカースイッチ (rocker switch)

ロッカースイッチは、操作ボタンの両端をシーソーのように交互に押すことで電気回路の ON，OFF を行うスイッチです。機器の主電源を ON，OFF する用途に多く使われています。

トグルスイッチ (toggle switch)

トグルスイッチは、つまみ状の操作部 (レバー) を上下または左右の一方向に倒すことで、回路を切り替える構造のスイッチをいいます。

スライドスイッチ (slide switch)

スライドスイッチは、つまみ状の操作部スライドさせることによって、電気回路の ON，OFF を行うスイッチです。

ここでは、タクトスイッチ (タクタイルスイッチ) の利用方法を学習しましょう。

今回利用するのは、図 7.1 のようなタクトスイッチです。

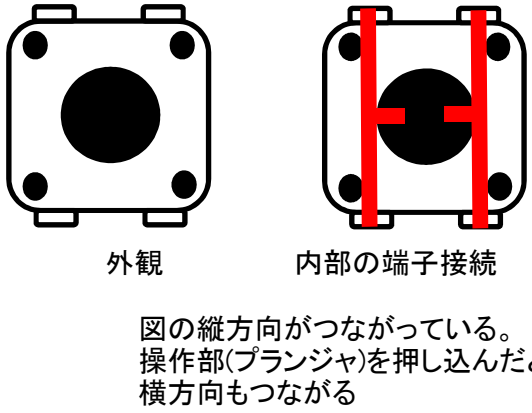


図 7.1　タクトスイッチ

図 7.1 右図のように、操作部 (プランジャ) を押し込まない状態では、図の縦方向の二つの脚がつながっています。

操作部 (プランジャ) を押し込んだときに、全ての脚がつながります。

7.1.2 回路図

ここではタクトスイッチの回路を図 7.2 のようにしました。マイコンの端子にタクトスイッチをつなぎ、グランドに落としています。

プルアップについては、マイコンの機能を用いることにします。詳細は 7.1.3 で説明します。

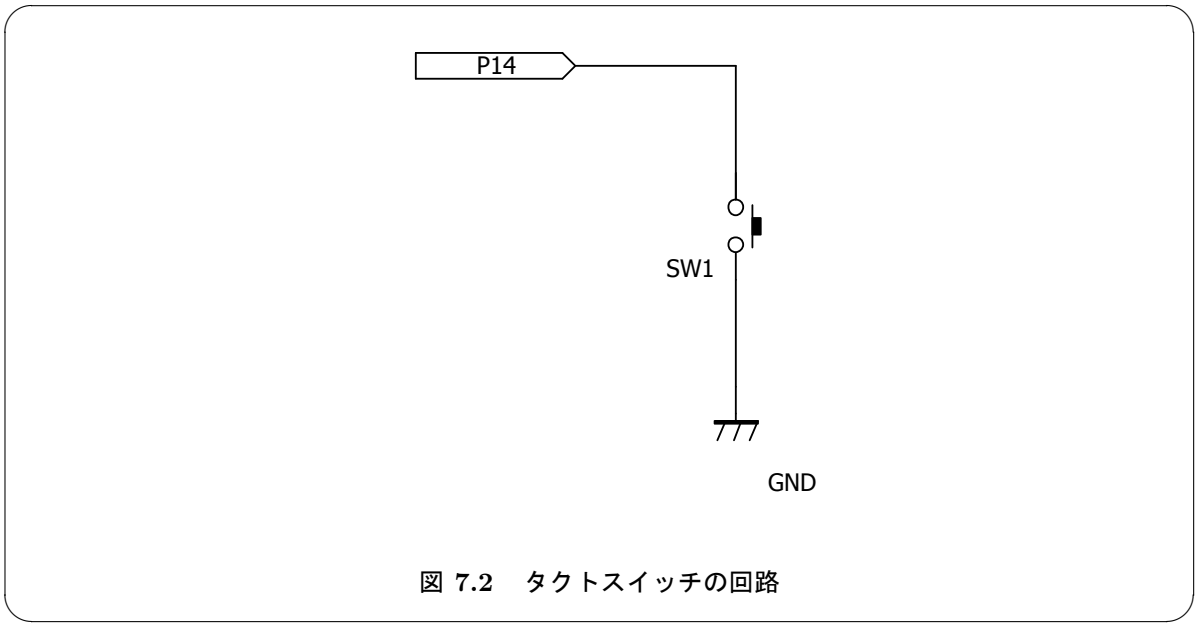


図 7.2 タクトスイッチの回路

この回路においては、タクトスイッチの ON，OFF と端子 P14 の HIGH，LOW の関係は、以下のようになります (表 7.1)。

表 7.1 スイッチの ON，OFF と端子 P14 の HIGH，LOW の関係

タクトスイッチ	P14
ON	LOW(0)
OFF	HIGH(1)

7.1.3 プルアップ抵抗 (pull-up resistor)

今考えている回路と同様に、組込みマイコンの端子にタクトスイッチをつなぎ、その先をグランドにつないだとしましょう。タクトスイッチを ON にすると、マイコンの端子はグランドにつながり LOW(0) になります。しかし、タクトスイッチを OFF にした場合にはどうでしょう。この場合には端子はどこにもつながっていないので、HIGH(1) であるとも LOW(0) であるともいえません (図 7.3)。

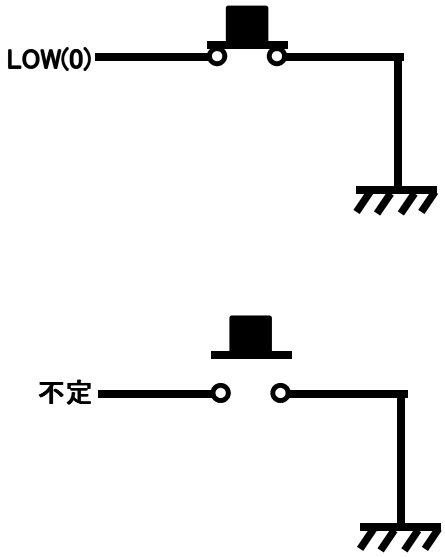


図 7.3　スイッチを押さない状態は不定になる

このような場合に、タクトスイッチが OFF のときには端子が HIGH(1) になるようにする必要があります。

このために抵抗を介して Vcc(5V) に接続します。このような回路を作ることをプルアップするといいます (図 7.4)。

抵抗を入れるのは、タクトスイッチを ON にしたときに Vcc とグランドが短絡しないようにするためです。この抵抗をプルアップ抵抗 (pull-up resistor) といいます。

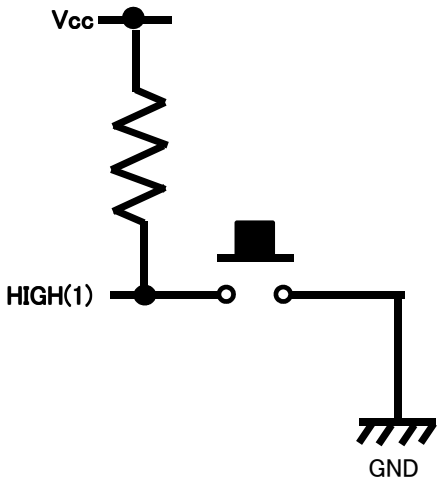


図 7.4　プルアップしてスイッチが OFF のとき端子を HIGH(1) にする

このことから、今回のタクトスイッチの回路もプルアップ抵抗をつける必要があるのですが、後で説明

するように、ポート 1 にはプルアップ機能が備わっており、レジスタを設定することでマイコン内部でプルアップ回路を実現することが可能です (図 7.5)。

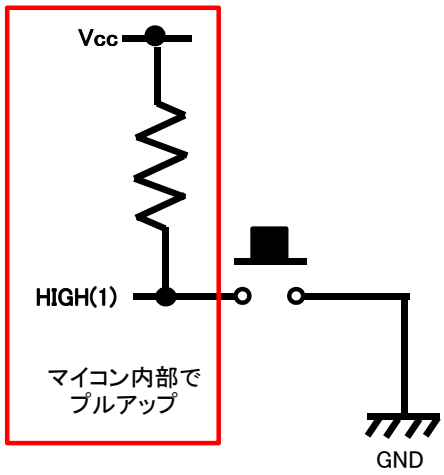


図 7.5 マイコン内部でプルアップ

7.1.4 接続例

今回は、タクトスイッチ以外に、前回の LED の回路も必要です。

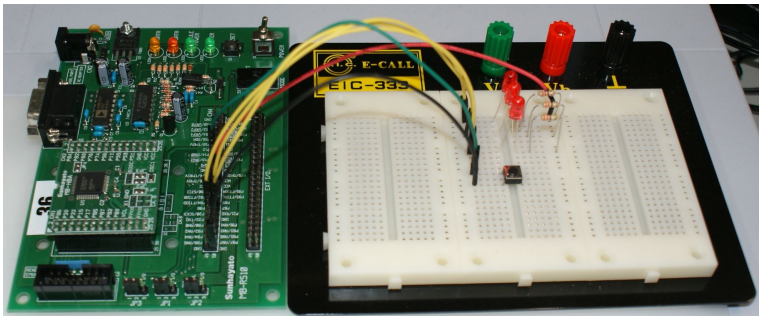


図 7.6 ブレッドボードへの接続例

7.1.5 関連レジスタ

上記回路において、スイッチの操作に必要なレジスタは以下のとおりです。

- ポートモードレジスタ 1(PMR1)
- ポートコントロールレジスタ 1(PCR1)
- ポートデータレジスタ 1(PDR1)

- ポートプルアップコントロールレジスタ 1(PUCR1)

上記のレジスタは次のアドレスに割り振られています。

表 7.2 P1 レジスタのアドレス

レジスタ名	アドレス
ポートモードレジスタ 1(PMR1)	H'FFE0
ポートコントロールレジスタ 1(PCR1)	H'FFE4
ポートデータレジスタ 1(PDR1)	H'FFD4
ポートプルアップコントロールレジスタ 1(PUCR1)	H'FFD0

ポートモードレジスタ 1(PMR1)

PMR1 はポート 1 とポート 2 の端子の機能を切り替えるために使います。

表 7.3 ポートモードレジスタ 1(PMR1)

ビット	ビット名	初期値	R/W	説 明
7	IRQ3	0	R/W	P17/$\overline{IRQ3}$/TRGV 端子の機能を選択します。 0 : 汎用入出力ポート 1 : $\overline{IRQ3}$ および TRGV 入力端子
6	IRQ2	0	R/W	P16/$\overline{IRQ2}$ の機能を選択します。 0 : 汎用入出力ポート 1 : $\overline{IRQ2}$ 入力端子
5	IRQ1	0	R/W	P15/$\overline{IRQ1}$ の機能を選択します。 0 : 汎用入出力ポート 1 : $\overline{IRQ1}$ 入力端子
4	IRQ0	0	R/W	P14/$\overline{IRQ0}$ の機能を選択します。 0 : 汎用入出力ポート 1 : $\overline{IRQ0}$ 入力端子
3 2	- -	1 1	- -	リザーブビットです。リードすると常に 1 が読み出されます。
1	TXD	0	R/W	P22/TXD 端子の機能を選択します。 0 : 汎用入出力ポート 1 : TXD 出力端子
0	TMOW	0	R/W	P10/TMOW 端子の機能を選択します。 0 : 汎用入出力ポート 1 : TMOW 出力端子

初期値が汎用入出力ポートになるようになっているので、汎用入出力ポートとして利用する場合には特に設定する必要はありません。

ポートコントロールレジスタ 1(PCR1)

PCR1 は、ポート 1 の汎用入出力ポートとして使用する端子にたいして、入力もしくは出力をビットごとに選択します。

表 7.4 ポートコントロールレジスタ 1(PCR1)

ビット	ビット名	初期値	R/W	説 明
7	PCR17	0	W	ポート 1 が汎用入出力ポートに選択されているとき、
6	PCR16	0	W	このビットを 1 にセットすると対応する端子は出力ポートとなり、
5	PCR15	0	W	0 にクリアすると入力ポートとなります。
4	PCR14	0	W	ビット 3 はリザーブビットです。
3	-	-	-	
2	PCR12	0	W	
1	PCR11	0	W	
0	PCR10	0	W	

ポートデータレジスタ 1(PDR1)

PDR1 はポート 1 の汎用入出力ポートデータレジスタです。出力に設定した端子の出力データを設定します。設定した値は保持され、新たに設定しなおすまで変化しません。

入力に設定した場合、レジスタの値にかかわらず端子の状態が読み出されます。

表 7.5 ポートデータレジスタ 1

ビット	ビット名	初期値	R/W	説 明
7	P17	0	R/W	汎用出力ポートの出力値を格納します。
6	P16	0	R/W	このレジスタをリードすると、
5	P15	0	R/W	PCR1 が 1 のビットはこのレジスタの値が読み出されます。
4	P14	0	R/W	PCR1 がの 0 ビットはこのレジスタの値にかかわらず
3	-	-	-	端子の状態が読み出されます。
2	P12	0	R/W	ビット 3 はリザーブビットです。
1	P11	0	R/W	リードすると常に 1 が読み出されます。
0	P10	0	R/W	

ポートプルアップコントロールレジスタ 1(PUCR1)

PUCR1 は入力ポートに設定された端子のプルアップ MOS をビットごとに制御します。

表 7.6 ポートプルアップコントロールレジスタ 1(PUCR1)

ビット	ビット名	初期値	R/W	説 明
7	PUCR17	0	W	PCR1 が 0 に設定されているビットのみ有効。
6	PUCR16	0	W	1 をセットすると対応する P17〜P14、P12〜P10 端子の
5	PUCR15	0	W	プルアップ MOS がオン状態となり、
4	PUCR14	0	W	0 にクリアするとオフします。
3	-	1	-	ビット 3 はリザーブビットです。
2	PUCR12	0	W	リードすると常に 1 が読み出されます。
1	PUCR11	0	W	
0	PUCR10	0	W	

7.1.6 レジスタ定義

I／Oポートのレジスタ定義は iodefine.h で行われていて、P.145 の「LED0,2 の点滅 (HEW の iodefine.h を利用)」で掲載した通りです。

ここでは、ポート 1 に関連する部分を抜き出しておきましょう。

ポート 1 のレジスタ定義 (iodefine.h の一部を抜粋)

```
struct st_io {                                          /* struct IO    */
             union {                                    /* PUCR1        */
                   unsigned char BYTE;                  /*  Byte Access */
                   struct {                             /*  Bit  Access */
                          unsigned char B7:1;           /*    Bit 7     */
                          unsigned char B6:1;           /*    Bit 6     */
                          unsigned char B5:1;           /*    Bit 5     */
                          unsigned char B4:1;           /*    Bit 4     */
                          unsigned char   :1;           /*    Bit 3     */
                          unsigned char B2:1;           /*    Bit 2     */
                          unsigned char B1:1;           /*    Bit 1     */
                          unsigned char B0:1;           /*    Bit 0     */
                          }      BIT;                   /*              */
                   }             PUCR1;                 /*              */
                     .
                     .
                     .
             union {                                    /* PDR1         */
                   unsigned char BYTE;                  /*  Byte Access */
                   struct {                             /*  Bit  Access */
                          unsigned char B7:1;           /*    Bit 7     */
                          unsigned char B6:1;           /*    Bit 6     */
                          unsigned char B5:1;           /*    Bit 5     */
                          unsigned char B4:1;           /*    Bit 4     */
                          unsigned char   :1;           /*    Bit 3     */
                          unsigned char B2:1;           /*    Bit 2     */
                          unsigned char B1:1;           /*    Bit 1     */
                          unsigned char B0:1;           /*    Bit 0     */
                          }      BIT;                   /*              */
                   }             PDR1;                  /*              */
                     .
                     .
                     .
             union {                                    /* PMR1         */
                   unsigned char BYTE;                  /*  Byte Access */
                   struct {                             /*  Bit  Access */
                          unsigned char IRQ3:1;         /*    IRQ3      */
                          unsigned char IRQ2:1;         /*    IRQ2      */
                          unsigned char IRQ1:1;         /*    IRQ1      */
                          unsigned char IRQ0:1;         /*    IRQ0      */
                          unsigned char     :2;         /*              */
                          unsigned char TXD :1;         /*    TXD       */
                          unsigned char TMOW:1;         /*    TMOW      */
                          }      BIT;                   /*              */
                   }             PMR1;                  /*              */
                     .
                     .
                     .
             unsigned char       PCR1;                  /* PCR1         */
                     .
                     .
                     .
};                                                      /*              */

#define IO      (*(volatile struct st_io    *)0xFFD0)   /* IO    Address*/
```

7.1.7 基本的なプログラム

まずは、iodefine.h を用いて、タクトスイッチの基本的なプログラムを書いてみましょう。

02_SW01.c

```
 1  /************************************************************************/
 2  /*                                                                      */
 3  /*  FILE        :02_SW01.c                                              */
 4  /*  DESCRIPTION :タクトスイッチを押したら LED0 が点灯                   */
 5  /*  CPU TYPE    :H8/3694F                                               */
 6  /*                                                                      */
 7  /*  タクトスイッチ                                                      */
 8  /*  SW0 P14                                                             */
 9  /*                                                                      */
10  /************************************************************************/
11
12  #include "led.h"
13  #include "iodefine.h"
14
15  #define SW0 0x01
16  #define SW_SHIFT 4
17
18  #define SW_MASK ( SW0 << SW_SHIFT )
19
20  void main(void)
21  {
22    LedInit();  /* LED 接続ポートの初期化 */
23    IO.PUCR1.BIT.B4 = 1;  /* プルアップを設定 */
24    IO.PCR1 &= (~SW_MASK);
25    while(IO.PDR1.BIT.B4){  /* タクトスイッチが押されるまで待つ */
26      ;
27    }
28    LedSet( LED0 );  /* LED0 点灯 */
29    while(1){
30      ;
31    }
32  }
```

End Of List

注意

LED の制御には led.h と led.c を利用しています。
P.156 の図 6.6 以降と同様の操作をしてください。
これ以降、必要に応じてこの操作を行うようにしてください。

実行結果

最初は LED はすべて消灯。
タクトスイッチを押したら LED0 が点灯する (その後はタクトスイッチを押しても何も起こらない)。

プログラム解説 (02_SW01.c)

```
12    #include "led.h"
```

LED に関しては、第 6.1 章 (P.151) で作った関数を用いることにしました。そのために、led.h を include しています。

```
15    #define SW0 0x01
16    #define SW_SHIFT 4
17
18    #define SW_MASK ( SW0 << SW_SHIFT )
```

タクトスイッチは P14 に接続されています。ここで行っている定義は、P14 に関連するレジスタを定義するのに使います。

```
24    IO.PUCR1.BIT.B4 = 1;  /* プルアップを設定 */
```

P14 のプルアップを設定しています。すでに説明したように、今利用している回路では、プルアップを外部の回路で作っていません。マイコンの機能を利用してプルアップを行うことにしました。

```
24    IO.PCR1 &= (~SW_MASK);
```

P14 を入力に設定しています。

SW_MASK は 18 行目の定義より、$(00010000)_2$ であることが分かります。したがって、ここでの演算は、$(11101111)_2$ と論理積を取ることになり、P14 のみを 0 に設定していることが分かります。

```
25    while(IO.PDR1.BIT.B4){  /* タクトスイッチが押されるまで待つ */
26      ;
27    }
```

タクトスイッチが押されるのを待つ処理です。

IO.PDR1.BIT.B4 は、タクトスイッチが OFF の場合には 1 になります。while 文は条件式が 1(0 以外) の場合にはループを繰り返します。したがって、タクトスイッチが押されてない場合には、このループを繰り返すことになります。

タクトスイッチが ON になると、IO.PDR1.BIT.B4 は 0 になるので、ループを抜けることになります。

```
28    LedSet( LED0 );  /* LED0 点灯 */
```

LED0 を点灯し、LED1 と LED2 を消灯します。

この後、無限ループになっています。したがって、LED0 を点灯させた後は何もしません。

7.1.8 チャタリング

現在タクトスイッチの回路を図 7.2(P.167) のようにしています。すでに説明したように、マイコン内部で図 7.5(P.169) のようにプルアップしています。この場合、スイッチが OFF だと端子 P14 は HIGH(1)、ON にすると LOW(0) になります。

しかし、実際には機械的な動作 (振動) を伴うために、スイッチを切り替えた直後は状態変化を繰り返します。このような現象を「チャタリング (chattering)」と言います (図 7.7)。

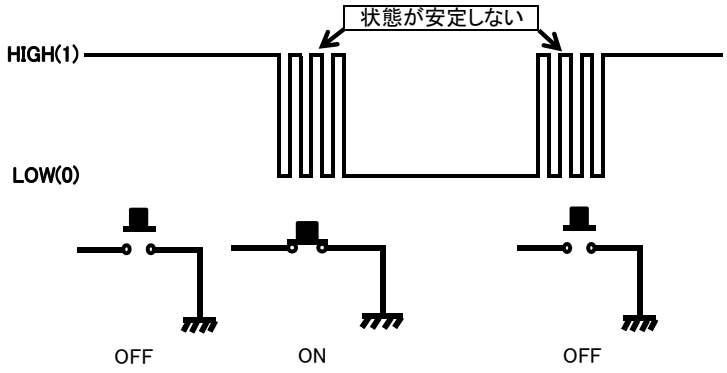


図 7.7 チャタリング (chattering)

マイコンの処理速度は非常に高速ですので、この変化を拾ってしまいます。

この現象は、スイッチの状態を何かに反映させるだけなら問題がない場合も少なくありません。たとえば、スイッチが ON の時には LED を点灯し、OFF の時には消灯するプログラムなどがそれにあたります。この場合には、チャタリングを拾って LED の点滅に反映させたとしても、人間には感知できません。そのため、特に不自然な表示にはならないわけです。

しかし、スイッチを押した回数をカウントする場合などには、チャタリングの影響で正しくカウントできないことがあります。

ここではまず、この例を確認してみましょう。

以下のプログラムは、タクトスイッチを押すごとに LED の点灯をカウントアップしていくプログラムです。タクトスイッチを押して、正しくカウントされていくかどうかを確認してみてください。

また、「タクトスイッチが押されるまで待つ」プログラムや、「タクトスイッチが離されるまで待つ」プログラムを削ってみたらどのようになるかを確認してみてください。

02_SW02.c

```
 1  /*********************************************************************/
 2  /*                                                                   */
 3  /*  FILE        :02_SW02.c                                           */
 4  /*  DESCRIPTION :タクトスイッチを押すごとに LED がカウントアップ         */
 5  /*               チャタリングに対応せず                               */
 6  /*  CPU TYPE    :H8/3694F                                            */
 7  /*                                                                   */
 8  /*  タクトスイッチ                                                    */
 9  /*  SW0 P14                                                          */
10  /*                                                                   */
11  /*********************************************************************/
12
13  #include "led.h"
14  #include "iodefine.h"
15
16  #define SW0 0x01
17  #define SW_SHIFT 4
18
19  #define SW_MASK ( SW0 << SW_SHIFT )
20  #define LED_MAX (LED0 | LED1 | LED2)
21
```

```
22  void main(void)
23  {
24    unsigned char led_data;
25
26    LedInit();  /* LED 接続ポートの初期化 */
27    IO.PUCR1.BIT.B4 = 1;  /* プルアップを設定 */
28    IO.PCR1 &= (~SW_MASK);
29    while(1){
30      for(led_data=0; led_data<=LED_MAX; led_data++){
31        while(IO.PDR1.BIT.B4){  /* タクトスイッチが押されるまで待つ */
32          ;
33        }
34        LedSet( led_data );  /* LED 点灯 */
35        while(!IO.PDR1.BIT.B4){  /* タクトスイッチが離されるまで待つ */
36          ;
37        }
38      }
39    }
40  }
```

End Of List

実行結果

最初は LED はすべて消灯。
タクトスイッチを押したら LED がカウントアップする (時々値が飛ぶ)。

プログラム解説 (02_SW02.c)

```
30    for(led_data=0; led_data<=LED_MAX; led_data++){
```

P.108 の 01_LED07.c でも同様の for 文を用いてカウントアップのプログラムを書きました。ここでは、ループの条件として LED_MAX という値を定義して利用しています。

```
31    while(IO.PDR1.BIT.B4){  /* タクトスイッチが押されるまで待つ */
32      ;
33    }
34    LedSet( led_data );  /* LED 点灯 */
35    while(!IO.PDR1.BIT.B4){  /* タクトスイッチが離されるまで待つ */
36      ;
37    }
```

31 行目から 33 行目では、タクトスイッチが押されるまで while ループの処理を繰り返します。スイッチが押されると、35 行目の LED の点滅がセットされます。35 行目から 37 行目では、条件式の前に!が付いていますので、スイッチが離されるまで while ループが繰り返されることになります。

スイッチを ON,OFF していると、ときどき LED の表示の順番が飛ぶことがわかるでしょうか？これがチャタリングによる影響です。

次に、このチャタリングの影響を少なくする方法について考えてみましょう。

7.1.9 チャタリング防止 (ソフトウェアで対処)

プログラム 02_SW02.c のサンプルで確認したチャタリングを回避するにはどうしたらよいのでしょうか。

一つには、チャタリングが起きないように、チャタリングを除去するための回路を付け加える方法が考えられます。例えば RS フリップフロップを利用すれば、チャタリングを防止する回路が構築できます。詳細は後ほど説明することにします。

ここでは、回路を変更せずに、プログラムで解決する方法を説明しましょう。

チャタリングはスイッチの切り替えが起こってから、短時間に発生する現象です。そこで、この時間だけスイッチの値を利用しないという方法をとれば、チャタリングの影響を抑えることができるはずです。

では、どのくらい待てばよいでしょうか。このあたりはスイッチの種類にもよるので一概には言えません。そこで、実際に待ち時間を作ってみて、できるだけ短い待ち時間でチャタリングの影響を防止できるように調整する必要があます (図 7.8)。チャタリングの継続時間は数百マイクロ秒 (1 万分の数秒) から数ミリ秒 (千分の数秒) 程度なので、これを目安に待ち時間を作ってみるのがよいでしょう。

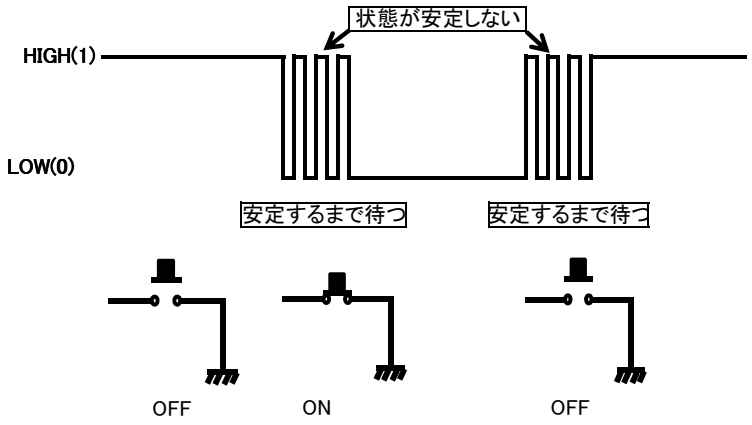


図 7.8 チャタリングの影響をソフト的に回避

もう少し確実な方法としては、スイッチの値を読んで、一定回数同じ値になるまで待つという方法もあります。

いずれもチャタリングが収まるまでの待ち時間が必要です。この待ち時間は、回路によってチャタリングを防止するよりも、長くかかることになります。

まずは、一定時間待つプログラムを作ってみましょう。

02_SW03.c

```
 1  /************************************************************************/
 2  /*                                                                      */
 3  /*  FILE        :02_SW03.c                                              */
 4  /*  DESCRIPTION :タクトスイッチを押すごとに LED がカウントアップ        */
 5  /*               チャタリングに対応                                     */
 6  /*  CPU TYPE    :H8/3694F                                               */
 7  /*                                                                      */
 8  /*  タクトスイッチ                                                      */
 9  /*  SW0 P14                                                             */
10  /*                                                                      */
11  /************************************************************************/
12
13  #include "led.h"
14  #include "iodefine.h"
15
16  #define SW0 0x01
```

```
17  #define SW_SHIFT 4
18
19  #define SW_MASK ( SW0 << SW_SHIFT )
20  #define LED_MAX (LED0 | LED1 | LED2)
21  #define SW_WAIT_LOOP 0x6FFFFL
22
23  void SwWaitLoop(void)
24  {
25    unsigned long count;
26
27    for(count=0; count<SW_WAIT_LOOP; count++){
28      ;
29    }
30  }
31
32  void main(void)
33  {
34    unsigned char led_data;
35
36    LedInit();  /* LED 接続ポートの初期化 */
37    IO.PUCR1.BIT.B4 = 1;  /* プルアップを設定 */
38    IO.PCR1 &= (~SW_MASK);
39    while(1){
40      for(led_data=0; led_data<=LED_MAX; led_data++){
41        while(IO.PDR1.BIT.B4){  /* タクトスイッチが押されるまで待つ */
42          ;
43        }
44        SwWaitLoop();
45        LedSet( led_data );  /* LED 点灯 */
46        while(!IO.PDR1.BIT.B4){  /* タクトスイッチが離されるまで待つ */
47          ;
48        }
49        SwWaitLoop();
50      }
51    }
52  }
```

End Of List

実行結果

最初は LED はすべて消灯。
タクトスイッチを押したら LED がカウントアップする (値はほとんど飛ばない)。

プログラム解説 (02_SW03.c)

```
21    #define SW_WAIT_LOOP 0x6FFFFL
22
23    void SwWaitLoop(void)
24    {
25      unsigned long count;
26
27      for(count=0; count<SW_WAIT_LOOP; count++){
28        ;
29      }
30    }
```

チャタリング防止のために作った待ちループです。関数にしてみました。

今回の待ちループは、for 文で何もしない処理を 0x6FFFF=458751 回行います。マイコンの周波数が 20MHz ですので、数十ミリ秒程度の待ち時間になっています。

タクトスイッチのチャタリングは 1 ミリ秒以下といわれていますので、ここでの待ち時間は少々長めかもしれません。

```
41    while(IO.PDR1.BIT.B4){  /* タクトスイッチが押されるまで待つ */
```

```
42      ;
43    }
44    SwWaitLoop();
```

44 行目がチャタリング防止用待ちループです。タクトスイッチが押されると、while ループを抜け、44 行目の待ちループ関数に入ります。この間にチャタリングが収まるわけです。

```
46    while(!IO.PDR1.BIT.B4){  /* タクトスイッチが離されるまで待つ */
47      ;
48    }
49    SwWaitLoop();
```

タクトスイッチが OFF になるまで待つループの後にも、チャタリング防止関数を入れておきました。

さて、タクトスイッチの基本的なプログラムが分かったところで、関数化することを考えてみましょう。

7.1.10 関数化

タクトスイッチのプログラムができたところで、これらを関数化してみましょう。

タクトスイッチは ON、OFF の状態だけでなく、押した回数を使用することも多いので、チャタリング防止のプログラムを組んでおくとよいでしょう。

7.1.9 で用いたような、一定時間待つ方法でもよいのですが、一定回数同じ値になるまで待つという方法のほうがスイッチによる違いを気にする必要も少なく、効率的でしょう。ここではこの方法を用いて関数を作ってみることにします。

以下は今回作るタクトスイッチ用の関数の仕様です。

```
タクトスイッチ関連関数 (H8/3694F ボード)

ヘッダファイル : sw.h

SW0 P14

*****************************************************************
void SwInit(void);

    形式  : #include"sw.h"
            void SwInit(void);
    引数 :  なし
    戻り値 : なし
    解説 :  スイッチ読み取りで使用する I/O ポートを初期化する関数。
            端子をプルアップする。
            スイッチ読み取り関数を使用する前に実行すること。

*****************************************************************
unsigned char SwGet(void);

    形式  : #include"sw.h"
            unsigned char SwGet(void);
    引数 :  なし
    戻り値 : unsigned char
            タクトスイッチの状態 (0:押されていない  1:押されている)
    解説 :  タクトスイッチの状態を読み取る関数。
            チャタリング対策のため同じ値に安定するまで内部で 100 回
            読み出しと比較を行っている。10000 回数読み出しを行っても安定しなかった
            場合は最後に読み取った状態を返す。
*****************************************************************
void SwWaitPush(void);

    形式  : #include"sw.h"
            void SwWaitPush(void);
    引数 :  なし
    戻り値 : なし
    解説 : タクトスイッチが押されるまで待つ関数。
           SwGet を利用している。
*****************************************************************
void SwWaitDetach(void);

    形式  : #include"sw.h"
            void SwWaitDetach(void);
    引数 :  なし
    戻り値 : なし
    解説 : タクトスイッチが離されるまで待つ関数。
           SwGet を利用している。
```

何回同じ値になったらチャタリングが終了したとみなすかですが、ここではとりあえず 100 回としておきました。

また、値が安定しない場合、何回まで値を見て、安定しなかったときの処理をどうするかも重要です。ここでは、10000 回としました。現在のスイッチの値を前の値と比較するのに 10 ステップ程度かかるとして、10000 回で数ミリ秒程度ということになります。

値が安定しなかった場合にはそのことが分かるように特別な値を返すことも考えたのですが、使い勝手を考えると最後の値をそのまま返してしまったほうがよいのではないかと思い、そのようにしています。

チャタリングの影響をきちんと取り除きたい場合には、ソフトウェアではなく、外部回路で対策したほうがよいでしょう。

それでは、この仕様にしたがって関数を作り、タクトスイッチを押している間だけ LED がすべて点灯するプログラムを作ってみましょう。

sw.h

```
 1  #ifndef _SW_H_
 2  #define _SW_H_
 3
 4  #define SWIO  IO.PCR1
 5  #define SWDAT IO.PDR1.BYTE
 6  #define SWPU  IO.PUCR1.BYTE
 7
 8  #define SW0 0x01
 9  #define SW_SHIFT 4
10
11  #define SW_MASK ( SW0 << SW_SHIFT )
12
13  #define SW_LOOP 10000 /* タクトスイッチ確認最大回数、安定しなかったら最後の値 */
14  #define SW_COUNT 100 /* 値がこの回数同じだったら、スイッチの値とする */
15
16  void SwInit(void);
17  unsigned char SwGet(void);
18  void SwWaitPush(void);
19  void SwWaitDetach(void);
20
21  #endif
```

End Of List

sw.c

```
 1  /************************************************************************/
 2  /*                                                                      */
 3  /*  FILE        :sw.c                                                   */
 4  /*  DATE        :Tue, Nov 20, 2007                                      */
 5  /*  DESCRIPTION :スイッチ用関数                                         */
 6  /*  CPU TYPE    :H8/3694F                                               */
 7  /*                                                                      */
 8  /*  タクトスイッチ                                                      */
 9  /*  SW0    P14                                                          */
10  /*                                                                      */
11  /************************************************************************/
12
13  #include "iodefine.h"
14  #include "sw.h"
15
16  /*******************************************
17    タクトスイッチのつながった端子を初期化
18  *******************************************/
19  void SwInit(void)
20  {
21    SWIO &= (~SW_MASK);
22    SWPU |= SW_MASK; /* プルアップ */
23  }
24
25  /*******************************************
26    タクトスイッチの状態を取得
27  *******************************************/
28  unsigned char SwGet(void)
29  {
30    /* i:スイッチの値を確認した回数 */
31    /* count:スイッチの値が連続して同じだった回数 */
32    int i=0, count=0;
33
34    /* old_state:前回のスイッチの値 */
35    /* new_state:現在のスイッチの値 */
36    unsigned char new_state, old_state=255;
37
38    while((i<SW_LOOP) && (count<SW_COUNT)){
39      new_state = (SWDAT & SW_MASK);
40      if(old_state == new_state){
41        count++;
42      }else{
43        count = 0;
44        old_state = new_state;
45      }
46      i++;
47    }
48    return (((~new_state) & SW_MASK) >> SW_SHIFT);
49  }
50
51  /*******************************************
52    タクトスイッチが押されるまで待つ
53  *******************************************/
54  void SwWaitPush(void)
55  {
56    while(!SwGet()){
57      ;
58    }
59  }
60
61  /*******************************************
```

```
62    タクトスイッチが離されるまで待つ
63  ****************************************/
64  void SwWaitDetach(void)
65  {
66    while(SwGet()){
67      ;
68    }
69  }
70
```

End Of List

02_SW04.c

```
 1  /*********************************************************************/
 2  /*                                                                   */
 3  /*  FILE        :02_SW04.c                                           */
 4  /*  DESCRIPTION :スイッチを押したら LED がすべて点灯                 */
 5  /*  CPU TYPE    :H8/3694F                                            */
 6  /*                                                                   */
 7  /*  タクトスイッチ                                                   */
 8  /*  SW0 P14                                                          */
 9  /*                                                                   */
10  /*********************************************************************/
11
12  #include "led.h"
13  #include "sw.h"
14
15  void main(void)
16  {
17    LedInit();      /* LED 接続ポートの初期化 */
18    SwInit();       /* タクトスイッチ接続ポートの初期化 */
19
20    while(1){
21      SwWaitPush();  /* タクトスイッチが押されるまで待つ */
22      LedSet( LED0 | LED1 | LED2 );    /* LED 点灯 */
23      SwWaitDetach();  /* タクトスイッチが離されるまで待つ */
24      LedSet( 0 );    /* LED 消灯 */
25    }
26  }
```

End Of List

実行結果

最初は LED はすべて消灯。
タクトスイッチを押したら LED が全て点灯。離すと LED はすべて消灯。

プログラム解説 (sw.h)

```
4    #define  SWIO  IO.PCR1
5    #define  SWDAT IO.PDR1.BYTE
6    #define  SWPU  IO.PUCR1.BYTE
```

タクトスイッチに関係するレジスタの定義です。名前をつけなおしています。

もうひとつ、モードコントロールレジスタもあります。端子の切り替えに利用するのですが、デフォルト値が汎用 I/O ポートになっているので、設定しませんでした。

```
13    #define SW_LOOP 10000 /* タクトスイッチ確認最大回数、安定しなかったら最後の値 */
14    #define SW_COUNT 100 /* 値がこの回数同じだったら、スイッチの値とする */
```

タクトスイッチを確認する最大回数 10000 を SW_LOOP という名前で定義しました。

今回はスイッチの値が 100 回同じだったら、チャタリングの影響がなくなったとみなします。この値を SW_COUNT という名前で定義しました。

いずれも、この定義を変えることで簡単に数値を変更することができます。

プログラム解説 (sw.c)

```
21    SWIO &= (~SW_MASK);
22    SWPU |= SW_MASK; /* プルアップ */
```

19 行目から 23 行目までが、タクトスイッチの初期化関数です。

21 行目では P14 端子を入力に設定しています。

22 行目では P14 にプルアップの設定を行っています。すでに説明したように、プルアップは外部回路ではなく、マイコンの設定によって行いました。

```
32    int i=0, count=0;
```

変数 i はスイッチの値を確認した回数 (ループの回数) を格納するために使います。

変数 count はスイッチの値が何回連続して同じだったかを格納するために使います。スイッチの値が前回と違った場合には、値を 0 に戻します。

```
36    unsigned char new_state, old_state=255;
```

変数 old_state は前回のスイッチの値を格納しておきます。現在のスイッチの値と比較するためです。スイッチの値は 0 もしくは 0x10(16) のどちらかになります。変数 old_state の初期値は、このどちらとも異なる値でなくてはなりません。ここでは、255 という値を設定しておきました。スイッチが取り得ない値でしたら、どんな値でも問題ありません。

変数 new_state は現在のスイッチの値を入れます。

```
38    while((i<SW_LOOP) && (count<SW_COUNT)){
……
47    }
```

チャタリング対策のループです。

ループが最大回数 10000(SW_LOOP) に達するか、スイッチの値が 100 回 (SW_COUNT) 同じになるまでループを繰り返します。

```
39    new_state = (SWDAT & SW_MASK);
```

タクトスイッチの値を変数 new_state に代入しています。

スイッチが ON の場合には 0 が、OFF の場合には 0x10(16) が格納されます。

```
40    if(old_state == new_state){
41      count++;
42    }
```

スイッチの値が前回と同じだったら、変数 count の値を 1 増やします。

```
42    }else{
43      count = 0;
44      old_state = new_state;
45    }
```

スイッチの値が前回と違った場合には、変数 count の値を 0 に戻し、変数 old_state に現在の値を代入します。

```
48    return (((~new_state) & SW_MASK) >> SW_SHIFT);
```

スイッチの値を返します。

ビット演算を行って、スイッチが ON の場合には 1 を、OFF の場合には 0 を返すようにしています。

```
54    void SwWaitPush(void)
55    {
56      while(!SwGet()){
57        ;
58      }
59    }
```

タクトスイッチが押されるまで待つ関数です。

SwGet はスイッチが OFF の場合 0 を返しますから、!SwGet() の値は 1 になります。この場合 while ループを実行します。

スイッチが ON になると、!SwGet() の値は 0 になります。この場合 while ループを抜けます。

```
64    void SwWaitDetach(void)
65    {
66      while(SwGet()){
67        ;
68      }
69    }
```

タクトスイッチが離されるまで待つ関数です。

SwGet はスイッチが ON の場合 1 を返しますから、SwGet() の値は 1 になります。この場合 while ループを実行します。

スイッチが OFF になると、SwGet() の値は 0 になります。この場合 while ループを抜けます。

プログラム解説 (02_SW04.c)

```
13    #include "sw.h"
```

タクトスイッチ用の関数を利用するために、ヘッダファイル sw.h を include します。

```
18    SwInit();  /* タクトスイッチ接続ポートの初期化 */
```

タクトスイッチの初期化関数です。タクトスイッチを利用する場合には、必ず最初に実行します。

```
21    SwWaitPush();  /* タクトスイッチが押されるまで待つ */
```

タクトスイッチが押されるまで待つ関数です。タクトスイッチが押されるまで関数を抜けません。

```
23    SwWaitDetach();  /* タクトスイッチが離されるまで待つ */
```

タクトスイッチが離されるまで待つ関数です。タクトスイッチが離されるまで関数を抜けません。

✍ 課題 7.1.1 (提出) タクトスイッチを押すたびに LED の点灯が左シフトするプログラム

初期状態は LED0 のみ点灯とします。

タクトスイッチを押すたびに LED の点灯が左にシフトし、LED1、LED2 へと移ります。LED2 が点灯した状態でタクトスイッチを押すと、LED0 のみが点灯の状態に戻るようにしてください。その後は同様の動作を繰り返すようにしてください。

なお、今回は、タクトスイッチを長時間押したままにしても、一旦離してから押すまでカウントしないようにしてください。

上記の sw.h、sw.c を利用して結構です。

プロジェクト名：e02_SW04

7.2 ロータリスイッチ

7.2.1 ロータリスイッチとは

今回使うスイッチはロータリディップスイッチと呼ばれるスイッチです。

メーカーによって、ロータリーディップスイッチ、ロータリコードスイッチ、 DIP コードスイッチなど名称は様々なようです。

第 7.1 章で紹介した、タクトスイッチはプログラム実行中にスイッチを押して、動作の変更を行います。この機能を用いて、タクトスイッチを押すことによって、LED の点灯をシフトさせるプログラムなどを作りました。

ロータリスイッチは、回転型になっていて、合わせた数値に対応してバイナリデータが接点として出るようになっています。

どちらかというと機器の初期設定を変更する際などに使われることが多いようです。あるいは、プログラム実行中に変更する場合でも、頻繁に切り替えるような使い方は一般的ではないようです。チャタリングが問題になるような場面で使われることは少ないでしょう。

ロータリスイッチとして COPAL ELECTRONICS 製の S-1211A を利用してみましょう (図 7.9)。

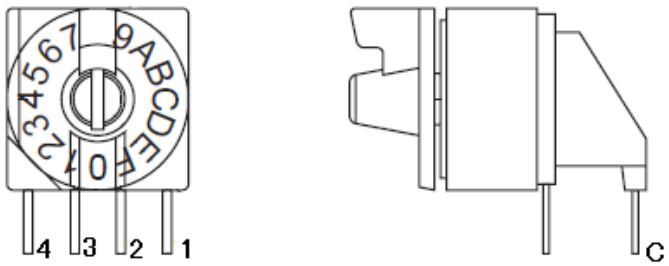


図 7.9 ロータリコードスイッチ S-1211A(マニュアルの図を抜粋)

上図のようなロータリーディップスイッチの場合、つまみを数値に合わせると、4 本の端子 1,2,3,4 と C が接続されます。この数値と接続の関係は以下のようになります (表 7.7)。

表 7.7　つまみの位置と端子の接続の関係

つまみの位置／端子	1	2	3	4
0				
1	接続			
2		接続		
3	接続	接続		
4			接続	
5	接続		接続	
6		接続	接続	
7	接続	接続	接続	
8				接続
9	接続			接続
A		接続		接続
B	接続	接続		接続
C			接続	接続
D	接続		接続	接続
E		接続	接続	接続
F	接続	接続	接続	接続

7.2.2 回路図

図 7.10 に今回利用するロータリスイッチの回路を示します。

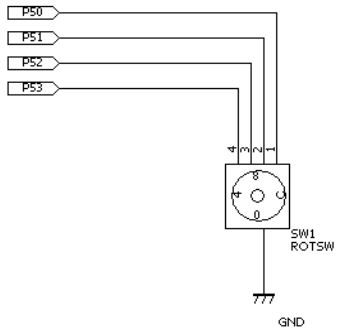


図 7.10　ロータリスイッチの回路

スイッチが接続されると 0(LOW)、接続されていないと 1(HIGH) となるように接続しています。

ここでも、マイコンの内部でプルアップをするようにしました。

ロータリスイッチのつまみとポートの 1(HIGH),0(LOW) の関係は表 7.8 のようになります。

表 7.8 つまみの位置と端子の 1(HIGH),0(LOW) の関係

つまみの位置／端子	P50	P51	P52	P53
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	1	1	0	0
4	0	0	1	0
5	1	0	1	0
6	0	1	1	0
7	1	1	1	0
8	0	0	0	1
9	1	0	0	1
A	0	1	0	1
B	1	1	0	1
C	0	0	1	1
D	1	0	1	1
E	0	1	1	1
F	1	1	1	1

7.2.3 関連レジスタ

上記回路において、スイッチの操作に必要なレジスタは以下のとおりです。

- ポートモードレジスタ 5(PMR5)
- ポートコントロールレジスタ 5(PCR5)
- ポートデータレジスタ 5(PDR5)
- ポートプルアップコントロールレジスタ 5(PUCR5)

上記のレジスタは次のアドレスに割り振られています。

表 7.9 P5 レジスタのアドレス

レジスタ名	アドレス
ポートモードレジスタ 5(PMR5)	H'FFE1
ポートコントロールレジスタ 5(PCR5)	H'FFE8
ポートデータレジスタ 5(PDR5)	H'FFD8
ポートプルアップコントロールレジスタ 5(PUCR5)	H'FFD1

ポートモードレジスタ 5(PMR5)

PMR5 はポート 5 の端子の機能を切り替えるために使います。

表 7.10 ポートモードレジスタ 5(PMR5)

ビット	ビット名	初期値	R/W	説 明
7-6	-	すべて 0	-	リザーブビットです。読み出すと常に 0 が読み出されます。
5	WKP5	0	R/W	P55/$\overline{WKP5}$/$\overline{ADTRG}$ の機能を選択します。 0 : 汎用入出力ポート 1 : $\overline{WKP5}$ 入力端子および $\overline{ADTRG}$ 入力端子
4	WKP4	0	R/W	P54/$\overline{WKP4}$ の機能を選択します。 0 : 汎用入出力ポート 1 : $\overline{WKP4}$ 入力端子
3	WKP3	0	R/W	P53/$\overline{WKP3}$ の機能を選択します。 0 : 汎用入出力ポート 1 : $\overline{WKP3}$ 入力端子
2	WKP2	0	R/W	P52/$\overline{WKP2}$ の機能を選択します。 0 : 汎用入出力ポート 1 : $\overline{WKP2}$ 入力端子
1	WKP1	0	R/W	P51/$\overline{WKP1}$ の機能を選択します。 0 : 汎用入出力ポート 1 : $\overline{WKP1}$ 入力端子
0	WKP0	0	R/W	P50/$\overline{WKP0}$ の機能を選択します。 0 : 汎用入出力ポート 1 : $\overline{WKP0}$ 入力端子

初期値が汎用入出力ポートになるようになっているので、汎用入出力ポートとして利用する場合には特に設定する必要はありません。

ポートコントロールレジスタ 5(PCR5)

PCR5 は、ポート 5 の汎用入出力ポートとして使用する端子にたいして、入力もしくは出力をビットごとに選択します。

表 7.11 ポートコントロールレジスタ 5(PCR5)

ビット	ビット名	初期値	R/W	説 明
7	PCR57	0	W	ポート 5 が汎用入出力ポートに選択されているとき、 このビットを 1 にセットすると対応する端子は出力ポートとなり、 0 にクリアすると入力ポートとなります。
6	PCR56	0	W	
5	PCR55	0	W	
4	PCR54	0	W	
3	PCR53	0	W	
2	PCR52	0	W	
1	PCR51	0	W	
0	PCR50	0	W	

ポートデータレジスタ 5(PDR5)

PDR5 はポート 5 の汎用入出力ポートデータレジスタです。出力に設定した端子の出力データを設定します。設定した値は保持され、新たに設定しなおすまで変化しません。

表 7.12 ポートデータレジスタ 5

ビット	ビット名	初期値	R/W	説 明
7	P57	0	R/W	汎用出力ポートの出力値を格納します。
6	P56	0	R/W	このレジスタをリードすると、
5	P55	0	R/W	PCR5 が 1 のビットはこのレジスタの値が読み出されます。
4	P54	0	R/W	PCR5 がの 0 ビットはこのレジスタの値にかかわらず
3	P53	0	R/W	端子の状態が読み出されます。
2	P52	0	R/W	
1	P51	0	R/W	
0	P50	0	R/W	

ポートプルアップコントロールレジスタ 5(PUCR5)

PUCR5 は入力ポートに設定された端子のプルアップ MOS をビットごとに制御します。

表 7.13 ポートプルアップコントロールレジスタ 5(PUCR5)

ビット	ビット名	初期値	R/W	説 明
7	-	0	-	リザーブビットです。読み出すと常に 0 が読み出されます。
6	-	0	-	
5	PUCR55	0	R/W	PCR5 が 0 に設定されているビットのみ有効。
4	PUCR54	0	R/W	1 をセットすると対応する端子の
3	PUCR53	0	R/W	プルアップ MOS がオン状態となり、
2	PUCR52	0	R/W	0 にクリアするとオフします。
1	PUCR51	0	R/W	
0	PUCR50	0	R/W	

7.2.4 レジスタ定義

I／Oポートのレジスタ定義は iodefine.h で行われていて、P.145 の「LED0,2 の点滅 (HEW の iodefine.h を利用)」で掲載した通りです。

ここでは、ポート 5 に関連する部分を抜き出しておきましょう。

ポート 15 のレジスタ定義 (iodefine.h の一部を抜粋)

```
struct st_io {                                          /* struct IO    */
                .
                .
                .
        union {                                         /* PUCR5        */
              unsigned char BYTE;                       /*  Byte Access */
              struct {                                  /*  Bit  Access */
                     unsigned char   :2;                /*    Bit 7,6   */
                     unsigned char B5:1;                /*    Bit 5     */
                     unsigned char B4:1;                /*    Bit 4     */
                     unsigned char B3:1;                /*    Bit 3     */
                     unsigned char B2:1;                /*    Bit 2     */
                     unsigned char B1:1;                /*    Bit 1     */
                     unsigned char B0:1;                /*    Bit 0     */
                     }      BIT;                       /*              */
              }            PUCR5;                       /*              */
                .
                .
                .
        union {                                         /* PDR5         */
              unsigned char BYTE;                       /*  Byte Access */
              struct {                                  /*  Bit  Access */
                     unsigned char B7:1;                /*    Bit 7     */
                     unsigned char B6:1;                /*    Bit 6     */
                     unsigned char B5:1;                /*    Bit 5     */
                     unsigned char B4:1;                /*    Bit 4     */
                     unsigned char B3:1;                /*    Bit 3     */
                     unsigned char B2:1;                /*    Bit 2     */
                     unsigned char B1:1;                /*    Bit 1     */
                     unsigned char B0:1;                /*    Bit 0     */
                     }      BIT;                       /*              */
              }            PDR5;                       /*              */
                .
                .
                .
        union {                                         /* PMR5         */
              unsigned char BYTE;                       /*  Byte Access */
              struct {                                  /*  Bit  Access */
                     unsigned char     :2;             /*              */
                     unsigned char WKP5:1;              /*    WKP5      */
                     unsigned char WKP4:1;              /*    WKP4      */
                     unsigned char WKP3:1;              /*    WKP3      */
                     unsigned char WKP2:1;              /*    WKP2      */
                     unsigned char WKP1:1;              /*    WKP1      */
                     unsigned char WKP0:1;              /*    WKP0      */
                     }      BIT;                       /*              */
              }            PMR5;                       /*              */
                .
                .
                .
        unsigned char      PCR5;                       /* PCR5         */
                .
                .
                .
};                                                      /*              */

#define IO      (*(volatile struct st_io    *)0xFFD0)   /* IO    Address*/
```

7.2.5 基本的なプログラム

すでに説明したように、ロータリスイッチではチャタリングが問題になるような使い方はあまりしませんので、チャタリングに関しては対処をしていません。

以下に、ロータリスイッチの値を LED に反映するプログラムを紹介しますが、このプログラムには LED の関数を使っています。led.h、led.c を利用できるように設定してください (P.156 の図 6.6 以降参照)。

03_ROTSW01.c

```
 1  /************************************************************************/
 2  /*                                                                      */
 3  /*  FILE        :03_ROTSW01.c                                           */
 4  /*  DESCRIPTION :ロータリスイッチの値 3 ビット分に対応して LED 点灯     */
 5  /*  CPU TYPE    :H8/3694F                                               */
 6  /*                                                                      */
 7  /*  ロータリスイッチ                                                    */
 8  /*  ROTSW1 P50                                                          */
 9  /*  ROTSW2 P51                                                          */
10  /*  ROTSW3 P52                                                          */
11  /*  ROTSW4 P53                                                          */
12  /*                                                                      */
13  /************************************************************************/
14
15  #include "led.h"
16  #include "iodefine.h"
17
18  #define ROTSW1 0x01
19  #define ROTSW2 0x02
20  #define ROTSW3 0x04
21  #define ROTSW4 0x08
22
23  #define SW_ROT_MASK ( ROTSW1 | ROTSW2 | ROTSW3| ROTSW4 )
24
25  void main(void)
26  {
27    LedInit();      /* LED 接続ポートの初期化 */
28    IO.PCR5 &= (~SW_ROT_MASK);   /* ロータリスイッチ接続ポートの初期化 */
29    IO.PUCR5.BYTE |= SW_ROT_MASK;/* プルアップの設定 */
30
31    while(1){
32      LedSet((~IO.PDR5.BYTE) & SW_ROT_MASK); /* ロータリスイッチの値 (0-7) を LED に反映 */
33    }
34  }
```

End Of List

実行結果

ロータリスイッチの値 (下位 3 ビット分) を LED に反映する (表 7.14 参照)。

表 7.14　つまみの位置と LED 点灯消灯の関係 (- は消灯の意味)

つまみの位置／端子	LED0	LED1	LED2
0	-	-	-
1	点灯	-	-
2	-	点灯	-
3	点灯	点灯	-
4	-	-	点灯
5	点灯	-	点灯
6	-	点灯	点灯
7	点灯	点灯	点灯
8	-	-	-
9	点灯	-	-
A	-	点灯	-
B	点灯	点灯	-
C	-	-	点灯
D	点灯	-	点灯
E	-	点灯	点灯
F	点灯	点灯	点灯

プログラム解説 (03_ROTSW01.c)

```
15    #include "led.h"
```

今回も LED 関連の関数を利用しています。P.156 の図 6.6 以降と同様の操作をしてください。

```
18    #define ROTSW1 0x01
19    #define ROTSW2 0x02
20    #define ROTSW3 0x04
21    #define ROTSW4 0x08
22
23    #define SW_ROT_MASK ( ROTSW1 | ROTSW2 | ROTSW3| ROTSW4 )
```

ロータリスイッチのプログラムに必要な定数の定義をしています。

SW_ROT_MASK の値は 0x0F になることを確認してください。

```
28    IO.PCR5 &= (~SW_ROT_MASK);   /* ロータリスイッチ接続ポートの初期化 */
```

ロータリスイッチのポートを入力に設定しています。

```
29    IO.PUCR5.BYTE |= SW_ROT_MASK;/* プルアップの設定 */
```

ロータリスイッチのポートにプルアップを設定しています。

```
32    LedSet((~IO.PDR5.BYTE) & SW_ROT_MASK); /* ロータリスイッチの値 (0-7) を LED に反映 */
```

(~IO.PDR5.BYTE) & SW_ROT_MASK でロータリスイッチの値を取り出しています。その値を関数 LedSet に渡しています。この値は 0 から 15 を取りますが、関数 LedSet は下位 3 ビットのみを反映させます。詳細は P.151 の 6.1 を確認してください。

7.2.6 関数化

では、ロータリスイッチ関連の関数も作っておきましょう。関数の仕様は以下の通りです。

ロータリスイッチ関連関数 (H8/3694F ボード)

ヘッダファイル：sw.h

```
ROTSW1 P50
ROTSW2 P51
ROTSW3 P52
ROTSW4 P53

*****************************************************************
void RotSwInit(void);

    形式  : #include"sw.h"
            RotSwInit(void);
    引数：  なし
    戻り値：なし
    解説：  ロータリスイッチ読み取りで使用する I/O ポートを初期化する関数。
            各端子をプルアップする。
            ロータリスイッチ読み取り関数を使用する前に実行すること。

*****************************************************************
unsigned char RotSwGet(void);

    形式  : #include"sw.h"
            RotSwGet(void);
    引数：  なし
    戻り値：unsigned char
            ロータリスイッチの状態 (下位 4 ビットのみ：0-15)。
            ロータリスイッチのつまみの数値と同じ値が返る。
    解説：  ロータリスイッチの状態を読み取る関数。
            チャタリング対策は行っていない。
```

タクトスイッチの関数とロータリスイッチの関数はともに、sw.h、sw.c に定義しました。

以下では、タクトスイッチの関数も掲載していますので、ロータリスイッチ関連の部分のみ追加しておいてください。

今回は、ロータリスイッチの値によって、LED の点滅の速度を変更してみました。

今回も LED の関数を利用できるように設定してください。

sw.h

```
 1  #ifndef _SW_H_
 2  #define _SW_H_
 3
 4  #include "iodefine.h"
 5
 6  #define ROTSWIO  IO.PCR5
 7  #define ROTSWDAT IO.PDR5.BYTE
 8  #define ROTSWPU  IO.PUCR5.BYTE
 9
10  #define SWIO  IO.PCR1
11  #define SWDAT IO.PDR1.BYTE
12  #define SWPU  IO.PUCR1.BYTE
13
14  #define ROTSW1 0x01
15  #define ROTSW2 0x02
16  #define ROTSW3 0x04
17  #define ROTSW4 0x08
18
19  #define SW0 0x01
20  #define SW_SHIFT 4
21
22  #define SW_ROT_MASK ( ROTSW1 | ROTSW2 | ROTSW3 | ROTSW4 )
23  #define SW_MASK ( SW0 << SW_SHIFT )
24
25  #define SW_LOOP 1000 /* タクトスイッチ確認最大回数、安定しなかったら最後の値 */
26  #define SW_COUNT 100 /* 値がこの回数同じだったら、スイッチの値とする */
27
28  void RotSwInit(void);
29  unsigned char RotSwGet(void);
30
31  void SwInit(void);
32  unsigned char SwGet(void);
33  void SwWaitPush(void);
34  void SwWaitDetach(void);
```

```
35
36  #endif
```

End Of List

sw.c

```
 1  /*********************************************************************/
 2  /*                                                                   */
 3  /*  FILE        :sw.c                                                */
 4  /*  DESCRIPTION :スイッチ用関数                                      */
 5  /*  CPU TYPE    :H8/3694F                                            */
 6  /*                                                                   */
 7  /*  ロータリスイッチ                                                 */
 8  /*  ROTSW1 P50                                                       */
 9  /*  ROTSW2 P51                                                       */
10  /*  ROTSW3 P52                                                       */
11  /*  ROTSW4 P53                                                       */
12  /*                                                                   */
13  /*  タクトスイッチ                                                   */
14  /*  SW0    P14                                                       */
15  /*                                                                   */
16  /*********************************************************************/
17
18  #include "sw.h"
19
20  /*******************************************
21    ロータリスイッチのつながった端子を初期化
22  *******************************************/
23  void RotSwInit(void)
24  {
25    ROTSWIO &= (~SW_ROT_MASK);
26    ROTSWPU |= SW_ROT_MASK;/* プルアップの設定 */
27  }
28
29  /*******************************************
30    ロータリスイッチの状態を取得
31  *******************************************/
32  unsigned char RotSwGet(void)
33  {
34    return ((~ROTSWDAT) & SW_ROT_MASK);
35  }
36
37  /*******************************************
38    タクトスイッチのつながった端子を初期化
39  *******************************************/
40  void SwInit(void)
41  {
42    SWIO &= (~SW_MASK);
43    SWPU |= SW_MASK; /* プルアップ */
44  }
45
46  /*******************************************
47    タクトスイッチの状態を取得
48  *******************************************/
49  unsigned char SwGet(void)
50  {
51    /* i:スイッチの値を確認した回数 */
52    /* count:スイッチの値が連続して同じだった回数 */
53    int i=0, count=0;
54
55    /* old_state:現在と異なる最近のスイッチの値 */
56    /* new_state:現在のスイッチの値 */
57    unsigned char new_state, old_state=255;
58
59    while((i<SW_LOOP) && (count<SW_COUNT)){
60      new_state = (SWDAT & SW_MASK);
61      if(old_state == new_state){
62        count++;
63      }else{
64        count = 0;
65        old_state = new_state;
66      }
67      i++;
68    }
69    return (((~new_state) & SW_MASK) >> SW_SHIFT);
70  }
71
72  /*******************************************
73    タクトスイッチが押されるまで待つ
74  *******************************************/
75  void SwWaitPush(void)
76  {
77    while(!SwGet()){
78      ;
79    }
80  }
81
82  /*******************************************
```

```
83    タクトスイッチが離されるまで待つ
84  ****************************************/
85  void SwWaitDetach(void)
86  {
87    while(SwGet()){
88      ;
89    }
90  }
91
```

End Of List

03_ROTSW02.c

```
 1  /*********************************************************************/
 2  /*                                                                   */
 3  /*  FILE        :03_ROTSW02.c                                        */
 4  /*  DESCRIPTION :ロータリスイッチの値に対応して LED 点滅速度変更     */
 5  /*  CPU TYPE    :H8/3694F                                            */
 6  /*                                                                   */
 7  /*  ロータリスイッチ                                                 */
 8  /*  ROTSW1 P50                                                       */
 9  /*  ROTSW2 P51                                                       */
10  /*  ROTSW3 P52                                                       */
11  /*  ROTSW4 P53                                                       */
12  /*                                                                   */
13  /*********************************************************************/
14
15  #include "led.h"
16  #include "sw.h"
17
18  #define LED_MAX (LED0 | LED1 | LED2)
19  #define ROT_SW_WAIT_LOOP 0x7FFFFL
20
21  void RotSwWaitLoop(unsigned char m)
22  {
23    unsigned long count;
24
25    for(count=0; count<ROT_SW_WAIT_LOOP*(m+1); count++){
26      ;
27    }
28  }
29
30  void main(void)
31  {
32    unsigned char i, led_data;
33
34    LedInit();      /* LED 接続ポートの初期化 */
35    RotSwInit();   /* ロータリスイッチ接続ポートの初期化 */
36
37    while(1){
38      for(led_data=0; led_data<=LED_MAX; led_data++){
39        LedSet(led_data);
40        i=RotSwGet(); /* ロータリスイッチの値を読む */
41        RotSwWaitLoop(i);
42      }
43    }
44  }
```

End Of List

実行結果

ロータリスイッチの値によって LED の点滅速度が変化する。
値が大きいほど点滅速度は遅くなる。

プログラム解説 (sw.h)

ロータリスイッチに関係する部分は、6 行目から 8 行目まで、14 行目から 17 行目まで、22 行目、28 行目から 29 行目までです。

ほとんどが、サンプルプログラム 03_ROTSW01.c で定義したものと同じです。

```
6    #define ROTSWIO  IO.PCR5
7    #define ROTSWDAT IO.PDR5.BYTE
8    #define ROTSWPU  IO.PUCR5.BYTE
```

ロータリスイッチに関係するポートの名前を定義しなおしました。すべてバイト単位で扱っています。すでに説明したように、レジスタはバイト単位で利用したほうが移植性のよいプログラムが書きやすいからです。

プログラム解説 (sw.c)

23 行目から 27 行目までの初期化関数と、32 行目から 35 行目までのロータリスイッチの状態を取得する関数が追加分です。

関数の中身は、03_ROTSW01.c で説明したプログラムを流用しています。そちらの説明を参照してください。

プログラム解説 (03_ROTSW02.c)

```
16   #include "sw.h"
```

ロータリスイッチの関数を利用する場合には、sw.h を include します。

```
19   #define ROT_SW_WAIT_LOOP 0x7FFFFL
20
21   void RotSwWaitLoop(unsigned char m)
22   {
23     unsigned long count;
24
25     for(count=0; count<ROT_SW_WAIT_LOOP*(m+1); count++){
26       ;
27     }
28   }
```

led 関連の関数として WaitLoop という待ち関数を定義しました。

これは、チャタリングとは関係なく、LED の点滅のスピードを変えるために用いています。

0x7FFFF と関数 RotSwWaitLoop に渡された引数 (0-255)+1 をかけた回数だけ for ループをまわします。1 足しているのは、0 が渡されたときに待ちループをすぐに抜けてしまわないようにです。

WaitLoop の仕様を変更して、このように書き換えてもよいかもしれません。

```
35   RotSwInit();   /* ロータリスイッチ接続ポートの初期化 */
```

ロータリスイッチを初期化する関数 RotSwInit を実行しています。ロータリスイッチを使用する際には必ず実行します。

```
40    i=RotSwGet(); /* ロータリスイッチの値を読む */
41    RotSwWaitLoop(i);
```

40 行目でロータリスイッチの値を読みます (0-15 が返ってきます)。

41 行目では、40 行目で取得した値を関数 RotSwWaitLoop に渡しています。ロータリスイッチの値が大きいほど LED の点滅が遅くなります。

なお、40 行目と 41 行目はまとめて、

RotSwWaitLoop(RotSwGet());

と 1 行で書くこともできます。その際には変数 i は不要になります。

✍ 課題 7.2.1 (提出) ロータリスイッチの値によって LED の動作を変更するプログラム

プログラム実行時にロータリスイッチの値を読んでください。

ロータリスイッチの値によって以下の動作を行うようにプログラムしてください。

```
    0       : 右シフト
    1       : 左シフト
その他の値  : カウントアップ
```

上記の sw.h、sw.c を利用してください。

プロジェクト名：e03_ROTSW02

8 IRQの利用

8.1 IRQ(Interrupt ReQuest)

これまで LED の点滅などをプログラムする際に、WaitLoop のような何もしないループを使って時間稼ぎをしてきました。

WaitLoop のような for ループを用いた時間待ちでは、正確な時間が計測できないという点をすでに指摘しておきましたが、それ以外にも問題があります。

このような待ち方をしていると、その間 CPU は無駄な演算を延々としていることになります。LED の点滅だけを行っている場合にはこれでも問題はないかもしれませんが、他の周辺機器も制御するとなると、このような待ち方は効率的ではありません。時間待ちをしている間に他の周辺機器を制御したいところです。

このような目的のためにマイコンには割り込みという機能が備わっています。割り込みの概念は少々分かりにくいところもあるかも知れませんが、マイコンのプログラムでは必要不可欠ですので、きちんと理解しておいてください。

8.1.1 割込みとは

これまで LED の点滅などに利用してきた WaitLoop 関数のことを考えてみましょう。この関数は何もしない for ループを、一定回数延々と実行しているだけです。時間待ちをしているという以外に何もしていません。CPU はこの無駄とも思える演算にかかりっきりになっているわけです。

われわれの日常生活でこれと似た例を探してみましょう。たとえば風呂に水をためてこれを沸かすとします。WaitLoop 関数がやっていることは、水を出して湯船にたまるまでその場で監視していることと同じです。水がたまるまで他の事もせずに、待っているのです。

少なくとも最近の家庭ではこのようなことはしないと思います。たいていが風呂の栓をして湯船にふたをし、スイッチを押すとあとは自動で水をため、湯を沸かし、知らせてくれます。その間にわれわれはテレビを見るなり本を読むなりと他の事をすることができるわけです。

割り込みというのは、このようにある条件を設定することによって、その条件が満たされたときに教えてくれる機能なのです。

たとえば 10 ミリ秒経過したら知らせてくれるように設定しておくことができるわけです。さらに、条件が満たされたときに実行するべき関数を登録しておくことで、自動的にその関数を実行してくれます。

マイコンではどのような割り込みが利用できるかはあらかじめ決められています。また、その際に実行する関数の登録場所も決められています。割込みは複数用意されているので、それらの間の優先順位も決められています。

H8/3694F のマニュアルでは、リセットとトラップ命令による例外処理と割込み例外処理をまとめて例外処理と呼んでいるようです。トラップ命令はデバッグの際などに利用される命令です。

表 8.1 に H8/3694F の例外処理の一覧を示します。

表 8.1 例外処理要因とベクタアドレス

発生元	例外処理要因	ベクタ番号	ベクタアドレス	優先度
$\overline{RES}$ 端子 ウォッチドッグタイマ	リセット	0	H'0000~H'0001	高
-	システム予約	1~6	H'0002~H'000D	
外部割り込み端子	NMI	7	H'000E~H'000F	
CPU	トラップ命令 #0	8	H'0010~H'0011	
	トラップ命令 #1	9	H'0012~H'0013	
	トラップ命令 #2	10	H'0014~H'0015	
	トラップ命令 #3	11	H'0016~H'0017	
アドレスブレーク	ブレーク条件成立	12	H'0018~H'0019	
外部割り込み端子	IRQ0	14	H'001C~H'001D	
	IRQ1	15	H'001E~H'001F	
	IRQ2	16	H'0020~H'0021	
	IRQ3	17	H'0022~H'0023	
	WKP	18	H'0024~H'0025	
タイマA	オーバフロー	19	H'0026~H'0027	
-	システム予約	20	H'0028~H'0029	
タイマ W	インプットキャプチャA／コンペアマッチA インプットキャプチャB ／コンペアマッチ B インプットキャプチャC ／コンペアマッチ C インプットキャプチャD ／コンペアマッチ D オーバフロー	21	H'002A~H'002B	
タイマV	コンペアマッチA コンペアマッチB オーバフロー	22	H'002C~H'002D	
SCI3	受信データフル 送信データエンプティ 送信終了 受信エラー	23	H'002E~H'002F	
IIC2	送信データエンプティ、送信終了 受信データフル、 アービトレーションロスト／オーバランエラー NACK 検出、停止条件検出	24	H'0030~H'0031	低
A/D 変換器	A/D 変換終了	25	H'0032~H'0033	

この表で、リセットとトラップ命令以外が割込み例外処理にあたります。割込みは外部割り込みと内部割込みに分類することができます。

外部割り込みとは、周辺回路など外部からの信号による割込みです。表 8.1 では、NMI と IRQ0~IRQ3 がそれにあたります。IRQ0~IRQ3 は割込みをマスクする (受け付けない設定にする) ことができますが、NMI はマスクすることはできません。NMI は緊急停止ボタンなどに用いることができます。

内部割込みとは、CPU 内部の演算結果によって割込みをかけるものを言います。タイマの設定によって、一定時間がきたら割込みをかけるというようなものがその一例です。表 8.1 の内部割込みは、アドレスブレーク以外は、マスクをかけることが出来るようです。

表 8.1 における「ベクタアドレス」の欄が割込みの際に呼び出す関数を登録しておくメモリのアドレスです。P.58 の図 3.2 で「H8/3694 グループのメモリマップ」を紹介しました。そこでは割込みベクトルが H'0000~H'0033 であることにふれましたが、その詳細が表 8.1 になります。

割込みベクタに割込み関数を登録する方法は開発環境によって異なります。また同じ HEW を利用していても、マイコンによって (デフォルトの環境における) 登録の仕方が異なるようです。

ここでは、外部割り込みの例として IRQ0 を例に、割込みの機能とその設定方法を確認していきましょう。

8.1.2 回路図

今回の回路は P.167 の図 7.2 と全く同じものです。P14 は IRQ0 と共用となっていて、レジスタの設定で機能を切り替えることが可能なのです。

今回もプルアップはマイコン内部の機能を利用することにしましょう。

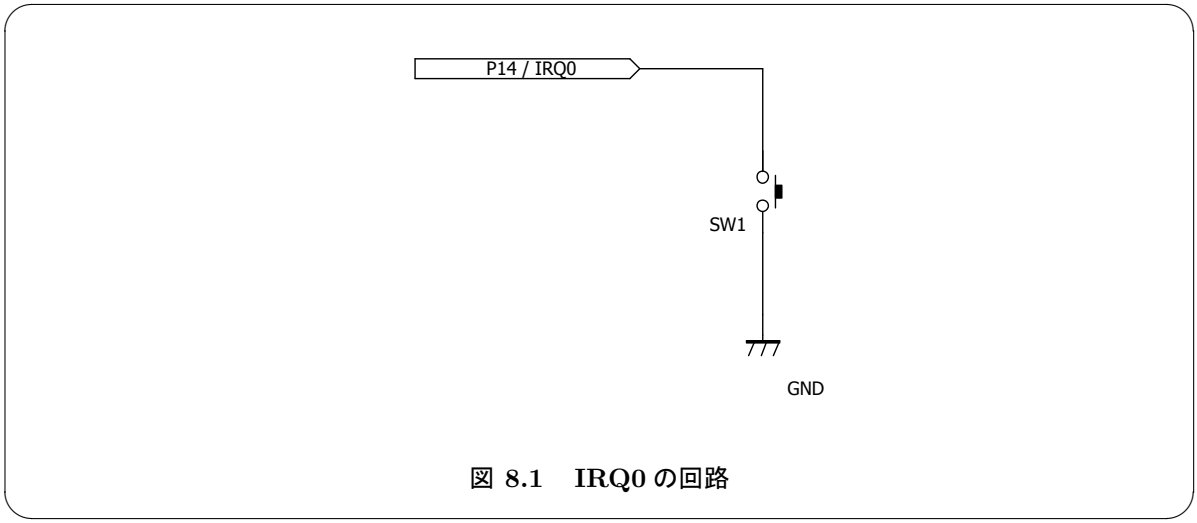


図 8.1 IRQ0 の回路

この回路においては、タクトスイッチの ON，OFF と端子 IRQ0(P14) の HIGH，LOW の関係は、以下のようになります (表 8.2)。これも P.167 の表 7.1 と同じです。

表 8.2 スイッチの ON，OFF と端子 IRQ0 の HIGH，LOW の関係

タクトスイッチ	P14/IRQ0
ON	LOW(0)
OFF	HIGH(1)

8.1.3 関連レジスタ

レジスタもポート 1 に関しては、P.169 の「関連レジスタ」と全く同じです。そちらを参考にしてください。

今回はポートモードレジスタ 1(PMR1) を用いて P14 端子を IRQ0 に設定します。

割込みを利用しますので、例外処理関連のレジスタも利用することになります。

例外処理関係で今回利用するレジスタは以下のものです。

- 割り込みエッジセレクトレジスタ 1(IEGR1)
- 割り込みイネーブルレジスタ 1(IENR1)
- 割り込みフラグレジスタ 1(IRR1)

上記のレジスタは次のアドレスに割り振られています。

表 8.3 例外処理関連レジスタのアドレス

レジスタ名	アドレス
割り込みエッジセレクトレジスタ 1(IEGR1)	H'FFF2
割り込みイネーブルレジスタ 1(IENR1)	H'FFF4
割り込みフラグレジスタ 1(IRR1)	H'FFF6

割り込みエッジセレクトレジスタ 1(IEGR1)

IEGR1 は NMI、IRQ3~IRQ0 端子の割り込み要求を発生させるエッジの方向を選択します。立ち上がりエッジとは、端子の状態が LOW(0) から HIGH(1) に変化をするところを言い、立ち下がりエッジとは、端子の状態が HIGH(1) から LOW(0) に変化をするところを言います (図 8.2)。

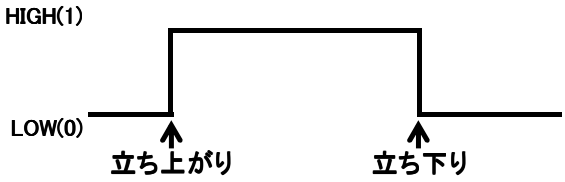


図 8.2 立ち上がりエッジと立ち下がりエッジ

表 8.4 割り込みエッジセレクトレジスタ 1(IEGR1)

ビット	ビット名	初期値	R/W	説 明
7	NMIEG	0	R/W	NMI エッジセレクト 0 : $\overline{NMI}$ 端子入力の立ち下がりエッジを検出 1 : $\overline{NMI}$ 端子入力の立ち上がりエッジを検出
6 5 4	- - -	1 1 1	- - -	リザーブビットです。読み出すと常に 1 が読み出されます。
3	IEG3	0	R/W	IRQ3 エッジセレクト 0 : $\overline{IRQ3}$ 端子入力の立ち下がりエッジを検出 1 : $\overline{IRQ3}$ 端子入力の立ち上がりエッジを検出
2	IEG2	0	R/W	IRQ2 エッジセレクト 0 : $\overline{IRQ2}$ 端子入力の立ち下がりエッジを検出 1 : $\overline{IRQ2}$ 端子入力の立ち上がりエッジを検出
1	IEG1	0	R/W	IRQ1 エッジセレクト 0 : $\overline{IRQ1}$ 端子入力の立ち下がりエッジを検出 1 : $\overline{IRQ1}$ 端子入力の立ち上がりエッジを検出
0	IEG0	0	R/W	IRQ0 エッジセレクト 0 : $\overline{IRQ0}$ 端子入力の立ち下がりエッジを検出 1 : $\overline{IRQ0}$ 端子入力の立ち上がりエッジを検出

割り込みイネーブルレジスタ 1(IENR1)

IENR1 は直接遷移割り込み、タイマ A オーバフロー割り込みおよび外部端子割り込みをイネーブルにします。「イネーブルにする」とは「有効にする」という意味です。

表 8.5 割り込みイネーブルレジスタ 1(IENR1)

ビット	ビット名	初期値	R/W	説 明
7	IENDT	0	R/W	直接遷移割り込み要求イネーブル このビットを 1 にセットすると 直接遷移割り込み要求がイネーブルになります。
6	IENTA	0	R/W	タイマ A 割り込み要求イネーブル このビットを 1 にセットすると タイマ A のオーバフロー割り込み要求がイネーブルになります。
5	IENWP	0	R/W	ウェイクアップ割り込み要求イネーブル このビットは $\overline{WKP5}$ ~ $\overline{WKP0}$ 端子共通のイネーブルビットで 1 にセットすると割り込み要求がイネーブルになります。
4	-	1	-	リザーブビットです。読み出すと常に 1 が読み出されます。
3	IEN3	0	R/W	IRQ3 割り込み要求イネーブル このビットを 1 にセットすると $\overline{IRQ3}$ 端子の 割り込み要求がイネーブルになります。
2	IEN2	0	R/W	IRQ2 割り込み要求イネーブル このビットを 1 にセットすると $\overline{IRQ2}$ 端子の 割り込み要求がイネーブルになります。
1	IEN1	0	R/W	IRQ1 割り込み要求イネーブル このビットを 1 にセットすると $\overline{IRQ1}$ 端子の 割り込み要求がイネーブルになります。
0	IEN0	0	R/W	IRQ0 割り込み要求イネーブル このビットを 1 にセットすると $\overline{IRQ0}$ 端子の 割り込み要求がイネーブルになります。

割り込みイネーブルレジスタをクリアすることにより割り込み要求をディスエーブル (無効) にする場

合、または割り込みフラグレジスタをクリアする場合は、割り込み要求をマスクした状態（I = 1）で行ってください。

割り込みフラグレジスタ 1(IRR1)

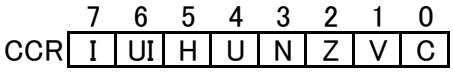
IRR1 は直接遷移割り込み、タイマ A オーバフロー割り込み、IRQ3~IRQ0 割り込み要求ステータスフラグレジスタです。

表 8.6 割り込みフラグレジスタ 1(IRR1)

ビット	ビット名	初期値	R/W	説 明
7	IRRDT	0	R/W	直接遷移割り込み要求フラグ [セット条件] SYSCR2 の DTON に 1 をセットした状態でスリープ命令を実行し直接遷移したとき [クリア条件] 0 をライトしたとき
6	IRRTA	0	R/W	タイマ A 割り込み要求フラグ [セット条件] タイマ A がオーバフローしたとき [クリア条件] 0 をライトしたとき
5 4	- -	1 1	- -	リザーブビットです。読み出すと常に 1 が読み出されます。
3	IRRI3	0	R/W	IRQ3 割り込み要求フラグ [セット条件] $\overline{IRQ3}$ 端子が割り込み入力に設定され、指定されたエッジを検出したとき [クリア条件] 0 をライトしたとき
2	IRRI2	0	R/W	IRQ2 割り込み要求フラグ [セット条件] $\overline{IRQ2}$ 端子が割り込み入力に設定され、指定されたエッジを検出したとき [クリア条件] 0 をライトしたとき
1	IRRI1	0	R/W	IRQ1 割り込み要求フラグ [セット条件] $\overline{IRQ1}$ 端子が割り込み入力に設定され、指定されたエッジを検出したとき [クリア条件] 0 をライトしたとき
3	IRRI0	0	R/W	IRQ0 割り込み要求フラグ [セット条件] $\overline{IRQ0}$ 端子が割り込み入力に設定され、指定されたエッジを検出したとき [クリア条件] 0 をライトしたとき

さらに、割込みを実行するためにはコンディションコードレジスタ (CCR) の割込みマスクビットをクリアする必要があります。

コンディションコードレジスタ (CCR) は CPU の内部レジスタです (図 8.3)。番地はなく、特別な名前が付いています。また、専用の機能を持っていて使い方が決められています。



I : 割り込みマスクビット
UI : ユーザビット／割り込みマスクビット
H : ハーフキャリフラグ
U : ユーザビット
N : ネガティブフラグ
Z : ゼロフラグ
V : オーバーフローフラグ
C : キャリフラグ

図 8.3 コンディションコードレジスタ (CCR)

割込みマスクビット (I ビット) が 1 にセットされると、割り込み要求がマスクされます。ただし、NMI は I ビットに関係なく受け付けられます。I ビットは例外処理の実行が開始されたときに 1 にセットされます。

CPU の内部レジスタは C 言語のプログラムで直接操作することができません。HEW ではこのために関数が用意されています。ヘッダファイル machine.h を読み込むことで、set_imask_ccr が使えるようになります。コンディションコードレジスタ (CCR) の割込みマスクビットはこれを用いて設定することにします。

関数の仕様は以下の通りです (参考文献 [2]P283 からの引用)。

```
void set_imask_ccr (unsigned char mask)

    説明：コンディションコードレジスタ (CCR) の割り込みマスクビット (I) に mask 値 (0 または 1) を設定します。
  ヘッダ：<machine.h>
    引数：mask 値 (0 または 1)

      例：#include <machine.h> /* 必ず<machine.h>をインクルードします。 */
          void main(void)
          {
            set_imask_ccr(0);  /* 割り込みマスクビットをクリアします。*/
          }
```

8.1.4 レジスタ定義

ポート 1 のレジスタ定義は、P.172 に掲載したものと同じです。

IRQ0 の割込みレジスタ定義 (iodefine.h の一部を抜粋) を掲載しておきます。

IRQ0 の割込みレジスタ定義 (iodefine.h の一部を抜粋)

```
union un_iegr1 {                                    /* union IEGR1  */
               unsigned char BYTE;                  /*  Byte Access */
               struct {                             /*  Bit  Access */
                      unsigned char NMIEG:1;        /*    NMIEG     */
                      unsigned char      :3;        /*              */
                      unsigned char IEG3 :1;        /*    IEG3      */
                      unsigned char IEG2 :1;        /*    IEG2      */
                      unsigned char IEG1 :1;        /*    IEG1      */
                      unsigned char IEG0 :1;        /*    IEG0      */
                      }      BIT;                   /*              */
};                                                  /*              */
union un_ienr1 {                                    /* union IENR1  */
               unsigned char BYTE;                  /*  Byte Access */
               struct {                             /*  Bit  Access */
                      unsigned char IENDT:1;        /*    IENDT     */
                      unsigned char IENTA:1;        /*    IENTA     */
                      unsigned char IENWP:1;        /*    IENWP     */
                      unsigned char      :1;        /*              */
                      unsigned char IEN3 :1;        /*    IEN3      */
                      unsigned char IEN2 :1;        /*    IEN2      */
                      unsigned char IEN1 :1;        /*    IEN1      */
                      unsigned char IEN0 :1;        /*    IEN0      */
                      }      BIT;                   /*              */
};                                                  /*              */
union un_irr1 {                                     /* union IRR1   */
              unsigned char BYTE;                   /*  Byte Access */
              struct {                              /*  Bit  Access */
                     unsigned char IRRDT:1;         /*    IRRDT     */
                     unsigned char IRRTA:1;         /*    IRRTA     */
                     unsigned char      :2;         /*              */
                     unsigned char IRRI3:1;         /*    IRRI3     */
                     unsigned char IRRI2:1;         /*    IRRI2     */
                     unsigned char IRRI1:1;         /*    IRRI1     */
                     unsigned char IRRI0:1;         /*    IRRI0     */
                     }      BIT;                   /*              */
};                                                  /*              */

......

#define IEGR1   (*(volatile union  un_iegr1 *)0xFFF2)   /* IEGR1 Address*/
#define IENR1   (*(volatile union  un_ienr1 *)0xFFF4)   /* IENR1 Address*/
#define IRR1    (*(volatile union  un_irr1  *)0xFFF6)   /* IRR1  Address*/
```

8.1.5 基本的なプログラム (割込み関数を利用しない)

まずは割込みなしで IRQ0 のプログラムをしてみましょう。タクトスイッチは OFF で 1 になるように接続されているので (P.167 図 7.2)、立ち下がりエッジを検出することにします。

割込みで関数を実行するのではなく、割り込みフラグレジスタ 1(IRR1) の IRQ0 割り込み要求フラグ (P.222 表 9.5) がセットされるのを監視してみましょう。

lib に格納されている LED の関数を利用する方法をもう一度まとめておきます。詳細は P.156 の図 6.6 以降を確認してください。

1. プロジェクトフォルダ内にある iodefine.h は削る。
2. プロジェクトに led.c を追加。
3. インクルードファイルディレクトリに lib を追加。

04_IRQ01.c

```
 1  /************************************************************************/
 2  /*                                                                      */
 3  /*  FILE        :04_IRQ01.c                                             */
 4  /*  DESCRIPTION :タクトスイッチを押したら LED1 が点灯                   */
 5  /*  CPU TYPE    :H8/3694F                                               */
 6  /*                                                                      */
 7  /*  タクトスイッチ                                                      */
 8  /*  SW0 P14/IRQ0                                                        */
 9  /*                                                                      */
10  /************************************************************************/
11
12  #include "led.h"
13  #include "iodefine.h"
14
15  void IrqInit(void)
16  {
17    IO.PMR1.BIT.IRQ0=1; /* IRQ0 入力端子 */
18    IO.PUCR1.BIT.B4=1;  /* プルアップを設定 */
19    IEGR1.BIT.IEG0=0;   /* IRQ0 端子入力の立ち下がりエッジを検出 */
20    IENR1.BIT.IEN0=0;   /* IRQ0 端子の割り込み要求なし */
21    IRR1.BIT.IRRI0=0;   /* IRQ0 割り込み要求フラグクリア */
22  }
23
24  void main(void)
25  {
26    LedInit();  /* LED 接続ポートの初期化 */
27    IrqInit();  /* IRQ 初期化 */
28    while(IRR1.BIT.IRRI0==0){ /* IRQ0 割り込み要求フラグセットを待つ */
29      ;
30    }
31    LedSet( LED1 );  /* LED1 点灯 */
32    while(1){
33      ;
34    }
35  }
```

End Of List

実行結果

最初は LED はすべて消灯。
タクトスイッチを押したら LED1 が点灯する (その後はタクトスイッチを押しても何も起こらない)。

プログラム解説 (04_IRQ01.c)

```
17    IO.PMR1.BIT.IRQ0=1; /* IRQ0 入力端子 */
18    IO.PUCR1.BIT.B4=1;  /* プルアップを設定 */
```

15 行目から 22 行目までが IRQ0 端子の初期化関数です。

17 行目と 18 行目はポート 1 関連のレジスタです。17 行目で P14 を IRQ0 入力端子に設定しています。18 行目はプルアップ設定です。

```
19    IEGR1.BIT.IEG0=0;   /* IRQ0 端子入力の立ち下がりエッジを検出 */
20    IENR1.BIT.IEN0=0;   /* IRQ0 端子の割り込み要求なし */
21    IRR1.BIT.IRRI0=0;   /* IRQ0 割り込み要求フラグクリア */
```

割込み関係のレジスタを定義しています。

19 行目では、IRQ0 の立ち下がりエッジを検出するように設定しています。20 行目では割り込みの設定をしていません。ここのプログラムでは、割り込みフラグレジスタ 1(IRR1) の IRQ0 割り込み要求フラグがセットされるのを監視することにします。21 行目は割り込み要求フラグのクリアです。このフラグが 1 にセットされるのを監視します。

```
27    IrqInit();  /* IRQ 初期化 */
```

IRQ0 の初期化関数を呼び出しています。

```
28    while(IRR1.BIT.IRRI0==0){ /* IRQ0 割り込み要求フラグセットを待つ */
29      ;
30    }
```

割り込みフラグレジスタ 1(IRR1) の IRQ0 割り込み要求フラグが 1 にセットされるまでループしています。

割込みは、この割り込み要求フラグが設定されたときに、登録されている関数を実行する機能です。

次にこの機能を利用してみましょう。

8.1.6 基本的なプログラム (割込み関数を利用する)

例外処理要因によって関数を実行するには、P.201 の表 8.1 の該当するベクタアドレスにその関数を登録する必要があります。HEW の現在のプロジェクトでは、この登録は intprg.c で行っています。

intprg.c(修正前)

```
// vector 14 IRQ0
__interrupt(vect=14) void INT_IRQ0(void) {/* sleep(); */}
```

この

```
/* sleep(); */
```

のところに実行したいプログラムを書くと、IRQ0 外部割込みが発生した際に実行されます。

ここでは intprg.c の中ではなく、main 関数と同じファイルに割込み関数を記述することにします。そのために、この行をコメントアウトしましょう。

intprg.c(修正後)

```
// vector 14 IRQ0
//__interrupt(vect=14) void INT_IRQ0(void) {/* sleep(); */}
```

04_IRQ02.c

```
 1  /************************************************************************/
 2  /*                                                                      */
 3  /*  FILE        :04_IRQ02.c                                             */
 4  /*  DESCRIPTION :タクトスイッチを押したら LED が全て点灯                */
 5  /*  CPU TYPE    :H8/3694F                                               */
 6  /*                                                                      */
 7  /*  タクトスイッチ                                                      */
 8  /*  SW0 P14/IRQ0                                                        */
 9  /*                                                                      */
10  /************************************************************************/
11
12  #include <machine.h>
13  #include "led.h"
14  #include "iodefine.h"
15
16  void IrqInit(void)
17  {
18    set_imask_ccr(1);   /* 割込み不可 */
19    IO.PMR1.BIT.IRQ0=1; /* IRQ0 入力端子 */
20    IO.PUCR1.BIT.B4=1;  /* プルアップを設定 */
21    IEGR1.BIT.IEG0=0;   /* IRQ0 端子入力の立ち下がりエッジを検出 */
22    IENR1.BIT.IEN0=1;   /* IRQ0 端子の割り込み要求 */
23    IRR1.BIT.IRRI0=0;   /* IRQ0 割り込み要求フラグクリア */
24    set_imask_ccr(0);   /* 割込み許可 */
25  }
26
27  void main(void)
28  {
29    LedInit();  /* LED 接続ポートの初期化 */
30    IrqInit();  /* IRQ 初期化 */
31    while(1){
32      ;
33    }
34  }
35
36  __interrupt(vect=14) void INT_IRQ0(void)
37  {
38    LedSet( LED0 | LED1 | LED2 );  /* LED の点灯 */
39  }
```

End Of List

実行結果

最初は LED はすべて消灯。
タクトスイッチを押したら LED がすべて点灯する (その後はタクトスイッチを押しても何も起こらない)。

プログラム解説 (04_IRQ02.c)

```
12    #include <machine.h>
```

割込みの設定のためにヘッダファイル machine.h を include します。

```
18    set_imask_ccr(1);   /* 割込み不可 */
……
24    set_imask_ccr(0);   /* 割込み許可 */
```

割込みを設定する際には、割込みを禁止しておいてから設定します。

18 行目で関数 set_imask_ccr を用いて、コンディションコードレジスタ (CCR) の割り込みマスクビット (I) を 1 に設定して割込みを禁止しています。

24 行目で割り込みマスクビット (I) を 0 に設定して、割込みを許可しています。

```
22    IENR1.BIT.IEN0=1;   /* IRQ0 端子の割り込み要求 */
```

今回のプログラムでは割込みを利用するので、割り込みイネーブルレジスタ 1(IENR1) の IRQ0 割り込み要求イネーブルを設定しています (P.221 の表 9.4)。コンディションコードレジスタ (CCR) の割り込みマスクビット (I) と両方をセットする必要があることに注意してください。

```
36    __interrupt(vect=14) void INT_IRQ0(void)
37    {
38      LedSet( LED0 | LED1 | LED2 );  /* LED の点灯 */
39    }
```

36 行目が割込み関数の定義です。IRQ0 割り込み要求フラグがセットされると、この関数が実行されます。

✍ 課題 8.1.1 (提出) IRQ0 割込みで LED がカウントアップするプログラム

初期状態は LED がすべて消灯とします。

タクトスイッチを押すと LED がカウントアップするプログラムを作ってください。2 回目以降はスイッチを押しても何も起こらないようにしてください。

IRQ0 割込みを使ってください。

プロジェクト名：e04_IRQ02

9 タイマ A の利用

9.1 タイマ A

組込み機器においては、時間にかかわる様々な機能が必要になります。これまでにも LED の点滅など一定の待ち時間を作ってきましたが、その時間設定はかなり大まかなものでした。タイマを利用するとこれを正確な時間で制御することができます。

9.1.1 タイマとは

タイマとは、時間にかかわる様々な機能です。

一定の時間を計測したり、特定の端子から正確な周期で ON、OFF を繰り返すパルスを出力したり、入力されたパルスの周期を測定したりと多くの機能を持っています。

マイコンボードに実装されている部品で時間に関係しているものは、通常は発信子だけです。したがって、タイマはこの発信子をもとに時間の計測をします。この場合計測できるのは時間間隔であって、時刻 (何時何分) を知ることはできません。

また、マイコンの発信子は数十 MHz と周波数が高いので、数ミリ秒をカウントするのにも何万回もカウントしなくてはなりません。そのためには大きなレジスタが必要になってしまいます。このために、発信子の周波数を分周して利用する機能が備わっています。これは 2 分周であれば周波数を半分に、4 分周であれば 4 分の 1 にして利用するというものです。20MHz の 2 分周は 10MHz、4 分周は 5MHz です。

タイマはカウントできるレジスタのビット数によって、8 ビットタイマとか 16 ビットタイマなどと呼ばれます。ビット数が大きい程計測できる時間間隔は大きいことになります。

H8/3694F には 8 ビットのタイマであるタイマ A と、8 ビットのタイマのタイマ V と、16 ビットタイマのタイマ W があります。

タイマ A とタイマ V は、機能が異なります。

タイマ A は 8 ビット (256) カウントして割込みを発生さることができます。また、CPU の発信子とは別の 32.768kHz 水晶発振器を使用することにより、1s、0.5s、0.25s、31.25ms の時間計測を行うことができます。

タイマ V は設定した値とカウンタが一致することによって (コンペアマッチ)、任意の周期で ON、OFF

を繰り返すパルス (PWM) を出力したり、特定のポート (TMOV) から ON、OFF などを出力することができます。

このテキストでは最終的に、タイマ A は音楽を演奏する際の音の長さを計測するのに利用し、タイマ V は音階を鳴らすのに利用します。タイマ W はサーボモータを制御するために用います。

それではまずタイマ A の使い方からみていきましょう。

9.1.2 タイマ A

タイマ A は 32.768kHz 水晶発振器を取り付けることによって、時計用タイムベースのタイマとして利用できます。この場合 4 種類のオーバフロー周期 (1s、0.5s、0.25s、31.25ms) が選択可能です。

しかし、マイコンボード MB-H8A は 32.768kHz 水晶発振器はついていません (自分でつけることは可能です)。そこでここではインターバルタイマを使って CPU の 20MHz の発信子を分周して使うことにします。

1 秒ごとに LED をカウントアップさせてみましょう。

9.1.3 回路図

タイマのためには特別の回路は必要ありません。LED を点滅させるために、P.91 の図 4.2 と同じ回路を利用します。

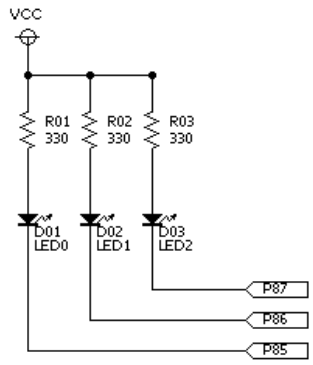


図 9.1　LED の回路図

9.1.4 関連レジスタ

タイマ A には以下のレジスタがあります。

- タイマモードレジスタ A(TMA)
- タイマカウンタ A(TCA)

上記のレジスタは次のアドレスに割り振られています。

表 9.1 タイマ A 関連レジスタのアドレス

レジスタ名	アドレス
タイマモードレジスタ A(TMA)	H'FFA6
タイマカウンタ A(TCA)	H'FFA7

タイマモードレジスタ A(TMA)

タイマ A は動作モードの選択、および分周クロック出力、入力クロックの選択を行います。

表 9.2 タイマモードレジスタ A(TMA)

ビット	ビット名	初期値	R/W	説 明
7	TMA7	0	R/W	アウトプットセレクト 7~5
6	TMA6	0	R/W	TMOW 端子から出力するクロックを選択します。
5	TMA5	0	R/W	000 : ϕ/32 001 : ϕ/16 010 : ϕ/8 011 : ϕ/4 100 : ϕW/32 (ϕW=32.768kHz) 101 : ϕW/16 110 : ϕW/8 111 : ϕW/4
4	-	1	-	
3	TMA3	0	R/W	インターナルクロックセレクト 3 タイマAの動作モードを選択します。 0 : プリスケーラ S の出力をカウントする インターバルタイマとして動作します。 1 : プリスケーラ W の出力をカウントする 時計用タイムベースとして動作します。
2	TMA2	0	R/W	インターナルクロックセレクト 2~0
1	TMA1	0	R/W	TMA3 = 0 のとき、TCA に入力するクロックを選択します。
0	TMA0	0	R/W	000 : ϕ/8192 001 : ϕ/4096 010 : ϕ/2048 011 : ϕ/512 100 : ϕ/256 101 : ϕ/128 110 : ϕ/32 111 : ϕ/8 TMA3 = 1 のとき、オーバフロー周期を選択します (ϕW として 32.768KHz の水晶発振器を使用した場合) 000 : 1s 001 : 0.5s 010 : 0.25s 010 : 0.03125s 1xx : PSW と TCA は共にリセット状態になります。

タイマカウンタ A(TCA)

TCA は 8 ビットのリード可能な (値が増えていく) アップカウンタで、入力する内部クロックによりカウントアップされます。入力するクロックは TMA の TMA3~TMA0 により選択します。TCA の値は、アクティブモード時は CPU からリードできます。TCA が (256 以上になり) オーバフローすると、割り込みフラグレジスタ 1(IRR1) の IRRTA が 1 にセットされます。TCA は TMA の TMA3~TMA2 を $(11)_2$ にセットすることでクリアできます。TCA の初期値は 0x00 です。

タイマ A のレジスタ定義 (iodefine.h の一部を抜粋)

以下は、タイマ A のレジスタ定義です。

タイマ A のレジスタ定義 (iodefine.h の一部を抜粋)

```
struct st_ta {                                          /* struct TA    */
             union {                                    /* TMA          */
                   unsigned char BYTE;                  /*  Byte Access */
                   struct {                             /*  Bit  Access */
                          unsigned char CKSO:3;         /*    CKSO      */
                          unsigned char     :1;         /*              */
                          unsigned char CKSI:4;         /*    CKSI      */
                          }      BIT;                   /*              */
                   }            TMA;                    /*              */
             unsigned char      TCA;                    /* TCA          */
};                                                      /*              */

......

#define TA      (*(volatile struct st_ta    *)0xFFA6)   /* TA    Address*/
```

9.1.5 基本的なプログラム (割込みなし)

それでは、インターナルクロック (内部クロック) を使って、1 秒のループを作ってみましょう。今回は割込みを使わず、タイマカウンタ A(TCA) の値を常時監視することで、時間経過を測定します。

マイコンボード MB-H8A のクロックは 20MHz ですので、8192 分周しても約 2441Hz です。タイマカウンタ A(TCA) で 256 カウントしても、約 104.9 ミリ秒です。

そこで 244 カウントして 0.1 秒の待ち時間を作り、それを 10 回繰り返せば約 1 秒になります。ここでは、このアルゴリズムでプログラムをしてみましょう。

20MHz を 8192 分周したときに、0.1 秒カウントするにはタイマカウンタ A(TCA) の値がいくつになればよいか、その計算方法を以下にまとめておきましょう。

タイマカウンタ A(TCA) の値を計算 (20MHz を 8192 分周したときに、0.1 秒をカウント)

20MHz を 8192 分周すると、

$$\frac{20000000}{8192}(Hz)$$

になります。
この時のパルスの周期は、周波数の逆数を取って、

$$\frac{8192}{20000000}(s)$$

です。
TCA に設定する値を仮に x とすると、上記の値に x をかけた結果が 0.1s になれば良いわけです。
したがって、

$$\frac{8192}{20000000} \times x = 0.1$$

となる x を求めればよいわけです。
結果は、

$$x = 244.14$$

となります。

以下にサンプルプログラムを示しますので、必要な設定は適宜行ってください (P.207 もしくは P.156 の図 6.6 以降)。

05_TMRA01.c

```
 1  /************************************************************************/
 2  /*                                                                      */
 3  /*  FILE        :05_TMRA01.c                                            */
 4  /*  DESCRIPTION :タイマ A で 1 秒ごとに LED カウントアップ               */
 5  /*  CPU TYPE    :H8/3694F                                               */
 6  /*                                                                      */
 7  /*  LED0 P85                                                            */
 8  /*  LED1 P86                                                            */
 9  /*  LED2 P87                                                            */
10  /*                                                                      */
11  /************************************************************************/
12
13  #include "led.h"
14  #include "iodefine.h"
15
16  #define LED_MAX (LED0 | LED1 | LED2)
17
18  /***************************************
19    タイマ A の初期化 (8192 分周)
20  ***************************************/
21  void TimerAInit(void)
22  {
23    TA.TMA.BIT.CKSI=0; /* インターバルタイマ 8192 分周 */
24  }
25
26  /***************************************
27    TCA の初期化 (カウントクリア)
28  ***************************************/
29  void TCAInit(void)
30  {
31    TA.TMA.BIT.CKSI=0xC;  /* TCA 初期化 */
32  }
33
34  /***************************************
35    タイマ A 1 秒待ち関数 (0.1 秒*10)
36  ***************************************/
37  void Wait1S(void)
38  {
```

```
39    int ta_count=0;
40
41    while(ta_count<10){
42      while(TA.TCA<244){ /* 0.1S カウントを待つ */
43        ;
44      }
45      ta_count++;
46      TCAInit();
47      TimerAInit();
48    }
49  }
50
51  /***************************************
52    main 関数
53  ***************************************/
54  void main(void)
55  {
56    unsigned char led_data;
57    LedInit();  /* LED 接続ポートの初期化 */
58    TimerAInit();  /* TimerA 初期化 */
59    while(1){
60      for(led_data=0; led_data<=LED_MAX; led_data++){
61        Wait1S();
62        LedSet( led_data );  /* LED 点灯 */
63      }
64    }
65  }
```

End Of List

実行結果

最初は LED がすべて消灯。
1 秒ごとに LED がカウントアップ。

プログラム解説 (05_TMRA01.c)

```
21    void TimerAInit(void)
22    {
23      TA.TMA.BIT.CKSI=0; /* インターバルタイマ 8192 分周 */
24    }
```

タイマ A の初期化関数です。タイマモードレジスタ A(TMA) の表 9.2 から分かるように、インターバルタイマを 8192 分周するには、TMA3~TMA0 までの 4 ビットをすべて 0 に設定すればよいわけです。

iodefine.h で行われているタイマ A のレジスタ定義 (P.216) によると、この 4 ビットは TA.TMA.BIT.CKSI で定義されていることが分かります。

23 行目ではこの定義を用いてインターバルタイマの分周を設定しています。

```
29    void TCAInit(void)
30    {
31      TA.TMA.BIT.CKSI=0xC;  /* TCA 初期化 */
32    }
```

タイマモードレジスタ A(TMA) を初期化する関数を定義しました。

P.215 の表 9.2 によると、TMA3=1、TMA2=1、TMA1=0、TMA0=0 と設定することによってタイマモードレジスタ A(TMA) がリセットされることが分かります。この 4 ビットは TA.TMA.BIT.CKSI で定義されるので (P.216 のレジスタ定義参照)、0xC をセットしました。

```
41    while(ta_count<10){
42      while(TA.TCA<244){ /* 0.1S カウントを待つ */
43        ;
44      }
45      ta_count++;
46      TCAInit();
47      TimerAInit();
48    }
```

37 行目から 48 行目まではタイマ A で 1 秒を計測する関数です。

42 行目から 44 行目までで、タイマカウンタ A(TCA) が 244 になるのを待つことで約 0.1 秒の待ちを作っています。41 行目から 48 行目まででそれを 10 回繰り返して 1 秒を計測しているわけです。

46 行目の TCAInit 関数でタイマカウンタ A(TCA) をクリアし、47 行目の初期化関数を実行してタイマ A を再度動かしています。

9.1.6 基本的なプログラム (割込みあり)

それでは、1 秒の待ち時間をタイマ A の割込みを使って実現してみましょう。

時計用タイムベースを用いれば簡単なのですが、ここではインターナルクロックを用いて実現することにします。

タイマカウンタ A(TCA) がオーバーフローした場合 (ここでは 256 以上になった場合) のみ割込みは起こりますから、この条件で考えなくてはなりません。

8192 分周した場合に、約 0.1049 秒でタイマカウンタ A(TCA) がオーバーフローします。1 秒カウントするには 9.537 と中途半端な回数になってしまいます。

分周を 4096 にすると、約 0.0524 秒でオーバーフローし、1 秒カウントするのに 19.07 回となります。誤差は 1 パーセント以下なのでこれで良いことにしましょう。

前と同様な計算方法でまとめておくと、以下のようになります。

タイマカウンタ A(TCA) の値を計算 (20MHz を 4096 分周したときに、1 秒をカウント)

20MHz を 4096 分周すると、

$$\frac{20000000}{4096}(Hz)$$

になります。
この時のパルスの周期は、周波数の逆数を取って、

$$\frac{4096}{20000000}(s)$$

です。
TCA は 256 でオーバーフローするので、カウントすべきオーバーフローの回数を仮に x とすると、上記の値に 256×x をかけた結果が 1s になれば良いわけです。
したがって、

$$\frac{4096}{20000000} \times x \times 256 = 1$$

となる x を求めればよいわけです。
結果は、

$$x = 19.07$$

となります。カウントする回数は整数値なので、19 回ということになります。
19 回カウントしたとき、約 0.996 秒です。

intprg.c の INT_TimerA をコメントアウト

```
// vector 19 Timer A Overflow
//__interrupt(vect=19) void INT_TimerA(void) {/* sleep(); */}
```

9.1.7 関連レジスタ

割込みに関しては、第 8 章「IRQ の利用」(P.199) で説明したこととほとんど同じです。以下に、関連レジスタの説明を再度しておきます。

例外処理関係で今回利用するレジスタは以下のものです。

- 割り込みイネーブルレジスタ 1(IENR1)
- 割り込みフラグレジスタ 1(IRR1)

上記のレジスタは次のアドレスに割り振られています。

表 9.3 例外処理関連レジスタのアドレス

レジスタ名	アドレス
割り込みイネーブルレジスタ 1(IENR1)	H'FFF4
割り込みフラグレジスタ 1(IRR1)	H'FFF6

割り込みイネーブルレジスタ 1(IENR1)

IENR1 は直接遷移割り込み、タイマ A オーバフロー割り込みおよび外部端子割り込みをイネーブルにします。「イネーブルにする」とは「有効にする」という意味です。

表 9.4 割り込みイネーブルレジスタ 1(IENR1)

ビット	ビット名	初期値	R/W	説 明
7	IENDT	0	R/W	直接遷移割り込み要求イネーブル このビットを 1 にセットすると 直接遷移割り込み要求がイネーブルになります。
6	IENTA	0	R/W	タイマ A 割り込み要求イネーブル このビットを 1 にセットすると タイマ A のオーバフロー割り込み要求がイネーブルになります。
5	IENWP	0	R/W	ウェイクアップ割り込み要求イネーブル このビットは $\overline{WKP5}$ ~ $\overline{WKP0}$ 端子共通のイネーブルビットで 1 にセットすると割り込み要求がイネーブルになります。
4	-	1	-	リザーブビットです。読み出すと常に 1 が読み出されます。
3	IEN3	0	R/W	IRQ3 割り込み要求イネーブル このビットを 1 にセットすると $\overline{IRQ3}$ 端子の 割り込み要求がイネーブルになります。
2	IEN2	0	R/W	IRQ2 割り込み要求イネーブル このビットを 1 にセットすると $\overline{IRQ2}$ 端子の 割り込み要求がイネーブルになります。
1	IEN1	0	R/W	IRQ1 割り込み要求イネーブル このビットを 1 にセットすると $\overline{IRQ1}$ 端子の 割り込み要求がイネーブルになります。
0	IEN0	0	R/W	IRQ0 割り込み要求イネーブル このビットを 1 にセットすると $\overline{IRQ0}$ 端子の 割り込み要求がイネーブルになります。

割り込みイネーブルレジスタをクリアすることにより割り込み要求をディスエーブル (無効) にする場合、または割り込みフラグレジスタをクリアする場合は、割り込み要求をマスクした状態（I = 1）で行ってください。

割り込みフラグレジスタ 1(IRR1)

IRR1 は直接遷移割り込み、タイマ A オーバフロー割り込み、IRQ3~IRQ0 割り込み要求ステータスフラグレジスタです。

表 9.5　割り込みフラグレジスタ 1(IRR1)

ビット	ビット名	初期値	R/W	説 明
7	IRRDT	0	R/W	直接遷移割り込み要求フラグ [セット条件] SYSCR2 の DTON に 1 をセットした状態でスリープ命令を実行し直接遷移したとき [クリア条件] 0 をライトしたとき
6	IRRTA	0	R/W	タイマ A 割り込み要求フラグ [セット条件] タイマ A がオーバフローしたとき [クリア条件] 0 をライトしたとき
5 4	- -	1 1	- -	リザーブビットです。読み出すと常に 1 が読み出されます。
3	IRRI3	0	R/W	IRQ3 割り込み要求フラグ [セット条件] $\overline{IRQ3}$ 端子が割り込み入力に設定され、指定されたエッジを検出したとき [クリア条件] 0 をライトしたとき
2	IRRI2	0	R/W	IRQ2 割り込み要求フラグ [セット条件] $\overline{IRQ2}$ 端子が割り込み入力に設定され、指定されたエッジを検出したとき [クリア条件] 0 をライトしたとき
1	IRRI1	0	R/W	IRQ1 割り込み要求フラグ [セット条件] $\overline{IRQ1}$ 端子が割り込み入力に設定され、指定されたエッジを検出したとき [クリア条件] 0 をライトしたとき
3	IRRI0	0	R/W	IRQ0 割り込み要求フラグ [セット条件] $\overline{IRQ0}$ 端子が割り込み入力に設定され、指定されたエッジを検出したとき [クリア条件] 0 をライトしたとき

さらに、割込みを実行するためにはコンディションコードレジスタ (CCR) の割込みマスクビットをクリアする必要があります。

ヘッダファイル machine.h を読み込むことで、set_imask_ccr が使えるようになります。コンディションコードレジスタ (CCR) の割込みマスクビットはこれを用いて設定することにします。

関数の仕様は以下の通りです (参考文献 [2]P283 からの引用)。

```
void set_imask_ccr (unsigned char mask)

    説明：コンディションコードレジスタ (CCR) の割り込みマスクビット (I) に mask 値 (0 または 1) を設定します。
  ヘッダ：<machine.h>
    引数：mask 値 (0 または 1)

      例：#include <machine.h> /* 必ず<machine.h>をインクルードします。 */
          void main(void)
          {
            set_imask_ccr(0);  /* 割り込みマスクビットをクリアします。*/
          }
```

タイマ A の割込みレジスタ定義 (iodefine.h の一部を抜粋)

タイマ A の割込みレジスタ定義 (iodefine.h の一部を抜粋) を掲載しておきます。

タイマ A の割込みレジスタ定義 (iodefine.h の一部を抜粋)

```
union un_ienr1 {                                        /* union IENR1  */
               unsigned char BYTE;                      /*  Byte Access */
               struct {                                 /*  Bit  Access */
                      unsigned char IENDT:1;            /*    IENDT     */
                      unsigned char IENTA:1;            /*    IENTA     */
                      unsigned char IENWP:1;            /*    IENWP     */
                      unsigned char      :1;            /*              */
                      unsigned char IEN3 :1;            /*    IEN3      */
                      unsigned char IEN2 :1;            /*    IEN2      */
                      unsigned char IEN1 :1;            /*    IEN1      */
                      unsigned char IEN0 :1;            /*    IEN0      */
                      }      BIT;                       /*              */
};                                                      /*              */
union un_irr1 {                                         /* union IRR1   */
              unsigned char BYTE;                       /*  Byte Access */
              struct {                                  /*  Bit  Access */
                     unsigned char IRRDT:1;             /*    IRRDT     */
                     unsigned char IRRTA:1;             /*    IRRTA     */
                     unsigned char      :2;             /*              */
                     unsigned char IRRI3:1;             /*    IRRI3     */
                     unsigned char IRRI2:1;             /*    IRRI2     */
                     unsigned char IRRI1:1;             /*    IRRI1     */
                     unsigned char IRRI0:1;             /*    IRRI0     */
                     }      BIT;                       /*              */
};                                                      /*              */

......

#define IENR1   (*(volatile union  un_ienr1 *)0xFFF4)   /* IENR1 Address*/
#define IRR1    (*(volatile union  un_irr1  *)0xFFF6)   /* IRR1  Address*/
```

では、割込みを使ってタイマ A で 1 秒ごとに LED をカウントアップするプログラムを作ってみましょう。

05_TMRA02.c

```
 1  /************************************************************************/
 2  /*                                                                      */
 3  /*  FILE        :05_TMRA02.c                                            */
 4  /*  DESCRIPTION :タイマ A で 1 秒ごとに LED カウントアップ (割込み)      */
 5  /*  CPU TYPE    :H8/3694F                                               */
 6  /*                                                                      */
 7  /*  LED0 P85                                                            */
 8  /*  LED1 P86                                                            */
 9  /*  LED2 P87                                                            */
10  /*                                                                      */
11  /************************************************************************/
12
13  #include <machine.h>
14  #include "led.h"
15  #include "iodefine.h"
16
17  #define LED_MAX (LED0 | LED1 | LED2)
18
19  static int tma_flag=0;
20
21  /*****************************************
22    タイマ A の初期化 (4096 分周)
23  *****************************************/
24  void TimerAInit(void)
25  {
26    set_imask_ccr(1);
27    TA.TMA.BIT.CKSI=1; /* インターバルタイマ 4096 分周 */
28    IENR1.BIT.IENTA=1; /* タイマ A のオーバフロー割り込み要求 */
29    IRR1.BIT.IRRTA=0;
30    set_imask_ccr(0);
31  }
32
33  /*****************************************
34    main 関数
35  *****************************************/
```

```
36  void main(void)
37  {
38    unsigned char led_data;
39    LedInit();  /* LED 接続ポートの初期化 */
40    TimerAInit();  /* TimerA 初期化 */
41    while(1){
42      for(led_data=0; led_data<=LED_MAX; led_data++){
43        while(tma_flag<19){
44          ;
45        }
46        tma_flag=0;
47        LedSet( led_data );  /* LED 点灯 */
48      }
49    }
50  }
51
52  /***************************************
53    タイマ A 割込み関数
54  ***************************************/
55  //vector 19 Timer A Overflow
56  __interrupt(vect=19) void INT_TimerA(void)
57  {
58    tma_flag++;
59    IRR1.BIT.IRRTA=0;
60  }
```

End Of List

実行結果

最初はすべて消灯。
1 秒ごとにカウントアップ。

プログラム解説 (05_TMRA02.c)

```
13    #include <machine.h>
```

割込みの設定のためにヘッダファイル machine.h を include します。

```
19    static int tma_flag=0;
```

割込みの回数をカウントするための変数です。現在のタイマの設定では、19 回カウントすると約 1 秒です。

```
26    set_imask_ccr(1);
……
30    set_imask_ccr(0);
```

割込みを設定する際には、割込みを禁止しておいてから設定します。

26 行目で関数 set_imask_ccr を用いて、コンディションコードレジスタ (CCR) の割り込みマスクビット (I) を 1 に設定して割込みを禁止しています。

30 行目で割り込みマスクビット (I) を 0 に設定して、割込みを許可しています。

関数 set_imask_ccr については、P.199 の仕様を確認してください。

```
27    TA.TMA.BIT.CKSI=1; /* インターバルタイマ 4096 分周 */
```

タイマモードレジスタ A (TMA) を設定して、分周を行っています。

P.216 レジスタ定義ファイルを見ると、ビットフィールドのメンバ CKSI は下位 4 ビットにあたるので、ここを 1 に設定すると 4096 分周になることが分かります。

```
28    IENR1.BIT.IENTA=1; /* タイマ A のオーバフロー割り込み要求 */
29    IRR1.BIT.IRRTA=0;
```

タイマ A の割込みレジスタの設定です。

割り込みイネーブルレジスタ 1(IENR1) に関しては P.221 の表 9.4 を、割り込みフラグレジスタ 1(IRR1) に関しては P.222 の表 9.5 を参照してください。

プログラム (05_TMRA02.c) の 28 行目では、タイマ A のオーバフロー割り込み要求を可能にしています。

29 行目ではタイマ A の割り込み要求フラグを (0 に) クリアしています。このフラグはタイマ A のオーバフローが起こったときに (1 に) セットされます。

```
40    TimerAInit();  /* TimerA 初期化 */
```

24 行目から 31 行目までの、タイマ A の初期化関数を呼び出しています。

```
43    while(tma_flag<19){
44      ;
45    }
46    tma_flag=0;
```

43 行目から 45 行目までは、変数 tma_flag を用いて、割込み回数が 19 回になるのを待っています。19 回になったらループを抜け、46 行目で値を 0 に戻しています。

```
56    __interrupt(vect=19) void INT_TimerA(void)
57    {
58      tma_flag++;
59      IRR1.BIT.IRRTA=0;
60    }
```

タイマ A のオーバフロー割り込み関数です。

P.220 で INT_TimerA をコメントアウトしましたが、その代わりにこちらに記述しています。

58 行目では、割込みのたびに変数 tma_flag に 1 を足しています。

59 行目ではタイマ A の割り込み要求フラグを (0 に) クリアしています。

これまでは、基本的な機能を理解した時点で関数化を行ってきましたが、タイマは機能が多いので、すべての機能を使えるように関数化しても便利にはなりません。

そこで、音楽を鳴らすとか、サーボモータを動かすとか目的別に関数を作ることにします。ここでは、タイマ A を用いた関数は作りません。

✍ 課題 9.1.1 (提出) タイマ A の分周を変更してみる

上のサンプルではクロックの分周は 4096 でしたが、これを変更して、1 秒ごとに LED がカウントアップするプログラムを作ってみましょう。

分周を 2048 にしてプログラムしてください。

それができたら、他の分周の時にもどこを変更すればよいか考えてみましょう。

割り込みが、オーバーフロー割込みであることに注意してください。

プロジェクト名：e05_TMRA02

10 タイマＶの利用(音楽)

10.1 タイマＶ

タイマＶは 8 ビットタイマです。外部のイベントのカウントが可能なほか、2 本のレジスタとのコンペアマッチ信号によりカウンタのリセット、割り込み要求、任意のデューティ比のパルス出力などが可能です。

ここでは、このタイマＶを用いて決められた周波数の音を鳴らしてみましょう。最終的にはタイマＡとタイマＶを用いて音楽を演奏してみることにします。

そのために、まずは音階と周波数の関係から説明します。

10.1.1 音階と周波数

人間の視聴可能な周波数はおよそ 20Hz から 18000Hz 程度と言われています。

通常音階の基準音として使用されるラの音 (時報の最初の 3 つの音の音程) は、440Hz 付近の音が使われているようです。

1 オクターブの中には、半音間隔で、

13	ド
12	シ
11	ラ♯ (シ♭)
10	ラ
9	ソ♯ (ラ♭)
8	ソ
7	ファ♯ (ソ♭)
6	ファ
5	ミ
4	レ♯ (ミ♭)
3	レ
2	ド♯ (レ♭)
1	ド

と、12 の間隔があります。

各音程の周波数は、1 オクターブ (12 半音。例えば、ドの音に対して一つ上のドの音) 上がる毎に倍になるように、音階に対して等比数列的に増えていきます。

したがって、基準音のラの音の一つ上のラは、

$$440 \times 2 = 880$$

と、なります。

また、基準音のラの音のすぐ上のラ ♯ は、

$$440 \times 2^{1/12} = 466.16$$

と、なるわけです。

✍ 課題 10.1.1　(提出) 基準音のラの下のドから、2 オクターブ上のドまでの周波数を計算

上の説明をもとに、2 オクターブ分の周波数を計算しましょう (基準音のラの下のドから、2 オクターブ上のドまで)。

10.1.2 圧電スピーカー

ここで利用するのは、株式会社村田製作所の圧電スピーカー PKM13EPYH4000-A0 です。極性はありませんので、2 本の足のどちらをグランドにつないでもかまいません。

この圧電スピーカーは電圧をかけただけでは鳴りません。数 kHz 程度の信号を入力すると鳴るようにできています。

10.1.3 回路図

以下に今回使うブザーの回路図を示します (図 10.1)。

音階出力のためにタイマ V を利用します。波形は、タイマ V の波形出力端子 (TMOV) からトグル出力することにします。

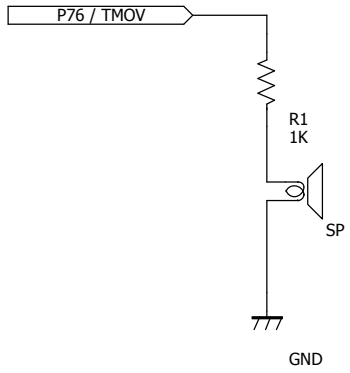


図 10.1 圧電スピーカー回路図

10.1.4 出力端子

タイマ V の端子構成のうち、今回利用する出力端子を示します。

表 10.1 タイマ V の出力端子

名称	略称	機能
タイマ V 出力	TMOV	タイマ V の波形出力端子

10.1.5 関連レジスタ

タイマ V には以下のレジスタがあります。

- タイマカウンタ V(TCNTV)
- タイムコンスタントレジスタ A(TCORA)
- タイムコンスタントレジスタ B(TCORB)
- タイマコントロールレジスタ V0(TCRV0)
- タイマコントロール／ステータスレジスタ V(TCSRV)
- タイマコントロールレジスタ V1(TCRV1)

上記のレジスタは次のアドレスに割り振られています。

表 10.2　タイマ V 関連レジスタのアドレス

レジスタ名	アドレス
タイマカウンタ V(TCNTV)	H'FFA4
タイムコンスタントレジスタ A(TCORA)	H'FFA2
タイムコンスタントレジスタ B(TCORB)	H'FFA3
タイマコントロールレジスタ V0(TCRV0)	H'FFA0
タイマコントロール／ステータスレジスタ V(TCSRV)	H'FFA1
タイマコントロールレジスタ V1(TCRV1)	H'FFA5

タイマカウンタ V(TCNTV)

TCNTV は、8 ビットの (1 ずつ増えていく) アップカウンタです。クロックをカウントした値がここに保存されます。クロックは TCRV0 の CKS2∼CKS0 により選択します。TCNTV の値は CPU から常にリード／ライトできます。TCNTV は、外部リセット入力信号またはコンペアマッチ信号 A、コンペアマッチ信号 B によりクリアすることができます。いずれの信号でクリアするかは、TCRV0 の CCLR1、CCLR0 により選択します。また、TCNTV がオーバフローすると、TCSRV の OVF が 1 にセットされます。

TCNTV の初期値は H'00 です。

ここでは、タイマカウンタ V(TCNTV) はクロックをカウントするのに利用します。これはマイコンが自動的にやってくれますので、プログラムで操作する必要はありません。

タイムコンスタントレジスタ A、B(TCORA、TCORB)

TCORA と TCORB は同一機能をもっています。TCORA は 8 ビットのリード／ライト可能なレジスタです。TCORA の値は TCNTV と常に比較され、一致すると TCSRV の CMFA が 1 にセットされます。このとき TCRV0 の CMIEA が 1 なら CPU に対して割り込み要求を発生します。また、この一致信号（コンペアマッチ A）と TCSRV の OS3∼OS0 の設定により、TMOV 端子からのタイマ出力を制御することができます。

TCORA、TCORB の初期値は H'FF です。

ここではすでに説明したように、音階出力のためにタイマ V を利用します。波形は、タイマ V の波形出力端子 (TMOV) からトグル出力します。その際の周期を設定するのに、タイムコンスタントレジスタ A(TCORA) を利用します。タイムコンスタントレジスタ B(TCORB) は使いません。

タイマコントロールレジスタ V0(TCRV0)

TCRV0 は TCNTV の入力クロックの選択、TCNTV のクリア条件指定、各割り込み要求の制御を行います。「イネーブルにする」とは「有効にする」という意味です。

表 10.3 タイマコントロールレジスタ V0(TCRV0)

ビット	ビット名	初期値	R/W	説 明
7	CMIEB	0	R/W	コンペアマッチインタラプトイネーブル B 1 のとき TCSRV の CMFB による割り込み要求がイネーブルになります。
6	CMIEA	0	R/W	コンペアマッチインタラプトイネーブル A 1 のとき TCSRV の CMFA による割り込み要求がイネーブルになります。
5	OVIE	0	R/W	タイマオーバフローインタラプトイネーブル 1 のとき TCSRV の OVF による割り込み要求がイネーブルになります。
4 3	CCLR1 CCLR0	0 0	R/W R/W	カウンタクリア 1~0 TCNTV のクリア条件を指定します。 00 : クリアされません。 01 : コンペアマッチ A でクリアされます。 10 : コンペアマッチ B でクリアされます。 11 : TMRIV 端子の立ち上がりエッジにでクリアされます。 クリア後の TCNTV の動作は TCRV1 の TRGE によって異なります。
2 1 0	CKS2 CKS1 CKS0	0 0 0	R/W R/W R/W	クロックセレクト 2~0 TCRV1 の ICKS0 との組合わせで、TCNTV に入力するクロックと カウント条件を選択します。表 10.4 を参照してください。

表 10.4 TCNTV に入力するクロックとカウント条件

TCRV0			TCRV1	説明
ビット 2	ビット 1	ビット 0	ビット 0	
CKS2	CKS1	CKS0	ICKS0	
0	0	0	-	クロック入力禁止
0	0	1	0	内部クロック ϕ/4 立ち下がりエッジでカウント
0	0	1	1	内部クロック ϕ/8 立ち下がりエッジでカウント
0	1	0	0	内部クロック ϕ/16 立ち下がりエッジでカウント
0	1	0	1	内部クロック ϕ/32 立ち下がりエッジでカウント
0	1	1	0	内部クロック ϕ/64 立ち下がりエッジでカウント
0	1	1	1	内部クロック ϕ/128 立ち下がりエッジでカウント
1	0	0	-	クロック入力禁止
1	0	1	-	外部クロックの立ち上がりエッジでカウント
1	1	0	-	外部クロックの立ち下がりエッジでカウント
1	1	1	-	外部クロックの立ち上がり／立ち下がり両エッジでカウント

タイマコントロール／ステータスレジスタ V(TCSRV)

TCSRV はステータスフラグの表示およびコンペアマッチによる出力制御を行います。

表 10.5　タイマコントロール／ステータスレジスタ V(TCSRV)

ビット	ビット名	初期値	R/W	説 明
7	CMFB	0	R/W	コンペアマッチフラグ B [セット条件] TCNTV の値と TCORB の値が一致したとき [クリア条件] CMFB = 1 の状態で、CMFB をリードした後、 CMFB に 0 をライトしたとき
6	CMFA	0	R/W	コンペアマッチフラグ A [セット条件] TCNTV の値と TCORA の値が一致したとき [クリア条件] CMFB = 1 の状態で、CMFA をリードした後、 CMFA に 0 をライトしたとき
5	OVF	0	R/W	タイマオーバーフローフラグ [セット条件] TCNTV の値が H'FF から H'00 にオーバフローしたとき [クリア条件] OVF = 1 の状態で、OVF をリードした後、 OVF に 0 をライトしたとき
4	-	1	-	リザーブビットです。読み出すと常に 1 が読み出されます。
3 2	OS3 OS2	0 0	R/W R/W	アウトプットセレクト 3~2 TCORB と TCNTV のコンペアマッチによる TMOV 端子の出力方法を選択します。 00：変化しない。 01：0 出力。 10：1 出力。 11：トグル出力。
1 0	OS1 OS0	0 0	R/W R/W	アウトプットセレクト 1~0 TCORA と TCNTV のコンペアマッチによる TMOV 端子の出力方法を選択します。 00：変化しない。 01：0 出力。 10：1 出力。 11：トグル出力。

タイマコントロールレジスタ V1(TCRV1)

TCRV1 は TRGV 端子のエッジセレクト、TRGV 入力イネーブル、TCNTV の入力クロックの選択を行います。

表 10.6 タイマコントロールレジスタ V1(TCRV1)

ビット	ビット名	初期値	R/W	説 明
7〜5	-	すべて 1	-	リザーブビットです。読み出すと常に 1 が読み出されます。
4 3	TVEG1 TVEG0	0 0	R/W R/W	TRGV 入力エッジセレクト TRGV 端子の入力エッジを選択します。 00：TRGV からのトリガ入力を禁止。 01：立ち上がりエッジを選択。 10：立ち下がりエッジを選択。 11：立ち上がり／立ち下がり両エッジを選択。
2	TRGE	0	R/W	TVEG1、TVEG0 で選択されたエッジの入力により、 TCNTV カウントアップが開始します。 0：TRGV 端子入力による TCNTV カウントアップの開始と コンペアマッチによる TCNTV クリア時の TCNTV カウントアップの停止を禁止。 1：TRGV 端子入力による TCNTV カウントアップの開始と コンペアマッチによる TCNTV クリア時の TCNTV カウントアップの停止を許可。
1	-	1	-	リザーブビットです。リードすると常に 1 が読み出されます。
0	ICKS0	0	R/W	インターナルクロックセレクト 0 TCRV0 の CKS2〜CKS0 との組合せで、 TCNTV に入力するクロックを選択します。 表 10.4 を参照してください。

なお、汎用 I/O ポート P76 と TMOV 出力端子の切り替えは、以下のようになります。

表 10.7 P76/TMOV 端子

レジスタ名	TCSRV	PCR7	
ビット名	OS3~OS0	PCR76	機能
設定値	0000	0	P76 入力端子
		1	P76 出力端子
	上記以外	X	TMOV 出力端子

タイマ V のレジスタ定義 (iodefine.h の一部を抜粋)

タイマ V のレジスタ定義 (iodefine.h の一部を抜粋) を掲載しておきます。

タイマ V のレジスタ定義 (iodefine.h の一部を抜粋)

```
struct st_tv {                                          /* struct TV    */
             union {                                    /* TCRV0        */
                   unsigned char BYTE;                  /*  Byte Access */
                   struct {                             /*  Bit  Access */
                          unsigned char CMIEB:1;        /*    CMIEB     */
                          unsigned char CMIEA:1;        /*    CMIEA     */
                          unsigned char OVIE :1;        /*    OVIE      */
                          unsigned char CCLR :2;        /*    CCLR      */
                          unsigned char CKS  :3;        /*    CKS       */
                          }      BIT;                   /*              */
                   }             TCRV0;                 /*              */
             union {                                    /* TCSRV        */
                   unsigned char BYTE;                  /*  Byte Access */
                   struct {                             /*  Bit  Access */
                          unsigned char CMFB:1;         /*    CMFB      */
                          unsigned char CMFA:1;         /*    CMFA      */
                          unsigned char OVF :1;         /*    OVF       */
                          unsigned char     :1;         /*              */
                          unsigned char OS  :4;         /*    OS        */
                          }      BIT;                   /*              */
                   }             TCSRV;                 /*              */
             unsigned char       TCORA;                 /* TCORA        */
             unsigned char       TCORB;                 /* TCORB        */
             unsigned char       TCNTV;                 /* TCNT         */
             union {                                    /* TCRV1        */
                   unsigned char BYTE;                  /*  Byte Access */
                   struct {                             /*  Bit  Access */
                          unsigned char     :3;         /*              */
                          unsigned char TVEG:2;         /*    TVEG      */
                          unsigned char TRGE:1;         /*    TRGE      */
                          unsigned char     :1;         /*              */
                          unsigned char ICKS:1;         /*    ICKS      */
                          }      BIT;                   /*              */
                   }             TCRV1;                 /*              */
};                                                      /*              */

......

#define TV      (*(volatile struct st_tv    *)0xFFA0)   /* TV    Address*/
```

10.1.6 基本的なプログラム (トグル出力)

圧電スピーカーを使って、まずは基準音のラ (440Hz) を出してみましょう。

音階は周波数で決まり、音色は波形で決まります。ここでは、波形は単純に ON、OFF を繰り返す矩形波を用いることにします (図 10.2)。

図 10.2　矩形波

ON、OFF で 1 周期なので、ON のみ OFF のみの周期は全体の周期の半分 (周波数は倍) になることに注意してください。

20MHz を 128 分周したときに、周波数を 880Hz にするには、タイムコンスタントレジスタ A(TCORA) の値をいくつにすればよいか、その計算方法を以下にまとめておきましょう。

タイムコンスタントレジスタ A(TCORA) の値を計算 (20MHz を 128 分周したときに、周波数を 880Hz にする)

20MHz を 128 分周すると、

$$\frac{20000000}{128}(Hz)$$

になります。
TCORA に設定する値を仮に x とすると、上記の値をさらに x で割った結果が 880 になれば良いわけです。
したがって、

$$\frac{20000000}{128 \times x} = 880$$

となる x を求めます。
結果は、

$$x = 177.56$$

となります。

では、コンペアマッチ A を用い、トグル出力を設定することで、440Hz のパルスを出力してみましょう。

06_TMRV01.c

```
 1  /************************************************************************/
 2  /*                                                                      */
 3  /*  FILE        :06_TMRV01.c                                            */
 4  /*  DESCRIPTION :タイマ V で「ラ」の音を鳴らす                          */
 5  /*  CPU TYPE    :H8/3694F                                               */
 6  /*                                                                      */
 7  /*  SP P76/TMOV                                                         */
 8  /*                                                                      */
 9  /************************************************************************/
10
11  #include "iodefine.h"
12
13  /*****************************************
14    タイマ V の初期化 (128 分周)TCORA:178
15  *****************************************/
16  void TimerVInit(void)
17  {
18    TV.TCRV0.BIT.CKS=3;
19    TV.TCRV1.BIT.ICKS=1; /* 128 分周 */
20    TV.TCRV0.BIT.CCLR=1; /* コンペアマッチ A でクリア */
21    TV.TCSRV.BIT.OS=3; /* コンペアマッチ A でトグル出力 */
22    TV.TCORA=177; /* 周波数 880Hz */
23  }
24
25  /*****************************************
26    main 関数
27  *****************************************/
28  void main(void)
29  {
30    TimerVInit();  /* TimerV 初期化 */
31    while(1){
32      ;
33    }
34  }
```

End Of List

実行結果

「ラ」の音 (周波数が約 440Hz の矩形波) が出力される。
図 10.3 は P76 から出力されるパルスを、オシロスコープで測定した結果です。周波数が 441Hz とほぼ計算通りの値になっていることに注目してください。

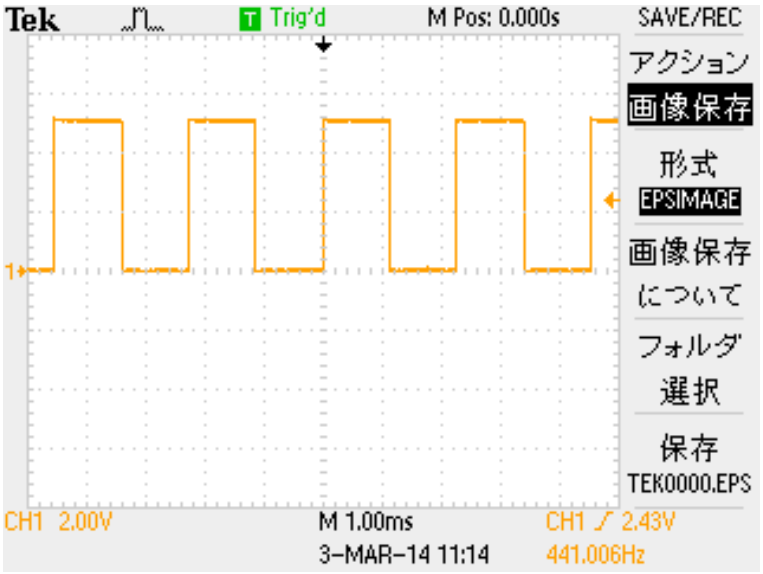


図 10.3　実行結果 (オシロスコープで計測)

プログラム解説 (06_TMRV01.c)

```
18    TV.TCRV0.BIT.CKS=3;
19    TV.TCRV1.BIT.ICKS=1; /* 128 分周 */
```

P.231 の表 10.4 と P.234 のタイマ V のレジスタ定義からこの 2 行の設定で、内部クロックの ϕ/128 立ち下がりエッジでカウントする設定になることが分かります。

```
20    TV.TCRV0.BIT.CCLR=1; /* コンペアマッチ A でクリア */
```

P.231 の表 10.3 から、コンペアマッチ A で TCNTV がクリアされる設定であることが分かります。

今回はコンペアマッチ A のみ利用します。

```
21    TV.TCSRV.BIT.OS=3; /* コンペアマッチ A でトグル出力 */
```

P.232 の表 10.5 と P.234 のタイマ V のレジスタ定義から、TCORA と TCNTV のコンペアマッチによって、TMOV 端子からトグル出力される設定であることが分かります。

```
22    TV.TCORA=177; /* 周波数 880Hz */
```

すでに説明したように、周波数 880Hz の矩形波を出力するための設定です。

ON、OFF のそれぞれが 880Hz になる周期を設定しているので、全体としては 440Hz になることに注意してください。

```
30    TimerVInit();  /* TimerV 初期化 */
```

16 行目から 23 行目までの TimerV 初期化関数、TimerVInit を呼んでいます。

この設定だけ行えば、後は何もしなくても、端子 P76 から 440Hz の矩形波が出力されます。

✍ 課題 10.1.2 (提出) 他の音も出してみる

上のサンプルでは 440Hz の矩形波を出力しましたが、他の音も出してみましょう。

プロジェクト名：e06_TMRV01

10.1.7 音符と休符

ここでは、楽譜をデータ化するために最低限必要と思われる情報を整理しておきましょう。

音楽を演奏するには、音の高さ (周波数) と音の長さを指定してやらなければなりません。例えば「ド」の四分音符というようにです。

音色は、同じ時間の ON と OFF を繰り返す矩形波を用いるとこにします (P.236 の図 10.3 など参照)。

まずは、音符と休符の長さを説明しましょう。

以下に、音符の記号と長さを表にしました。長さは全音符を 16 として、他の音符や休符の相対的な長さを示しています。

なお、一番右の列の「記号」は、今回のプログラムで、対応する長さを指定するために利用する記号です。これらは、後ほど music.h というヘッダファイルの中で定義します。

表 10.8　音符と休符

音符		休符		長さ	記号 (独自定義)
	全音符		全休符	16	L1
	付点 2 分音符		付点 2 分休符	12	F2
	2 分音符		2 分休符	8	L2
	付点 4 分音符		付点 4 分休符	6	F4
	4 分音符		4 分休符	4	L4
	8 分音符		8 分休符	2	L8
	16 分音符		16 分休符	1	L16

以下に、音の高さについてまとめました。右端の列の「記号」が今回プログラムで使用する記号です。

データには構造体の配列を用いて、メンバのとして音の高さと音の長さを指定するようにします。

表 10.9: 音階

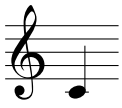
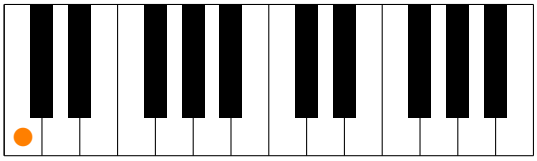
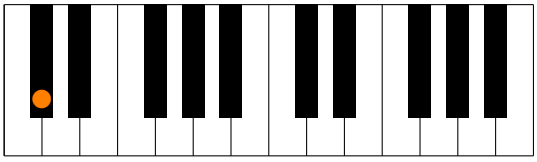
音階	楽譜	鍵盤	記号 (独自定義)
ド			do0
ド#			dos0

表 10.9: 音階

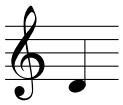
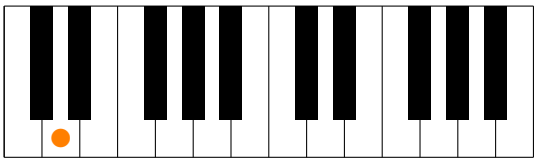
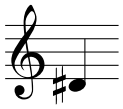
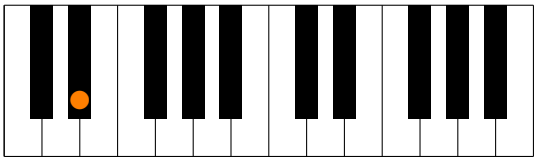
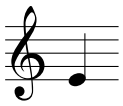
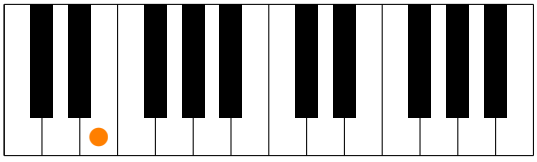
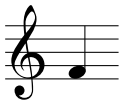
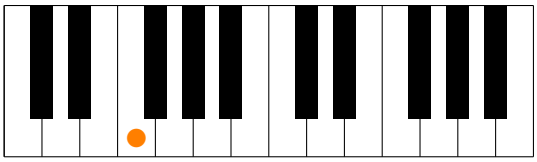
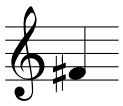
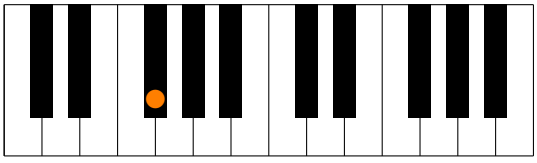
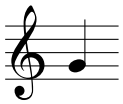
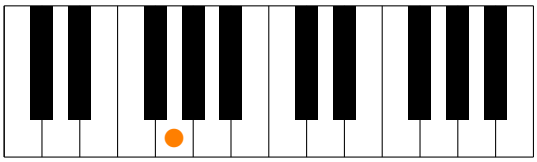
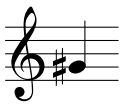
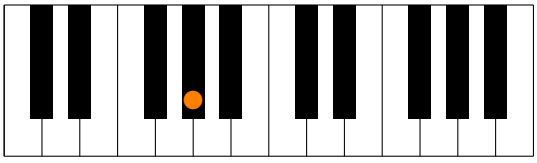
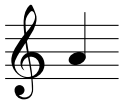
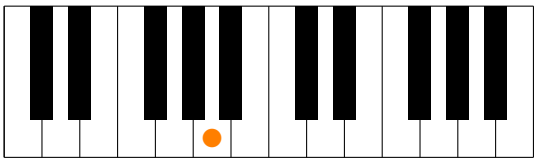
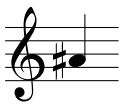
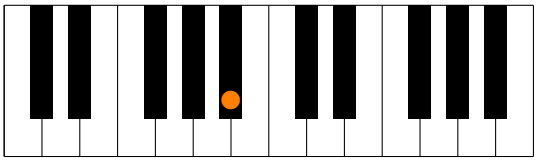
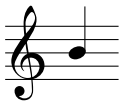
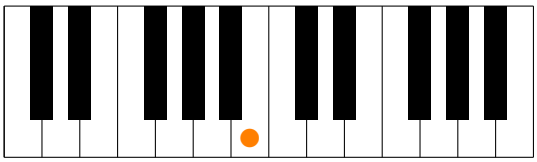
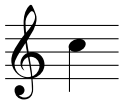
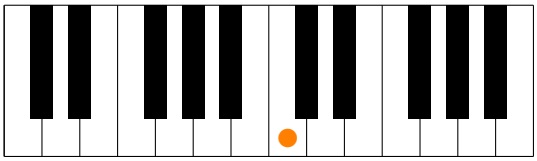
音階	楽譜	鍵盤	記号 (独自定義)
レ			re0
レ#			res0
ミ			mi0
ファ			fa0
ファ#			fas0
ソ			so0
ソ#			sos0
ラ			ra0
ラ#			ras0

表 10.9: 音階

音階	楽譜	鍵盤	記号 (独自定義)
シ			si0
ド			do1
休符は音階を記号「0」で指定する。休符の長さは上述の通り。			

既存の音楽データの形式は複雑ですので、ここでは独自の形式で音楽データを作ることにします。

上述のように、音の高さと音の長さを組み合わせて指定することにしましょう。

たとえば、四分音符の「レ」は、

```
{re0,L4}
```

と書きます。

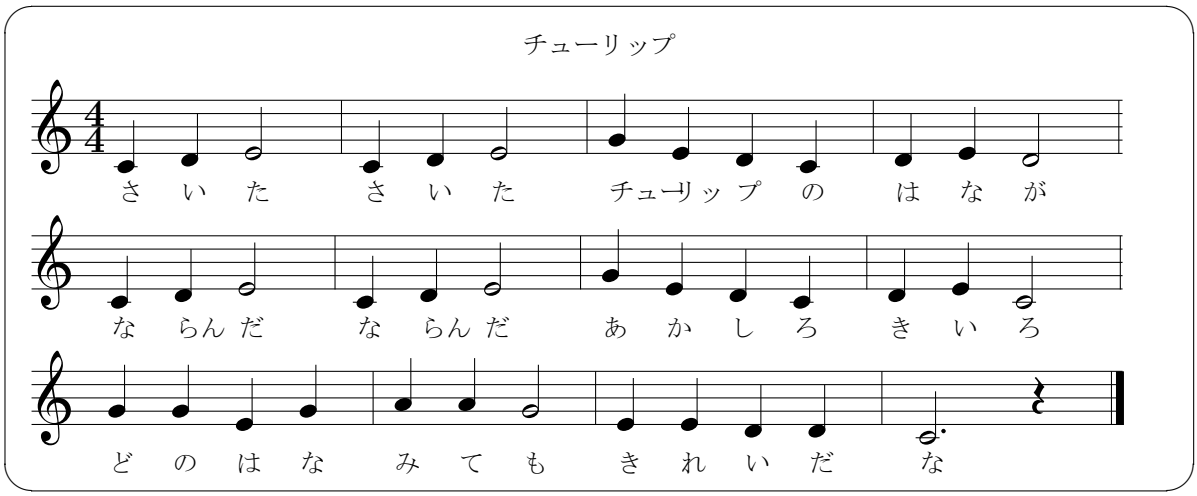
2 分休符は、

```
{0,L2}
```

です。

P.238 の表 10.8 と、P.240 の表 10.9 を参考にしてください。

では、童謡「チューリップ」の楽譜をもとに、プログラム用のデータに書きなおしてみます。



この楽譜から音楽データを書き下すと、以下のようになります。

music_data

```
 1  //チューリップ
 2
 3  #define NOTEMAX 39
 4
 5  const struct NoteData Note[NOTEMAX]={
 6    {do0,L4},{re0,L4},{mi0,L2},{do0,L4},{re0,L4},{mi0,L2},
 7    {so0,L4},{mi0,L4},{re0,L4},{do0,L4},{re0,L4},{mi0,L4},{re0,L2},
 8    {do0,L4},{re0,L4},{mi0,L2},{do0,L4},{re0,L4},{mi0,L2},
 9    {so0,L4},{mi0,L4},{re0,L4},{do0,L4},{re0,L4},{mi0,L4},{do0,L2},
10    {so0,L4},{so0,L4},{mi0,L4},{so0,L4},{ra0,L4},{ra0,L4},{so0,L2},
11    {mi0,L4},{mi0,L4},{re0,L4},{re0,L4},{do0,F2},{  0,L4}
12  };
```

End Of List

データ解説 (music_data)

```
3  #define NOTEMAX 39
```

音符の数を定義しています。

今回の「チューリップ」は休符も入れて、音符が 39 個あるので 39 にしています。

NOTEMAX をいう名前は後で参照しますので、このまま定義しておください。

```
 5  const struct NoteData Note[NOTEMAX]={
   ……
12  };
```

楽譜データのための構造体 struct NoteData 型の配列 Note を宣言しています。

配列の要素数は、3 行目で定義した NOTEMAX が使われます。

今の場合 39 です。

```
6  {do0,L4},{re0,L4},{mi0,L2},{do0,L4},{re0,L4},{mi0,L2},
```

最初の 2 小節をデータ化したものです。楽譜と P.238 の表 10.8 や表 10.9 を確認してみてください。

楽譜さえあれば、他の音楽を演奏することはさほど難しくありません。

10.1.8 関数化 (音楽の演奏)

すでに説明したようにタイマは機能が多いので、音楽を演奏するとかサーボモータを動かすとか目的別に関数を作ることにします。

ここでは、音楽を演奏するプログラムを考えてみましょう。

すでに説明したように、音楽を演奏するには、音の高さ (周波数) と音の長さを指定してやらなければなりません。例えば「ド」の四分音符というようにです。

ここでは、タイマ V を音の高さを決めるのに、タイマ A を音の長さを決めるのに利用しましょう。

移植性を考慮して、タイマの設定を決めておきましょう。

音の長さを測るタイマは、タイマ割込みの周期に TEMPO(後述) をかけた時間が約 50ms になるように設定することにします。

ここではタイマ A を用いて 128 分周の設定をしましょう。このときオーバーフローは 1.64mS で起きるので、TEMPO の値を 30 にすれば、タイマ割込みの周期に TEMPO をかけた時間が約 50ms になります。

音階に関しては、タイマ V で 128 分周します。H8_3694F ではこれ以上分周できませんので、ここで行う設定より低い音を出すことはできないことに注意してください。

以下は今回作る音楽用の関数の仕様です。

音楽関連関数 (H8/3694F ボード)

ヘッダファイル : music.h

```
SP P76/TMOV

*****************************************************************
void MusicInit(void);

    形式  : #include"music.h"
            void MusicInit(void);
    引数 :  なし
    戻り値 : なし
    解説 :  音楽演奏用のポート初期化し、演奏を始める関数。
            音楽を演奏する際に実行すること。

*****************************************************************
void StopMusic(void);

    形式  : #include"music.h"
            void StopMusic(void);
    引数 :  なし
    戻り値 : なし
    解説 :  音楽の演奏を停止する関数。

*****************************************************************
void StartMusic(void);

    形式  : #include"music.h"
            void StartMusic(void);
    引数 :  なし
    戻り値 : なし
    解説 :  音楽の演奏を開始する関数。

*****************************************************************
```

タイマ A 割込みを利用しますので、P.220 のと同様に、intprg.c の INT_TimerA をコメントアウトしてください。

intprg.c の INT_TimerA をコメントアウト

```
// vector 19 Timer A Overflow
//__interrupt(vect=19) void INT_TimerA(void) {/* sleep(); */}
```

music.h

```
 1  #ifndef _MUSIC_H_
 2  #define _MUSIC_H_
 3
 4  /*---------------------------------------------------------------------*/
 5  /*  ファイル名：  music.h                                              */
 6  /*        内容：  音楽用タイマ関数ヘッダファイル                       */
 7  /*                                                                     */
 8  /*    SP P76/TMOV                                                      */
 9  /*                                                                     */
10  /*---------------------------------------------------------------------*/
11
12  //音の長さの定義 (すべての値が偶数になるように定義)
13  //奇数は拡張性のためにとっておく
14  //TEMPO:0.05s, L1:1.6s
15  #define TEMPO 30       /* L1 の値に反映させる */
16  #define L1  32*TEMPO   //全音符長
17  #define L2  (L1/2)     //2 分音符長
18  #define L4  (L1/4)     //4 分音符長
19  #define L8  (L1/8)     //8 分音符長
20  #define L16 (L1/16)    //16 分音符長
21  #define L32 (L1/32)    //32 分音符長
22  #define F2  ((L1*3)/4) //付点 2 分音符長
23  #define F4  ((L1*3)/8) //付点 4 分音符長
24  #define L3  (L4/3)     //3 連符長
25  #define L6  (L4/6)     //6 連符長
26
27  //音の高さ (音程) の定義
28  #define do0  149 //ド
29  #define dos0 141 //ド#
30  #define re0  133 //レ
31  #define res0 126 //レ#
32  #define mi0  119 //ミ
33  #define fa0  112 //ファ
34  #define fas0 106 //ファ#
35  #define so0  100 //ソ
36  #define sos0  94 //ソ#
37  #define ra0   89 //ラ    880Hz
38  #define ras0  84 //ラ#
39  #define si0   79 //シ
40  #define do1   75 //ド
41  #define dos1  70 //ド#
42  #define re1   67 //レ
43  #define res1  63 //レ#
44  #define mi1   59 //ミ
45  #define fa1   56 //ファ
46  #define fas1  53 //ファ#
47  #define so1   50 //ソ
48  #define sos1  47 //ソ#
49  #define ra1   44 //ラ   1760Hz
50  #define ras1  42 //ラ#
51  #define si1   40 //シ
52  #define do2   37 //ド
53
54  //音のデータ
55  struct NoteData{
56    unsigned int Freq;  //音程
57    unsigned int  Len;  //長さ
58  };
59
60  void MusicInit(void);
61  void StopMusic(void);
62  void StartMusic(void);
63  #endif
```

End Of List

music.c

```
1  /*********************************************************************/
2  /*                                                                   */
```

```
 3  /*  FILE        :music.c                                                  */
 4  /*  DESCRIPTION :音楽用関数                                              */
 5  /*  CPU TYPE    :H8/3694F                                                 */
 6  /*                                                                        */
 7  /*    SP P76/TMOV                                                         */
 8  /*                                                                        */
 9  /************************************************************************/
10
11  /* インクルードファイル */
12  #include<machine.h>
13  #include"iodefine.h"
14  #include "music.h"    /* 音階や音の長さの定義 */
15  #include "music_data"       /* 楽譜データ */
16
17  struct NoteData CNote; /* Freq：音程, Len：音の長さ */
18  int NoteCnt=0;   /* 音符の何番目を演奏しているか */
19  int ZeroCnt=30;   /* 音の最後の消音 50ms 1.6mS 単位 */
20  unsigned int fdata; /* 音階を格納する変数 */
21  int note_max=NOTEMAX; /* 楽譜データの要素数 */
22
23  /*********************************************
24      音の開始・停止操作 (タイマ V)
25  *********************************************/
26  void StopPwmTimer(void)
27  {
28    TV.TCSRV.BIT.OS=0;
29  }
30  void StartPwmTimer(void)
31  {
32    TV.TCSRV.BIT.OS=3; /* コンペアマッチ A でトグル出力 */
33  }
34  /*********************************************
35      音の長さをカウントするタイマの初期化・リセット操作 (タイマ A)
36  *********************************************/
37  void StopLengthTimer(void)
38  {
39    TA.TMA.BIT.CKSI=0xC;
40  }
41  void StartLengthTimer(void)
42  {
43    TA.TMA.BIT.CKSI=5; /* インターバルタイマ 128 分周 */
44  }
45
46  /*********************************************
47      音用タイマの初期化 20MHz 128 分周 (タイマ V)
48      P76/TMOV に出力
49  *********************************************/
50  void PwmTimerInit(void)
51  {
52    TV.TCRV0.BIT.CKS=3;
53    TV.TCRV1.BIT.ICKS=1; /* 128 分周 */
54    TV.TCRV0.BIT.CCLR=1; /* コンペアマッチ A でクリア */
55    TV.TCSRV.BIT.OS=3; /* コンペアマッチ A でトグル出力 */
56  }
57
58  /*********************************************
59      音の長さ用タイマの初期化 1.6ms ごとに割り込みをかける設定
60      20MHz 128 分周 (タイマ A)
61  *********************************************/
62  void LengthTimerInit(void)
63  {
64    set_imask_ccr(1);
65    IENR1.BIT.IENTA=1; /* タイマ A のオーバフロー割り込み要求 */
66    IRR1.BIT.IRRTA=0;
67    set_imask_ccr(0);
68  }
69
70  /*********************************************
71      音をセット
72  *********************************************/
73  void  SetData(unsigned int freq)
74  {
75    TV.TCORA=freq;   /* ON, OFF は半々 エンベロープを入れるならここを変更 */
76  }
77
78  /*********************************************
79      音の長さのタイマ割込みフラグをクリア
80  *********************************************/
81  void  ClearLegthFlag(void)
82  {
83    IRR1.BIT.IRRTA=0;
84  }
85
86  /*-------------------------------------------
87     以下、ハードウェア依存性の無いプログラム
88     ただし、二つのタイマの割り込み関数は修正の必要あり
89     (今の場合音はトグル出力で出しているので割込み関数は無い)
90  -------------------------------------------*/
91
92  /*********************************************
93      最初のデータをセット　休符のときの処理
```

```
 94  *****************************************/
 95  void StopMusic(void){
 96    StopPwmTimer();
 97    StopLengthTimer();
 98  }
 99
100  void StartMusic(void){
101    StartPwmTimer();
102    StartLengthTimer();
103  }
104
105  void  SetInitData(void)
106  {
107    CNote=Note[NoteCnt];
108    if(CNote.Freq == 0){
109      StopPwmTimer();
110    }else{
111      SetData(CNote.Freq);
112      StartPwmTimer();
113    }
114  }
115
116  /*******************************************
117      音楽演奏のための初期化関数
118  *******************************************/
119  void MusicInit(void)
120  {
121    PwmTimerInit();
122    LengthTimerInit();
123    SetInitData();
124    StartLengthTimer();
125  }
126
127    /*******************************************
128      IMIA1 割込み関数 (音の長さ)1.6ms ごとの割込み
129    *******************************************/
130  __interrupt(vect=19) void INT_TimerA(void)
131  {
132    ClearLegthFlag();
133    if(CNote.Len>0){
134      CNote.Len--; /* エンベロープを入れるならここで対応 */
135    }else if(NoteCnt<(note_max-1) && ZeroCnt>0){ /* 消音を入れて同じ音が続いたときの対処 */
136      if(ZeroCnt==50){
137        StopPwmTimer();
138      }
139      ZeroCnt--;
140    }else if(NoteCnt<(note_max-1) && ZeroCnt<=0){  /* CNote.Len==0 音の終了時 */
141      ZeroCnt=50;
142      NoteCnt++;
143      CNote=Note[NoteCnt];
144      fdata=CNote.Freq;   /* Freq : 音階 タイマ A0 */
145      if(fdata==0){       /* 音階データが 0 の場合には音を出さない */
146        StopPwmTimer(); /* タイマ A0 を停止 */
147      }else{
148        SetData(fdata);
149        StartPwmTimer();
150      }
151    }else{ /* 楽譜の最後で音を止める */
152      StopMusic();
153    }
154  }
```

End Of List

06_TMRV02.c

```
 1  /*********************************************************************/
 2  /*                                                                   */
 3  /*  FILE        :06_TMRV02.c                                         */
 4  /*  DESCRIPTION :音楽を鳴らす                                        */
 5  /*  CPU TYPE    :H8/3694F                                            */
 6  /*                                                                   */
 7  /*  SP P76/TMOV                                                      */
 8  /*                                                                   */
 9  /*********************************************************************/
10
11  #include "music.h"    /* 音階や音の長さの定義 */
12
13  void main(void)
14  {
15
16    /*******************************************
17      音楽用タイマの初期化
18    *******************************************/
19    MusicInit();
20
21    while(1){
```

```
22        ;
23    }
24  }
25
```

End Of List

music_data

```
 1  //チューリップ
 2
 3  #define NOTEMAX 39
 4
 5  const struct NoteData Note[NOTEMAX]={
 6    {do0,L4},{re0,L4},{mi0,L2},{do0,L4},{re0,L4},{mi0,L2},
 7    {so0,L4},{mi0,L4},{re0,L4},{do0,L4},{re0,L4},{mi0,L4},{re0,L2},
 8    {do0,L4},{re0,L4},{mi0,L2},{do0,L4},{re0,L4},{mi0,L2},
 9    {so0,L4},{mi0,L4},{re0,L4},{do0,L4},{re0,L4},{mi0,L4},{do0,L2},
10    {so0,L4},{so0,L4},{mi0,L4},{so0,L4},{ra0,L4},{ra0,L4},{so0,L2},
11    {mi0,L4},{mi0,L4},{re0,L4},{re0,L4},{do0,F2},{  0,L4}
12  };
```

End Of List

実行結果

「チューリップ」を演奏する。
演奏が終わったら、何もしない。

プログラム解説 (music.h)

```
15  #define TEMPO 30        /* L1 の値に反映させる */
16  #define L1  32*TEMPO    //全音符長
17  #define L2  (L1/2)      //2 分音符長
18  #define L4  (L1/4)      //4 分音符長
19  #define L8  (L1/8)      //8 分音符長
20  #define L16 (L1/16)     //16 分音符長
21  #define L32 (L1/32)     //32 分音符長
22  #define F2  ((L1*3)/4) //付点 2 分音符長
23  #define F4  ((L1*3)/8) //付点 4 分音符長
24  #define L3  (L4/3)      //3 連符長
25  #define L6  (L4/6)      //6 連符長
```

音の長さを設定しています。

音の長さと記号の関係は、P.238 の表 10.8 に示した通りです。

数値はタイマ A の割込み回数を表しています。今、タイマ A のオーバーフロー割込みは、約 1.64mS ですので、

$$L1 = 32 \times TEMPO = 32 \times 30 = 96$$

は、

$$96 \times 1.64 = 1574$$

と、約 1.6 秒 (1574m 秒) になります。つまり、全音符長を約 1.6 秒に設定しているわけです。

音楽のテンポを遅くしたい場合には、TEMPO を値を変更してください。

なお、32 という数字はすべての音符長が整数になるように設定した値です。

```
28  #define do0  149 //ド
  ......
52  #define do2   37 //ド
```

音の高さの定義です。

28 行目よりもう一オクターブ低い「ド」は、298 と 1 バイトのレジスタの範囲に収まりません。そこで今回は定義しませんでした。

31 行目よりもう一オクターブ低い「レ#」は、252 と 1 バイトのレジスタの範囲に収まります。つまり、31 行目よりもう一オクターブ低いレ#から上の音は定義可能ですので、必要に応じて書き足してください。

```
54  //音のデータ
55  struct NoteData{
56    unsigned int Freq;  //音程
57    unsigned int  Len;  //長さ
58  };
```

音楽データ格納のための構造体 struct NoteData の定義です。

メンバは音程のデータ Freq と、音の長さのデータ Len です。

```
60  void MusicInit(void);
61  void StopMusic(void);
62  void StartMusic(void);
```

音楽用関数のプロトタイプ宣言です。

10.1.8 で定義したように、MusicInit は初期化関数、StopMusic は演奏停止関数、StartMusic は演奏開始関数です。

プログラム解説 (music.c)

```
26  void StopPwmTimer(void)
27  {
28    TV.TCSRV.BIT.OS=0;
29  }
30  void StartPwmTimer(void)
31  {
32    TV.TCSRV.BIT.OS=3; /* コンペアマッチ A でトグル出力 */
33  }
```

TV.TCSRV.BIT.OS はタイマコントロール／ステータスレジスタ V(TCSRV)(P.232 表 10.5) の 0-3 ビットまでを表しています (P.233 のレジスタ定義参照)。

0-1 ビットが TCORA と TCNTV のコンペアマッチによる TMOV 端子出力の設定です。

0 に設定すると出力されず、3 に設定するとトグル出力 (ON、OFF の繰り返し) になります。

ここでは、これを用いて PWM の停止と開始関数を作っています。

PWM が停止すると音が鳴らなくなります。

```
37  void StopLengthTimer(void)
38  {
39    TA.TMA.BIT.CKSI=0xC;
40  }
41  void StartLengthTimer(void)
42  {
43    TA.TMA.BIT.CKSI=5; /* インターバルタイマ 128 分周 */
44  }
```

TA.TMA.BIT.CKSI は、タイマモードレジスタ A(TMA) (P.215 表 9.2) の 0-3 ビットまでを表しています (P.216 のレジスタ定義参照)。

P.215 の表 9.2 によると、TMA3=1、TMA2=1、TMA1=0、TMA0=0 と設定することによってタイマモードレジスタ A(TMA) がリセットされることが分かります。この 4 ビットは TA.TMA.BIT.CKSI で定義されるので、0xC をセットしました。

タイマモードレジスタ A(TMA) の表 9.2 から分かるように、インターバルタイマを 128 分周するには、TMA3~TMA0 までの 4 ビットを 5 に設定すればよいわけです。

```
50  void PwmTimerInit(void)
51  {
52    TV.TCRV0.BIT.CKS=3;
53    TV.TCRV1.BIT.ICKS=1; /* 128 分周 */
54    TV.TCRV0.BIT.CCLR=1; /* コンペアマッチ A でクリア */
55    TV.TCSRV.BIT.OS=3;   /* コンペアマッチ A でトグル出力 */
56  }
```

音楽用タイマの初期化関数です。音を鳴らすタイマの初期化になります (タイマ V)。

P.231 の表 10.4 から分かるように、内部クロックを 128 分周するには、TV.TCSRV.BIT.OS を 3 に設定する必要があります。また、同時に TV.TCRV1.BIT.ICKS を 1 に設定する必要があります。52 行目と 53 行目でこの設定を行っているわけです。

P.231 の表 10.3 から分かるように、TCNTV をコンペアマッチ A でクリアするには、TV.TCRV0.BIT.CCLR を 1 にする必要があります。54 行目でこの設定を行っています。

P.232 の表 10.5 から分かるように、TCORA と TCNTV のコンペアマッチにより、TMOV(P76) 出力端子にトグル出力 (0 出力と 1 出力を交互に行う) を行うには、TV.TCSRV.BIT.OS を 3 にする必要があります。

この関数を実行すると、音楽の演奏が始まることに注意してください。

```
62  void LengthTimerInit(void)
63  {
64    set_imask_ccr(1);
65    IENR1.BIT.IENTA=1; /* タイマ A のオーバフロー割り込み要求 */
66    IRR1.BIT.IRRTA=0;
67    set_imask_ccr(0);
68  }
```

音の長さを測っている、タイマ A の割込み設定関数です。

64 行目と 67 行目は、P.222 で説明した set_imask_ccr 関数を利用しています。64 行目で割り込みを禁止し、67 行目で割込みそ許可しています。割込みの設定を行うためには、割り込みを禁止して行います。

65 行目ではタイマ A の割込み要求を許可しています (P.221 の表 9.4)。

66 行目では、タイマ A の割込み要求フラグをクリアしています (P.222 の表 9.5)。

なお、タイマ A のオーバフロー割込みは、20MHz を 128 分周していますので、約 1.6ms になります。

```
73  void  SetData(unsigned int freq)
74  {
75    TV.TCORA=freq;   /* ON, OFF は半々 エンベロープを入れるならここを変更 */
76  }
```

音の設定をする関数です。

音の高さはタイマ V で設定しています。コンペアマッチ V でトグル出力をしていますので、レジスタ TV.TCORA の値で音の高さが決まるわけです。

なお、引数としては music.h で設定した値を渡します。

```
81  void  ClearLegthFlag(void)
82  {
83    IRR1.BIT.IRRTA=0;
84  }
```

タイマ A の割込みフラグをクリアする関数です。

タイマ A 割込みの中で利用しています (132 行目)。

```
95  void StopMusic(void){
96    StopPwmTimer();
97    StopLengthTimer();
98  }
```

ここから先は、ハードウェア依存性のない関数です。ただし、割込み関数はマイコンやコンパイラに依存します。

関数 StopMusic は音楽を停止する関数です。関数 StopPwmTimer と StopLengthTimer から構成されます。

関数 StopPwmTimer は音階を鳴らすタイマ V を停止する関数です。

関数 StopLengthTimer は音の長さをはかる、タイマ A を停止します。

```
100  void StartMusic(void){
101    StartPwmTimer();
102    StartLengthTimer();
103  }
```

関数 StartMusic は音楽を開始する関数です。関数 StartPwmTimer と StartLengthTimer から構成されます。

関数 StartPwmTimer は音階を鳴らすタイマ V を開始する関数です。

関数 StartLengthTimer は音の長さをはかる、タイマ A を開始します。

```
105  void  SetInitData(void)
106  {
107    CNote=Note[NoteCnt];
108    if(CNote.Freq == 0){
109      StopPwmTimer();
110    }else{
111      SetData(CNote.Freq);
112      StartPwmTimer();
113    }
114  }
```

関数 SetInitData は音楽の最初のデータを設定します。

最初が休符だった時に必要な処理です。

107 行目は楽譜から音の高さのデータを取り出しています。関数 SetInitData は、演奏の最初に実行するので、NoteCnt の値はゼロです (ゼロと明示的に書いてもかまいません)。

108 行目から 110 行目は、最初が休符だった時の処理です。CNote.Freq に音のデータが入っていますので、それがゼロだった場合は休符とみなします。109 行目で音を止めています。

110 行目から 113 行目までは、最初の音符が休符以外の場合の処理です。

111 行目で音の高さをセットしています。

112 行目で音のタイマをスタートしています。

```
119  void MusicInit(void)
120  {
121    PwmTimerInit();
122    LengthTimerInit();
123    SetInitData();
124    StartLengthTimer();
125  }
```

上で定義した関数を利用して、音楽演奏のための初期化関数 MusicInit を定義しました。

121 行目では、音の高さを決めているタイマ A の初期化関数 PwmTimerInit を呼び出しています。

122 行目では、音の長さを決めるタイマ V の初期化関数 LengthTimerInit を呼び出しています。

123 行目は、最初の音符のための処理です。タイマ A がスタートします。

124 行目は、音の長さを決めるタイマ V をスタートさせる関数 StartLengthTimer を呼び出しています。

```
132    ClearLegthFlag();
133    if(CNote.Len>0){
134      CNote.Len--; /* エンベロープを入れるならここで対応 */
135    }
```

130 行目から 154 行目までは、タイマ V の割込み関数です。このプログラムでは、1.6ms ごとに割り込みがかかる設定になっています。

このテキストでは、intprg.c に記述されている割込み関数の定義をコメントアウトし、ファイル music.c に記述するようにしましたが、移植性を考えたら、intprg.c から呼び出したほうが良いかもしれません。

たとえば、intprg.c では、

```
// vector 19 Timer A Overflow
__interrupt(vect=19) void INT_TimerA(void) { IntTimerA(); }
```

のように記述し、music.c で割込み関数を IntTimerA という名前で記述するわけです。

特に、割込み関数をプログラムによって変更したい場合には、この記述の仕方は有効であるように思われます。

132 行目では、割込みフラグをクリアしています。次の割込みをかけるために必要な処理です。

133 行目から 135 行目まででは、音の長さをカウントしています。134 行目で CNote.Len を減らしています。指定された音の長さまでカウントを続けるわけです。

```
135    }else if(NoteCnt<(note_max-1) && ZeroCnt>0){ /* 消音を入れて同じ音が続いたときの対処 */
136      if(ZeroCnt==50){
137        StopPwmTimer();
138      }
139      ZeroCnt--;
140    }
```

ここでは、音と音の間に消音を入れて同じ音が続いた場合に、音がつながらないようにしています。

135 行目の条件式は、指定された音の長さをカウントした後、楽譜の最後でなく、ZeroCnt が正の値だったらというものです。ZeroCnt は新しい音符を読みこむときに 50 に設定しています (141 行目)。この設定により、音符間では $1.6ms \times 50 = 80ms$ の消音が入ることになります。

136 行目では、最初にこの条件を満たしたときに、という条件になります。そのとき、137 行目で、音の高さをカウントしているタイマ A を停止しています。

139 行目では、消音の時間をカウントしています。

```
140    }else if(NoteCnt<(note_max-1) && ZeroCnt<=0){  /* CNote.Len==0 音の終了時 */
```

```
141      ZeroCnt=50;
142      NoteCnt++;
143      CNote=Note[NoteCnt];
144      fdata=CNote.Freq;   /* Freq：音階 タイマ A0 */
145      if(fdata==0){       /* 音階データが 0 の場合には音を出さない */
146        StopPwmTimer(); /* タイマ A0 を停止 */
147      }else{
148        SetData(fdata);
149        StartPwmTimer();
150      }
151    }
```

音符を演奏し終わったときの処理を記述しています。

140 行目の条件式は、楽譜の最後まで到達しておらず、音符間の消音の実行が終わったというものです。次の音符を演奏することになります。

141 行目では、まず音符間の消音の設定をしています。

142 行目では、音符のカウントをしている変数 NoteCnt を 1 増やしています。

143 行目では、変数 NoteCnt が指定する音符のデータを CNote に読みこんでいます。

144 行目では、音の高さのデータを変数 fdata に代入しています。

145 行目から 147 行目まででは、変数 fdata の値がゼロの場合には、休符とみなしてタイマ A を停止しています。

147 行目から 150 行目では、休符以外の音符の場合で、148 行目で音の高さのデータをセットし、149 行目でタイマ A をスタートさせています。

```
151    }else{ /* 楽譜の最後で音を止める */
152      StopMusic();
153    }
```

すべての演奏が終わったときの処理です。

音楽 (タイマ A とタイマ V) を停止しています。

プログラム解説 (06_TMRV02.c)

```
11  #include "music.h"    /* 音階や音の長さの定義 */
```

音楽関数用のヘッダファイル music.h を読みこんでいます。これによって、音楽用関数が利用できることになります。

```
19    MusicInit();
20
21    while(1){
22       ;
23    }
```

19 行目で音楽演奏のための初期化関数 MusicInit を実行しています。これだけで、音楽の演奏が始まります。

21 行目から 23 行目は、無限ループです。今の場合、OS がないのでプログラムは無限ループで終わらせます。

✍ 課題 10.1.3 (提出) いろいろな音楽を演奏してみよう

上のサンプルではチューリップを演奏しましたが、音楽データを変更して、他の音楽を演奏してみましょう。

プロジェクト名：e06_TMRV02