

C から体感する RL78 マイコン(CC-RL 版)

(C)2010-2012, 2017 てきーらサンドム

R1.05 2017/8/15

3章 ハードなC言語 (マイコン特有部分)

3.1 Cから見たマイコンの基礎

3.1.1 main を取り巻くマイコン全体の動き

3.1.2 アドレス空間

3.1.3 関数・変数の配置と型修飾子

3.1.4 オプション・バイト

3.2 ポート操作の基本

3.2.1 LED 点滅とキー検出

3.2.2 ポート制御変数

3.3 キー入力

3.3.1 チャタリング除去

3.3.2 キー入力ドライバ (KEY.c) 概説

3.3.3 キーコード変換ドライバ (FEP.c) 概説

3.4 LCD制御

3.4.1 LCDのリセット解除

3.4.2 制御コマンドや表示データの書き込み

3.4.3 ステータスや表示データの読み込み

3.4.4 コマンドの種類

3.4.5 表示データ

3.4.6 市松模様の表示例

3.4.7 LCDドライバ (LCD.c) 概説

3.4.8 フォント・ドライバ (FONT.c) 概説

3.5 定期的な処理 (インターバル・タイマと割り込み)

3.5.1 タイマ概要

3.5.2 インターバル・タイマとしての使用

3.5.3 割り込み処理

3.5.4 定周期ドライバ (BTIMER.c) 概説

- 3.6 メロディ制御（方形波出力）
 - 3.6.1 サンプル・メロディの演奏
 - 3.6.2 方形波の生成
 - 3.6.3 ブザー出力の簡単な例
 - 3.6.4 音階周波数
 - 3.6.5 メロディ制御ドライバ（BEEP. c）概説
- 3.7 LED 調光（PWM 制御）
 - 3.7.1 明るさの制御例
 - 3.7.2 PWM 波形の生成
 - 3.7.3 LED 調光ドライバ（PWM. c）概説
- 3.8 温度測定（A/D 変換）
 - 3.8.1 温度測定例
 - 3.8.2 内蔵 A/D コンバータの使用法
 - 3.8.3 温度測定ドライバ（AD. c）概説
- 3.9 時計機能（RTC）
 - 3.9.1 時計の表示と設定
 - 3.9.2 RTC の使い方
 - 3.9.3 目覚まし時計
 - 3.9.4 RTC ドライバ（RTC. c）概説
- 3.10 パソコン COM ポート接続（UART 通信）
 - 3.10.1 パソコン側の準備
 - 3.10.2 パソコンとの通信例
 - 3.10.3 UART（非同期シリアル・インタフェース）の使い方
 - 3.10.4 UART 通信ドライバ（UART. c）概説
- 3.11 赤外線通信（パルス間隔測定）
 - 3.11.1 赤外線通信の例
 - 3.11.2 通信フォーマット
 - 3.11.3 パルス間隔測定の方法
 - 3.11.4 赤外線通信ドライバ（IR. c）概説

3章 ハードなC言語

この章ではC言語でハードウェアを操作する方法について説明します。

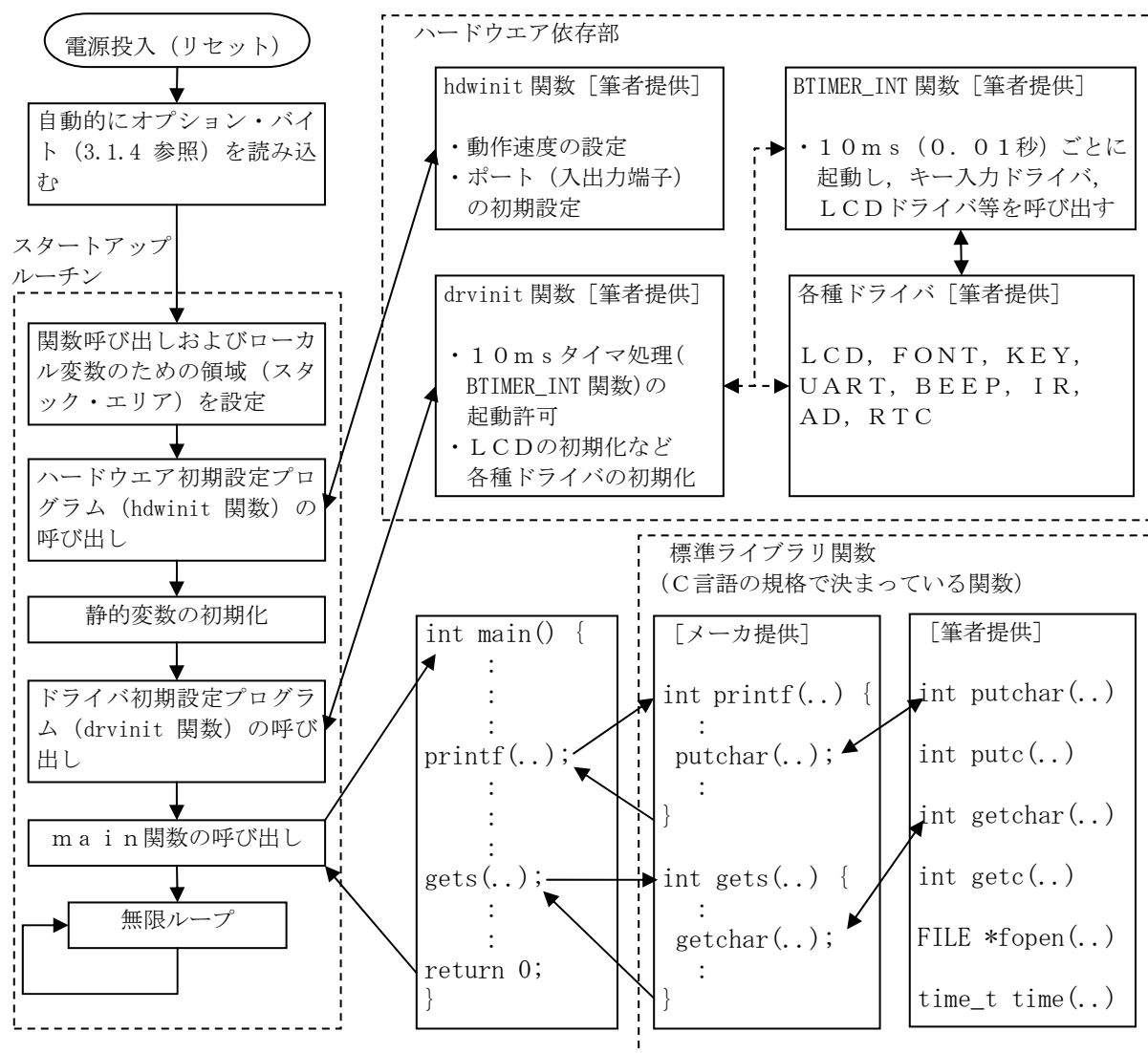
3.1 Cから見たマイコンの基礎知識

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



(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、RL78 は POR (パワー・オン・リセット) 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

RL78 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、一部のハードウェ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。RL78 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、ここからユーザがプログラムを書かねばなりません。CS+ for CC 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、ハードウェ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・ルーチンは、CS+ for CC でプロジェクトを作成する時に自動的にプロジェクト・フォルダに格納されます。

ハードウェ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。

ドライバ・キットではドライバ初期化プログラムをスタートアップから呼び出すために専用のスタートアップ・ルーチン (`RESET.asm`) を同梱しています。

なお、標準のスタートアップ・ルーチンには、図 3-1 に示すとおり「静的変数の初期化」が入っていますが、ドライバ・キットでは `RESET.asm` ではなく `hdwinit` 関数の中で行っています。

(4)ハードウェア初期設定 (`hdwinit` 関数)

`main` 関数の中でハードウェ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。

RL78 はリセット直後の初期状態では、動作速度がオプション・バイトで指定した周波数

(1~32MHz), 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 (外部発振) に変更します。
- ・LCD やキー等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します。
- ・C 言語用の変数領域の初期化およびスタック領域の初期化 (スタック消費量測定用) を行います。

(5) ドライバ初期設定 (`drvinit` 関数)

各種ドライバの初期設定を行います。`hdwinit` 関数が C 言語変数の初期化前に呼び出されるのに対し, 本関数は初期化後に呼び出されます。一般的には `main` 関数の冒頭で初期化しますが, 本キットではドライバ部分になるべく目に触れないようにしているため, `main` 関数の直前で別に呼び出しています。

主たる初期化の内容は次のとおりです。

- ・OS (オペレーティング・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します。
- ・LCD モジュールに初期化コマンドを送って表示可能とします。
- ・RTC (時計) の動作を開始します。

(6) その他ハードウェア依存部

`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` 関数に戻るといった動きをします。

`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と, `printf` からの表示要求が干渉して表示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。

(7) `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（デフォルトのデバッグ設定の場合）。

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

(8) 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ェアがまちまちなので、一部提供されていなかったり、ハードウェ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。

RL78 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、周辺機能 (SFR) 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。RL78 はフラッシュ ROM と呼ばれる書き換え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。0xFFFF0 番地より下側に配置されています。

③SFR 領域：

内蔵ハードウェアに対応付けられた領域です。CC-RL では、プログラムの先頭で

```
#include "iodefine.h"
```

と書くと、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ます。1 ビットのフラ

グは専用の名前がついたもの以外は、「レジスタ名_bit.no ビット位置」の形式で書きます。
ここでビット位置は0～7の数字です。

④ミラー領域：

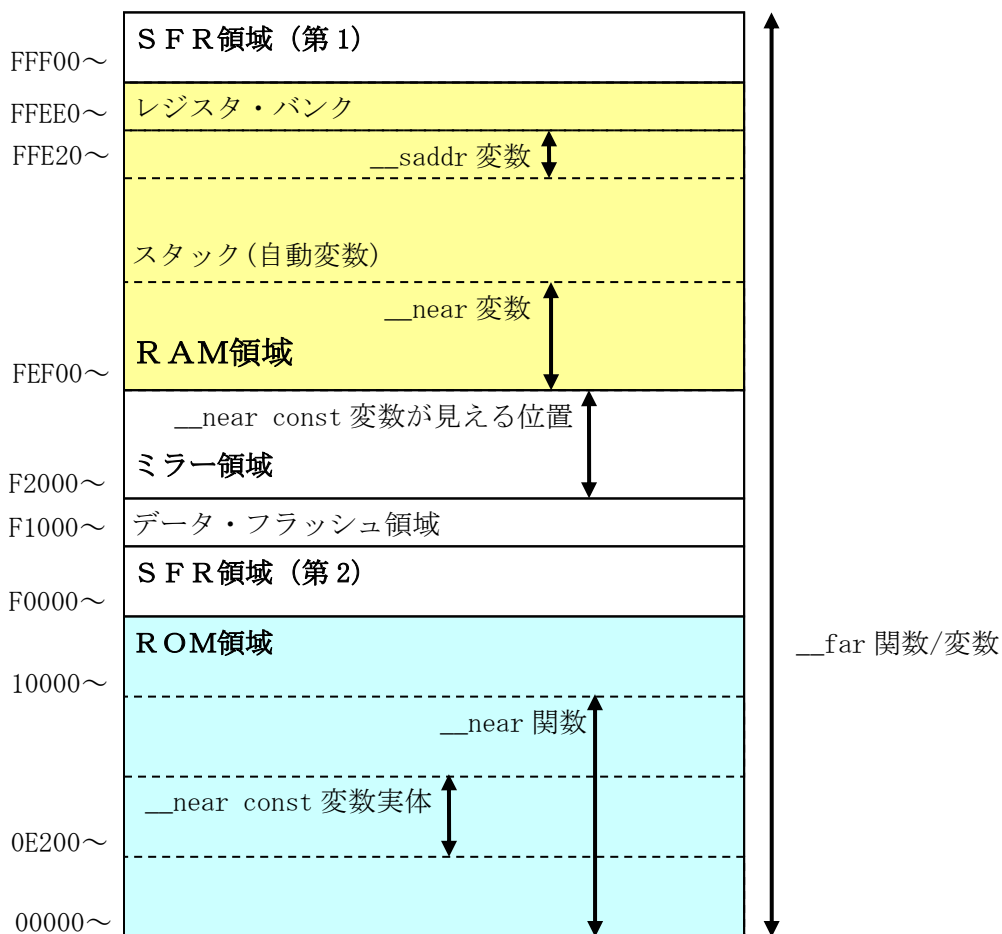
ROM領域に格納した定数の一部が写像されて見える領域です。

これはRL78および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。

⑤データ・フラッシュ領域

RL78で新たに付け加えられた領域です。

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、RL78 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、CC-RL 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、

“ビルド・ツールのプロパティ” → “コンパイル・オプション” → “メモリ・モデル” → “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`

デフォルト修飾を変更する場合は、`__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、0xf2000 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 `__saddr` 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
<code>__near</code>	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。
<code>__far</code>	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。
<code>const</code>	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。
<code>__saddr</code>	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 192 バイト以内と非常に小さい。

3.1.4 オプション・バイト

RL78 はリセット直後に自動的にオプション・バイトを読み取って設定を行います。

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
- ・ウォッチドッグ・タイマの設定
- ・内蔵発振回路, LVD (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、“ビルド・ツールのプロパティ” → “リンク・オプション” → “デバイス” → “オンチップ・デバッグ・オプション・バイト制御値” のところです。

制御値は、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。

(2) ユーザ・オプション・バイト値

設定箇所は、“ビルド・ツールのプロパティ” → “リンク・オプション” → “デバイス” → “ユーザ・オプション・バイト値” のところです。

①1 バイト目 (ウォッチドッグ・タイマの設定)

ウォッチドッグ・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ェアです。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。設定例は以下の通りです。

00 : ウォッチドッグを使用しない (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

ドライバ・キットでは、7C にしてあります。

②2 バイト目 (LVD (電圧検出回路) の設定)

ドライバ・キットでは使用していないため、FF を設定しています。

③3 バイト目 (内蔵発振回路の設定)

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。

RL78 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。

特にリセット直後は内蔵発振器 (高速オンチップ・オシレータ) で動作し、その周波数を 3 バイト目で指定できます。ドライバ・キットでは 16MHz (設定値 E9) としています。起動後に周波数変更することも可能です。

3.2 ポート操作の基本

ここではポート（入出力端子）への信号出力や信号入力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。

メーカー定義のポート名は P6 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ェ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の代入や参照ができます (ハードウェアによっては制限あり)。

メーカー定義の 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)。

```
#include "iodefine.h"
```

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をご容赦下さい。

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の設定やプルアップ抵抗のオン/オフ設定が必要です。RL78 では、おもに次の 4 種類の SFR 変数名でポートを制御します。

①ポート・モード・レジスタ

PM(x) : (8 ビット単位)

PM(x)_bit.no(y) : (1 ビット単位)

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。

②ポート・レジスタ

P(x) : (8 ビット単位)

P(x)_bit.no(y) : (1 ビット単位)

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。

③プルアップ抵抗オプション・レジスタ

PU(x) : (8ビット単位)

PU(x)_bit.no(y) : (1ビット単位)

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。

④ポート・モード・コントロール・レジスタ

PMC(x) : (8ビット単位)

PMC(x)_bit.no(y) : (1ビット単位)

アナログ入力兼用ポートの内、ANI16以上のポートをデジタル入出力にする場合に0を代入します。

リセット後の初期値は1、すなわちアナログ入力になっています。

なお、ANI14以下のアナログ入力兼用ポートはADPCレジスタで設定します。

ADPCレジスタは、リセット後に0となり、全てアナログ入力になります。

一部をデジタル入出力に変更したい場合は、(アナログ入力として残す本数)+1をADPCレジスタへ代入します。

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 端子名, ピン番号 【RL78/G13 64ピン用】

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。

PM : ポート・モード指定。M=1 (デフォルト値) で入力, M=0 で出力。

PMC : アナログ入力指定。C=1 (デフォルト値) でアナログ入力, C=0 でデジタル入出力。

注 : OD はオープン・ドレイン端子 (Hが出ない)

PU	PM	PMC	端子名	ピン
U	M	—	P00	62
U	M	—	P01	61
U	M	C	P02	60
U	M	C	P03	59
U	M	—	P04	58
U	M	—	P05	31
U	M	—	P06	30
U	M	—	P10	46
U	M	—	P11	45
U	M	—	P12	44
U	M	—	P13	43
U	M	—	P14	42
U	M	—	P15	41
U	M	—	P16	40
U	M	—	P17	39
—	M	—	P20	56
—	M	—	P21	55
—	M	—	P22	54
—	M	—	P23	53
—	M	—	P24	52
—	M	—	P25	51
—	M	—	P26	50
—	M	—	P27	49
U	M	—	P30	32
U	M	—	P31	21
U	M	—	P40	5
U	M	—	P41	4
U	M	—	P42	3
U	M	—	P43	2

PU	PM	PMC	端子名	ピン
U	M	—	P50	33
U	M	—	P51	34
U	M	—	P52	35
U	M	—	P53	36
U	M	—	P54	37
U	M	—	P55	38
—	M	—	P60 (OD)	17
—	M	—	P61 (OD)	18
—	M	—	P62 (OD)	19
—	M	—	P63 (OD)	20
U	M	—	P70	29
U	M	—	P71	28
U	M	—	P72	27
U	M	—	P73	26
U	M	—	P74	25
U	M	—	P75	24
U	M	—	P76	23
U	M	—	P77	22
U	M	C	P120	1
—	—	—	P121	11
—	—	—	P122	10
—	—	—	P123	8
—	—	—	P124	7
—	—	—	P130	57
—	—	—	P137	9
U	M	—	P140	64
U	M	—	P141	63
U	M	—	P146	47
U	M	C	P147	48

表 3-3 ドライバ・キットの再定義名一覧 【RL78/G13 64ピン用】

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	P2.3	o	0	LCD ライト/リード指定
LCD_DI	P2.4	o	0	LCD データ/コマンド指定
LCD_E	P2.7	o	0	LCD ライト・リードパルス
LCD_CS1~2	P2.5~2.6	o	0	LCD チップセレクト (左半分, 右半分)
LCD_RESET	P14.1	o	0	LCD リセット
KEYIN1~5	P5.1~P5.5	i	—	キー入力。内蔵プルアップ。
KEYOUT1~5	P0.0, P0.1, P0.5, P0.6, P4.3	o / z	0	キー・スキャン出力
KEYOUTM1~5	P0.0, P0.1, P0.5, P0.6, P4.3	—	0	キー・スキャンの o / z 切り替え
KEYOUTU1~5	P0.0, P0.1, P0.5, P0.6, P4.3	—	0	キー・スキャン内蔵プルアップ切り替え
LED0~3	P6.2, P6.3, P14.6, P14.7	o	0, 0, 0, 1	LED 表示 (キーのシフト状態表示) 0 で点灯。
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	P12.0	i	—	送信許可入力。内蔵プルアップ。
UART_RTS	P4.1	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	P13.0	O	1	赤外受信, 温度センサの電源制御出力。
AD_TMP	P2.2	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,

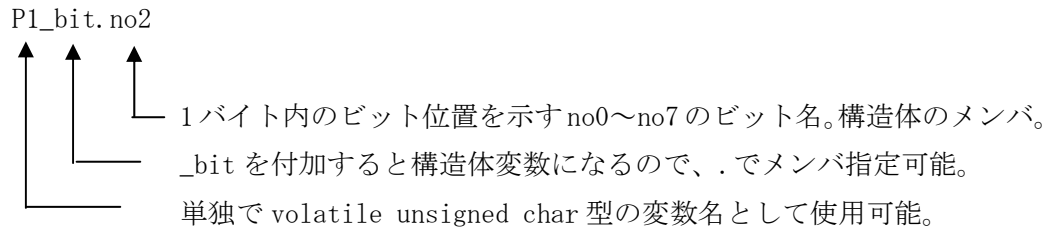
i : 入出力ポートを入力に設定, o : 入出力ポートを出力に設定,

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)

注: 上表の SFR 変数名の . は、_bit.no と読み替えてください。

(2) 複数ビットの一括操作

CC-RL では、1 バイトの SFR 変数名の多くが、`_bit.no` を付加することにより構造体メンバ変数として扱えます。



構造体名では、メンバの間接指定 (->) も使えます。

なお volatile とは C コンパイラが関知できない方法で値が変化するを意味します。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。

```
Pl_bit.no2 = 1;
while (Pl_bit.no2);
```

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、RL78 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 PM(x)_bit.no(y) を 1 から 0 に書き換える場合は、その直前に P(x)_bit.no(y) に出力したい値を代入する必要があります。

良い例：

```
P1_bit.no2 = 1;
PM1_bit.no2 = 0;
P1_bit.no5 = 1;
PM1_bit.no5 = 0;
```

誤動作する可能性のある例：

```
P1_bit.no2 = 1;
P1_bit.no5 = 1;
PM1_bit.no2 = 0;
PM1_bit.no5 = 0;
```

これは、P1_bit.no5 を操作したときに、P1 の中の他の入力設定（ここでは P1_bit.no2）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_bit.no0～P1_bit.no3 をプログラム A、P1_bit.no4 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_bit.no4 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ンネルから構成されており、一部の制御ビットは複数チャンネル分が 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いない）SFR もあるため、他のチャン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いない 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（T00）であれば、

```
set1    T00.1    → この記述はエラーになる
set1    !1B8H.1  → T00 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いないレジスタ（SAU の S0 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 of 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。

①TASK.h, KEY_close

これは筆者提供のドライバ群のヘッダ・ファイルです。

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

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 "TASK.h" /* 3.2.3① */
#include <stdio.h>
#include <time.h>
#define CLOCK() ( (int) clock() / (CLOCKS_PER_SEC / 100) )
int getkey(void);

int main (void) {
    int c, told;
    KEY_close(); /* 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 $prev_detect$ と一致していない場合は、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いないので、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 5×5 のマトリクス構成になっている点で、リスト 3-2⑤の部分で別の関数 KEY_scan になっています。

関数 KEY_scan は、本来であれば端子 $KEYOUT1 \sim 5$ を順番に 0 にして、そのつど端子 $KEYIN1 \sim 5$ 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 $KEYOUT1 \sim 5$ を全て 0 にして、 $KEYIN1 \sim 5$ を読み取ります。

$KEYIN1 \sim 5$ 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。

キーが押されていない場合は 0 を返します。

(2) 上記で番号が返ってきたら、端子 $KEYOUT1 \sim 5$ 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)。

$KEYIN1 \sim 5$ 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。

$$5(n - 1) + \text{“(1)で得た番号”}$$

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 $SW1 \sim SW25$ 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となります。

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、CA78K0R 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ヘ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ので、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(2012年8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 $1\mu\text{s}$ 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。

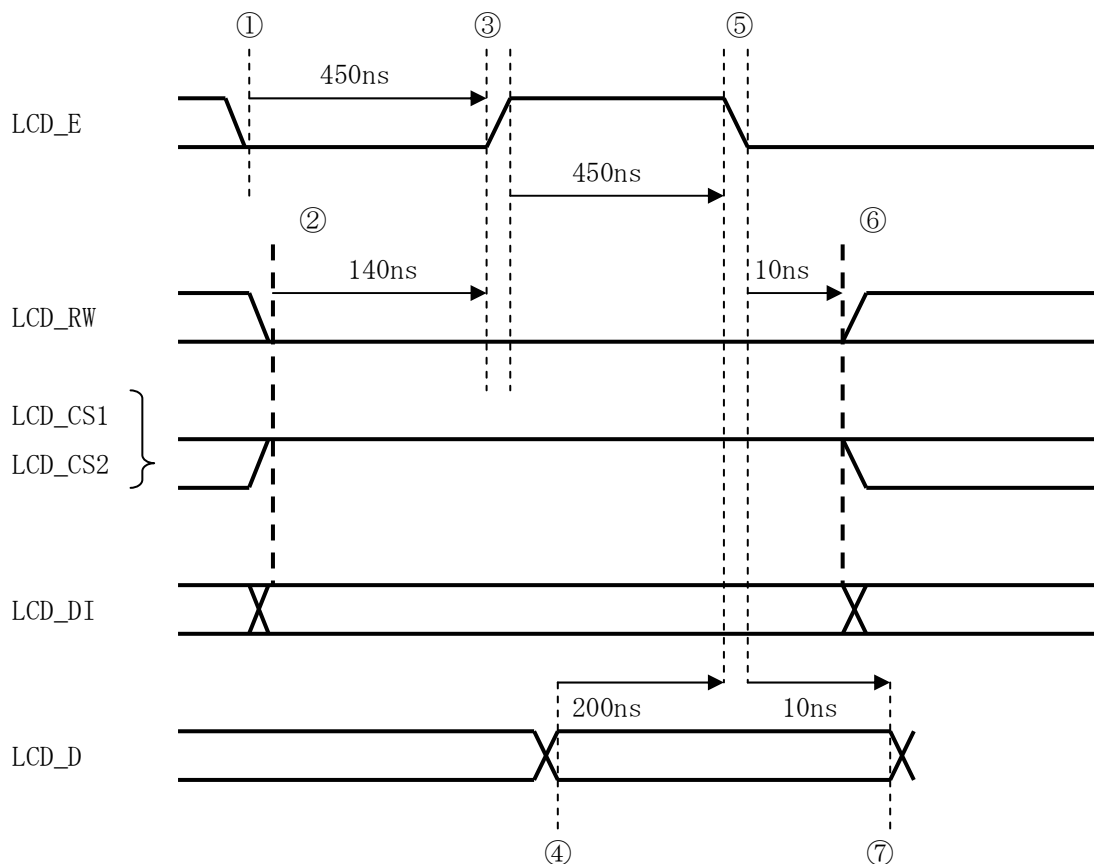
本ボードでは、ハードウェア初期設定関数(hdwinit)の中で0を出力していて、以降の処理では $1\mu\text{s}$ は十分経過しているの、単に

```
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; } SG12864A には LCD コントローラが 2 つ入っているの、制御
 LCD_CS2 = 1; } 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; } 書き込みが終了したら 0 に戻します。
 LCD_CS2 = 0; } 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、 $0.5[\mu s]$ にしてありますが、本版の wait_clock 関数は $1\mu s$ 以下の指定をしても $1\mu s$ 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 $6[\mu s]$ にしてあります。 $5[\mu s]$ でも動きますが、③、⑤の時間が将来設定通り $0.5[\mu s]$ 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 "TASK.h" /* ① */
#include <stdio.h>
#include <time.h>

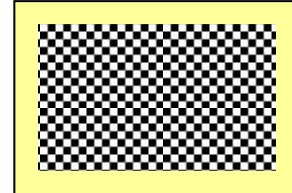
int main(void) {
    int i, j;
    KEY_close();
    while (!LCD_ready()); /* ② */

    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 ドライバの初期化を確認します。キーは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（0x55 と 0xAA の繰り返し）。
- 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。

- ②`bsearch` 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]`

3.5.2 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はタイマのチャンネル番号です。

(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、

$$\text{周期数} = 0.01 * \frac{(20 \times 10^6)}{16} = 12500$$

└─ 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に注意します。

(3) 動作開始

TSOL_bit.no(n) = 1; //注:(n)の部分がチャンネル番号

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

TTOL_bit.no(n) = 1; //注:(n)の部分がチャンネル番号

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。

これを割り込処理と言います。

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マニュアルのライブラリ関数仕様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、関数の宣言に先立って、次の記述を行います。

```
#pragma interrupt BTIMER_INT(vect=INTTM07, bank=RB1)
                ①                ②                ③
```

①自動実行する関数名

②割り込みベクタ・テーブル・アドレス

割り込みとの対応は、RL78/G13 ユーザーズ・マニュアル「割り込み機能」の章の割り込み要因一覧を参照してください。

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、割り込み処理に入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 RL78 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できます。

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。

```
void BTIMER_INT(void) {
    _____ 関数名。#pragma interrupt の指定と一致させます。
}
```

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ハードウェ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ハードウェア的にレジスタ・バンクを元に戻し、割り込まれるのを許可する。

(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、RL78 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す。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
INTTM07	定周期処理 (0.01 秒間隔)	3	1
INTRTC	時計表示 (0.5s(min), [予約])		
INTTM02	赤外線受信 (1.2ms(min)程度)	2	2
INTTM06	赤外線送信 (0.6ms(min)程度)		
INTSR2	UART 受信 (0.5ms(min)程度@19.2kbps)		
INTST2	UART 送信 (0.5ms (min) 程度@19.2kbps)	1	3
—	音声コーデック用 [予約]		

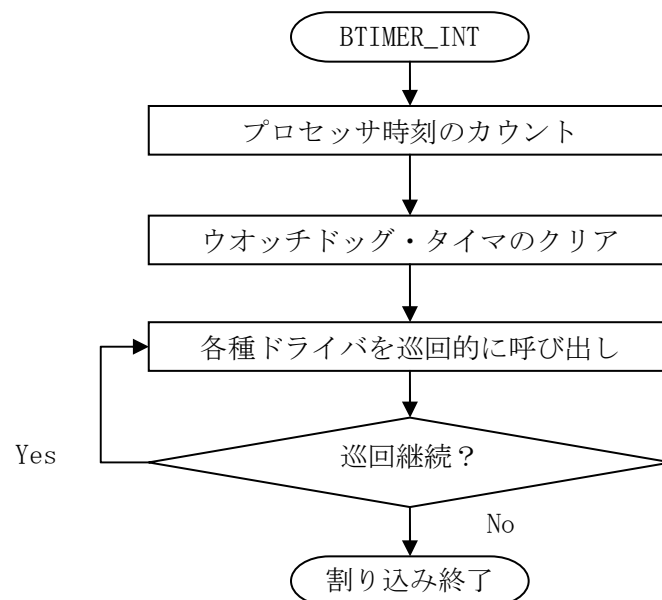
3.5.4 定周期ドライバ (BTIMER.c) 概説

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定周期処理 (BTIMER_INT 関数) から構成しています。

BTIMER_open 関数はドライバ初期設定関数 (drvinit 関数) から呼び出しています。

BTIMER_INT 関数では, まず C 言語標準のプロセッサ時刻のカウントと, ウォッチドッグ・タイマのクリアを行います。その後, 各種ドライバを巡回的に呼び出しています。

(下図は概念図)



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 3-6A 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 "PORT.h"
#include "TASK.h" /*①*/
#include <stdio.h>
#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メロディ演奏のドライバを使用するために TASK.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 チャン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 はチャンネル番号)。

(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、

$$\text{周期数} = \frac{20 \times 10^6}{16} / 1000 / 2 = 625$$

└─ 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 T00n 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。

TOL0L &= ~(1 << n); リセット初期値のままなら省略可能。注 1。

T00L &= ~(1 << n); 初期レベル不定か, 使用后 0 に戻すなら省略可能。注 1。

TOE |= 1 << n; 必須

TSOL |= 1 << n; 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 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, 端子 T00n 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 |= 1 << n;

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 &= ~(1 << n);

T00L &= ~(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 (= TDR0n + 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 "PORT.h"
#include "TASK.h"
#include <stdio.h>
#include <stdlib.h>    /* atoi 関数用 */

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。

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

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 \flat
1	ラ#・シ \flat	7	ミ
2	シ	8	ファ
3	ド	9	ファ#・ソ \flat
4	ド#・レ \flat	10	ソ
5	レ	11	ソ#・ラ \flat

ちなみにピアノの範囲は $m = -48 \sim +39$ 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 \flat	32141
ラ#・シ \flat	42903	ミ	30337
シ	40495	ファ	28635
ド	38223	ファ#・ソ \flat	27027
ド#・レ \flat	36077	ソ	25511
レ	34052	ソ#・ラ \flat	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 (ソラシーラソ, ソラシラソラー) 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 D0, 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) x 0.01 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 "PORT.h"
#include "TASK.h"
#include <stdio.h>
#include "BEEP_ONPU.h"

const ONPU melody[] = {
/* ソラシーラソ, ソラシラソラー */
/*00*/ {mSET(0, 1)}, {mSET(1, 1)},          /*①*/
/*02*/ { S0, T8 }, { LA, T8 }, { SI, T4T},  /*②*/
/*05*/ { LA, T8 }, { S0, T8 }, { mQF(T8)},  /*③*/
/*08*/ { S0, T8 }, { LA, T8 }, { SI, T8 },
/*11*/ { LA, T8 }, { S0, 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 3-7A 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 */
#include "PORT.h" /*①*/
#include "TASK.h" /*②*/
#include <stdio.h>
#include <time.h>
#include <math.h>
#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

①CA78K0R 用ドライバ・キットでは PORT.h の中に HALT 関数を使うための記述がありました。CC-RL では関数名が `__halt` となり宣言不要になりました。
`__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[%], ガンマ補正值

明滅は、sin 関数を使って、 $\text{センター値} + \text{振幅値} * \sin(t)$ 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、ガンマ補正值 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。

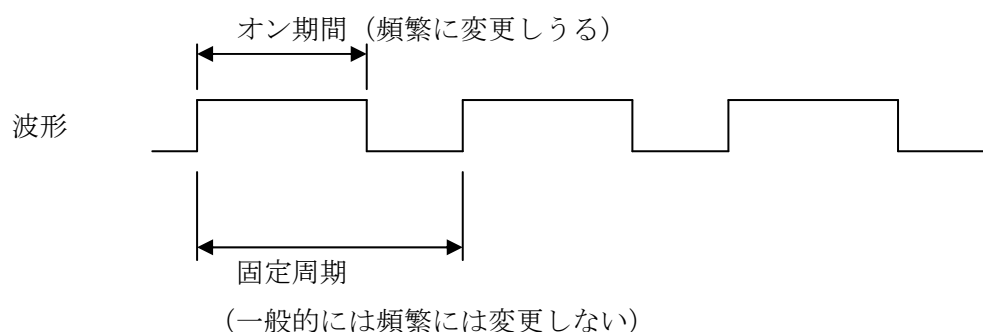
引数は、LED の明るさ 0~100%を 100 倍した数値 (0~10,000) を指定します。

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 つのチャンネルを使用します。固定周期を作る方をマスタ・チャンネル (以下チャンネル番号 m), オン期間を作るチャンネルをスレーブ・チャンネル (以下チャンネル番号 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、

$$\text{周期数} = (20 \times 10^6 / 16) / 280 = 4464$$

└ 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に注意します。

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。

T00L &= ~(1 << s); ‘H’ でオンの場合。注 1。

(T00L |= 1 << s; ‘L’ でオンの場合。) 注 1。

TOE |= (1 << s); 必須

TSOL |= 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 T00s 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 T00s 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

TTOL |= 1 << m | 1 << s; 注 1。
TOE &= ~(1 << s);
TOOL &= ~(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 が動作していない場合は、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 "PORT.h"
#include "TASK.h"
#include <stdio.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°C, %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°C, 27, 27

△

28°C, 28, 27

①温度測定ドライバを組み込むために PORT.h, TASK.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ンネル番号です。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 TEMPERATURE 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°C 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、特定のチャンネル（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）

```
ADM0 = 0x29;   セレクト・スキャン，変換速度 4.75 μs (20MHz 動作時)
```

```
ADM1 = 0x20;   ワンショット変換モード
```

他の変換モードにする場合は，RL78 のユーザズ・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 $1\mu\text{s}$ 以上待ちます。

(4) 端子設定，チャンネル選択

```
ADPC = d;      d は，アナログ入力可能な端子（P20～）の内，何本目から  
                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  
                ハードウェア初期設定で 4 に設定してあります。
```

```
ADS = ch;      ch はチャンネル番号です。P20 端子が 0 になります。温度センサ  
                （P22 端子）は 2 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
```

```
結果格納変数 = ADCR;  変換結果の保存。連続変換モードの場合で，変換時間  
                        以上の割り込みがかかる場合は次の変換が始まります  
                        が，ADCR の読み出しは可能です。
```

```
ADCS = 0;      変換終了（ワンショット変換モードでは自動的に 0 になる  
                ので不要）
```

(6) 停止

```
ADM0 = 0;      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、3.8.2 の手順で A/D 変換値を読み取っています。

もし、多少なりとも精度を上げたいければ、割り込み禁止にして測定開始 (ADCS = 1) した後、HALT 関数 (3.7.1 参照) を実行します。A/D 変換以外の要因で HALT 解除された場合 (ADIF が 0 のままの場合) は再測定 (再度 ADCS = 1) するようにします。

(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°C 電圧を以下のように指定します。

```
#define Vref_voltage    3300.0 /* Vref 端子電圧 [mV] */
                        (ただしコンパイラ・オプションの定義で VDD5V を指定した場合は 5000.0)
#define Coef_degree     10.0  /* 温度センサの温度係数 [mV/°C] */
#define Base_voltage    600.0 /* 温度センサの 0°C 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°C 単位に丸めた整数にしています。

3.9 時計機能 (RTC)

RL78にはカレンダー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。

実行結果 3-9A

(赤字は入力例)

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。

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)。

```

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

```

```

/* リスト 3-9A 時計の表示と設定*/
#include "PORT.h"
#include "TASK.h"
#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", &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RL78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・キットのドライバ初期設定DRVINIT.cから呼び出しているRTC_open関数では、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RL78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)。

HOURL = 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

筆者ハードウェアで周波数誤差を実測したところ、25°Cで +35ppm (月差 +92 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、 $-0.04\text{ppm}/\text{C}^2$ 程度の係数と思われます。

RL78 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°C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 (補正值 > 0) 補正值++;
    if (補正值 < -62) 補正值 = -62;
    if (補正值 > 62) 補正值 = 62;
    SUBCUD = 0x80 + (補正值 & 0x7f);
}
```

3.9.3 目覚まし時計

(1) アラーム機能の設定方法

RL78 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 RL78 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 */
#include "PORT.h"
#include "TASK.h"
#include <stdio.h>
#include <time.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 西曆%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", &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、RL78 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。BCD コードというのは、10 進数の 0~9 だけを 4 ビットであらわしたもので、C 言語では 1 桁なら `char` 型で扱えますが、多桁となると加減乗除などの算術演算ができません(大小比較のみは可能)。秒単位などの数値に換算すれば `long` 型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、RL78 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

本版では、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。筆者ハードウェアの実測値が +35ppm 程度だったので、誤差補正は -33ppm 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、“共通オプション” → “よく使うオプション (コンパイラ)” →

“定義マクロ”に `RTC_CHECK` を追加して、リビルドしてください。P141 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值が得られません。

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。

(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 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 : ハードウェア	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。

最近の Tera Term であれば 10 キー横の Enter キーも使用可能です。

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 "PORT.h"
#include "TASK.h"
#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)

```

Tera Term - COM4 VT
File Edit Setup Control
通信開始
test message
test message
簡素に極意あり
簡素に極意あり
end
通信終了

```

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（ $\backslash n$ ）が付加されますが、fputs 関数は付加されません。

1 バイトずつ送る場合は、putc 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（ $\backslash n$ ）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（ $\backslash n$ ）が捨てられますが、fgets 関数は改行（ $\backslash n$ ）も読み込みます。最後には gets と同じく $\backslash 0$ が付加されます。

ドライバ・キットの fgets 関数は、 $\backslash n$ が来るか引数 2 - 1 文字を読み込むまで NULL を返し続けます。

1 バイトずつ受信する場合は、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

RL78 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 I²C 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、RL78/G13 の場合の SAU1 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。

SAU1 は、シリアル・チャネル 4 本 (64 ピン以下の品種は 2 本で、ピン数により制約あり) 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 I²C、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)。

本書ではチャネル 0, 1 を使う UART2 を COM ポート接続用として割り当てています。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。

```
SAUIEN = 1;                ... SAU1 を有効にします。
__nop(), __nop(), __nop(), __nop();    ... RL78 では不要になりましたが、
                                         78K0R 互換のソースにする場合は 4 クロック待ちます。
```

```
SPS1 = PRS11 << 4 | PRS10;    ... 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 PRS11, PRS10 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 SAU1 の機能選択 (RL78/G13。80 ピン以上の場合)

チャネル	3 線シリアル	UART	簡易 I ² C
0	CSI20	UART2	IIC20
1	CSI21		IIC21
2	CSI30	UART3	IIC30
3	CSI31		IIC31

(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,

$$\text{マイコン動作周波数} / 2^m < 128$$

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

$$\text{分周値} = \text{マイコン動作周波数} / 2^m / \text{ボーレート}$$

を計算します。小数点以下は四捨五入して整数にします。

$$\text{SPS1} = \text{SPS1} \& \text{0xf0} | m - 1; \quad \text{PRS11 を使わないなら } \text{SPS1} = m - 1; \text{ で可。}$$

$$\text{SMR1s} = \text{0x0023}; \quad \text{PRS11 の場合は } \text{0x8000} \text{ を加算。}$$

$$\text{SCR1s} = \text{0x8097}; \quad \text{パリティ無しの設定の場合。}$$

$$\text{SDR1s} = (\text{分周値} - 1) \ll 9;$$

$$\text{SOL1} = \sim(1 \ll s); \quad \text{初期値から変更してなければ省略可。}$$

$$\text{SO1} = \text{0x0808} | 1 \ll s;$$

$$\text{SOE1L} |= 1 \ll s;$$

$$\text{TxDn 端子を出力に設定し } 1 \text{ を出力。} \quad \text{ただし初期設定済なら省略可能。}$$

$$\text{SS1L} |= 1 \ll s;$$

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。

$$\text{SMR1s} \&= \sim 1; \quad \text{送信完了を検出できるモードに設定。}$$

$$\text{while}(\text{SSR1s} \& \text{0x40}); \quad \text{送信完了待ち (TSF フラグ判定)。}$$

$$\text{ST1L} |= 1 \ll s; \quad \text{送信チャネル停止。}$$

$$\text{SOE1L} \&= \sim(1 \ll s); \quad \text{TxDn 端子を使用しない場合は省略可能。}$$

(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)

$$\text{SPS1} = \text{SPS1} \& \text{0xf0} | m - 1; \quad \text{PRS11 を使わないなら } \text{SPS1} = m - 1; \text{ で可。}$$

$$\text{SMR1s} = \text{0x0023}; \quad \text{送信側も設定必要。}$$

$$\text{SMR1r} = \text{0x0122}; \quad \text{PRS11 の場合は } \text{0x8000} \text{ を加算。}$$

$$\text{SCR1r} = \text{0x4497}; \quad \text{パリティ無しの設定の場合。}$$

$$\text{SDR1r} = (\text{分周値} - 1) \ll 9;$$

$$\text{NFEN0} |= 1 \ll (1 \ll 2 * n); \quad \text{受信ノイズ・フィルタのオン (必要に応じて)}$$

$$\text{SS1L} |= 1 \ll r;$$

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。

$$\text{ST1L} \&= \sim(1 \ll r); \quad \text{受信チャネル停止。}$$

$$\text{NFEN0} \&= \sim(1 \ll (1 \ll 2 * n)); \quad \text{受信ノイズ・フィルタのオフ (必要に応じて)}$$

(4) フロー制御

マイコンはパソコンに比べてメモリが少ないため、大量のデータを転送する場合は転送可能状態かどうか（ストリームに空きがあるか）を判断しながら転送します。

最も簡単な制御は、1パケットのサイズを決めておき、1パケット処理する毎に相手に応答を送って次の転送を許す方式です（ピンポン伝送方式）。

転送能力をもっと上げたい場合は、ストリーム（バッファ）の空き状態を監視しながらきめ細かく転送制御する必要があり、それをフロー制御と言います。以下の2種類があります。

①ハードウェア・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。

②Xon/Xoff制御（ソフトウェア・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 赤外線通信(ループバック) */
#include "PORT.h"
#include "TASK.h"
#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 バイトずつ送る場合は、putc 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, 失敗すると 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 関数は改行（\n）が来るか指定文字数を読み取るまでは NULL を返し続けます。赤外線通信の場合は、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）を読み落とし、かつストリームが満杯になってしまうと二度と入力が取れなくなってしまうます。

そこで getc 関数で 1 バイトずつ受信し、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。

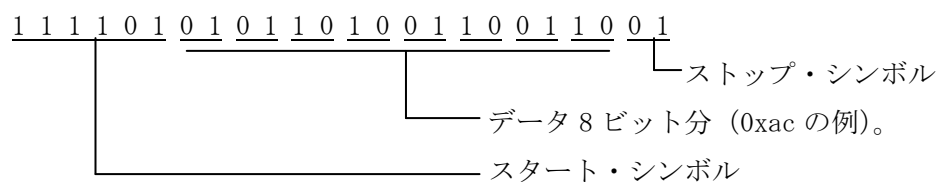
戻り値 : 成功すると 0, 失敗すると EOF。

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 : $0.6 \times 5 = 3.0\text{ms}$ (111101)

データ 0→1→0 変化 : $0.6 \times 4 = 2.4\text{ms}$ (011001)

データ 0→1→1 変化 : $0.6 \times 3 = 1.8\text{ms}$ (011010)

データ 1→0 変化 : $0.6 \times 3 = 1.8\text{ms}$ (1001)

データ 1→1 変化 : $0.6 \times 2 = 1.2\text{ms}$ (1010)

データ 0→0 変化 : $0.6 \times 2 = 1.2\text{ms}$ (0101)

実際の判定は、 $\pm 0.3\text{ms}$ 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。

3.11.4 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、ストリームに文字列分の空きがない場合は空くまで NULL を返し続けます。

putc 関数はストリームが満杯になったら EOF を返すので、その時点でユーザ・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くもありません。