【実践で使えるプログラミング編】

Appendix 共用体とビット操作

組み込み開発でのC言語記述におけるトレードオフ

山際 伸一

C言語をマスタしてくると，文法を駆使して，プログラムを短くすることにも気を配るようになるでしょう．また，ソフトウェア・エンジニアに組み込みプログラムを頼むと，非常にトリッキーなコードを提示されることもあります．そのようなC言語の文法の中で，メモリ領域のビット操作を行うための特殊な記述である共用体とビット・フィールドという文法があります．

● 共用体の特徴

複数のデータをまとめたデータ構造を構造体といいますが，その構造体の定義とともに用いられることが多いのが共用体です．共用体とは，データの見え方が，その参照方法によって異なるという定義の方法です．たとえば，以下のような共用体を用いて定義された構造体aの場合，

```
struct a {
    union{
            int x;
            int y;
            short z[2];
            short zz;
    };
};
```

a.xとa.yはまったく同じデータにアクセスしていることになります．つまり，参照する際のデータの名前が異なる，というものです．a.zに関しても同様なことがいえますが，zはshortの配列なので大きさが異なります．したがって，a.z[0]はa.x(またはa.y)の下位16ビット，a.z[1]は上位16ビットになるわけです．

それでは，a.zzはどうなるかというと，a.z[0]に一致します．つまり，共用体は，「もっとも大きいなデータ構造に大きさが合わせられ，もっとも大きなデータ構造より小さいメンバにアクセスする場合，先頭のバイトから該当する大きさのデータ部分が対象となる」という特徴をもちます．

● ビット・フィールドの記述方法

さらに，データを宣言する際には，ビット・フィールドというデータの一部のビット列にアクセスすることが可能な記述方法があります．たとえば，以下のような構造体の場合，

```
struct a {
    union{
            int x;
            struct BYTE {
                    int x_low:16;
                    int x_high:16;
            };
    };
};
```

a.x_lowはintで宣言されているにもかかわらず，この宣言は「:」のあとにビット数を指定することで，16ビットのデータであると宣言できます．

● 共用体とビット・フィールドで可読性の高いコードを書く

実は，共用体とビット・フィールドを用いると，SH7144の場合のようなメモリ・マップド・レジスタにアクセスするコードが非常にシンプルで可読性の高いものにできます．

SH7144のI/Oに関するレジスタを書いてみましょう(**リストA**)．_PEIORL構造体はLONGWORDとWORDとBYTEという三つの共用体メンバをもちます．さらにWORDとBYTEはPE15MD～PE0MDのそれぞれ2ビットのビット・フィールドをもちます．同様にして，_PEIORL構造体を**リストB**のように宣言できます．

これらの構造体へのポインタを以下のようにPECRL，PEIORLと

リストA　_PECRL構造体

```
struct _PECRL {
  union {
    unsigned int LONGWORD;
    struct {
      unsigned short PE15MD:2;
      unsigned short PE14MD:2;
      unsigned short PE13MD:2;
      unsigned short PE12MD:2;
      unsigned short PE11MD:2;
          ⋮ (中略)
      unsigned short PE1MD:2;
      unsigned short PE0MD:2;
    } WORD;
    struct {
      unsigned char PE15MD:2;
      unsigned char PE14MD:2;
      unsigned char PE13MD:2;
      unsigned char PE12MD:2;
      unsigned char PE11MD:2;
          ⋮ (中略)
      unsigned char PE1MD:2;
      unsigned char PE0MD:2;
    } BYTE;
  };
};
```

リストB　_PEIORL構造体

```
struct _PEIORL {
  union{
    unsigned short WORD;
    struct {
      unsigned char PE15IOR:1;
      unsigned char PE14IOR:1;
      unsigned char PE13IOR:1;
      unsigned char PE12IOR:1;
      unsigned char PE11IOR:1;
          ⋮ (中略)
      unsigned char PE1IOR:1;
      unsigned char PE0IOR:1;
    } BYTE;
  };
};
```

リストC　ビット・フィールドを用いない場合

```
111   PECRL.LONGWORD &= 0x3FFFFFFF;
0x10 <main+8>:    mov.w     0x7c <main+116>,r3 ! 0x0x83b8
0x12 <main+10>:   mov.w     0x7c <main+116>,r1 ! 0x0x83b8
0x14 <main+12>:   mov.l     @r1,r2
0x16 <main+14>:   mov.l     0x88 <main+128>,r1 ! 0x0x3fffffff
0x18 <main+16>:   and  r2,r1
0x1a <main+18>:   mov.l     r1,@r3
113   PEIORL.WORD |= (1<<15);
0x1c <main+20>:   mov.w     0x7e <main+118>,r3 ! 0x0x83b4
0x1e <main+22>:   mov.w     0x7e <main+118>,r1 ! 0x0x83b4
0x20 <main+24>:   mov.w     @r1,r2
0x22 <main+26>:   mov.w     0x80 <main+120>,r1 ! 0x0x8000
0x24 <main+28>:   or   r2,r1
0x26 <main+30>:   mov.w     r1,@r3
```

リストD　ビット・フィールドを用いた場合

```
117   PECRL.WORD.PE15MD = 0;
0x10 <main+8>:    mov.w     0x72 <main+106>,r3 ! 0x0x83b8
0x12 <main+10>:   mov.l     @r3,r2
0x14 <main+12>:   mov.l     0x7c <main+116>,r1 ! 0x0x3fffffff
0x16 <main+14>:   and  r2,r1
0x18 <main+16>:   mov.l     r1,@r3
118   PEIORL.BYTE.PE15IOR = 1;
0x1a <main+18>:   mov.w     0x74 <main+108>,r3 ! 0x0x83b4
0x1c <main+20>:   mov.w     @r3,r2
0x1e <main+22>:   mov.w     0x76 <main+110>,r1 ! 0x0x8000
0x20 <main+24>:   or   r2,r1
0x22 <main+26>:   mov.w     r1,@r3
```

して定義すると，

```
#define PECRL (*((struct _PECRL *)0xFFFF83B8))
#define PEIORL (*((struct _PEIORL *)0xFFFF83B4))
```

practice1，practice2とまったく同じレジスタ・アクセスの記述は，

```
// PE15を入出力ポートに設定
PECRL.LONGWORD &= 0x3FFFFFFF;
// PE15を出力に設定
PEIORL.WORD |= (1<<15);
```

となりますが，ビット・フィールドを用いることで，

```
PECRL.WORD.PE15MD = 0;
PEIORL.BYTE.PE15IOR = 1;
```

と記述することが可能です．ビット・マスクとか，ビット位置などを考えずにストレートで可読性の高いコードを書けることが理解できたと思います．

● 組み込み特有の注意事項

しかし!! 組み込みシステムではこの記述方法に注意を払う必要があります．なぜならば，ビット・フィールドへのアクセスにどのような命令が使われるかを気にしなければならないからです．

ビット・フィールドを用いないときは，GCCは**リストC**のように，LONGWORD，WORDの長さのmov命令を出力します．

その一方で，ビット・フィールドを用いた場合も，実はまったく同じmov命令を出力しています(**リストD**)．

前者は，|=演算を行っているため，読みと書きの演算であるとGCCが解釈し，1命令多く出力しています．しかし，後者では単純な書き込みアクセスであるとGCCが判断し，アドレスは一つのレジスタのみにロードされています．このように，命令を減らすことが可能である一方で，ロード命令の種類に対する危険が伴うのです．たとえば，SH7144のPECRLは，8，16，32ビットのどのアクセスでも許容します．しかし，SCIの送受信データ・レジスタの仕様を見ると，8ビットでのアクセスしか許されません．また，外部にハードウェアがあり，そのハードウェアの連続アドレスを共用体で宣言し，さらにビット・フィールドを用いた場合，コンパイラの共用体の解釈部分の作り込みによって，出力命令が何になるのかわからないのです．

上記のように，レジスタごとに定義している場合は，そのレジスタの長さに共用体の最大サイズをあわせておけるので安全ですが，レジスタ領域すべてを含んだ構造体を共用体で定義すると，この問題は確実に発生してしまいます．

ここでの例は，yamagiwa_code.tar.gzを展開した後に作成されるbit_assignディレクトリに例があるので，検証するとよい勉強になると思います．GCCの命令出力は，Insightでbin_assignを読み込み，ソース・ウィンドウの表示モードをMIXにすると非常によくわかります．

やまぎわ・しんいち