SmartRA 学習キット
チュートリアル 2

ルネサス エレクトロニクス社　RA マイコン搭載
HSB シリーズマイコンボード　評価キット

-本書を必ずよく読み、ご理解された上でご利用ください

株式会社北斗電子
REV.1.0.0.0

ー目　次ー

注意事項

本書を必ずよく読み、ご理解された上でご利用ください

【ご利用にあたって】

1. 本製品をご利用になる前には必ず取扱説明書をよく読んで下さい。また、本書は必ず保管し、使用上不明な点がある場合は再読し、よく理解して使用して下さい。
2. 本書は株式会社北斗電子製マイコンボードの使用方法について説明するものであり、ユーザシステムは対象ではありません。
3. 本書及び製品は著作権及び工業所有権によって保護されており、全ての権利は弊社に帰属します。本書の無断複写・複製・転載はできません。
4. 弊社のマイコンボードの仕様は全て使用しているマイコンの仕様に準じております。マイコンの仕様に関しましては製造元にお問い合わせ下さい。弊社製品のデザイン・機能・仕様は性能や安全性の向上を目的に、予告無しに変更することがあります。また価格を変更する場合や本書の図は実物と異なる場合もありますので、御了承下さい。
5. 本製品のご使用にあたっては、十分に評価の上ご使用下さい。
6. 未実装の部品に関してはサポート対象外です。お客様の責任においてご使用下さい。

【限定保証】

1. 弊社は本製品が頒布されているご利用条件に従って製造されたもので、本書に記載された動作を保証致します。
2. 本製品の保証期間は購入戴いた日から1年間です。

【保証規定】

保証期間内でも次のような場合は保証対象外となり有料修理となります

1. 火災・地震・第三者による行為その他の事故により本製品に不具合が生じた場合
2. お客様の故意・過失・誤用・異常な条件でのご利用で本製品に不具合が生じた場合
3. 本製品及び付属品のご利用方法に起因した損害が発生した場合
4. お客様によって本製品及び付属品へ改造・修理がなされた場合

【免責事項】

弊社は特定の目的･用途に関する保証や特許権侵害に対する保証等、本保証条件以外のものは明示・黙示に拘わらず一切の保証は致し兼ねます。また、直接的・間接的損害金もしくは欠陥製品や製品の使用方法に起因する損失金・費用には一切責任を負いません。損害の発生についてあらかじめ知らされていた場合でも保証は致し兼ねます。
ただし、明示的に保証責任または担保責任を負う場合でも、その理由のいかんを問わず、累積的な損害賠償責任は、弊社が受領した対価を上限とします。本製品は「現状」で販売されているものであり、使用に際してはお客様がその結果に一切の責任を負うものとします。弊社は使用または使用不能から生ずる損害に関して一切責任を負いません。
保証は最初の購入者であるお客様ご本人にのみ適用され、お客様が転売された第三者には適用されません。よって転売による第三者またはその為になすお客様からのいかなる請求についても責任を負いません。
本製品を使った二次製品の保証は致し兼ねます。

安全上のご注意

製品を安全にお使いいただくための項目を次のように記載しています。絵表示の意味をよく理解した上でお読み下さい。

表記の意味

⚠ 警告	取扱を誤った場合、人が死亡または重傷を負う危険が切迫して生じる可能性がある事が想定される
⚠ 注意	取扱を誤った場合、人が軽傷を負う可能性又は、物的損害のみを引き起こすが可能性がある事が想定される

絵記号の意味

	**一般指示** 使用者に対して指示に基づく行為を強制するものを示します		**一般禁止** 一般的な禁止事項を示します
	**電源プラグを抜く** 使用者に対して電源プラグをコンセントから抜くように指示します		**一般注意** 一般的な注意を示しています

⚠ 警告

以下の警告に反する操作をされた場合、本製品及びユーザシステムの破壊・発煙・発火の危険があります。マイコン内蔵プログラムを破壊する場合もあります。

1. 本製品及びユーザシステムに電源が入ったままケーブルの抜き差しを行わないでください。
2. 本製品及びユーザシステムに電源が入ったままで、ユーザシステム上に実装されたマイコンまたはIC等の抜き差しを行わないでください。
3. 本製品及びユーザシステムは規定の電圧範囲でご利用ください。
4. 本製品及びユーザシステムは、コネクタのピン番号及びユーザシステム上のマイコンとの接続を確認の上正しく扱ってください。

発煙・異音・異臭にお気付きの際はすぐに使用を中止してください。

電源がある場合は電源を切って、コンセントから電源プラグを抜いてください。そのままご使用すると火災や感電の原因になります。

⚠注意

以下のことをされると故障の原因となる場合があります。

1. 静電気が流れ、部品が破壊される恐れがありますので、ボード製品のコネクタ部分や部品面には直接手を触れないでください。
2. 次の様な場所での使用、保管をしないでください。

   ホコリが多い場所、長時間直射日光があたる場所、不安定な場所、衝撃や振動が加わる場所、落下の可能性がある場所、水分や湿気の多い場所、磁気を発するものの近く
3. 落としたり、衝撃を与えたり、重いものを乗せないでください。
4. 製品の上に水などの液体や、クリップなどの金属を置かないでください。
5. 製品の傍で飲食や喫煙をしないでください。

ボード製品では、裏面にハンダ付けの跡があり、尖っている場合があります。

取り付け、取り外しの際は製品の両端を持ってください。裏面のハンダ付け跡で、誤って手など怪我をする場合があります。

CD メディア、フロッピーディスク付属の製品では、故障に備えてバックアップ（複製）をお取りください。

製品をご使用中にデータなどが消失した場合、データなどの保証は一切致しかねます。

アクセスランプがある製品では、アクセスランプの点灯中に電源を切ったり、パソコンをリセットをしないでください。

製品の故障や、データ消失の原因となります。

本製品は、医療、航空宇宙、原子力、輸送などの人命に関わる機器やシステム及び高度な信頼性を必要とする設備や機器などに用いられる事を目的として、設計及び製造されておりません。

医療、航空宇宙、原子力、輸送などの設備や機器、システムなどに本製品を使用され、本製品の故障により、人身や火災事故、社会的な損害などが生じても、弊社では責任を負いかねます。お客様ご自身にて対策を期されるようご注意ください。

1. RA2L1_SW_LED

ベースボード上の

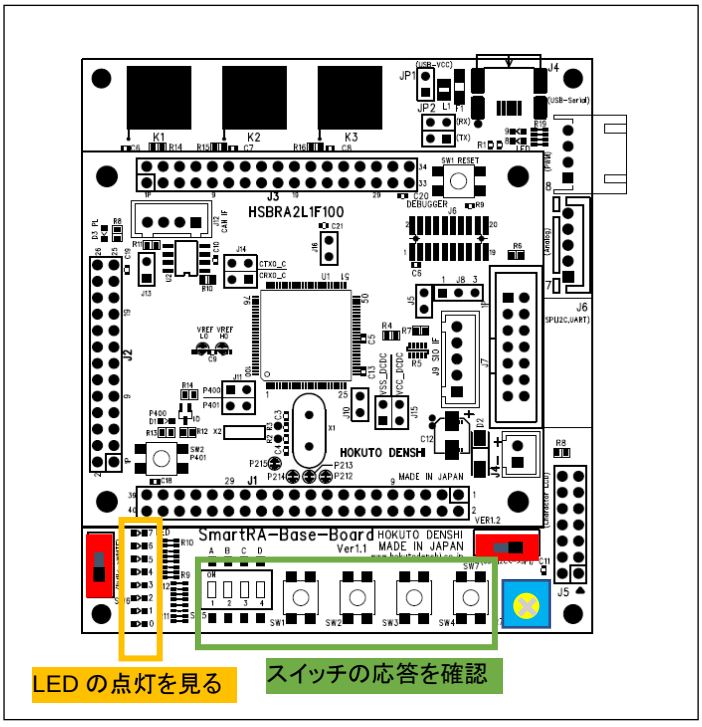
LED（B-LED0~7 の 8 つの LED）

DIP スイッチ(B-SW5, 4ch)

プッシュスイッチ(B-SW1~B-SW4)

を制御する方法を学ぶチュートリアルとなります。

1.1. プログラムの動作



最初の 6 秒間

LED7					○	
LED6						○
LED5					○	
LED4						○
LED3				○		○
LED2			○	○	○	
LED1		○	○	○		○
LED0	○	○	○	○	○	

1 秒ごとにパターンが変化

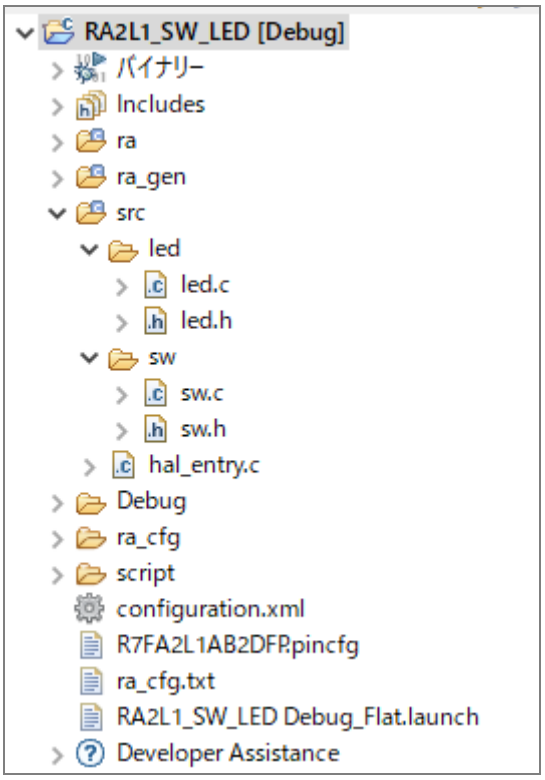
6 秒経過後

SW5-D	→	LED7
SW5-C	→	LED6
SW5-B	→	LED5
SW5-A	→	LED4
SW4	→	LED3
SW3	→	LED2
SW2	→	LED1
SW1	→	LED0

プログラムを実行すると、LED が 1 秒ごとに点灯パターンが移り変わります。上表の最後までいくと、スイッチの状態が LED に反映される動作モードとなります。プッシュスイッチを押すと LED が点灯、DIP スイッチを上側（ON 側）に倒すと LED が点灯となります。

1.2. プロジェクトとツリーとソースファイル

・プロジェクトツリー



RA2L1_SW_LED プロジェクトを e2studio の環境にインポートすると上記の様なツリーとなります。この中でユーザが作成、編集する部分は、基本的には src 以下と Configuration.xml（FSP の設定）のみです。

・src 以下の構成

led	led.c	LED 制御関数
	led.h	LED 制御ヘッダ
sw	sw.c	スイッチ制御関数
	sw.h	スイッチ制御ヘッダ
hal_entry.c		メイン処理

1.3. LED 制御のハード的な側面

まず、LED がマイコンとどの様に接続されているか（回路図）を理解する必要があります。

・LED とマイコンの接続

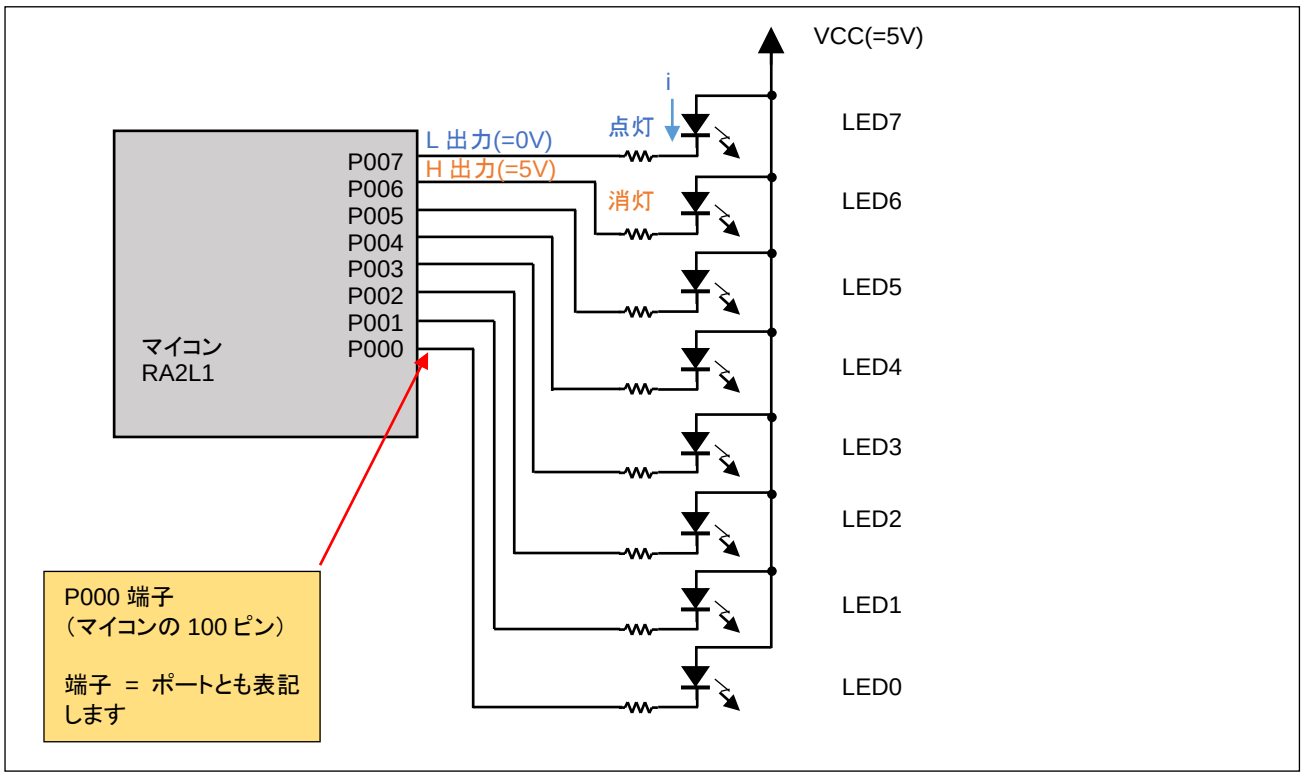


図 1-1 LED の接続

LED とマイコンは図の様に接続されています。ボードに 5V の電源を印加した場合、VCC と書かれている場所は 5V となります。

ここで、マイコンの P007 端子を L 出力制御すると、LED7 の両端に電圧の差が生じ、電流（図中の i）が流れますので、LED7 が点灯する事となります。

・端子と LED の関係

P007=L 出力	LED7 は点灯
P007=H 出力	LED7 は消灯

ここで、注意したいのが、LED とマイコンの端子の接続状態です。

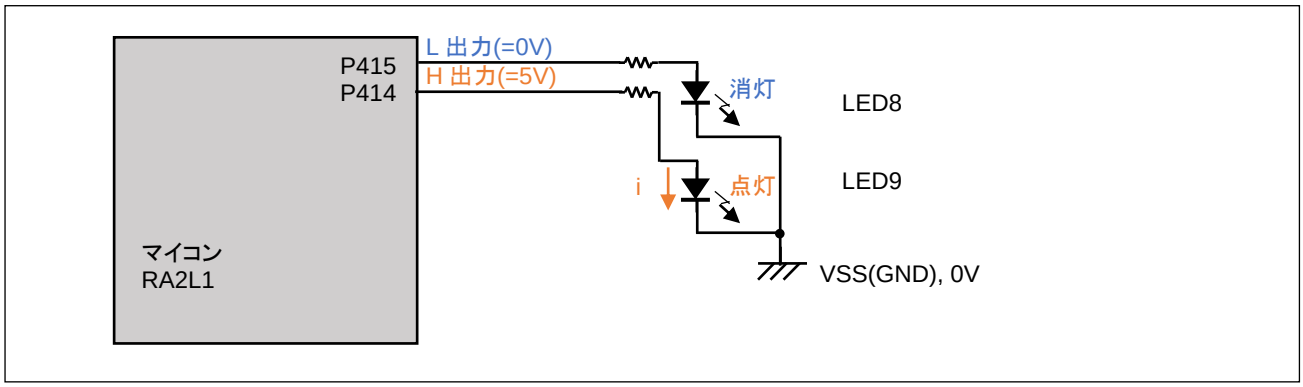


図 1-2 LED の接続(2)

回路図が LED の接続(2)の様になっている場合は、まったく逆の結果（L 出力で消灯、H 出力で点灯）となります。要は回路図に応じて、制御を考えなくてはならないという事です。

1.4. LED 制御の手法

LED の点灯・消灯を制御する場合に使用するマイコンの機能は「I/O ポート」です。「I/O ポート」の機能では、マイコンの端子を

・出力モード
　出力は H とする
　出力は L とする
・入力モード

に設定可能です。

出力の L/H の切り替えは、マイコンの特定のアドレスにデータを書き込む事で行われます。

アドレスとしては、0x4004 0020 です。このアドレスを操作すれば良いのですが、アドレスというのは数字の羅列で覚えにくいし、取り違いを起こしやすいものだと思います。

そこで、開発環境側で予め、アドレスと「名前」の対応表を用意してくれています。対応表はファイルになっており、
[プロジェクトフォルダ]¥ra¥fsp¥src¥bsp¥cmsis¥Device¥RENESAS¥Include¥renesas.h
です。このファイルは基本的には、プログラマが意識しなくとも良いです。

プログラムでは、

hal_entry.c[抜粋]

```
#include "hal_data.h"
```

という 1 行を書いておけば良いです。(ツールが自動生成するテンプレートファイルに予め記載されています)

このファイルを使用すると、先程のアドレスが

R_PORT0->PODR

という名称でアクセスできる様になります。端子名が P0(P000~P007, ポートの 0)なので、PORT0。Port Output Direction Register 略で、PODR です。ポートの出力の L/H を決めるレジスタ(記憶領域)です。

なお、P000 端子が出力か入力かを決めるのは、

R_PORT0->PDR

となります。Port Direction Register の略です。本プログラムでは、PDR は FSP の GUI で設定してしまっているので、ユーザが書き下すプログラムコードには出てきません。

R_PORT0->PODR は、16bit(Word)のレジスタとなっており、Word 単位でアクセスする場合は

R_PORT0->PODR = 0x001F;

の様に記載します。

MSB(b15) LSB(b0) ※MSB(Most Significant Bit)最上位ビット, LSB(Least..)最下位ビット

000x1F = 0b00000000 00011111

ですので、

b15-8	b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	0	1	1	1	1	1
P015~008(*1) =L	P007 =L	P006 =L	P005 =L	P004 =H	P003 =H	P002 =H	P001 =H	P000 =H
ボードでは未使用	LED7 点灯	LED6 点灯	LED5 点灯	LED4 消灯	LED3 消灯	LED2 消灯	LED1 消灯	LED0 消灯

(*1)RA2L1 の 100 ピンのマイコンでは存在しないポートもあります

となります。当該ビットを 0 にした場合、端子からは L が出力され、1 にした場合は端子からは H が出力されます。

なお、R_PORT0->PODR レジスタは、ビット単位でのアクセスも定義されており、

R_PORT0->PODR_b.PODR3 = 1;

と書けば、bit3, P003 を H にするという意味となります。

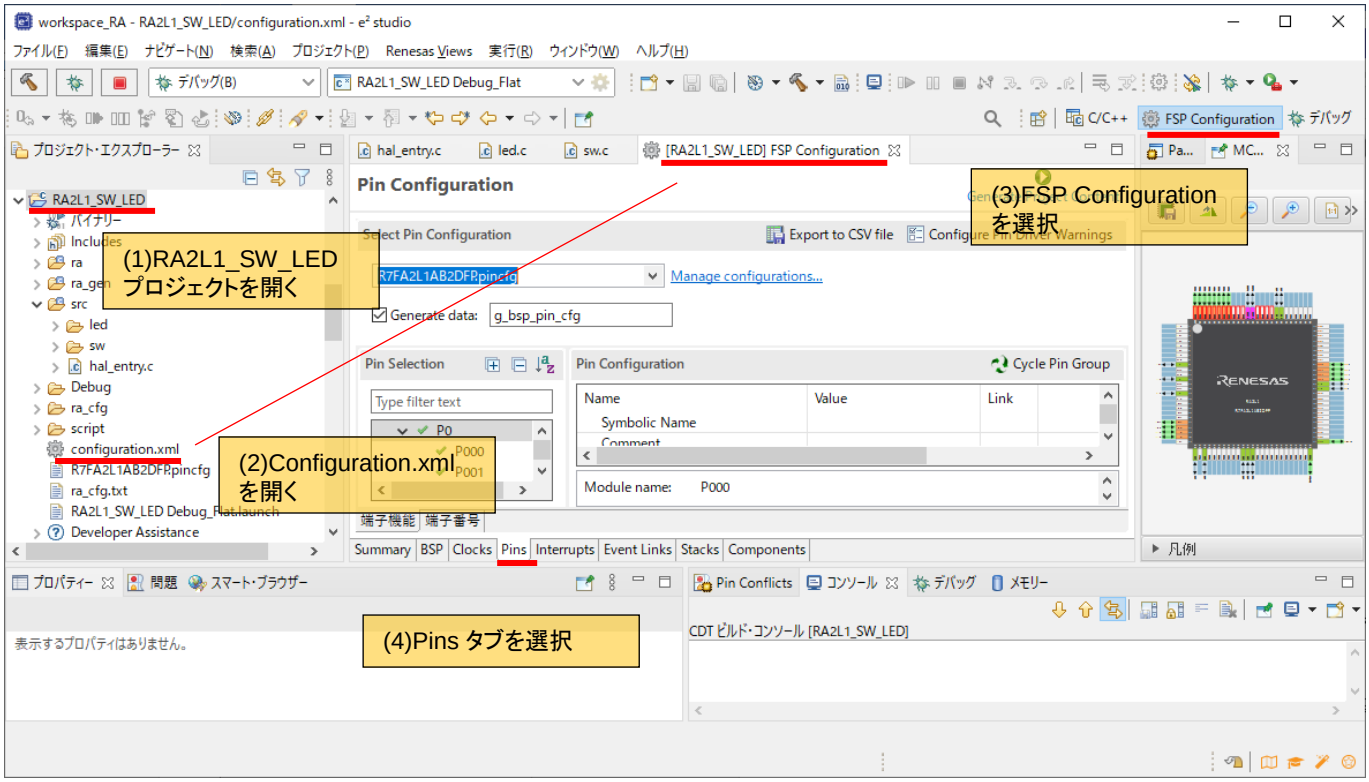
なお、端子の入力、出力方向を決めるレジスタは、

R_PORT0->PDR.PDR = 0x00FF;

→P000~P007 を出力に設定。bit を 0 にすると入力モード、1 にすると出力モードとなります。(本チュートリアルでは、ユーザが書き下すプログラムコードでは設定していません)

1.5. FSP の設定(LED)

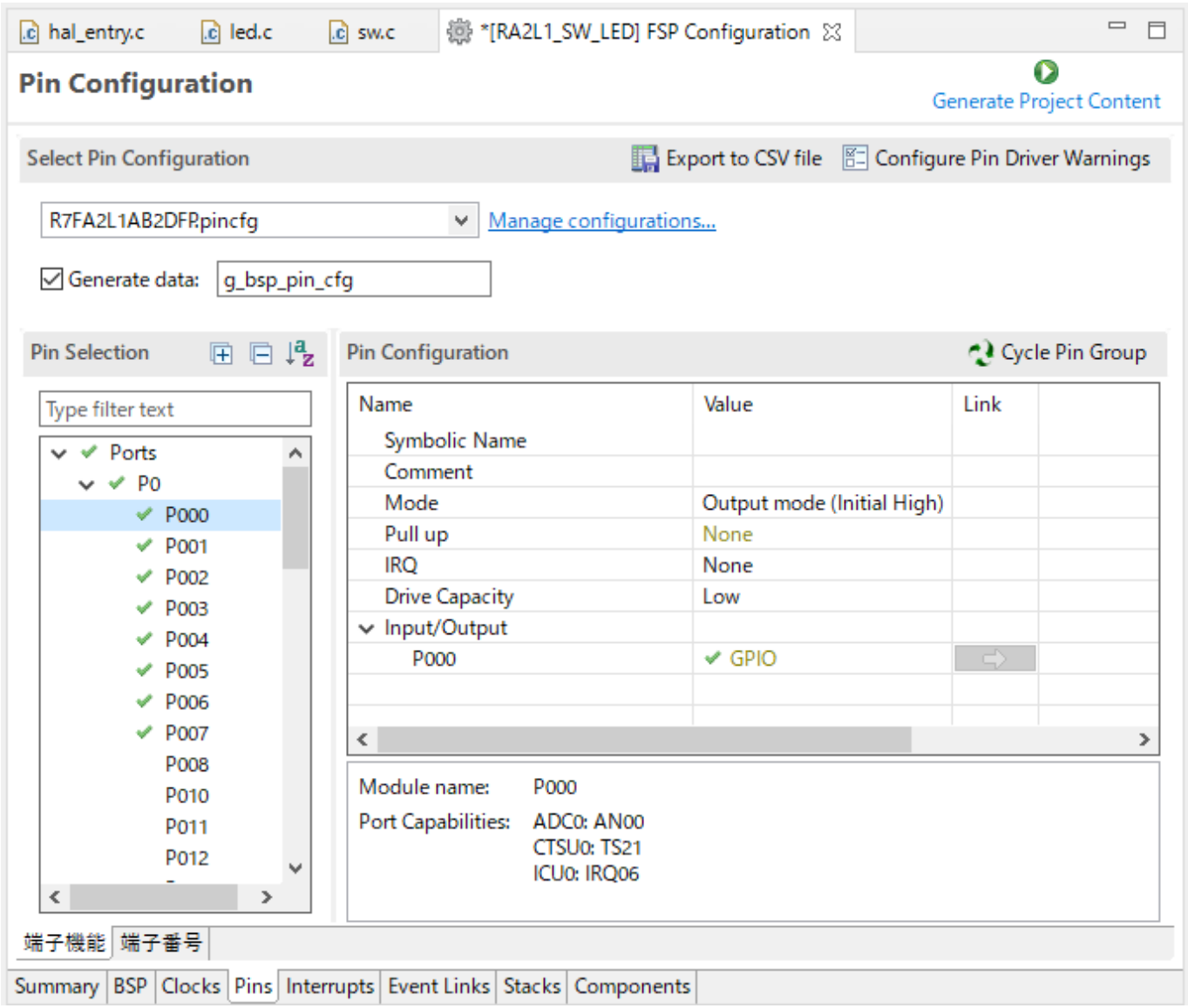
端子の入出力区分等は、FSP にて設定可能です(本チュートリアルでも、FSP で設定しています)。



(1)当該プロジェクト(ここでは、RA2L1_SW_LED)を開いた状態で
(2)プロジェクト・エクスプローラのプロジェクトツリーの Configuration.xml をダブルクリックで開く
(3)右上のペイン(表示領域)選択で、FSP Configuration を選択

(「C/C++」はソースを作成、変更する際、「FSP Configuration」は、FSP の設定を行う際、「デバッグ」はデバッグを行う際に、良く使用する項目がアクティブになります。作業に応じて適切な選択を行う様にしてください。)

(4)Pins タブを選択



Ports－P0－P000 を開き、

Mode を　Output mode(Initial High)を選択します。

Drive Capacity は Low（初期値）で構いません。

※RA2L1 には、Drive Capacity（駆動能力）切り替え機能はありません

Mode は

選択肢	PDR	PODR	備考
Disable	0	-	I/O ポートを使わない設定（実質的には Input と同じ）
Input mode	0	-	入力モード
Output mode (Initial Low)	1	0	出力モード（L 出力）
Output mode (Initial High)	1	1	出力モード（H 出力）

となります。ここでは、出力モードに設定して、最初は LED を OFF（消灯）にしておきたいので、「Output mode (Initial High)」を選択します。P001~P007 も同様の設定としてください。この画面で設定を行い、「コード生成」（後述）を行うと、設定が反映されたプログラムコードが自動的に出力されます。

1.6. スイッチとマイコンの接続

次に、スイッチがマイコンとどの様に接続されているか（回路図）を理解する必要があります。

・スイッチとマイコンの接続

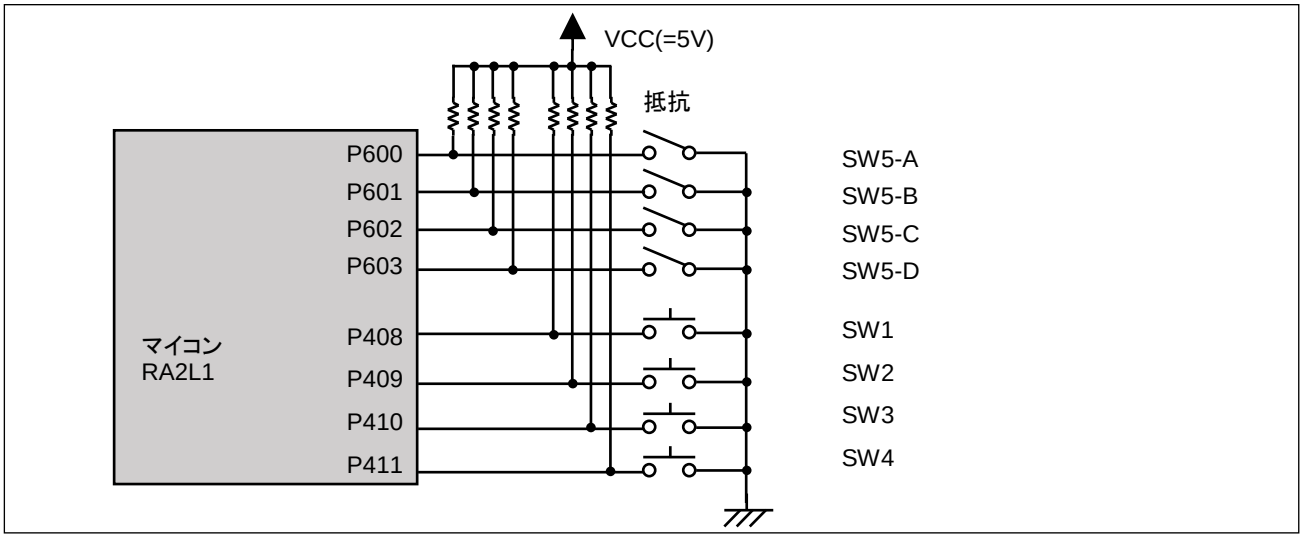


図 1-3 スイッチの接続

マイコンとスイッチ(SW5:DIP-SW, SW1~SW4:Push-SW)は、上図の様に接続されています。スイッチをオフ状態とした場合は、マイコンの端子には H が入力されます。スイッチをオン状態とした場合は、端子は L となります。

この様な接続を、抵抗で上（電源側）に引き上げることから、プルアップと呼びます。

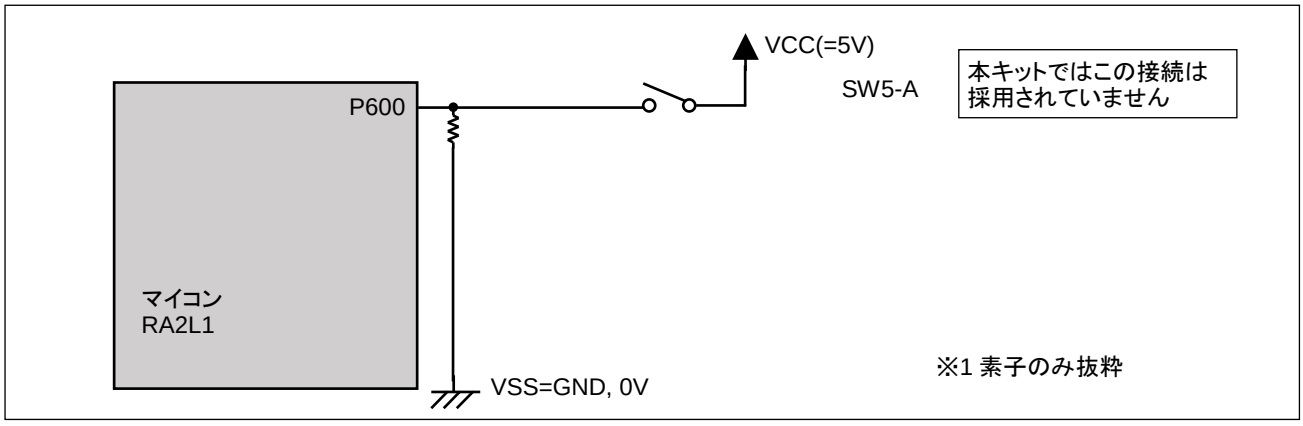
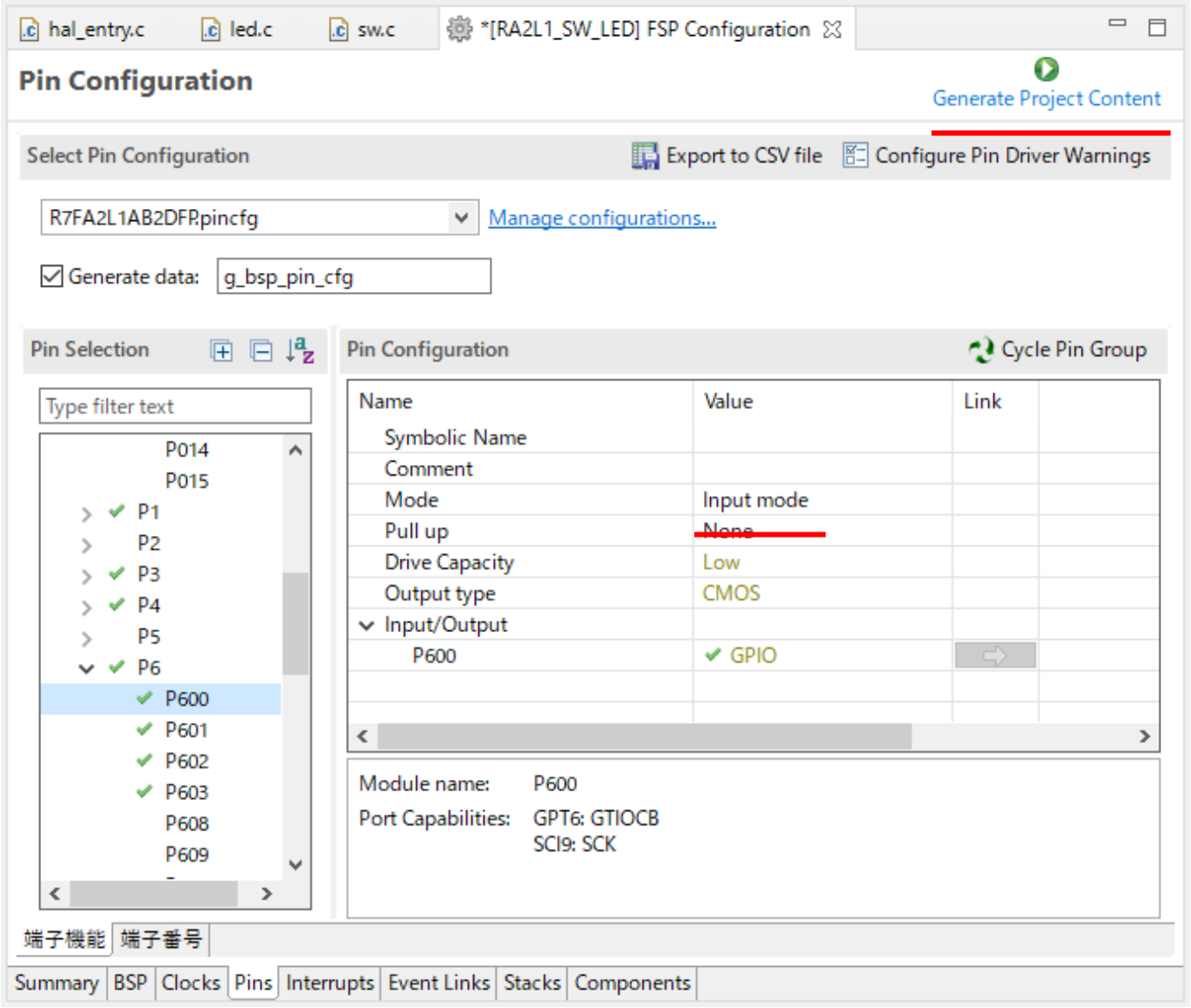


図 1-4 スイッチの接続(2), プルダウン

図 1-4 の様にスイッチとマイコンを接続した場合は、スイッチオン時、端子は H。スイッチオフ時、端子は L となります。図 1-3 の接続とは、極性が逆になります。マイコンとスイッチがどの様に接続されているかは、プログラムを作成する上でも必要な情報となります。

マイコンの端子をスイッチ入力として使用するためには、マイコン側から出力するモードに設定してはいけません。

1.7. FSP の設定（スイッチ）



Ports－P6－P600 を開き、
Mode を　Input mode を選択します。
Pull up は None（初期値）で構いません。
※Pull up はボード上に存在するので、マイコン内蔵の Pull up は OFF で問題ありません
（有効にしても問題はありません）
P601~P603, P408~P411 も同様の設定としてください。

一通り、設定が終われば、「Generate Project Content」を押してください。ツールで設定した項目が、ソースコードに反映されて出力されます。

1.8. スイッチの読み取り手法

次に、ポートのレベル(L/H)を読み取る方法ですが、

R_PORT0->PIDR

というのが、端子が L レベルか H レベルかを読み取るレジスタとなります。16bit(P000~P015)一括での読み取り結果です。（RA2L1 で存在しない端子は、基本的に読み出し結果が常に 0 となります。※正式には存在しない端子の値は未定義）

ビット（1 端子）毎の読み取りの場合は、

R_PORT0->PIDR_b.PIDR3

と指定すれば、P003 端子のレベルを読み取ることができます。

PDR　ポートの入出力方向の指定（0:入力, 1:出力）※FSP で設定しているのでユーザプログラムでは未使用
PODR　出力ポートのレベル（0:L レベル, 1:H レベル）
PIDR　ポートの入力レベル（0:L レベル, 1:H レベル）

上記の 3 つのレジスタは、I/O ポートを制御する上で一番使用するレジスタなので、覚えておいてください。

1.9. プログラムソースの説明(LED)

本プロジェクトのプログラム本体は以下の様になります。

・hal_entry.c

```
#include "led/led.h"
#include "sw/sw.h"

void hal_entry(void)
{
    /* TODO: add your own code here */

    unsigned char pattern;

    led_on_1_tmp(0);        //LED0をON

    wait_1s();              //1秒ウェイト

    led_on_2_tmp(1);        //（別な関数で）LED1をON

    wait_1s();              //1秒ウェイト

    led_on_3_tmp(2);        //（別な関数で）LED2をON

    wait_1s();              //1秒ウェイト

    led_on(3);              //（別な関数で）LED3をON

    wait_1s();              //1秒ウェイト

    led_set(0x5a);          //LED7,..,0を  OFF, ON, OFF, ON, OFF, ON, OFF, ONにセット

    wait_1s();              //1秒ウェイト

    pattern = led_get();                //現在のLEDの値を取得
    pattern = (unsigned char)~pattern;  //状態を反転
    led_set(pattern);                   //反転させた結果を反映

    wait_1s();

    //DIP-SWとPush-SWの情報をLEDに反映させる
    while(1)
    {
        //スイッチ読み取り関数を変えてみて挙動が同じかどうか確かめる
        //pattern = sw_read_1_tmp();
        //pattern = sw_read_2_tmp();
        //pattern = sw_read_3_tmp();
        pattern = sw_read();

        led_set(pattern);
    }
}
```

←自作関数のヘッダファイルです
hal_entry 内自作の関数を使用する場合、関数のプロトタイプ宣言をしている
ヘッダファイルを追加するようにしてください

前半部分

後半部分

led_on_1_tmp()

led_on_2_tmp()

led_on_3_tmp() = led_on()

上記の関数は全て同じ動作となります。指定した引数(0~7)の LED を点灯させる関数です。但し、ポートの駆動のコードが微妙に異なります。

指定した LED を ON する関数を 3 種類作成してみました。動作はどの関数でも同一です。

(1)led_on_1_tmp()

```
void led_on_1_tmp(unsigned char no)
{
    //引数として与えたLEDの番号(0~7)のLEDを点灯させる関数（その1）

    //引数
    //  unsigned char no : 点灯させたいLEDの番号を指定(no=0-7)

    //戻り値
    //  なし

    //switch文で条件分岐させ、PODR（ポート出力レジスタ）を設定する方法です

    switch(no)
    {
        case 0:
            R_PORT0->PODR_b.PODR0 = LED_ON;     //LED_ON→0がled.h内で定義されています
            break;

        case 1:
            R_PORT0->PODR_b.PODR1 = LED_ON;
            break;

        case 2:
            R_PORT0->PODR_b.PODR2 = LED_ON;
            break;

        case 3:
            R_PORT0->PODR_b.PODR3 = LED_ON;
            break;

        case 4:
            R_PORT0->PODR_b.PODR4 = LED_ON;
            break;

        case 5:
            R_PORT0->PODR_b.PODR5 = LED_ON;
            break;

        case 6:
            R_PORT0->PODR_b.PODR6 = LED_ON;
            break;

        case 7:
            R_PORT0->PODR_b.PODR7 = LED_ON;
            break;

        default:
            break;                              //0~7以外の数値の場合は何もしない
    }
}
```

switch case 文を用いて指定した番号の LED を ON させるプログラムになっています。

LED1 を ON させる場合、

R_PORT0->PODR_b.PODR1 = 0;

です。P001、ポート（グループ）0 の 1 番（端子）を L レベル(=0)に設定します。

LED_ON は、led.h 内で

#define LED_ON (0)

と定義されています。

(2)led_on_2_tmp()

```
void led_on_2_tmp(unsigned char no)
{
    //引数として与えたLEDの番号(0~7)のLEDを点灯させる関数（その2）

    //引数
    //  unsigned char no : 点灯させたいLEDの番号を指定(no=0-7)

    //戻り値
    //  なし

    //0x1をビットシフト演算と反転の演算を使って処理しています
    //no=3のとき、
    //0x1 << no : 0x8(0b1000)となり、その値を反転させると
    //（int型で）0xffff fff7(0b1111 1111 1111 1111 1111 1111 1111 0111)となります
    //現在のPORT0のPODRと0xfff7(0b1111 1111 1111 0111)のANDを取れば、bit3だけ0に変更できます

    if(no > 7) return;      //8以上の番号が指定されたときは何もしない

    R_PORT0->PODR &= (unsigned short)~(0x1 << no);

}
```

別バージョンです。(1)のプログラムは、愚直に処理していて処理が見た目に判り易いですが、ちょっと煩雑です。本プログラムでは、ビットシフト(<<)演算と AND 演算を使って、特定のビットのみ 0 に変更する様にしています。

(3)led_on_3_tmp()

```
void led_on_3_tmp(unsigned char no)
{
    //引数として与えたLEDの番号(0~7)のLEDを点灯させる関数（その3）

    //引数
    //  unsigned char no : 点灯させたいLEDの番号を指定(no=0-7)

    //戻り値
    //  なし

    //PCNTR3.PORRは、0b0000 0000 0000 1000(=0x8)を書き込むと、P03をLに変更するレジスタです

    if(no > 7) return;      //8以上の番号が指定されたときは何もしない

    R_PORT0->PCNTR3_b.PORR = (unsigned short)(0x1 << no);

    //led_on_2_tmpは、1行で書いていますが
    //unsigned short tmp
    //tmp = R_PORT0->PODR;                  //レジスタの読み込み（リード）
    //tmp &= (unsigned short)~(0x1 << no);  //レジスタ値の変更（モディファイ）
    //R_PORT0->PODR = tmp;                  //レジスタの書き込み（ライト）
    //処理としては、3ステップ（リード・モディファイ・ライト）の処理となります
    //それに対し、本関数ではレジスタの書き込みのみで処理が完結しています

    //PCNTR3というレジスタは、RXマイコンでプログラムを行っていた方には馴染みのないレジスタかもしれませんが
    //リード・モディファイ・ライト処理を簡略化できる便利なレジスタです

    //実験的に3パターンの処理を考えましたが、本関数(led_on_3_tmp)が一番シンプルですので、こちらを本番の関数
(led_on)で採用します

}
```

更に別バージョンです。本プログラムでは、新たなレジスタである、PORR を使っています。

このレジスタは便利なレジスタで、PODR の特定のビットを 0 に変更することができるものです。LED3 を ON にする場合、PODR を使って制御する場合、

tmp = R_PORT0->PODR;　　//tmp に現在の PODR の値を保存
tmp = tmp & 0xFFF7;　　//bit3 のみ 0 に変更する
R_PORT0->PODR = tmp;

の 3 ステップの動作となります。レジスタからの読み出し、レジスタ値の変更、レジスタへの書き込みの処理で、この様な処理を「リード・モディファイ・ライト」と言います。マイコンのレジスタ値を変更する、一番一般的な処理フローです。

それに対し、PCNTR3.PORR を使った場合、

R_PORT0->PCNTR3_b.PORR = 0x0008;　　//bit3 のみ、0 に変更する。

プログラム的には、1 行で完結します。

RA マイコンには、1 ステップでポートの値を変更できる、PCNTR3 というレジスタがあるという事を覚えておくと役に立つ場面があるかも知れません。

PCNTR3 は、32bit のレジスタで

b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
PORR 15	PORR 14	PORR 13	PORR 12	PORR 11	PORR 10	PORR 9	PORR 8	PORR 7	PORR 6	PORR 5	PORR 4	PORR 3	PORR 2	PORR 1	PORR 0

b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
POSR 15	POSR 14	POSR 13	POSR 12	POSR 11	POSR 10	POSR 9	POSR 8	POSR 7	POSR 6	POSR 5	POSR 4	POSR 3	POSR 2	POSR 1	POSR 0

PORR(Port Output Reset Resistor)
POSR(Port Output Set Resistor)

で、1 を書くと、書いたところの PODR が Reset（0 になる）か Set（1 になる）という動作となります。

例えば、LED3 を ON（PODR3=0)させて、LED2 を OFF(PODR2=1)させる場合、

```
R_PORT0->PCNTR3_b.PORR=0x0008;      //PODR を 0 にしたいビットを指定
R_PORT0->PCNTR3_b.POSR=0x0004;      //PODR を 1 にしたいビットを指定
```

となります。さらに、これをまとめると、

```
R_PORT0->PCNTR3 = 0x00080004;
```

となります。1 回のレジスタ操作で、L にしたい端子と H にしたい端子を操作できるので、便利です。

※R_PORT0->PCNTR3 = 0x00080008;　の様に、PORR と POSR の両方のビットを立てる操作は禁止されています

この、led_on_3_tmp()が一番短いので、この関数を led_on()関数として採用する事とします。

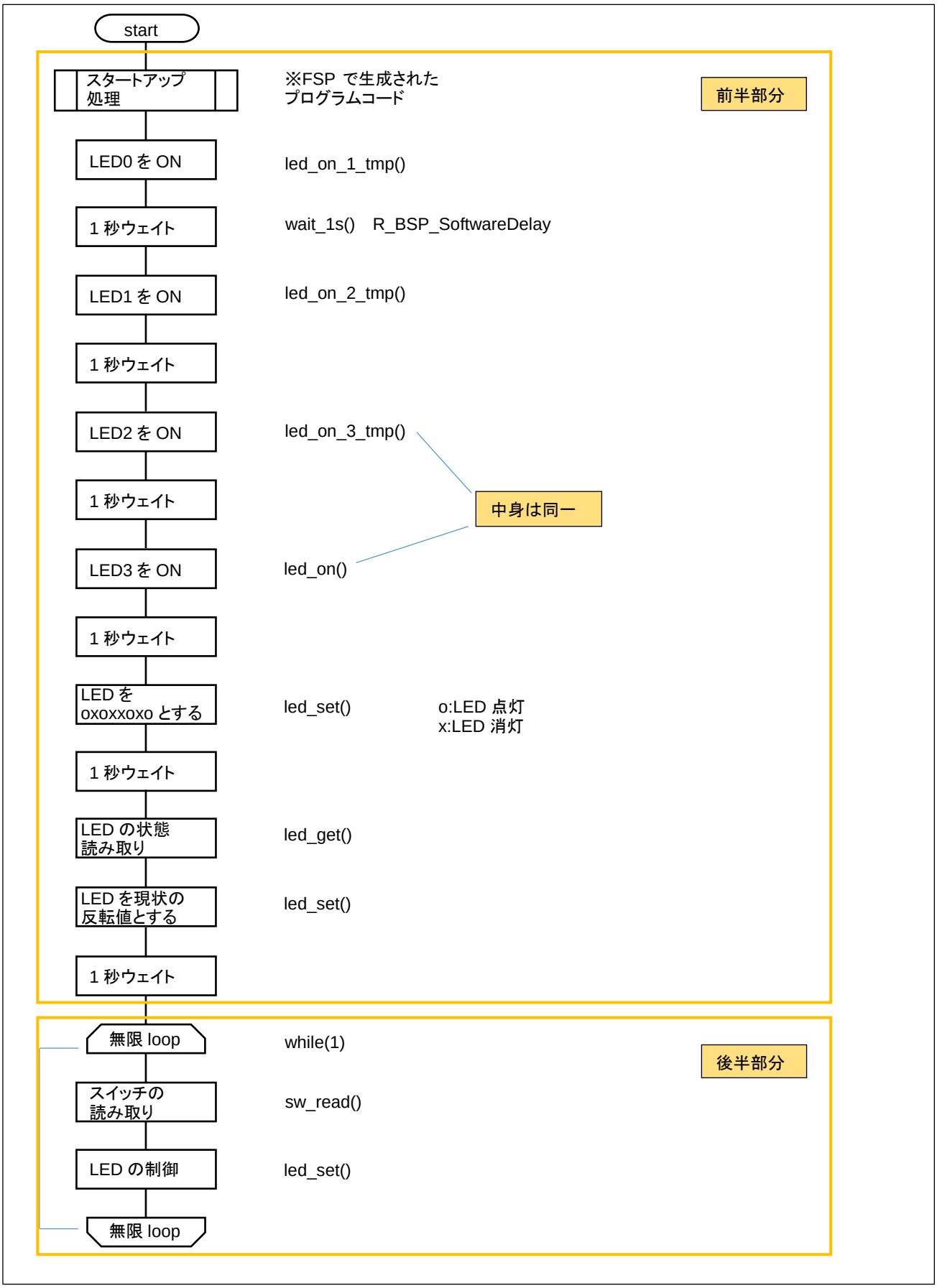
led_set()関数は、8 個の LED を同時に操作する関数です。

```
led_set(0x1F);
```

で、LED7~5 は ON。LED4~0 は OFF となります。

led_get()関数は、現在の LED の ON/OFF を取得する関数です。

1.10.フローチャート


start
スタートアップ処理
※FSP で生成されたプログラムコード
前半部分
LED0 を ON
led_on_1_tmp()
1 秒ウェイト
wait_1s()　R_BSP_SoftwareDelay
LED1 を ON
led_on_2_tmp()
1 秒ウェイト
LED2 を ON
led_on_3_tmp()
1 秒ウェイト
中身は同一
LED3 を ON
led_on()
1 秒ウェイト
LED を oxoxxoxo とする
led_set()
o:LED 点灯
x:LED 消灯
1 秒ウェイト
LED の状態読み取り
led_get()
LED を現状の反転値とする
led_set()
1 秒ウェイト
無限 loop
while(1)
後半部分
スイッチの読み取り
sw_read()
LED の制御
led_set()
無限 loop

全体のフローチャートは上記となります。

前半は、3 種類の LED を ON にする関数で、LED0 から LED3 まで順次点灯させていきます。

LED を ON させる関数を変えても（LED を制御する構文が多少異なっていても）、動作は変わりません。

その後、LED を一括で制御する led_set()を使って、LED0~LED7 を交互に点灯（及びその反転）させる様な動作となります。

後半は、これから説明しますが、スイッチの読み取りを行い、LED の ON/OFF に反映させています。SW1~4 が LED0~3 に、SW5-A~D が LED4~7 に 1:1 で対応します。

1.11.プログラムソースの説明（スイッチ）

スイッチを読み取りる関数も３種類作ってみました。（プログラムで使っているのは１種類だけです。他はコメントアウトしています。コメントアウトを外して、他の関数を使っても動作は変わりません。）

DIP-SW とプッシュ SW を読み取る関数 3 種類です。

(1)sw_read_1_tmp()

```
unsigned char sw_read_1_tmp(void)
{
    //SWの状態を読み取る関数（その1）

    //引数
    //  なし

    //戻り値
    //    bit7 = SW5-D
    //    bit6 = SW5-C
    //    bit5 = SW5-B
    //    bit4 = SW5-A
    //    bit3 = SW4
    //    bit2 = SW3
    //    bit1 = SW2
    //    bit0 = SW1
    //    各スイッチの状態を8bitで返す

    unsigned char ret = 0x00;   //初期値は全部ONの状態とする

    if(R_PORT6->PIDR_b.PIDR3 == SW_OFF)     //SW5-D
    {
        ret |= 0x80;                        //bit7を1にする
    }

    if(R_PORT6->PIDR_b.PIDR2 == SW_OFF)     //SW5-C
    {
        ret |= 0x40;                        //bit6を1にする
    }
```

sw_read_1_tmp()[続き]

```
    if(R_PORT6->PIDR_b.PIDR1 == SW_OFF)     //SW5-B
    {
        ret |= 0x20;                        //bit5を1にする
    }

    if(R_PORT6->PIDR_b.PIDR0 == SW_OFF)     //SW5-A
    {
        ret |= 0x10;                        //bit4を1にする
    }

    if(R_PORT4->PIDR_b.PIDR11 == SW_OFF)    //SW4
    {
        ret |= 0x08;                        //bit3を1にする
    }

    if(R_PORT4->PIDR_b.PIDR10 == SW_OFF)    //SW3
    {
        ret |= 0x04;                        //bit2を1にする
    }

    if(R_PORT4->PIDR_b.PIDR9 == SW_OFF)     //SW2
    {
        ret |= 0x02;                        //bit1を1にする
    }

    if(R_PORT4->PIDR_b.PIDR8 == SW_OFF)     //SW1
    {
        ret |= 0x01;                        //bit0を1にする
    }

    return ret;
}
```

単純に、1 ポートずつ読み取っていき、スイッチが OFF ならば、

#define SW_OFF (1)

戻り値(8bit の変数）のビットを 1 にするというものです。

sw_read_1_tmp()の戻り値は、b7-b4 が DIP-SW(SW5), b3-b0 がプッシュスイッチ(SW4-1)です。

(2)sw_read_2_tmp()

```
unsigned char sw_read_2_tmp(void)
{
    //SWの状態を読み取る関数（その2）

    //引数
    //  なし

    //戻り値
    //    bit7 = SW5-D
    //    bit6 = SW5-C
    //    bit5 = SW5-B
    //    bit4 = SW5-A
    //    bit3 = SW4
    //    bit2 = SW3
    //    bit1 = SW2
    //    bit0 = SW1
    //    各スイッチの状態を8bitで返す

    unsigned char ret = 0xff;   //初期値は全部OFFの状態とする

    if(R_PORT6->PIDR_b.PIDR3 == SW_ON)      //SW5-D
    {
        ret &= 0x7f;                        //bit7を0にする
    }

    if(R_PORT6->PIDR_b.PIDR2 == SW_ON)      //SW5-C
    {
        ret &= 0xbf;                        //bit6を0にする
    }

[中略]

    if(R_PORT4->PIDR_b.PIDR8 == SW_ON)      //SW1
    {
        ret &= 0xfe;                        //bit0を0にする
    }

    return ret;

    //本関数は、sw_read_1_tmpを色々逆にしただけです
    //ret &= の部分は、sw_read_1_tmpの方が判り易いかと思います
    //SW5(DIP-SW)の状態はどちらになっているかは、プログラム作成時には判断できませんが
    //SW1~4(PUSH-SW)は、押されていない状態(OFF)がデフォルトである（押されていない確率が高い）と推測されます
    //SW=OFFのときの処理が
    //sw_read_1_tmp: 条件判断(if)＋代入(ret |=)
    //sw_read_2_tmp: 条件判断(if)
    //となり、本関数の方が処理が軽くなります

}
```

基本的に、sw_read_1_tmp()と変わりませんが、スイッチが OFF ならば、とスイッチが ON ならばの条件判断が逆になっています。スイッチが OFF の出現頻度が高いとすれば、こちらのプログラムの方が多少処理が軽くなるかと思います。

(2)sw_read_3_tmp()

```
unsigned char sw_read_3_tmp(void)
{
    //SWの状態を読み取る関数

    //引数
    //  なし

    //戻り値
    //    bit7 = SW5-D
    //    bit6 = SW5-C
    //    bit5 = SW5-B
    //    bit4 = SW5-A
    //    bit3 = SW4
    //    bit2 = SW3
    //    bit1 = SW2
    //    bit0 = SW1
    //    各スイッチの状態を8bitで返す

    return (unsigned char)( ((R_PORT6->PIDR & 0x000f) << 4) |  ((R_PORT4->PIDR & 0x0f00) >> 8) );

    //P600~603の状態を4bitシフトさせbit7-bit4に、P408~411の状態を8bitシフトさせbit3-bit0にセットします
}
```

DIP-SW(PORT6)とプッシュスイッチ(PORT4)の値を、ビットシフトさせて、8 ビットの読み取り結果に反映させています。コードが短くなるので、本コードを、sw_read()関数に採用します。

以上、LED の ON/OFF の関数と、スイッチの読み取り関数を何種類か作ってみるというプロジェクトです。switch case 文を使ったり、ビットシフト演算を用いたり、PCNTR3 というレジスタを使ったりしています。

LED の ON/OFF：マイコンからすると、I/O ポートの出力制御
スイッチの読み取り：マイコンからすると、I/O ポートの入力

は、組み込みプログラムとしては、まず最初に取り組むプログラムです。C 言語では、画面に「Hello, world」と表示させるプログラムが入門書の一番最初に書いてあったりしますが、組み込みプログラムの世界では LED をチカチカさせるプログラムは、「L チカ」などと呼ばれ、誰もが最初にプログラムする題材です。

L チカでも、何種類かの書き方があるという事を理解して頂ければ、このプロジェクトの目的は達成します。

※なお、どの書き方が効率が良いかは、本ドキュメントの最後にコラムとしてまとめています。

1.12.本チュートリアルで定義した関数の説明

－本チュートリアルで定義されている関数（抜粋）－
led¥led.c

led_on

概要：LED を ON（点灯）させる関数
宣言：
　　void led_on(unsigned char no)
説明：
　・指定した番号の LED を ON させる処理
を行います

引数：
　　unsigned char no: 0~7 の LED 番号を指定します
戻り値：
　　なし

led_off

概要：LED を OFF（消灯）させる関数
宣言：
　　void led_on(unsigned char no)
説明：
　・指定した番号の LED を ON させる処理
を行います

引数：
　　unsigned char no: 0~7 の LED 番号を指定します
戻り値：
　　なし

led_set

概要:8 個の LED を指定したパターンで点灯させる関数
宣言:
　　void led_set(unsigned char pattern)
説明:
　・LED の OF, OFF を指定したパターンとする処理
を行います

引数:
　　unsigned char pattern: 8bit で点灯(0)消灯(1)を指定します（b7:LED7, b0:LED0)
戻り値:
　　なし

led_cntl

概要:LED の点灯、消灯の状態を変化させる関数
宣言:
　　void led_cntl(unsigned char on_pattern, unsigned char off_pattern)
説明:
　・指定したパターンの LED を ON/OFF させる処理
を行います

引数:
　　unsigned char on_pattern: 8bit で、(1)を指定したビットの LED を ON に変化させます
　　unsigned char off_pattern: 8bit で、(1)を指定したビットの LED を OFF に変化させます
戻り値:
　　なし

使用例:
　　led_cntl(0x4, 0x2);　　//LED2 を ON に変更、LED1 を OFF に変更
補足:
　led_set()関数は 8 個全ての LED のパターンを与える必要がありますが、本関数は変更、もしくは設定したい LED のみ設定します。引数のビットを立てていない LED は、現状の状態を維持します。

led_get

概要:8 個の LED の状態を取得する関数
宣言:
　　unsigned char led_get(void)
説明:
　・LED の ON, OFF の状態を取得する処理
を行います

引数:
　　なし
戻り値:
　　8bit で点灯(0)消灯(1)の状態を返します

sw¥sw.c

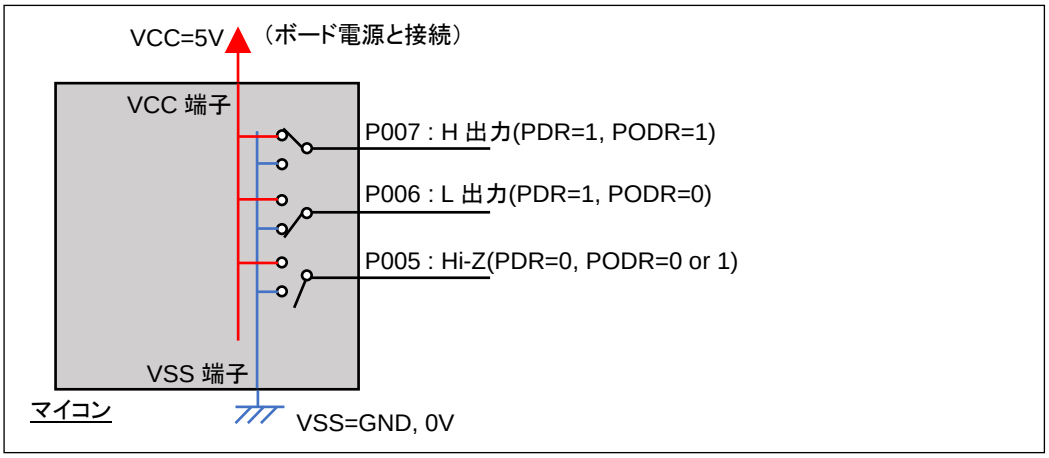
sw_read

概要:スイッチの状態を取得する関数
宣言:
　　unsigned char sw_read(void)
説明:
　・スイッチの ON, OFF の状態を取得する処理
を行います

引数:
　　なし
戻り値:
　　8bit でスイッチの状態を返します ON(0), OFF(1)
　　b7 SW5-D
　　b6 SW5-C
　　b5 SW5-B
　　b4 SW5-A
　　b3 SW4
　　b2 SW3
　　b1 SW2
　　b0 SW1

コラム　L 出力, H 出力とは

デジタル回路では、L/H,0/1 という表現を良く使います。



H 出力(1)とは P007 等の端子につながっているマイコンの中のスイッチが VCC に接続されている状態です。

ボード電源電圧（=マイコン VCC 端子電圧）が 5V のときは、5V となります。ボード電源電圧が 3.3V のときは、H 出力は 3.3V となります。

L 出力(0)とは、マイコンの中のスイッチが VSS(GND)に接続されている状態で、0V となります。

PDR を 0 に設定した場合は、スイッチが VCC にも VSS にもつながらない状態となります。この様な状態を、ハイ・インピーダンス状態といい、記号では、Z または Hi-Z と表します。端子を入力端子として使用する場合はこの設定にします。

H,L,Z の 3 状態を設定できる回路を、トライステート（3 条件）バッファなどと呼ぶ事があり、マイコンの I/O ポートは、ほとんどがトライステートバッファの構成となっています。

H 出力は、デジタル的には 1 と表現しますが、ボード（システム）の電源電圧により、電気的には 5V だったり 3.3V だったりしますので、注意が必要です。

端子の真理値表を示します。端子を入力モードに設定した場合は、LED は消灯となります。

真理値表

PDR	PODR	端子	備考
0	X	Z	LED は消灯
1	0	L	LED は点灯
1	1	H	LED は消灯

コラム　特定のビットのみ変更する演算

2 進数(0b00010010)、10 進数(18)、16 進数(0x12)に関しては、本書では解説を省略します。2 進数と 16 進数は、本書でも頻繁に出てきますので、ピンと来ないという方は、プログラムの入門書等で勉強してください。

プログラムを行う際、1 バイトの変数の特定のビットのみ 0 に変更したいといった処理がよく出てきます。

0bXXX0000X　　(1)

(1)で、0 のところは 0 に設定したい。X のところは、現在の設定値を変えたくないとします。

ここで、(1)の X のビットを、1 とした値を考えます

0b11100001

です。16 進数では、0xE1 となります。

```
unsigned char a = 0xD9;
a = a & 0xE1;
```

とすれば、（&は、ビット単位での AND 演算）

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
元の値 a=0xD9 とする	1	1	0	1	1	0	0	1
0xE1	1	1	1	0	0	0	0	1
0xD9 & 0xE1	1	1	0	0	0	0	0	1

bit1,2,3,4 を 0 に変更する事ができます。上記式は

```
a &= 0xE1;
```

と演算、代入を 1 つの演算子(&=)で行う記載でも等価で、こちらの方がスマートです。

今度は、逆の操作

0bXXX1111X　　(2)

bit1,2,3,4 のみ 1 にしたい場合はどうすれよいのでしょうか。(2)の X のところは 0, 1 のところは 1 の値を作り、

0b00011110 = 0x1E

元の数と OR を取れば良いです。

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
元の値 a = 0xD9 とする	1	1	0	1	1	0	0	1
0x1E	0	0	0	1	1	1	1	0
0xD9 | 0x1E	1	1	0	1	1	1	1	1

```
unsigned char a = 0xD9;
a |= 0x1E;
```

です。

特定ビットを 0 にしたい時は、変更したくないところを 1 とした値を作り、特定のビットを 0 にしたい時は、変更したいところを 1 にした値を作りましたが、頭の切り替えが面倒という場合、

常に変更したい値を 1 （この場合は、0x1E）

で考え、

```
a |= 0x1E;        b4-1 を 1 にしたい場合
a &= ~0x1E;       b4-1 を 0 にしたい場合
```

~（ビット毎の反転）

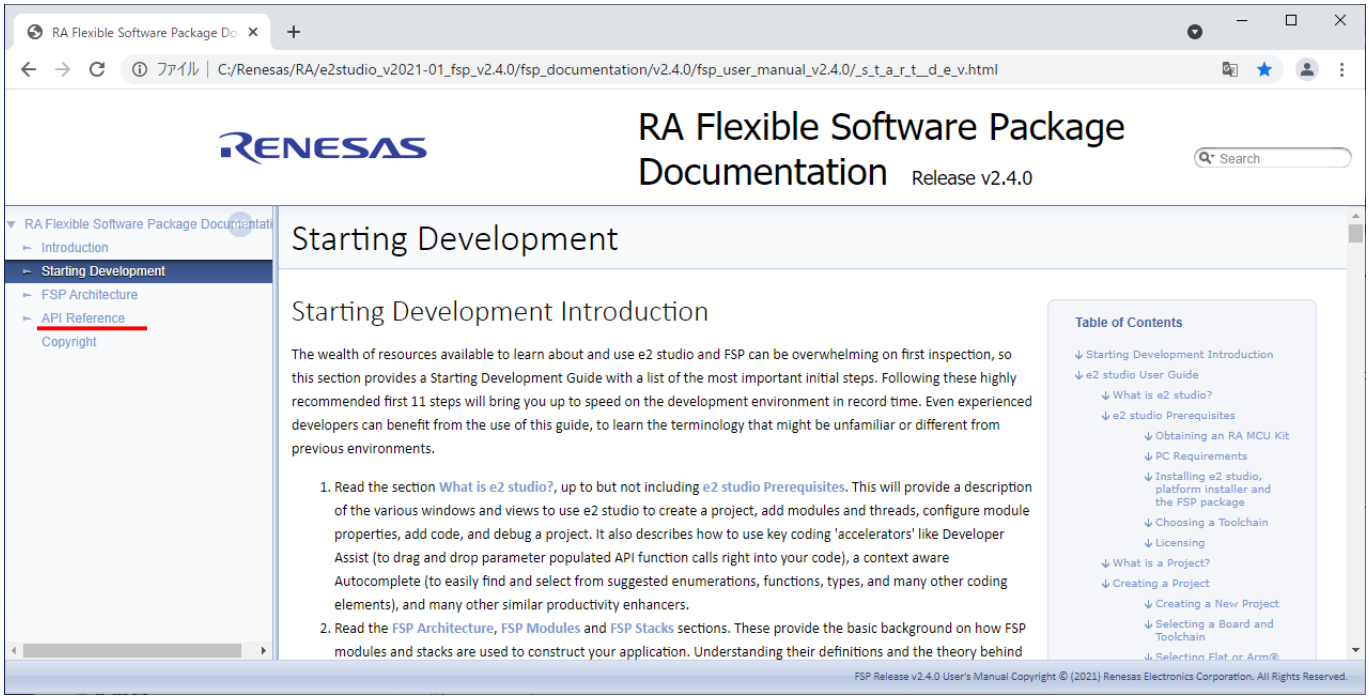
を使うという手もあります。

本チュートリアルでのサンプルプログラムは、大抵 "&= ~ " の形式で特定のビットを 0 に変更しています。

2. RA2L1_FSP_SW_LED

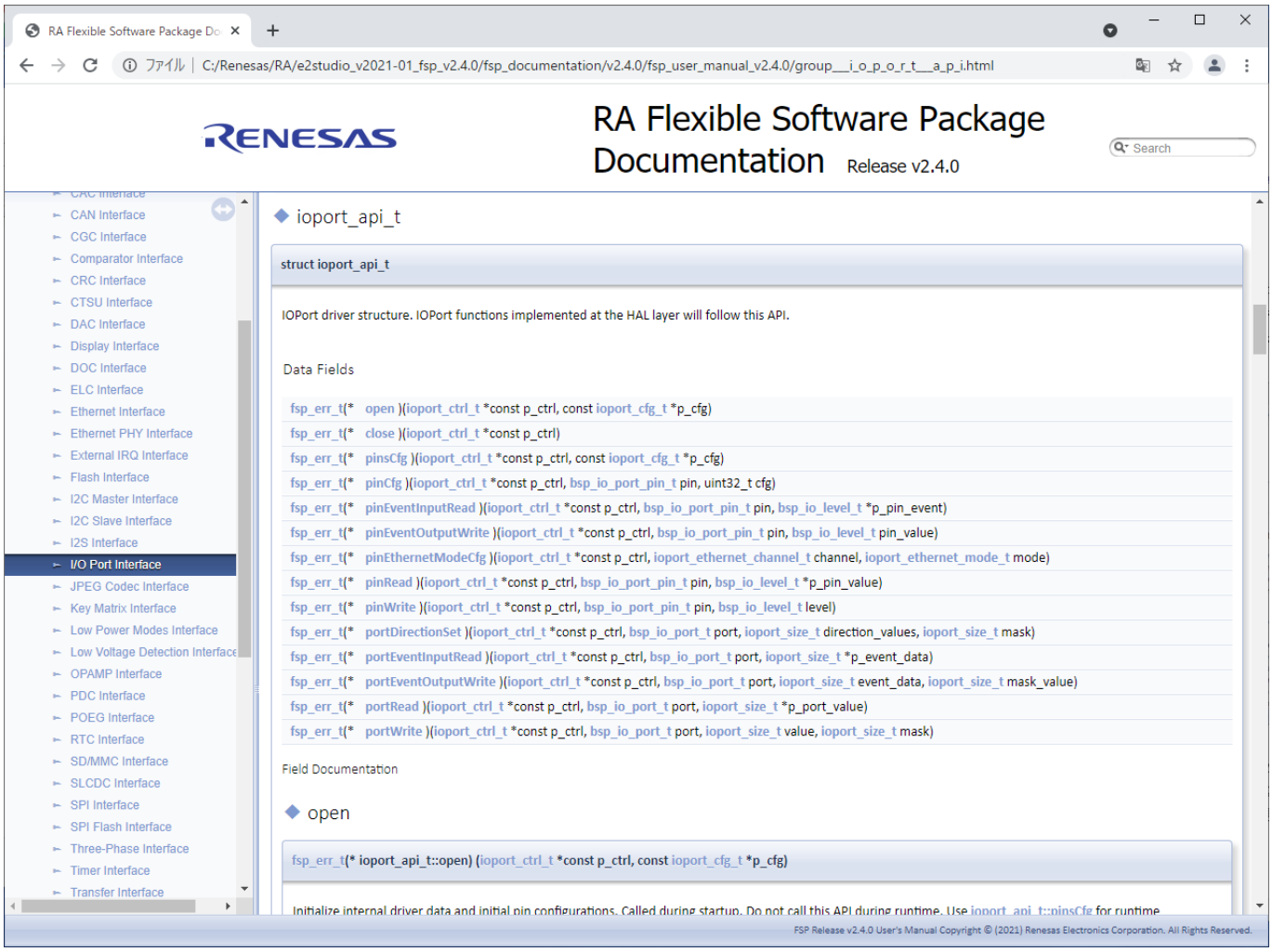
先のチュートリアルでは、PODR, PCNTR3, PIDR といったレジスタを操作して、I/O ポートの出力と入力を行いました。

レジスタを操作してマイコンを動かすというのは、組み込みプログラムの基本ですが、現代においては古いやり方になりつつあります。FSP では、マイコンの I/O ポートを動かす API が用意されています。このチュートリアルでは、API を使って前チュートリアルと同じ動作をさせようと思います。



FSP の API のリファレンスマニュアルは、e2studio をインストールしたフォルダに、HTML のオンラインマニュアルという形で、格納されています。FSP Ver2.4.0, 2021/5 の時点では英語版しかありません。今後、日本語版が選べるようになる事を期待しています。（最近は機械翻訳の精度もかなり向上しているので、日本語版で読みたいという方は、ネットでの機械翻訳を活用してください。）

I/O ポート関連の API としては、以下のようなものが用意されています。



本チュートリルでは、前チュートリアルで作成したプログラムを、API 関数を使って置き換えます。

2.1. FSP API 関数を使用したプログラム(LED)

LED を switch case 文を使って ON させる関数です。

(1)led_on_1_tmp()

```
void led_on_1_tmp(unsigned char no)
{

    //引数として与えたLEDの番号(0~7)のLEDを点灯させる関数（その1）

    //引数
    //  unsigned char no : 点灯させたいLEDの番号を指定(no=0-7)

    //戻り値
    //  なし

    switch(no)
    {
        case 0:
            (void)R_IOPORT_PinWrite(&g_ioport_ctrl, BSP_IO_PORT_00_PIN_00, BSP_IO_LEVEL_LOW);
            break;

        case 1:
            (void)R_IOPORT_PinWrite(&g_ioport_ctrl, BSP_IO_PORT_00_PIN_01, BSP_IO_LEVEL_LOW);
            break;

        case 2:
            (void)R_IOPORT_PinWrite(&g_ioport_ctrl, BSP_IO_PORT_00_PIN_02, BSP_IO_LEVEL_LOW);
            break;

        case 3:
            (void)R_IOPORT_PinWrite(&g_ioport_ctrl, BSP_IO_PORT_00_PIN_03, BSP_IO_LEVEL_LOW);
            break;

        case 4:
            (void)R_IOPORT_PinWrite(&g_ioport_ctrl, BSP_IO_PORT_00_PIN_04, BSP_IO_LEVEL_LOW);
            break;

        case 5:
            (void)R_IOPORT_PinWrite(&g_ioport_ctrl, BSP_IO_PORT_00_PIN_05, BSP_IO_LEVEL_LOW);
            break;

        case 6:
            (void)R_IOPORT_PinWrite(&g_ioport_ctrl, BSP_IO_PORT_00_PIN_06, BSP_IO_LEVEL_LOW);
            break;

        case 7:
            (void)R_IOPORT_PinWrite(&g_ioport_ctrl, BSP_IO_PORT_00_PIN_07, BSP_IO_LEVEL_LOW);
            break;

        default:
            break;                              //0~7以外の数値の場合は何もしない
    }
}
```

R_PORT0->PODR_b.PODR0 = LED_ON;

前チュートリアルの関数とそれ程差異はないかと思います。

I/O ポートを制御する関数は、R_IOPORT_PinWrite()関数です。

R_IOPORT_PinWrite(&g_ioport_ctrl, BSP_IO_PORT_00_PIN_00, BSP_IO_LEVEL_LOW);
&g_ioport_ctrl

第 1 引数(&g_ioport_ctrl)がどこから来ているのか？？となるかも知れません。
第 2 引数は、(BSP_IO_PORT_00_PIN_00)、Port0 の pin00(P000)を操作するというのが、なんとなく伝わるかと思います。
第 3 引数は、(BSP_IO_LEVEL_LOW)L レベルを出力すると判ると思います。

hal_entry.c 内で、ツールが出力するテンプレート内に、

```
    if (BSP_WARM_START_POST_C == event)
    {
        /* C runtime environment and system clocks are setup. */

        /* Configure pins. */
        R_IOPORT_Open (&g_ioport_ctrl, &g_bsp_pin_cfg);
    }
```

上記の行があります。FSP の API 関数は、基本的に Open した後に使うという流れとなります。I/O の API 関数を Open する部分は、テンプレートで出力されているので、hal_entry.c 内で、いきなり R_IOPORT_PinWrite()関数を使っても大丈夫です。(I/O ポート以外の FSP API 関数を使う場合は、最初に Open する必要があります)

R_IOPORT_PinWrite() の第 1 引数は、Open の時に指定している引数と同じものを指定する必要があります。(テンプレートで設定される部分で、&g_ioport_ctrl を使っているので、ユーザプログラムでも同じキーワード(本来は構造体)を指定します。

例えば、P007 を H 出力に設定するときは、
R_IOPORT_PinWrite(&g_ioport_ctrl, BSP_IO_PORT_00_PIN_07, BSP_IO_LEVEL_HIGH);
で良いです。

BSP_IO_PORT_00_PIN_07, BSP_IO_LEVEL_LOW 等は、FSP の定数定義です

R_IOPORT_PinWrite()は、端子ごとの出力設定ですが、ポートグループまとめて(P000~P007 を一括で)設定する関数も用意されています。指定された LED を ON する関数を、別な関数を使って表現してみます。

(2)led_on()

```
void led_on(unsigned char no)
{

    //引数として与えたLEDの番号(0~7)のLEDを点灯させる関数（その1）

    //引数
    //  unsigned char no : 点灯させたいLEDの番号を指定(no=0-7)

    //戻り値
    //  なし

    ioport_size_t   mask;

    if(no > 7) return;  //noが8以上の時は何もしない

    mask = (unsigned short)(0x1 << no);
    (void)R_IOPORT_PortWrite(&g_ioport_ctrl, BSP_IO_PORT_00, 0x0000, mask);

}
```

ポートグループで、まとめて操作するのは、R_IOPORT_PortWrite()関数となります。

R_IOPORT_PortWrite(&g_ioport_ctrl, BSP_IO_PORT_00, 0x0000, mask);

第 1 引数は同様。第 2 引数は、ピン番号ではなく、ポート番号です。Port0, P0 です。第 3 引数は設定値。ここでは、LED を ON させたいので、0 を指定しています。

一見意味が掴みにくいのが、第 4 引数の mask です。ここでは、設定したくない値を（I/O ポートの状態を変えたくない情報を）与えます。

		b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
設定値	0x0000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
mask	0x0008	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
I/O 設定														0			

例えば、設定値を 0x0000, mask を 0x0008 とした場合、mask が 1 のところ、P003 のみ L となります。

		b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
設定値	0x0052	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0
mask	0x00ff	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
I/O 設定									0	0	1	0	1	0	0	1	0

設定値と mask を上記の様にした場合は、P007~P000 が 0x52 に設定されます。mask のビットが立っているところのみ、I/O ポートの値が設定されます。

2.2. FSP API 関数を使用したプログラム（スイッチ）

次に、スイッチ（ポート）の入力を読み取る関数ですが、以下の様になります。

(1)sw_read()

```
unsigned char sw_read(void)
{
    //SWの状態を読み取る関数

    //引数
    //  なし

    //戻り値
    //  bit7 = SW5-D
    //  bit6 = SW5-C
    //  bit5 = SW5-B
    //  bit4 = SW5-A
    //  bit3 = SW4
    //  bit2 = SW3
    //  bit1 = SW2
    //  bit0 = SW1
    //  各スイッチの状態を8bitで返す

    ioport_size_t   p_port_value_port_4;
    ioport_size_t   p_port_value_port_6;

    (void)R_IOPORT_PortRead(&g_ioport_ctrl, BSP_IO_PORT_04, &p_port_value_port_4);
    (void)R_IOPORT_PortRead(&g_ioport_ctrl, BSP_IO_PORT_06, &p_port_value_port_6);

    return (unsigned char)( ((p_port_value_port_6 & 0x000f) << 4) |  ((p_port_value_port_4 & 0x0f00)
>> 8) );
}
```

R_IOPORT_PortRead(&g_ioport_ctrl, BSP_IO_PORT_04, &p_port_value_port_4);

上記で、Port4(P400~P415)の入力（PIDR レジスタに相当する値）を読み取れます。戻り値は、第 3 引数に入ります。第 3 引数はアドレス（&付き）で与える必要があります。

ioport_size_t は、uint16_t の別名で、16bit の符号なし変数となります（unsigned short = uint16_t です）。

ここでは、ポートまとめて読み取る関数を使っていますが、P408(SW1)だけを読み取る場合、

```
    bsp_io_level_t readLevel;

    R_IOPORT_PinRead(&g_ioport_ctrl, BSP_IO_PORT_04_PIN_08, &readLevel);
```

R_IOPORT_PortRead()の代わりに、R_IOPORT_PinRead を使えば良いです。

read_level は、BSP_IO_LEVEL_LOW または BSP_IO_LEVEL_HIGH となります。

RA2L1_LED_SW チュートリアルでは、直接レジスタ操作を行っていたものを、RA2L1_FSP_LED_SW チュートリアルでは API 関数に置き換えました。

どちらも動作は同じですが、どちらが判り易いでしょうか。I/O ポートを制御するだけであれば、レジスタを直接制御した方が正直判り易いかと思います。しかし、色々なマイコンの機能を使う場合、決まった順番で複数のレジスタを操作しなければならなくなってきます。その場合、FSP で用意されている API 関数を使った方が圧倒的に楽に、速くプログラムを組む事ができます。

以降のチュートリアルでは、API が用意されているものは、積極的に API を使い、手間を省こうと思います。API が用意されていないものや、レジスタを直接操作した方が速いケースでは、API を使わない処理も併用します。

[参考]関数の速度比較

RA2L1_LED_SW では、何種類かの関数を作成してみました。

(1)led_on_1_tmp

```
void led_on_1_tmp(unsigned char no)
{
    switch(no)
    {
        case 0:
            R_PORT0->PODR_b.PODR0 = LED_ON;
            break;

        case 1:
            R_PORT0->PODR_b.PODR1 = LED_ON;
            break;

        case 2:
            R_PORT0->PODR_b.PODR2 = LED_ON;
            break;

        case 3:
            R_PORT0->PODR_b.PODR3 = LED_ON;
            break;

        case 4:
            R_PORT0->PODR_b.PODR4 = LED_ON;
            break;

        case 5:
            R_PORT0->PODR_b.PODR5 = LED_ON;
            break;

        case 6:
            R_PORT0->PODR_b.PODR6 = LED_ON;
            break;

        case 7:
            R_PORT0->PODR_b.PODR7 = LED_ON;
            break;

        default:
            break;
    }
}
```

(2)led_on_2_tmp

```
void led_on_2_tmp(unsigned char no)
{
    if(no > 7) return;

    R_PORT0->PODR &= (unsigned short)~(0x1 << no);
}
```

(3)led_on_3_tmp

```
void led_on_3_tmp(unsigned char no)
{
    if(no > 7) return;

    R_PORT0->PCNTR3_b.PORR = (unsigned short)(0x1 << no);
}
```

コメントを削除すると、プログラムコードとしては、上記の様なボリュームの差があります。

それぞれの関数を 10,000 回実行して、実行時間を測定してみます。

・コンパイラの最適化 OFF(-O0)

関数	実行時間[ms]	プログラムコードサイズ[bytes]
(1)led_on_1_tmp	16.2	172
(2)led_on_2_tmp	17.1	44
(3)led_on_3_tmp	14.6	58

(3)が一番シンプルなコードだと思っていたので、(3)が一番速いのは予想通りなのですが、一見冗長に思える(1)が(2)よりも速いのは意外でした。プログラムのコードサイズは、(1)が大きいのは納得ですが、(2)(3)も意外にサイズを食っています。これは、コンパイラの最適化を OFF にした時の結果です。FSP の環境では、デフォルトでは最適化 ON(-O2)ですので、最適化を ON にして同じ測定を行ってみます。

・コンパイラの最適化 ON(-O2) [デフォルト]

関数	実行時間[ms]	プログラムコードサイズ[bytes]
(1)led_on_1_tmp	9.6~10.4	144
(2)led_on_2_tmp	7.9	24
(3)led_on_3_tmp	10.9	31

3 種類とも高速に動作する様になりましたが、今度は(3)が一番遅いという結果に。また、(2)が一番速く、速度的には先程と真逆の結果となりました。

なお、(1)の実行時間はばらつきがありますが、引数に 7 を指定した際に 9.6ms で、5 を指定した場合は 10.4ms になる等の差が出ます。コンパイルされた結果を見てみましたが、比較として 7,6,0,1,2,3,4,5 の順で行っていましたので、7 は速く 5 は遅いという差が出ます。

プログラムサイズに関しては、コードの行数とそれなりに相関があると考えて良いかと思いますが、実行速度はコードの書き方や、最適化によって大きく変わる（一見効率の良さそうなコードだから実行も速いという事はない）という事かと思います。

（極端に実行速度に拘るのであれば、アセンブラコードベースで書き下す必要があるものと思います。）

同じプログラムを、FSP 関数を使って書いた場合も、同様に比較してみます。

(1)led_on_1_tmp

```
void led_on_1_tmp(unsigned char no)
{
    switch(no)
    {
        case 0:
            (void)R_IOPORT_PinWrite(&g_ioport_ctrl, BSP_IO_PORT_00_PIN_00, BSP_IO_LEVEL_LOW);
            break;

        case 1:
            (void)R_IOPORT_PinWrite(&g_ioport_ctrl, BSP_IO_PORT_00_PIN_01, BSP_IO_LEVEL_LOW);
            break;

        case 2:
            (void)R_IOPORT_PinWrite(&g_ioport_ctrl, BSP_IO_PORT_00_PIN_02, BSP_IO_LEVEL_LOW);
            break;

        case 3:
            (void)R_IOPORT_PinWrite(&g_ioport_ctrl, BSP_IO_PORT_00_PIN_03, BSP_IO_LEVEL_LOW);
            break;

        case 4:
            (void)R_IOPORT_PinWrite(&g_ioport_ctrl, BSP_IO_PORT_00_PIN_04, BSP_IO_LEVEL_LOW);
            break;

        case 5:
            (void)R_IOPORT_PinWrite(&g_ioport_ctrl, BSP_IO_PORT_00_PIN_05, BSP_IO_LEVEL_LOW);
            break;

        case 6:
            (void)R_IOPORT_PinWrite(&g_ioport_ctrl, BSP_IO_PORT_00_PIN_06, BSP_IO_LEVEL_LOW);
            break;

        case 7:
            (void)R_IOPORT_PinWrite(&g_ioport_ctrl, BSP_IO_PORT_00_PIN_07, BSP_IO_LEVEL_LOW);
            break;

        default:
            break;
    }
```

(2)led_on

```
void led_on(unsigned char no)
{
    ioport_size_t   mask;

    if(no > 7) return;

    mask = (unsigned short)(0x1 << no);
    (void)R_IOPORT_PortWrite(&g_ioport_ctrl, BSP_IO_PORT_00, 0x0000, mask);

}
```

・コンパイラの最適化 OFF(-O0)

関数	実行時間[ms]	プログラムコードサイズ[bytes]
(1)led_on_1_tmp	42.0	204（ユーザコード） + FSP API 関数
(2)led_on	38.4	64（ユーザコード） + FSP API 関数

・コンパイラの最適化 ON(-O2) [デフォルト]

関数	実行時間[ms]	プログラムコードサイズ[bytes]
(1)led_on_1_tmp	17.0	148（ユーザコード） + FSP API 関数
(2)led_on	14.2	32（ユーザコード） + FSP API 関数

単純な I/O を制御するプログラムでは、FSP のオーバヘッドが見える形となりました。
プログラムコードサイズは、ユーザプログラム部分とは別に、FSP の API 関数が追加されます。

同様にスイッチの読み取り関数の実行速度も見てみました。

まずは、FSP を使用しない場合です。

・コンパイラの最適化 OFF(-O0)

関数	実行時間[ms]	プログラムコードサイズ[bytes]
(1)sw_read_1_tmp	51.8~44.2	266
(2)sw_read_2_tmp	43.4~53.4	248
(3)sw_read_3_tmp	17.1	56

(1)の 51.8~44.2ms は、スイッチが全部 OFF の時、51.8ms で、スイッチを全部 ON にした場合 44.2ms です。同様に(2)は、スイッチが全部 OFF の時、43.4ms で、スイッチを全部 ON にした場合 53.4ms です。これは、思惑通りで、す。(1)はスイッチ OFF 時の処理が増えます。(2)はスイッチ OFF 時の処理が軽くなるようなプログラムとなっています。スイッチの読み取りに関しては、順次処理よりポートまとめて処理した方がかなり速くなっています。

・コンパイラの最適化 ON(-O2) [デフォルト]

関数	実行時間[ms]	プログラムコードサイズ[bytes]
(1)sw_read_1_tmp	32.0	124
(2)sw_read_2_tmp	32.0	128
(3)sw_read_3_tmp	9.6	28

最適化を ON すると、実行時間がスイッチの ON の個数に影響されなくなります。（(2)の方が軽くなるだろうと考えてプログラムを書いても、最適化されれば、関係なくなるという事です。）最適化後も、(3)の方が速くプログラムサイズも小さいですので、スイッチの読み取りに関しては、順次処理よりポートまとめての方が有利と言い切れると思います。

次に、FSP を使った場合です。

・コンパイラの最適化 OFF(-O0)

関数	実行時間[ms]	プログラムコードサイズ[bytes]
(1)sw_read_1_tmp	221~214	346（ユーザコード）+ FSP API 関数
(2)sw_read_2_tmp	197~206	328（ユーザコード）+ FSP API 関数
(3)sw_read	41.8	82（ユーザコード）+ FSP API 関数

・コンパイラの最適化 ON(-O2) [デフォルト]

関数	実行時間[ms]	プログラムコードサイズ[bytes]
(1)sw_read_1_tmp	101	236（ユーザコード）+ FSP API 関数
(2)sw_read_2_tmp	98.4	228/（ユーザコード）+ FSP API 関数
(3)sw_read	24.6	60（ユーザコード）+ FSP API 関数

FSP を使った場合、実行時間やプログラムサイズの点でそれなりのオーバヘッドが見られます。でも、これはある意味当たり前で、I/O 制御の様な単純なプログラムではレジスタを直に制御する方が有利なのは当然です。

I/O 制御に関しては、初期設定だけ FSP を使い、L/H や入力値を見るのは、レジスタアクセスでも良いのかとも思います。この辺りはプログラムの開発方針にも関わる話ですので、プログラムを作成する方が判断されれば良いと思います。

取扱説明書改定記録

バージョン	発行日	ページ	改定内容
REV.1.0.0.0	2021.5.7	—	初版発行

お問合せ窓口

最新情報については弊社ホームページをご活用ください。

ご不明点は弊社サポート窓口までお問合せください。

株式会社 北斗電子

〒060-0042 札幌市中央区大通西 16 丁目 3 番地 7

TEL 011-640-8800　FAX 011-640-8801

e-mail：support@hokutodenshi.co.jp (サポート用)、order@hokutodenshi.co.jp (ご注文用)

URL:http://www.hokutodenshi.co.jp

ルネサス エレクトロニクス RA マイコン搭載
HSB シリーズマイコンボード 評価キット

SmartRA 学習キット
チュートリアル 2

株式会社北斗電子

 2021 年 5 月 7 日改訂 REV.1.0.0.0 (210507)