C から体感する 78K0R マイコン

R1.00 2010/5/18
R1.50 2011/2/27
R2.00 2011/3/29

3章　ハードなC言語

この章ではC言語でハードウエアを操作する方法について説明します。

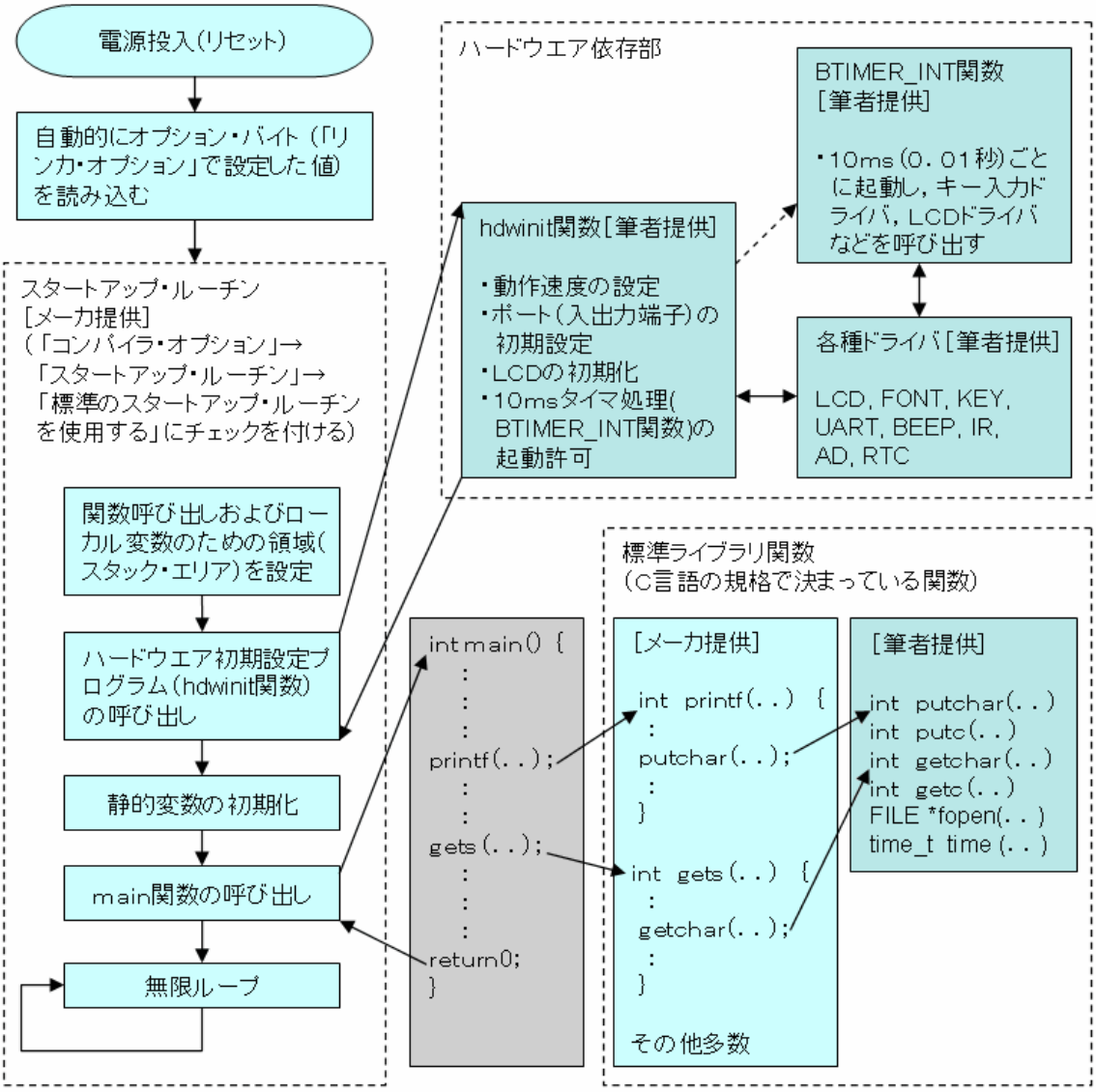
3.1　Cから見たマイコンの基礎知識

まずはマイコンも初めてという人のために，マイコンの基礎的な振る舞いや基本構造（アーキティクチャ）を説明します。

3.1.1　main を取り巻くマイコン全体の動き

これまでは main 関数だけ動作しているように書いてきましたが，実際は図 3−1 のようにさまざまなプログラムが動作します。順を追って概要を説明します。

図 3−1　全体のプログラム構成（３章までの構成）



(1)電源投入

マイコンに電源を入れただけでは，一般的には暴走してしまいます。適切な場所からプログラムを開始させるためにリセット信号を入力する必要があります。

ただし，78K0R は POC（パワー・オン・クリア）と呼ばれるリセット回路を内蔵しているので，通常は電源を入れただけでも動作を開始します。

(2)オプション・バイトの読み込み

78K0R にはメモリ上に「オプション・バイト」と呼ばれる領域があり，リセット直後にこの値を自動的に読み取り，一部のハードウエア機能を自動的に初期化しています。

詳細は 3.1.4　オプション・バイトを見てください。

(3)スタートアップ・ルーチン

マイコンはリセット後（オプション・バイト読み込み後）にメモリ上の特定のアドレスからプログラムの実行を開始します。78K0R の場合は，メモリの 0 番地（リセット・ベクタ格納アドレス）に書かれているアドレスから実行を開始します。

一般的には，ここからユーザーがプログラムを書かねばなりませんが，CC78K0R では main 関数を呼び出すための標準のスタートアップ・ルーチンが提供されていて，ハードウエア依存部以外は自分でプログラムを書かなくても良いようになっています。

標準のスタートアップ・ルーチンを使うには，

「コンパイラ・オプション」→「スタートアップ・ルーチン」→

「標準のスタートアップ・ルーチンを使用する」

にチェックを付けます。

ハードウエアに依存した部分については，スタートアップ・ルーチンから hdwinit 関数を呼び出すようになっており，ユーザが hdwinit 関数を書くことになります。

(4)ハードウエア初期設定（hdwinit 関数）

main 関数の中でハードウエア初期設定をすることも可能ですが，一般的にはリセット後のなるべく早い段階で済ましておきたい設定が多いので，hdwinit 関数を書く方が良いでしょう。

78K0R はリセット直後の初期状態では，動作速度が 4MHz（内蔵 8MHz 発振器の半分），ポート（入出力端子）の多くが入力状態，その他のほとんどの内蔵機能は停止となっています。

どれを設定（変更）するかは外部回路の状況にもよりますが，筆者提供の hdwinit 関数では，次の設定を行っています。

・動作速度を 20MHz に上げる。

・LCD やキーなどの外部回路を制御するためにポートの入出力方向や初期値を設定する。

・OS(ｵﾍﾟﾚｰﾃｨﾝｸﾞ・ｼｽﾃﾑ)の代わりとなる定周期タイマの初期化関数を呼び出す。

(5)その他ハードウエア依存部

BTIMER_INT 関数はパソコンで言うと OS に相当する部分です。大きな目で見ると，この関数は main 関数と平行して独立に動作します。細かく見ると，0.01 秒ごとに main 関数を中断して，BTIMER_INT 関数が終わったらまた main 関数に戻るという動きをします。

BTIMER_INT 関数ではキー入力ドライバ，LCD 表示ドライバなどの定期的な呼び出しを行っています。

キー入力ドライバは，どのキーが押されているか判別したり，文字コードへの変換を行います。

LCD 表示ドライバは原理的には printf 関数から LCD への表示要求があるごとに呼び出せばよいのですが，本環境ではキーからの表示要求が printf からの表示要求に割り込んで漢字が乱れるのを防ぐため，定期呼び出しの中で LCD 表示制御を行っています。

(6)main 関数

ここからが本格的なユーザ・プログラムとなります。実際にプログラムをデバッガで起動してリセットしてみると，main 関数の最初の行にカーソルが来ます。

つまりスタートアップ・ルーチンや hdwinit 関数は見えなくて，本当に main 関数から動き始めるように見えます。

(7)標準ライブラリ関数

C 言語の規格ではいろいろなライブラリ関数が決まっています。パソコン用のＣコンパイラであれば，それらが全て標準で提供されているでしょう。

マイコンの場合はユーザが作るハードウエアがまちまちなので，一部提供されていなかったり，ハードウエアにあわせて変更しなければならない関数があります。

筆者提供の内，putchar，getchar は書き換えを行ったもので，その他は新規に作ったものです。

3.1.2　アドレス空間

一般的にマイクロコントローラには，プログラムを格納する ROM と変数を格納する RAM および各種周辺機能が内蔵されています。これらは大概 1 バイト単位でアドレス（番地）が振られており，マイコンの種類によって単一または複数のアドレス空間の中に配置されます。

78K0R は，20 ビット（=1M バイト=1,048,576 バイト）の単一アドレス空間を持っていて，この中に ROM／RAM などが配置されています（図 3－2 参照）。

①ROM 領域：

プログラムや定数を格納する領域です。78K0R はフラッシュ ROM と呼ばれる書換可能タイプを搭載しています。0 番地から配置されています。

②RAM 領域：

変数（自由に値を変更できる変数）を格納する領域です。電源を切ると内容が消えます。0xFFF00 番地より下側に配置されています。

③SFR 領域：

内蔵ハードウエアに対応付けられた領域です。

CC78K0R では，プログラムの先頭に

#pragma SFR

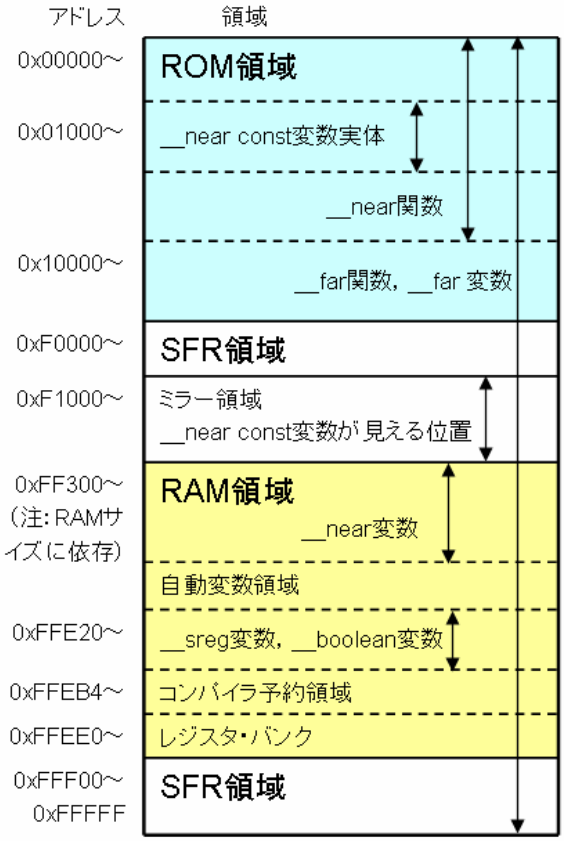
と書くと，特に変数宣言をしなくてもマニュアルに記載してあるレジスタ名を変数名として扱えるようになります。

④ミラー領域：

ROM 領域に格納した定数の一部が写像されて見える領域です。

これは 78K0R 特有の機能で，64K バイトのデータ空間内で ROM/RAM 両方のデータをアクセスするための仕組みです。

図 3－2　アドレス空間（実サイズは品種依存）



3.1.3　関数・変数の配置と型修飾子

さて，78K0R は 16 ビット・マイコンですから，アドレス操作は 16 ビットで行うのが効率的です。そこで，CC78K0R では，16 ビットで操作する__near 領域と，20 ビットで操作する__far 領域に分け，変数や関数の定義の時に__near，__far の型修飾子を付けて区別できるようにしています（表 3－1 参照)。

・__near 関数：0～0xFFFF に配置されます。

・__near 変数：0xF0000 以上に配置され，下位 16 ビットのアドレスで操作します。

・__far 関数/変数：20 ビット・アドレスで操作します。

ただし，いちいち宣言に型修飾子を付けるのは面倒なので,「ツール」→「コンパイラ・オプション」→「メモリ・モデル」のところでデフォルト修飾を選択できるようになっています。

・スモール　：関数，変数とも__near

・ミディアム：関数は__far，変数は__near

・ラージ　　：関数，変数とも__far

デフォルト修飾を変更する関数，変数には__near，__far を付けます。

また静的変数の内，内容が変化しない変数は const 修飾を付けることにより，ROM 領域に配置することが出来ます。__near const 変数の場合，実体は 0xffff 以下の ROM 領域に配置されますが，0xf1000 以上のミラー領域に配置されているように見えます。

ただし ROM からのデータ読み出し速度は RAM より遅いので，高速処理が必要な場合は定数であっても RAM 配置のままにします。

変数の中で特に頻繁に使う静的変数は__sreg 修飾を付けることにより，より効率の良い操作を行うことが出来ます。

表 3－1　おもな型修飾子

型修飾子	内容
__near	関数を 0xffff 以下, 変数を 0xf0000 以上に配置。ポインタのサイズは 2 バイト。
__far	関数，変数を全域に配置可能。ポインタのサイズは 4 バイト。 ただし，１つの変数は 64K バイト以内，ポインタ変数を増減（++，--）しても 64K バイトの境界を越えられない。
const	内容が変更出来ない変数を宣言する。静的変数を ROM に配置する場合に用いる。
sreg, __sreg	使用頻度が高い静的変数につけるとメモリ・サイズや実行時間を節約出来る場合が多い。ただし割当可能なサイズが 147 バイト以内と非常に小さい。

3.1.4　オプション・バイト

　オプション・バイトで設定できる機能は，次の通りです。

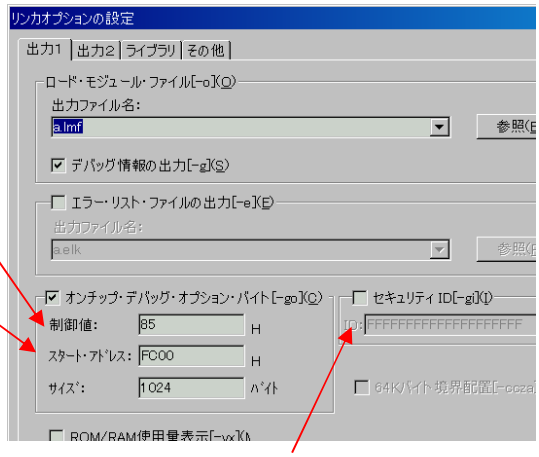
・デバッグに関する設定
・ウオッチドッグ・タイマの設定
・内蔵発振回路，LVI（低電圧検出回路）の設定

(1)デバッグに関する設定

　デバッグの許可／禁止を設定します。プログラム開発する場合は許可に設定します。デバッグが完了して本番のプログラムをビルドする場合に禁止にします。マイコンに書いたプログラムを他人に解析されても問題がなければ許可のままでも構いません。

　設定箇所は，「ツール」→「リンカ・オプション」→「出力１」→「オンチップ・デバッグ・オプション・バイト」のところです。

図 3－3　デバッグ・オプション



制御値：デバッグ許可時は 85 で良いでしょう。
　　禁止時は 04 にします。
スタート・アドレスとサイズ：
　　デフォルト値で良いはずです。
　　マイコンに実際に搭載されて
　　いる ROM 領域の一番最後 1024
　　バイトを指定します。
セキュリティ ID：チェックし，FFFFFFFFFFFFFFFFFFFF か 16 進パスワードを入力します。

(2)ウオッチドッグ・タイマの設定

　ウオッチドッグ・タイマとは，プログラムが暴走したときにマイコンをリセットしてくれるハードウエアです。リセットしたからと言って暴走の原因が解消されるとは限りませんが，ましな状態になる可能性も少なくありません。

　本格的に使いこなすのは難しいので，ここではいくつかの設定例のみ記載します。

　設定箇所は，「ツール」→「リンカ・オプション」→「出力 2」→「ユーザ・オプション・バイト」の最初の 1 バイト目（2 文字）です。

　00：使用しない（動作せず）
　7C：約 1 秒間の暴走でリセット
　7E：約 4 秒間の暴走でリセット

(3)　内蔵発振回路，LVI（低電圧検出回路）の設定

マイコンを動かすにはクロック信号と呼ばれる一定周波数の信号が必要です。

78K0R では，クロックを内蔵発振器で生成したり，外部から入力したり，端子に発振子を接続して生成することが出来ます。

特に 78K0R/Kx3-L シリーズでは，内蔵発振器の精度が高く，かつ 1MHz，8MHz，20MHz の周波数が選択できるように改良されています。オプション・バイトでは，1MHz を選択するか否かを指定します。

また，78K0R では電源電圧の低下を検出できる LVI 回路が付いています。この設定も発振器指定と同時に行います。

設定箇所は，「ツール」→「リンカ・オプション」→「出力 2」→「ユーザ・オプション・バイト」の 2 バイト目（3 文字目と 4 文字目）です。設定例は次の通りです。

①78K0R/Kx3-L（64 ピン以下）の場合

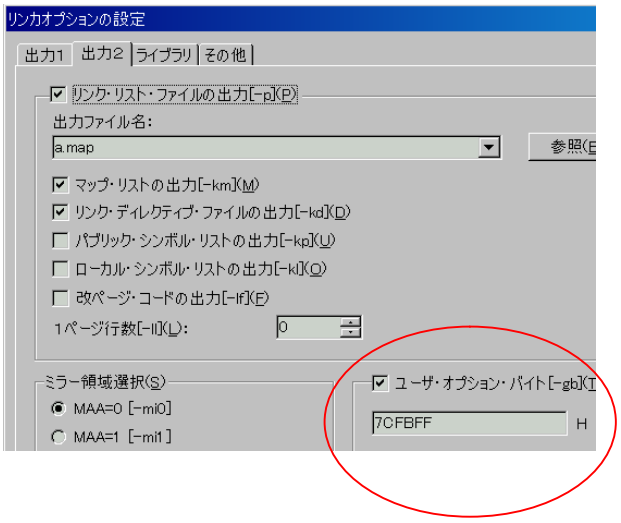
FB：8MHz または 20MHz の内蔵発振器を使用。
　　LVI はリセット後 OFF。

FD：1MHz 内蔵発振器を使用。
　　LVI はリセット後 OFF。

②78K0R/Kx3 の場合

FF：LVI はリセット後 OFF。

図 3－4　ユーザ・オプション



3.2　ポート操作の基本

ここではポート（入出力端子）へ信号を出したり，信号を読み取る操作の基本について説明します。

3.2.1　LED 点滅とキー検出

LED 点滅のプログラム例をリスト 3−1 に示します。

```
/* リスト 3−1 LED 点滅 */
#include "PORT.h"                          /* 3.2.1① */
#include <stdio.h>
#include <time.h>                          /* 3.2.1② */
#define CLOCK() \
    ( (int) clock() / (CLOCKS_PER_SEC / 100) )

int main (void) {
    char line[33];
    int  i, g = 25, r = 50;
    int  gold, rold = gold = CLOCK();      /* 3.2.1③ */

    while (1) {
        if (gets(line) != NULL) {
            rold = gold = CLOCK();         /* 3.2.1③ */
            sscanf(line, "%d %d",
                         &g, &r);          /* 3.2.1④ */
        }
        i = CLOCK();
        if ( i  - gold >= g) {             /* 3.2.1⑤ */
            LED1 ^= 1;                     /* 3.2.1⑥ */
            gold += g;
        }
        if ( i  - rold >= r) {             /* 3.2.1⑤ */
            LED0 ^= 1;                     /* 3.2.1⑥ */
            rold += r;
        }
        if (!KEYIN5)  break;               /* 3.2.1⑦ */
    }
    return 0;
}
```

実行結果 3−1
(赤字は入力例)

```
10 20⊿
```

①PORT.h のインクルード

これはポート名を定義したヘッダ・ファイルです。

メーカ定義のポート名は P6.1 などのように実際の用途と対応づけるのが大変です。

そこで筆者提供環境では PORT.h の中で対応しやすい名前に再定義しています（表 3－3 参照)。

ポート名など内蔵ハードウエアに対応した変数名を SFR 変数名と呼びます。

通常の変数のように値を代入したり参照できます（ハードウエアによっては制限あり)。

メーカ定義の SFR 変数名を使うには,プログラム先頭に次の記述をします(PORT.h 参照)。

```
#pragma SFR
```

このリストで実際使っているのは，次の 3 本のポートです。

LED0，LED1：LED の駆動ポートです（0 を代入すると点灯)。

KEYIN5：キー・マトリクスの最上段，横一列のキーを検出するポートです

（押されている状態で 0)。

なお，これらの端子はキー入力ドライバでも制御しているので，キー操作を行うとちらつきが出る点はご容赦下さい。

②time.h のインクルードと clock 関数

time.h は，時間に関する標準ライブラリ関数のヘッダ・ファイルです。

ここでは,プロセッサ時間を返す clock 関数を使用しています。

C 言語で言うプロセッサ時間とはシステムが起動してからの経過カウント値のことで，時間へ換算するには 1 秒間のクロック数 CLOCKS_PER_SEC で割ります。

上限は処理系依存になっていて，筆者提供環境では 249 日目で 0 に戻ります。

リスト 3－1 では点滅周期を 0.01 秒単位で制御します。また周期はせいぜい数秒の範囲でよいので，clock 関数の戻り値を 0.01 秒単位に換算して int 型でキャストする CLOCK マクロを定義しています。

③点滅開始時間を設定

現在の時刻を点滅開始時刻として gold（LED1 用）,rold（LED0 用）に格納します。

④点滅周期を入力

点滅周期の指定のため，点灯時間，消灯時間を 0.01 秒単位で変数 g（LED1 用）,r（LED0 用）に入力します。初期状態では LED1 が 0.25 秒，LED0 が 0.5 秒で点灯／消灯を繰り返します。なおキー入力時は数字と数字の間はスペースで区切ります。

⑤時間の判定

現在の時間から変数 gold，rold に格納された最初の時間を引いて経過時間を算出し，それを指定値 g,r と比較しています。

⑥ポートの反転

排他論理和演算を使用して，出力ポートを反転（0→1，1→0）しています。

⑦キー検出

KEYIN5 が 0 であれば，break 文によって while(1)のループを抜けます。このように入力信号の読み取りはポート変数を単純に参照すれば良いのです。

押すキーは，BS,NUM，/，*，－のどれでも同じです。回路的にはポート KEYOUT1～KEYOUT5 に 0 を出力しないと検出できませんが，キー入力ドライバが普段そのように設定しているので，このプログラムの中では設定を省略しています。

3.2.2 ポート制御変数

(1)ポート制御変数の種類

リスト 3－1 はポートの初期設定が済んでいる状態でのプログラムでしたが，一般的にはポートを使用する前に入出力方向を決めたり，プルアップ抵抗のオン／オフを設定する必要があります。78K0R では，おもに次の 3 種類の SFR 変数名でポートを制御します。

PMx.y：ポート・モード・レジスタ。

ポートを入力にする場合は 1 を代入し，出力にする場合は 0 を代入します。

リセット後の初期値は一部を除き 1 です。

なお，x，y は次のとおりです。

x：表 3－2 の端子名の番号から下 1 桁を除いた数。

y：表 3－2 の端子名の番号の下 1 桁。

Px.y：ポート・レジスタ。

入力ポートの場合は，外部回路によって 0 または 1 が代入されます。

出力ポートの場合は，代入値（0 または 1）が出力されます。

リセット後の初期値は 0 です。

PUx.y：プルアップ抵抗オプション・レジスタ。

プルアップ抵抗は，主として，ポートを入力にする場合で，かつ入力信号が 0 か 1 か定まらないような時に 1 へ引っ張るために使います。

1 を代入するとプルアップ抵抗がオン，0 を代入するオフになります。

リセット後の初期値は 0 です。全てのポートにこの機能があるわけではありません。

上記以外に一部の端子では PIMx.y（入力モード），POM x.y（出力モード）を設定することが出来ますが，これについてはマイコンのユーザーズ・マニュアルを参照してください。

表 3－2 (a)　端子名，ピン番号（各パッケージごと）　【78K0R/Kx3-L(64 ピン以下)用】

PU	PM	端子名	44 pin 版	48 pin 版	52 pin 版	64 pin 版
U	M	P00			52	62
U	M	P01			51	61
U	M	P10	27	18	33	42
U	M	P11	26	17	32	41
U	M	P12	25	16	31	40
U	M	P13	24	15	30	39
U	M	P14				38
U	M	P15				37
U	M	P16				36
U	M	P17				35
	M	P20	43	35	50	60
	M	P21	42	34	49	59
	M	P22	41	33	48	58
	M	P23	40	32	47	57
	M	P24	39	31	46	56
	M	P25	38	30	45	55
	M	P26	37	29	44	54
	M	P27	36	28	43	53
U	M	P30	12	3	16	19
U	M	P31	13	4	17	20
U	M	P32	14	5	18	21
U	M	P33				22
U	M	P40	2	39	4	5
U	M	P41	1	38	3	4
U	M	P42				3
U	M	P43				2
	M	P60(OD)		1	14	17
	M	P61(OD)		2	15	18

PU	PM	端子名	44 pin 版	48 pin 版	52 pin 版	64 pin 版
U	M	P50	23	14	29	34
U	M	P51	22	13	28	33
U	M	P52	21	12	27	32
U	M	P53				31
U	M	P70	20	11	26	30
U	M	P71	19	10	25	29
U	M	P72	18	9	24	28
U	M	P73	17	8	23	27
U	M	P74	16	7	22	26
U	M	P75	15	6	21	25
U	M	P76			20	24
U	M	P77			19	23
	M	P80	31	22	37	46
	M	P81	30	21	36	45
	M	P82	29	20	35	44
	M	P83	28	19	34	43
U	M	P120	44	37	2	1
0	1	P121	8	45	10	11
0	1	P122	7	44	9	10
0	1	P123	5	42	7	8
0	1	P124	4	41	6	7
0	0	P140		36	1	64
U	M	P141				63
	M	P150	35	27	42	52
	M	P151	34	26	41	51
	M	P152		25	40	50
	M	P153				49

注：OD はオープンドレイン端子（'H'が

PU：プルアップ抵抗オプション指定。U=0（デフォルト値）でオフ，U=1 でオンになる。
PM：ポート・モード指定。M=1（デフォルト値）で入力ポート，M=0 で出力ポートになる。
バイト単位で扱う場合，存在しないビットはデフォルト値で埋めること。

表 3－2（b） 端子名，ピン番号（各パッケージごと） 【78K0R/KE3～KG3 用】

PU	PM	端子名	64 pin 版	80 pin 版	100 pin 0.5	100 pin 0.65
U	M	P00	62	74	97	74
U	M	P01	61	73	96	73
U	M	P02	60	72	95	72
U	M	P03	59	71	94	71
U	M	P04	58	70	93	70
U	M	P05	31	39	42	19
U	M	P06	30	38	41	18
U	M	P10	46	54	69	46
U	M	P11	45	53	68	45
U	M	P12	44	52	67	44
U	M	P13	43	51	66	43
U	M	P14	42	50	65	42
U	M	P15	41	49	64	41
U	M	P16	40	48	63	40
U	M	P17	39	47	62	39
	M	P20	56	68	90	67
	M	P21	55	67	89	66
	M	P22	54	66	88	65
	M	P23	53	65	87	64
	M	P24	52	64	86	63
	M	P25	51	63	85	62
	M	P26	50	62	84	61
	M	P27	49	61	83	60
U	M	P30	32	40	52	29
U	M	P31	21	25	28	5
U	M	P40	5	9	12	89
U	M	P41	4	8	11	88
U	M	P42	3	7	10	87
U	M	P43	2	9	9	86
U	M	P44		5	8	85
U	M	P45		4	7	84
U	M	P46		3	6	83
U	M	P47		2	5	82
U	M	P50	33	41	54	31
U	M	P51	34	42	55	32
U	M	P52	35	43	56	33
U	M	P53	36	44	57	34
U	M	P54	37	45	58	35
U	M	P55	38	46	59	36
U	M	P56			60	37
U	M	P57			61	38
	M	P60(OD)	17	21	24	1
	M	P61(OD)	18	22	25	2
	M	P62(OD)	19	23	26	3
	M	P63(OD)	20	24	27	4

PU	PM	端子名	64 pin 版	80 pin 版	100 pin 0.5	100 pin 0.65
U	M	P64		26	29	6
U	M	P65		27	30	7
U	M	P66		28	31	8
U	M	P67		29	32	9
U	M	P70	29	37	40	17
U	M	P71	28	36	39	16
U	M	P72	27	35	38	15
U	M	P73	26	34	37	14
U	M	P74	25	33	36	13
U	M	P75	24	32	35	12
U	M	P76	23	31	34	11
U	M	P77	22	30	33	10
U	M	P80			44	21
U	M	P81			45	22
U	M	P82			46	23
U	M	P83			47	24
U	M	P84			48	25
U	M	P85			49	26
U	M	P86			50	27
U	M	P87			51	28
U	M	P90		55		
	M	P110		57	71	48
	M	P111		58	72	49
U	M	P120	1	1	4	81
0	1	P121	11	15	18	95
0	1	P122	10	14	17	94
0	1	P123	8	12	15	92
0	1	P124	7	11	14	91
0	0	P130	57	69	91	68
U	M	P131			92	69
U	M	P140	64	80	3	80
U	M	P141	63	79	2	79
U	M	P142		78	1	78
U	M	P143		77	100	77
U	M	P144		76	99	76
U	M	P145		75	98	75
	M	P150			82	59
	M	P151			81	58
	M	P152			80	57
	M	P153			79	56
	M	P154			78	55
	M	P155			77	54
	M	P156			76	53
	M	P157			75	52

記号については，表 3－2（a)下の説明を参照

表 3－3 (a)　ドライバ・キットの再定義名一覧　【78K0R/Kx3-L(64 ピン以下)用】

定義名	SFR 変数名	入出力	初期値	用途
LCD_D	P2	ｏ／ｉ	0x00	LCD データバス
LCD_DM	PM2	－	0x00	LCD データバスの入出力切り替え
LCD_WR	P4.1	ｏ	0	LCD ライト／リード指定
LCD_DI	P8.0	ｏ	0	LCD データ／コマンド指定
LCD_E	P8.1	ｏ	0	LCD ライト・リードパルス
LCD_CS1～2	P8.2～8.3	ｏ	0	LCD チップセレクト（左半分，右半分）
LCD_RESET	P15.0	ｏ	0	LCD リセット
KEYIN1～3	P12.0, P5.1, P5.2	ｉ	－	キー入力。内蔵プルアップ。
KEYIN4～5	P12.1,P12.2	Ｉ	－	キー入力。外部プルアップ。
KEYOUT1～5	P7.0～P7.4	ｏ／ｚ	0	キー・スキャン出力
KEYOUTM1～5	PM7.0 ～ PM7.4	－	0	キー・スキャンのｏ／ｚ切り替え
KEYOUTU1～5	PU7.0 ～ PM7.4	－	0	キー・スキャン内蔵プルアップ切り替え
LED0～3	P5.0，P1.2，P6.0，P6.1	ｏ	0,0,0,1	ＬＥＤ表示（キーのシフト状態表示）0 で点灯。P6 は 44pin 版には無い。
BEEP	P1.1	ｏ	0	ブザー出力
UART_TXD	P3.0	ｏ	0	シリアル出力
UART_RXD	P3.1	ｉ	－	シリアル入力。内蔵プルアップ。
UART_CTS	P3.2	ｉ	－	送信許可入力。内蔵プルアップ。
UART_RTS	P7.5	ｏ	1	送信要求出力
IR_IN	P1.0	ｉ	－	赤外受信入力。内蔵プルアップ。
IR_OUT	P1.3	ｏ	0	赤外送信出力
SUB_POWER	P14.0	Ｏ	1	赤外受信，温度センサの電源制御出力。44pin 版には無い。
AD_TMP	P15.1	ａ	－	温度センサ入力

入出力欄の記号

Ｉ：入力固定ポート，Ｏ：出力固定ポート，ａ：入出力ポートをアナログ入力に設定，

ｉ：入出力ポートを入力に設定，ｏ：入出力ポートを出力に設定，

ｚ：入出力ポートをハイインピーダンス出力に設定（実際の設定上はｉと同じ）

表 3−3 (b)　ドライバ・キットの再定義名一覧　　【78K0R/KE3〜KG3 用】

定義名	SFR 変数名	入出力	初期値	用途
LCD_D	P7	o／i	0x00	LCD データバス
LCD_DM	PM7	−	0x00	LCD データバスの入出力切り替え
LCD_WR	P2.2	o	0	LCD ライト／リード指定
LCD_DI	P2.1	o	0	LCD データ／コマンド指定
LCD_E	P2.3	o	0	LCD ライト・リードパルス
LCD_CS1〜2	P2.4〜2.5	o	0	LCD チップセレクト（左半分，右半分）
LCD_RESET	P13.0	O	0	LCD リセット
KEYIN1〜5	P12.0,	i	−	キー入力。内蔵プルアップ。
KEYOUT1〜5	P0.6, P0.5,	o／z	0	キー・スキャン出力
KEYOUTM1〜5	PM0.6,PM0.5,	−	0	キー・スキャンのｏ／ｚ切り替え
KEYOUTU1〜5	PU0.6,PU0.5,	−	0	キー・スキャン内蔵プルアップ切り替え
LED0〜3	P6.3, P6.2,	o	0,0,	ＬＥＤ表示（キーのシフト状態表示）
BEEP	P3.1	o	0	ブザー出力
UART_TXD	P1.3	o	0	シリアル出力
UART_RXD	P1.4	i	−	シリアル入力。内蔵プルアップ。
UART_CTS	P5.0	i	−	送信許可入力。内蔵プルアップ。
UART_RTS	P2.0	o	1	送信要求出力
IR_IN	P1.7	i	−	赤外受信入力。内蔵プルアップ。
IR_OUT	P1.6	o	0	赤外送信出力
SUB_POWER	P0.0	O	1	赤外受信，温度センサの電源制御出力。
AD_TMP	P2.7	a	−	温度センサ入力
SD_CD	P4.2	i	−	SD カード検出入力。内蔵プルアップ。
SD_WP	P5.1	i	−	SD 書き込み保護入力。内蔵プルアップ。
SD_CS	P1.5	o	0	SD チップ・セレクト出力
SD_CK	P0.4	o	0	SD クロック出力。OD 出力選択。
SD_SI	P0.3	i	−	SD データ入力。TTL 入力選択。
SD_SO	P0.2	o	0	SD データ出力。OD 出力選択。
AU_CK	P4.3	i	−	AU クロック入力。TTL 入力選択。
AU_SI	P4.4	i	−	AU データ入力。TTL 入力選択。
AU_SO	P4.5	o	0	AU データ出力。OD 出力選択。
AU_MCK	P14.0	o	0	AU マスタ・クロック出力。
AU_SCL	P14.2	o	1	AU 制御クロック出力。OD 出力選択。
AU_SDA	P14.3	o	1	AU 制御データ出力。OD 出力選択。
AU_FRM	P3.0	i	−	AU フレーム同期信号入力。

SD(SD カード)，AU(音声コーデック)の割り当ては参考です。KF3 ﾋL，KG3 ﾋL で AU を使うには，VDD=3.3V 以下で直接 I/F するか P43〜P45 を P10〜12 と入れ換える必要があります。

表3－3（c）　ドライバ・キットの再定義名一覧

【CQ出版「ディジタル・デザイン・テクノロジNo.8」添付基板用】

定義名	SFR変数名	入出力	初期値	用途
LCD_D	P2	o／i	0x00	LCDデータバス
LCD_DM	PM2	－	0x00	LCDデータバスの入出力切り替え
LCD_WR	P4.1	o	0	LCDライト／リード指定
LCD_DI	P8.0	o	0	LCDデータ／コマンド指定
LCD_E	P8.1	o	0	LCDライト・リードパルス
LCD_CS1～2	P8.2～8.3	o	0	LCDチップセレクト（左半分，右半分）
LCD_RESET	P15.0	o	0	LCDリセット
KEYIN1～3	P12.0, P5.1, P5.2	i	－	キー入力。内蔵プルアップ。
KEYIN4～5	P3.1,P3.2	i	－	キー入力。内蔵プルアップ。
KEYOUT1～5	P7.0～P7.2, P5.0，P7.5	o／z	0	キー・スキャン出力
KEYOUTM1～5	PM7.0～7.2, PM5.0, PM7.5	－	0	キー・スキャンのo／z切り替え
KEYOUTU1～5	PU7.0～7.2, PU5.0, PM7.5	－	0	キー・スキャン内蔵プルアップ切り替え
LED0～1	P3.0，P1.2,	o	0,0,	ＬＥＤ表示（キーのシフト状態表示） 0で点灯。
BEEP	P1.1	o	0	ブザー出力
UART_TXD	P7.3	o	0	シリアル出力
UART_RXD	P7.4	i	－	シリアル入力。内蔵プルアップ。
UART_CTS	‘0’	(i)	－	フロー制御無し（常時送信許可）
UART_RTS	DUMMY	(o)	－	フロー制御無し（常時受信可能）
IR_IN	P1.0	i	－	赤外受信入力。内蔵プルアップ。
IR_OUT	P1.3	o	0	赤外送信出力
AD_TMP	P15.1	a	－	温度センサ入力

入出力欄の記号

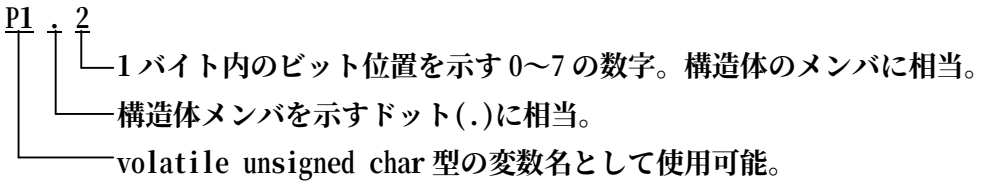
Ｉ：入力固定ポート，Ｏ：出力固定ポート，ａ：入出力ポートをアナログ入力に設定，

ｉ：入出力ポートを入力に設定，ｏ：入出力ポートを出力に設定，

ｚ：入出力ポートをハイインピーダンス出力に設定（実際の設定上はｉと同じ）

(2)複数ビットの一括操作

CC78K0R では，1 バイトの SFR 変数名の多くと__sreg 修飾変数名は，構造体変数のように扱えます（ドット付きが可能な SFR 変数名については，78K0R ユーザーズ・マニュアル 4.2.4 項参照）。



つまりドットを付けずに使うと，1 バイト単位でポート操作ができることになります。

本当の構造体ではありませんので，メンバの間接指定(->)は使えません。

volatile とは C コンパイラが関知できない方法で値が変化しうることを意味します。次のように C 言語的には無限ループになる記述でも，ポート変化によって抜けることが出来ます。

```
P1.2 = 1;
while (P1.2);
```

一般に，C コンパイラで最適化オプションを指定すると，ループ内で値が変化しない処理をループ外へ追い出すことがあります。この追い出しを禁止するという意味が volatile にあります。

(3)出力ポートの参照について

出力ポートの変数名を参照すると，その変数名に代入した値が参照されます。

一見当たり前ですが，実際の端子状態はポートの特性や外部回路の状況によっては代入値と一致しないことがあります。

例えば，’H’レベルが出力できないオープン・ドレイン端子を出力ポートにして 1 を代入した場合，実際の端子の状態は外部回路で決まります。このような場合に端子の状態を見るには，78K0R ではその端子をいったん入力にするか，別の端子を入力として接続して読みます。

(4)入力ポートへの代入

入力ポートも変数ですから値を代入することが出来ますが，代入した値は無視されます。参照する値は，代入した値ではなく実際の端子の状態になります。

(5)入出力を切り替えるときの注意

1つの端子を入力から出力へ切り替える場合，すなわち PMx.y を 1 から 0 に書き換える場合は，その直前に Px.y に出力したい値を代入する必要があります。

良い例：

```
P1.2 = 1;
PM1.2 = 0;
P1.5 = 1;
PM1.5 = 0;
```

誤動作する可能性のある例:

```
P1.2 = 1;
P1.5 = 1;
PM1.2 = 0;
PM1.5 = 0;
```

これは，P1.5 を操作したときに，P1 の中の他の入力設定（ここでは P1.2）が書き換わる可能性があるからです。

(6) 複数プログラムによる SFR 操作時の注意

1つの SFR を複数のプログラムから操作する場合は干渉に注意します。特に，割り込みの中や RTOS（リアルタイム OS）のタスクで SFR を操作する場合は，「ごくまれに誤動作する」という厄介な現象を引き起こします。

例：P1.0～P1.3 をプログラムA，P1.4 を割り込みプログラムBで操作している場合

プログラムA

```
static int i = 15;        /* P1.0～P1.3 から all '1' を出力したい場合 */
・・・
P1 ╞ i;  → オブジェクトは以下のようになります。
・・・          MOV A, P1        ①ポート P1 の内容読み込み
               OR  A, !_i       ②OR 演算
               MOV P1, A        ③ポート P1 に書き込み
```

もし②の場所で，割り込みプログラムBが P1.4 を操作しても，割り込み終了後に③のところで元の値に書き戻されてしまいます。②で割り込みがかかる確率は，一般的には低いでしょうから，誤動作が希にしか起こらず，原因究明が困難になりがちです。

一番簡単な回避方法は，ポート操作中は割り込まれるのを禁止にすればよいです。
次のように書けば（プログラム先頭に#pragma DI，#pragma EI の記述も必要），

```
DI();
P1 |= i;
EI();
```

ポート P1 更新中は割り込みがかかりません。

なお，記述によってはオブジェクトが 1 命令になる場合があり，その場合は割り込み禁止にしなくても問題が出ません。その例を次に示します。ただし生成されるオブジェクトを想定して C 言語記述を行うのは本来は邪道なのでお勧めできません。どうしても ROM サイズ圧縮や速度ネック解消をしたい場合に慎重に使用します。

・アドレス FFF00H～FFF20H のバイトアクセス可能な SFR にたいしては，
　定数の複合代入演算（加算・減算・AND・OR・XOR）は 1 命令になります。
・アドレス FFF00H～FFFFFH の SFR でビット名称が定義されていれば，
　ビット・セット（1 を代入），ビット・クリア（0 を代入）は 1 命令になります。
・それ以外の SFR（2nd SFR）でもビット名称が定義されていれば，
　ビット・セット，ビット・クリアは 1 命令になります。

逆に 1 命令にならなくて，トラブルが出やすいと思われる SFR が，後述のタイマ・アレイ・ユニット（TAU），シリアル・アレイ・ユニット（SAU）です。これらのユニットは複数のチャネルから構成されており，一部の制御ビットは複数チャネル分が 1 バイトの SFR に詰め込まれています。しかもビット操作命令が使えない（ビット・アクセスが許可されてない）SFR もあるため，他のチャネルの操作に影響されやすくなります。

邪道かもしれませんが，ビット名が定義されてない SFR でもバイトアクセス可能であれば，アセンブラでアドレス指定すれば，ビット・セット，ビット・クリアは 1 命令で可能です。例えば，タイマの出力レジスタ（TO0）であれば，

```
set1    TO0.1      →  この記述はエラーになる
set1    !1B8H.1    →  TO0 のアドレスを直接書けばエラーにならない。
```

となります。
しかし，16 ビット・アクセスしか許可してないレジスタ（SAU の SO レジスタ）もあり，この場合は DI,EI で保護するしかなさそうです。

3.3　キー入力

ここでは，ポート操作の応用として，キー入力について説明します。

3.3.1　チャタリング除去

キー入力で注意するのは，キーを押したり離したりする瞬間にチャタリングという 0,1 がばたつく現象が出る点です。

一般的なチャタリング除去として，ある一定間隔（0.01 秒間隔とか）でキーの状態を読み取って，何回か状態が一致したらオンとかオフを確定させる方法があります。

その例をリスト 3−2 に示します。このプログラムはキーマトリクスの縦 5 種類のキーを検出するもので，組み合わせて押すことにより 1〜31 のコードを表示します。キーを離すと 0 を表示します。

以下，順に説明します。

①BTIMER.h

これは筆者提供のキー入力ドライバのヘッダ・ファイルです。

キー入力ドライバを停止させる BTIMER_key_disable 関数を呼び出すためにインクルードしています。

②0.01 秒ごとに起動

プロセッサ時間を使って，0.01 秒ごとにループを抜けて処理を行います。

③getkey 関数呼び出し

getkey 関数はキー入力が確定するとコードを返してきます。入力がない状態やチャタリング除去中の期間では EOF を返してきます。

④制御に使うスタティック変数

prev_code：最後に確定したキーコードを保持。

prev_detect：前回（0.01 秒前）のキーコードを保持。

count：一致回数のカウント。

⑤現在のキー状態を読み取り

5 段のキーの状態をそれぞれ読み取り，2 進数に対応させてコード化しています。

```
/* リスト 3−2 チャタリング除去 */
#include "PORT.h"
#include <stdio.h>
#include <time.h>
#define CLOCK() ( (int) clock() / (CLOCKS_PER_SEC / 100) )
#include "BTIMER.h"                                    /* 3.2.3① */
int getkey(void);

int main (void) {
  int  c, told;
  BTIMER_key_disable();                                /* 3.2.3① */

  while (1) {
    while (told == CLOCK());                           /* 3.2.3② */
    told = CLOCK();
    if ((c = getkey()) != EOF)  printf("%d,", c);      /* 3.2.3③ */
  }
  return 0;
}

int getkey(void) {
  static char prev_code, prev_detect, count;           /* 3.2.3④ */
  char i = 0;

  /*** read key port ***/                              /* 3.2.3⑤ */
  if (!KEYIN1) i += 1;
  if (!KEYIN2) i += 2;
  if (!KEYIN3) i += 4;
  if (!KEYIN4) i += 8;
  if (!KEYIN5) i += 16;

  /*** eliminate chattering ***/
  #define nDETECTED  2
  if (i == prev_code  || i != prev_detect) {           /* 3.2.3⑥ */
    prev_detect = i;
    count = 0;
    return EOF;
  }
  else if (++count >= nDETECTED) {                     /* 3.2.3⑦ */
    prev_code = i;
    count = 0;
    return i;
  }
  return EOF;
}
```

⑥キー状態の変化を判定

現在値 i が最後に確定したコード prev_code と等しければ変化なしとします。また，直前の値 prev_detect と一致してない場合は，状態変化の 1 回目と判定します。

上記どちらの場合も，prev_detect に現在値を保存して一致カウンタ count をクリアします。まだコードは確定してないので，EOF を返します。

⑦一致回数を判定

⑥の条件にないとき，つまり最後に確定したコードとは別のコードであって，かつ直前の値とは一致している場合は一致回数をインクリメントします。

一致回数が指定値 nDETECTED 以上になったら，新しいコードが確定したと判定し，prev_code に保存します。また，一致カウンタ count をクリアし，戻り値として新たに確定したコードを返します。

3.3.2 キー入力ドライバ（KEY.c）概説

筆者提供のキー入力ドライバ（関数名 KEY_get）も基本はリスト 3−2 と同じ処理をしています。KEY_get も 0.01 秒ごとに呼び出されることを想定しています。

リスト 3−2 の getkey と異なるのは，キーが 5x5 のマトリクス構成になっている点で，リスト 3−2⑤の部分が別の関数 KEY_scan になっています。

関数 KEY_scan は，本来であれば端子 KEYOUT1〜5 を順番に 0 にして，そのつど端子 KEYIN1〜5 の状態を読み取るのですが，本ドライバはどれか 1 キーのみの検出となっているため次のように簡素化しています。

(1)端子 KEYOUT1〜5 を全て 0 にして，KEYIN1〜5 を読み取ります。
　KEYIN1〜5 の読み取りは別関数で，最初に 0 を検出した番号（1〜5）を返します。
　キーが押されてない場合は 0 を返します。

(2)上記で番号が返ってきたら，端子 KEYOUT1〜5 の順で，順に 1 を代入します
　（実際には複数キーを押したときのショートを回避するため，
　プルアップ抵抗をオンし，端子を入力モードに設定しています）。
　KEYIN1〜5 を読み取り，(1)で返ってきた番号と異なるならば，1 を代入した KEYOUTn が押されていたと判定します。

(3)KEYOUTn の n と(1)で得た番号から，次の式でキーの番号（1〜25）を生成します。
　5(n − 1) + “(1)で得た番号”

押されたキー番号（1〜25）から文字コードへの変換は，キーコード変換ドライバ（FEP.c）で行っています。またキー番号は，回路図の部品番号 SW1〜SW25 にも対応しています。

3.3.3 キーコード変換ドライバ（FEP.c）概説

本版のキーコード変換ドライバには，次の役割があります。

・KEY_get で取得した 1～25 の数値をシフト状態を加味して文字コード（半角は ASCII コード，全角はシフト JIS コード）に変換します。
・Enter キーが押されるまでは，仮のバッファに格納して，BS（バックスペース）による修正を可能としています。また画面にエコーバック（押したキーの表示）を行います。
・Enter キーが押されると，仮のバッファの内容を stdin と呼ばれる標準入力ストリームに格納します。

ここで重要なのが，C 言語の標準入力関数 getchar，gets，scanf は標準入力ストリーム stdin からデータを取得するということです。つまり本版では Enter キーが押されない間はキー入力無しとして扱われることになります。

ストリームというのはデータが時系列的に到来もしくは掃き出される機構で，実体としてはデータを格納するバッファになります。ストリームは本来，使用前にオープン手続きをし，使用が終わったらクローズ手続きをするのですが，C 言語では，標準入力ストリーム stdin，標準出力ストリーム stdout は何も手続きしなくても常時使用可能になるように定められています。

ただしマイコンには標準入出力が付いているとは限らず，CC78K0R 提供のライブラリ関数ではストリーム処理まで含まれていません。

関数呼び出しの階層構造は次のようになっています。

gets，scanf → getchar 関数を何度か呼び出して文字を取得
getchar → 標準入力ストリームを指定して getc 関数を呼び出し
getc → 指定されたストリームから 1 バイトを取得

FEP.c では逆にストリームへデータを入れるための putc 関数を使用しています。stdin だけでなく，エコーバック用のストリーム stdecho も新たに定義して使用しています。

さて，FEP.c でのコード変換は，全角かな入力に対応しているのでやや複雑です。

半角だけであれば，4 つのシフト状態（NUM,ALPA,ALPB,SYMB）とキー値（1～25）からテーブルを引けば簡単にコード変換できます。

仮名の場合は子音 1 文字目は半角で表示して，2 文字目が来た時点で仮名に変換して 1 文字目を消すという手順になっています。母音のように 1 文字目で仮名になるものもあります。2 文字の組合せは，表 1－4 の中の 76 組あります。これを順に比較したのでは効率が悪いため，C 言語の標準ライブラリにあるバイナリサーチ（bsearch）関数を使って比較しています。

3.4 LCD 制御

ここでは，秋月電子通商で取り扱い(2010 年 5 月時点)している 128x64 ドットのモノクロ LCD モジュール SG12864ASLB -GB -R01 の制御方法を説明します。

なお，端子名は PORT.h で定義されている名前で説明します。

3.4.1 LCD のリセット解除

資料には，電源が立ち上がってから 1μs 以降に LCD_RESET 端子を 0 から 1 にするように書かれています。

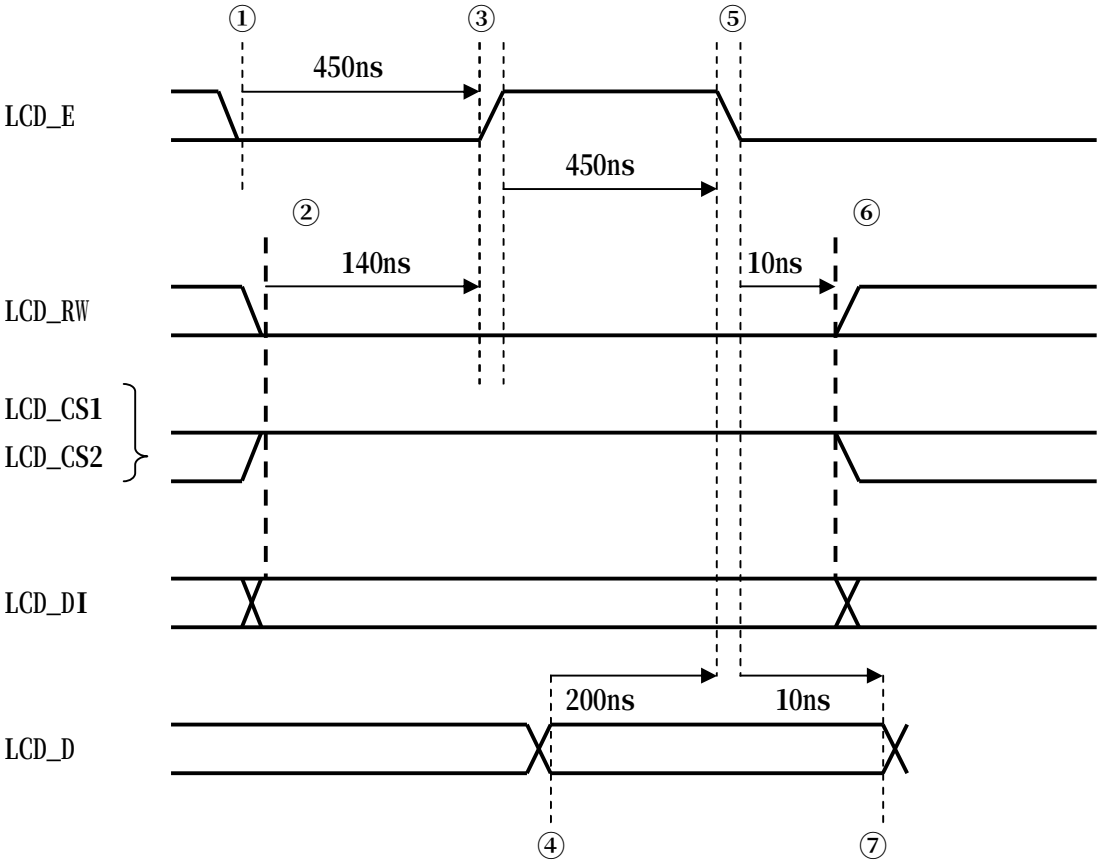
本ボードでは，ハードウエア初期設定関数(hdwinit)の中で 0 を出力していて，以降の処理では 1μs は十分経過しているので，単に

```
LCD_RESET = 1;
```

と書くだけです。

3.4.2 制御コマンドや表示データの書き込み

LCD モジュールに書きこむための波形は次のようになっています。



この波形をプログラムに落とすと次のようになります。

①LCD_E = 0;　　ただし，LCD.c では，最後必ず 0 に戻す前提なので省略。

②LCD_RW = 0;　　ただし，LCD.c では，リードをしないため常時 0 として省略。

LCD_CS1 = 1;
LCD_CS2 = 1;　　SG12864A には LCD コントローラが 2 つ入っているので，制御対象のチップを 1 にします。
表示オンのような共通コマンドでは両方を 1 にしても良いです。
表示データを書きこむ場合は，左半分の表示時に LCD_CS1，右半分の表示時に LCD_CS2 を制御します。

LCD_DI = 0 または 1;　コマンド書き込み時は 0，データ書き込み時は 1 にします。

③時間調整　　①から 450ns，②から 140ns 以上待ちます。
LCD_E = 1;

④LCD_D = コマンドまたはデータ;　　詳細後述。③の前に出力しても良いです。

⑤時間調整　　③から 450ns 以上待ちます。
LCD_E = 0;

⑥時間調整　　⑤から 10ns 以上待ちますが，動作速度より十分短いので省略。

LCD_CS1 = 0;
LCD_CS2 = 0;　　書き込みが終了したら 0 に戻します。
連続して書きこむ場合は，いちいち 0 に戻さなくても良いです。

LCD_DI = 0 または 1;　　次にコマンドまたはデータを連続して書きこむ場合です。

⑦時間調整　　⑤から 10ns 以上待ちますが，動作速度より十分短いので省略。
LCD_D = コマンドまたはデータ;　　連続して書きこむ場合です。

LCD.c では，③，⑤の時間調整に time.c で定義している wait_clock 関数を使用しています。指定値は，0.5[μs]にしてありますが，本版の wait_clock 関数は 1μs 以下の指定をしても 1μs 経過します。

また，LCD へのコマンド書き込みの後，コマンド実行が終わったかどうかをステータス読み込みで確認する必要がありますが，LCD.c では手抜きをして一定時間待つようにしています。待ち時間は wait_clock 関数で指定値 6[μs]にしてあります。5[μs]でも動きますが，③，⑤の時間が将来設定通り 0.5[μs]なる場合に備えて長くしています。

3.4.3　ステータスや表示データの読み込み

LCD.c には入れていませんが，書き込みと同様に波形をプログラムに落とせば読み込みが可能です。

・LCD_D を入力ポートに設定するには，LCD_DM = 0xff; とします。

・データ読み込みの場合，１回目はゴミデータなので破棄します。

3.4.4　コマンドの種類

下記の１バイトの制御コマンドがあります。

0x3f：　表示オン。リセット後に１回必要です。

0x3e：　表示オフ。電源オフの前に表示オフにするとゴミ表示が残りません。

0xc0 + n：　表示開始ライン位置。0xc0 を設定します。本モジュールでは省略可能なはずですが，筆者環境で１度だけ表示乱れがリセットしても直らず，これを指定したら直りました。

0xb8 + n：　表示メモリのページ・アドレス指定。
縦 8 ドット単位で n = 0～7 のアドレスが指定できます。
画面上では，0 が一番上の段，7 が一番下の段になります。

0x40 + n：　表示メモリのカラム・アドレス指定。
横１ドット単位で n = 0～63 のアドレスが指定できます。
LCD モジュールの左右でコントローラが別れており，左半分は LCD_CS1，右半分は，LCD_CS2 で制御します。
左右それぞれの領域の左端が 0，右端が 63 になります。

表示メモリのカラム・アドレスは書き込むたびに自動的に+1 されるので，カラム方向に連続して書き込む場合はカラム・アドレスは１回だけ設定すれば良いです。

3.4.5　表示データ

画面上の縦 8 ドットが１バイトの書き込みデータに対応しています。

データの LSB（=ビット位置 0）が画面上側になります。

文字を表示するには，文字コードに対応したビットパターン(フォント)を書き込みます。

フォントは，FONT.c に格納しています。LCD.c は，文字コードに対応したフォント・データ位置を取得する関数（FONT.c で提供されている関数）を使って文字表示を行っています。

3.4.6　市松模様の表示例

リスト 3−3 に市松模様の表示例を示します。

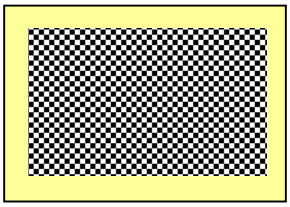
```
/* リスト 3−3 市松模様の表示 */
#include "PORT.h"
#include <stdio.h>
#include <time.h>
#include "BTIMER.h"
#include "LCD.h"                          /* ① */

int main(void) {
    int i, j;
    BTIMER_key_disable();
    BTIMER_lcd_disable();                 /* ② */

    LCD_CS1 = 1;                          /* ③ */
    LCD_CS2 = 1;
    for (i = 0; i < 8; i++) {
        LCD_DI = 0;                                 /* ④ */
        LCD_D = (unsigned char) (0xB8 + i);         /* ⑤ */
        LCD_write_pulse();
        LCD_D = 0x40;                               /* ⑥ */
        LCD_write_pulse();

        LCD_DI = 1;                           /* ⑦ */
        for (j = 0; j < 32; j++) {
                LCD_D = 0x55;                 /* ⑧ */
                LCD_write_pulse();
                LCD_D = 0xAA;
                LCD_write_pulse();
        }
    }
    LCD_CS1 = 0;                              /* ⑨ */
    LCD_CS2 = 0;
    return 0;
}
```

実行結果 3−3



①書き込み波形をつくる LCD_write_pulse 関数を使用するためにインクルードします。
②LCD ドライバの動作を停止します。キーもエコー表示があるので停止します。
③左右の領域に同じデータを書きこむため，両方 1 にしています。
④アドレス設定コマンドの書き込みを指定します。
⑤ページ・アドレスを設定します。
⑥カラム・アドレスを設定します。0 から書き始めます。
⑦データ書き込みを指定します。
⑧表示データ（市松模様）を書きこみます。
⑨コントローラ制御を終了します。

3.4.7 LCD ドライバ（LCD.c）概説

本版の LCD ドライバ（LCD_display 関数）には，次の役割があります。

・表示位置の管理とカーソル表示
・改行，タブ，BS（後退末梢）処理
・ストリームからデータを取り出して半角，全角文字の表示

本ドライバでは，次に書き込むべき文字の位置を半角単位（横 8x 縦 16 ドット）で管理しています。書き込み位置が分るように縦棒カーソルをブリンクさせています。

ストリーム（3.3.3 項参照）は引数で指定可能で，標準出力ストリーム stdout やエコーバック用ストリーム stdecho から文字コードを取り出して，文字の表示や改行などの制御を行います。

改行（\n）は，現在位置から右端までをクリアして次の行に移動します。タブ（\t）は半角 4 文字単位の位置まで移動します。BS（\b）は，基本的には半角 1 文字を後退して消します。ただしキーエコーの場合，Enter キーを押す前は，半角／全角とも 1 文字分が消えます。これは FEP.c で全角の場合に 2 回 BS を stdecho に出力しているためです。

文字表示は，半角と全角で異なります。半角と判断すると（手抜きして 0x7f 以下を半角と判定），get_font_ascii 関数（FONT.c 内）でフォント・データ位置を取得します。全角 1 バイト目と判定するといったん保存し，次のバイトがストリームから来たら get_font_kanji 関数（FONT.c 内）でフォント・データ位置を取得します。

半角フォント・データの並びは上 7 バイト，下 7 バイトの順です。1 ドット分の空白を手前に付加して横 8 ドットとして書きこみます。

全角フォントは左右半分づつを半角の並びと同じ構成にしています。これは，文字の途中で画面の左端に達したときに，改行して残り半分を表示するためです。

3.4.8 フォント・ドライバ（FONT.c）概説

半角フォントは，配列 __far const char font_ascii[95][14]に格納しています。文字コード 0x20～0x7E の 95 文字に対応しているので，フォント位置取得関数 get_font_ascii は，各文字の先頭位置として font_ascii[文字コード -0x20]を返します（戻り値は，__far 修飾した char 型ポインタ）。コードが範囲外の場合はダミーデータが格納された領域のポインタを返します。

全角フォントは，漢字フォント・データ生成ユーティリティ Font_78K0R を使用してデータを作っています。この場合，次の 2 つの配列が自動的に生成されます。

・漢字のコード一覧　const unsigned short font_kanji_index[FONT_KANJI_NUM]
　__near 領域に配置されます（スモールまたはミディアム）。
　FONT_KANJI_NUM は，組み込まれている漢字数です。

・フォント・データ　__far const char font_kanji1[][28]
　明示してある通り__far 領域に配置されます。漢字コード順に 1 文字 28 バイトのデータが連続して並んでいます。

フォント位置取得関数 get_font_kanji は，上の 2 つから次のように処理します。

①探したい漢字のコードが配列 font_kanji_index の何番目に格納されているかを，C 言語の標準ライブラリにあるバイナリサーチ（bsearch）関数を使って調べます。
　bsearch 関数は次のように使います。
　bsearch(探したい漢字コードを格納した変数へのポインタ，
　　font_kanji_index，FONT_KANJI_NUM，sizeof(unsigned short)，
　　大小比較関数へのポインタ)
　ここで，大小比較関数は，次のように定義しておきます。

```
int bsearch_cmp(const void* a, const void* b) {
    return ( *(unsigned int*)a  - *(unsigned int*)b );
}
```

　コードが見つからない場合はダミーデータが格納された領域のポインタを返します。

②bsearch の結果はポインタなので，整数型にキャストして何番目の要素か計算します。
　番号=((int)bsearch の結果　-(int) font_kanji_index)/ sizeof(unsigned short)

③番号が分ったら，配列 font_kanji1 上の位置を戻り値とします。
　戻り値 = font_kanji1[番号]

3.5 定期的な処理（インターバルタイマと割り込み処理）

キー入力のように一定時間間隔で処理したい場合には，一定時間間隔を作り出すインターバル・タイマと，割り込み処理機能を使用します。

3.5.1 タイマ概要

78K0R には，タイマ・アレイ・ユニット（TAU）と呼ばれるタイマの集合ユニットが内蔵されており，一定時間を作り出したり，方形波生成，パルス幅測定などの機能として使用できます。

品種により TAU が複数搭載されているものもありますが，ここでは基本となる TAU0 について説明します。

TAU0 は，8 本のチャネルと呼ばれるタイマから構成されています。個々のチャネルは単独で簡単なタイマ機能（インターバル・タイマなど）を実現できますが，PWM（パルス幅変調）波形を作るような場合は 2 チャンネル以上を使用します。

本書では 8 チャネルを表 3−4 のように割り当てています。

TAU を使うには，個々のチャネル設定の前に全体の設定が必要です。

```
TAU0EN = 1;                    ・・・TAU0 を有効にします。
TPS0 = PRS01 << 4 | PRS00;     ・・・個々のチャネルで使うクロック設定
```

ここで，個々のチャネルでは 2 種類のクロックしか選択できず，それぞれのクロック周波数を PRS01，PRS00 で指定します。設定値は CPU クロックの分周比コードです。

0:1/1，1:1/2，2:1/4，3:1/8，4:1/16，5:1/32，・・・，15:1/32768

本書では，TPS0 = 0x40;　（1/16 と 1/1）を設定しています。

表 3−4　ドライバ・キットにおける TAU0 のチャネル割当

チャネル		用途	対応端子の使用
～KE3 ㇾ	KE3～		
0	4	基本周期間隔（0.01 秒）の生成	−
1	5	短い待ち時間（1ms 以下）の作成	−
2	2	赤外線受信のパルス間隔測定	赤外線受信入力
3	3	メロディ用方形波の生成	ブザー出力
4，5	0，1	LED 調光用 PWM 波形出力または赤外線送信キャリアの生成	LED 調光出力または赤外線送信出力
6	6	赤外線送信間隔の生成	−
7	7	予備	−

3.5.2 インターバル・タイマとしての使用

インターバル・タイマは，一定時間周期で割り込み信号を発生します。
以下，設定方法と動作を説明します。n はタイマのチャネル番号です。

(1)モード設定（クロック選択）

TMR0n = 0x0000 または 0x8000;　分周クロック PRS01 を使用する場合に 0x8000 を選択します。

(2)周期設定

TDR0n = 周期数 - 1;
ここで，周期数 = 間隔時間 * 選択したクロック周波数，です。
例えば，PRS01 を使用して 0.01 秒間隔を作るには，
周期数 = 0.01 * ($20x10^6$ / 16) = 12500
　　└ HDWINIT.h の CPU_FREQUENCY に定義値あります

となります。TDR0n には－１した値を設定する点を注意します。

(3)動作開始

TS0L.n = 1;

以降カウンタは次の動作をします。
①カウンタ TCR0n に周期指定 TDR0n の値を代入。
②TMR0n で指定された PRS00 または PRS01 のクロック周期で，TCR0n を減算
③TCR0n が 0 になったら，割り込み信号 INTTM0n を出力し，
　割り込みフラグ TMIF0n を 1 に設定し，①へ戻る。

動作中は，カウンタ TCR0n の途中経過を参照できます。
割り込みフラグ TMIF0n が 1 かどうかで，1 周期経過したかどうかが判ります。
また TMIF0n = 0; とすることにより，このフラグをクリアできます。
割り込み信号 INTTM0n に対応した関数を自動起動する方法は，3.5.3 項を参照してください。

(4)動作を停止する場合

TT0L.n = 1;

3.5.3 割り込み処理

割り込み信号によって，自動的に所定の関数を実行させることが可能です。
これを割込処理と言います。
割り込み処理が発生すると，今まで実行していたプログラムはいったん中断します。
割り込み処理が終わると中断した位置から再開します。

(1)割り込み処理機能の許可

インターバル・タイマによる割り込み処理機能を有効／無効にするには,

```
TMMK0n = 0;      /* こっちが有効 */
TMMK0n = 1;      /* 無効に戻す（デフォルト） */
```

とします。

(2)割り込まれることの禁止／許可

main 関数は割り込まれることを許可していますが，プログラムの状況によっては，許可できない場合があります。

割り込まれるのを禁止するには，まずプログラムの始めで,

```
#pragma DI
#pragma EI
```

を記述し，割り込まれては困る部分を DI()関数と EI()関数の間に記述します。

例：

```
DI();
LCD_CS1=1;
LCD_D = 0x55;
LCD_CS1=0;
EI();
```

ただし，割り込みの種類によっては，緊急に処置が必要な場合もありますので，長く禁止することは好ましくありません。上記のように，LCD 制御の干渉防止が目的ならば，LCD 制御部分を 1 本化することを先に検討します。

また標準ライブラリ関数の中には，割り込み禁止時間が設定されている物があります（例えば long の掛算）。これについてはコンパイラのヘルプを参照してください。

(3)割り込み処理関数の定義

割り込み要求時に自動的に実行する関数を定義するには，まずプログラムの始めに次の記述を行います。

```
#pragma interrupt INTTM0n BTIMER_INT rb1
                    ①       ②          ③
```

①割り込み要因名

タイマ割り込みの場合，チャネル番号に応じて INTTM0n という名前になります。

②自動実行する関数名

③レジスタのバンク指定

ここで，レジスタとはプログラム実行に必要な途中経過を格納する変数です。

一般的には，割り込み処理にはいる前にレジスタの値を保存し，割り込みから戻る時に復元しないと，元のプログラム処理を継続できません。

しかし 78K0R には 4 つのレジスタ群（バンク）があり，退避する代わりにレジスタ・バンクを切り換えるだけで済ますことが出来ます。

main 関数はバンク 0（rb0）を使用するので，一般的な割り込みではバンク 1（rb1）を使用します。割り込み処理の中からさらに割り込みをかける場合や，バンクの中に static 変数を定義して高速処理するような場合はバンク 2～3（rb2～rb3）も使用します。

次に関数の本体を定義します。引数も戻り値も無しとします。

```
__interrupt void BTIMER_INT(void) {
```

- `BTIMER_INT` ― 関数名。#pragma interrupt の指定と一致させます。
- `__interrupt` ― #pragma interrupt の記述と同じファイルにあれば省略可能です。別ファイルに書く場合は必須です。

なお割り込み関数のプログラムが実行される直前に，次の処理が入ります。

①ハードウエア的に割り込みフラグがクリアされ，割り込まれるのを禁止する。
（TMIF0n = 0; DI(); と等価な動作）

②レジスタ・バンクを切り替え。

③コンパイラや標準ライブラリ関数が使用する RAM 領域を退避（記述内容依存）。

終了時には次の処理が入ります。

①コンパイラや標準ライブラリ関数が使用する RAM 領域を復元（退避していた場合）。

②ハードウエア的にレジスタ・バンクを元に戻し，割り込まれるのを許可する。

CC78K0R の場合，上述の RAM 領域の退避／復元のオーバーヘッドが大きくなりやすい点に注意が必要です。高速応答が必要な場合は次の点に注意します。

・関数を呼ぶ処理があるとオーバーヘッドが大きくなる。
・特にライブラリ関数は退避領域が増える傾向にある。
・int の比較というような一般的演算でもランタイム・ライブラリ関数が呼ばれる。
　なるべく，unsigned char，char，unsigned int 型演算を使う。
・同じ理由でコンパイル・オプションで char の int 拡張をしない指定をする（このオプションはソースファイル単位で指定が変更できる)。

(4)多重割り込み

多重割り込みとは，割り込み処理中に別の割り込み処理を行うことです。
多重割り込みを許可するには，

```
EI();
```

を記述します。
ただし，78K0R では優先度の高い割り込みしか許可されません。
つまり多重割り込みするには，割り込み要因の優先度を高く設定する必要があります。
優先度の設定は，次のようにします。

xxxPR1x = 1，xxxPR0x = 1;　優先度 3(最低，デフォルト状態)
xxxPR1x = 1，xxxPR0x = 0;　優先度 2
xxxPR1x = 0，xxxPR0x = 1;　優先度 1
xxxPR1x = 0，xxxPR0x = 0;　優先度 0(最優先)

ここで，xxx は要因に応じた名前で INTTMmn の場合は，TMPR1mn，TMPR0mn となります。
優先度 0 は，例外として EI 状態なら常に割り込み可能です。

筆者提供環境での割り込み要因と優先度，使用レジスタ・バンクは次のとおりです。

要因名		割り込み要因	割り込み優先度	レジスタバンク
～KE3 -L	KE3～			
INTTM00	INTTM04	定周期処理（0.01 秒間隔）	3	1
INTRTC	INTRTC	時計表示（0.5s(min)，［予約］）		
INTTM02	INTTM02	赤外線受信（1.2ms(min)程度）	2	2
INTTM06	INTTM06	赤外線送信（0.6ms(min)程度）		
INTSR1*	INTSR3	UART 受信（0.5ms(min)程度@19.2kbps）		
INTST1*	INTST3	UART 送信（0.5ms（min）程度@19.2kbps）		
－	別途	音声コーデック用［予約］	1	3

*注：DDT8 は，INTSR0 および INTST0

3.5.4 定周期ドライバ（BTIMER.c）概説

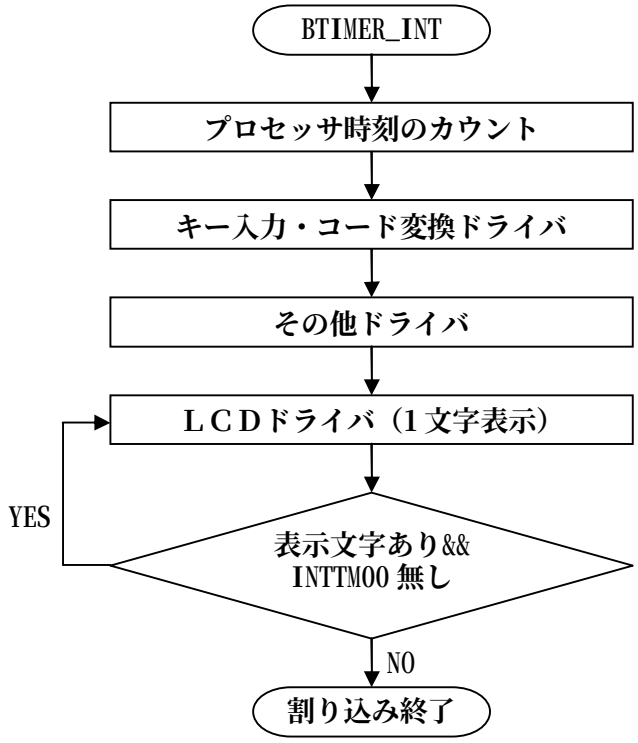
定周期タイマ初期設定（BTIMER_open 関数）と，0.01 秒ごとの割り込みで起動する各種定周期処理（BTIMER_INT 関数）から構成しています。

ドライバ類も C 言語で記述しているので，本来は静的変数の初期化が終わった後，すなわち main 関数に入ってからドライバの呼び出しを行うのが筋です。ここでは，C 言語学習を容易にするために，あえて main ではなく hdwinit 関数で BTIMER_open を呼び出し，BTIMER_INT 関数が起動するようにして，各種ドライバを自動的に呼び出しています。

静的変数の初期化前にドライバを呼び出している関係で，一部の変数を先に初期化したり，初期値になっているかチェックするなどの変則的な処理が BTIMER_INT 関数やドライバ内部の一部に入っています。

BTIMER_INT 関数の概要フローは次のとおりです。

最後に文字表示を行いますが，表示文字が多くて次の 0.01 秒割り込みが入ったらいったん関数を終了します。関数を終了するとすぐに割り込みにより BTIMER_INT 関数を最初から実行します。すなわち各種ドライバを 0.01 秒に 1 回は必ず呼び出します。



3.6　メロディ制御（方形波出力）

ブザーからメロディを出す方法について説明します。まずはサンプルのメロディを聴いてください。その後で，音の出し方の基本とメロディ・データについて順に説明します。

3.6.1　サンプル・メロディの演奏

リスト 3−5A にメロディ演奏のプログラム例を示します。テンポと音程をキー入力によって変更できます。

```
/* リスト 3−6A メロディ演奏 */
#include <stdio.h>
#include "BEEP.h"                                   /*①*/
#include "BEEP.inc"                                 /*②*/

int main(void) {
    char line[33];
    int tempo, ichou;

    BEEP_play(whistle_polka, 0, 85, 0);             /*③*/

    while(1) {
        while (gets(line) == NULL);
        sscanf(line, "%d%d", &tempo, &ichou);
        if (!tempo) break;
        if (tempo > 10 && tempo < 250 &&
                ichou > -36 && ichou < 24) {
            BEEP_stop();                            /*④*/
            while(BEEP_refer());                    /*⑤*/
            BEEP_play(whistle_polka, 0, tempo, ichou);
        }
    }
    BEEP_stop();
    return 0;
}
```

実行結果 3−6A
(赤字は入力例)

```
100 12⊿
70 -12⊿
```

①ブザーを鳴らしたり，メロディ演奏するには，BEEP.h をインクルードします。
②BEEP.inc にサンプルのメロディ・データが格納されています。
③BEEP_play 関数はメロディ演奏の開始を要求します。

引数 1 ： ONPU *p;	音符の配列へのポインタ
引数 2 ： int offset;	配列 p の途中から始める場合に要素番号を指定。
引数 3 ： int tempo;	演奏速度を 1 分間の四分音符の個数で指定。
引数 4 ： int ichou;	音程を変更する場合に，半音単位で指定。
戻り値: int;	成功時は 0 以上，失敗時(演奏中)は EOF

実際にメロディ・データを解釈して音を出すのは BEEP_check 関数です。この関数は 0.01 秒周期で呼び出す必要があり，定周期処理（BTIMER.h）から呼び出しています。
④BEEP_stop 関数はメロディ演奏の停止を要求します。
⑤BEEP_stop から出した停止要求を BEEP_check で受け付けて停止したかを確認するのが BEEP_refer 関数（マクロ定義）です。

3.6.2 方形波の生成

圧電ブザーから音を出すには，音程に応じた周波数のクロック信号を供給します。
スピーカ，ヘッドホンの場合も同様ですが，これらを駆動するにはアンプが必要です。
クロック信号として最も単純な方形波はタイマ・アレイ・ユニット（TAU）の 1 チャネルだけを使用して生成できます。設定手順は次のとおりです（n はチャネル番号）。

(1)TAU 共通設定（3.5.1 参照）

(2)モード設定（クロック選択）

TMR0n = 0x8000 または 0x0000;　分周クロック PRS01 を使用する場合に 0x8000 を選択します。

(3)周期設定

TDR0n = 周期数 - 1;
ここで，周期数 = 選択したクロック周波数 / 生成する方形波の周波数 / 2
例えば，PRS01(本書では 1/16 に設定)を使用して 1,000 Hz の信号を作るには，
周期数 = ($\underline{20x10^6}$ / 16) / 1000 / 2 = 625
　　└ HDWINIT.h の CPU_FREQUENCY に定義値あります

となります。TDR0n には－1 した値を設定する点を注意します。

(4)動作開始

端子 TOOn を出力ポートに設定し 0 を出力。ただし初期設定済なら省略可能。

```
TOLOL &= ~(1 << n);        リセット初期値のままなら省略可能。注 1。
TOOL &= ~(1 << n);         初期レベル不定か,使用後 0 に戻すなら省略可能。注 1。
TOE.n = 1;                 必須
TSOL.n = 1;                必須。動作開始。
```

注 1：記述する場合は DI,EI で挟む SFR 更新保護を推奨（3.2.2 (6)参照）

以降カウンタは次の動作をします。

①カウンタ TCROn に周期指定 TDROn の値を代入。

②TMROn で指定された PRSOO または PRSO1 のクロック周期で，TCROn を減算

③TCROn が 0 になったら，端子 TOOn の出力レベルを反転し，①へ戻る。

(5)動作を停止する場合

```
TTOL.n = 1;
```

これで方形波の出力は停止しますが，出力レベルは最後の状態を保ったままです。
圧電ブザーは，直流電圧をかけたままにすると良くないので，一時的な停止ではなく
完全停止する場合は，次の処置によって 0 出力にします。

```
TOE.n = 0;
TOOL &= ~(1 << n);         DI,EI で挟む SFR 更新保護を推奨（3.2.2 (6)参照）
```

3.6.3 ブザー出力の簡単な例

単純なブザー出力の例をリスト 3－6B に示します。
キー入力で周期数（= TDROn + 1）を指定すると，それに対応する音が出ます。
0 を入力すると音が止まります。

①atoi 関数

文字列を数値に変換する標準ライブラリ関数です。
使用するには，stdlib.h をインクルードします。

②BEEP_timer_start 関数

タイマ設定を行って音を出す関数です。引数は，周期数 - 1（TDR 設定値）です。
メロディ制御ドライバ（BEEP.c）に含まれています。
関数内で 3.6.2 (1)～(4)で説明した設定を行っています。

③BEEP_timer_stop 関数

タイマ設定を行って音を止める関数です。

メロディ制御ドライバ（BEEP.c）に含まれています。

関数内で 3.6.2 (5)で説明した設定を行っています。

```
/* リスト 3－6B 単純なブザー出力 */
#include <stdio.h>
#include <stdlib.h>      /* atoi 関数用 */
#include "BEEP.h"

int main(void) {
    char line[33];
    int i;

    while(1) {
        while (gets(line) == NULL);
        i = atoi(line);                          /*①*/
        if (i) BEEP_timer_start(i  - 1);         /*②*/
        else   BEEP_timer_stop();                /*③*/
    }
    return 0;
}
```

実行結果 3－6B

(赤字は入力例)

```
600⊿
1200⊿
300⊿
0⊿
```

3.6.4　音階周波数

音階に対応した音を出したい場合は，音階周波数から周期数を計算します。

ヨーロッパ音楽の平均律 12 音階(12 平均律)では，基準点（ラ，440Hz）からの音階差（半音単位）m より，ある音階の周波数を次のように計算できます。

$$440 \times 2^{m/12} \quad [Hz]$$

表 3－6A　基準点から 1 オクターブ分の m 値と音階名

m	音階名	m	音階名
0	ラ	6	レ＃・ミ♭
1	ラ＃・シ♭	7	ミ
2	シ	8	ファ
3	ド	9	ファ＃・ソ♭
4	ド＃・レ♭	10	ソ
5	レ	11	ソ＃・ラ♭

ちなみにピアノの範囲は m = -48～+39　です。

マイコンから音階を出す場合,いちいち 2 のべき乗計算していたのでは効率が悪いので，12 音階分の周波数から周期数を計算し，あとは 1 オクターブ違う音は 2 倍あるいは半分にします。

表 3－6B に，カウントクロック 1.25MHz（20MHz/16）における 440/32 [Hz]時の周期数を示します。440 Hz を出したいときは，この値を 32 で割るか，5 ビット右シフトします。より正確には丸めの計算をします。TDR0n の設定値は周期数 - 1 です。

表 3－6B　クロック・ソース 1.25MHz，13.75Hz から 1 オクターブ分の周期数

音階名	周期数	音階名	周期数
ラ	45455	レ＃・ミ♭	32141
ラ＃・シ♭	42903	ミ	30337
シ	40495	ファ	28635
ド	38223	ファ＃・ソ♭	27027
ド＃・レ♭	36077	ソ	25511
レ	34052	ソ＃・ラ♭	24079

3.6.5 メロディ制御ドライバ（BEEP.c）概説

(1)機能

指定されたメロディ・データに基づいて自動演奏します。

(2)使用方法

0.01 秒周期で次の関数を呼び出します。

void BEEP_check(void);

演奏の開始と停止は 3.6.1 項で説明した BEEP_play 関数，BEEP_stop 関数を使用します。

(3)メロディ・データの構造

リスト 3－6C に簡単なメロディ例（ソラシ－ラソ,ソラシラソラ－）を示します。

１つの音符を ONPU 型データで表し，音符を並べた配列で曲を構成します。

音符だけではなく，リピートなどの制御コードも並べることができます。

音符やリピートを簡単に記述できるマクロが BEEP_ONPU.h で定義されています。

以下，マクロを使ったリスト 3－6C の記述について説明します。

①{ mSET(カウンタ ID，回数) }：反復回数を設定します。カウンタ ID = 0～3 の 4 種類まで独立にカウント制御できます。

②{ 音程，音の長さ }：音程を DO，RE，MI，・・・，音の長さを T4（四分音符），T8（八分音符）のように指定します。具体的には BEEP_ONPU.h を見てください。

また，属性を追加して {音程，音の長さ，属性}　の形式でも指定できます。属性は，音の長さに占める最後の無音期間を意味し，タイやスラーのように音をつなげる場合は Q0，スタッカートのように切りたければ Q40 などを指定します。属性を省略したときは，char 型変数 BEEP_quiet に格納した値（初期値 4）ｘ0.01 秒が無音期間となります。

③{ mQF(休止の長さ) }：休符を意味します。長さコードは音の長さと同じです。

④{ mSKP(カウンタ ID，進み先) }：設定された回数までは次の音符に進みます。回数を越えると，進み先（音符配列の要素番号で指定）へ移動します。

⑤{ mREP(カウンタ ID，戻り先) }：設定された回数までは戻り先（音符配列の要素番号で指定）へ移動します。回数を越えると次の音符に進みます。

⑥{ mFINISH }：演奏を終了します。再演奏する場合は { mREPLAY } と指定します。

```
/* リスト3−6C メロディ・データの例 */
#include <stdio.h>
#include "BEEP.h"
#include "BEEP_ONPU.h"

const ONPU melody[] = {
/* ソラシ−ラソ,ソラシラソラ− */
/*00*/ {mSET(0, 1) }, {mSET(1, 1)},              /*①*/
/*02*/ { SO, T8 }, { LA, T8 }, { SI, T4T},       /*②*/
/*05*/ { LA, T8 }, { SO, T8 }, { mQF(T8)},       /*③*/
/*08*/ { SO, T8 }, { LA, T8 }, { SI, T8 },
/*11*/ { LA, T8 }, { SO, T8 }, { mSKP(1, 17)},   /*④*/
/*14*/ { LA, T4T}, { mQF(T8)}, { mREP(0, 2) },   /*⑤*/
/*17*/ { LA, T2T}, { mFINISH}                    /*⑥*/
};

int main(void) {
        char line[33];
        int tempo, ichou;

        while(1) {
                while (gets(line) == NULL);
                sscanf(line, "%d%d", &tempo, &ichou);
                if (!tempo) break;
                BEEP_play(melody, 0, tempo, ichou);
        }
        BEEP_stop();
        return 0;
}
```

実行結果 3−6C
(赤字は入力例)

```
120 0⊿
```

3.7　LED 調光（PWM 制御）

LED の明るさを変えるには，PWM（パルス幅変調）波形で LED を駆動します。まずは調光の例を示した後，PWM 波形の生成方法を説明します。

3.7.1　明るさの制御例

周期的に明るくなったり暗くなったりを繰り返す例をリスト 3－7C に示します。
キー入力により，明滅周期などのパラメータを入力できます。

```
/* リスト 3－7A LED 調光 */
#pragma HALT                                    /*①*/
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <PWM.h>                                /*②*/
#define CLOCK() \
    ((unsigned int) clock() / (CLOCKS_PER_SEC / 100))
#define PAI     3.14

int main(void) {
    char line[33];
    int  x, peri = 200, center = 50, level = 50;
    float z, gamma = 2.5;

    while (1) {
        if (gets(line) != NULL) {
            sscanf(line, "%d%d%d%f",
                    &peri, &center, &level, &gamma); /*③*/
            if (peri == 0) break;
        }
        z = level * sin(CLOCK() % peri * 2 * PAI / peri);
        z = pow((center + z) / 100, gamma) * 10000;
        if ((x = (int) z) < 0) x = 0;
        PWM_start(x);                           /*④*/
        HALT();                                 /*⑤*/
    }

    PWM_stop();                                 /*⑥*/
    printf("closed\n");
    return 0;
}
```

実行結果 3－7A
(赤字は入力例)

```
800 40 30 2.5⊿
```

①#pragma HALT は，HALT 関数を使うという指定です。
HALT 関数は，なんらかの割り込みがかかるまで，マイコンを待機状態にします。
ここでは，定周期（0.01 秒）割り込みしか発生しないため，時間待ち代わりに使っています。ループで判定するより消費電流を押さえることができます。

②PWM 制御関数を呼び出すためのヘッダです。

③明滅制御のためのパラメータです。次の４つを指定します。
周期[0.01s 単位]，センター値[%]，振幅値[%]，ガンマ補正値
明滅は，sin 関数を使って，センター値 + 振幅値 * sin(t) で計算しています。従って，センター値 50，振幅値 50 を指定すると，0～100%の範囲で明るさが変わります。
ただし，人間の目の感覚は対数的なので，このままでは明るさが直線的に変わったように感じません。そこで，べき乗関数（pow 関数）を使って補正を行います。筆者環境では，ガンマ補正値 2.5 ぐらいが丁度良い感じでした。
なお，0 を入力すると終了します。

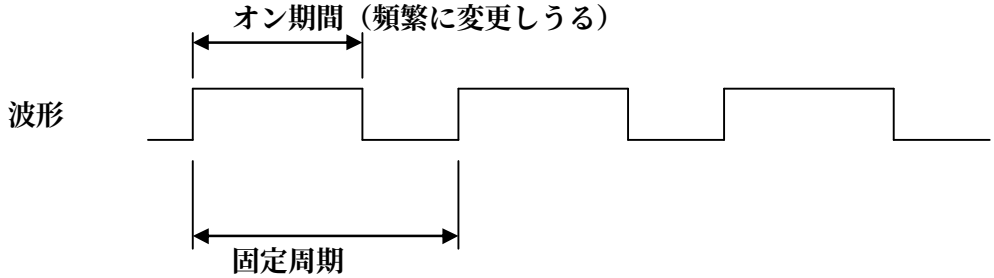
④PWM_start 関数は，LED 調光ドライバ（PWM.c）の関数で，PWM 波形を生成します。
引数は，LED の明るさ 0～100%を 100 倍した数値（0～10,000）を指定します。

⑤明るさの変化は 0.01 秒ごとで十分なので，HALT 関数で待機します。

⑥PWM_stop 関数は，LED 調光ドライバの関数で，PWM 波形生成を停止します。

3.7.2 PWM 波形の生成

PWM 波形とは，一定周期の内の何%をオン期間にするかを頻繁に変更できる波形です。



PWM 波形を生成するためには，タイマ・アレイ・ユニット（TAU）の 2 つのチャネルを使用します。固定周期を作る方をマスタ・チャネル（以下チャネル番号 m），オン期間を作るチャネルをスレーブ・チャネル（以下チャネル番号 s）と呼びます。

設定手順は次のとおりです。

(1)TAU 共通設定（3.5.1 参照）

(2)モード設定（クロック選択）

```
TMR0m = 0x0801;
TMR0s = 0x0409;
```

ただし分周クロック PRS01 を使用する場合には，それぞれ 0x8000 を加算します。

(3)周期設定

```
TDR0m = 周期数 - 1;
```

ここで，周期数 = 選択したクロック周波数 / 固定周期の周波数

例えば，PRS01(本書では 1/16 に設定)を使用して 280 Hz の周期を作るには，

周期数 = ($\underline{20x10^6}$ / 16) / 280 = 4464

HDWINIT.h の CPU_FREQUENCY に定義値あります

となります。TDR0m には－１した値を設定する点を注意します。

```
TDR0s = マスタの周期数 * オン期間の比率(%)
```

0%なら 0，100%ならマスタの周期数を設定します。

TDR0s は随時変化させる前提なので，最初は 0 でも構いません。

(4)動作開始

端子 T00s を出力ポートに設定し 0 を出力。ただし初期設定済なら省略可能。

```
TOM0 |= 1 << s;          必須。注 1。

TOL0L &= ~(1 << s);      ‘H’でオンの場合。初期値のままなら省略可能。注 1。
（TOL0L |= 1 << s;       ‘L’でオンの場合。）注 1。

TOOL &= ~(1 << s);       ‘H’でオンの場合。注 1。
（TOOL |= 1 << s;        ‘L’でオンの場合。）注 1。

TOE.s = 1;                    必須
TS0L |= 1 << m | 1 << s;      必須。動作開始。注 1。
```

以降カウンタは次の動作をします。

①カウンタ TCR0m に TDR0m の値を代入。
　カウンタ TCR0s に TDR0s の値を代入して端子 TO0s からオン・レベルを出力。
　ただし 0%時はオンになりません。
②指定された PRS00 または PRS01 のクロック周期で，TCR0m，TCR0s を減算。
③TCR0s が 0 になったら，端子 TO0s からオフ・レベルを出力。
　以降 TCR0s は 0 のままです。
　なお 100%時はオフになりません。
④TCR0m0 になったら，①へ戻る。

(5)動作を停止する場合

```
TT0L |= 1 << m |1 << s;          注 1。
TOE.s = 0;
TO0L &= ~(1 << s);               注 1。
```

注 1：DI,EI で挟む SFR 更新保護を推奨（3.2.2 (6)参照）

3.7.3 LED 調光ドライバ（PWM.c）概説

PWM.c には，3.7.1 で説明した PWM_start 関数と PWM_stop 関数だけ入っています。

・PWM_start 関数
PWM が動作してない場合は，3.7.2 (1)～(4)の設定を行います。
すでに動作中であれば，TDR0s だけ書き換えます。
周期は，280 Hz 固定になっています。

赤外送信と共用のため，赤外通信ドライバでタイマを使用中の場合は，戻り値が EOF になります。

・PWM_stop 関数
3.7.2 (5)の設定を行い，出力を停止します。

3.8　温度測定（A/D 変換）

温度を測定するには，温度センサの出力を内蔵 A/D コンバータで読み取ります。まずは測定例を示した後，A/D コンバータの使い方を説明します。

3.8.1　温度測定例

リスト 3－8A のプログラムは，Enter キーを押すごとに温度を測定し，今回の測定値，過去の最大値・最小値を表示します。

```
/* リスト 3－8A 温度測定 */
#include <stdio.h>
#include "AD.h"

int main(void) {
    char line[33];
    int imax = -100, imin = 1000, inew;

    while (1) {
        inew = TEMPERATURE( AD_read(ADch_TEMPERATURE) );
        if (inew > imax) imax = inew;
        if (inew < imin) imin = inew;
        printf("%d℃, %d, %d\n", inew, imax, imin);
        while(gets(line) == NULL);
    }
    return 0;
}
```

実行結果 3－8A
(赤字は入力例)

```
27℃, 27, 27
⊿
28℃, 28, 27
```

①最初に AD.h をインクルードします。
②温度測定は，次の２段階で行います。

・温度測定ドライバ（AD.c）に含まれる AD_read 関数で，温度センサの出力値を読み取ります。引数は，A/D コンバータの入力チャネル番号です。戻り値は，unsigned int 型で，上位 10 ビットに変換結果が格納されています。

・AD_read の戻り値を TEMPERTURE マクロで温度に変換します。
このマクロは AD.h で定義されています。結果は 1℃単位の整数になります。

3.8.2 内蔵 A/D コンバータの使用方法

ここでは，特定のチャネル（端子）の信号読み取り方法について説明します。
手順は次のとおりです。

(1)A/D コンバータに動作クロック供給

```
ADCEN = 1;
```

(2)モード・レジスタ（速度設定，コンパレータ・オン）

```
ADM = 0x1f;
```

これは，AVref = 2.7～5.5V，変換速度 5.2μs(20MHz 動作時)～52μs(2MHz 動作時)の設定です。他の設定は，78K0R のユーザース・マニュアルを参照してください。

(3)コンパレータ安定待ち

(2)の設定でコンパレータもオンになります。そこから 1μs 以上待ちます。

(4)端子設定，チャネル選択

```
ADPC = d;
```

d は，アナログ入力可能な端子（P20～P15x）の内，何本をデジタル入出力で使うかの指定です。ドライバ・キットではハードウエア初期設定で 9 に設定してあります。

```
ADS = ch;
```

ch はチャネル番号です。P20 端子が 0 になります。温度センサ（P151 端子）は 9 になります。

(5)動作開始

```
ADIF = 0;                変換終了フラグをクリア。
ADCS = 1;                変換開始。
while(!ADIF);            変換終了待ち（下記補足参照）。
結果格納変数 = ADCR;     変換結果保存。連続で動作するのですぐ格納します。
ADCS = 0;                変換終了（すぐに ADM = 0 を行う場合は不要）
```

補足：他の割り込み処理に 5.2μs(20MHz 動作時)以上かかるものがあると連続で変換動作します。2 回目の変換終了と ADCR 読み出しがぶつかった場合は，ADCR 読み出しが優先され，変な値が出ることはありません。

(6)停止

```
ADM = 0;                 変換終了およびコンパレータ・オフ。
ADCEN = 1;               A/D コンバータの動作クロック停止。
```

3.8.3　温度測定ドライバ（AD.c）概説

本版では，AD_read 関数，温度変換マクロの 2 つが定義されています。

(1)AD_read 関数

3.8.1 で説明した AD_read 関数は，ほぼ 3.8.2 の手順で A/D 変換値を読み取っています。

ただし，多少なりとも精度を上げるため，割り込み禁止にして測定開始後に HALT 関数（3.7.1 参照）を実行しています。A/D 変換以外の要因で HALT 解除された場合は再測定するようにしています。

もし割り込み禁止期間（5.2μs+α）に問題があれば，割り込み禁止にはせず，3.8.2 の手順通りにソースを修正してください。

(2)温度変換マクロ TEMPERATURE

温度センサの種類が変わっても良いように，計算式を AD.h でマクロ定義してあります。

マイコンの AVref 電圧と，温度センサの温度係数および 0℃電圧を以下のように指定します。

```
#define Vref_voltage     5000.0   /* Vref 端子電圧 [mV] */
#define Coef_degree      10.0     /* 温度センサの温度係数 [mV/℃] */
#define Base_voltage     600.0    /* 温度センサの 0℃電圧  [mV] */
```

計算式は精度を考慮してやや複雑になっていますが，基本は次のとおりです。

```
センサ出力電圧 = AD 読み取り値 * Vref_voltage / 1024 / 64;
                          1024 は A/D が 10bit 分解能のため。
                          64 は A/D 読み取り値が 6 ビットシフトされているため。
温度 = (センサ出力電圧  - Base_voltage) / Coef_degree;
```

結果は 1℃単位に丸めた整数にしています。

3.9　時計機能（RTC）

78K0R にはカレンダ機能を持ったリアルタイム・カウンタ（RTC）が内蔵されています。時計表示の例を示した後，RTC の使い方，目覚まし時計の例について説明します。

3.9.1　時計の表示と設定

リスト 3－9A にプログラム例を示します。

実行結果 3－9A
(赤字は入力例)

```
西暦 2010 年
 5 月 27 日(木)
 9 時 59 分 48 秒
⊿
>>2010  8  22  0  22
 4  15⊿
西暦 2010 年
 8 月 22 日(日)
22 時  4 分 15 秒
```

①time.h：時刻の取得と設定の関数を使うためのヘッダです。

③youbi：曜日の表示用テーブルです。

④time_t：C 言語規格で定められている暦時刻の型です。
　暦時刻とは日付と時刻を合体した算術型データですが，本版では 0 時，12 時からの秒数です。

⑤struct tm：C 言語規格で定められている“要素別の時刻”構造体です。以下のメンバを含んでいます。
　tm_year：年（西暦 1900 年を 0 とした数値），
　tm_mon：月（1 月を 0 とした数値），tm_wday：曜日（日曜日を 0 とした数値），
　tm_mday：1～31 日，tm_hour：0～23 時，tm_min：0～59 分，tm_sec：0～59 秒，

⑦time：C 言語規格で定められている関数で，暦時刻を取得します。
　引数　：戻り値を格納する暦時刻へのポインタ。通常は NULL を指定します。
　戻り値：取得された暦時刻。本版では 0 時,12 時からの秒数です（規格外）。

⑧localtime:C 言語規格で定められている関数で，暦時刻を要素別の時刻へ変換します。
　引数　：暦時刻。本版ではダミーで，RTC から時刻を得ています（規格外）。
　戻り値：要素別の時刻へのポインタ。

⑨printf の先頭で，表示開始位置を制御コード 0x18,0x01,0x01（画面左上）で指定しています。

⑩暦時刻が変化するか，Enter キーが押されるまで待ちます。Enter キーが押されると日付と時刻の入力になります。HALT 関数については，3.7.1①を参照してください。

⑪日付と時刻（次の 7 つ）を空白で区切って入力します。
　西暦 4 桁，月，日，曜日（日曜を 0 とするコード），時，分，秒

⑫settime：要素別の時刻を RTC に設定します。C 言語規格外の関数です。
　引数　：要素別の時刻へのポインタ。
　戻り値：成功時 0，失敗時 EOF（本版はノーチェックのため EOF は出ません）。

```
/* リスト３-9A 時計の表示と設定*/
#pragma HALT
#include <stdio.h>
#include <time.h>                                                    /*①*/

const char youbi[][3] = { "日", "月", "火", "水", "木", "金" , "土" };  /*③*/

int main(void) {
    char line[33];
    time_t timer;                                /*④*/
    struct tm *calendar, tset;                   /*⑤*/

    while (1) {
        timer = time(NULL);                                          /*⑦*/
        calendar = localtime(&timer);                                /*⑧*/
        printf("\x18\x01\x01西暦%d年\n" "%2d月%2d日(%2s)\n" "%2d時%2d分%2d秒\n",
            calendar->tm_year + 1900, calendar->tm_mon + 1,
            calendar->tm_mday, youbi[calendar->tm_wday],
            calendar->tm_hour, calendar->tm_min, calendar->tm_sec);

        line[0] = 1;
        while (timer == time(NULL) && gets(line) == NULL) HALT();    /*⑩*/
        if (line[0] == '\0') {
            printf("\n\n\n\n\x18\x01\x01>>");
            while(gets(line) == NULL);                               /*⑪*/
            if (sscanf(line, "%d%d%d%d%d%d%d", &tset.tm_year,
                &tset.tm_mon, &tset.tm_mday, &tset.tm_wday,
                &tset.tm_hour, &tset.tm_min, &tset.tm_sec) == 7) {
                tset.tm_year  -= 1900;
                tset.tm_mon --;
                settime(&tset);                                      /*⑫*/
            }
        }
    }
    return 0;
}
```

3.9.2 RTC の使い方

(1)RTC の起動手順

①32.768kHz 安定待ち

発振が安定するのを待ちます。この周波数帯の発振子は安定に 1 秒以上かかることがあります。特に 78K0R では発振電力を CMC レジスタで選択でき，電力が低いほど安定に時間がかかるようです。

ドライバ・キットのハードウエア初期設定 HDWINIT.c では，手抜きして安定待ちをしていません。リスト 3－9A の表示開始遅れから見て“通常発振”設定で 1 秒程度，“低消費発振”設定で 2 秒程度かかっていると思われます。

②レジスタ設定

```
RTCEN = 1;
RTCC0 = 8;          24 時制の設定です。
                    他の設定については 78K0R のマニュアルを参照してください。
SEC = 秒初期値(BCD コード 0x00～0x59)。
MIN = 分初期値(BCD コード 0x00～0x59)。
HOUR = 時初期値(BCD コード 0x00～0x23)。
WEEK = 曜日コード初期値(BCD コード 0x00～0x06，日曜が 0)。
DAY = 日初期値(BCD コード 0x01～0x31)。
MONTH = 月初期値(BCD コード 0x01～0x12)。
YEAR = 年初期値(BCD コード 0x00～0x99)。
```

③スタート

```
RTCE = 1;
```

(2)日付・時刻の読み取り方法

時計動作を妨害せずに読み取るには，次の手順で読みます。

```
RWAIT = 1;
while (!RWST);
SEC～YEAR の読み取り（順序は不問）。
RWAIT = 0;
while (RWST);
```

(3)日付・時刻の更新方法

全く別の値に書き換えるならば，RTCE = 0 にして SEC～YEAR を書き換え，RTCE = 1 にすればよいです。

現状より 1 秒進めるなど，時計動作を妨害せずに更新するには，次の手順にします。

```
RWAIT = 1;
while (!RWST);
SEC～YEAR の更新（順序は不問）。
RWAIT = 0;
while (RWST);
```

(4)誤差補正

筆者ハードウエアで周波数誤差を実測したところ，25℃で +10ppm（月差 +27 秒に相当）でした。また，この手の水晶発振子は 2 次の温度特性を持っており， -0.04ppm／℃^{2}程度の係数と思われます。

78K0R では，これらを補正するためのレジスタ（SUBCUD）を持っています。

温度センサを使った定期的な補正例を次に示します（1 分以上の間隔で実行）。

```
int x; /* テンポラリ変数 */

if (SEC == 0x50) {     /* 00, 20, 40 近辺以外で補正を実行 */
    x = TEMPERATURE( AD_read(ADch_TEMPERATURE) ) - 25;
    x = 25℃での誤差 * 100 - 4 * x * x;          /* 4 は温度係数の 100 倍 */
    x = - 補正量 / 102;                          /* 補正単位 1.02 ppm の 100 倍 */
    if (補正値 > 0) 補正値++;
    if (補正値 < -62) 補正値 = -62;
    if (補正値 > 62) 補正値 = 62;
    SUBCUD = 0x80 + (補正値 & 0x7f);
}
```

3.9.3　目覚まし時計

(1)アラーム機能の設定方法

78K0R の RTC には，スタンバイ中でも指定時刻を検出できるアラーム機能がついています。

設定方法は次のとおりです。

WALE = 1;　　アラーム機能を有効にします。

WALIE = 1;　　アラーム割り込みを使用する場合に指定します。

ALARMWW = 対象曜日;　　アラームを発生させる曜日を指定します。毎日であれば 0x7f を指定し，いくつかの曜日限定であれば，(1 << 曜日コード)を必要分加算します。

ALARMWH = 時;　アラームを発生する時を指定します。

ALARMWM = 分;　アラームを発生する分を指定します。

WAFG = 0;　　アラーム・フラグをクリアします。

RTCMK = 0;　　アラーム割り込みを許可する場合に設定します。ただし RTC インターバル割り込み機能を使う場合は，割り込みフラグが共用のため，割り込み内で WAFG が 1 かどうか判定します。

以降，RTC の曜日，時，分が指定と一致したら WAFG フラグが立ち，割り込みを許可している場合は割り込み処理に入ります。

(2)アラーム機能の使用例

リスト 3－9B に 78K0R のアラーム機能を使用した目覚まし時計の例を示します。

Enter キーを押すと時計設定または目覚まし時刻の設定ができます。

・時計設定の場合はリスト 3－9A と同じく西暦～秒までの 7 つを入力します。

・目覚まし設定の場合は，時，分のみ入力します。

・99 とだけ入力すると目覚まし設定を取消します。

指定時刻になると，サンプル・メロディが流れます。Enter キーを 2 回押すと止まります。

```
/* リスト 3-9B 目覚まし時計 */
#pragma SFR
#pragma HALT
#include <stdio.h>
#include <time.h>
#include "BEEP.h"
#include "BEEP.inc"
const char youbi[][3] = { "日", "月", "火", "水", "木", "金" , "土" };
#define  BCD(x) ( (unsigned char) (((x) / 10 << 4) + (x) % 10) )
```

```
int main(void) {
    char line[33];
    time_t timer;
    struct tm *calendar, tset;

    WALE = 1;
    while (1) {
        if (WAFG) {
            BEEP_play(whistle_polka, 0, 85, 0);
            WAFG = 0;
        }
        timer = time(NULL);
        calendar = localtime(&timer);
        printf("\x18\x01\x01西暦%d年\n" "%2d月%2d日(%2s)\n" "%2d時%2d分%2d秒\n",
                calendar->tm_year + 1900, calendar->tm_mon + 1,
                calendar->tm_mday, youbi[calendar->tm_wday],
                calendar->tm_hour, calendar->tm_min, calendar->tm_sec);
        if (ALARMWW == 0) printf("アラーム無し\n");
        else printf("アラーム %2x:%2x\n", ALARMWH, ALARMWM);

        line[0] = 1;
        while (timer == time(NULL) && gets(line) == NULL) HALT();
        if (line[0] == '\0') {
            tset.tm_year = 999;
            BEEP_stop();
            printf("\n\n\n\n\x18\x01\x01>>");
            while(gets(line) == NULL);
            if (sscanf(line, "%d%d%d%d%d%d%d", &tset.tm_year,
                    &tset.tm_mon, &tset.tm_mday, &tset.tm_wday,
                    &tset.tm_hour, &tset.tm_min, &tset.tm_sec) == 7) {
                tset.tm_year  -= 1900;
                tset.tm_mon --;
                settime(&tset);
            }
            else if (tset.tm_year == 99) ALARMWW = 0;
            else if (tset.tm_year >= 0 && tset.tm_year <= 23 &&
                     tset.tm_mon  >= 0 && tset.tm_mon  <= 59    )  {
                ALARMWH = BCD(tset.tm_year);
                ALARMWM = BCD(tset.tm_mon);
                ALARMWW = 0x7f;
            }
        }
    }
    return 0;
}
```

3.9.4 RTC ドライバ（RTC.c）概説

RTC ドライバの中には，3.9.1 で説明した time 関数，localtime 関数，settime 関数が入っています。また起動手順に対応した RTC_open 関数マクロが RTC.h で定義されています。

(1)time 関数，localtime 関数

これら C 言語規格の関数を見ると，時間は算術型データで管理し，そこから要素別の値に変換する，という方式に適した構成になっています。

一方，78K0R の RTC は年，月，日，時，分，秒というように要素別の BCD コード・カウンタで構成されています。要素別カウンタのビット長を単純に合計すると 32 ビットを越えるため，このままでは CC78K0R では算術型データとして扱えません。秒単位などの数値に換算すれば 32 ビットの数値に収まりますが，

time 関数で，要素別の値を秒単位などの数値に換算し，

localtime 関数で，秒単位などの数値を要素別に分解する。

というのは，78K0R クラスのマイコンにとって無駄が多すぎる気がします。

そこで本書の RTC ドライバでは，localtime 関数で直接 RTC から要素別の値を取得するというようにしてあります。この場合 time 関数は無くてもよいのですが，一応 16 ビット範囲内でのカウント値を返す関数として定義してあります。

(2)settime 関数

本版（R1.20）も，数値の範囲や曜日の整合についてチェックを行っていません。正しい数値を設定するようにしてください。

(3) RTC_open 関数マクロ

オープン関数の代わりとしてマクロ定義が RTC.h に記述されています。3.9.2 (1)②～③の手順が反映してあります。

さらにここで誤差補正のデータも記述しています。筆者ハードウエアの実測値が+10ppm 程度だったので，誤差補正は -8ppm にしてあります。実測値に合わせて変更してください。

実測する時は，コンパイラ・オプション→プリプロセッサ→定義マクロに RTCCHECK を追加して，リビルドしてください。P12 端子から 32.768kHz のクロックが出ます。測定に使用する周波数カウンタは原発振精度±1ppm 以下で，少なくとも 0.1 Hz 単位の表示がないと効果的な補正値が得られません。

3.10　パソコン COM ポート接続（UART 通信）

マイコンの内蔵シリアルインタフェースをパソコンの COM ポート（USB 経由の CDC，仮想 COM ポート含む）に接続して，パソコン側のソフトと通信する方法について説明します。

3.10.1　パソコン側の準備

ここでは，USB 経由の仮想 COM ドライバのインストールと汎用ターミナル・ソフトのセットアップについて説明します。

(1)仮想 COM ドライバのインストール

同梱回路図（DDT8 以外）では，秋月電子通商の USB－シリアル変換モジュールを使用しています。ドライバのインストール方法は，そのモジュールの説明書を参照してください。

説明書のドライバ名は古いかもしれません。最新のドライバをインストールします。

DDT8 の場合は，雑誌記載のインストール方法に従ってください。

(2)汎用ターミナル・ソフトのセットアップ

①ターミナル・ソフトの種類

送信時に改行コード（\n：LF）を付加できるものなら何でもよく，Windows のハイパーターミナル，Tera Term あるいは Tera Term Pro（ベクターからダウンロード可能）などが使えます。

②ターミナル・ソフトの設定

次の条件を設定します。

設定項目	Tera Term / Pro	ハイパーターミナル
ボーレート：19200bps（変更可能）， データ：8 ビット，パリティ：無し， ストップ：1 ビット， フロー制御：ハードウエア	設定(setup)→シリアルポート(serial port)	ファイル→新しい接続
送信時：改行（LF）を付加する。 漢字コード：シフト JIS ローカルエコー:あり 端末タイプ：VT100J など	設定(setup)→端末(terminal)	ファイル→プロパティ→設定タブ→各詳細

なお，改行キーとしては英字キー横の Enter キーを使用してください。

Tera Term で 10 キー横の Enter キーを有効にするには，インストール・フォルダの KEYBOARD.CNF の NumEnter=284 をコメントアウト（先頭に;付加）します。

3.10.2　パソコンとの通信例

リスト 3－10A に，パソコンから送ったメッセージをマイコン側の LCD に表示し，同じ内容をパソコンへ送り返す例を示します。

```
/* リスト 3－10A パソコンへのループバック */
#include <stdio.h>
#include <string.h>

int main(void) {
    char tmp[100];
    FILE *fpr, *fpw;

    fpw = fopen("COM:", "wb");                    /*①*/
    fpr = fopen("COM:", "rb");
    if (fpr == NULL || fpw == NULL) {
        printf("error\n");
        return 0;
    }

    fputs("通信開始\r\n", fpw);                    /*②*/
    while (1) {
        while (fgets(tmp, sizeof(tmp), fpr)       /*③*/
                == NULL);
        printf("%s", tmp);
        fputs(tmp, fpw);
        if (!strcmp(tmp, "end\r\n")) break;
    }

    fputs("通信終了\r\n", fpw);
    if (fclose(fpr) == EOF) printf("error\n");   /*④*/
    if (fclose(fpw) == EOF) printf("error\n");
    printf("closed\n");
    return 0;
}
```

実行結果 3－10A

パソコン側(赤枠が入力)

マイコン側(表示結果)

```
test message
簡素に極意あり
end
closed
```

①通信ポートのオープン

fopen 関数でストリームをオープンします。

引数１ ：オープンする外部ファイル（物理装置含む）の指定で，UART 通信の場合は”COM:”を指定します。

引数２ ：オープンモードを指定します。ドライバ・キットでは，送信の場合は”wb”，受信の場合は”rb”を指定します。

戻り値：ストリームへのポインタ。

②文字列の送信

fputs 関数でストリームへ文字列を送ります。

引数１ ：送る文字列へのポインタあるいは文字列リテラルを指定します。

引数２ ：ストリームへのポインタ。

戻り値：成功すると 0，失敗すると EOF。

なお，LCD へ文字列を表示する puts 関数では最後に自動的に改行（\n）が付加されますが，fputs 関数は付加されません。

１バイトづつ送る場合は，fputc 関数を使用します。この場合，引数１は１バイトのコードを指定します。

③文字列の受信

fgets 関数でストリームから改行（\n）までの文字列を取得します。

引数１ ：取得文字列を格納する配列へのポインタを指定します。

引数２ ：最大取得文字数 + 1 を指定します。通常は配列サイズ（sizeof 配列名）を指定します（改行が来ないときの制限用）。

引数３ ：ストリームへのポインタ。

戻り値：成功すると引数１と同じ値，失敗すると NULL。

なおキーから文字列を取得する gets 関数は改行（\n）が捨てられますが，fgets 関数は改行（\n）も読み込みます。最後には gets と同じく\0 が付加されます。

ドライバ・キットの fgets 関数は，１バイトでもデータがあると\n が来るか引数２-１文字を読み込むまで待ち状態になります。

１バイトづつ受信する場合は，getc 関数を使用します。この場合，引数はストリームへのポインタで，戻り値は受信コード（ただし受信がなければ EOF）になります。

④通信ポートのクローズ

fclose 関数で終了します。送信の場合は，送信完了まで待ち状態になります。

引数１ ：ストリームへのポインタ。

戻り値：成功すると 0，失敗すると EOF。

3.10.3　UART（非同期シリアル・インタフェース）の使い方

(1)概要

78K0R には，シリアル・アレイ・ユニット（SAU）と呼ばれるシリアル・チャネルの集合ユニットが内蔵されており，3 線クロック同期シリアル，UART（非同期シリアル），簡易 I^2C として使用できます。

品種により SAU が複数搭載されているものもあります。以下の説明は，本書の 78K0R/KC3-L～KE3-L の場合の SAU0 について説明します。3.2.2 (6)の注意事項も参照してください。

SAU0 は，シリアル・チャネル 4 本から構成されています。個々のチャネルは単独で 3 線クロック同期シリアルあるいは簡易 I^2C，2 チャネルで UART として使用できます。ただし組合せは自由でなく，選択できる機能が決まっています（表 3－10A 参照）。

本書（78K0R/KC3-L～KE3-L）ではチャネル 2，3 を使う UART1 を COM ポート接続用として割り当てています。チャネル 2 が送信側，チャネル 3 が受信側になります。ただし DDT8 では，チャネル 0 が送信側，チャネル 1 が受信側になります。

SAU を使うには，個々のチャネル設定の前に全体の設定が必要です。

```
SAU0EN = 1;                    ・・・SAU0 を有効にします。
NOP(),NOP(),NOP(),NOP();       ・・・4 クロック待ちます。NOP 関数を使うには，#pragma NOP をプログラムの始めの方に記述します。
SPS0 = PRS01 << 4 | PRS00;     ・・・個々のチャネルで使うクロック設定
```

ここで，個々のチャネルでは 2 種類のクロックしか選択できず，それぞれのクロック周波数を PRS01，PRS00 で指定します。設定値は CPU クロックの分周比コードです。

0:1/1，1:1/2，2:1/4，3:1/8，4:1/16，5:1/32，・・・，11:1/2048

本書（78K0R/KC3-L～KE3-L）では PRS00 を UART1 用として専用に割り当てています。設定値はボーレートに応じて決定します。

表 3－10A　SAU0 の機能選択（78K0R/KC3-L～KE3-L）

チャネル	3 線シリアル	UART	簡易 I^2C
0	CSI00	UART0	－
1	CSI01		－
2	CSI10	UART1	IIC10
3	－		－

(2)送信チャネルの設定方法（送信チャネル番号を s, UART 番号を n とします）

まずボーレートから設定値を計算します。まず,

マイコン動作周波数 ／ 2^m ＜ 128

となる最小の m を決定します。ただし 0 < m < 13 の範囲です。次に

分周値 = マイコン動作周波数 ／ 2^m ／ ボーレート

を計算します。小数点以下は四捨五入して整数にします。

```
SPS0 = SPS0 & 0xf0 | m - 1;      PRS01 を使わないなら SPS0 = m - 1; で可。
SMR0s = 0x0023;                  PRS01 の場合は 0x8000 を加算。
SCR0s = 0x8097;                  パリティ無しの設定の場合。
SDR0s = (分周値 - 1) << 9;
SOL0 = ~(1 << s);                初期値から変更してなければ省略可。
SO0 = 0x0808 | 1 << s;
SOE0L.s = 1;
TxDn 端子を出力に設定し 1 を出力。        ただし初期設定済なら省略可能。
SS0L.s = 1;
```

これで送信可能状態になります。TXDn = データ; でデータを送信できます。

また送信バッファに空きができると，INTSTn 割り込み信号がでます。

停止の手順は次のとおりです。

```
SMR0s &= ~1;                     送信完了を検出できるモードに設定。
while(SSR0s & 0x40);             送信完了待ち。
ST0L.s = 1;                      送信チャネル停止。
SOE0L.s = 0;                     TxDn 端子を使用しない場合は省略可能。
```

(3) 受信チャネルの設定方法（受信チャネル番号を r, UART 番号を n とします）

```
SPS0 = SPS0 & 0xf0 | m - 1;      PRS01 を使わないなら SPS0 = m - 1; で可。
SMR0s = 0x0023;                  送信側も設定必要。
SMR0r = 0x0122;                  PRS01 の場合は 0x8000 を加算。
SCR0r = 0x4497;                  パリティ無しの設定の場合。
SDR0r = (分周値 - 1) << 9;
NFEN0.(1 << 2 * n) = 1;          受信ノイズフィルタのオン（必要に応じて）。
SS0L.r = 1;
```

これで受信可能状態になります。データを受信すると INTSRn 割り込み信号がでます。

停止の手順は次のとおりです。

```
ST0L.r = 0;                      受信チャネル停止。
NFEN0.(1 << 2 * n) = 0;          受信ノイズフィルタのオフ（必要に応じて）。
```

(4)フロー制御

マイコン側とパソコン側でデータ転送可能な状態かどうかを確認しながら転送する方法をフロー制御と言います。

・ハードウエア・フロー制御

データ線の他に何本かの制御線を使う方式です。本書の UART 通信ドライバでは RTS，CTS の 2 本を使っています。

RTS：アクティブ・レベルで相手に送信要求する出力信号です。すなわち自分が受信可能状態でアクティブ・レベルにします。

CTS：アクティブ・レベルで相手が受信可能と判断する入力信号です。

もし接続相手に RTS，CTS がなければ，自分の RTS と CTS を接続します。

またアクティブ・レベルは一般的にはロー・アクティブが多く，本書の UART 通信ドライバもロー・アクティブで動作します。

なお DDT8 版は，RTS，CTS が実ポートに接続されてないので，1 回の転送メッセージを 120 バイト以内とし，メッセージ間隔を十分あけてください。

・Xon／Xoff 制御（ソフトウエア・フロー制御）

Xoff コード（0x13）を受信したら送信を停止し，Xon コード（0x11）を受信したら送信を再開する方式です。本書の UART 通信ドライバでは対応していません。

3.10.4 UART通信ドライバ（UART.c）概説

(1)使用方法

フロー制御用の下記関数を0.01秒あるいはそれ以下の周期で呼び出します。

```
void UART_TX_check(void);
void UART_RX_check(void);
```

その他は3.10.2で説明した標準関数から呼び出されるか，割り込みで自動起動します。
また定周期割り込み処理中でも送信バッファ空き／受信の割り込みがかかるように割り込み優先レベルを2に設定してあります。

デフォルトのfopen関数，fclose関数を使うと赤外線通信ドライバも組み込まれます。それを避けたい場合は，driver_list.hの中でIR.hをインクルードしている部分をコメントアウトします。あるいは直接UART通信オープン／クローズ関数を呼び出します。その場合はUART.hをインクルードします。戻り値は，fopen関数，fclose関数と同じです。

```
extern FILE *UART_TX_open(void);        /* UART送信オープン */
extern FILE *UART_RX_open(void);        /* UART受信オープン */
extern int UART_TX_close(void);         /* UART送信クローズ */
extern int UART_RX_close(void);         /* UART受信クローズ */
```

(2)ボーレート変更方法

①UART.h内の下記ボーレート定義値を希望のボーレートに変更します。

```
#define UART_SPEED 19200
```

②UART.hをインクルードし，UARTをオープンする前に次の記述を行います。

```
UART_set_baud();
```

ただし，ドライバのソースを再コンパイルする場合は不要です。

(3)ストリームのバッファ・サイズ

送受信とも127バイトです。
ドライバ・キットのfputs関数は，ストリームが満杯の場合は空くまで待ちます。
putc関数はストリームが満杯ならEOFを返すので，ユーザプログラム側で対処します。

3.11　赤外線通信（パルス間隔測定）

赤外線によるデータ通信の例と，赤外線受信に使用しているパルス間隔測定の方法について説明します。

3.11.1　赤外線通信の例

リスト 3−11A にキー入力したメッセージの送信と，それを自分で受信する例について示します。

```
/* リスト 3−11A 赤外通信(ループバック) */
#include <stdio.h>
#include <string.h>
#include <time.h>
#define CLOCK() \
    ((int) clock() / (CLOCKS_PER_SEC / 100))

int main(void) {
    int i, c, t;
    char tmp[100];
    FILE *fpr, *fpw;

    fpr = fopen("IR:", "rb");                    /*①*/
    fpw = fopen("IR:", "wb");
    if (fpr == NULL || fpw == NULL) {
        printf("error\n");
        return 0;
    }

    fputs("通信開始\n", fpw);                    /*②*/
    while (1) {
        i = 0;
        t = CLOCK();
        while (CLOCK() - t < 200 && i < sizeof tmp - 1) {
            if ( (c = getc(fpr)) != EOF)         /*③*/
                tmp[i++] = (char) c;
            if (c == '\n') break;
        };
        tmp[i] = '\0';
        printf("Rec:%s", tmp);

        while (gets(tmp) == NULL);
        if (!strcmp(tmp, "e")) break;
        strcat(tmp, "\n");
        fputs(tmp, fpw);
    }

    if (fclose(fpr) == EOF) printf("error\n");  /*④*/
    if (fclose(fpw) == EOF) printf("error\n");
    printf("closed\n");
    return 0;
}
```

実行結果 3−11A
(赤字は入力例)

```
Rec:通信開始
test さつき⊿
Rec:test さつき
e⊿
closed
```

①通信ポートのオープン

fopen 関数でストリームをオープンします。

引数１ ：オープンする外部ファイル（物理装置含む）の指定で，赤外線通信の場合は”IR:”を指定します。

引数２ ：オープンモードを指定します。ドライバ・キットでは，送信の場合は”wb”，受信の場合は”rb”を指定します。

戻り値：ストリームへのポインタ。

②文字列の送信

fputs 関数でストリームへ文字列を送ります。

引数１ ：送る文字列へのポインタあるいは文字列リテラルを指定します。

引数２ ：ストリームへのポインタ。

戻り値：成功すると 0，失敗すると EOF。

なお，LCD へ文字列を表示する puts 関数では最後に自動的に改行（\n）が付加されますが，fputs 関数は付加されません。

１バイトづつ送る場合は，fputc 関数を使用します。この場合，引数１は１バイトのコードを指定します。

③文字の受信

getc 関数で１バイト単位に受信しています。

引数１ ：ストリームへのポインタ。

戻り値：成功すると取得した値，失敗する EOF。

UART 通信のように fgets 関数を使うこともできますが，ドライバ・キットの fgets 関数は １ バイトでもデータがあると改行（\n）が来るか指定文字数を読み取るまで待ち状態になります。赤外通信の場合は，距離や障害物の状況によりエラーになる可能性が高く，万一改行（\n）を読み落とすと戻れなくなります。

そこで getc 関数で１バイトづつ受信し，２秒たっても改行（\n）が来ない場合は強制終了しています。

④通信ポートのクローズ

fclose 関数で終了します。送信の場合は，送信完了まで待ち状態になります。

引数１ ：ストリームへのポインタ。

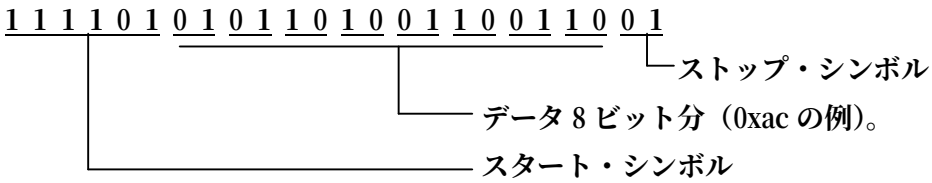
戻り値：成功すると 0，失敗すると EOF。

3.11.2　通信フォーマット

赤外線通信ドライバ（IR.c）で使用しているフォーマットを説明します。適当に作ったフォーマットなので，特に何かの規格に準拠しているわけではありません。

1 シンボル：長さ 0.6ms，'1'の時に 38kHz キャリア出力，'0'の時は無信号。

1 パケット：24 シンボル。載せられる情報は 1 バイト。構成は次のとおり。



情報 1 ビット：2 シンボルで構成。LSB ファーストでパケットに載せます。

ビット情報'0'：シンボル 01

ビット情報'1'：シンボル 10

パケット間隔：最低 0.6ms の無信号。

このフォーマットを受信側で解析する方法の一つとして，0→1 のエッジ間隔で判定する方法を次に示します。ただし赤外受光モジュールはアクティブ・ロウ出力なので，マイコンから見ると立ち下がりエッジ間隔になります。

スタートシンボル　：0.6 x 5 = 3.0ms　　(111101)
データ 0→1→0 変化：0.6 x 4 = 2.4ms　　(011001)
データ 0→1→1 変化：0.6 x 3 = 1.8ms　　(011010)
データ 1→0 変化　：0.6 x 3 = 1.8ms　　(1001)
データ 1→1 変化　：0.6 x 2 = 1.2ms　　(1010)
データ 0→0 変化　：0.6 x 2 = 1.2ms　　(0101)

実際の判定は，±0.3ms 範囲を許容値とします。解析手順は次のとおりです。

①間隔 3.0ms でスタート・シンボルと判定し，前回データ 0 とします。

②間隔 2.4ms で前回データ 0 ならば，1, 0 の 2 ビットをデータに追加します。ただし，1 を追加した時点で 1 バイト完成なら 0 はストップ・シンボルとして破棄します。

③間隔 1.8ms の場合，前回データ 0 なら 1，1 の 2 ビット，前回データ 1 なら 0 と判定します。

④間隔 1.2ms の場合，前回データ 0 なら 0，前回データ 1 なら 1 と判定します。

あり得ない組合せになった場合はスタート待ちに戻ります。

3.11.3 パルス間隔測定の方法

パルス間隔測定には，タイマ・アレイ・ユニット（TAU）の 1 チャネルを使用します。
設定手順は次のとおりです（n はチャネル番号）。

(1)TAU 共通設定（3.5.1 参照）

(2)ノイズ・フィルタ設定

```
NFEN1.n = 1;                    必要に応じて設定します。
```

(3)モード設定（クロック選択）

```
TMR0n = 0x0104;                 分周クロック PRS00，立ち下がり検出の場合。
```

分周クロック PRS01 を使用する場合は 0x8000 を加算します。
立ち上がり検出の場合は,0x0040，両エッジ検出の場合は 0x0080 を加算します。

(4)動作開始

```
TS0L.n = 1;                     動作開始。
TMMK0n = 0;                     割り込み許可。
```

以降カウンタは次の動作をします。

①カウンタ TCR0n に 0 を代入。
②TMR0n で指定された PRS00 または PRS01 のクロック周期で，TCR0n を+ 1。
③エッジを検出したら，TCR0n の値を TDR0n に保存したうえで①へ戻る。また割り込み信号も発生する。

TDR0n に保存された測定値は，次のエッジまでに読み取る必要があります。通常は③で発生する割り込みによって保存のための割り込み関数を起動します。

(5)動作を停止する場合

```
TMMK0n = 1;                     割り込み禁止。
TT0L.n = 1;                     停止。
NFEN1.n = 0;                    ノイズ・フィルタのオフ設定。
```

3.11.4 赤外線通信ドライバ（IR.c）概説

(1)使用方法

0.01 秒あるいはそれ以下の周期で下記関数を呼び出します。

```
void IR_TX_check(void);          送信シンボルの組立
void IR_RX_check(void);          受信データ・ビット組立
```

その他は，3.11.1 で説明した標準関数から呼ばれるか，割り込みで自動起動します。
定周期割り込み中でも割り込みがかかるように割り込み優先レベルを 2 にしています。

デフォルトの fopen 関数，fclose 関数を使うと UART 通信ドライバも組み込まれます。それを避けたい場合は，driver_list.h の中で UART.h をインクルードしている部分をコメントアウトします。あるいは直接 IR オープン／クローズ関数を呼び出します。その場合は，IR.h をインクルードします。戻り値は fopen 関数，fclose 関数と同じです。

```
extern FILE *IR_TX_open(void);          /* 赤外送信オープン */
extern int IR_TX_close(void);           /* 赤外送信クローズ */
extern FILE *IR_RX_open(void);          /* 赤外受信オープン */
extern int IR_RX_close(void);           /* 赤外受信クローズ */
```

(2)ストリームのバッファ・サイズ

送受信とも 31 バイトです。
ドライバ・キットの fputs 関数は，ストリームが満杯の場合は空くまで待ちます。
putc 関数はストリームが満杯なら EOF を返すので，ユーザプログラム側で対処します。

受信は最短で 1 バイト 15ms 周期なので，連続 32 バイト以上を受信する場合は，480ms 以内にストリームから読み出しが必要です。

(3)38kHz 送信キャリアのデューティ

節電のため 33%にしてあります。到達距離やエラー率の状況によっては 50%程度に変更した方が良いかもしれません。