R8Cマイコンでビットフィールドは 使えるか

前回の宿題で、R8Cマイコン上で、ビットフィールドは使えるか。 また、構造体中のビット並びは、上から b7、b6、b5 になるのか、b0、b1、b2に なるのか確認する事になってました。

出来れば、構造体、共用体、ビットフィールド等の 基本の話を したい気もしますが、かなり長くなりそうな気がしますので、また別の機会に回します。 今回は I/Oポートの バッファとしての ビットフィールドに的を絞った話とします。

まず、最初に結果から報告します。
ビットフィールドの先頭行の bitは、バイトデータ内の 最下位ビットでした。 その方が 都合がいいかなとも思いました。 例で 示します。

I/O Exp Adr=0 Port A	b7	出力	RELAY_1	High Active
	b6	出力		
	b5	出力	LED_2	Low Active
	b4	出力	LED_1	Low Active
	b3	入力		
	b2	入力		
	b1	入力	H_LIM_SW	Low Active
	b0	入力	L_LIM_SW	Low Active

この前回のI/Oポート表を ビットフィールド構造体に 並べると、メンバーが 上下逆になります。

```
typedef struct {
    BYTE l_lim_sw: 1;    // b0 (最下位)
    BYTE h_lim_sw: 1;    // b1
    BYTE pad2:     2;    // b2-b3
    BYTE led_1:    1;    // b4
    BYTE led_2:    1;    // b5
    BYTE pad1:     1;    // b6
    BYTE relay_1:  1;    // b7 (最上位)
} EX_PORT_A;   // bit field データ型名
```

余談ですが
　何故、コンパイラによって ビットフィールドの並びが違う事があるのか。に関しては、かなり古い話ですが、MS-DOSの時代に Lattice-Cという　Cコンパイラで開発されたアプリが ありました。
　その後、MS-Cが 出てきてコンパイラの置き換えの話が、出て来ましたが、データファイルに互換性が無い事が分かり、その原因は構造体中のビットフィールドの並びが異なる事だったようです。 そのプロジェクトには、私は関与して無かったので詳細は分かりません。

　昔、そんな事があったという話です。
その他、CPUの違いにより、リトルエンディアン ビッグエンディアン、バイトマシン、16bitマシン、32bitマシンで、メモリアライメントの問題で、障害が 発生する場合が あります。

　R8Cの場合は、ビットフィールド宣言の 先頭行は、b0から順に並ぶ事が　分かったので、I/O ポート表も　上の行から b0、b1、b2の 順に 書いていく方が 良さそうですね。

　因みに どのように調べたかというと、プログラムで示します。 前ページの ビットフィールド構造体の変数宣言を 行います。 構造体が、1byteに納まっているかは、sizeof演算子を　用いて確認します。 1 byte である事を 確認しました。

```
UX_PORT_A  Uxpa;  // ビットフィールド宣言

    sio_prin( "Size = " );
    sio_prin_word_dec( sizeof(Uxpa), 1 );
    // bit fieldの byte サイズ
```

　ビットフィールド構造体型名が 前のページと異なる事に気付いた方もいるかもしれません。その理由を　次のページで説明します。

ビットフィールド構造体変数の 中身を 16進数で、テラタームで表示しようと思いましたが、byte変数を 16進数2文字でテラターム側で表示する関数に サイズ1byteの ビットフィールド構造体変数を byteでキャストして渡そうとしましたが、不正な CAST宣言という事でエラーになりました。 予測して いましたが、やはりという感じでした。 ビットフィールド構造体変数を byteデータとして 引数に渡すため union という 型宣言を 追加します。 この共用体宣言を 行った関係で データ型名が変わりました。

```
typedef union {      // 共用体宣言
     EX_PORT_A  xa;   // bit field変数名
     BYTE       byt; // byte 変数名
} UX_PORT_A;          // 共用体 型名

  UX_PORT_A  Uxpa; // bit field & byteの
                        変数名 宣言
```

共用体 union とは、どのような宣言なのかというと、

```
EX_PORT_A   xa;    // bit field変数名
BYTE        byt;   // byte 変数名
```

xa という変数と byt という変数は、全く同じアドレスに配置されます。これを別の表現をすると、一つの 1byteの データに xa と bytの 2つの 名前が付いているという事です。

ビットフィールドでアクセスする時は Uxpa.xa、関数の引数に byte データで 渡す時は Uxpa.byt と 言う事に なります。

```
Uxpa.xa.relay_1 = 1;
    // ビットフィールド relay_1 に 1を設定
sio_prin_byte_hex2( Uxpa.byt );
                        // byteで 引数を渡す
```

ちょっと面倒ですが、こうすれば、ビットフィールドのデータを バイトデータとして ポートへ出力出来ます。逆にポートから読出しも 可能です。

ソースを部分的に小出しにしたので、全体像が分かり難いところもあったと思います。 一連の流れを お見せします。

```
typedef struct {  // bit field 構造体 宣言
   BYTE  l_lim_sw: 1;   // b0 (最下位)
   BYTE  h_lim_sw: 1;   // b1
   BYTE  pad2:     2;   // b2-b3
   BYTE  led_1:    1;   // b4
   BYTE  led_2:    1;   // b5
   BYTE  pad1:     1;   // b6
   BYTE  relay_1:  1;   // b7 (最上位)
} EX_PORT_A;     // bit field データ型名

typedef union {     // 共用体宣言
    EX_PORT_A xa;   // bit field変数名
    BYTE      byt;  // byte 変数名
} UX_PORT_A;  // 共用体 型名

  UX_PORT_A  Uxpa; // bit field & byte変数
```

```
void  main( void )
{
   init_proc();       // 初期化処理
   sio_recv_wait();  //  1文字 受信待ち

   sio_prin( "Size = " );
   sio_prin_word_dec( sizeof( Uxpa ), 1 );
       // bit field & byte変数の byteサイズ
   sio_put_crlf();    // 改行

   Uxpa.xa.relay_1 = 1; // xaで bit fieldの
          //名前指定で  relay_1 に 1を 設定

   sio_prin( "RELAY_1 = 1 : Hex=" );
   sio_prin_byte_hex2( Uxpa.byt ); // bytで
        // バイト指定で 引数を渡す
   sio_put_crlf();    // 改行
```

で、上記プログラムを実行して テラタームにどのように表示されるかを お見せします。

上のテラタームの画面ですが、Size = 1 はビットフィールド宣言した変数が 1 byteのサイズに収まっている。 という事です。

仮に あと 1 bit ビットフィールドを追加すると 2 byte になります。

そして ビットフィールドの relay_1 にだけ 1を設定して、ビットフィールド変数の内容を16進数表示を行うと 80h という事で、b7だけが、1 に なっています。

RELAY_1は、最上位ビットの b7 なので、これで正解です。

前ページで、使用したビットフィールド構造体をバイトデータとして扱える Uxpa.byt を 使って I/Oポートをアクセスする コーディングサンプルを表示します。

R8Cマイコン内部 I/Oポートのアクセス

```
p1 = Uxpa.byt;
// ポート１に バイトデータ出力する

Uxpa.byt = p1; // p1ポートの状態を
               // Uxpa.bytに 代入する
```

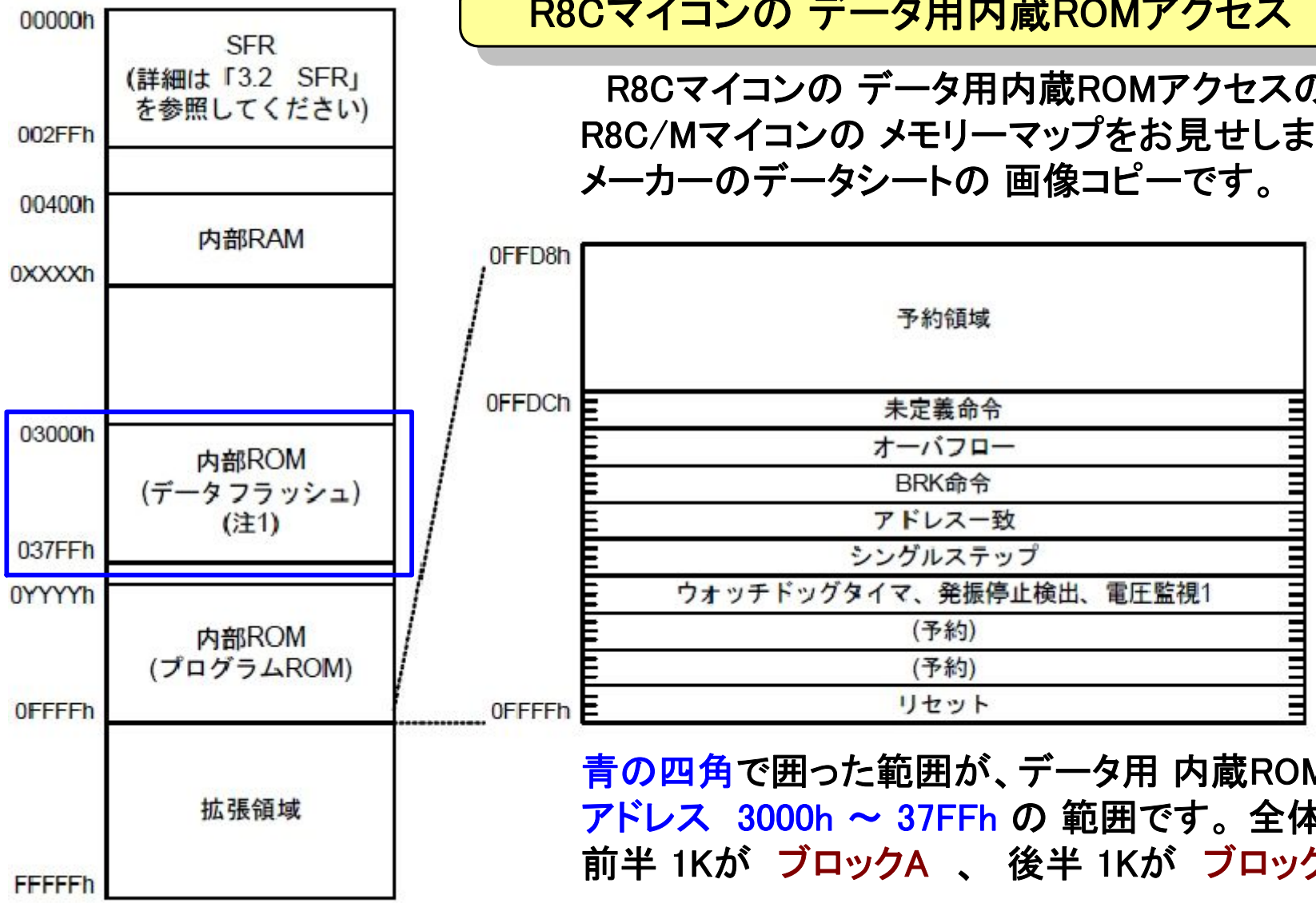
I2Cバスに接続した MCP23017のポートアクセス

```
ioexp_pa_out( adr, Uxpa.byt );
// MCP23017の ポートAに、Uxpa.bytの
// 内容を 出力する

Uxpa.byt = ioexp_pb_in( adr );
// MCP23017の ポートBの状態を、
// Uxpa.byt に 代入する
```

R8Cマイコンの データ用内蔵ROMアクセス

R8Cマイコンの データ用内蔵ROMアクセスの　話の前に
R8C/Mマイコンの メモリーマップをお見せします。
メーカーのデータシートの 画像コピーです。



青の四角で囲った範囲が、データ用 内蔵ROMの部分です。
アドレス　3000h ～ 37FFh の 範囲です。全体で 2kbyte ですが
前半 1Kが　ブロックA　、　後半 1Kが　ブロックBになってます。

データ用内蔵ROMの アドレスは 3000h ～ 37FFh の 範囲です。全体で 2kbyte ですが前半 1Kが ブロックA 、 後半 1Kが ブロックB になってます。

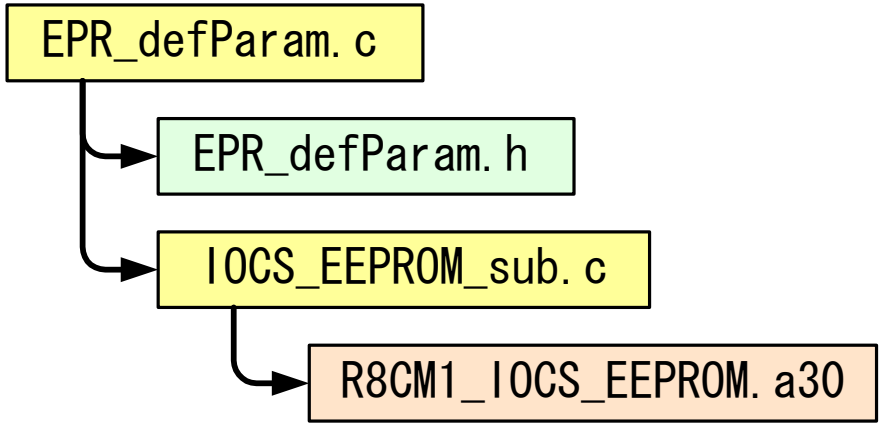
ブロックA ブロックBとは、データを 消去する時の ひとまとまりの単位の様です。
で、消去できる回数は、ブロック毎に 最大 10000回程度 との 事です。

データを書き込む際にも、独自のシーケンスが、あります。 EW1、サスペンドモードとか、メーカーのデータシートに書いてあります。

とはいえ、データシートの ディテールフローを元に プログラムを作成したので、細かい意味は いまいち理解していません。 それと この書き込みプログラムを作成してから、9年ほど経過している事もあり、だいぶ忘れてます。

使い始めると 多少 思い出すかも しれません。

R8Cマイコンの データ用内蔵ROMの 書き込みプログラムは、以下のモジュールで出来ています。



で、今回は、出来るだけ簡単に使えるように EPR_defParam.c を 用意しました。 より汎用的に細かく使う場合は、IOCS_EEPROM_sub.c を 直接アクセスして下さい。 EPR_defParam.c が サンプルプログラムになると 思います。
EPR_defParam.c 使い方は、次に説明します。

EPR_defParam.c の使い方

組み込みマイコンを使用した システムでは、運用時に、状況に応じて パラメータを 変更する場合があります。 で、組込みプログラム内に、固定的にパラメータを持ってしまうと、パラメータを 変更する必要が生じた時に、毎回プログラムを書き直す事になるので、不自由です。
現場で間違って別のプログラムを書き込む等の トラブルが起こる場合もあります。

よって、運用プログラムにより、パラメータを書き込んだ小さなファイルを 受信して、データ用ROMに、書き込む方法が、一般的と思います。

R8C/Mマイコンの場合は RAM容量が 小さい（1280byte）ので、パラメータ構造体の サイズは、大きくても 500byte 以下に して下さい。
データROMは、1kbyteのブロックが 2本ありますが、 片方のブロックで十分という事になります。 R8C/Mシリーズのマイコンは、足ピン数が少ないので、あまり複雑な制御は しないと思います。 という事で、RAM上に パラメータ構造体を宣言して、それを、データROMに 二次記憶的に、書き込んだり、読み出したりする事になります。 書き込み回数の制限は、約 10000回ですが、読み出し回数は、制限は有りませんので、CPUリセット後に、毎回 起動時の初期化処理で パラメータ構造体を データROMから RAMへ構造体変数を 転送する事になります。
よって、パラメータの管理は 全てのパラメータを 1本の構造体変数に、入れ込んでおく 方が、管理が楽です。

今回は、EPR_defParam.h 内に データROMに読み、書きする構造体変数の スケルトンを用意しました。 パラメータを入れ込む箱を用意した形に、なってますので、ROMに書き込んだり、読み出したりが、簡単に出来ます。

EPR_defParam.h の構造体 宣言

EPR_defParam.h の 内容　前半 2/3

```
//  ***  構造体データ宣言  ***  ( 共通部分 )
// ============================================
typedef struct  {
     WORD id;              // 識別子
     BYTE pgm;             // プログラム番号
     BYTE ver;             // バージョン番号
     WORD dt_siz;          // データブロックサイズ
     WORD pad;             // 予備
} EPR_HEADER;              // ROMパラメータ ヘッダー

//  ★  構造体データ宣言  ★( 個別データ部分 )
//  ( プロジェクトにより、変わる部分 )
// ============================================
typedef struct  {
     //  **  用途に応じて 必要なデータを宣言  **
     BYTE buf[128];        // 仮の データ
} EPR_DATA;          // ROMパラメータ データ宣言部
```

左の　構造体宣言で、EPR_HEADER は、データの前に付ける、ヘッダー情報です。

EPR_DATA は、パラメータ データを格納する構造体です。BYTE　buf[128]; は、ダミーで 入れているデータですので、実際の パラメータデータに 入れ替えて下さい。

あと、この2つの構造体を　取り込む全体の構造体が、1つありますが、次のページに示します。

EPR_defParam.h の 内容　後半 1/3

```
//  ***  構造体データ宣言  ***  （ 全体 ）
// ============================================
typedef struct  {
     EPR_HEADER hd;     // ROMパラメータ ヘッダー
     EPR_DATA   dt;     // ★ ROMパラメータ データ部
    WORD   sum;         // パラメータ チェック用サム値
}  EPR_PARAM;           // ROMパラメータ　全体
```

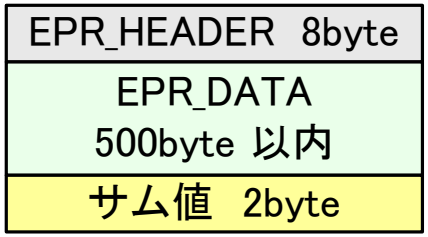
EPR_defParam.c の 内容　先頭 構造体変数 宣言

```
//  ROMパラメータ構造体 変数宣言
// ------------------------------------------
EPR_PARAM  Epr;                    // ROMパラメータ全体
```

左の EPR_defParam.h 内の 構造体宣言で、EPR_PARAM は、データROMに格納されるデータの 構造体宣言です。

その下の EPR_defParam.c 内 先頭の EPR_PARAM 型の変数 Epr を 実際使う事になります。

データROMに格納するデータ Epr



上記のような、データ構造になります。

R8CM12_IOCS.h 内の EPR_defParam.c 内関数の プロトタイプ宣言

```
//  ★★★  EPR_defParam.c  ★★★
void  param_make( void );        // パラメータデータ 仮生成
int   param_eep_load( void );    // パラメータを データROMから 呼び出し
int   param_eep_save( void );    // パラメータを データROMへ 書き込み
void  param_dump( void );        // パラメータ Eprの内容を テラタームに
                                 // ダンプ表示します。
static  WORD  pac_calc_sum( BYTE *buf, int cnt ); // 内部関数 サム値計算
```

EPR_defParam.c 内の 関数で、データROMをアクセスする関数は、param_eep_load 関数と、param_eep_save関数の 2つです。

param_eep_load 関数は、ROM内のデータを 読み出し 構造体変数 Eprに 転送する関数です。 この 関数は ROM上の絶対番地から RAM上の Epr変数に、2つのポインターを使って データのメモリ間コピーをしているだけです。

param_eep_save関数は、構造体変数 Epr の内容を、ROM内に 格納する関数です。 ROM上にデータを書き込む場合は、ややこしい特殊な 書き込みシーケンスが あります。 今回の作業の中で これが 一番厄介な 作業でした。

```
//  パラメータをフラッシュメモリから読み込む
int param_eep_load( void )
{
    BYTE    *ptr;

    ptr = (BYTE*)EEP_A_ADDRESS;  // ROM ブロックAの 先頭アドレス 0x3000
    mem_copy((BYTE*)&Epr, ptr, sizeof( Epr ));  // メモリ間転送処理
    if( Epr.sum == pac_calc_sum( (BYTE*)&Epr, sizeof( Epr ) - 2 ) )
        return  1;    // パラメータ正常読み出し（ サム値 一致 ）

    return  -1;      // 読み出しパラメータ異常（ サム値 不一致 ）
}
```

説明していませんでしたが、mem_copy関数の下に if文が あり Epr.sum と pac_calc_sum関数の関数値を 比較してますが、これは データが壊れていないかを調べる一つの方法で、サムチェックといいます。 サムチェックは、メモリ上の データあるいは、プログラムが 壊れていないかを 調べる手法です。

やり方は単純で、メモリ上のデータを、先頭から byte単位で 足し算を行い格納されているサム値の手前で 止めます。そして 格納されているサム値と合計値が、一致するか調べる手法です。

```
int  param_eep_save( void )
{
    BYTE    *ptr;
    int     sts;

    ptr = EEP_A_ADDRESS;  // ポインタに データROM ブロックAの アドレスを設定する
    Epr.sum = pac_calc_sum( (BYTE*)&Epr, sizeof( Epr ) - 2 ); // サム値計算

     sts = eep_block_erase( EEP_A_ADDRESS );         // EEPROM Block1 消去
     if( sts == 0 )  sio_print("* ROM Erase Error.");

     sts = eep_write_data( EEP_A_ADDRESS, (BYTE*)&Epr, sizeof( Epr )); // データ ROM書込み
     if( sts == 0 )  sio_print("* ROM Write Error.");

    return  1;
}
```

最初に、ptr に ROM先頭アドレスを設定し、サム値の計算をして、構造体最後の Epr.sum に書き込みます。 書き込み処理は、① データROM ブロック1の消去 ② データROMに データを書き込みの2つです。先頭に eep が付く 2つの関数は IOCS_EEPROM_sub.c 内に実態が あります。IOCS_EEPROM_sub.c の内容は メーカーのデータシートを見ても 難解です。

では、今回は EPR_defParam.c の 関数を
使って、パラメータデータを データROMに
書いたり 読んだりする実験を 行います。