

スタータ・キット「78K0S/KA1+-Do it! (EZ-0002)」を利用した温度計

愛知県
高橋 泰雄 様

はじめに

78K0S/Ka1+-Do it!を使ったPCと接続できるデジタル温度計

今回はスタータ・キット 78K0S/KA1+-Do it!を使ったデジタル温度計を紹介します(写真1)。鍵となる温度センサには小型のデジタル出力温度センサIC(LM74)を使用し、スタータ・キット基板には4桁の7セグメントLEDを付けて温度を2+1形式(整数部2桁(+100℃以上の時は3桁)+小数点以下1桁)で表示しました。更にスタータ・キットのUSBポートを利用して、温度データを1秒ごとにPCへ送信するようにしました。

図1がデジタル温度計の回路図です。右端がスタータ・キットのボードで、上の点線で囲まれた部分がLM74を使った温度センサ部分、右側が7セグメントLEDです。

写真2はVisualBasicで作成したデータ表示アプリケーションの画面です。PC側でこのデータをロギングすることで温度データロガーとしても利用できます。図2はこのようにして作成したデータをExcelに読み込ませてグラフ化したものです。

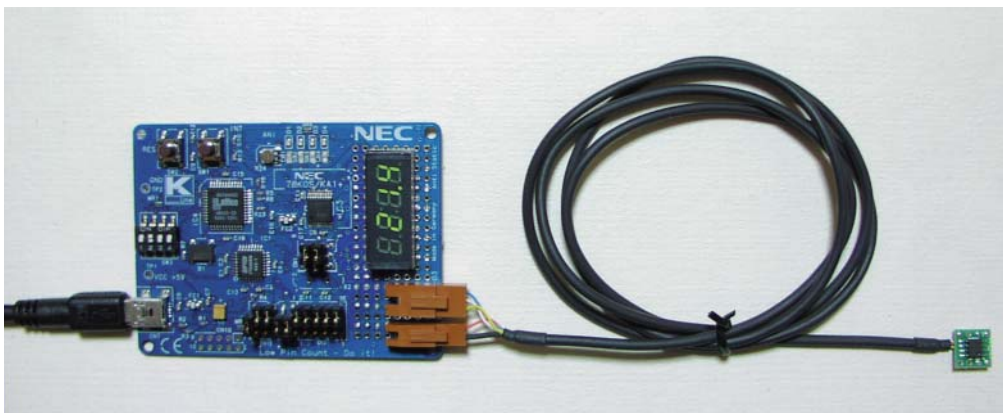


写真1



写真2

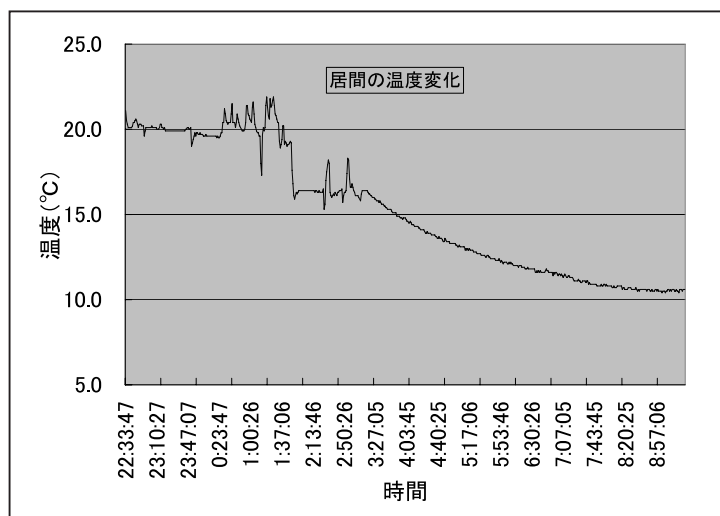


図2

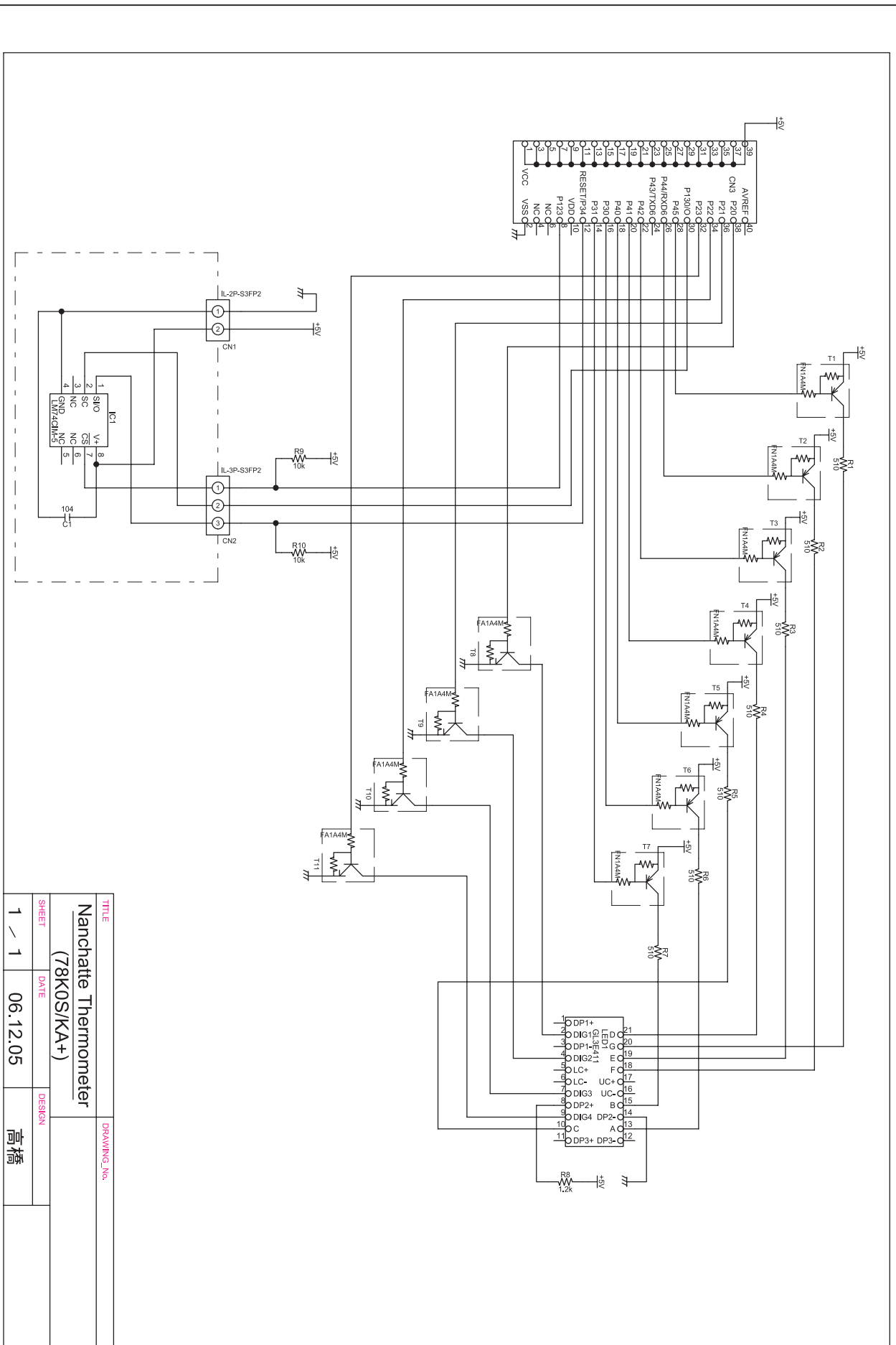


図 1 デジタル温度計回路図

TITLE		DRAWING No.	
Nanchatte Thermometer (78K0S/KA1+)			
SHEET	DATE	DESIGN	
1 / 1	06.12.05	高橋	

温度計測

● 温度センサ素子

温度を測定するという事は、温度によって何らかの物理的な性質が変わるものを利用して、その変化を捉えるということになります。液体や固体の体積が変化するという性質を利用したものには、温度計が一例として挙げられるでしょう。

電子回路で扱う場合は、センサ部分で直接電圧を発生したり、抵抗値や静電容量が変化したりという具合に、検出部分そのものの電気的な性質が変わることで温度を測定する方法が考えられます。また、温度センサは検出部分全体の温度が均一になっていないと誤差が生じますし、測定対象物の熱容量が小さいと温度センサを付けただけで温度が大きく変化してしまうので、検出部分はなるべく小型な方が望ましいということになります。

このような電気的な温度測定素子として昔からよく利用されているのは、異種金属の接合部分に温度差があると電流が流れる現象(熱電効果のひとつ、ゼーベック効果)を使った熱電対と温度によって抵抗値が変化する素材を利用したサーミスタです。

熱電対の構造自体は異なる金属の端同士を溶接した単純な構造です。計測範囲によって材料となる金属の組み合わせは変わりますが、非常に高精度でかつ測定温度範囲も広いという特徴があります。ただし、熱電対の出力電圧は非常に小さく、また出力インピーダンスも高い(少し電流を流すとすぐ電圧が落ちてしまう)ため、ノイズを受けやすい回路には工夫が必要です。

一方、サーミスタ(Thermistor)は Thermally Sensitive Resistor、すなわち熱に敏感な抵抗器を略したもので、温度によって抵抗値が大きく変化する物を指します。

サーミスタには

- 温度が上昇すると抵抗値が小さくなる NTC
- 温度上昇と共に抵抗値も大きくなる PTC
- 特定の温度で急激に抵抗値が変化する CTR

の三種類がありますが、電子回路用として一般的に利用されるのは NTC タイプ、すなわち温度上昇に伴って抵抗値が下がっていく性質を持ったデバイスです。

温度と抵抗値の関係は一般に次のような関係になっています。

$$R1=R2 \cdot \exp\{B \cdot (1/T1 - 1/T2)\}$$

ここで、R1、R2はそれぞれ絶対温度 T1、T2における抵抗値です。カタログでは通常 25℃(絶対温度 298K)における抵抗値と共に B の値が「B 定数」という名称で掲載されています。

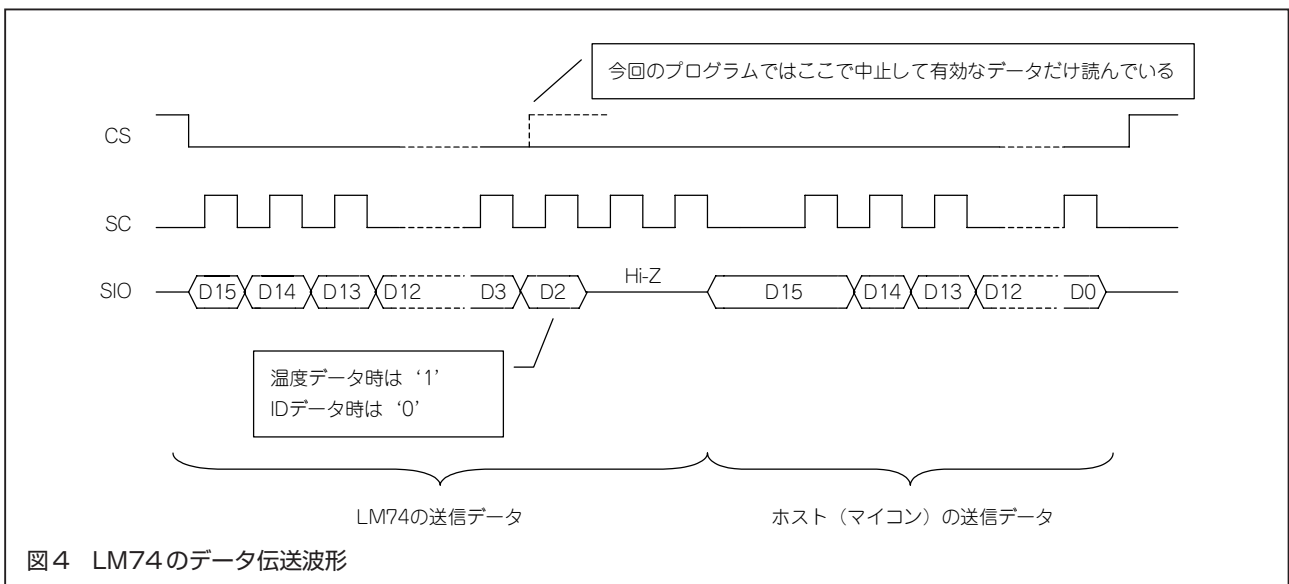
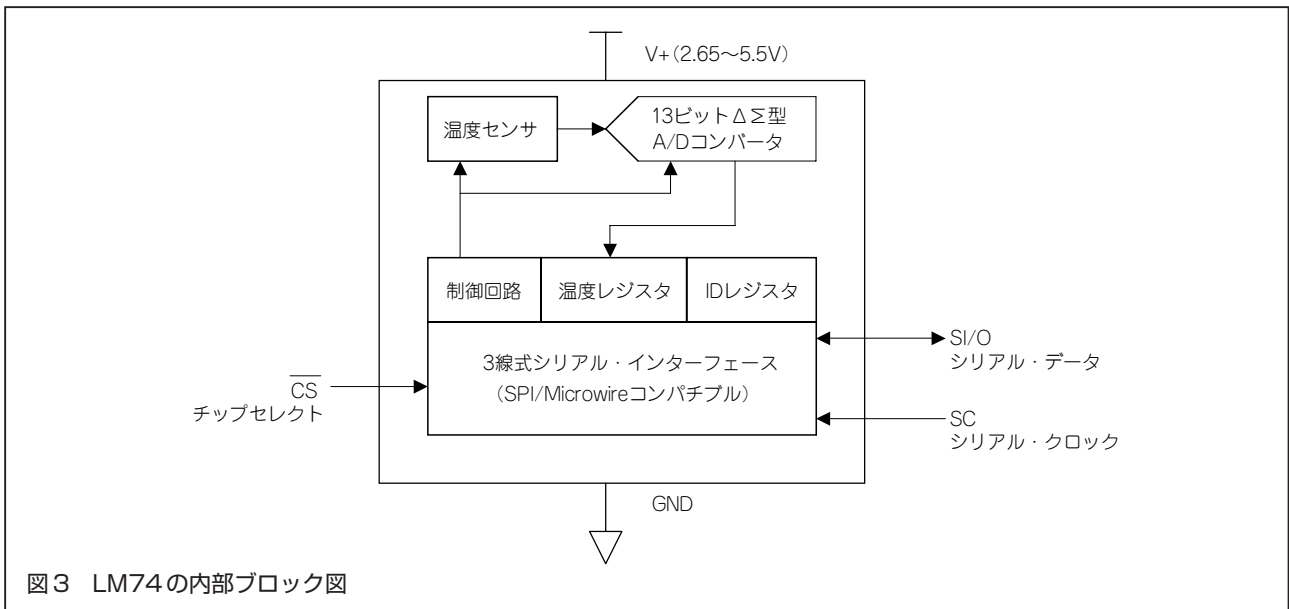
サーミスタが使用できる温度範囲は一般的なもので -50℃から 100℃程度ですが、高耐熱なものでは 300℃くらいまで対応しているものもあります。なお、サーミスタによる温度測定の際には抵抗値を測定する必要がありますが、このためにはサーミスタに多少なりとも電流を流さなくてはなりません。この電流が大きいとサーミスタ自身の発熱やそれにとまなう抵抗値変化が無視できなくなってしまいますので、測定回路の設計には注意が必要です。

● デジタル出力温度センサ IC

サーミスタと同様に半導体の PN 接合部分での順方向降下電圧(ダイオードの順方向降下電圧)も温度によって変化します。この電圧は常温(25℃)ではおおよそ 0.6V 程度で、温度が 1℃下がる毎に約 2mV あがるという、NTC 型のサーミスタに似た特性を示し、絶対零度では計算上約 1.2V になります。この電圧はバンドギャップ電圧と呼ばれています。今回使用した温度センサ LM74 もこのバンドギャップを利用しています。

温度センサの出力はアナログ電圧値として得られるので、CPU で扱うためには A/D コンバータを使って電圧データをデジタル化する必要があります。今回使用した温度センサでは図 3 のとおり、バンドギャップ電圧による温度センサに加えて 13 ビット(12 ビット+符号)の $\Delta \Sigma$ 型の A/D コンバータや 3 線式シリアルデータ通信インターフェースまで内蔵しています。ブロック図には描かれていませんが、基準電圧源も内蔵しているため、外部で安定した電圧を作成するといった面倒もありません。

つまり、LM74 は電源電圧さえ与えれば、シリアルデータポートから変換済みの温度データを得ることができます。



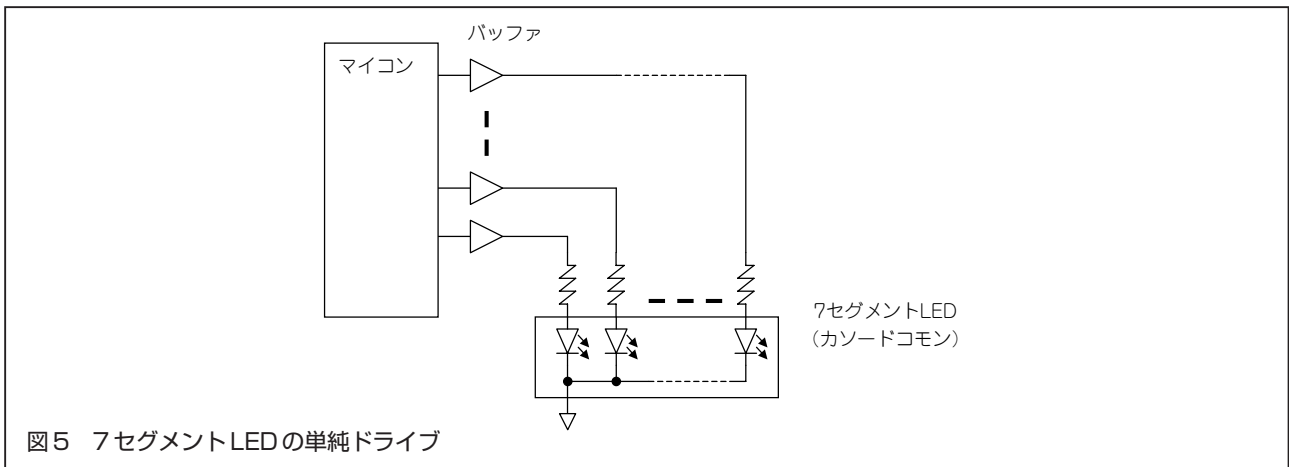
● LM74のシリアル通信方法

LM74のシリアル通信インターフェースの動きを図4に示します。これはSPIやMicrowireに似せたクロック同期式のシリアル・インターフェースで、CS(チップセレクト)信号をアサートした状態の時、ホスト(今回は78K0S/KA1+)が出力するクロックに同期して、データ入出力を行います。最初の16クロックがLM74のデータ出力期間、続く16クロックがデータ入力(LM74への書込み)期間になります。

データ出力期間には、温度データまたはIDが上位ビットから順に送り出されます。データ入力期間はコンフィグレーションレジスタへの書込み期間も兼ねており、こちらも上位ビットから与えます。データが不要な場合には、前半の16クロックだけで終了してもかまいません。なお、今回は温度データとして有効な上位13ビット分だけ読んだ後でCSをネゲートし、アクセスを終了していますが、これでも特に問題なく動作するでしょう。

LM74からの出力データがIDコードになるか、温度データになるかはLM74がシャットダウンモードになっているか、連続変換モードのいずれになっているかによって決まります。このモードの切り替えはコンフィグレーションレジスタへの書込みで行いますが、電源投入後、LM74は連続変換モードで起動しますので、温度データを読むだけであれば、特に何もしなくてもかまいません。

なお、LM74から出力される16ビットデータのうち有効なのは上位13ビットです。ビット2(D2)は温度データ時は常に'1'、IDデータ時は'0'です。下位2ビット(D1、D0)はハイ・インピーダンスになります。データラインは通常プルアップしますので下位3ビットが'1'として読み出されます。



温度データの表示

● 7セグメントLED

スタータ・キット上で温度データを表示するため、7セグメントLED(いわゆる日の字のLED)を4個搭載しました。右端は小数点1桁目のデータに使用し、一番左の桁は氷点下のときに“-”を、100℃以上のときに“1”を表示するために使用しています。つまり、-99.9℃から199.9℃まで表示可能なので、LM74自体の温度測定範囲(-50℃~+150℃)には十分です。

LEDはアノードとカソードの二つの電極を持っているため端子をそのまま出してしまうと14個の端子が必要になりますが、実際の使われ方ではいずれか一方は共通にし、共通ピン1本とセグメント毎に7個の端子を持つものが普通です。

LED nアノード側を共通化したものをアノードコモン、カソード側を共通化したものをカソードコモンと呼びます。今回はカソードコモンタイプのLEDを使っています。

● 7セグメントLEDの駆動

7セグメントLEDは7セグメント+コモン端子という構成なので、セグメント1個ごとにマイコンのI/Oピンを割り付けていけば図5のようにLEDの数の出力ポート、7セグメントならば7個のポートが必要になります。

この調子で4桁にすると、7セグメント×4桁で28個のポートが必要になります。これでは必要なポートの数が多すぎるので、次のような考え方でポートを節約します。

- 1)各桁の同じセグメント(a同士、b同士・など)のアノードを接続
- 2)ある桁に表示したいパターンデータでアノードをドライブ
- 3)点灯させたい桁のカソード(コモン端子)だけを‘L’レベルにする
- 4)点灯させる桁をひとつずつずらしていく

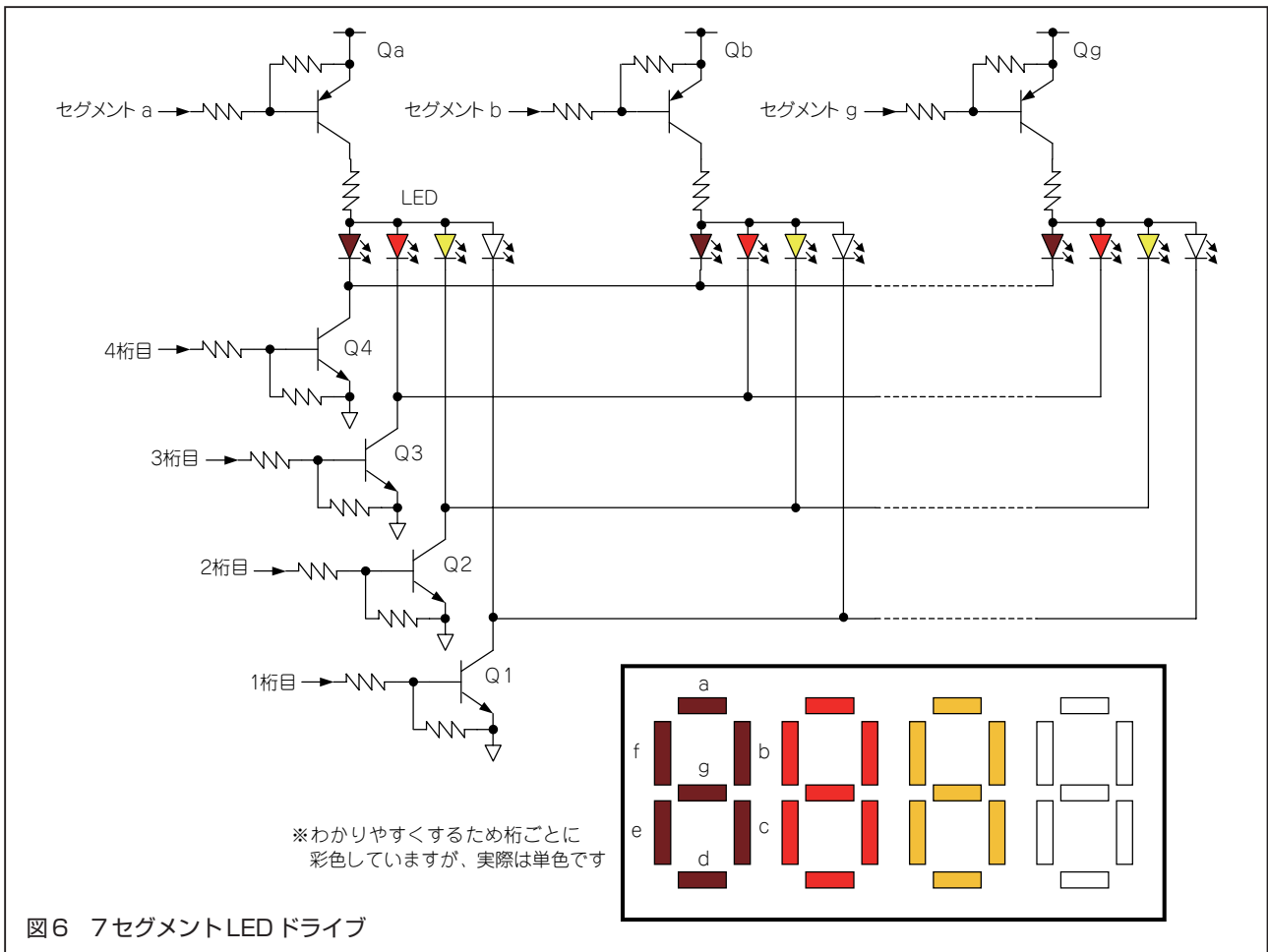
ポイントは4)です。実際には1桁ずつ順に点灯しているのですが、速い速度で移動させていけば、人間の目の残像現象のおかげで全てが同時に点灯しているように見えるという理屈です。

図6は今回の回路図からLED駆動部分を取り出して書き直したものです。電源側にあるQa~QgのPNPトランジスタがセグメントの駆動用のスイッチで、入力信号が‘L’レベルになる(ポートに‘0’を書く)とON状態になります。ひとつのPNPトランジスタに4桁分のLEDが接続されています。

下の方にあるQ1~Q4のNPNトランジスタは、どの桁のLEDを点灯させるかを指定するもので、入力を‘H’レベルにする(ポートに‘1’を書く)とその桁のトランジスタがONになります。こちらは桁単位でひとつのトランジスタにつながります。ここではQ4が一番左の桁、Q1を一番右の桁にしています。CPU側ではQ1~Q4のうちONにするのはひとつだけにします(複数同時にONにすると、ONになった桁がすべて同じ表示になります)。

LEDは両端にあるPNPとNPNの両方がONになっている時だけ点灯します。たとえば、図の一番左側のLEDはQaとQ4が両方ONのときだけ点灯し、一番右側のLEDはQgとQ1が両方ONのときだけ点灯します。

もし、Qa、Qb、Qc、Qd、Qe、Qfの6つがON、QgがOFFの状態でもQ4をONにすれば、一番左側の7セグメントLEDに“0”が表示されますし、Q3をONにすれば左から2桁目の7セグメントLEDが“0”表示になります。



プログラム

プログラムは全てアセンブラで記述しています。プログラムのフローチャートは図7のようになっています。ここで、具体的にどのような処理になっているのか、補足説明をします。

● オプション・バイトとプロテクト・バイト

CPUが動く前の挙動として、フローチャートには現れませんが、0080H番地にあるオプション・バイトは78K0S/KA1+のリセット後の安定時間やリセット端子の機能、システムクロック源の選択、低速オシレータをCPUで停止可能にするか否かなど、CPUが動き出す前の設定を指定します。リセット時にオプション・バイトの値が自動的に読み取られて、設定が行われます。また、0081H番地にはプロテクト・バイトがあり、内蔵フラッシュメモリブロックの消去や書き換えをCPUから行えるようにするか否かを設定するようになっています。

オプション・バイトの説明は、78K0S/KA1+のユーザーズ・マニュアル(*)の17章、プロテクト・バイトについては18章に説明がありますので、参考にしてください。

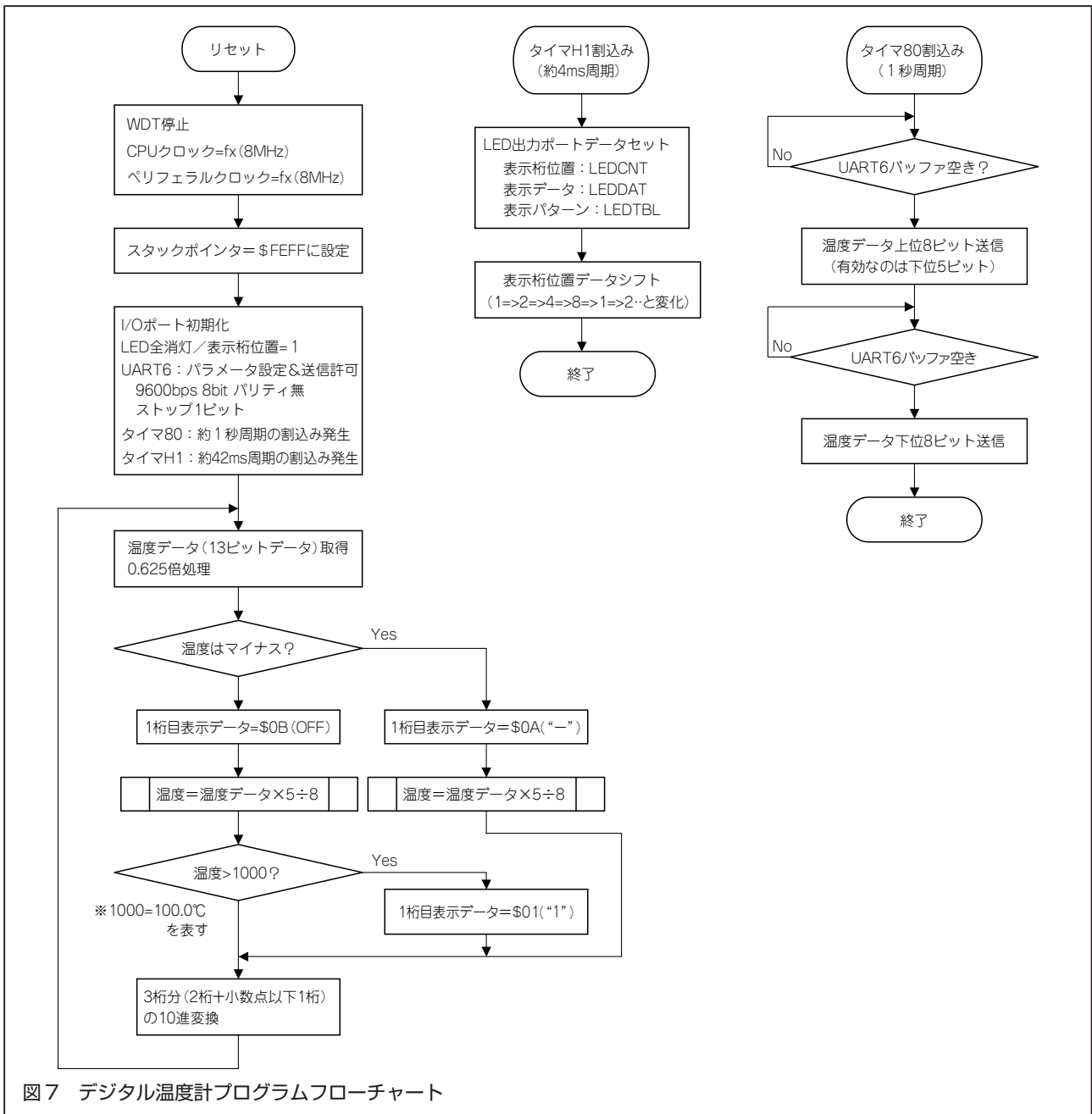
これらのレジスタへの書込みはチップのプログラミング時に行われます。ソースコード中では以下のようになっています。

```
ORG 0080H ;Option Byte
DB 11110110B ;13ms,Port,lnCLK,Stop OK
DB 11111111B ;Protect Byte ALL Disable Set
```

この設定ではオプション・バイトの設定によって

- ・リセット解除後の発振安定時間：13.1ms
- ・リセット端子はP34(入力ポート)として利用

*ユーザーズ・マニュアルは<http://www.necel.com/micro/ja/campaign/kousaku/try/index.html>からダウンロードいただけます。



・高速内蔵発振クロック(約8MHz)で動作
となり、更にプロテクト・バイトはFFHを設定してロック、書き込み、消去など全てを許可しています。

●リセット

電源投入後、78K0/KA1+は内蔵のリセット回路によってリセットがかかります。電源投入後も外部からやWDTなどによってもリセットがかかります。

リセット後、内部レジスタ類も初期化され、割り込みは禁止状態(プログラムステータスワードのIEビットが'0'状態)になります。その後リセットが解除されると、まずオプション・バイトの設定に基づく内部設定が行われ、続いてCPUのリセットが解除されCPUが動き始めます。

78K0Sではリセット時は0000H番地と0001H番地に書かれたデータ(リセットベクタと呼んでいます)を読み出して、この値を命令アドレスポインタであるPC(プログラムカウンタ)にセットします。つまり、0000H番地と0001H番地に書かれたアドレスからプログラムが実行されていくこととなります。

プログラムリスト中では

```
ROM CSEG AT 0000H
```

```
DW START ;Reset
```

と書かれている部分がこのリセットベクタです。この場合にはSTARTラベルのアドレスが0000H番地と0001H番地にセットされますので、STARTラベルからプログラムの実行が開始されることとなります。

● WDTやクロックソースの選択

プログラムのスタート位置のコードは以下のようになっています。ここではWDTの停止、CPUクロックや周辺回路用のクロック選択を行っています。今回はCPUクロック、周辺回路用のクロックとも8MHzにしています。

```
START:MOV WDTM,#01110111B ;WDT Stop
      MOV PCC,#00000000B ;Fx 1/1 Set
      MOV PPCC,#00000000B ;PFx 1/1 Set
```

● スタックポインタの初期化

スタックポインタ(SP)というのは、CALL命令実行時や割り込み処理時の戻り先番地や、PUSH命令でのデータ保存を行ったり、RET/RETI/POP命令実行時に保存したデータの取り出しを行うためのアドレスポインタで、データの格納や取り出しのときに自動的にデクリメント/インクリメントされます。

78K0Sの場合、スタックポインタは最後にデータを書き込んだアドレスを指すようになっており、データを格納するとき

- スタックポインタの値を-1する
 - スタックポインタの指すアドレスにデータを格納する
- という動作になります。逆にデータを取り出すときには
- スタックポインタの指すアドレスからデータを取得する
 - スタックポインタの値を+1する

となります。

たとえば、図8のようにスタックポインタの値がFFE0HだったときにPUSH DE命令(DEレジスタの値をスタックに格納する)という命令を実行すると、FFDFH番地にDレジスタの値が、続いてFFDE番地にEレジスタの値が書き込まれます。スタックポインタの値は、この命令実行後にFFDEH番地(最後に書き込んだEレジスタの格納先)になります。

この後DEレジスタを使用して値が変更されてもPOP DE命令を実行すれば、先ほどスタックに格納した値がDEレジスタに戻されます。

スタックポインタの初期化を行っているのは、プログラムリスト中では以下の部分です。

```
MOVW AX,#0FEFFH ;SP Set
MOVW SP,AX
```

78K0S/KA1+ではSPレジスタで指定できるアドレスは高速SRAM領域(FE00H~FEFFHの256バイト)ですので、この最後尾アドレスをセットしました。SPレジスタに直接セットできないので、一回AXレジスタにセットしてから転送しています。

● I/Oポートの初期化

I/Oポートの設定関係のレジスタは78K0S/KA1+ ユーザーズ・マニュアルの4.3章に説明が出ています。レジスタを機能別に大きく分けると、

- ポートデータアクセス用のレジスタ
 - ポートの入出力モード設定用のレジスタ
- の二つに分類できます。

前者はポートレジスタで、P2、P3、P4、P12、P13という名称です。これをリード/ライトすることで、入力ポートのデータを読み出したり、出力ポートにデータをセットすることができます。リセット後は00Hになっていますので、なにもせずにポートを出力ポートに設定すると、'L'レベルが出力されます。

後者は次の3つのレジスタが関係しています。

(1)ポート・モード・レジスタ(PM2、PM3、PM4、PM12)

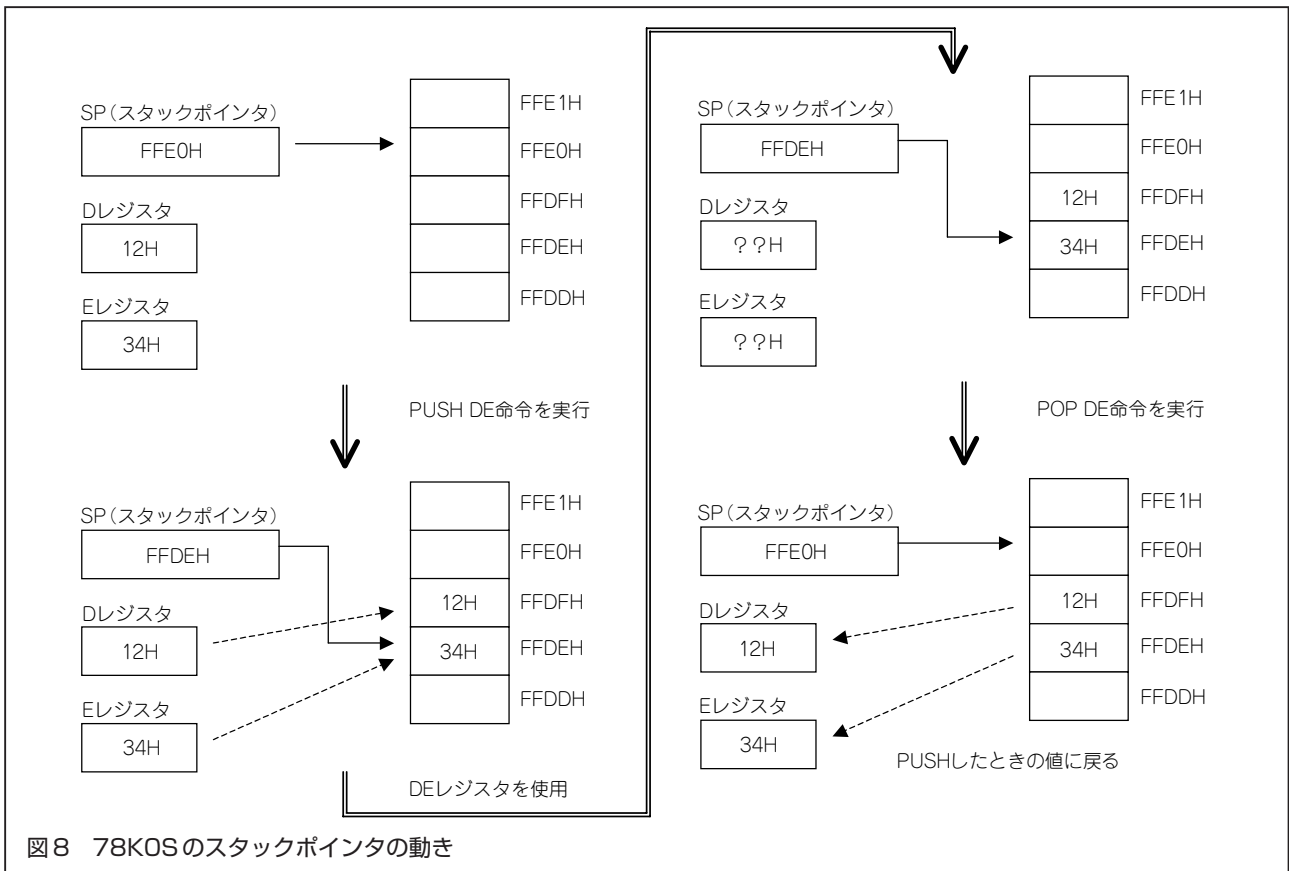


図8 78K0Sのスタックポインタの動き

(2)ポート・モード・コントロールレジスタ(PMC2)

(3)プルアップ抵抗オプションレジスタ(PU2、PU3、PU4、PU12)

ポート・モード・レジスタはそのポートが入力になるか、出力になるかを定めるものです。該当するビットが'1'だとそのポートが入力ポート、'0'だと出力ポートになります。リセット後はFFHになりますので、全て入力ポートになっています。

ポート・モード・コントロールレジスタはP20～P23の4本をアナログ入力(A/Dコンバータへの入力端子)として使うか否かを設定するものです。'1'をセットすると、アナログ入力モードになります。リセット後は00Hになりますので、すべてI/Oポートになります。プルアップ抵抗オプションレジスタはポートに内蔵されているプルアップ抵抗を利用するか否かを定めるもので、'1'を設定するとプルアップ抵抗が接続されます。リセット後は00Hになっており、プルアップされていない状態です。

● UART6の設定

UART6の説明は78K0S/KA1+ユーザズ・マニュアルの11章にあります。今回はUART6を温度データの送信用に使います。通信パラメータは次のとおりです。

- ・ビットレート：9600bps
- ・データ長：8ビット
- ・パリティ：無し
- ・ストップビット長：1ビット

UART6の通信速度は、CKSR6レジスタで選択したクロックソースを1/2にして、更にBRGC6レジスタに設定された値で更に分周したものになります。今回はCKSR6を4分周、BRGC6レジスタで104分周を指定しましたので、

$$8\text{MHz} \div 4 \div 2 \div 104 \approx 9615$$

となり、ほぼ9600bpsになります。データ長などを設定するASIM6レジスタはリセット後は00Hになっていますので、'1'をセットする必要がある部分だけ設定しました。今回はビット2でデータ長8ビットの指定を行っている部分、およびUARTの動作開始(Power Enable)、送信許可(Tx Enable)ビットの三つが'1'に変更しなくてはならないビットです。

モード設定後、ASIM6のビット7でUART6の動作許可を行い、ビット6で送信動作をイネーブルすることで送信動作が行えるようになります。受信動作は使用しないので、ディセーブルのままにしています。

UART6の初期化部分のソースコードは以下のようになっています。

```
MOV BRGC6,#104 ;UART 9600bps Set(2MHz ÷ 104 ÷ 2 ÷ 9615 ÷ 9600bps)
MOV CKSR6,#00000010B ;F=1/4(内蔵8MHz ÷ 4=2MHz)
SET1 ASIM6.2 ;Data 8Bit Set
SET1 ASIM6.7 ;UART Power Enable
SET1 ASIM6.6 ;Tx Enable(Rx(ASIM6.5)は使わない)
```

● タイマの初期化

今回は8ビットタイマ80とタイマH1を使用して、約1秒周期の割り込みと、約4msおきの割り込みを生成させています。タイマ80は78K0S/KA1+ユーザーズ・マニュアルの7章、タイマH1は8章に説明があります。

タイマ80、タイマH1とも、あらかじめモードコントロールレジスタ(TMC80レジスタ/TMHMD1レジスタ)で選択したクロックでカウンタ(TM80レジスタ/H1レジスタ)がインクリメントされていき、コンペアレジスタ(CR80レジスタ/CMP01レジスタ)の値と一致すると、CPUに割り込みが入ると共にカウンタが0に戻るという動作を繰り返します。

割り込み発生周期は

(カウンタ用のクロック周期) × (コンペアレジスタの設定値)

になります。ソースコード上では

```
MOV TMC80,#00000110B (周辺クロック ÷ (216)を選択)
MOV CR80,#7AH
```

```
MOV TMHMD1,#01000000B (周辺クロック ÷ (212)を選択)
MOV CMP01,#8
```

となっています。今回、周辺クロックは8MHzですので、それぞれのタイマの周期は

タイマ80 : $1 \div 8\text{MHz} \times (2^{16}) \times 122 (= 7AH) \div 1$ (秒)

タイマH1 : $1 \div 8\text{MHz} \times (2^{12}) \times 8 \div 4$ ms

となります。

続いてIF1レジスタ/IF0レジスタの割り込みフラグをクリアし、MK0レジスタ/MK1レジスタでタイマの割り込みを許可した後、TMC80とTMHMD1レジスタでタイマ動作をスタートさせます。

このままではまだCPU自体の割り込みが許可になっていないので、EI命令でCPUの割り込みも許可します。ソースコードは以下のようになります。

```
CLR1 IF1.3 ;Timer80 Interrupt Flag Clear
CLR1 MK1.3 ;Interrupt Enable
CLR1 IF0.4 ;TimerH1 Interrupt Flag Clear
CLR1 MK0.4 ;Interrupt Enable
SET1 TMC80.7 ;Timer80 On
SET1 TMHMD1.7 ;TimerH1 On
EI ;Interrupt Enable
```

● 温度データの取得

温度センサとの間は、

- P34:温度センサデータ(SIO)
- P123:温度センサチップセレクト(nCS)
- P130:温度センサシリアルクロック(SC)

の3ビット分のI/Oポートを使っています。プログラムがアセンブラで書かれているので、C言語風に整理したものが次のリストです。アセンブラのソースコードとつき合わせて見ていただければわかりやすいと思います。

```

unsigned short TMPDHL,TMPDHLU;
SC = 0;           // クロックを 'L' にしておいて
nCS = 0;         // チップセレクトアサート
for (B = 13; B > 0; B--) {
  SC = 1;        // クロックを 'H' にして
  TMPDHL <<= 1; // データ取得
  if(P34 & 0x10)
    TMPDHL |= 1;
  SC = 0;        // クロックを 'L' に戻す
}
nCS = 1;         // チップセレクトネゲート
TMPDHLU = TMPDHL; //UART 送信用データバッファにコピー

```

● 0.625倍の処理

取得されるのは13ビットのバイナリデータなので、これを10進データに変換する必要があります。温度センサから出力されるデータは0℃のときに0で、0.0625℃単位の2の補数表現です。たとえば、0010Hならば、 $0.0625 \times 0010H (= 16) = 1.0^\circ\text{C}$ で、1FFFHなら -0.0625°C 、1FFEhなら -0.125°C となります。

温度は小数点以下まで表示します。今回のように小数点以下の桁数が固定でよい場合には浮動小数点演算よりも簡単な固定小数点演算で行う方法があります。固定小数点というのは、整数演算で処理しておいて、下位何桁かを小数点以下とみなす方法です。先ほどの例であれば

$$\begin{aligned}
 4 \times 1.3 &= 4 \times (1.3 \times 10) \div 10 \\
 &= 4 \times 13 \div 10 \\
 &= 52 \div 10
 \end{aligned}$$

という具合に変形することに相当します。つまり、1.3という小数点を含んだ数値をあらかじめ10倍して整数にして計算し、後で一番下の桁は小数点以下1桁目とみなせば良いという理屈です。今回は、温度センサからのデータをNとすると

$$\begin{aligned}
 N \times 0.0625 &= N \times (0.0625 \times 10) \div 10 \\
 &= N \times 0.625 \div 10 \\
 &= N \times (5 \div 8) \div 10 \\
 &= (N \times 5 \div 8) \div 10
 \end{aligned}$$

となりますので、Nを5倍して8で割ったものを10進変換して、下位1桁を小数点以下とみなせば良いのです。

この $N \times 5 \div 8$ の演算をおこなっているのはソースコード中のTPAX58サブルーチンです。2で割るという操作と1ビット右にシフトするのは同じことなので、2回右シフトは4で割ること、3回右シフトは8で割ることとなります。これらを踏まえて処理内容をC言語風に見直し直したのを見てみましょう。

```

DE = TMPDHL;
for (B = 4; B > 0; B--) { // ×5
  DE+ = TMPDHL;
}
for (B = 3; B > 0; B--) { // ÷8
  DE >>= 1;
}
AX = DE;
5倍にする演算は
N × 5 = N × 4 + N

```

と書き直せますので、2ビットの左シフト演算と加算で行うこともできますが、今回はループで5回加算して5倍するという簡単な方法をとっています。

● 10進変換

以上の方法で、温度データを10倍した値は得られましたが、この段階ではたとえば25.0℃ならば、FAH(=250)という、バイナリデータになっているだけです。LEDに10進表示するためにはこれを“250”という10進データ(BCDデータ)に変換しなくてはなりません。

今回の温度センサICの測定範囲は-55℃～+150℃ですので、

(1)温度がマイナス(ビット12が‘1’)ならば4桁目を“-”にして符号反転(全ビットを反転して1を加える)

(2)温度が1000以上(100.0℃以上)ならば、1桁目を“1”にして1000を引く

という方法をとれば、以下の処理では000H～3E7H(0～999)の範囲を変換できれば良いことになります。

バイナリから10進への変換はセオリーどおりなら、10で割り算して余りを求めるという操作を繰り返すこととなりますが、今回は引き算で処理をしてみました。割られる数は0～999の範囲にあるので、まず100を引けるだけ引いて、引けた回数を100の桁、次に10を引けるだけ引いて、引けた回数が10の桁、余ったのが1の桁の数という方法をとってみました。

たとえば、3D5(=981)ならば、100を引く、引けなくなるまでの回数は9で、この段階での余りは51H(=81)です。続いて10を引いていくと同様に8回引けて余りが1になります。

これで、9、8、1という値が得られ、10進で981であることがわかるという仕組みです。C風に書いてみると次のようになります。

```
B = 0xff;          // 一回目で0になるようにするため、最初はFFHにしておく
do {
    B++;
    AX -= 100;
} while(AX >= 0);
AX += 100;        // 引きすぎてからループを抜けるので補正しておく
LEDDAT[1] = B;    // 100の桁
B = 0xff;
do {
    B++;
    AX -= 10;
} while(AX >= 0);
AX += 10;        // 引きすぎてからループを抜けるので補正しておく
LEDDAT[2] = B;    // 10の桁
LEDDAT[4] = X;    // 1の桁 (今回はLEDDAT[3]は使わずに飛ばしてLEDDAT[4]を使う)
```

ソースコードからわかるとおり、実際には引きすぎたことを知ると元に戻すという操作を行いますので、最大で10×(桁数-1)回の引き算を行うことになります。

できあがった表示データはLEDDATに格納されます。普通ならばここでLEDDAT～LEDDAT+3の範囲を使うところですが、今回はLEDDAT、LEDDAT+1、LEDDAT+2、LEDDAT+4を使うようにしています。これはLEDの表示桁指定用の変数であるLEDCNTが1=>2=>4=>8=>1・・・と変化するため、LEDCNTを1/2にすれば(1ビット右シフトすれば)0=>1=>2=>4=>8となり、そのままLEDDATに足すオフセット値にできるためです。LEDDAT+3が使われないので1バイト分無駄になりますが、RAM容量が厳しい状態ではありませんので問題はないでしょう。

● 割り込み時の挙動

タイマ割り込みを見ていく前にCPUの割り込み応答動作について説明しておきましょう。割り込みを受け付けたときの挙動もまたCPUの種類によって異なりますが、78K0Sの場合には次の動きをします。

- スタックに現在のフラグレジスタ(PSW)の値を格納
- プログラムカウンタの値(上位、下位の順)に格納
- 割り込み許可フラグ(PSWレジスタのIEフラグ)をクリア(割り込み禁止)

- 割り込み要因に応じたアドレスに書かれたとび先番地データ(ベクタ)を読み出し
- 読み出されたアドレスにジャンプ

どの割り込み要因の場合にどの番地に書かれたデータ(ベクタ)が参照されるかは製品によって決まっています。今回はタイマH1とタイマ80を使いますが、78K0S/KA1+ではそれぞれ000CH、001AHが使われます。

ベクタの設定はソースコード中では以下のようになっています。タイマH1 割り込み処理プログラムはINTTH1、タイマ80 割り込み処理プログラムはINTTM8 というラベルのところから記述されています。

```
ORG 000CH ;TimerH1
DW INTTH1

ORG 001AH ;Timer80
DW INTTM8
```

なお、割り込み処理の終わりはRETI命令を使用します。RET命令ではありませんので、注意してください。RETI命令を実行すると、スタックに格納されていた割り込み発生時のPCの値や、フラグレジスタの値が再セットされ、中断されていたプログラムの処理を再開します。なお、PCとフラグ以外のレジスタはPUSH/POP命令を使うなどして、RETI命令実行前に戻しておきます。たとえば、割り込み処理の中でAXレジスタとHLレジスタを使うならば

```
PUSH AX
PUSH HL
... (割り込み処理)
POP HL
POP AX
RETI
```

という具合にすれば良いのです。PUSHとPOPの順番や数を間違えないように気をつけてください。

● タイマH1 割り込み

タイマH1は約4mS周期の割り込みで、割り込みが入る度に点灯させる7セグメントLEDを1桁ずつシフトしながら表示を行います。

表示する桁位置を指定するのがLEDCNT変数で、“-”表示、ブランク表示(いずれも4桁目用)、そして0~9の数値のなかからどれを表示するかを決めるのがLEDDAT[]配列の中身です。

LEDDAT[]配列の中にあるデータから7セグメントLEDのa~gのどのセグメントをドライブするかを決めるのがLEDTBLです。今回はドライブ用にポート3とポート4を使っていますので、ポート3とポート4用のデータが交互に並んでいます。

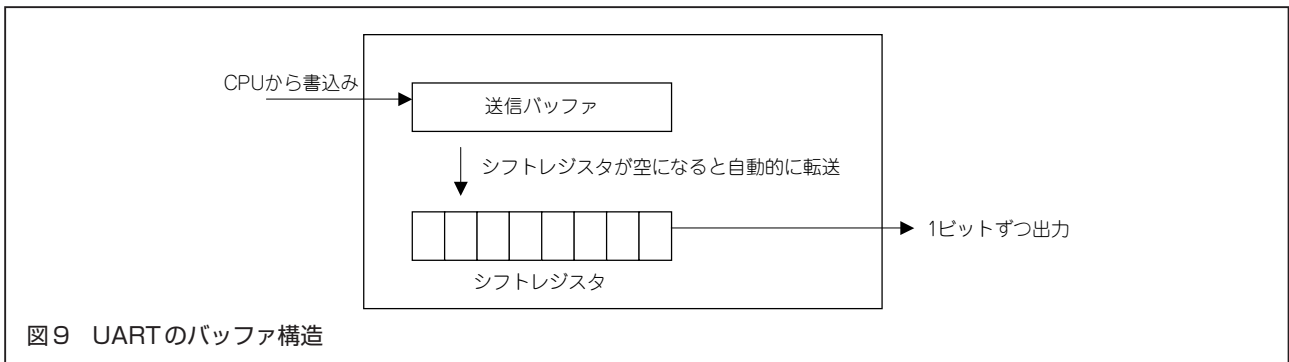
LEDTBLは次のようになっています。

```
LEDTBL: DB 10H,28H,11H,3EH,10H,19H,10H,1CH ;0-3
        DB 11H,0EH,12H,0CH,12H,08H,10H,3EH ;4-7
        . . . . . (以下略) . . . . .
```

たとえば、“0”を表示したいならば、ポート3に10H、ポート4に28Hをセット、“5”を表示したいならば、ポート3に12H、ポート4に0CHをセットします。

LEDCNTとあわせて、C言語風には書けば次のようになります。

```
P3 = LEDTBL[LEDDAT[LEDCNT/2]][0];
P4 = LEDTBL[LEDDAT[LEDCNT/2]][1];
P2 = LEDCNT;
LEDCNT <<= 1;
if (LEDCNT & 0x80) LEDCNT = 1;
```



● タイマ80 割り込み

タイマ80は1秒周期の割り込みで、この中でシリアルポートに温度データを送ります。UART6は図9のように

- 送信バッファ
- シフトレジスタ

の二つのレジスタを持っていて、

- (1)シフトレジスタが空になる（送信が完了する）
- (2)送信バッファからシフトレジスタにデータを自動転送
- (3)送信バッファが空になる
- (4)送信完了

(5)次のデータが送信バッファに入っていないならば送信バッファも空になる

という具合に動作します。もし(3)の段階で、CPUが次の送信データを送信バッファに書き込んでおけば、(1)の段階に戻り間髪いれずに次のデータ送信が始まるため、効率の良い転送が行えます。

送信データバッファの状態を示すのはASIF6レジスタで、ビット0のTXSF6は送信が完全に終わった(送信バッファ、シフトレジスタの両方も空になった)事を示します((5)の段階に相当)。一方、ビット1のTXBF6はデータバッファが空になったことを示すもので、(3)の段階に相当します。

今回はビット1のTXBF6を見て、送信バッファが空くまでウェイトして上位データを送信、再び送信バッファに空きができるまでウェイトして下位データを送信という手順で16ビット分のデータを送信しています。

● アセンブル

このようにして出来たソースがmain.asmです。これをアセンブル、リンクしてダウンロード用のHEXファイルなどを生成するのですが、PM+を使わずに、DOSウィンドウから直接アセンブラ、リンカ、オブジェクトコンバータを起動して、処理させてみました。

とはいえ、扱いは簡単で、次のようなコマンドを実行するだけです。

```
RA78K0S - CF9222 MAIN78.ASM
LK78K0S MAIN78.REL
OC78K0S MAIN78.LMF
```

これを毎回手で入力するのは面倒なので、asm.batというバッチファイルを用意しました。

このようにして出来上がったHEXファイルを書き込んで実際に製作した温度計を動かしてみたのが最初の写真です。

— おわりに —

無償で用意されたAppliletも、GUIベースの統合開発環境であるPM+も、Cコンパイラも使わずに、すべて手作業でアセンブラで記述し、DOSウィンドウ上で動くアセンブラベースのコマンドラインツールとして利用した例を紹介しました。

なんとモクラシックな開発方法ではありますが、250行程度のコンパクトなソースコードの中にリセット直後のコードや割り込みのエントリ部分、オプションバイトの設定などのCPU動作の細かな部分や、10進変換、固定小数点演算の考え方なども含まれており、勉強用の素材として参考になる部分も多いのではないかと思います。

編集 桑野 雅彦