WIKIPEDIA

Mask (computing)

In computer science, a **mask** is data that is used for bitwise operations, particularly in a bit field. Using a mask, multiple bits in a byte, nibble, word etc .. can be set either on, off or inverted from on to off (or vice versa) in a single bitwise operation.

Contents

Common bitmask functions

Masking bits to 1

To turn certain bits on, the bitwise OR operation can be used, following the principle that `Y OR 1 = 1` and `Y OR 0 = Y`. Therefore, to make sure a bit is on, OR can be used with a `1`. To leave a bit unchanged, OR is used with a `0`.

Example: Masking *on* the higher nibble (bits 4, 5, 6, 7) the lower nibble (bits 0, 1, 2, 3) unchanged.

```
   10010101   10100101
OR 11110000   11110000
 = 11110101   11110101
```

Masking bits to 0

More often, in practice, bits that are to be ignored are "masked *off*" (or masked to `0`) rather than "masked *on*" (or masked to `1`). There is no way to change a bit from *on* to *off* using the `OR` operation. Instead, bitwise `AND` is used. When a value is `AND`ed with a `1`, the result is simply the original value, as in: `Y AND 1 = Y`. However, `AND`ing a value with `0` is guaranteed to return a `0`, so it is possible to turn a bit off by using `AND` with `0`: `Y AND 0 = 0`. To leave the other bits alone, use `AND` with `1`.

Example: Masking *off* the higher nibble (bits 4, 5, 6, 7) the lower nibble (bits 0, 1, 2, 3) unchanged.

```
   10010101   10100101
AND 00001111   00001111
  = 00000101   00000101
```

Querying the status of a bit

It is possible to use bitmasks to easily check the state of individual bits regardless of the other bits. To do this, turning off all the other bits using the bitwise `AND` is done as discussed above and the value is compared with `1`. If it is equal to `0`, then the bit was off, but if the value is any other value, then the bit was on. What makes this convenient is that it is not necessary to figure out what the value actually is, just that it is not `0`.

Example: Querying the status of the 4th bit

```
   10011101   10010101
AND 00001000   00001000
  = 00001000   00000000
```

Toggling bit values

So far the article has covered how to turn bits on and turn bits off, but not both at once. Sometimes it does not really matter what the value is, but it must be made the opposite of what it currently is. This can be achieved using the `XOR` (exclusive or) operation. `XOR` returns `1` if and only if an odd number of bits are `1`. Therefore, if two corresponding bits are `1`, the result will be a `0`, but if only one of them is `1`, the result will be `1`. Therefore inversion of the values of bits is done by `XOR`ing them with a `1`. If the original bit was `1`, it returns `1 XOR 1 = 0`. If the original bit was `0` it returns `0 XOR 1 = 1`. Also note that `XOR` masking is bit-safe, meaning that it will not affect unmasked bits because `Y XOR 0 = Y`, just like an `OR`.

Example: Toggling bit values

```
   10011101   10010101
XOR 00001111   11111111
  = 10010010   01101010
```

To write arbitrary 1s and 0s to a subset of bits, first write 0s to that subset, then set the high bits:

```
register = (register & ~bitmask) | value;
```

Uses of bitmasks

Arguments to functions

In programming languages such as C, bit fields are a useful way to pass a set of named boolean arguments to a function. For example, in the graphics API OpenGL, there is a command, `glClear()` which clears the screen or other buffers. It can clear up to four buffers (the color, depth, accumulation, and stencil buffers), so the API authors could have had it take four arguments. But then a call to it would look like

```
glClear(1,1,0,0); // This is not how glClear actually works and would make for unstable code.
```

which is not very descriptive. Instead there are four defined field bits, `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT`, `GL_ACCUM_BUFFER_BIT`, and `GL_STENCIL_BUFFER_BIT` and `glClear()` is declared as

```
void glClear(GLbitfield bits);
```

Then a call to the function looks like this

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Internally, a function taking a bitfield like this can use binary `and` to extract the individual bits. For example, an implementation of `glClear()` might look like:

```
void glClear(GLbitfield bits) {
  if ((bits & GL_COLOR_BUFFER_BIT) != 0) {
    // Clear color buffer.
  }
  if ((bits & GL_DEPTH_BUFFER_BIT) != 0) {
    // Clear depth buffer.
  }
  if ((bits & GL_ACCUM_BUFFER_BIT) != 0) {
    // Clear accumulation buffer.
  }
  if ((bits & GL_STENCIL_BUFFER_BIT) != 0) {
    // Clear stencil buffer.
  }
}
```

The advantage to this approach is that function argument overhead is decreased. Since the minimum datum size is one byte, separating the options into separate arguments would be wasting seven bits per argument and would occupy more stack space. Instead, functions typically accept one or more 32-bit integers, with up to 32 option bits in each. While elegant, in the simplest implementation this solution is not type-safe. A `GLbitfield` is simply defined to be an

`unsigned int`, so the compiler would allow a meaningless call to `glClear(42)` or even `glClear(GL_POINTS)`. In C++ an alternative would be to create a class to encapsulate the set of arguments that glClear could accept and could be cleanly encapsulated in a library (see the external links for an example).

Inverse Masks

Masks are used with IP addresses in IP ACLs (Access Control Lists) to specify what should be permitted and denied. To configure IP addresses on interfaces, masks start with 255 and have the large values on the left side: for example, IP address 209.165.202.129 with a 255.255.255.224 mask. Masks for IP ACLs are the reverse: for example, mask 0.0.0.255. This is sometimes called an inverse mask or a wildcard mask. When the value of the mask is broken down into binary (0s and 1s), the results determine which address bits are to be considered in processing the traffic. A 0 indicates that the address bits must be considered (exact match); a 1 in the mask is a "don't care". This table further explains the concept.

Mask Example:

network address (traffic that is to be processed) 10.1.1.0

mask 0.0.0.255

network address (binary) 00001010.00000001.00000001.00000000

mask (binary) 00000000.00000000.00000000.11111111

Based on the binary mask, it can be seen that the first three sets (octets) must match the given binary network address exactly (00001010.00000001.00000001). The last set of numbers is made of "don't cares" (.11111111). Therefore, all traffic that begins with 10.1.1. matches since the last octet is "don't care". Therefore, with this mask, network addresses 10.1.1.1 through 10.1.1.255 (10.1.1.x) are processed.

Subtract the normal mask from 255.255.255.255 in order to determine the ACL inverse mask. In this example, the inverse mask is determined for network address 172.16.1.0 with a normal mask of 255.255.255.0.

255.255.255.255 - 255.255.255.0 (normal mask) = 0.0.0.255 (inverse mask)

ACL equivalents

The source/source-wildcard of 0.0.0.0/255.255.255.255 means "any".

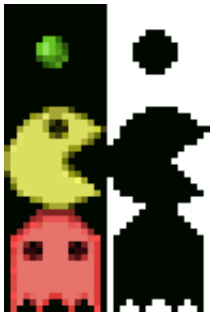
The source/wildcard of 10.1.1.2/0.0.0.0 is the same as "host 10.1.1.2"

Image masks

In computer graphics, when a given image is intended to be placed over a background, the transparent areas can be specified through a binary mask. This way, for each intended image there are actually two bitmaps: the actual image, in which the unused areas are given a pixel value with all bits set to 0s, and an additional *mask*, in which the correspondent

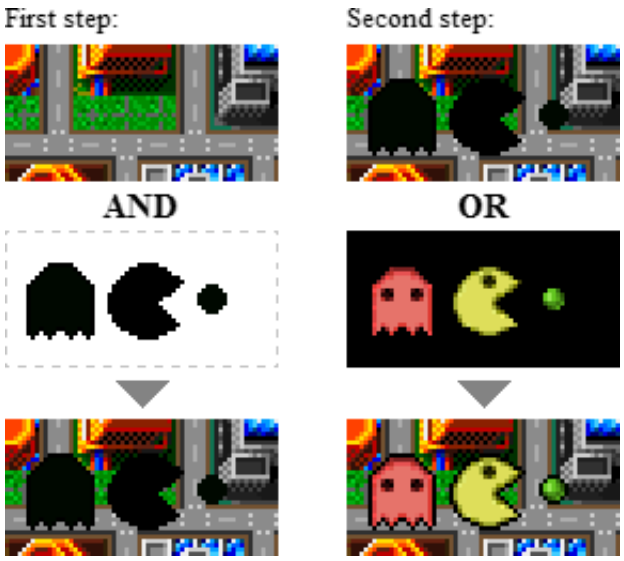
image areas are given a pixel value of all bits set to 0s and the surrounding areas a value of all bits set to 1s. In the sample at right, black pixels have the all-zero bits and white pixels have the all-one bits.

At run time, to put the image on the screen over the background, the program first masks the screen pixel's bits with the image mask at the desired coordinates using the bitwise AND operation. This preserves the background pixels of the transparent areas while resets with zeros the bits of the pixels which will be obscured by the overlapped image.



Raster graphic sprites (left) and masks (right)

Then, the program renders the image pixel's bits by combining them with the background pixel's bits using the bitwise OR operation. This way, the image pixels are appropriately placed while keeping the background surrounding pixels preserved. The result is a perfect compound of the image over the background.



This technique is used for painting pointing device cursors, in typical 2-D videogames for characters, bullets and so on (the sprites), for GUI icons, and for video titling and other image mixing applications.

Although related (due to being used for the same purposes), transparent colors and alpha channels are techniques which do not involve the image pixel mixage by binary masking.

Hash tables

To create a hashing function for a hash table, often a function is used that has a large domain. To create an index from the output of the function, a modulo can be taken to reduce the size of the domain to match the size of the array; however, it is often faster on many processors to restrict the size of the hash table to powers of two sizes and use a bitmask instead.

See also

- Affinity mask
- Bit field
- Bit manipulation

- Bitwise operation
- Subnetwork
- Tagged pointer
- umask

External links

- bit_enum: a type-safe C++ library for bitwise operations (https://bitbucket.org/fudepan/mili/wiki/BitwiseEnums)
