

Bitwise Flags and Conditionals

Bitwise and: &

Bitwise exclusive or: ^

Bitwise inclusive or: |

Bitwise complement: ~

Left shift: <<

Right shift: >>

Table 1: Bitwise operators acting on 1-bit fields—the truth table

a	b	a & b	a ^ b	a b
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Table 2: Bitwise operators acting on 1-byte (8-bit) fields

a	00110101	53
b	00101111	47
a & b	00100101	37
a ^ b	00011010	26
a b	00111111	63
~(a b)	11000000	192
(~a & ~b)	11000000	192

Left Shift—Right Shift

“The shift operators << and >> perform left and right shifts of their left operand by the number of bit positions given by the right operand, which must be positive. Thus $x \ll 2$ shifts the value of x left by two positions, filling vacated bits with zero; this is equivalent to multiplication by 2^2 . Right shifting an unsigned quantity always fills the vacated bits with zero. Right shifting a signed quantity will fill the sign bits (“arithmetic shift”) on some machines and with 0-bits (“logical shift”) on others.

“The function `getbits(x, p, n)` returns the (right-adjusted) n -bit field of x that begins at position p . Bit position 0 is at the right end and n and p are sensible, positive values. For example, `getbits(x, 4, 3)` returns the three bits in bit positions 4, 3, and 2, right-adjusted.

```
/* getbits: get n bits from poosition p */
unsigned getbits(unsigned x, int p, int n)
{
    return(x >> (p+1-n)) & ~(~0 << n);
}
```

“The expression $x \gg (p+1-n)$ moves the desired field to the right end of the word. ~ 0 is all 1-bits; shifting it left n bit positions with $\sim 0 \ll n$ places zeros in the rightmost n bits; complementing that with \sim makes a mask with ones in the rightmost n bits.” (Kernighan and Ritchie, 1988)

Conditional Expressions

The statements:

```
if (a > b)
    z = a;
else
    z = b;
```

can alternately be written:

```
z = (a > b) ? a : b;
```

both assign the maximum of a or b to z .

Example—seNumberOfSelectedPoints()

The function `seNumberOfSelectedPoints` uses a bitwise flag called "which". This flag does the job of three boolean flags. Notice how the values of `FPOINT`, etc. are carefully chosen to be powers of 2.

```
#define FPOINT 1
#define FNODE 2
#define FVERTEX 4

numpoints = seNumberOfSelectedPoints(FPOINT | FNODE | FVERTEX);

/* -----
PURPOSE: count the number of selected points, nodes,
        and/or vertices
PRE:     "which" is a bitwise flag that can be FPOINT,
        FNODE or FVERTEX or any combination of the
        three.
POST:
RETURN:  number of selected points, nodes, and/or
        vertices
AUTHOR:  Michael Kennard
----- */
Mint seNumberOfSelectedPoints (Mint which)
{
    fpointtype currpoint;
    Mint numsel = 0;

    if (which & FPOINT) {
        currpoint = g_activecoverage->pointlist.list;
        while (currpoint) {
            if (currpoint->selected) {
                numsel++;
            }
            currpoint = currpoint->next;
        }
    }
    if (which & FNODE) {
        currpoint = g_activecoverage->odelist.list;
        while (currpoint) {
            if (currpoint->node && currpoint->selected) {
                numsel++;
            }
            currpoint = currpoint->next;
        }
    }
}
```

```

if (which & FVERTEX) {
    currpoint = g_activecoverage->nodelist.list;
    while (currpoint) {
        if (currpoint->selected && !currpoint->node) {
            numsel++;
        }
        currpoint = currpoint->next;
    }
}
return (numsel);
} /* seNumberOfSelectedPoints */

```

Example—Setting booleans based on bitwise flags

Sometimes it is easier to create separate boolean flags in the function and work with the booleans,

```

#define bwDISPLAYLINE    1
#define bwDISPLAYTICKS  2
#define bwDISPLAYNUMBERS 4

boolean topaxis, topticks, topnumbers;

topaxis = (thePlot->topaxis & bwDISPLAYLINE) ? 1 : 0;
topticks = (thePlot->topaxis & bwDISPLAYTICKS) ? 1 : 0;
topnumbers = (thePlot->topaxis & bwDISPLAYNUMBERS) ? 1 : 0;

```

Example—Setting and unsetting one bitwise flag

```

char x;

x |= bwFLAG; /* "adds" bwFLAG to x, leaves other flags unchanged */

Same as:
if (!(x & bwFLAG))
    x += bwFLAG;

x &= ~bwFLAG; /* "removes" only bwFLAG from x, leaves other flags
              unchanged */

Same as:
x &= 255-bwFLAG;
-OR-
if (x & bwFLAG)
    x -= bwFLAG;

x ^= bwFLAG; /* "toggles" only bwFLAG in x, leaves other flags unchanged */

```

Examples—GMS flineatt.h

The macro PointAttFlags takes a pointer to a structure, and returns an integer bitwise flag. The flag's bits are set such that they show which pointers in the structure are NULL and which aren't.

```
/* bitwise flags */
#define bwGENERICATT          0
#define bwSPECHEADATT        1
#define bwSPECCONCATT        2
#define bwGENHEADATT         4
#define bwDRAINATT           8
#define bwRIVERATT           16
#define bwSTREAMATT          32
#define bwBARRIERATT        64
#define bwWELLATT            128
#define bwREFINEATT          256
#define bwOBSERVATIONATT     512
#define bwFLUXATT            1024
#define bwSPECFLUXATT        2048

#define PointAttFlags(att) ((att)->spehead ? bwSPECHEADATT : 0x0 ) | \
                           ((att)->speconc ? bwSPECCONCATT : 0x0 ) | \
                           ((att)->genhead ? bwGENHEADATT : 0x0 ) | \
                           ((att)->drain ? bwDRAINATT : 0x0 ) | \
                           ((att)->river ? bwRIVERATT : 0x0 ) | \
                           ((att)->stream ? bwSTREAMATT : 0x0 ) | \
                           ((att)->barrier ? bwBARRIERATT : 0x0 ) | \
                           ((att)->well ? bwWELLATT : 0x0 ) | \
                           ((att)->refine ? bwREFINEATT : 0x0 ) | \
                           ((att)->observe ? bwOBSERVATIONATT : 0x0 ) | \
                           ((att)->flux ? bwFLUXATT : 0x0 ) | \
                           ((att)->specflux ? bwSPECFLUXATT : 0x0 )
```

Example—GMS flineatt.c

The function feDeletePolyAttributes receives a pointer of an attribute structure, and deletes attributes in the structure according to the bitwise flag that is passed in. See if you can figure out which attributes are going to be deleted.

```
feDeletePolyAttributes(&(fpoly->atts), bwPROPERTYATT | bwRECHARGEATT | \
                      bwETATT);

feDeletePolyAttributes(&(fpoly->atts), bwALLATTS & (~bwPROPERTYATT));

feDeletePolyAttributes(&(fpoly->atts), bwALLATTS & ~(bwRECHARGEATT | \
                      bwETATT));
```

Example—WMS dem.h and dem.c

Bitwise flags are handy for conserving memory and time. You can use 1 bitwise flag in a large array instead of several booleans. In addition, you can save computation time and program more efficiently by just checking the bitwise flag instead of having several #define's for a single flag. Bitwise boolean operations are faster than relational or logical operators (>, >=, <, <=, ==, !=, &&, and ||).

```
#define DATA_MASK          1
#define WATERSHED_CELL     2
#define STREAM_CELL        4
#define VISITED_CELL       8
#define TERMINUS_CELL      16
#define ACTIVE_CELL        32
#define SELECTED_CELL      64

typedef struct dempoint {
    float elevation;
    char flags;
} dempoint;
```

The single char can potentially replace 8 boolean variables (1 for each bit). If each boolean is #defined as a Mint (4 bytes), you will save 27 bytes for each “dempoint” structure allocation. Using bitwise operators, each structure takes up 5 bytes. Without using bitwise operators, each structure takes up 32 bytes. Say you have a 2000X3000 DEM. Using bitwise operators, this will take up 2000X3000X5 bytes = 30,000,000 bytes. Without using bitwise operators, 2000X3000X32 bytes = 192,000,000 bytes of valuable memory (it's impractical to handle this much memory on most machines).

Defining DEM drainage basins without getting stuck in an infinite loop...

```
/* loop through all the DEM Cells and set all the
   cells to not visited. */
for (i=0; i<g_demcols; i++) {
    for (j=g_demrows-1; j>=0; j--) {
        g_dem[i][j].flags &= 255-VISITED_CELL;
    }
}
```

For each DEM Cell with data defined that has not already been assigned a basin ID...

Follow the flow directions until a visited cell, a cell with a basin ID, a NODATA cell, or a cell on the edge of the DEM is found:

```
if (!done && !(g_dem[curi][curj].flags & DATA_MASK))
    done = MTRUE;
if (!done && (g_demattri[curi][curj].basinid != NONE)) {
    curbasinid = g_demattri[i][j].basinid =
        g_demattri[curi][curj].basinid;
    done = MTRUE;
}
if (!done && (g_dem[curi][curj].flags & VISITED_CELL))
    done = MTRUE;
if (!done) {
    g_dem[curi][curj].flags |= VISITED_CELL;
}
```

When a basin or edge is found, backtrack using the flow directions, assigning basins and marking unvisited until an unvisited cell is found:

```
g_demattri[curi][curj].basinid = curbasinid;  
g_dem[curi][curj].flags &= 255 - VISITED_CELL;
```