Bitmasks for nicer APIs

Martin Tournoij

Bitmasks is one of those things where the basic idea is simple to understand: it's just `0`s and `1`s being toggled on and off. But actually "having it click" to the point where it's easy to work with can be a bit trickier. At least, it is (or rather, was) for me 😅

With a bitmask you hide (or "mask") certain bits of a number, which can be useful for various things as we'll see later on. There are two reasons one might use bitmasks: for efficiency or for nicer APIs. Efficiency is rarely an issue except for some embedded or specialized use cases, but everyone likes nice APIs, so this is about that.

A while ago I added colouring support to my little zli library. Adding colours to your terminal is not very hard as such, just print an escape code:

```
fmt.Println("\x1b[34mRed text!\x1b[0m")
```

But a library makes this a bit easier. There's already a bunch of libraries out there for Go specifically, the most popular being Fatih Arslan's `color`:

```
color.New(color.FgRed).Add(color.Bold).Add(color.BgCyan).Println("bold
red")
```

This is stored as:

```
type (
    Attribute int
    Color     struct { params  []Attribute }
)
```

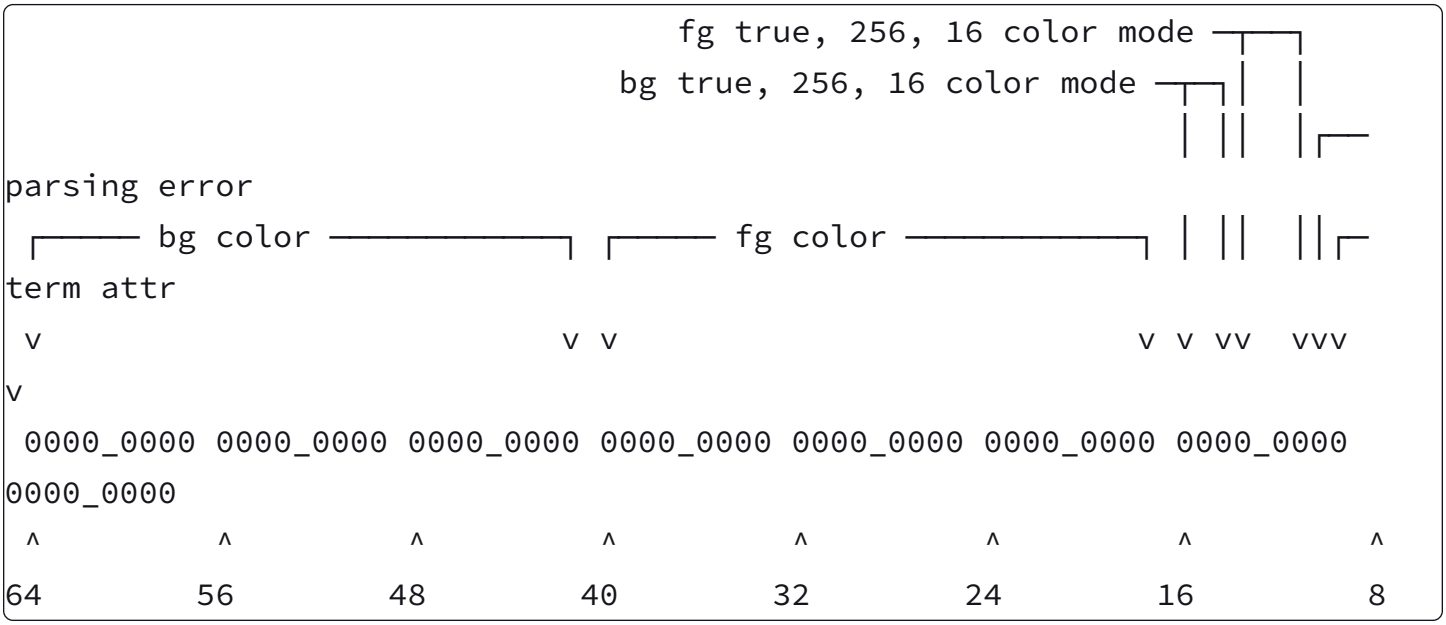
I wanted a simple way to add some colouring, which looks a bit nicer than the method chain in the `color` library, and eventually figured out you don't need a `[]int` to store all the different attributes but that a single `uint64` will do as well:

```
zli.Colorf("bold red", zli.Red | zli.Bold | zli.Cyan.Bg())

// Or alternatively, use Color.String():
fmt.Printf("%sbold red%s\n", zli.Red|zli.Bold|zli.Cyan.Bg(), zli.Reset)
```

Which in my eyes looks a bit nicer than Fatih's library, and also makes it easier to add 256 and true colour support.

All of the below can be used in any language by the way, and little of this is specific to Go. You will need Go 1.13 or newer for the binary literals to work.

Here's how `zli` stores all of this in a `uint64`:



I'll go over it in detail later, but in short (from right to left):

The first 9 bits are flags for the basic terminal attributes such as bold, italic, etc.

The next bit is to signal a parsing error for true colour codes (e.g. `#123123`).

There are 3 flags for the foreground and background colour each to signal that a colour should be applied, and how it should be interpreted (there are 3 different ways to set the colour: 16-colour, 256-colour, and 24-bit "true colour", which use different escape codes).

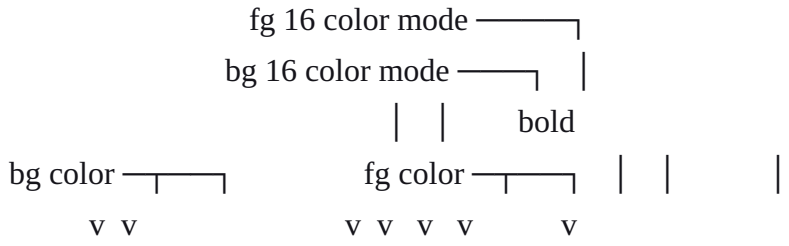
The colours for the foreground and background are stored separately, because you can apply both a foreground and background. These are 24-bit numbers.

A value of 0 is reset.

With this, you can make any combination of the common text attributes, the above example:

```
zli.Colorf("bold red", zli.Red | zli.Bold | zli.Cyan.Bg())
```

Would be the following in binary layout:



0000_0000 0000_0000 0000_0**11**0 0000_0000 0000_0000 0000_000**1** 00**1**0_0**1**00 0000_000**1**

^ ^ ^ ^ ^ ^ ^ ^

64 56 48 40 32 24 16 8

We need to go through several steps to actually do something meaningful with this. First, we want to get all the flag values (the first 9 bits); a “flag” is a bit being set to true (`1`) or false (`0`).

```
const (
    Bold          = 0b0_0000_0001
    Faint         = 0b0_0000_0010
    Italic        = 0b0_0000_0100
    Underline     = 0b0_0000_1000
    BlinkSlow     = 0b0_0001_0000
    BlinkRapid    = 0b0_0010_0000
    ReverseVideo = 0b0_0100_0000
    Concealed     = 0b0_1000_0000
    CrossedOut    = 0b1_0000_0000
)

func applyColor(c uint64) {
    if c & Bold != 0 {
        // Write escape code for bold
    }
    if c & Faint != 0 {
        // Write escape code for faint
    }
    // etc.
}
```

& is the bitwise AND operator. It works just as the more familiar && except that it operates on every individual bit where `0` is `false` and `1` is `true`. The end result will be `1` if both bits are “true” (`1`). An example with just four bits:

This can be thought of as four separate operations (from left to right):

```
0 AND 0 = 0      both false
0 AND 1 = 0      first value is false, so the end result is false
1 AND 0 = 0      second value is false
1 AND 1 = 1      both true
```

So what `if c & Bold != 0` does is check if the “bold bit” is set:

Only bold set:
 0 0000 000**1**
& 0 0000 000**1**
= ———————————
 0 0000 000**1**

Underline bit set:
 0 0000 **1**000
& 0 0000 000**1**
= ———————————
 0 0000 0000 0 since there are no cases of "1 AND 1"

Bold and underline bits set:
 0 0000 **1**00**1**
& 0 0000 000**1**
= ———————————
 0 0000 000**1** Only "bold AND bold" is "1 AND 1"

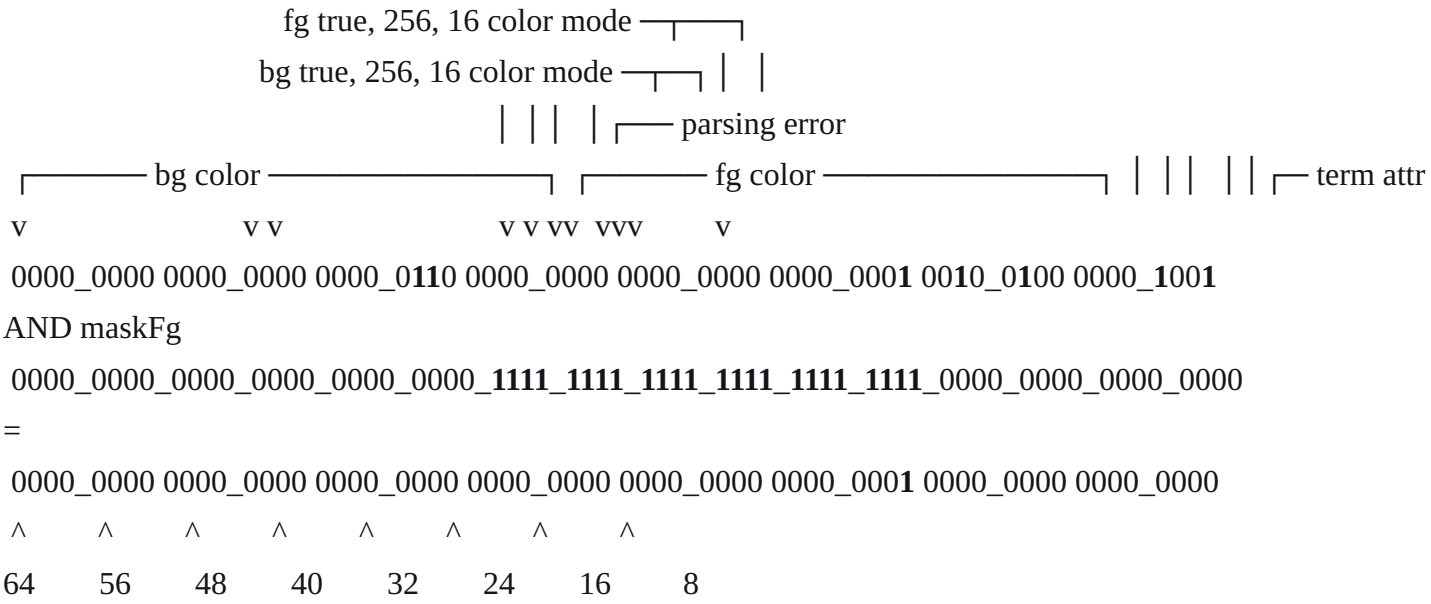
As you can see, `c & Bold != 0` could also be written as `c & Bold == Bold`.

The colours themselves are stored as a regular number like any other, except that they’re “offset” a number of bits. To get the actual number value we need to clear all the bits we don’t care about, and shift it all to the right:

```
const (
    colorOffsetFg   = 16

    colorMode16Fg   = 0b0000_0100_0000_0000
    colorMode256Fg  = 0b0000_1000_0000_0000
    colorModeTrueFg = 0b0001_0000_0000_0000

    maskFg          =
0b00000000_00000000_00000000_11111111_11111111_11111111_00000000_00000000
)

func getColor(c uint64) {
    if c & colorMode16Fg != 0  {
        cc := (c & maskFg) >> colorOffsetFg
```

```
        // ..write escape code for this color..
    }
}
```

First we check if the “16 colour mode” flag is set using the same method as the terminal attributes, and then we AND it with `maskFg` to clear all the bits we don’t care about:



After the AND operation we’re left with just the 24 bits we care about, and everything else is set to `0`. To get a normal number from this we need to shift the bits to the right with `>>`:

10**1**0 >> 1 = 0**1**0**1** All bits shifted one position to the right.
10**1**0 >> 2 = 00**1**0 Shift two, note that one bit gets discarded.

Instead of `>> 16` you can also subtract `65535` (a 16-bit number): `(c & maskFg) - 65535`. The end result is the same, but bit shifts are much easier to reason about in this context.

We repeat this for the background colour (except that we shift everything 40 bits to the right). The background is actually a bit easier since we don’t need to AND anything to clear bits, as all the bits to the right will just be discarded:

For 256 and “true” 24-bit colours we do the same, except that we need to send different escape codes for them, which is a detail that doesn’t really matter for this explainer about bitmasks.

To set the background colour we use the `Bg()` function to transforms a foreground colour to a background one. This avoids having to define `BgCyan` constants like Fatih’s library, and makes working with 256 and true colour easier.

```
const (
    colorMode16Fg   = 0b00000_0100_0000_0000
```

```
    colorMode16Bg   = 0b0010_0000_0000_0000

    maskFg          =
0b00000000_00000000_00000000_11111111_11111111_11111111_00000000_00000000
)

func Bg(c uint64) uint64 {
    if c & colorMode16Fg != 0 {
        c = c ^ colorMode16Fg | colorMode16Bg
    }
    return (c &^ maskFg) | (c & maskFg << 24)
}
```

First we check if the foreground colour flags is set; if it is then move that bit to the corresponding background flag.

`|` is the OR operator; this works like `||` except on individual bits like in the above example for &. Note that unlike `||` it won't stop if the first condition is false/0: if any of the two values are `1` the end result will be `1`:

0 OR 0 = 0 both false
0 OR 1 = 1 second value is true, so end result is true
1 OR 0 = 1 first value is true
1 OR 1 = 1 both true

0011
| 0101
= ————————
0111

`^` is the "exclusive or", or XOR, operator. It's similar to OR except that it only outputs `1` if exactly one value is `1`, and not if both are:

0 XOR 0 = 0 both false
0 XOR 1 = 1 second value is true, so end result is true
1 XOR 0 = 1 first value is true
1 XOR 1 = 0 both true, so result is 0

0011
^ 0101
= ————————

0**11**0

Putting both together, `c ^ colorMode16Fg` clears the foreground flag and `| colorMode16Bg` sets the background flag.

The last line moves the bits from the foreground colour to the background colour:

```
return (c &^ maskFg) | (c & maskFg << 24)
```

`&^` is “AND NOT”: these are two operations: first it will inverse the right side (“NOT”) and then ANDs the result. So in our example the `maskFg` value is inversed:

0000_0000_0000_0000_0000_0000_1111_1111_1111_1111_1111_1111_0000_0000_0000_0000
NOT
1111_1111_1111_1111_1111_1111_0000_0000_0000_0000_0000_0000_1111_1111_1111_1111

We then used this inversed `maskFg` value to clear the foreground colour, leaving everything else intact:

1111_1111_1111_1111_1111_1111_0000_0000_0000_0000_0000_0000_1111_1111_1111_1111
AND
0000_0000 0000_0000 0000_0**11**0 0000_0000 0000_0000 0000_000**1** 00**1**0_0**1**00 0000_**1**00**1**
=
0000_0000 0000_0000 0000_0**11**0 0000_0000 0000_0000 0000_0000 00**1**0_0**1**00 0000_**1**00**1**
^ ^ ^ ^ ^ ^ ^ ^
64 56 48 40 32 24 16 8

C and most other languages don’t have this operator and have ~ for NOT (which Go doesn’t have), so the above would be `(c & ~maskFg)` in most other languages.

Finally, we set the background colour by clearing all bits that are not part of the foreground colour, shifting them to the correct place, and ORing this to get the final result.

I skipped a number of implementation details in the above example for clarity, especially for people not familiar with Go. The full code is of course available. Putting all of this together gives a fairly nice API IMHO in about 200 lines of code which mostly avoids boilerplateism.

I only showed the 16-bit colours in the examples, in reality most of this is duplicated for 256 and true colours as well. It’s all the same logic, just with different values. I also skipped over the details of terminal colour codes, as this article isn’t really about that.

In many of the above examples I used binary literals for the constants, and this seemed the best way to communicate how it all works for this article. This isn’t necessarily the best or easiest way to write things in actual code, especially not for such large numbers. In the actual code it looks like:

```
const (
    ColorOffsetFg = 16
    ColorOffsetBg = 40
)

const (
    maskFg Color = (256*256*256 - 1) << ColorOffsetFg
    maskBg Color = maskFg << (ColorOffsetBg - ColorOffsetFg)
)

// Basic terminal attributes.
const (
    Reset Color = 0
    Bold  Color = 1 << (iota - 1)
    Faint
    // ...
)
```

Figuring out how this works is left as an exercise for the reader :-)

Another thing that might be useful is a little helper function to print a number as binary; it helps visualise things if you're confused:

```
func bin(c uint64) {
    reBin := regexp.MustCompile(`([01])([01])([01])([01])([01])([01])
([01])([01])`)
    reverse := func(s string) string {
        runes := []rune(s)
        for i, j := 0, len(runes)-1; i < j; i, j = i+1, j-1 {
            runes[i], runes[j] = runes[j], runes[i]
        }
        return string(runes)
    }
    fmt.Printf("%[2]s → %[1]d\n", c,
        reverse(reBin.ReplaceAllString(reverse(fmt.Sprintf("%064b", c)),
            `$1$2$3${4}_$5$6$7$8 `)))
}
```

I put a slighly more advanced version of this at zgo.at/zstd/zfmt.Binary.

You can also write a little wrapper to make things a bit easier:

```
type Bitflag64 uint64 uint64

func (f Bitflag64) Has(flag Bitflag64) bool { return f&flag != 0 }
func (f *Bitflag64) Set(flag Bitflag64)     { *f = *f | flag }
func (f *Bitflag64) Clear(flag Bitflag64)   { *f = *f &^ flag }
func (f *Bitflag64) Toggle(flag Bitflag64)  { *f = *f ^ flag }
```

If you need more than 64 bits then not all is lost; you can use `type thingy [2]uint64`.

Here's an example where I did it wrong:

```
type APITokenPermissions struct {
    Count      bool
    Export     bool
    SiteRead   bool
    SiteCreate bool
    SiteUpdate bool
}
```

This records the permissions for an API token the user creates. Looks nice, but how do you check that *only* `Count` is set?

```
if p.Count && !p.Export && !p.SiteRead && !p.SiteCreate && !p.SiteUpdate {
.. }
```

Ugh; not very nice, and neither is checking if multiple permissions are set:

```
if perm.Export && perm.SiteRead && perm.SiteCreate && perm.SiteUpdate { ..
}
```

Had I stored it as a bitmask instead, it would have been easier:

```
// Only Count is set.
if perm &^ Count == 0 { .. }

// Everything in permSomething is set.
const permSomething = perm.Export | perm.SiteRead | perm.SiteCreate |
perm.SiteUpdate
if perm & permSomething == 0 { .. }
```

No one likes functions with these kind of signatures either:

```
f(false, false, true)
f(true, false, true)
```

But with a bitmask things can look a lot nicer:

```
const (
    AddWarpdrive   = 0b001
    AddTractorBeam = 0b010
    AddPhasers     = 0b100
)

f(AddPhasers)
f(AddWarpdrive | AddPhasers)
```