2019 年 5 月 15 日　第 1 版

組込みソフトの安全設計

ー基礎から二足歩行ロボットによる実践まで

付録

ソースコードの説明の補足

この資料では、C言語に馴染みのない方のために、書籍に掲載したプログラムの文法等の説明をします。C言語のプログラミングの理解の一助にしてください。説明が書籍と重複する部分もあります。図や表の番号は書籍と異なりますが、書籍中の番号を図のタイトルの後の（）の中に示します。新しく掲載した図や表はタイトルの後の（）の記述がありません。

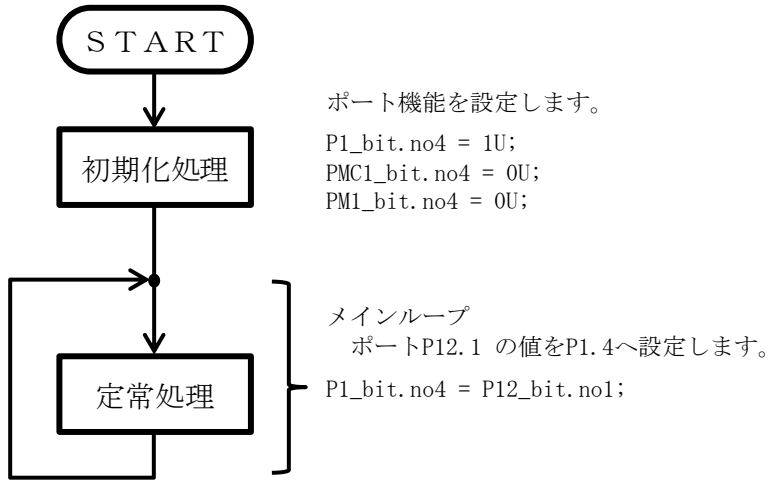
第1章　導入

1.5 プログラミングの肩慣らし

スイッチを押している間だけLEDランプが点灯するプログラムを説明します（**ソースコード 1.1**）。このプログラムは、ワンチップマイコンのプログラミングに必ず出てくるメイン処理を紹介するものです。（ファイル名：maincc）

【メイン処理】

メイン処理は初期化処理をする部分と、定常処理を行うメインループからなる部分とで構成されます（**図 1.1**）。初期化処理では、マイコンハードウェアの設定などをします。メインループでは、マイコンが動作している間、一連の処理を常に繰返し実行します。入力変化やタイマのタイムアップなどのイベントに対して、処理が多少遅れても良い（数msから数十ms程度）ものに適用されます。人の反応速度に比べてマイコンの処理速度が速いため、多くの処理はメインループで実行することで対処できます。



■図 1.1 ワンチップマイコンのプログラムの基本構造（図 1.7）

ソースコード 1.1 において、メインループの中の処理は、スイッチの状態を LED の出力としています。つまり、スイッチが押される(P12.1 → Low)と LED を点灯(P1.4 → Low)しています。このプログラムは、おもちゃの二足歩行ロボットの制御基板で実行できます。

■ ソースコード 1.1 LED 点灯プログラム

```
/* main.c */

/* 特殊機能レジスタ（SFR）へのアクセス記述を使用する */
#include "iodefine.h"

/* 関数プロトタイプ宣言 */
void main(void);

void main(void){
    /* ポートの設定(P10～P13:出力(LED)、P14:入力(LED) */
    P1_bit.no4 = 1U;                    /* ポート 14 の出力を High(1)に設定 */
    PMC1_bit.no4 = 0U;                  /* ポート 14 をデジタル入出力に設定 */
    PM1_bit.no4 = 0U;                   /* ポート 14 を出力に設定 */

    while(1){                           /* メインループ */
        /* スイッチが押される(P14 → Low)とＬＥＤが点灯(P121 → Low) */
        P1_bit.no4 = P12_bit.no1;
    }
}
```

以下に、ソースコードにおける処理内容やＣ言語の基本的な文法等の解説をします。

/* 特殊機能レジスタ（SFR）へのアクセス記述を使用する */

これはコメント（注釈）です。コメントは「/*」で始まり「*/」で終わります。プログラムの実行には影響を与えません。

プログラマがプログラムの意図を記録しておくものです。以後も多く出てきます。

#include "iodefine.h"

#include ″iodefine.h″ はマイコンの周辺回路を利用するための特殊機能レジスタ（SFR）を記号で指定できるようにします。

ソースコード中で、マイコンの「ユーザーズマニュアル　ハードウェア編」で記載している記号とほぼ同じ記号で指定できます。このプログラムでは P1_bit.no4、PMC1_bit.no4、PM1_bit.no4、P12_bit.no1 などです。

#include は指定するファイル（iodefine.h）の内容をここに取込み展開します。

【プリプロセッサ指令とは】

#include などの「#」で始まる文はプリプロセッサ指令と言います。

文字の置き換えやファイルの取り込みなどの処理を行います。プリプロセッサ指令は、コンパイルする前に、翻訳前処理で処理されます。

【特殊機能レジスタとは】

特殊機能レジスタはSFR（Special Function Register）と言います。

SFR はマイコンに付加機能として搭載されている入出力ポートやタイマなどの周辺ハードウェアを制御するレジスタです。本マイコンではSFRをメモリ領域に割り付けています。そのため、変数のアクセスと同様に扱えます。ピリオド(.)を用いることによりビット単位でアクセスできるものもあります。

void main(void);

これは、メイン関数の関数プロトタイプ宣言です。このファイルに記述した関数が使用する関数の宣言です。コンパイラが関数呼出しを機械語に変換しようとしたときに、その関数の引数と戻り値の情報が必要になります。コンパイラは先頭行から順に解析していくので、ある関数が定義される前に他の関数がある関数を呼び出そうとしたとき、ある関数の引数と戻り値が分からないとコンパイルできなくなります。そのため、関数のプロトタイプ宣言をファイルの先頭に置くことにより、コンパイラが関数の戻り値と引数の型を認識することができるようにします。関数の外で定義された変数を関数が使用する場合も、変数をファイルの先頭で宣言します。メイン関数は他の関数により呼び出されないので、プロトタイプ宣言をしなくてもコンパイルできますが、念のためプロトタイプ宣言しておきます。

宣言は関数外で定義された変数や関数を利用しますという宣言のみで、関数の処理内容や変数の格納場所などの実体をメモリ上に配置しません。

void main(void){

この記述は、ここからメイン関数であることを示します。

先頭の void は、この関数の呼出し元に返す値がないことを示します。main は識別子と言い、関数の名前を表わします。main 関数（メイン関数）はプログラムが開始されるときにのみ実行される関数です。C 言語では最初に実行される関数であると規定されています。基本的に関数は他の関数から呼出されて実行されますが、メイン関数はC言語で書かれた他の関数から呼出されることはありません。メイン関数はアセンブラで書かれたスタートアップルーチンから呼出されますが、CS+ ではスタートアップルーチンをプログラマが記述しなくても済みます。

関数には、名前の後に必ず()が付きます。ここでは()の中が void となっています。これは呼出し元からこの関数に引き渡す値（引数）がないことを示します。

()の次の{}で囲まれた中に main 関数の処理の内容を記述します。{}で囲まれた部分をブロックと言います。ブロック内の文は、そのブロックの外側の文より字下げすることでプログラムの構造が分かりやすくなります。(字下げ＝インデント、indentation)

【識別子とは】

識別子は変数や関数などに付ける名前を言います。

識別子は非数値文字で始まる英数字で構成されます。大文字と小文字を区別し、異なる識別子として扱います。下線（_）も使用できます。識別子の先頭を下線にすることもできますが、先頭に下線が付く識別子はライブラリ等で使用していることがあるので注意して下さい。識別子は先頭から 31 文字までがコンパイラで認識されます。ANSI 規格で規定されています（ANSI：American National Standards Institute)。for や if などの C 言語があらかじめ定めている言葉（予約語）は使用できません。ただし、予約語を含む言葉 forward などは使用できます。

【関数とは】

関数はある機能を実現する処理をまとめたものです。

C 言語のプログラムはいくつかの関数で作られます。基本的に関数は他の関数から呼び出されて実行されます。関数の名前の前に、呼出し元に返す値の型（int,char,void など）を記載します。関数は名前の後に必ず()が付きます。()の中には、呼出し元からこの関数に引き渡す値（引数）を記載します。引数は「,」で区切って複数指定することができます。

関数の処理の内容を記述したものを関数定義と言います。関数定義に書く引数を仮引数と言います。関数の呼出し側に書く引数を実引数と言います。配列を除き、呼出し側で指定した実引数の値をコピーして仮引数として使いますので、配列とあとで述べるポインタを除いて呼び出し元で指定した実引数の変数の値を変えることはありません。()の次の{}で囲まれた中に main 関数の処理の内容を記述します。引数があれば、その中で引数を利用します。関数定義では、実際のプログラム（実体）をメモリ上に配置します。呼出し元へ返す値は return の後の式（変数等）で指定します。return 文で呼出し元に返ります。呼出し元に返す値の型が(void)の場合などでは、return 文を記述しない場合があります。

```
P1_bit.no4 = 1U;
```

LED が接続されているポート(P14)へ 1 を出力し、出力が High となり LED を消灯します。

P1 はポートレジスタ 1 を表します。P1_bit.no4 はポートレジスタ 1 の 5 ビット目のビットを表します。ビットは 0 ビット目から始まります。P14 は LED が接続されている端子（13 番ピン）となります。= は右側の値を左側に代入（設定）する 代入演算子です。1U の U は定数が符号(±)のない整数定数であることを表す接尾語です。このマイコンでは、符号なし整数定数は 0 から 65535 までの値を設定できます。1U をビットイメー

ジで表すと次のようになります。

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
3	1														
2	6	8	4	2	1										
7	3	1	0	0	0	5	2	1							
6	8	9	9	4	2	1	5	2	6	3	1				
8	4	2	6	8	4	2	6	8	4	2	6	8	4	2	1
の	の	の	の	の	の	の	の	の	の	の	の	の	の	の	の
位	位	位	位	位	位	位	位	位	位	位	位	位	位	位	位

■図 1.2 1U のビットイメージ（図 1.8）

;

セミコロン(;)は文の終わりを示します。 P1_bit.no4 = 1U; は 1U の１の位のビットを P14 に代入することになります。 P1_bit.no4 = 1U; とすると、出力が High となり LED が消灯します。おもちゃの二足歩行ロボットで使用する回路では出力が Low で LED が点灯します。

【文とは】

プログラミングにおける指示命令の単位を文といいます。

C 言語では、文の終わりにセミコロン「;」を置きます。改行は空白と同じように扱われ文の終わりとされません。ただし、「#」で始まる文（プリプロセッサ指令）は改行で終わります。プリプロセッサ指令は、コンパイルする前に、文字の置き換えやファイルの取り込みなどの前処理を行う文です。

【ポートとは】

マイコンでは、入出力端子をポートと呼びます。

多くの場合、8 ビット単位で管理しています。（ただし、8 ビットすべてに端子が割り付けられている訳ではありません）今回のマイコンでは P1、P2、P4、P6、P12、P13 のポート群があります。一つの端子の指定方法は P1_bit.no4 などです。

【代入演算子とは】

代入演算子（＝）は右側の値（演算結果を含む）を左側の変数に代入することを表わします。数学などで使う等号とは異なります。そのため、実行する前に変数 x が 5 であったとすると、以下の式は実行後に x が 6 になります。

x ＝ x ＋ 1 ;

PMC1_bit.no4 = 0U;

ポート(P14)をデジタル入出力として使うことを設定します。

ポート(P14)をアナログ入力として使う場合は PMC1_bit.no4 = 1U;とします。PMC1 はポート・モード・コントロール・レジスタ 1 を表します。(ポートをデジタル入出力として使用するか、アナログ入力として使用するかを切替えるレジスタ)

PM1_bit.no4 = 0U;

ポート(P14)を出力として使うことを設定します。

ポート(P14)を入力として使う場合は PM1_bit.no4 = 1U;とします。PM1 はポート・モード・レジスタ 1 を表します。(ポートを入力として使用するか、出力として使用するかを切替えるレジスタ)

while(1){

while は()の中が真（0 以外）であれば、{}の中の文を繰返し実行します。偽（0）であれば{}の中を実行せず、次の文に実行が移ります。

()の中が 1 ですので{}の中の文を、上から下へ永久に繰り返します。これがメインループとなります。

P1_bit.no4 = P12_bit.no1;

スイッチが接続されたポート P121 の値(ビット)を LED が接続されたポート(P14)へ設定します。

そのため、スイッチを押すと LED が点灯します。(入力が Low になると出力が Low になり LED が点灯する)

第 3 章　ソフトウェア開発の効率化と信頼性向上

3.1　ソフトウェア開発における課題

機能が混在するプログラムの例を**ソースコード 3.1** に示します。

（ファイル名：spaghetticc）

このプログラムはスイッチを押すごとに LED が消灯、点灯、点滅と切り替わるものです。このプログラムも、おもちゃの二足歩行ロボットの制御基板で実行できます。

①で示した部分でスイッチ入力機能、状態制御機能、LED 表示制御機能が混在しています。

■ ソースコード 3.1　機能が混在するプログラムの例

```
/* main.c */

/* 特殊機能レジスタ（SFR）へのアクセス記述を使用する */
#include "iodefine.h"

/* 12 ビット・インターバル・タイマ割込み処理割付 */
#pragma interrupt it_interrupt(vect = INTIT)

#define ONOFF_TIME_05S 100U                    /* LED 点滅時間：ON と OFF 各 0.5 秒 */

static unsigned char flag_5ms = 0U;            /* 5ms 経過フラグ */
void main(void);                               /* 関数プロトタイプ宣言 */

void main(void){
    __DI();                                    /* 割込みを禁止する */

    /* ポートの設定(P121:入力(ｽｲｯﾁ)、P14:出力(LED)) */
    P1_bit.no4 = 1U;                           /* ポート 14 の出力を High(1)に設定 */
    PMC1_bit.no4 = 0U;                         /* ポート 14 をデジタル入出力に設定 */
    PM1_bit.no4 = 0U;                          /* ポート 14 を出力に設定 */

    /* インターバル・タイマの設定（5 ms 周期） */
    TMKAEN = 1U;                               /* クロック供給 */
    /* 低速オンチップ・オシレータ・クロック（fIL）供給 */
    OSMC = 0x10U;
    ITMC = 0x004AU;                            /* 割込み周期 5 ms = [1/15kHz]×(74 + 1) */
    TMKAMK = 1U;                               /* INTIT 割込みの禁止 */
    TMKAIF = 0U;                               /* INTIT 割込み要求フラグのクリア */
    TMKAPR0 = 1U;                              /* INTIT の割込み優先順位をレベル 3 に設定*/
    TMKAPR1 = 1U;                              /*（最低優先順位）*/

    /* インターバル・タイマ開始 */
    TMKAIF = 0U;                               /* INTIT 割込み要求フラグのクリア */
    TMKAMK = 0U;                               /* INTIT 割込みの許可 */
    ITMC |= 0x8000U;                           /* カウンタ動作開始 */

    __EI();                                    /* 割込みを許可する */
```

```
    while(1){                                   /* メインループ */
        static unsigned char switch_tmp = 1U;   /* スイッチ入力の一次記憶値 */
        static unsigned char switch_level = 1U; /* スイッチ入力の確定値 */
        static unsigned char f_led_blink = 0U;  /* 点滅状態フラグ */
        static unsigned int counter_5ms = 0U;   /* 点滅用カウンタ(5 ms 単位) */

        /* 5ms 周期となるように待つ */
        while(flag_5ms == 0U){                  /* 5 ms 経過していなければループ */
        }
        flag_5ms = 0U;                          /* フラグをクリア */
        switch_tmp = (switch_tmp << 1) + P12_bit.no1;/* スイッチ入力の取り込み */
        if(switch_tmp == 0xFFU){                /* スイッチ入力 8 回連続 1 の場合 */
            switch_level = 1U;                  /* スイッチレベル 1 の確定 */
        }else if(switch_tmp == 0x00U){          /* スイッチ入力 8 回連続 0 の場合 */
            if(switch_level == 1U){             /* スイッチ押下（前回 OFF） */
                if(f_led_blink == 1U){          /* LED 点滅の場合は消灯にする */
                    f_led_blink = 0U;           /* LED 点滅フラグ OFF */
                    P1_bit.no4 = 1U;            /* LED 消灯 */
                }else if(P1_bit.no4 == 1U){     /* LED 消灯の場合は点灯にする */
                    f_led_blink = 0U;           /* LED 点滅フラグ OFF */
                    P1_bit.no4 = 0U;            /* LED 点灯 */
                }else{                          /* LED 点灯の場合は点滅にする */
①                   f_led_blink = 1U;           /* LED 点滅フラグ ON */
                    counter_5ms = 0U;           /* 点滅用カウンタをクリア */
                }
            }
            switch_level = 0U;                  /* スイッチレベル 0 の確定 */
        }
        if(f_led_blink == 1U){                  /* 点滅の実行 */
            if(counter_5ms > 0U){               /* カウンタが 0 より大きければ */
                counter_5ms = counter_5ms – 1U;/* カウンタをカウントダウン */
            }else{                              /* カウンタが 0 になったら */
                P1_bit.no4 = ~P1_bit.no4;       /* LED 出力を反転（消灯⇔点灯） */
                counter_5ms = ONOFF_TIME_05S;/* カウンタを 0.5 秒に設定 */
            }
        }
    }
}

/* インターバル・タイマ割込み処理（5 ms 周期） */
static void it_interrupt(void){
    flag_5ms = 1U;                              /* 5 ms 経過毎にフラグを立てる */
}
```

ソースコード 3.1 のメインループの部分を機能別の記述に変えたものが、**ソースコード 3.2** です。（ファイル名：functionalcc）

■ ソースコード 3.2　機能別の記述に変えたプログラムの例

```
while(1){                                   /* メインループ */
    static unsigned char switch_tmp = 1U;  /* スイッチ入力の一次記憶値 */
    static unsigned char switch_level = 1U; /* スイッチ入力の確定値 */
    static unsigned char switch_level_before = 1U;/* 前回のスイッチ入力の確定値 */
    static unsigned char flag_switch_on = 0U;/* スイッチ押下フラグ */
    static unsigned int counter_5ms = 0U;  /* カウンタ(5 ms 単位) */
    enum STATELED {LED_OFF = 0,LED_ON,LED_BLINK};/* LED の状態の列挙を宣言 */
    static enum STATELED e_state_led = LED_OFF;/* LED の状態変数を定義し消灯に */

    /* 5 ms 周期となるように待つ */
    while(flag_5ms == 0U){                  /* 5 ms 経過していなければループ */
    }
    flag_5ms = 0U;                          /* フラグをクリア */
①  /* チャタリング除去 */
    switch_tmp = (switch_tmp << 1) + P12_bit.no1;/* スイッチ入力の取り込み */
    if(switch_tmp == 0xFFU){                /* スイッチ入力 8 回連続 1 の場合 */
        switch_level = 1U;                  /* スイッチレベル 1 の確定 */
    }else if(switch_tmp == 0x00U){          /* スイッチ入力 8 回連続 0 の場合 */
        switch_level = 0U;                  /* スイッチレベル 0 の確定 */
    }

    /* スイッチ押下判定 */
    if(switch_level != switch_level_before){/* スイッチ確定レベルが変化 */
        switch_level_before = switch_level;/* 前回スイッチレベルの更新 */
        if(switch_level == 0U){             /* High から Low に変化 */
            flag_switch_on = 1U;            /* スイッチ押下確定 */
        }
    }

    /* 状態制御 */
    switch(e_state_led){                    /* LED の状態 */
        case LED_OFF:                       /* 消灯の場合 */
            if(flag_switch_on == 1U){       /* スイッチ押下 */
                flag_switch_on = 0U;
                e_state_led = LED_ON;/* LED の状態を点灯へ */
            }
            break;
        case LED_ON:                        /* 点灯の場合 */
            if(flag_switch_on == 1U){       /* スイッチ押下 */
                flag_switch_on = 0U;
②               counter_5ms = 0U;           /* カウンタを 0.5 秒に設定 */
                e_state_led = LED_BLINK;/* LED の状態を点滅へ */
            }
            break;
        case LED_BLINK:                     /* 点滅の場合 */
            if(flag_switch_on == 1U){       /* スイッチ押下 */
                flag_switch_on = 0U;
                e_state_led = LED_OFF;/* LED の状態を消灯へ */
            }
            break;
        default:
            break;
    }
```

```
③   /* LED 表示制御 */
    switch(state_led){                  /* LED の状態 */
        case LED_OFF:                   /* 消灯の場合 */
            P1_bit.no4 = 1U;            /* LED 消灯 */
            break;
        case LED_ON:                    /* 点灯の場合 */
            P1_bit.no4 = 0U;            /* LED 点灯 */
            break;
        case LED_BLINK:                 /* 点滅の場合 */
            /* 0.5 秒毎に LED の点灯・消灯を切替える */
            if(counter_5ms > 0U){       /* カウンタが 0 より大きければ */
                /* カウンタをカウントダウン */
                counter_5ms--;
            }else{                      /* カウンタが 0 になったら */
                /* LED 出力を反転（消灯⇔点灯） */
                P1_bit.no4 = ~P1_bit.no4;
                /* カウンタを 0.5 秒に設定 */
                counter_5ms = ONOFF_TIME_05S;
            }
            break;
    }
}
```

以下に、ソースコードにおける処理内容や C 言語の基本的な文法等の解説をします。1章で説明した内容は省きます。

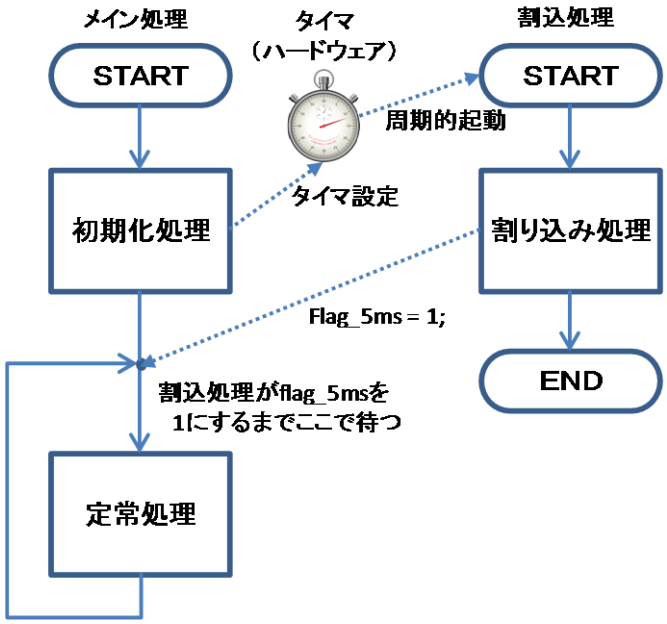
【メインループの定周期化について】

本サンプルプログラムではメインループを定周期にしています。ワンチップマイコンのプログラムは、必ずしもメインループを定周期にする必要はありません。しかし、定周期にすることで、処理ごとにハードウェアタイマを用いて個別にタイミングを作成しなくても良くなることがあります。そのため、メインループを定周期にすることがよく行われます。

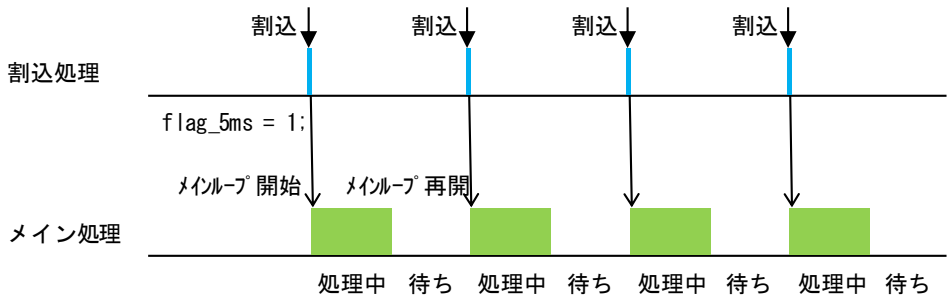
周期は、数 ms から数十 ms の間にすることが多いようです。処理間隔が長すぎると、操作に対する応答が遅くなり操作する人に操作性が悪いという印象を与えます。処理間隔が短すぎると、処理間隔以内にメインループの処理が終わらず、周期が伸びてしまい、メイン周期でタイミングをとっている処理のタイミングがずれてしまいます。例えば、時間をメイン周期で計数し計測する処理では、タイマの時間がずれてしまうことがあります。

メインループを定周期にする方法はいくつかありますが、ここでは、定周期で割込みをする 12 ビット・インターバル・タイマを使用します。割込み処理では 5 ms 毎にフラグを立てます（1 にする）。メインループの先頭ではフラグが 0 の間は待機し、フラグが 1 になったらクリアし（0 にする）、メインループ内の処理を実行するようにします。処理が完了すれば再びメインループの先頭で待機します。このようにして、5 ms 毎にメ

インループ内の処理を実行することができます。図 3.1 にメイン処理と割込み処理のフローチャートを、図 3.2 にメイン処理と割込み処理の実施タイミングを示します。



■図 3.1 メイン処理と割込み処理のフローチャート（図 3.2）



■図 3.2 メイン処理と割込み処理の実施タイミング（図 3.3）

【割込み処理について】

割込み処理は、メインループとは別に、タイマのタイムアップや通信の受信完了などのイベントにより一時的に割込みんで実行する処理です。ワンチップマイコンの多数ある周辺回路の機能として実行されます。

割込み処理は、イベントに対して素早く処理する（数μｓ 程度以内）ことが要求される処理に適用します。そうでない処理は、メインループで処理するようにした方が良いでしょう。割込みを多用すると、割込み処理が頻発して、他の処理の実行を妨げたり、割込み処理とメイン処理や異なる割込み処理同士間で共通に使用する変数の処理などで、トラブルが発生したりするリスクが増加するからです。(トラブルの例：メイン処理が該

当変数を処理している途中で割込み処理がその変数を書き換えてしまって、メイン処理の演算結果が不正になる）

また、割込み処理に時間を掛けすぎると、他の処理の実行が遅れるなどの悪影響を及ぼすことがあるので、割込み処理では必要最小限の処理にします。

ここではメイン周期を定周期にするため 12 ビット･インターバル･タイマを使用します。そこで参考までに、定周期で割込みをする 12 ビット・インターバル・タイマの設定の初期化について説明します。詳細は、マイコンのハードウェアマニュアルを参考にしてください。ソースコード 3.1 はこの手順で 12 ビット・インターバル・タイマの設定を初期化しています。

12 ビット・インターバル・タイマの設定の初期化手順

① 割込みの禁止

② 12 ビット・インターバル・タイマ回路全体の動作クロックを供給します。（TMKAEN = 1U;）

③ 12 ビット・カウンタのカウントクロックを供給可能にします。（WUTMMCK0 = 1U;）

④ インターバルタイマ・コントロール・レジスタの下位 12 ビットに割込み周期に対応するカウント値を設定します。

⑤ 割込みの設定をします。実際には設定とカウント動作開始で以下のような設定をします。

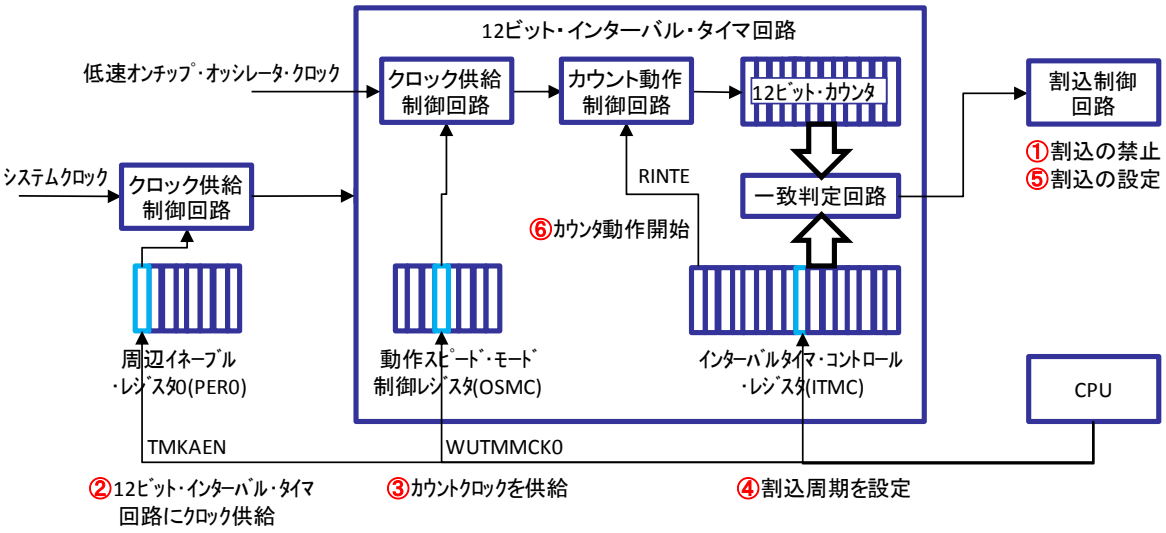
割込みフラグのクリア

割込み優先順位の設定

割込みの許可

⑥ カウント動作を開始します。（RINTE = 1U;）

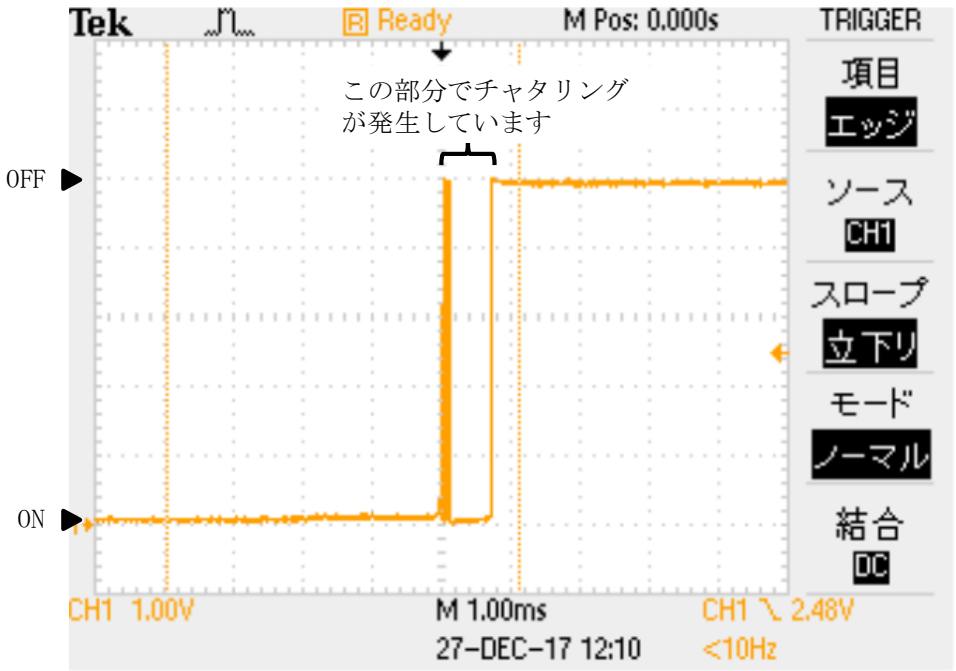
上記初期化手順でどのようなことをしているのか分かるようにするために、**図 3.3** の 12 ビット・インターバル・タイマ回路のブロック図に、上記初期化手順に対応した番号を記載します。



■図 3.3 12 ビット・インターバル・タイマのブロック図

【チャタリングについて】

チャタリングとはスイッチやリレーの接点が開閉するときに、短い期間にオン・オフを繰り返す現象を言います。機械的構造やスイッチの押し方等で異なりますが、長いものでは数十 ms になることもあります。図 3.4 にスイッチをオフするときに発生したチャタリングの状況を示します。マイコンは処理速度が速いので、チャタリングが発生するとスイッチが複数回押されたと判断し処理をするので、スイッチを操作した人にとって思いがけない動作となることがあります。そのため、ソースコード 3.2 のようなチャタリング除去の処理を入れます。



■図 3.4 スイッチのチャタリング発生状況

#pragma interrupt it_interrupt(vect = INTIT)

12 ビット・インターバル・タイマ割込み(INTIT)の処理関数名を it_interrupt とします。

interrupt はコンパイルオプションの割込み処理であることを示します。

vect = INTIT は、この関数の実行開始番地を、12 ビット・インターバル・タイマ割込み発生時に実行を開始する番地を指定する、ベクタテーブルに設定することを示します。INTIT は iodefine.h で定義されていますので、この文の前に、#include ″iodefine.h″が必要です。

#define ONOFF_TIME_05S 100U

LED 点滅時に消灯・点灯時間 0.5 秒をカウントする値(100)を ONOFF_TIME_05S と記述できるようにします。U は定数が符号なし整数であることを表す接尾語です。

これはマクロ定義と呼ばれ、#define に続く文字列をその後に続く文字列に置換えます。

マクロ定義をした次の行から置換えが有効になります。

static unsigned char flag_5ms = 0U;

5ms 経過したことを示す変数 flag_5ms を宣言･定義し､0 を代入し初期化しています。

static は静的変数で、unsigned は符号がなく、char は 8ﾋﾞｯﾄであり、flag_5ms は変数の名前であることを示します。この変数では 0～255 までの値が記録できます。C 言語の仕様では static の付く変数は 0 で初期化されることになっていますが､初期化忘れでないことを明示するため 0 を代入しています。

静的変数はプログラムの開始から終了まで記憶場所が固定されます。そのため、新たに値を代入しなければ値は保持されます。初期化はプログラムの開始時点で 1 回のみ行われます。

メイン関数と割込み関数の両方で使用するためブロック{}外で宣言しています。ブロック外で宣言した静的変数はファイル内の関数で使えます。同じファイルの異なる関数の間で情報を受け渡すために使います。

0	0	0	0	0	0	0	0
1							
2	6	3	1				
8	4	2	6	8	4	2	1
の	の	の	の	の	の	の	の
位	位	位	位	位	位	位	位

■図 3.5 flag_5ms のメモリーイメージ

【char の読み方】

char は英語ではチャーと発音します。

以下の理由で日本のプログラマはキャラと呼ぶ人も多いようです。

- C 言語の char は character（文字） の略である。
- 技術評論社の ANSI C 言語辞典ではキャラとしている。
- 英語で char の意味は炭であり、チャーに抵抗感がある。

ちなみに int は英語ではイントと発音します。

__DI();

割込みを禁止します。(DI:disable interrupt)

割込みに関する設定を変更している間は割込みを禁止するため__DI();を記載しました。

この後に続く文で、割込みに関する設定を変更しています。

割込みに関する設定を変更している途中で割込みが発生すると、中途半端な設定になり暴走する可能性があるため、割込みを禁止しています。ハードウェアの設定を初期化する場合は、特別な理由がない限り割込みを禁止して行った方が良いでしょう。

__DI()は組込み関数といい、アセンブリ言語の di に置換します。簡易にアセンブリ命令を記述するため用います。

TMKAEN = 1U;

12 ビット・インターバル・タイマへクロックを供給します。

TMKAEN は iodefine.h で定義されていますので、この文の前に、#include ″iodefine.h″が必要です。この後のハードウェア設定に使われている識別子も同様です。1 ビットのビットフィールドの変数として定義されています。ビットフィールド変数については、7 章で説明します。

OSMC = 0x10U;

低速オンチップ・オシレータ・クロック（fIL = 15kHz）を供給します。

OSMC は符号なし char 型変数として定義されています。

0x10U の 0x はその後に続く文字が 16 進数であることを表します。16 進数は 0,1,2,・・・,9,A,B,・・・,F の 16 文字で表されます。

ITMC = 0x004AU;

割込み周期を 5 ms に設定します。 [1/15kHz]×(74 + 1)

TMKAMK = 1U;

12 ビット・インターバル・タイマ割込みを禁止します。

割込みを許可するには 0U を設定します。

TMKAIF = 0U;

12 ビット・インターバル・タイマ割込み要求フラグをクリアします。

TMKAPR0 = 1U;

TMKAPR1 = 1U;

12 ビット・インターバル・タイマの割込み優先順位をレベル 3（最低優先順位）に設定します。

ITMC |= 0x8000U;

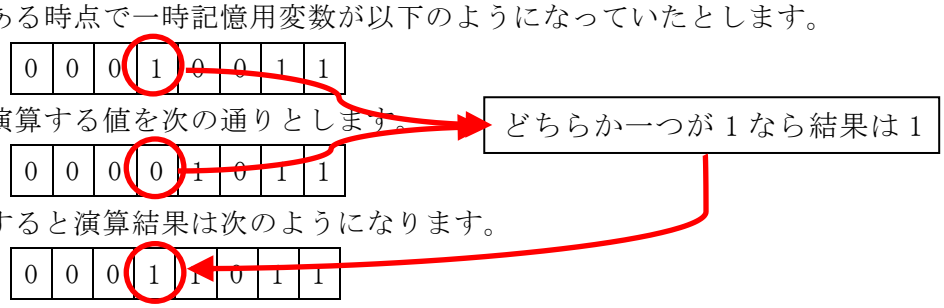
12 ビット・インターバル・タイマのカウント動作を開始します。

ここより前の特殊機能レジスタ（SFR）の読み書きは 1 ビット単位で行うことができました。しかし、インターバル・タイマ・コントロール・レジスタ（ITMC）は 16 ビット単位で読み書きする必要があります。ITMC の最上位ビットを 1 にするとカウント動作が開始されます。

|= の | はビット毎に OR 演算（演算される 2 つのビットがどちらか一つが 1 のとき 1 になる）を行うビット単位の OR 演算子です。

最上位ビットだけを 1 にし、他のビットの値を保持するために 0x8000U とビット毎の OR 演算をします。1 にしたいビットのみを 1 にした定数とビット毎の OR 演算をすると、他のビットを変更せずに必要なビットだけを 1 にすることができます。

以下にビットごとの OR 演算の状況を示します。



■図 3.6 OR 演算のメモリーイメージ

|=

|= は複合代入演算子です。ITMC |= 0x8000U;は ITMC = ITMC | 0x8000U;と同じになります。複合代入演算子には += 、-= など他にも多数あります。

__EI();

割込みに関連する設定の変更が終了しましたので、割込みを許可します。(EI:enable

interrupt) __DI() と同様の組込み関数です。

static unsigned int counter_5ms = 0;

static は静的変数で、unsigned は符号がなく、int は 16 ビットの整数であり、counter_5ms という名前の変数を定義しています。この変数は 0～65535 までの値が記録できます。ブロック{}の中で宣言した変数はブロック内でのみ使えます。そのブロックの中のブロックでも使えますが、そのブロックの外では使えません。

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	← 0 または 1
3	1															
2	6	8	4	2	1											
7	3	1	0	0	0	5	2	1								
6	8	9	9	4	2	1	5	2	6	3	1					
8	4	2	6	8	4	2	6	8	4	2	6	8	4	2	1	
の	の	の	の	の	の	の	の	の	の	の	の	の	の	の	の	
位	位	位	位	位	位	位	位	位	位	位	位	位	位	位	位	

■ 図 3.7 counter_5ms のメモリーイメージ

```
while(flag_5ms == 0){
}
flag_5ms = 0;
```

これが、メインループの中で 5 ms 周期を作り出すためのしくみです。

メインループの先頭のこの while 文で flag_5ms が 1 になるまで待ちます。右側の値が左側の値と等しいかを比較する演算子は==で表わし、等価演算子と言います。

flag_5ms は 12 ビット・インターバル・タイマの割込み処理で 5 ms ごとに 1 が設定されます。

メイン処理では flag_5ms が 1 になったら while 文を抜けて flag_5ms を 0 にクリアして下に続くメインループの処理を実行します。メインループの処理を終了すると再びメインループの先頭に戻ります。

このようにして、5 ms 周期のメインループが実行されます。

switch_tmp = (switch_tmp << 1) + P12_bit.no1;

変数のデータを 1 ビット左へシフトし、今回のスイッチ入力のレベルを最下位ビットに入れます。

<<

<< は左側の変数のビットを右側の値だけ左へ移動させる左シフト演算子です。あいた右側には 0 が入ります。上の例では switch_tmp << 1 で 1 ビット左へシフトしています。

①ある時点での一時記憶用変数が以下のようになっていたとします。

switch_tmp のメモリーイメージ

0	0	0	1	0	1	1	1

② 演算結果は次のようになります。

0	0	1	0	1	1	1	0

■図 3.8 左シフト演算のメモリイメージ

```
if(counter_5ms > 0){
```

> は左側の値が右側の値よりも大きければ真(1)となり、そうでなければ偽(0)となります。

> は比較演算子といいます。

比較演算子には他に以下のようなものがあります。

>=

左側の値が右側の値よりも大きいか等しければ真(1)となり、そうでなければ偽(0)となります。

<

左側の値が右側の値よりも小さければ真(1)となり、そうでなければ偽(0)となります。

<=

左側の値が右側の値よりも小さいか等しければ真(1)となり、そうでなければ偽(0)となります。

==

左側の値が右側の値と等しければ真(1)となり、そうでなければ偽(0)となります。

!=

左側の値が右側の値と等しくなければ真(1)となり、そうでなければ偽(0)となります。

```
counter_5ms = counter_5ms - 1;
```

学校で学んだ数学ではおかしな表現に見えます。2 章でも説明しましたが、C 言語では = は等号ではなく代入であるので、右側の値を左側に代入するということになります。

つまり、counter_5ms が最初に 5 であったとすると、5 から 1 を引いた 4 が counter_5ms に代入されるということになります。- は減算の四則演算子です。

演算の優先順位を含めて四則演算子を示します。式の中では優先順位の高い演算が先に実行されます。

例えば 5 + 2 * 3 では 2 * 3 が先に実行されます。

■表 3.1　四則演算子と優先順位

四則演算子	機能
+、−	正符号、負符号
*、/、%	乗算、除算、余り
+、−	加算、減算

高い
↕ 優先順位
低い

P1_bit.no4 = ~P1_bit.no4;

LED が接続されたポート(P1.4) の値(ビット)を反転させて再びポート(P1.4)へ設定します。そのため、LED の出力が反転します。(消灯⇔点灯)

~ (チルダ) はビット単位否定演算子で、ビット単位で値を反転します。P1_bit.no4 は 1 ビットなので、1 ならば 0 に、0 ならば 1 になります。

static void it_interrupt(void){

これは 12 ビット・インターバル・タイマ割込みが発生した時の処理を行う関数です。

このファイルの先頭の#pragma で 12 ビット・インターバル・タイマ割込みの関数であることを指定しています。

12 ビット・インターバル・タイマ割込みは今回設定した、5ms 周期で発生します。

【列挙型について】

enum STATELED {LED_OFF = 0,LED_ON,LED_BLINK};

enum STATELED は STATELED という名前の列挙型の宣言です。

列挙型とは、名前をつけた整数定数 (列挙定数) を一つの集合として定義し、取り扱うことができる型です。

STATELED は列挙体タグと言います。後の{}内の文字列が列挙定数です。LED_OFF = 0 は LED_OFF に 0 を割り当てることを示します。その後の列挙定数には 1,2 の順に自動的に値が割り当てられます。switch case 文のラベルには数値を指定しますが、列挙型を用いることで、プログラムの意味が分かりやすくなります。

static enum STATELED e_state_LED = LED_OFF;

静的な STATELED という列挙型の変数 e_state_LED を宣言し、列挙定数 LED_OFF で初期化し定義しています。つまり、初期化時点では消灯状態(LED_OFF)になります。

if(switch_level != switch_level_before){

!=は非等価演算子です。右側の値と左側の値が等しくなければ真 (1) となり、等しければ偽 (0) となります。if 文の( )の中が真となれば、続く{ }の中に書かれたプログ

ラムが実行されます。偽となれば、{ }の中に書かれたプログラムは実行されません。C 言語では 0 以外を真としています。

```
switch(e_state_LED){
    case LED_OFF:
        ・・・・・・・・・・
        break;
    case LED_ON:
        ・・・・・・・・・・
        break;
    case LED_BLINK:
        ・・・・・・・・・・
        break;
    default:
        break;
}
```

switch case 文です。

switch の()の中の式（変数値）と一致する定数式（定数値）を示す case 節以降を実行します。break 文があると switch case 文を抜けます。

break 文を置かないとその下に続く文を実行してしまいます。break 文の記載漏れは発見しにくいバグになることがあるので注意してください。意図的に break 文を置かないでその下に続く文を実行させるときは、break 文を省いた理由をコメントで記載しましょう。時間が経って読み返したときに、書き忘れのミスと勘違いして break 文を書いてしまいバグになることがあります。

switch の()の中の式が case 節が示す定数式と一致するものがない場合 default:節に続く文を実行します。実行する可能性のない default:節に分岐してきたときは、異常ですので、安全や信頼性が要求される分野では、そのシステムに応じた適切な処理を記述する必要があります。

ここでも、switch case 文を用いて状態遷移を制御していますが、状態遷移の制御の詳細については、書籍の 4 章を参照してください。

```
counter_5ms --;
```

--（減分）演算子

counter_5ms = counter_5ms - 1; と同じ結果になります。-- counter_5ms; もこの場合は同じ結果となりますが、以下の違いがあるので注意が必要です。実行前に

counter_5ms の値が 5 であった場合 result = counter_5ms --;は実行後 counter_5ms の値は 4 となりますが、result は減算前の値 5 となります。counter_5ms の値の result への代入が行われた後に counter_5ms の減算が行われるためです。

result = --counter;は実行後 counter も result も減算後の値 4 となります。counter_5ms の減算が行われた後、counter_5ms の値の result への代入が行われるためです。

++（増分）演算子は変数の値を+1 増加するものですが、演算の順序は減分演算子と同様になります。

第 4 章 ソフトウェアアーキテクチャ

4.2 ソフトウェアへの要求事項の明確化

電気ケトルの状態遷移を実現する例を**ソースコード 4.1** に示します。

■ソースコード 4.1 電気ケトルのシステム状態を制御する関数

```
/* 電気ケトルのシステム状態の制御処理 */
void system(void){
    /* システム状態変数 */
    enum SYSTEM_STATE{WAIT = 0,HEAT,COMPLETE,OVERHEAT,TH_BREAKDOWN};
    static enum SYSTEM_STATE e_system_state = WAIT;
    /* ９８℃以上での２分計数用カウンタ */
    static signed int s16_time_counter_98 = 0;
    /* ９０℃以上での１０分計数用カウンタ */
    static signed long int s32_time_counter_90 = 0;
    switch(e_system_state){
        case WAIT:                                  /* 待機状態 */
            if((u16_temperature <= TEMP_40) || (TEMP_150 <= u16_temperature)){
                /* サーミスタ故障であればサーミスタ故障状態へ */
                e_system_state = TH_BREAKDOWN;
            }else if(f_switch.on_off == ON){        /* スイッチが押下された場合 */
                f_switch.on_off = OFF;              /* スイッチ押下フラグクリア */
                /* 98℃以上での沸騰完了待ち時間を 2 分に設定 */
                s16_time_counter_98 = BOILING_PERIOD_2M;
                /* 90℃以上での沸騰完了待ち時間を 10 分に設定 */
                s16_time_counter_90 = BOILING_PERIOD_10M;
                e_system_state = HEAT;              /* スイッチが押下されれば加熱状態へ */
            }
            e_heat_output = HEAT_OFF;               /* ヒータオフ */
            e_lighting_type_oder = LIGHT_OFF;       /* LED 消灯 */
            break;
        case HEAT:                                  /* 加熱状態 */
            if((us16_temperature <= TEMP_40) || (TEMP_150 <= u16_temperature)){
                /* サーミスタ故障であればサーミスタ故障状態へ */
                e_system_state = TH_BREAKDOWN;
            }else if(f_switch.on_off == ON){        /* スイッチが押下された場合 */
                f_switch.on_off = OFF;              /* スイッチ押下フラグクリア */
                e_system_state = WAIT;              /* 待機状態へ */
            }else if(TEMP_110 <= u16_temperature){
                e_system_state = OVERHEAT;          /* 110℃以上であれば空焚き状態へ */
            }else if(TEMP_98 <= u16_temperature) {
                /* ９８℃以上で２分経過すれば沸騰完了状態へ */
                if(s16_time_counter_98-- <= 0){
                    e_system_state = COMPLETE;
                }
            }else if(TEMP_90 <= u16_temperature) {
                /* 90 ℃以上で 10 分経過すれば沸騰完了状態へ */
                if(s32_time_counter_90-- <= 0){
                    e_system_state = COMPLETE;
                }
            }
            e_heat_output = HEAT_ON;                /* ヒータオン */
            e_lighting_type_oder = LIGHT_ON;        /* LED 点灯 */
            break;
```

```
            case  COMPLETE:                         /* 完了状態 */
                  if((u16_temperature <= TEMP_40) || (TEMP_150 <= u16_temperature)){
                        /* サーミスタ故障であればサーミスタ故障状態へ */
                        e_system_state = TH_BREAKDOWN;
                  }else if(f_switch.on_off == ON){      /* スイッチが押下された場合 */
                        f_switch.on_off = OFF;          /* スイッチ押下フラグクリア */
                        e_system_state = WAIT;          /* 待機状態へ */
                  }else if(TEMP_110 <= u16_temperature){
                        e_system_state = OVERHEAT;      /* 110℃以上であれば空焚き状態へ */
                  }
                  e_heat_output = HEAT_OFF;             /* ヒータオフ */
                  e_lighting_type_oder = COMPLETION_INFORM;/* LED 長周期点滅 */
                  break;
            case  OVERHEAT:                         /* 空焚き状態 */
                  if((u16_temperature <= TEMP_40) || (TEMP_150 <= u16_temperature)){
                        /* サーミスタ故障であればサーミスタ故障状態へ */
                        e_system_state = TH_BREAKDOWN;
                  }else if(f_switch.on_off == ON){      /* スイッチが押下された場合 */
                        f_switch.on_off = OFF;
                        e_system_state = WAIT;          /* 待機状態へ */
                  }else if(u16_temperature <= TEMP_98) {
                        e_system_state = HEAT;          /* 98 ℃以下で加熱状態へ */
                  }
                  e_heat_output = HEAT_OFF;             /* ヒータオフ */
                  e_lighting_type_oder = ALARM;         /* LED 短周期点滅 */
                  break;
            case  TH_BREAKDOWN:                     /* サーミスタ故障状態 */
                  e_heat_output = HEAT_OFF;             /* ヒータオフ */
                  e_lighting_type_oder = ALARM;         /* LED 短周期点滅 */
                  break;
            default:                                /* 状態変数異常 */
                  e_system_state = TH_BREAKDOWN;    /* 異常発生のためサーミスタ故障状態へ */
                  e_heat_output = HEAT_OFF;             /* ヒータオフ */
                  e_lighting_type_oder = ALARM;         /* LED 短周期点滅 */
                  break;
      }
}
```

以下に、ソースコードにおける処理内容やC言語の基本的な文法等の解説をします。以前に説明した内容は省きます。

void system(void){

void system(void)は電気ケトルのシステム状態を制御する関数です。この関数はメインループから周期（8 ms）ごとに呼出されます。

switch(e_system_state){

case WAIT:　　/* 待機状態 */

・・・・・・・・・・・・・・・・

switch 文の()の中の状態を表わす変数の値に対応する定数と一致する case 文の処理

を実行することにより状態の制御が行われます。

各状態に対応するcase文からbreak文の間に、状態遷移先の設定とアクションおよび、メイン周期毎に実行するアクティビティを記載します。詳細は以下の通りです。

if 文の()の中で時間経過や他の機能モジュールから通知された入力などのイベントとガード条件を判断し、対応するイベントが発生すれば状態変数を変更し、実施すべきアクションがあれば処理します。

if 文で状態変数を変更することにより、次のメインループの周期では、変更した状態に対応する case 文が実行されます。これにより、状態の遷移が達成されます。if 文での比較順序は、安全性や故障対応、操作性を考慮して、優先度の高いものを先にします。

次に、その状態で毎回実施すべきアクティビティを実施します。例では、他の機能モジュールへ通知するヒータの制御状態の指示とLEDの点灯種別の指示を毎回設定しています。このようにして出力状態が制御されます。状態遷移のアクションで出力を設定しても可能ですが、状態遷移時の設定漏れが発生しないように、状態のアクティビティとしてメインループの周期毎に実施しています。

最後に、break;で switch case 文を抜けます。

また、switch case 文の default:節でサーミスタ故障へ遷移させています。正常状態では default:節へ分岐する可能性がありませんが、安全性、信頼性を必要とするシステムでは default:節でそのシステムに適した異常処理を入れた方が良いでしょう。

このように、switch case 文と状態を表す変数を用いて状態を制御すると、状態遷移が分かりやすくなります。if 文でも同じような制御はできますが、状態遷移が複雑になったときに、switch case 文の方がはるかに分かりやすくなり、状態の見落としもなくなります。

以上のように、状態ごとにイベントを処理する状態遷移の制御をステートドリブン(State driven)型と言います。また、イベントを監視して、イベント毎に現在の状態に応じて処理をするイベントドリブン(Event driven)型もあります。Windows 等のプログラミングではイベントドリブン型がほとんどです。しかし、ワンチップマイコンのプログラミングでは、状態を管理することが主になりますから、ステートドリブン型の方が制御しやすいでしょう。

```
if((u16_temperature <= TEMP_40) || (TEMP_150 <= u16_temperature)){
```

if 文の中には、複数の条件の比較を書くことができます。上の文では演算の優先順位が比較演算子（<=）の方が論理演算子（||）よりも高いので、内側の()を省略しても結

果は同じとなります。以下の if 文でも内側の( ) を省略しても結果は同じとなります。

```
if((4 * u8_test) - u8_temp) > (u8_temp * u8_test)){
```

しかし、以下の if 文ではビット単位の AND 演算子は比較演算子（>）よりも優先順位が低いので、内側の()がある場合とない場合とで結果が異なります。複数の条件の比較を行うときは、( )を付けて、意図通りに比較が行われるようにすることを推奨します。

```
if((u8_test & 0xAAU) > (u8_temp & 0xAAU)){
```

|| 論理和（OR）演算子

左側の値と右側の値のどちらか一つでも真（0 でない）であれば 1 となり、両方とも偽（0）であれば 0 となります。

その他以下の論理演算子があります。

&& 論理積（AND）演算子

左側の値と右側の値の両方とも真（0 でない）であれば 1 となり、どちらか一つでも偽（0）であれば 0 となります。

! 論理否定（NOT）演算子

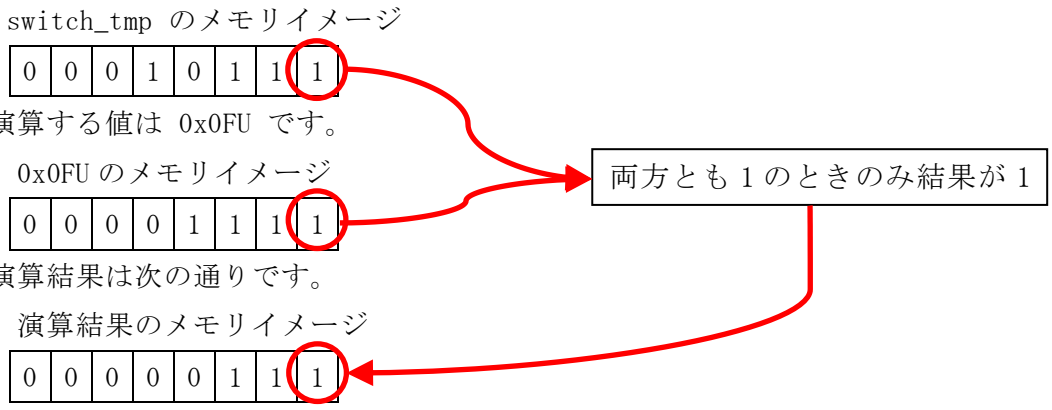
右側の値が真（0 でない）であれば 0 となり、偽（0）であれば 1 となります。

【ビット単位の AND 演算子】

&

& はビット毎に AND 演算(演算される 2 つのビットが両方とも 1 のときだけ 1 になる)を行うビット単位の AND 演算子です。

①ある時点での一時記憶用変数が以下のようになっていたとします。



■図 4.1 AND 演算のメモリイメージ

f_switch.on_off

f_switch.on_off は変数をビット単位で指定するビットフィールドですが、宣言や定義を含め後の章で説明します。

第 6 章　C 言語が備えるモジュール化のしくみ

6.6 モジュール化のサンプルプログラム

この章のソースコードでは、これまで説明しなかった内容（文法等）は条件コンパイルのみとなりますのでソースコード 6.1 のみ説明します。(ファイル名：modularizationcc)

【スイッチ入力検知モジュール】

■ソースコード 6.1 スイッチ入力検知モジュールインタフェース

```
/* switch.h　スイッチ入力検知モジュールインタフェース */

#ifndef SWITCH_H
#define SWITCH_H

/* 公開変数の宣言 */
extern unsigned char flag_switch_on;          /* スイッチ押下フラグ */

/* 公開関数の宣言 */
extern void switch_input(void);               /* スイッチ入力検知処理 */

#endif
```

#ifndef SWITCH_H

#ifndef SWITCH_H　はこれ以前に SWITCH_H が定義されていなければ、この文から #endif の間の文を有効とする（コンパイルの対象とする）ものです。#ifndef は古い形式の表記です。新しい形式の表記では#if !define とします。

#ifndef SWITCH_H の次に#define SWITCH_H として SWITCH_H を定義しています。

このため、このヘッダファイルが#include 文で同じファイルに複数回取り込まれたとしても、ここに記述された変数や関数は 1 度だけコンパイルされます。

この章では、オーム社の Web サイトに掲示したおもちゃの二足歩行ロボットのソースコードの文法等で、これまで説明しなかった内容を説明します。(ファイル名：walkcc)

SFR の設定等はマイコンのハードウェアマニュアルを参照してください。

７章の説明に入る前に、この章で使う配列と構造体、共用体、ポインタについて簡単に説明します。

【配列について】

配列は、同じ型のデータをメンバとしてひとまとめにしたものです。宣言と初期化は次のように行います。

型名　識別子[要素数 n]={要素 1,要素 2,・・・,要素 n};

これは、添字[]が一つの一次元配列です。宣言時の要素数は定数であり、変数は使えません。要素番号は 0 から n-1 までとなります。

配列を要素とする配列も次のように作れます。

型名　識別子[要素数 1][要素数 2]

={{要素 1_1,要素 1_2},{要素 2_1,要素 2_2},{要素 3_1,要素 3_2}・・・};

これは、添字が二つの二次元配列と言います。更に、添字を増やすことによって多次元の配列を作ることができます。

関数において配列を渡す場合は、参照渡しを用います。

通常、関数において、関数を呼び出す側の実引数の値を、関数の仮引数にコピーして渡す値渡しが行われます。そのため、関数の中で引数に値を代入しても、呼出し側の値を書き換えることがありません。ところが、配列を引数とした場合は、配列のアドレスを渡す参照渡しが行われます。関数において配列の要素に値を代入すると、呼出し側の配列の要素を書き換えてしますことになりますので注意が必要です。

【構造体について】

構造体は、異なる型のデータをメンバとしてひとまとめにしたものです。宣言と初期化は次のように行います。

```
struct タグ名{
  型名　メンバ名 1;
  型名　メンバ名 2;
  型名　メンバ名 2;
}識別子={データ 1,データ 2,データ 3};
```

関数において構造体を渡す場合は値渡しができますが、構造体の配列では、参照渡しとなります。

【共用体について】

共用体は、同じメモリ領域を異なる変数型に割付けます。宣言と初期化は次のように行います

```
union タグ名{
  型名　メンバ名 1;
  型名　メンバ名 2;
  型名　メンバ名 3;
}識別子={データ};
```

共用体は、先頭の型名（メンバ 1）で初期化されます。

共用体は、あるデータを、型を変えて取り出す場合に使います。本書では、8 バイトの受信パケットデータを構造体の配列に変換するために用いました。共用体では、構造体配列や多次元の配列では初期化できないため、先頭にはダミーの一次元の配列のメンバを割り当て、初期化しています。

【ポインタについて】

ポインタとは、変数、定数、関数の格納番地（アドレス）を保持するデータ型です。ポインタに格納されるのはアドレスです。ポインタを介してデータを格納したり、取り出したりすることができます。ポインタを宣言するには派生型宣言子「*」を用いて例えば次のように宣言します。

```
int *p;
```

これは、整数型を示すポインタを宣言しています。ポインタは宣言しただけでは値が設定されていないので、使用することはできません。ポインタを初期化せずにポインタを用いて値を代入すると、どこにデータを書込むか分からないためプログラムが暴走する恐れがあります。ポインタの初期化は、例えば次のようにします。整数の変数 a が定義されていたとします。

```
int a = 250;
```

アドレス演算子「&」を用いてポインタへ変数 a のアドレスをポインタに代入します。

```
p = &a;
```

すると、ポインタ p に変数 a のアドレスが設定され初期化されました。次に、ポインタ変数 p を用いて変数 a の値を取り出して整数の変数 b に代入するには間接参照演算子「*」を用います。

```
b = *p;
```

これで、変数 b に変数 a の値が代入されました。

ところで、ポインタは使い方を誤るとプログラムが暴走するので、どうしてもポインタでなければ実現できない用途に限定すべきです。本書では、複数の配列や構造体の配列を一つの関数で切り替えて使う場合にのみ用いています。その場合でも、ポインタを配列形式で用いています。

第 7 章　具体例によるワンチップマイコンソフトウェア設計プロセスの解説

【システム状態制御モジュール】

〈system.c〉

```
static void execute_sequence_data(struct SEQDATA *seq_data){
    switch(seq_data[sequence_counter].category){
        ・・・・・・・・・・・・・・・・
        s16_count_target_request = seq_data[sequence_counter].count;
        ・・・・・・・・・・・・・・・・
```

構造体の配列を、構造体へのポインタとして引数で渡します。

関数内において､構造体の各メンバへのアクセスは構造体配列形式の表記で行います。

構造体自身は値渡しできますが、構造体の配列は渡せないため、ポインタを用いて参照渡ししています。

【歩行動作シーケンスデータモジュール】

〈sequence_data.h〉

```
struct SEQDATA{
    unsigned char category;      /* 歩行指示 */
    unsigned char count;             /* 回数・時間 */
};
```

構造体 SEQDATA を宣言しています。

SEQDATA は構造体名（タグ名）といいます。この構造体は category、count という 2 つのメンバ名の変数からなります。一般的に構造体は、異なる変数型からなるメンバで構成します。同じ変数型であれば配列で構成することも出来ますが、ここでは、メンバを識別しやすくするため構造体で構成しています。

〈sequence_data.c〉

```
const struct SEQDATA builtin_data[SQ_NO_MAX] = {
    {0x0FU, 0x02U},{0x01U, 0x04U},{0x06U, 0x02U},{0x02U, 0x04U},
    {0x08U, 0x08U},{0x02U, 0x04U},{0x04U, 0x04U},{0x00U, 0x00U},
    {0x00U, 0x00U},{0x00U, 0x00U},{0x00U, 0x00U},{0x00U, 0x00U},
    {0x00U, 0x00U},{0x00U, 0x00U},{0x00U, 0x00U},{0x00U, 0x00U},
    {0x00U, 0x00U},{0x00U, 0x00U},{0x00U, 0x00U},{0x00U, 0x00U},
    {0x00U, 0x00U},{0x00U, 0x00U},{0x00U, 0x00U},{0x00U, 0x00U},
```

```
    {0x00U, 0x00U},{0x00U, 0x00U},{0x00U, 0x00U},{0x00U, 0x00U},
    {0x00U, 0x00U},{0x00U, 0x00U},{0x00U, 0x00U},{0x00U, 0x00U}
};  /* 内蔵プログラムシーケンスデータ格納領域 */
```

構造体 SEQDATA の定数配列を宣言、定義しています。

上記では、**{0x0FU, 0x02U}**が配列の 0 番目の要素への代入値です。

つまり、builtin_data[0].category に 0x0FU が、builtin_data[0].count に 0x02U が割り当てられます。

この配列のデータを変更することで、内蔵プログラム動作モードの歩行パターンを変更できます。category は歩行動作の種類、count は目標歩数または目標停止時間（0.5 秒単位）を指定します。

歩行動作の種類の値は以下の通りです。

停止	0x0F
前進	0x01
右回転	0x02
後退	0x04
左回転	0x08
ダンス	0x06

```
static union SEQUENCE sequence_buf = {
    0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U,
    0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U,
    0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U,
    0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U,
    0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U,
    0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U,
    0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U,
    0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U
};
```

共用体 SEQUENCE の変数 sequence_buf を宣言し、定義しています。

共用体 SEQUENCE の定義は【通信制御モジュール】のところで説明します。

初期化代入は先頭のメンバ u8_dummy[2 * SQ_NO_MAX]に対して行われます。

【単位ステップシーケンスデータモジュール】

```
/* ホームポジションのパルス幅宣言 */
#define M01HOME 31199U
#define M02HOME 31199U
#define M03HOME 31199U
#define M04HOME 31199U
```

このモジュールで、ホームポシション（正立姿勢）のパルス幅を指定します。

ロボットの組み立てにおいて、RC サーボモータの中心位置（1300μs）を正立姿勢の位置に設定しますが、サーボホーンやギア、その他部品の組み付けによる姿勢のずれが発生します。それを、これらの値を調整して補正します。

31199U が（1300μs）になります。姿勢の調整は目視で行います。停止姿勢の状態で姿勢に歪みがあれば、ホームポジションのパルス幅を増減します。

1° あたり 240 になりますので、上記の値を増減させて姿勢の歪みを矯正します。

設定値を間違えてロボットを壊さないように注意してください。

姿勢調整手順の詳細は「付録メカの作成方法」を参考にしてください。

【単位ステップシーケンス制御モジュール】

〈unit_sequence.c〉

単位ステップシーケンス制御処理では二次元配列を引数として渡します。ここでは、二次元配列を関数の引数として渡す方法を説明します。

```
static unsigned int (*u16_sequence_unit)[2];
```

静的な列の要素数 2 の unsigned int 型配列へのポインタ変数 u16_sequence_unit を宣言します。このポインタ変数を宣言している理由は以下の通りです。

次に示す関数が引数として配列へのポインタを取得しますが、引数の値（ここでは配列のアドレス）は、その関数呼出しが終了すると消滅します。そのため、静的なポインタ変数に代入し、関数の終了後も記憶しておくことで、その関数が属するファイル（モジュール）の他の関数がその配列を利用できるようになります。

宣言では列の要素数のみ指定しています。これは､行の要素数を可変とするためです。

```
void sequence_shift_preprocess(const unsigned int u16_sunit[][2],unsigned int
u16_snumber_max){
```

静的な列の要素数 2 の unsigned int 型の二次元定数配列 u16_sunit と unsigned int 型の変数 u16_snumber_max を引数とする関数 sequence_shift_preprocess を定義しています。u16_snumber_max で二次元定数配列の行の要素数を渡します。

```
u16_sequence_unit = (unsigned int (*)[2])u16_sunit; /* 目標姿勢配列へのポインタを設定 */
```

引数 u16_sunit を要素数 2 の unsigned int 型の配列へのポインタにキャストし、ポインタ変数 u16_sequence_unit に代入します。

```
u16_pulse_width_target[i] = u16_pulse_homeposition[i]
    + s16_pulse_width[u16_sequence_unit[u16_sequence_step_number][0]][i];
```

引数で渡された二次元定数配列 u16_sunit へのアクセスを二次元配列形式（u16_sequence_unit[][]）の表記で行い、列の 1 番目の要素（目標姿勢の差分パルス幅）を抽出します。

```
u16_posture_duration_counter = u16_sequence_unit[u16_sequence_step_number][1];
```

二次元定数配列 u16_sunit へのアクセスを二次元配列形式（u16_sequence_unit[][]）の表記で行い、列の 2 番目の要素（姿勢数）を抽出します。

【モータ駆動制御モジュール】

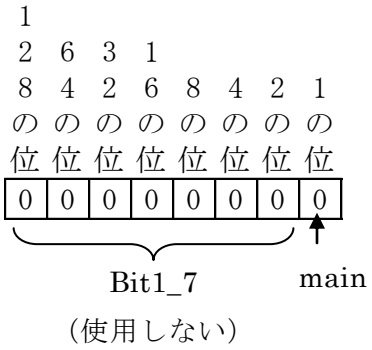
〈motor.h〉

```
struct _F_TIMER{
    unsigned char main:1;
    unsigned char bit1_7:7;
};
```

これは、構造体型の宣言を用いた、ビットフィールドの宣言です。

メモリを節約する目的で、フラグとして使用する変数などをビット単位でメモリに割付けるために、ビットフィールドを用います。

_F_TIMER は構造体名です。main や bit1_7 はフィールド名です。フィールド名の「:」のつぎの数字でフィールドのビット幅を示します。ビットフィールドを指定できる型はコンパイラに依存しますが、CC-RL では unsigned char も用いることができます。CC-RL では、最初に記述したビットフィールドが下位ビットから割り当てられます。ビットフィールド bit1_7 は変数を 8 ビットに調整するためのパディング（詰め物）です。



■図 7.1 構造体_F_TIMER のビットフィールドの割付け

```
struct _F_TIMER f_timer = {0,0};        /* 20ms 経過フラグ  */
```

ビットフィールドを指定した構造体_F_TIMER の変数 f_timer を宣言・定義し、各フィールドに 0 を代入しています。

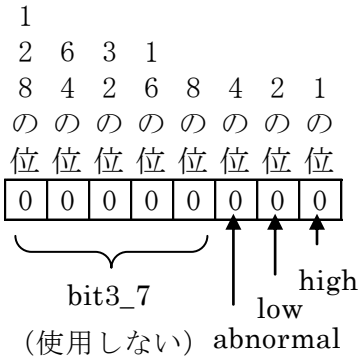
```
f_timer.main = 1U;        /* 20ms 経過  */
```

ビットフィールドは、このようにビットごとに読み書きします。

この例では 1 ビットしか利用していませんが、電池電圧検知モジュールでは、以下のように 3 ビット利用しています。

```
struct _F_VB_ALARM{
    unsigned char high:1;          /*  電池高電圧  */
    unsigned char low:1;           /*  電池低電圧  */
    unsigned char abnormal:1;      /*  電池電圧異常  */
    unsigned char bit3_7:5;        /* padding(詰物) 3～8 ビット目を埋める  */
};
```

f_vb_alarm のビットイメージは以下のようになります。



■図 7.2 f_vb_alarm のビットイメージ

TO0 &= ~0x0001U; /* TO00 をﾎﾟｰﾄ機能として使用 */

&= はビット単位の AND 演算の複合代入演算子です。

このような記述は定数の 1 としたビットを強制的に 0 にし、その他のビットの値を保持するときに使われます。

【通信制御モジュール】

<serial.c>

static unsigned char uart0_tx[PACKET_SIZE] = {TX_STARTCODE,0,0,0,0,0,0,0,0,0,0,0};

静的な 8 ビット符号なし文字型の要素数 12(PACKET_SIZE)の配列 uart0_tx を宣言し、定義しています。

宣言時の[]の中は要素数で、定数式で指定します。配列を読み書きするときの[]の中の添字は必ず 0 からとなります。そのため、使用時の添字の上限は定数式-1 となります。

メモリ上には**表 7.17** のように配置されます。ただし、番地は概念を説明するためのものです。実際に割り当てられる番地は異なります。[ ] の中の添字は各要素を指します。

（注）番地は先頭番地からの相対値です。

■表 7.1 配列 uart0_tx[ ]のメモリ上の配置イメージ

番地	配列要素
11	uart0_tx[11]
10	uart0_tx[10]
・・ ・・	・・・・・・ ・・・・・・
3	uart0_tx[3]
2	uart0_tx[2]
1	uart0_tx[1]
0	uart0_tx[0]

```
union SEQUENCE{   /* シーケンスデータ構造体　⇔　パケット配列　変換 */
    unsigned char u8_dummy[2 * SQ_NO_MAX];      /* 共用体初期化のための代用配列 */
    struct SEQDATA u8_seqpair[SQ_NO_MAX];  /* シーケンスデータ参照用 */
    unsigned char u8_seq[SQ_NO_MAX / 4][NET_DATA_SIZE];/* 通信データ格納用 */
};
```

共用体 SEQUENCE を宣言しています。

SEQUENCE は共用体名（タグ名）といいます。共用体は、同じ RAM 上のデータを異なる型で扱うときに利用します。ここでは、歩行動作のシーケンスデータを歩行動作の実行時と、通信データ受信時の両方でデータを扱いやすくするために用いています。

・ 歩行動作の実行時は、歩行動作の種別と目標歩数の 2 バイト単位で利用します。
u8_seqpair は歩行動作の種別と目標歩数をメンバとする構造体です。
・ 通信データ受信時は、1 回の通信において 8 バイト単位で受信します。
u8_seq は列が 8 つの 1 バイトデータからなる配列です。

また、共用体のデータ初期化では複数次元のデータを代入できないので、全領域を初期化するために､ダミーの 1 次元配列のメンバ u8_dummy を先頭のメンバとして宣言しています。

__nop();

NOP 命令を実行する組込み関数です。NOP は何もしない機械語命令です。ハードウェア設定をするときなどに、短時間の時間待ちが必要なときに使われます。

u8_rx_data = (unsigned char)RXD0;

受信完了割込み処理の中で、受信用のシリアル・データ・レジスタ(RXD0)から u8_rx_data にデータを代入している文です。

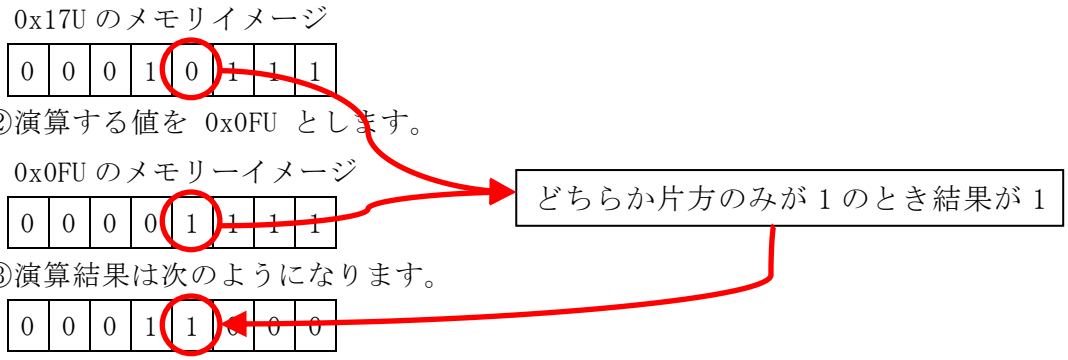
(unsigned char) は明示的な型変換（キャスト）といいます。RXD0 は 16 ビットの符号なし短整数型 unsigned short で定義されています。

受信データは RXD0 の下位 8 ビットが格納されます。そのため、キャストにより 8 ビットの符号なし文字型に変換してから 8 ビットの符号なし文字型変数 u8_rx_data に代入します。この例ではキャストがなくても暗黙の型変換により u8_rx_data に受信データが代入されますが、より明確にするためにキャストをつけました。

u8_rx_checkcode ^= u8_rx_data; /* 受信 BCC 算出 */

ビット単位の XOR 演算子の複合代入演算子です。^はビット毎に XOR 演算（演算される 2 つのビットが異なるときに 1 になる）を行います。

①演算される値を 0x17U とします。



■図 7.3 XOR 演算のメモリイメージ

【「^」の読み方】

「^」はハット（hat)という読み方が一般的なようですが、キャレット(caret)とも読みます。しかし、日本工業規格の JIS X0201－1997 「7 ビット及び 8 ビットの情報交換用符号化文字集合」では「^」の名称を circumflex accent 、日本語通用名称をアクサンシルコンフレックスとしています。特殊記号は他にも複数の読み方のあるものがありますが、聞き手に伝われば良いのではないでしょうか。

```
static void uart0_send(unsigned char * tx_buf){
    u8p_uart0_tx_address = tx_buf;   /* 送信ﾊﾞｯﾌｧの先頭ｱﾄﾞﾚｽを設定 */
    ・・・・・・・・・・・・・・・・・・・・・・・・・
    TXD0 = u8p_uart0_tx_address[u8_uart0_tx_count];
    ・・・・・・・・・・・・・・・・・・・・・・・・・
}
```

C 言語では、配列を関数に渡すとき、その配列の先頭アドレスをポインタ変数として引数で渡します。関数は配列を値渡し（値をコピーして関数に渡すこと）できないので、このようにポインタを用いて参照渡しをします。ここでは unsigned char 型のポインタ変数を引数で渡しています。

そして、引数を静的な unsigned char 型のポインタ変数 u8p_uart0_tx_address に代入しています。引数の値（ここでは配列のアドレス）は、その関数呼出しが終了すると消滅します。その関数の終了後も、その関数が属するファイル（モジュール）の他の関数がその配列を利用できるようにするために、静的なポインタ変数を用いてアドレスを記憶しておく必要があるからです。

配列の各要素へのアクセスは配列形式の表記 u8p_uart0_tx_address[]で行うことができます。何番目の要素へのアクセスであるか明確にするために配列表記にしています。

引数にポインタを用いるので、関数内で値を変更すると関数を呼び出す側の配列の値が変更されることに留意してください。

関数を呼び出す側では、配列 uart0_tx[]を以下のように引数として渡します。

```
static unsigned char uart0_tx[PACKET_SIZE] = {TX_STARTCODE,0,0,0,0,0,0,0,0,0,0,0};
・・・・・・・・・・・・・・・・・・・・・・・・・
uart0_send(uart0_tx);                /* 送信開始 */
```