

# Working with bits and bitfields

2008-05-28 23:11

- 1 Introduction
- 2 Basic bit operations
- 3 Bitfields
- 4 Summary

## 1 Introduction

There are several levels at which one can program. At the top level, there are things like GUIs and forms and Wizards that can do much of the programming for you. You make something without ever having seen any actual code. If you screw up, chances are that the environment points out the mistake and perhaps even fixes it for you. At the other end are the raw bits and bytes. That's where the real work is actually done. Programming at that level is rough, messy and unforgiving; but if you need things to be memory efficient, that's where you have to go.

Real Programmers like the lower levels exactly *because* it's unforgiving. You have to be clever to make it work, and make it work well. Normal people, however, wouldn't mind if manual bit-fiddling would die a slow and horrible death (perhaps taking the Real Programmers with it; the last thing you want is to maintain code that tries to be clever).

For the most part, the latter group has got their wish: computers are fast enough and memory cheap enough that bit-fiddling are mostly irrelevant. However, there are still a few niches where it's important and one of these is, of course, console programming.

In this post, I'll show the basics of fiddling with individual bits and bitfields, and present a few routines that should make things a little easier for everyone. This includes:

- Macros for setting, clearing and flipping bits.
- Easy creation of masks for bits and bitfields.
- Macros for bitfield manipulation.
- Native C bitfields, some potential problems with them, and how you can use both C bitfields and manual bit-manipulation on the same variables.

## 2 Basic bit operations

There are three things you can do to a bit:

- Set bit:** force into a '1' state.
- Clear bit:** force into a '0' state.
- Flip bit:** go from '0' to '1' and vice versa.

Now, suppose you have a variable *y* and you want to do something to the bits indicated by bit-mask *mask*. The general form for this would be `y = y OP mask`, where *OP* is OR (inclusive OR), AND NOT, and XOR (exclusive OR), respectively. Note that it's 'AND NOT' for the second: 'AND' by itself keeps bits and clear the non-masked bits. In C, this can be done with the following macros:

```
#define BIT(n)                ( 1<<(n) )

#define BIT_SET(y, mask)      ( y |= (mask) )
#define BIT_CLEAR(y, mask)    ( y &= ~(mask) )
#define BIT_FLIP(y, mask)     ( y ^= (mask) )

/*
   Set bits      Clear bits      Flip bits
y      0x0011    0x0011         0x0011
mask   0x0101 |   0x0101 &~    0x0101 ^
-----
result 0x0111    0x0010         0x0110
*/

// Examples:
mask= BIT(0) | BIT(8); // Create mask with bit 0 and 8 set
(0x0101)

    BIT_SET(y, mask); // Bits 0 and 8 of y have been set.
    BIT_CLEAR(y, mask); // Bits 0 and 8 of y have been
cleared.
    BIT_FLIP(y, mask); // Bits 0 and 8 of y have been
flipped.
```

The `BIT()` macro may be useful to create the bitmask: you can create the full mask by ORring the bits together. The other macros, `BIT_SET()`, `BIT_CLEAR()` and `BIT_FLIP()` will operate on the the bits indicated by the mask, and *only* those bits. They leave the rest intact. Note that these procedures must be macros; functions won't do because we have to write back to the variable. Before you mention pointers and references, remember that datatypes have sizes and you can't pass a u16-pointer to a u32-pointer parameter.

## 3 Bitfields

### 3.1 Manual bitfield manipulation

A **bitfield** is a range of bits working as a single number. You usually can't access these ranges directly because memory is accessed in (multi-)byte-sized datatypes (yes I know about C bitfield, we'll get to that later). Each bitfield starts at bit *start* and has a length *len*.

Table 1: GBA OAM attribute 2.

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Palbank				Prio											

As an example, lets look at the GBA OAM attributes, specifically attribute 2 (table 1). The `attr2` is 16 bits long with bits 0-9 acting as a tile index, 10 and 11 as the priority field and 12-15 as the palette-bank index.

Suppose you want to get the value of the priority, which starts at bit 10 and is 2 bits long. You can't get it directly because it's buffeted at both sides and because it's ten bits up. We'll take care of the last part first: right-shift by `start = 10` to move the field's bits into the lowest bits. Note that this also removes the tile bitfield. Next we mask off `len = 2` bits (i.e., `mask = 3`) to remove the higher bits.

```
bbbbPPtttttttttt // y= attr2
0000000000bbbbPP // y>>10
00000000000000PP // (y>>10)&3
```

Setting a bitfield to a new value, `x` is basically the inverse of this: left-shift the new value by `start` and OR it in. However, this doesn't quite work for two reasons: first, OR would combine it with the value already present, which is rarely what you want. Second, if the new value is too large for the field it'll bleed into the next bitfield. The result of this could at best be described as 'interesting'. To get around these problems, you have to clear the whole bitfield first and mask off the excess bits from the new value before insertion.

```
x: *****nn // x= new value
x: 00000000000000nn // x&3
x: 0000nn0000000000 // x= (x&3)<<10

y: bbbbPPtttttttttt // y= attr2
y: bbbb00tttttttttt // y= (y&~(3<<10))
y: bbbbnntttttttttt // y= y | x
```

So the general expressions for getting and setting bitfields are `(y>>start)&mask` and `y = (y&~mask) | ((x&mask)<<start)`, respectively. I'm giving you the expressions now because in macro-form, they become quite horrible thanks to the required parentheses. The one thing I've left out is how to get the mask in the first place. There are a number of expressions for this, but the simplest one is to take `BIT(len)` and subtract one.

```
//! Create a bitmask of length \a len.
#define BIT_MASK(len) ( BIT(len)-1 )

//! Create a bitfield mask of length \a starting at bit \a start.
#define BF_MASK(start, len) ( BIT_MASK(len)<<(start) )

//! Prepare a bitmask for insertion or combining.
#define BF_PREP(x, start, len) ( ((x)&BIT_MASK(len)) << (start) )

//! Extract a bitfield of length \a len starting at bit \a start
from \a y.
#define BF_GET(y, start, len) ( ((y)>>(start)) & BIT_MASK(len) )

//! Insert a new bitfield value \a x into \a y.
#define BF_SET(y, x, start, len) \
( y= ((y) &~ BF_MASK(start, len)) | BF_PREP(x, start, len) )
```

Yeah, yeah, I know how it looks. Now think of how things would look if you had to write out these expressions manually every time. If you want them to look a little more presentable, you can also use inline functions instead (mostly anyway):

```
static inline u32 bf_get(u32 y, u32 shift, u32 len)
{
    return (y>>shift) & BIT_MASK(len);
}
```

```
static inline u32 bf_merge(u32 y, u32 x, u32 shift, u32 len)
{
    u32 mask= BIT_MASK(len);
    return (y &~ (mask<<shift)) | (x & mask)<<shift;
}
```

Usage of these routines is pretty simple. Note that for `bf_merge()`, you'll have to actually assign the merged value yourself. That's why I've named it `bf_merge()` and not `bf_set()`.

```
// Retrieve priority value from attr2
prio= BF_GET(attr2, 10, 2);
// or
prio= bf_get(attr2, 10, 2);

// Set priority value of attr2 to x
BF_SET(attr2, x, 10, 2);
// or
attr2= bf_merge(attr2, x, 10, 2);
```

As complicated as these functions look, they can actually be quite fast. As the start and length of the bitfield will generally be constant, creating the mask and shifts will be done at compile time. It'll still be slower than having separate variables, of course.

There is one potential snare with all of this, however. Once you extract a bitfield, they are unshifted. This could be problematic if you want to compare the values to named constants you may have. If something like `ATTR2_Prio_1` is defined for priority 1, it will almost certainly be defined as `BIT(10)` and not 1. In such cases, it'd be better to compare with  ``(attr2 & BF_MASK(10, 2)) == ATTR2_Prio_1` or something like it.`

### 3.2 Named bitfields

Having these routines for bitfield manipulation is nice and all, but the magic numbers are a little evil. For one thing, you might forget with argument goes first (that's one of the reasons by inline functions are better: they are visible by intellisense). So you could create named constants for the starts and lengths and use those as arguments.

A variation of this is to not use the start and length, but the start and *mask*. Those two could be useful in other areas as well anyway. For example, for the priority bitfield you could have `ATTR2_Prio_SHIFT` (=10) and `ATTR2_Prio_MASK` (=BF\_MASK(10, 2)). If you use this naming structure, consistently, you can do something very nice: let the macros create the full names themselves using the concatenation operator `##`. This saves typing and reading.

```
//! Message \a x for use in bitfield \a name.
#define BFN_PREP(x, name)    ( ((x)<<name##_SHIFT) & name##_MASK )

//! Get the value of bitfield \a name from \a y. Equivalent to
(var=) y.name
#define BFN_GET(y, name)    ( ((y) & name##_MASK)>>name##_SHIFT )

//! Set bitfield \a name from \a y to \a x: y.name= x.
#define BFN_SET(y, x, name) ( y = ((y)&~name##_MASK) |
BFN_PREP(x, name) )

// Usage: prio get/set like before:
prio= BFN_GET(attr2, ATTR2_Prio);
BFN_SET(attr2, x, ATTR2_Prio);
```

The *name* parameter is expanded into *name\_SHIFT* and *name\_MASK*. Assuming those exist in your headers, these named-bitfield macros shouldn't be fairly simple.

#### Tonclib compatibility

Tonclib has the named bitfield macros (indicated by `BFN_` here), but not the non-named variants (here `BF_`). To make matters worse, the `BFN_` macros used here carry the `BF_` prefix in tonclib, because the named variants came first and it's too late to change them. We apologize for the inconvenience.

### 3.3 Native bitfield constructs in C

So far I've done all the bit-manipulation manually, but C also has bitfield construct of its own. The following is an example of GBA screen entries expressed as C bitfields.

```
// GBA screen entry as a bitfield struct.
typedef struct scr_t
{
    u16 tile      : 10;
    u16 hflip     : 1;
    u16 vflip     : 1;
    u16 pal       : 4;
} scr_t;
```

```
// --- Usage: ---
scr_t se;
se.tile = 42;    // Set tile index to 42.
se.hflip= 1;    // Set horizontal flipping.
se.pal  = 15;    // Use palette 15.
```

C bitfields are indicated by the colon&number combination in the declaration; in this case 10 bits for the `tile` field,  $2 \times 1$  for the flipping flags and 4 for the palette-bank for a total of 16. When part of a struct, you can use the bitfields exactly like regular members. Note that all the masking and ORing still happens, but now it's done out of view by the compiler.

As you can see, using bitfields like this is very easy. You may wonder why I didn't start with this in the first place. Well, for a number of reasons, really. First, all the slow bitwork is still there, and because all fields are accessed separately you can't set multiple fields in one statement. There's also a very occasional bug in the way signs are handled: if you're using signed integers, the bitfields will also be signed, and a 1-bit signed integer can take values 0 and -1, not 0 and 1. Also, how bitfields are implemented depends on the platform (think Endianness). But even worse, it depends on compiler too. For example, it may look like a 16-bit variable here, but the size of `scr_t` depends on the version of GCC you have: in old versions structs were always word-aligned, but nowadays (devkitPro r19 and up) they are aligned to the size of their largest member (see also [tonc:data](#)). Lastly, **it doesn't always work!** At least, not always when it comes to the GBA or DS.

That last part should be a little surprising, so let me elaborate. It has to do with the way GCC implements bitfields and how some GBA memory sections work. GCC's bitfield implementation is quite clever. For each bitfield, it will always access the smallest amount of space required for the action. For example, `scr_t.tile` is 10-bits long, so it uses a halfword. `scr_t.pal` is 4-bits long, so it will use a byte. And that's exactly the problem: the screen-entries exist in VRAM which cannot be written to in bytes. Here's an illustration of what really happens:

```
// Memory-map VRAM as a screen-entry array.
#define scr_mem ((scr_t*)0x06000000)

// Set a pal-bank value (hopefully):
{
    scr_mem[0].pal= 1;
}

// What really happens here:
{
    u8 pal, *ptr;
    ptr = (u8*)0x06000000;
    pal = ptr[1];           // Get high-byte from scr_mem[0]
    pal &= 15;              // Clear top 4 bits (the pal field)
    pal |= 16;              // Insert 1<<4
    ptr[1] = pal;           // Write byte to VRAM, OH SHI-
```

In the last step, a simple byte is written back into memory. In most types of memory this should work fine, but GBA VRAM (as well as PAL-RAM, OAM, and ROM) has the nasty quirk of writing that byte to **both** bytes of that halfword. So you end up with 0x1010 instead of 0x1000. There's a simple way around this: buffer the data in RAM and work on it there. That might not always be a convenient option, though.

### C bitfields and GBA memory

GCC may use byte-writes for the C bitfield construct, which won't work with certain sections of GBA memory. Do not use them for PAL-RAM, VRAM, OAM or ROM directly.

Finally there's a way of using both C bitfields and manual manipulation, thanks to the joy (\*cough\* \*cough\*) of unions. The syntax can be used to frighten small children, but it does work:

```
//! Normal screen-entry type.
typedef struct scr_t
{
    union {
        u16 data;
        struct {
            u16 tile    : 10;
            u16 hflip   : 1;
            u16 vflip   : 1;
            u16 pal     : 4;
        };
    };
} scr_t;

// --- Usage: ---
scr_t se;

se.pal= 1;           // Set via C bitfields
```

```
se.data= 0x1000;           // Set manually (overwrites all fields)
BF_SET(se.data, 1, 12, 4); // Set manually (no overwrite)
```

Note how the `scr_t` works. With a union, you can create overlays for memory; in this case for data and the bitfields. The struct is necessary because otherwise the bitfield parts wouldn't work. Once the compiler hits the semi-colon of a member, the next member starts, bitfields be damned. But if you encapsulate them inside a struct, the struct is seen as a unit and it all works out. By using *anonymous* unions and structs, you can use the union and struct members as if they were members of the `scr_t` struct itself, rather than having to do something like `se.union_name.struct_name.pa1`. One small downside of this method is that you can't initialize variables and arrays of `scr_t`'s because the compiler doesn't know with part of the union to use. It will accept something like

```
scr_t array= { 0x1111, 0x2222 };
```

but you will get warnings for it.

Using both bitfield members and full members can be useful, but you need to remember carefully which names are the bitfields. With manual bitfield manipulation, you have to apply the shifts yourself; but with the bitfield versions the shifts should not be present. For example, the palette field starts at bit 12. So to manually set this, you'd use `0x1000`, for example. But if you have a bitfield member, you'd just use `1`. The difference is obvious here, but sometimes there may be macros for the manual way, such as this:

```
#define SE_PALBANK(n)    ((n)<<12)
```

This is to be used on the full member, not the bitfield. Please keep an eye out for such occurrences.

## 4 Summary

So, recap.

- **Bit operations**

There are three basic bit operations: *set*, *clear* and *flip*. These correspond to OR, AND NOT and XOR; or, in C speak, `|`, `&~` and `^`. The macros for these are `BIT_SET()`, `BIT_CLEAR()` and `BIT_FLIP()`.

- **Bitfield operations**

Bitfields are ranges of bits inside a variable that act as a sub-variable. The basic operations here are *set* and *get*. For both of these, you need to be careful not to disturb the other bits in the same variable (masking) and to shift left/right to align the value with the bitfield. The macros for these are `BIT_SET()` and `BIT_GET()`.

- **Macros for named bitfields**

If you're doing manual bitfields, it is often useful to have named literals for the starting bit and the number of bits in the bitfield, say `foo_SHIFT` and `foo_MASK`. The `BFN_SET()` and `BFN_GET()`. These macros make use of the preprocessor concatenation operator so that you only have to add the name-part (*foo* here) as the last argument.

- **C bitfields**

C also has a construct for bitfields so you don't have to do all the stuff manually if you don't want to. The best way to use them would be to build a struct around them and then access the bitfields just as if they were normal members. The compiler will take care of all the masks, shifts and ors, but how it does that exactly is compiler dependent, and may not be terribly efficient.

One particular danger of C bitfields for GBA/NDS purposes is that it may use byte-writes where those are not allowed (addresses above `0500:0000`, basically). Another ... *interesting* ... part is how signed and unsigned bitfields work. Use unsigned types unless you have a good reason not to.

To have your cake and eat it too, you can create a struct with an anonymous union so that you can have access to both the bitfields separately and the variable as a whole. The format for this is not pretty but it does allow maximum efficiency in terms of code-size and speed. Do remember which member is the whole thing and which is the bitfield.

16 THOUGHTS ON "WORKING WITH BITS AND BITFIELDS"




Kawa

on 2008-05-30 at 19:52 said:

I \*HATE\* bit fiddling!




Ged

 on 2008-10-12 at 20:27 said:

Great article!

Travis H.

 on 2009-05-12 at 15:09 said:

Some of these bit field things are wrong.

Here's a tested version:

```
/* Create a bitmask of length len */
#define BIT_MASK(len) (BIT(len)-1)

/* Create a bitfield mask of length len starting at
bit start */
#define BF_MASK(start, len) ( BIT_MASK(len)
<<(start) )


/* Prepare a bitmask for insertion or combining */
#define BF_PREP(x, start, len) (((x)&BIT_MASK(len))
<<(start)) & BIT_MASK(len) )

/* Extract a bitfield of length len starting at bit
start from y */
#define BF_GET(y, start, len) ( ((y)>>
(start))&BIT_MASK(len) )

/* Insert a new bitfield value x into y */
#define BF_SET(y, x, start, len) \
    (y=((y) &~ BF_MASK(start, len)) |
BF_PREP(x, start, len))
```

**NOTE: this comment is a reconstruction after a recent DB fail - jv**

cearn

 on 2009-05-12 at 15:55 said:

Yeah, BF\_GET ( ) was wrong. As far as I can see, the rest are correct though.

cloud9ine


 on 2009-09-03 at 19:58 said:

Is there a way to use prioritized bitfields and directly retrieve the highest priority bit that is set / cleared within it?

For example say, I have a 10-bit bitfield that represents whether each of faults 0 through 9 are active now. Say fault 9 is highest priority and fault 0 is lowest. Is there a way I can just use a function and say 'get\_highest\_priority\_position\_set\_in\_bitfield(bitfield)' and the function will return, say, 6?

I know a loop could but I am looking for better.


cearn

 on 2009-09-09 at 18:16 said:

I don't believe it's possible, no. I'm not sure that a ten-iteration loop is that bad, though. If you must, you can grab the whole set and go through it in a binary search rather than one by one.

If you have it, a Count Leading Zero instruction (CLZ) would help out as well. And even if you don't, there are fast algorithms out there that can do this as well.

Dan

 on 2010-10-12 at 16:28 said:

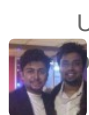
@cloud9ine:

Sure, just grab the log-2 of the number. For example:

48:  $\log(48) / \log(2) = 5.58$  or thereabouts; bit 6 ( $2^5 = 32$ ) is the highest bit set.

8248:  $\log(8248) / \log(2) = 13.01$  or so; bit 14 ( $2^{13} = 8192$ ) is the highest bit set.

Don't forget to special-case for a value=0, and remember, you want  $\text{floor}(\log(x)/\log(2))$ , possibly with a +1 at the end; trying to shortcut by indiscriminately rounding up will return the wrong value if only your target bit is set (e.g.  $\log(2)/\log(2) = 1.0000$ , and shouldn't be rounded up even if lack of FP precision means you get 1.00000001).



Ujjwal Thaakar

on 2011-12-23 at 18:50 said:

Pretty nice article !



Kevin

on 2013-03-14 at 17:06 said:

Just a tip, if you want a set of bits (example, three in a row, 0b00000111) Just shift a one over that many places then subtract one.

// If you want 4 1s in a row:

```
mask = (1<<4)-1;
```

// You want 2 0s at the beginning.

```
mask = (1<<2)-1;
```

```
mask = ~mask;
```

This is nice because it works even when you don't know the number you need at compile time.



cearn

on 2013-03-19 at 1:41 said:

That's basically what the `BIT_MASK( )` macro does :P. Though I did forget to mention inverting it for creating a hi-mask.



Santhi P

on 2014-12-23 at 7:12 said:

Thanks for the useful information on Macros in Cpp. You can find more information in Macros with examples here in the following link.

[Macros in Cpp with examples](#)



Alexander

on 2017-01-26 at 21:12 said:

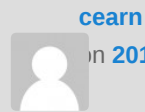
Are the macros to set bits really correct ? I'd like to set the exponent of a C double to 0, but MinGW/GCC 5.3.0 always complains. (The exponent is stored in bits 52 to 62.) `BF_GET`, however, works perfectly.

```
typedef union {
double f; /* input and output, the 64-bit double precision value */
uint64_t i; /* unsigned 64-bit int representation */
} Double;
```

```
static int math_mantissa2 (lua_State *L) { /* 2.10.0 */
Double d;
d.f = agn_checknumber(L, 1);
BF_SET(d.i, 0, 52, 11); /* zero exponent */
```

```
lua_pushnumber(L, d.f);
return 1;
}
```

```
gcc -Wall -O3 -DLUA_BUILD_AS_DLL -c -o lmathlib.o lmathlib.c
lmathlib.c: In function 'math_mantissa2':
lmathlib.c:1199:48: warning: left shift count >= width of type [-Wshift-count-overflow]
#define BF_MASK(start, len) ( BIT_MASK(len)<= width of type [-Wshift-count-overflow]
#define BF_PREP(x, start, len) (( (x)&BIT_MASK(len))<<((start) & BIT_MASK(len) )
```



cearn

on 2017-01-29 at 2:18 said:

Hi Alexander.

The macros are correct, but it's the BIT() macro that's the problem in your case. The "1" in (1<n) is 32-bit, so the result will as well. This will be wrong for n>32. If you change that to 1ull, it should work.

The best way to deal with this, though would be to use a set of templated inline functions, rather than macros. Those will almost always be better than macros. Something like:

```
template<typename T, typename S>
inline T bf_set(T& y, S x, int start, int len)
{
    T mask = (T(1)<<len)-1;
    y = y & ~(mask<<start) | (x&mask)<<start;
    return y;
}
```

Note that setting y inside the function is probably redundant, but you get the idea.

Pingback: [Hello world! | Siguino](#)



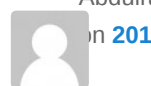
chrlie

on 2018-03-25 at 18:08 said:

Your macro BIT\_MASK does not work for len == 32.

You should instead write:

```
#define BIT_MASK(len) ((~(uint32_t)0) >> (32 - (len)))
```



Abdulrahman

on 2018-05-05 at 16:28 said:

Hmm didn't know how to apply those macros.