Create account Log in

Article Talk Read Edit View history Search

Bit field

From Wikipedia, the free encyclopedia

A **bit field** is a term used in computer programming to store multiple, logical, neighboring bits, where each of the sets of bits, and single bits can be addressed. A bit field is most commonly used to represent integral types of known, fixed bit-width. A well-known usage of bit-fields is to represent a set of bits, and/or series of bits, known as flags.[*citation needed*] For example, the first bit in a bit field can be used to determine the state of a particular attribute associated with the bit field.

A bit field is distinguished from a bit array in that the latter is used to store a large set of bits indexed by integers and is often wider than any integral type supported by the language. Bit fields, on the other hand, typically fit within a machine word, and the denotation of bits is independent of their numerical index.

Contents [hide]

Implementation [edit]

"A bit field is set up with a structure declaration that labels each field and determines its width."[1] In C and C++ bit fields can be created using unsigned int, signed int, or _Bool (in C99).

You can set, test, and change the bits in the field using a mask, bitwise operators, and the proper membership operator of a struct (. or ->). ORing a value will turn the bits on if they are not already on, and leave them unchanged if they are, e.g. bf.flag |= MASK; To turn a bit off, you can AND its inverse, e.g. bf->flag &= ~MASK; And finally you can toggle a bit (turn it on if it is off and off if it is on) with the XOR operator, e.g. (*bf).flag ^= MASK; To test a bit you can use an AND expression, e.g. (flag_set & MASK) ? true : false;

Having the value of a particular bit can be simply done by left shifting (<<) 1, n amount of times (or, x << n - log2(x) amount of times, where x is a power of 2), where n is the index of the bit you want (the right most bit being the start), e.g. if you want the value of the 4th bit in a binary number, you can do: 1 << 3; which will yield 8, or 2 << 2; etc. The benefits of this become apparent when iterating through a series of bits one at a time in a for loop, or when needing the powers of large numbers to check high bits.

If a language doesn't support bit fields, but supports bit manipulation, you can do something very similar. Since a bit field is just a group of neighboring bits, and so is any other primitive data type, you can substitute the bit field for a primitive, or array of primitives. For example, with a 32 bit integer represented as 32 contiguous bits, you could use it the same way as a bit field with one difference; with a bitfield you can represent a particular bit or set of bits using its named member, and a flag whose value is between 0, and 2 to the nth power, where n is the length of the bits.

Examples [edit]

Declaring a bit field in C:

```
#include <stdio.h>

// opaque and show
#define YES 1
#define NO  0

// line styles
#define SOLID  1
#define DOTTED 2
#define DASHED 3
```

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Deutsch
日本語
Polski
Русский
Tagalog
中文
Edit links

```
// primary colors
#define BLUE  4  /* 100 */
#define GREEN 2  /* 010 */
#define RED   1  /* 001 */

// mixed colors
#define BLACK   0                    /* 000 */
#define YELLOW  (RED | GREEN)        /* 011 */
#define MAGENTA (RED | BLUE)         /* 101 */
#define CYAN    (GREEN | BLUE)       /* 110 */
#define WHITE   (RED | GREEN | BLUE) /* 111 */

const char * colors[8] = {"Black", "Red", "Green", "Yellow", "Blue", " Magenta",
"Cyan", "White"};

// bit field box properties
struct box_props
{
     unsigned int opaque       : 1;
     unsigned int fill_color   : 3;
     unsigned int              : 4; // fill to 8 bits
     unsigned int show_border  : 1;
     unsigned int border_color : 3;
     unsigned int border_style : 2;
     unsigned int              : 0; // fill to nearest byte (16 bits)
     unsigned char width       : 4, // Split a byte into 2 fields of 4 bits
                   height      : 4;
};
```

[2]

Example of emulating bit fields with a primitive and bit operators in C:

```
/* Each prepocessor directive defines a single bit */
#define KEY_UP       (1 << 0)  /* 000001 */
#define KEY_RIGHT    (1 << 1)  /* 000010 */
#define KEY_DOWN     (1 << 2)  /* 000100 */
#define KEY_LEFT     (1 << 3)  /* 001000 */
#define KEY_BUTTON1  (1 << 4)  /* 010000 */
#define KEY_BUTTON2  (1 << 5)  /* 100000 */

int gameControllerStatus = 0;

/* Sets the gameCtrollerStatus using OR */
void keyPressed(int key) {
     gameControllerStatus |= key;
}

/* Turns the key in gameControllerStatus off using AND and ~ */
void keyReleased(int key) {
    gameControllerStatus &= ~key;
}

/* Tests whether a bit is set using AND */
int isPressed(int key) {
    return gameControllerStatus & key;
}
```

See also [edit]

- Mask (computing)
- Bitboard, used in chess and similar games.
- Bit array
- Flag word

External links [edit]

- Explanation from a book
- Description from another wiki

- Use case in a C++ guide
- C++ libbit bit library (alternative URL)

References [edit]

1. ^ Prata, Stephen (2007). *C primer plus* (5th ed. ed.). Indianapolis, Ind: Sams. ISBN 0-672-32696-5.
2. ^ Prata, Stephen (2007). *C primer plus* (5th ed. ed.). Indianapolis, Ind: Sams. ISBN 0-672-32696-5.

Categories: Bit data structures