eFlagBit

Values

Enum Name	Integer Value	Description
eFlagBit_0	1	
eFlagBit_1	2	
eFlagBit_2	4	
eFlagBit_3	8	
eFlagBit_4	16	
eFlagBit_5	32	
eFlagBit_6	64	
eFlagBit_7	128	
eFlagBit_8	256	
eFlagBit_9	512	
eFlagBit_10	1024	
eFlagBit_11	2048	
eFlagBit_12	4096	
eFlagBit_13	8192	
eFlagBit_14	16384	
eFlagBit_15	32768	
eFlagBit_16	65536	
eFlagBit_17	131072	
eFlagBit_18	262144	
eFlagBit_19	524288	
eFlagBit_20	1048576	
eFlagBit_21	2097152	
eFlagBit_22	4194304	
eFlagBit_23	8388608	
eFlagBit_24	16777216	
eFlagBit_25	33554432	
eFlagBit_26	67108864	
eFlagBit_27	134217728	
eFlagBit_28	268435456	
eFlagBit_29	536870912	
eFlagBit_30	1073741824	
eFlagBit_31	-2147483648	
eFlagBit_None	0	
eFlagBit_All	-1	

Remarks

A flag bit is a specialized form of storing a number of boolean flags within a single integer value. This is a form of data storage that is known as a Bit Field. How it works is that, rather than store values within separate boolean fields, you instead leverage bit-shift operations to use the individual bits of an integer.

First, notice that each of the above values represent a certain power of two, going from 2 (1) to 2^3 (1073741824). The value for 2^{31} is also included, but due to the nature of signed integers, its value is -2147483648. Each of these flags represents a single 1 in the integer's binary value. For example, the binary value of 2^5, or 32, in an integer would be 00000000000000000000000000100000 (a 1 in the sixth position).

Because of this, if you take different flags and add them together, they will result in a value that represents a 1 in each of those flag's position. For example, adding together eFlagBit_2, eFlagBit_7, eFlagBit_12, and eFlagBit_22 will result an integer that has 1's in the third, eighth, thirteenth, and twenty-third position. (00000000010000000001000010000100, or 4,198,532.)

In code, the way to combine different flags into a single value is pretty simple. All you do is use the bitwise-OR operator to combine each desired flag into a single value.

```
int lFlags = eFlagBit_2 | eFlagBit_7 | eFlagBit_12 | eFlagBit_22;
```

To check an integer for the presence of a flag, conversely you would use a bitwise-AND operator. Because of how the bitwise-AND works, if you perform it on an integer and a flag, the result will be the value of the flag if the integer holds that flag, or it will be zero if not.

```
bool bContainsFlag = (lFlags & eFlagBit_12) != ;
```

There are also two utility flags called eFlagBit_None and eFlagBit_All, equal to 0 and -1 respectively. The way these flags work is that the None flag is set to the integer equivalent of all binary digits being 0, and the All flag has all binary digits set to 1 (again, this is due to the nature of signed integers).

Because bitwise operations are so fast and efficient, using a bit field in this way will result in much more efficient code than if you used standard booleans. The other upside is that a boolean variable only needs one bit to store its value (0 for false, 1 for true), but booleans still take up a full byte. Using a bit field, you can store up to 32 different flags in a single integer, meaning you are only using 4 bytes instead of 32.