【組込みシステム編】第４回 ビット操作と浮動小数点数、その落とし穴

こんにちは、南角です。
まず前回の問題からです。
以下の２種類の関数を C 言語で作ってみてください。C 言語なので参照は使用不可とします。
一つ目はありふれていますが、2 つの整数を入れ替える関数 Myswap、プロトタイプは次の通りです。
void Myswap(適切な引数);
もう一つは 3 つの整数の整数値のうち、真ん中のものを戻り値として返す関数 Mymid です。
プロトタイプは次の通りです。
int Mymid(int, int, int);
どちらもビット演算を用いるということになっていましたよね。

答えの一例です。

```
int Mymax(int x, int y) {
return (x > y) ? x : y; }

int Mymid(int x, int y, int z) {
return MAX(x, y) ^ MAX(y, z) ^ MAX(z, x); }

void Myswap(int* xp, int* yp) {
*xp = *xp ^ *yp;
*yp = *yp ^ *xp;
*xp = *xp ^ *yp;

return; }
```

どうですか？どちらも効率はともかく面白くないですか？
解説は省略します。

例えば次のようなプログラムで確認してみてください。

```
int main()
{
int x = 333, y = 111, z = 200;
```

```
printf("mid = %d\n", Mymid(x, y, z));
printf("x = %d, y = %d, z = %d\n", x, y, z);
Myswap(&y, &z);
printf("x = %d, y = %d, z = %d\n", x, y, z);
printf("mid = %d\n", Mymid(x, y, z));
return 0; }
```

今回はビット関連でまとめてみましょう。

C 多少プログラムに慣れてきた人が犯しやすい間違いに、シフト演算があります。
例えば、
```
#define PORT_A *(volatile unsigned char* )0x1000
```
などと定義してあり、この内容を 1 ビット左シフトするような場合です。
ついつい PORT_A << 1; とかですませてしまう人が結構います。
間違いがわかりますか？

正しくは PORT_A <<= 1; （あるいは PORT_A = PORT_A << 1;）ですよね。
これが 1 の加算であれば PORT_A++; あるいは ++PORT_A;でも構いませんよね。
これがついつい PORT_A << 1; で済ませてしまう原因ではないかと、個人的には考えています。

もう一つのビット関連は浮動小数点数です。
浮動小数点ってビットの対極にあるのではないかと、思う人もいると思います。
しかし、そもそもコンピュータが扱えるのは 0、1 の 2 進数の整数だけですよね。
これで、どうやって浮動小数点数、実数を扱えるのでしょうか？

それは、浮動小数点数を指数形式で扱っているからです。
具体的には 2 進数のある実数は、次の形で表せます。
```
1.0110010100111 * 2 ** 1001100110110111
```
ここで仮数部と指数部に注目してみると、それぞれは整数ですよね。
そこで仮数部と指数部をそれぞれ適当なビット数として、コンピュータで扱いやすい 32 ビットやや 64 ビットにしたものが、それぞれ float や double となります。
（実際には float や double では、正規化によって仮数部を 1 ビット増やしているので、33 ビットと 65 ビットと言ってもいいのかもしれませんが…）
ここで考えなくてはならないのが、精度と符号です。
例えば float と long はどちらも 32 ビットです。
しかし float の方はその 32 ビットの中に仮数部と指数部を持っています。
それに対して long のほうはいわば仮数部だけを持てばいいわけです。

当然 long の方が精度は高いことになります。
```
long a = xxx, b;
float x;
```

として x = (float)a;
b = (long)x;
とすると a != b となる場合があるということです。
つまり浮動集点数とは、同じビット数の整数型よりも大きな範囲の数を、ある程度の精度で記憶できる数ということになります。
これが double であれば、仮数部も 32 ビット以上割り当てられているので
long a = xxx, b;
double x;

として x = (double)a;
b = (long)x;
とした場合、常に a == b
となります。

しかし、long long は 64 ビット整数なので float と long と同じことが起こりえます。
つまり何が言いたいかというと、現状は整数は long で浮動小数点数は double で扱っているシステムにおいて、将来精度を上げるために、整数の桁数を上げ、整数のみ long long とした場合に、精度関係の何らかの新たな不具合が発生する可能性があるということです。
これは、関連しそうな人は心の片隅にでも置いておいてください。

もう一つ、浮動所数点数間の差異の問題があります。
浮動小数点数の形式も IEEE で決められていますが、float や double 以外にも拡張 double というものがあります。
そして x86 系の CPU（の FPU）は double の計算も内部的には 10 バイトの拡張 double で演算しています。
ここでも少し問題になったことがあります。
VxWorks を使用していた工業用ロボットがありましたが、それが使用していた CPU は double の演算を 8 バイトで行っていました。
そしてそのロボットのシミュレーターを PC 上でも動かしていましたが、動きが異なる場合がありました。
原因は両者の double の精度の違いです。PC は double の計算を内部的には 10 バイトで行っていました。その結果、両方の動き（この場合は本当に動き、ロボットの動作）が異なる場合がありました。
この場合は、しょうがないので、最終確認は実機で行うことにしました。

浮動小数点数に関しては、まだ round や 符号、 メモリ上の形式などの話などが残っていますので、次回も続けたいと思います。
では、また次回。

組込みシステム編　第 4 回　おわり