C Bitwise Operations Reference

For Embedded Systems & Register Manipulation

1 The Six Bitwise Operators

Operator	Name	Description	Example
`&`	AND	1 if BOTH bits are 1	`0b1100 & 0b1010 = 0b1000`
`|`	OR	1 if EITHER bit is 1	`0b1100 | 0b1010 = 0b1110`
`^`	XOR	1 if bits are DIFFERENT	`0b1100 ^ 0b1010 = 0b0110`
`~`	NOT	Flip ALL bits	`~0b1100 = 0b0011...`
`<<`	Left Shift	Shift bits left, fill with 0s	`0b0011 << 2 = 0b1100`
`>>`	Right Shift	Shift bits right	`0b1100 >> 2 = 0b0011`

2 Truth Tables

A	B	A & B	A — B	A ˆ B	˜A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Key Insights:

- **AND with 0** → Always 0 (clears bits)
- **AND with 1** → Keeps original value (preserves bits)
- **OR with 1** → Always 1 (sets bits)
- **OR with 0** → Keeps original value (preserves bits)
- **XOR with 1** → Flips the bit (toggles)
- **XOR with 0** → Keeps original value

3 Common Bit Manipulation Patterns

3.1 Single Bit Operations

Operation	Code	Explanation
Set bit n to 1	`reg |= (1 << n);`	OR with 1 at position n
Clear bit n to 0	`reg &= ~(1 << n);`	AND with 0 at position n
Toggle bit n	`reg ^= (1 << n);`	XOR with 1 at position n
Check if bit n is set	`if (reg & (1 << n))`	Non-zero if bit is 1
Read bit n as 0 or 1	`(reg >> n) & 1`	Shift down, mask lowest bit

3.2 Multiple Bit Operations (Bit Fields)

Operation	Code	Explanation
Clear 3 bits at pos n	`reg &= ~(7 << n);`	7 = 0b111 (3-bit mask)
Clear 4 bits at pos n	`reg &= ~(0xF << n);`	0xF = 0b1111 (4-bit mask)
Clear w bits at pos n	`reg &= ~(((1<<w)-1) << n);`	General formula
Set field to value	`reg |= (val << n);`	After clearing first!
Read w bits at pos n	`(reg >> n) & ((1<<w)-1)`	Shift, then mask

3.3 The Read-Modify-Write Pattern

Essential for changing specific bits without affecting others:

```
    // Change 3-bit field at position 12 to value 5
```

```
reg &= ~(7 << 12);     // Step 1: Clear the 3 bits
reg |=  (5 << 12);     // Step 2: Set new value
```

Why two steps?

- `=` overwrites ALL bits (destroys other fields)
- `|=` can only turn bits ON, never OFF
- `&=` can only turn bits OFF, never ON
- Need both: clear first, then set

4 Visual Examples

4.1 Setting a Single Bit

Goal: Set bit 5 in register

```
reg:         0110 0100        (original)
(1 << 5):    0010 0000        (bit 5 mask)

reg |= ...:  0110 0100        (result: bit 5 now set)
                  ^
                  bit 5
```

4.2 Clearing a Single Bit

Goal: Clear bit 2 in register

```
reg:         0110 0100        (original)
(1 << 2):    0000 0100        (bit 2 mask)
~(1 << 2):   1111 1011        (inverted: 0 at bit 2, 1s elsewhere)

reg &= ...:  0110 0000        (result: bit 2 cleared)
                  ^
                  bit 2
```

4.3 Modifying a 3-Bit Field

Goal: Set bits 12-14 to value 5 (0b101)

```
Original:    ... 0 110 0000 0000 0000    (bits 12-14 = 6)
                   ^^^
                   field to change

Step 1: Clear with &= ~(7 << 12)
(7 << 12):   ... 0 111 0000 0000 0000    (mask)
~(7 << 12):  ... 1 000 1111 1111 1111    (inverted mask)
After &=:    ... 0 000 0000 0000 0000    (field cleared)

Step 2: Set with |= (5 << 12)
(5 << 12):   ... 0 101 0000 0000 0000    (value to set)
After |=:    ... 0 101 0000 0000 0000    (field = 5)
                   ^^^
                   now contains 101
```

4.4 Reading a Bit Field

Goal: Read 3 bits starting at position 4

```
reg:         1010 1110 0000    (we want bits 4-6)
                  ^^^
                  these bits (value = 0b110 = 6)
```

```
Method: (reg >> 4) & 7

reg >> 4:    0000 1010 1110    (shift right by 4)
                      ^^^
                      bits now at position 0-2

& 7:         0000 0000 0110    (mask with 0b111)
                      ^^^
                      result = 6
```

5 Common Masks

Bits Needed	Decimal	Hex	Binary
1 bit	1	0x1	`0b1`
2 bits	3	0x3	`0b11`
3 bits	7	0x7	`0b111`
4 bits	15	0xF	`0b1111`
5 bits	31	0x1F	`0b11111`
6 bits	63	0x3F	`0b111111`
7 bits	127	0x7F	`0b1111111`
8 bits	255	0xFF	`0b11111111`
16 bits	65535	0xFFFF	16 ones
32 bits	4294967295	0xFFFFFFFF	32 ones

Formula: For w bits, mask = `(1 << w) - 1` or equivalently `(1U << w) - 1`

```
(1 << 3):      0b1000      (1 followed by 3 zeros)
(1 << 3) - 1:  0b0111      (3 ones) = 7
```

6 Compound Assignment Operators

Operator	Equivalent To	Use Case
`reg |= mask;`	`reg = reg | mask;`	Set bits
`reg &= mask;`	`reg = reg & mask;`	Clear bits (with ~)
`reg ^= mask;`	`reg = reg ^ mask;`	Toggle bits
`reg <<= n;`	`reg = reg << n;`	Shift left
`reg >>= n;`	`reg = reg >> n;`	Shift right

7 Shift Operations In Depth

7.1 Left Shift (<<)

```
x << n  =  x * 2^n  (multiply by power of 2)

0b0001 << 0 = 0b0001  (1)
0b0001 << 1 = 0b0010  (2)
0b0001 << 2 = 0b0100  (4)
0b0001 << 3 = 0b1000  (8)
```

Bits shifted out on the left are lost. Zeros fill from the right.

7.2 Right Shift (>>)

```
x >> n  =  x / 2^n  (divide by power of 2, truncated)

0b1000 >> 0 = 0b1000  (8)
0b1000 >> 1 = 0b0100  (4)
0b1000 >> 2 = 0b0010  (2)
0b1000 >> 3 = 0b0001  (1)
```

Warning: For signed types, right shift may fill with sign bit (implementation-defined). Use `unsigned` types for predictable behavior.

7.3 Creating Bit Positions

```
(1 << 0)  = 0b00000001  = 0x01  (bit 0)
(1 << 1)  = 0b00000010  = 0x02  (bit 1)
(1 << 2)  = 0b00000100  = 0x04  (bit 2)
(1 << 3)  = 0b00001000  = 0x08  (bit 3)
(1 << 4)  = 0b00010000  = 0x10  (bit 4)
(1 << 7)  = 0b10000000  = 0x80  (bit 7)
```

8 NOT Operator (˜) Deep Dive

The ˜ operator inverts ALL bits, including leading zeros:

```
8-bit example:
x      = 0b00001111  (0x0F)
~x     = 0b11110000  (0xF0)

32-bit example:
x      = 0x0000000F  (15)
~x     = 0xFFFFFFF0  (4294967280, or -16 if signed)
```

Common Pitfall:

```
// WRONG: ~ creates huge number!
reg |= ~(mode << 4);     // Probably not what you want

// RIGHT: Only use ~ for clearing
reg &= ~(7 << 4);        // Clear bits, preserves others
```

Rule: Use ˜ with `&=` (clearing). Never with `|=` (setting).

9 Register Access Patterns

9.1 Memory-Mapped I/O

```
#define PERIPHERAL_BASE  0xFE200000
#define REG_CONTROL      (*(volatile uint32_t *)(PERIPHERAL_BASE + 0x00))
#define REG_STATUS       (*(volatile uint32_t *)(PERIPHERAL_BASE + 0x04))

// Usage:
REG_CONTROL |= (1 << 3);         // Set bit 3
if (REG_STATUS & (1 << 7)) { }   // Check bit 7
```

9.2 Using Structs

```
struct PeripheralRegs {
    volatile uint32_t control;
    volatile uint32_t status;
    volatile uint32_t data;
};

#define PERIPH ((struct PeripheralRegs *)0xFE200000)

// Usage:
PERIPH->control |= (1 << 3);
if (PERIPH->status & (1 << 7)) { }
```

Why `volatile`? Tells compiler the value can change unexpectedly (hardware). Prevents dangerous optimizations.

10 Checking Multiple Bits

10.1 Check If ANY Bit Is Set

```
if (reg & mask) {
    // At least one bit in mask is set
}
```

10.2 Check If ALL Bits Are Set

```
if ((reg & mask) == mask) {
    // All bits in mask are set
}
```

10.3 Check If ALL Bits Are Clear

```
if ((reg & mask) == 0) {
    // All bits in mask are clear
}
```

10.4 Check Specific Field Value

```
// Read 3-bit field at position 4, check if equals 5
if (((reg >> 4) & 7) == 5) {
    // Field value is 5
}
```

11 XOR Tricks

Operation	**Code**	**Result**
Toggle bit	`x ^= (1 << n)`	Flips bit n
Swap without temp	`a^=b; b^=a; a^=b;`	a and b swapped
Check if equal	`if ((a ^ b) == 0)`	True if a == b
Self-XOR	`x ^ x`	Always 0

12 Operator Precedence (Pitfalls!)

Bitwise operators have **lower precedence** than comparison operators!

```
// WRONG! Parsed as: reg & (1 == 3)
if (reg & 1 == 3) { }

// RIGHT! Use parentheses
if ((reg & 1) == 3) { }
if (reg & (1 << 3)) { }
```

Precedence (high to low):

1. `~` (NOT)
2. `<< >>` (shifts)
3. `&` (AND)
4. `^` (XOR)
5. `|` (OR)

Rule: When in doubt, add parentheses!

13 Quick Reference Card

Essential Patterns	
Set bit n	`reg |= (1 << n);`
Clear bit n	`reg &= ~(1 << n);`
Toggle bit n	`reg ^= (1 << n);`
Read bit n	`(reg >> n) & 1`
Clear w bits at pos n	`reg &= ~(((1<<w)-1) << n);`
Set field to val	`reg |= (val << n);`
Read w bits at pos n	`(reg >> n) & ((1<<w)-1)`
Modify field	Clear first, then set

Mask	1-bit	2-bit	3-bit	4-bit	8-bit
Decimal	1	3	7	15	255
Hex	0x1	0x3	0x7	0xF	0xFF

Golden Rules:

1. Use `|=` to SET bits (with mask of 1s where you want to set)
2. Use `&= ~` to CLEAR bits (with mask of 1s where you want to clear)
3. To MODIFY a field: clear it first, then set it
4. Always use `unsigned` types for predictable shifting
5. Always use parentheses around bitwise operations in conditions
6. Use `volatile` for hardware registers