Search

Keyboard Shortcuts

?
: This help

j
: Next menu item

k
: Previous menu item

g p
: Previous man page

g n
: Next man page

G
: Scroll to bottom

g g
: Scroll to top

g h
: Goto homepage

g s
: Goto search
(current page)

/
: Focus search box

Сравнение »
« Присваивание

- Руководство по PHP
- Справочник языка
- Операторы

Change language: Russian

Побитовые операторы

Побитовые операторы разрешают считывать и устанавливать конкретные биты целых чисел.

Побитовые операторы

Пример	Название	Результат
`$a & $b`	И	Биты, которые установлены и в переменной *$a*, и в переменной *$b*.
`$a | $b`	Или	Будут заданы биты, которые установлены или в переменной *$a*, или в переменной *$b*.
`$a ^ $b`	Исключающее или	Будут заданы биты, которые установлены либо только в переменной *$a*, либо только в переменной *$b*, но не в обоих одновременно.
`~ $a`	Отрицание	Будут заданы биты, которые не установлены в переменной *$a*, и наоборот.
`$a << $b`	Сдвиг влево	Все биты переменной *$a* сдвигаются влево на количествво позиций, указанных в переменной *$b* (каждая позиция предполагает «умножение на 2»)
`$a >> $b`	Сдвиг вправо	Все биты переменной *$a* сдвигаются вправо на количество позиций, указанных в переменной *$b* (каждая позиция предполагает «деление на 2»)

Побитовый сдвиг в PHP — это арифметическая операция. Биты, сдвинутые за границы числа, отбрасываются. Сдвиг влево дополняет число нулями справа, при этом сдвигая знаковый бит числа влево, что означает что знак операнда не сохраняется. Сдвиг вправо сохраняет копию сдвинутого знакового бита слева, что означает что знак операнда сохраняется.

Приоритет операторов изменяют скобками. Например, выражение `$a & $b == true` сначала проверяет на равенство, а потом выполняет побитовое «И»; тогда как выражение `($a & $b) == true` сначала выполняет побитовое «И», а потом выполняет проверку на равенство.

Если оба операнда для операторов `&`, `|` и `^` строки, то операция будет проведена с кодами ASCII всех символов строки и в результате вернёт строку. Во всех остальных случаях, оба операнда будут преобразованы к целому и результатом будет целое число.

Если операнд для оператора `~` строка, то операция будет проведена с кодами ASCII всех символов строки и в результате вернёт строку, иначе как операнд, так и результат, будут считаться целыми.

И операнды, и результат выполнения операторов `<<` и `>>` рассматриваются как целые числа.

```
В PHP ini-настройка error_reporting использует побитовые значения,
показывая, как практически снимать значения битов.
Чтобы показать все ошибки, кроме замечаний,
инструкции в файле php.ini говорят, что нужно указать:
E_ALL & ~E_NOTICE


Начинаем со значения E_ALL:
00000000000000000111011111111111
Затем берём значение E_NOTICE...
00000000000000000000000000001000
... и инвертируем его оператором ~:
11111111111111111111111111110111
Наконец, указываем побитовое И (&), чтобы установить только те биты,
которые установлены в единицу в обоих значениях:
00000000000000000111011111110111


Другой способ достичь этого  – использовать ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR, ^),
чтобы получить только те биты, которые установлены в единицу
либо только в одном, либо только в другом значении:
E_ALL ^ E_NOTICE


Через настройку опции error_reporting можно также показать,
как устанавливать биты. Показать только ошибки и обрабатываемые ошибки можно
так:
E_ERROR | E_RECOVERABLE_ERROR


Здесь процесс сочетает E_ERROR
00000000000000000000000000000001
и
00000000000000000001000000000000
```

```
    через оператор ИЛИ (|),
    чтобы получить биты, установленные хотя бы в одном операнде:
    00000000000000000001000000000001
```

Пример #1 Побитовыми операции И, ИЛИ и ИСКЛЮЧАЮЩЕЕ ИЛИ (AND, OR и XOR) над целыми числами

```php
<?php

/*
* Не обращайте внимания на верхний раздел кода,
* это просто форматирование для более ясного вывода.
*/
$format = '(%1$2d = %1$04b) = (%2$2d = %2$04b)'
. ' %3$s (%4$2d = %4$04b)' . "\n";

echo <<<EOH
---------- ----------- -- ----------
результат значение оп тест
---------- ----------- -- ----------
EOH;


/*
* Вот сами примеры.
*/

$values = array(0, 1, 2, 4, 8);
$test = 1 + 4;

echo "\n Побитовое И (AND) \n";
foreach ($values as $value) {
$result = $value & $test;
printf($format, $result, $value, '&', $test);
}

echo "\n Побитовое (включающее) ИЛИ (OR) \n";
foreach ($values as $value) {
$result = $value | $test;
printf($format, $result, $value, '|', $test);
}

echo "\n Побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR) \n";
foreach ($values as $value) {
$result = $value ^ $test;
printf($format, $result, $value, '^', $test);
}
```

Результат выполнения приведённого примера:

```
 ---------     ---------  -- ---------
 result         value     op test
 ---------     ---------  -- ---------
 Побитовое И
( 0 = 0000) = ( 0 = 0000) & ( 5 = 0101)
( 1 = 0001) = ( 1 = 0001) & ( 5 = 0101)
( 0 = 0000) = ( 2 = 0010) & ( 5 = 0101)
( 4 = 0100) = ( 4 = 0100) & ( 5 = 0101)
( 0 = 0000) = ( 8 = 1000) & ( 5 = 0101)

 Побитовое ИЛИ
( 5 = 0101) = ( 0 = 0000) | ( 5 = 0101)
( 5 = 0101) = ( 1 = 0001) | ( 5 = 0101)
( 7 = 0111) = ( 2 = 0010) | ( 5 = 0101)
( 5 = 0101) = ( 4 = 0100) | ( 5 = 0101)
(13 = 1101) = ( 8 = 1000) | ( 5 = 0101)

 Побитовое исключающее ИЛИ (XOR)
```

```
( 5 = 0101) = ( 0 = 0000) ^ ( 5 = 0101)
( 4 = 0100) = ( 1 = 0001) ^ ( 5 = 0101)
( 7 = 0111) = ( 2 = 0010) ^ ( 5 = 0101)
( 1 = 0001) = ( 4 = 0100) ^ ( 5 = 0101)
(13 = 1101) = ( 8 = 1000) ^ ( 5 = 0101)
```

Пример #2 Побитовая операция ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR) над строками

```php
<?php

echo 12 ^ 9; // Выводит '5'

echo "12" ^ "9"; // Выводит символ Backspace (ascii 8)
// ('1' (ascii 49)) ^ ('9' (ascii 57)) = #8

echo "hallo" ^ "hello"; // Выводит ascii-значения #0 #4 #0 #0 #0
// 'a' ^ 'e' = #4

echo 2 ^ "3"; // Выводит 1
// 2 ^ ((int)"3") == 1

echo "2" ^ 3; // Выводит 1
// ((int)"2") ^ 3 == 1
```

Пример #3 Сдвиг битов в целых числах

```php
<?php

/*
* Несколько примеров.
*/

echo "\n--- СДВИГ ВПРАВО НАД ПОЛОЖИТЕЛЬНЫМИ ЦЕЛЫМИ (НАТУРАЛЬНЫМИ) ЧИСЛАМИ ---\n";

$val = 4;
$places = 1;
$res = $val >> $places;
p($res, $val, '>>', $places, 'слева была вставлена копия знакового бита');

$val = 4;
$places = 2;
$res = $val >> $places;
p($res, $val, '>>', $places);

$val = 4;
$places = 3;
$res = $val >> $places;
p($res, $val, '>>', $places, 'биты были выдвинуты за правый край');

$val = 4;
$places = 4;
$res = $val >> $places;
p($res, $val, '>>', $places, 'то же, что и выше; нельзя сдвинуть дальше 0');

echo "\n--- СДВИГ ВПРАВО НАД ОТРИЦАТЕЛЬНЫМИ ЦЕЛЫМИ ЧИСЛАМИ ---\n";

$val = -4;
$places = 1;
$res = $val >> $places;
p($res, $val, '>>', $places, 'слева была вставлена копия знакового бита');

$val = -4;
$places = 2;
$res = $val >> $places;
```

```php
p($res, $val, '>>', $places, 'биты были выдвинуты за правый край');

$val = -4;
$places = 3;
$res = $val >> $places;
p($res, $val, '>>', $places, 'то же, что и выше; нельзя сдвинуть дальше -1');

echo "\n--- СДВИГ ВЛЕВО НАД ПОЛОЖИТЕЛЬНЫМИ ЦЕЛЫМИ (НАТУРАЛЬНЫМИ) ЧИСЛАМИ ---\n";

$val = 4;
$places = 1;
$res = $val << $places;
p($res, $val, '<<', $places, 'правый край был дополнен нулями');

$val = 4;
$places = (PHP_INT_SIZE * 8) - 4;
$res = $val << $places;
p($res, $val, '<<', $places);

$val = 4;
$places = (PHP_INT_SIZE * 8) - 3;
$res = $val << $places;
p($res, $val, '<<', $places, 'знаковые биты были выдвинуты');

$val = 4;
$places = (PHP_INT_SIZE * 8) - 2;
$res = $val << $places;
p($res, $val, '<<', $places, 'биты были выдвинуты за левый край');

echo "\n--- СДВИГ ВЛЕВО НАД ОТРИЦАТЕЛЬНЫМИ ЦЕЛЫМИ ЧИСЛАМИ ---\n";

$val = -4;
$places = 1;
$res = $val << $places;
p($res, $val, '<<', $places, 'правый край был дополнен нулями');

$val = -4;
$places = (PHP_INT_SIZE * 8) - 3;
$res = $val << $places;
p($res, $val, '<<', $places);

$val = -4;
$places = (PHP_INT_SIZE * 8) - 2;
$res = $val << $places;
p($res, $val, '<<', $places, 'биты были выдвинуты за левый край, включая знаковый бит');

/*
* Не обращайте внимания на этот нижний раздел кода,
* это просто форматирование для более ясного вывода.
*/

function p($res, $val, $op, $places, $note = '') {
$format = '%0' . (PHP_INT_SIZE * 8) . "b\n";

printf("Выражение: %d = %d %s %d\n", $res, $val, $op, $places);

echo " Десятичный вид:\n";
printf(" val=%d\n", $val);
printf(" res=%d\n", $res);

echo " Двоичный вид:\n";
printf(' val=' . $format, $val);
```

```
printf(' res=' . $format, $res);

if ($note) {
echo " ЗАМЕЧАНИЕ: $note\n";
}

echo "\n";
}
?>
```

Результат выполнения приведённого примера на 32-битных машинах:

```
--- СДВИГ ВПРАВО НАД ПОЛОЖИТЕЛЬНЫМИ ЦЕЛЫМИ (НАТУРАЛЬНЫМИ) ЧИСЛАМИ ---
Выражение: 2 = 4 >> 1
 Десятичный вид:
  val=4
  res=2
 Двоичный вид:
  val=00000000000000000000000000000100
  res=00000000000000000000000000000010
 ЗАМЕЧАНИЕ: слева была вставлена копия знакового бита

Выражение: 1 = 4 >> 2
 Десятичный вид:
  val=4
  res=1
 Двоичный вид:
  val=00000000000000000000000000000100
  res=00000000000000000000000000000001

Выражение: 0 = 4 >> 3
 Десятичный вид:
  val=4
  res=0
 Двоичный вид:
  val=00000000000000000000000000000100
  res=00000000000000000000000000000000
 ЗАМЕЧАНИЕ: биты были выдвинуты за правый край

Выражение: 0 = 4 >> 4
 Десятичный вид:
  val=4
  res=0
 Двоичный вид:
  val=00000000000000000000000000000100
  res=00000000000000000000000000000000
 ЗАМЕЧАНИЕ: то же, что и выше; нельзя сдвинуть дальше 0

--- СДВИГ ВПРАВО НА ОТРИЦАТЕЛЬНЫХ ЦЕЛЫХ ЧИСЛАХ ---
Выражение: -2 = -4 >> 1
 Десятичный вид:
  val=-4
  res=-2
 Двоичный вид:
  val=11111111111111111111111111111100
  res=11111111111111111111111111111110
 ЗАМЕЧАНИЕ: слева была вставлена копия знакового бита

Выражение: -1 = -4 >> 2
 Десятичный вид:
  val=-4
  res=-1
 Двоичный вид:
  val=11111111111111111111111111111100
  res=11111111111111111111111111111111
 ЗАМЕЧАНИЕ: биты были выдвинуты за правый край

Выражение: -1 = -4 >> 3
 Десятичный вид:
  val=-4
  res=-1
 Двоичный вид:
  val=11111111111111111111111111111100
  res=11111111111111111111111111111111
 ЗАМЕЧАНИЕ: то же, что и выше; нельзя сдвинуть дальше -1
```

```
--- СДВИГ ВЛЕВО НАД ПОЛОЖИТЕЛЬНЫМИ ЦЕЛЫМИ (НАТУРАЛЬНЫМИ) ЧИСЛАМИ ---
Выражение: 8 = 4 << 1
 Десятичный вид:
  val=4
  res=8
 Двоичный вид:
  val=00000000000000000000000000000100
  res=00000000000000000000000000001000
 ЗАМЕЧАНИЕ: правый край был дополнен нулями

Выражение: 1073741824 = 4 << 28
 Десятичный вид:
  val=4
  res=1073741824
 Двоичный вид:
  val=00000000000000000000000000000100
  res=01000000000000000000000000000000

Выражение: -2147483648 = 4 << 29
 Десятичный вид:
  val=4
  res=-2147483648
 Двоичный вид:
  val=00000000000000000000000000000100
  res=10000000000000000000000000000000
 ЗАМЕЧАНИЕ: знаковые биты были выдвинуты

Выражение: 0 = 4 << 30
 Десятичный вид:
  val=4
  res=0
 Двоичный вид:
  val=00000000000000000000000000000100
  res=00000000000000000000000000000000
 ЗАМЕЧАНИЕ: биты были выдвинуты за левый край

--- СДВИГ ВЛЕВО НАД ОТРИЦАТЕЛЬНЫМИ ЦЕЛЫМИ ЧИСЛАМИ ---
Выражение: -8 = -4 << 1
 Десятичный вид:
  val=-4
  res=-8
 Двоичный вид:
  val=11111111111111111111111111111100
  res=11111111111111111111111111111000
 ЗАМЕЧАНИЕ: правый край был дополнен нулями

Выражение: -2147483648 = -4 << 29
 Десятичный вид:
  val=-4
  res=-2147483648
 Двоичный вид:
  val=11111111111111111111111111111100
  res=10000000000000000000000000000000

Выражение: 0 = -4 << 30
 Десятичный вид:
  val=-4
  res=0
 Двоичный вид:
  val=11111111111111111111111111111100
  res=00000000000000000000000000000000
 ЗАМЕЧАНИЕ: биты были выдвинуты за левый край, включая знаковый бит
```

Результат выполнения приведённого примера на 64-битных машинах:

```
--- СДВИГ ВПРАВО НАД ПОЛОЖИТЕЛЬНЫМИ ЦЕЛЫМИ (НАТУРАЛЬНЫМИ) ЧИСЛАМИ ---
Выражение: 2 = 4 >> 1
 Десятичный вид:
  val=4
  res=2
 Двоичный вид:
  val=0000000000000000000000000000000000000000000000000000000000000100
  res=0000000000000000000000000000000000000000000000000000000000000010
 ЗАМЕЧАНИЕ: слева была вставлена копия знакового бита

Выражение: 1 = 4 >> 2
 Десятичный вид:
```

```
  val=4
  res=1
 Двоичный вид:
  val=00000000000000000000000000000000000000000000000000000000000000100
  res=00000000000000000000000000000000000000000000000000000000000000001

Выражение: 0 = 4 >> 3
 Десятичный вид:
  val=4
  res=0
 Двоичный вид:
  val=00000000000000000000000000000000000000000000000000000000000000100
  res=00000000000000000000000000000000000000000000000000000000000000000
 ЗАМЕЧАНИЕ: биты были выдвинуты за правый край

Выражение: 0 = 4 >> 4
 Десятичный вид:
  val=4
  res=0
 Двоичный вид:
  val=00000000000000000000000000000000000000000000000000000000000000100
  res=00000000000000000000000000000000000000000000000000000000000000000
 ЗАМЕЧАНИЕ: то же, что и выше; нельзя сдвинуть дальше 0

--- СДВИГ ВПРАВО НАД ОТРИЦАТЕЛЬНЫМИ ЦЕЛЫМИ ЧИСЛАМИ ---
Выражение: -2 = -4 >> 1
 Десятичный вид:
  val=-4
  res=-2
 Двоичный вид:
  val=11111111111111111111111111111111111111111111111111111111111111100
  res=11111111111111111111111111111111111111111111111111111111111111110
 ЗАМЕЧАНИЕ: слева была вставлена копия знакового бита

Выражение: -1 = -4 >> 2
 Десятичный вид:
  val=-4
  res=-1
 Двоичный вид:
  val=11111111111111111111111111111111111111111111111111111111111111100
  res=11111111111111111111111111111111111111111111111111111111111111111
 ЗАМЕЧАНИЕ: биты были выдвинуты за правый край

Выражение: -1 = -4 >> 3
 Десятичный вид:
  val=-4
  res=-1
 Двоичный вид:
  val=11111111111111111111111111111111111111111111111111111111111111100
  res=11111111111111111111111111111111111111111111111111111111111111111
 ЗАМЕЧАНИЕ: то же, что и выше; нельзя сдвинуть дальше -1

--- СДВИГ ВЛЕВО НАД ПОЛОЖИТЕЛЬНЫМИ ЦЕЛЫМИ (НАТУРАЛЬНЫМИ) ЧИСЛАМИ ---
Выражение: 8 = 4 << 1
 Десятичный вид:
  val=4
  res=8
 Двоичный вид:
  val=00000000000000000000000000000000000000000000000000000000000000100
  res=00000000000000000000000000000000000000000000000000000000000001000
 ЗАМЕЧАНИЕ: правый край был дополнен нулями

Выражение: 4611686018427387904 = 4 << 60
 Десятичный вид:
  val=4
  res=4611686018427387904
 Двоичный вид:
  val=00000000000000000000000000000000000000000000000000000000000000100
  res=01000000000000000000000000000000000000000000000000000000000000000

Выражение: -9223372036854775808 = 4 << 61
 Десятичный вид:
  val=4
  res=-9223372036854775808
 Двоичный вид:
  val=00000000000000000000000000000000000000000000000000000000000000100
  res=10000000000000000000000000000000000000000000000000000000000000000
 ЗАМЕЧАНИЕ: знаковые биты были выдвинуты
```

```
Выражение: 0 = 4 << 62
 Десятичный вид:
  val=4
  res=0
 Двоичный вид:
  val=0000000000000000000000000000000000000000000000000000000000000100
  res=0000000000000000000000000000000000000000000000000000000000000000
 ЗАМЕЧАНИЕ: биты были выдвинуты за левый край

--- СДВИГ ВЛЕВО НАД ОТРИЦАТЕЛЬНЫМИ ЦЕЛЫМИ ЧИСЛАМИ ---
Выражение: -8 = -4 << 1
 Десятичный вид:
  val=-4
  res=-8
 Двоичный вид:
  val=1111111111111111111111111111111111111111111111111111111111111100
  res=1111111111111111111111111111111111111111111111111111111111111000
 ЗАМЕЧАНИЕ: правый край был дополнен нулями

Выражение: -9223372036854775808 = -4 << 61
 Десятичный вид:
  val=-4
  res=-9223372036854775808
 Двоичный вид:
  val=1111111111111111111111111111111111111111111111111111111111111100
  res=1000000000000000000000000000000000000000000000000000000000000000

Выражение: 0 = -4 << 62
 Десятичный вид:
  val=-4
  res=0
 Двоичный вид:
  val=1111111111111111111111111111111111111111111111111111111111111100
  res=0000000000000000000000000000000000000000000000000000000000000000
 ЗАМЕЧАНИЕ: биты были выдвинуты за левый край, включая знаковый бит
```

Внимание

Для побитовых операций над числами, большими чем значение константы `PHP_INT_MAX`, вызывают функции модуля gmp.

Смотрите также

- pack()
- unpack()
- gmp_and()
- gmp_or()
- gmp_xor()
- gmp_testbit()
- gmp_clrbit()

＋add a note

User Contributed Notes 27 notes

up
down
113
wbcarts at juno dot com ¶
11 years ago

BITWISE FLAGS for Custom PHP Objects

Sometimes I need a custom PHP Object that holds several boolean TRUE or FALSE values. I could easily include a variable for each of them, but as always, code has a way to get unweildy pretty fast. A more intelligent approach always seems to be the answer, even if it seems to be overkill at first.

I start with an abstract base class which will hold a single integer variable called $flags. This simple integer can hold 32 TRUE or FALSE boolean values. Another thing to consider is to just set certain BIT values without disturbing any of the other BITS -- so included in the class definition is the setFlag($flag, $value) function, which will set only the chosen bit. Here's the abstract base class definition:

```
<?php

# BitwiseFlag.php

abstract class BitwiseFlag
{
protected $flags;

/*
* Note: these functions are protected to prevent outside code
* from falsely setting BITS. See how the extending class 'User'
* handles this.
*
*/
protected function isFlagSet($flag)
{
return (($this->flags & $flag) == $flag);
}

protected function setFlag($flag, $value)
{
if($value)
{
$this->flags |= $flag;
}
else
{
$this->flags &= ~$flag;
}
}
}

?>
```

The class above is abstract and cannot be instantiated, so an extension is required. Below is a simple extension called User -- which is severely truncated for clarity. Notice I am defining const variables AND methods to use them.

```
<?php

# User.php

require('BitwiseFlag.php');

class User extends BitwiseFlag
{
const FLAG_REGISTERED = 1; // BIT #1 of $flags has the value 1
const FLAG_ACTIVE = 2; // BIT #2 of $flags has the value 2
const FLAG_MEMBER = 4; // BIT #3 of $flags has the value 4
const FLAG_ADMIN = 8; // BIT #4 of $flags has the value 8

public function isRegistered(){
return $this->isFlagSet(self::FLAG_REGISTERED);
}

public function isActive(){
return $this->isFlagSet(self::FLAG_ACTIVE);
}

public function isMember(){
return $this->isFlagSet(self::FLAG_MEMBER);
}
```

```
public function isAdmin(){
return $this->isFlagSet(self::FLAG_ADMIN);
}

public function setRegistered($value){
$this->setFlag(self::FLAG_REGISTERED, $value);
}

public function setActive($value){
$this->setFlag(self::FLAG_ACTIVE, $value);
}

public function setMember($value){
$this->setFlag(self::FLAG_MEMBER, $value);
}

public function setAdmin($value){
$this->setFlag(self::FLAG_ADMIN, $value);
}

public function __toString(){
return 'User [' .
($this->isRegistered() ? 'REGISTERED' : '') .
($this->isActive() ? ' ACTIVE' : '') .
($this->isMember() ? ' MEMBER' : '') .
($this->isAdmin() ? ' ADMIN' : '') .
']';
}
}

?>
```

This seems like a lot of work, but we have addressed many issues, for example, using and maintaining the code is easy, and the getting and setting of flag values make sense. With the User class, you can now see how easy and intuitive bitwise flag operations become.

```
<?php

require('User.php')

$user = new User();
$user->setRegistered(true);
$user->setActive(true);
$user->setMember(true);
$user->setAdmin(true);

echo $user; // outputs: User [REGISTERED ACTIVE MEMBER ADMIN]

?>
```

up
down
36

grayda dot NOSPAM at DONTSPAM dot solidinc dot org ¶

14 years ago

Initially, I found bitmasking to be a confusing concept and found no use for it. So I've whipped up this code snippet in case anyone else is confused:

```
<?php

// The various details a vehicle can have
```

```
$hasFourWheels = 1;
$hasTwoWheels = 2;
$hasDoors = 4;
$hasRedColour = 8;

$bike = $hasTwoWheels;
$golfBuggy = $hasFourWheels;
$ford = $hasFourWheels | $hasDoors;
$ferrari = $hasFourWheels | $hasDoors | $hasRedColour;

$isBike = $hasFourWheels & $bike; # False, because $bike doens't have four wheels
$isGolfBuggy = $hasFourWheels & $golfBuggy; # True, because $golfBuggy has four wheels
$isFord = $hasFourWheels & $ford; # True, because $ford $hasFourWheels

?>
```

And you can apply this to a lot of things, for example, security:

```
<?php

// Security permissions:
$writePost = 1;
$readPost = 2;
$deletePost = 4;
$addUser = 8;
$deleteUser = 16;

// User groups:
$administrator = $writePost | $readPosts | $deletePosts | $addUser | $deleteUser;
$moderator = $readPost | $deletePost | $deleteUser;
$writer = $writePost | $readPost;
$guest = $readPost;

// function to check for permission
function checkPermission($user, $permission) {
if($user & $permission) {
return true;
} else {
return false;
}
}

// Now we apply all of this!
if(checkPermission($administrator, $deleteUser)) {
deleteUser("Some User"); # This is executed because $administrator can $deleteUser
}

?>
```

Once you get your head around it, it's VERY useful! Just remember to raise each value by the power of two to avoid problems

up
down
15

frankemeks77 at yahoo dot com ¶

11 years ago

Just learning Bitwise Shift Operators.

The easiest way to resolve a bitwise shift operators is my multiply or dividing each step by two for left shift or right shift respectively

Example:

```
LEFT SHIFT
<?php echo 8 << 3; //64 ?>

//same as
<?php echo 8 * 2 * 2 * 2; ?>

RIGHT SHIFT
<?php echo 8 >> 3; //1 ?>

//same as
<?php echo ((8/2)/2)/2; //1 ?>

//Solving on a paper 8/2 = 4/2 = 2/2 = 1
```

up
down
6

m0sh at hotmail dot com ¶

15 years ago

```
@greenone - nice function, thanks. I've adapted it for key usage:

<?php
function bitxor($str, $key) {
$xorWidth = PHP_INT_SIZE*8;
// split
$o1 = str_split($str, $xorWidth);
$o2 = str_split(str_pad('', strlen($str), $key), $xorWidth);
$res = '';
$runs = count($o1);
for($i=0;$i<$runs;$i++)
$res .= decbin(bindec($o1[$i]) ^ bindec($o2[$i]));
return $res;
}
?>
```

up
down
15

S?b. ¶

18 years ago

```
A bitwise operators practical case :

<?php
// We want to know the red, green and blue values of this color :
$color = 0xFEA946 ;

$red = $color >> 16 ;
$green = ($color & 0x00FF00) >> 8 ;
$blue = $color & 0x0000FF ;

printf('Red : %X (%d), Green : %X (%d), Blue : %X (%d)',
$red, $red, $green, $green, $blue, $blue) ;

// Will display...
// Red : FE (254), Green : A9 (169), Blue : 46 (70)
?>
```

up
down
9

zewt at hotmail dot com ¶

17 years ago

```
if you use bitwise you MUST make sure your variables are integers, otherwise you can get incorrect results.
```

I recommend ALWAYS

(int)$var & (int)$var2

This will save you many headaches when troubleshooting a completely illogical result.

up
down
8

zooly at globmi dot com ¶

14 years ago

Here is an example for bitwise leftrotate and rightrotate.

Note that this function works only with decimal numbers - other types can be converted with pack().

```
<?php

function rotate ( $decimal, $bits) {

$binary = decbin($decimal);

return (
bindec(substr($binary, $bits).substr($binary, 0, $bits))
);

}

// Rotate 124 (1111100) to the left with 1 bits

echo rotate(124, 1);

// = 121 (1111001)

// Rotate 124 (1111100) to the right with 3 bits

echo rotate(124, -3);

// = 79 (1001111)

?>
```

up
down
8

Silver ¶

14 years ago

Regarding what Bob said about flags, I'd like to point out there's a 100% safe way of defining flags, which is using hexadecimal notation for integers:

```
<?php
define("f0", 0x1); // 2^0
define("f1", 0x2); // 2^1
define("f2", 0x4); // 2^2
define("f3", 0x8); // 2^3
define("f4", 0x10); // 2^4
define("f5", 0x20); // 2^5
// ...
define("f20", 0x1000000); // 2^20
define("f21", 0x2000000); // 2^21
define("f22", 0x4000000); // 2^22
define("f23", 0x8000000); // 2^23
define("f24", 0x10000000); // 2^24
// ... up to 2^31
?>
```

I always avoid using decimal notation when I have a large amount of different flags, because it's very easy to misspell numbers like 2^20 (1048576).

up
down
7

zlel grxnslxves13 at hotmail dot com~=s/x/ee/g ¶

18 years ago

I refer to Eric Swanson's post on Perl VS PHP's implementation of xor.

Actually, this is not an issue with the implementation of XOR, but a lot more to do with the lose-typing policy that PHP adopts.

Freely switching between int and float is good for most cases, but problems happen when your value is near the word size of your machine. Which is to say, 32-bit machines will encounter problems with values that hover around 0x80000000 - primarily because PHP does not support unsigned integers.

using bindec/decbin would address this issue as a work-around to do unsigned-int xor, but here's the real picture (i'm not claiming that this code will perform better, but this would be a better pedagogical code):

```
<?php

function unsigned_xor32 ($a, $b)
{
$a1 = $a & 0x7FFF0000;
$a2 = $a & 0x0000FFFF;
$a3 = $a & 0x80000000;
$b1 = $b & 0x7FFF0000;
$b2 = $b & 0x0000FFFF;
$b3 = $b & 0x80000000;

$c = ($a3 != $b3) ? 0x80000000 : 0;

return (($a1 ^ $b1) |($a2 ^ $b2)) + $c;
}

$x = 3851235679;
$y = 43814;
echo "<br>This is the value we want";
echo "<br>3851262585";

echo "<br>The result of a native xor operation on integer values is treated as a signed integer";
echo "<br>".($x ^ $y);

echo "<br>We therefore perform the MSB separately";
echo "<br>".unsigned_xor32($x, $y);

?>
```

This is really foundation stuff, but for those of you who missed this in college, there seems to be something on 2's complement here:

http://www.evergreen.edu/biophysics/technotes/program/2s_comp.htm

up
down
2

ASchmidt at Anamera dot net ¶

4 years ago

Setting, unsetting and testing single and multiple bits in a bitmask:

```
<?php
const FLAG_A = 0b0001,
```

```
FLAG_B = 0b0010,
FLAG_C = 0b0100,
FLAG_D = 0b1000;

const COMBO_BC = FLAG_B | FLAG_C;

$bitmask = 0b000;

// Setting individual flags.
$bitmask |= FLAG_B; // Sets FLAG_B (=2)
$bitmask |= FLAG_C; // also sets FLAG_C (=4)

// Testing single or multiple flags.
echo (bool)( $bitmask & FLAG_B ); // True, B is set.

echo (bool)( $bitmask & (FLAG_A | FLAG_B) ); // True, A or B is set.

echo (bool)( $bitmask & FLAG_B and $bitmask & FLAG_C ); // True, B and C are set.
echo (bool)( ( $bitmask & (FLAG_B | FLAG_C) ) ^ (FLAG_B | FLAG_C) ); // False if B and C are set.
echo (bool)( ( $bitmask & COMBO_BC ) ^ COMBO_BC ); // False if B and C are set.

echo (bool)( $bitmask & FLAG_C and $bitmask & FLAG_D ); // False, C and D are NOT BOTH set.
echo (bool)( ( $bitmask & (FLAG_C | FLAG_D) ) ^ (FLAG_C | FLAG_D) ); // True, if C and D are NOT BOTH set.

// Resetting single flag.
$bitmask &= $bitmask ^ FLAG_B; // Unsets B
$bitmask &= $bitmask ^ FLAG_A; // A remains unset.
var_dump( $bitmask ); // Only C still set (=4)

// Resetting multiple flags.
$bitmask &= $bitmask ^ ( FLAG_C | FLAG_D ); // Unsets C and/or D
var_dump( $bitmask ); // No flags set (=0)
```

up
down
4

cw3theophilus at gmail dot com ¶

14 years ago

For those who are looking for a circular bit shift function in PHP (especially useful for cryptographic functions) that works with negtive values, here is a little function I wrote:

(Note: It took me almost a whole day to get this to work with negative $num values (I couldn't figure out why it sometimes worked and other times didn't), because PHP only has an arithmatic and not a logical bitwise right shift like I am used to. I.e. 0x80000001>>16 will ouputs (in binary) "1111 1111 1111 1111 1000 0000 0000 0000" instead of "0000 0000 0000 0000 1000 0000 0000 0000" like you would expect. To fix this you have to apply the mask (by bitwise &) equal to 0x7FFFFFFF right shifted one less than the offset you are shifting by.)

```
<?php
function circular_shift($num,$offset) { //Do a nondestructive circular bitwise shift, if offset positive shift left, if negative shift right
$num=(int)$num;
$mask=0x7fffffff; //Mask to cater for the fact that PHP only does arithmatic right shifts and not logical i.e. PHP doesn't give expected output when right shifting negative values
if ($offset>0) {
$num=($num<<$offset%32) | (($num>>(32-$offset%32)) & ($mask>>(31-$offset%32)));
}
elseif ($offset<0){
$offset=abs($offset);
$num=(($num>>$offset%32) & ($mask>>(-1+$offset%32))) | ($num<<(32-$offset%32));
}
return $num;
}
?>
```

up
down
6
vivekanand dot pathak25 at gmail dot com ¶
10 years ago

```
$a = 9;
$b = 10;
echo $a & $b;

place value 128 64 32 16 8 4 2 1
$a 0 0 0 0 1 0 0 1 =9
$b 0 0 0 0 1 0 1 0 =10

result 8

only bit they share together is the 8 bit. So 8 gets returned.

$a = 36;
$b = 103;
echo $a & $b;

place value 128 64 32 16 8 4 2 1
$a 0 0 1 0 0 1 0 0 =36
$b 0 1 1 0 0 1 1 1 =103

result 32+4 = 36
the only bits these two share together are the bits 32 and 4 which when added together return 36.

$a = 9;
$b = 10;
echo $a | $b;

place value 128 64 32 16 8 4 2 1
$a 0 0 0 0 1 0 0 1 =9
$b 0 0 0 0 1 0 1 0 =10

result 8+2+1 = 11
3 bits set, in the 8, 2, and 1 column.add those up 8+2+1 and you get 11

$a = 9;
$b = 10;
echo $a ^ $b;

place value 128 64 32 16 8 4 2 1
$a 0 0 0 0 1 0 0 1 =9
$b 0 0 0 0 1 0 1 0 =10

result 2+1 = 3
the 2 bit and the 1 bit that they each have set but don't share. Soooo 2+1 = 3
```

up
down
3
icy at digitalitcc dot com ¶
18 years ago

```
Say... you really want to have say... more than 31 bits available to you in your happy bitmask. And you don't want to use floats. So, one solution would to have an array of bitmasks, that are accessed through some kind of interface.

Here is my solution for this: A class to store an array of integers being the bitmasks. It can hold up to 66571993087 bits, and frees up unused bitmasks when there are no bits being stored in them.

<?php
/*
```

```
Infinite* bits and bit handling in general.

*Not infinite, sorry.

Perceivably, the only limit to the bitmask class in storing bits would be
the maximum limit of the index number, on 32 bit integer systems 2^31 - 1,
so 2^31 * 31 - 1 = 66571993087 bits, assuming floats are 64 bit or something.
I'm sure that's enough enough bits for anything.. I hope :D.
*/

DEFINE('INTEGER_LENGTH',31); // Stupid signed bit.

class bitmask
{
protected $bitmask = array();

public function set( $bit ) // Set some bit
{
$key = (int) ($bit / INTEGER_LENGTH);
$bit = (int) fmod($bit,INTEGER_LENGTH);
$this->bitmask[$key] |= 1 << $bit;
}

public function remove( $bit ) // Remove some bit
{
$key = (int) ($bit / INTEGER_LENGTH);
$bit = (int) fmod($bit,INTEGER_LENGTH);
$this->bitmask[$key] &= ~ (1 << $bit);
if(!$this->bitmask[$key])
unset($this->bitmask[$key]);
}

public function toggle( $bit ) // Toggle some bit
{
$key = (int) ($bit / INTEGER_LENGTH);
$bit = (int) fmod($bit,INTEGER_LENGTH);
$this->bitmask[$key] ^= 1 << $bit;
if(!$this->bitmask[$key])
unset($this->bitmask[$key]);
}

public function read( $bit ) // Read some bit
{
$key = (int) ($bit / INTEGER_LENGTH);
$bit = (int) fmod($bit,INTEGER_LENGTH);
return $this->bitmask[$key] & (1 << $bit);
}

public function stringin($string) // Read a string of bits that can be up to the maximum amount of bits long.
{
$this->bitmask = array();
$array = str_split( strrev($string), INTEGER_LENGTH );
foreach( $array as $key => $value )
{
if($value = bindec(strrev($value)))
$this->bitmask[$key] = $value;
}
}

public function stringout() // Print out a string of your nice little bits
{
$string = "";
```

```
$keys = array_keys($this->bitmask);
sort($keys, SORT_NUMERIC);

for($i = array_pop($keys);$i >= 0;$i--)
{
if($this->bitmask[$i])
$string .= sprintf("%0" . INTEGER_LENGTH . "b",$this->bitmask[$i]);
}
return $string;
}

public function clear() // Purge!
{
$this->bitmask = array();
}

public function debug() // See what's going on in your bitmask array
{
var_dump($this->bitmask);
}
}
?>
```

It treats a positive integer input as a bit, so you don't have to deal with the powers of 2 yourself.

```
<?php
$bitmask = new bitmask();

$bitmask->set(8979879); // Whatever

$bitmask->set(888);

if($bitmask->read(888))
print 'Happy!\n';

$bitmask->toggle(39393); // Yadda yadda

$bitmask->remove(888);

$bitmask->debug();

$bitmask->stringin("10010100010100100010101001010101
00000001000001");

print $bitmask->stringout() . "\n";

$bitmask->debug();

$bitmask->clear();

$bitmask->debug();
?>
```

Would produce:

```
Happy!

array(2) {
[289673]=>
int(65536)
[1270]=>
```

```
int(8388608)
}

00000000000000001001010001010010001010100101010100
0000001000001

array(2) {
[0]=>
int(355106881)
[1]=>
int(37970)
}

array(0) {
}
```

up
down
4
ivoras at gmail dot com ¶
12 years ago
As an additional curiosity, for some reason the result of the operation ("18" & "32") is "10". In other words, try avoiding using the binary operators on strings :)

up
down
3
josh at joshstrike dot com ¶
13 years ago
More referencing this for myself than anything... if you need to iterate through every possible binary combination where $n number of flags are set to 1 in a mask of $bits length:

```
<?php
echo masksOf(3,10);

function masksOf($n,$bits) {
$u = pow(2,$bits)-1; //start value, full flags on.
$masks = array();
while ($u>0) {
$z = numflags($u);
if ($z==$n) array_push($masks,$u);
$u--;
}
return ($masks);
}

function numflags($n) {
$k = 0;
while ($n) {
$k += $n & 1;
$n = $n >> 1;
}
return ($k);

// alternately:
// $u = 0;
// for ($k=1;$k<=$n;$k*=2) {
// $u+=($n&$k?1:0);
// }
// return ($u);
}
?>
```

up
down

4

aba at example dot com ¶

12 years ago

It is true that if both the left-hand and right-hand parameters are strings, the bitwise operator will operate on the characters' ASCII values. However, a complement is necessary to complete this sentence.
It is not irrelevant to point out that the decimal character's ASCII value have different binary values.

```
<?php
if (('18' & '32') == '10') {
echo ord('18'); //return decimal value 49, which have binary value 110001
echo ord('32'); //return decimal value 51, which have binary value 110011
echo ord('10'); //return decimal value 49, which have binary value 110001
//Therefore 110001 & 110011 = 110001
}
?>
```

up
down

3

Tbrendstrup ¶

18 years ago

note that the shift operators are arithmetic, not logic like in C. You may get unexpected results with negative numbers, see http://en.wikipedia.org/wiki/Bitwise_operation

here's a function to do logic right shifts.

```
<?php

function lshiftright($var,$amt)
{
$mask = 0x40000000;
if($var < 0)
{
$var &= 0x7FFFFFFF;
$mask = $mask >> ($amt-1);
return ($var >> $amt) | $mask;
}
return $var >> $amt;
}

$val = -10;

printf("arithmetic shift on a negative integer<br>%1\$032b<br>%2\$032b<br>%1\$0d<br>%2\$0d<br>",$val, $val >> 1 );

printf("logic shift on a negative integer<br>%1\$032b<br>%2\$032b<br>%1\$0d<br>%2\$0d<br>",$val, lshiftright($val, 1));

printf("logic shift on a positive integer<br>%1\$032b<br>%2\$032b<br>%1\$0d<br>%2\$0d<br>",-$val, lshiftright(-$val, 1));
?>
```

gives the output:

```
arithmetic shift on a negative integer
11111111111111111111111111110110
11111111111111111111111111111011
-10
-5

logic shift on a negative integer
11111111111111111111111111110110
01111111111111111111111111111011
-10
2147483643
```

```
logic shift on a positive integer
00000000000000000000000000001010
00000000000000000000000000000101
10
5
```

up
down
2

amckenzie4 at gmail dot com ¶

13 years ago

If, like me, you've never thought about how PHP deals with binary, the output of the bitwise NOT may confuse you. For instance, this:

```
$bin = 2;
$notbin = ~$bin;

echo "Bin: " . decbin($bin) . " !bin: " . decbin($notbin) . "\n";
```

returns this:

```
Bin: 10 !bin: 11111111111111111111111111111111111111111111111111111111111111101
```

The reason is that all binary numbers are treated as 32 bits, even if you've manually entered less. In order to get the result I expected (01), it was necessary to AND the result with the number of bits I wanted: in this case, 2 (the number 3, in decimal). Be aware that all return values will have zeros removed from the left until they reach a bit that is set to 1. Continuing the above example, the following:

```
$notbin_2 = ~$bin & '3';
echo "!bin & 3: " . decbin($notbin_2) . "\n";
```

returns this:

```
!bin & 3: 1
```

Note that the actual value was a string of 31 zeros followed by a 1, but the zeros were not shown. This is probably a good thing.

Furthermore, the NOT operator uses two's complement, which means the number you get may be even stranger than you expect: using two's complement means that ~2 == -3. There are plenty of good explanations of two's complement online, so I won't go into that question here.

If what you want is just to reverse a string of bits without any interpretation, you can use a function like this:

```
function bitnot($bin)
{
$not = "";
for ($i = 0; $i < strlen($bin); $i++)
{
if($bin[$i] == 0) { $not .= '1'; }
if($bin[$i] == 1) { $not .= '0'; }
}
return $not;
}
```

It takes a binary string of any length, reverses the bits, and returns the new string. You can then treat it as a binary number, use bindec() to turn it into a decimal, or whatever you want.

I hope this helps someone as much as it would have helped me a week ago!

up
down
4

spencer-p-moy at example dot com ¶

12 years ago

The NOT or complement operator (~) and negative binary numbers can be confusing.

~2 = -3 because you use the formula ~x = -x - 1 The bitwise complement of a decimal number is the negation of the number minus 1.

NOTE: just using 4 bits here for the examples below but in reality PHP uses 32 bits.

Converting a negative decimal number (ie: -3) into binary takes 3 steps:
1) convert the positive version of the decimal number into binary (ie: 3 = 0011)
2) flips the bits (ie: 0011 becomes 1100)
3) add 1 (ie: 1100 + 0001 = 1101)

You might be wondering how does 1101 = -3. Well PHP uses the method "2's complement" to render negative binary numbers. If the left most bit is a 1 then the binary number is negative and you flip the bits and add 1. If it is 0 then it is positive and you don't have to do anything. So 0010 would be a positive 2. If it is 1101, it is negative and you flip the bits to get 0010. Add 1 and you get 0011 which equals -3.

up
down
1

Bob ¶

14 years ago

Here is an easy way to use bitwise operation for 'flag' functionality.
By this I mean managing a set of options which can either be ON or OFF, where zero or more of these options may be set and each option may only be set once. (If you are familiar with MySQL, think 'set' datatype).
Note: to older programmers, this will be obvious.

Here is the code:

```
<?php
function set_bitflag(/*variable-length args*/)
{
$val = 0;
foreach(func_get_args() as $flag) $val = $val | $flag;
return $val;
}
function is_bitflag_set($val, $flag)
{
return (($val & $flag) === $flag);
}
// Define your flags
define('MYFLAGONE', 1); // 0001
define('MYFLAGTWO', 2); // 0010
define('MYFLAGTHREE', 4); // 0100
define('MYFLAGFOUR', 8); // 1000
?>
```

I should point out: your flags are stored in a single integer. You can store loads of flags in a single integer.

To use my functions, say you wanted to set MYFLAGONE and MYFLAGTHREE, you would use:

```
<?php
$myflags = set_bitflags(MYFLAGONE, MYFLAGTHREE);
?>
```

Note: you can pass set_bitflags() as many flags to set as you want.

When you want to test later if a certain flag is set, use e.g.:

```
<?php
if(is_bitflag_set($myflags, MYFLAGTWO))
{
echo "MYFLAGTWO is set!";
}
?>
```

The only tricky part is defining your flags. Here is the process:
1. Write a list of your flags
2. Count them
3. Define the last flag in your list as 1 times 2 to the power of <count> minus one. (I.E. 1*2^(<count>-1))
3. Working backwards through your list, from the last to the first, define each one as half of the previous one. You should reach 1 when you get to the first

If you want to understand binary numbers, bits and bitwise operation better, the wikipedia page explains it well - http://en.wikipedia.org/wiki/Bitwise_operation.

up
down
1

forlamp at msn dot com ¶

16 years ago

two's complement logical operation for 32-bit.

$x must be (int) when passing it to this function to work properly.

```
function comp2($x) // 32bit bitwise complement
{
$mask = 0x80000000;

if ($x < 0)
{
$x &= 0x7FFFFFFF;
$x = ~$x;

return $x ^ $mask;
}
else
{
$x = $x ^ 0x7FFFFFFF;

return $x | $mask;
}
}
```

up
down
2

Eric Swanson ¶

18 years ago

Perl vs. PHP implementation of the ^ operator:

After attempting to translate a Perl module into PHP, I realized that Perl's implementation of the ^ operator is different than the PHP implementation. By default, Perl treats the variables as floats and PHP as integers. I was able to verify the PHP use of the operator by stating "use integer;" within the Perl module, which output the exact same result as PHP was using.

The logical decision would be to cast every variable as (float) when using the ^ operator in PHP. However, this will not yield the same results. After about a half hour of banging my head against the wall, I discovered a gem and wrote a function using the binary-decimal conversions in PHP.

```
/*
not having much experience with bitwise operations, I cannot tell you that this is the BEST solution, but it certainly is a solution that finally works and always returns the EXACT same result Perl provides.
*/
function binxor($a, $b) {
return bindec(decbin((float)$a ^ (float)$b));
}

//normal PHP code will not yeild the same result as Perl
$result = 3851235679 ^ 43814; //= -443704711
```

```
//to get the same result as Perl
$result = binxor(3851235679, 43814); //= 3851262585
//YIPPEE!!!

//to see the differences, try the following
$a = 3851235679 XOR 43814;
$b = 3851235679 ^ 43814; //integer result
$c = (float)3851235679 ^ (float)43814; //same as $b
$d = binxor(3851235679, 43814); //same as Perl!!

echo("A: $a<br />");
echo("B: $b<br />");
echo("C: $c<br />");
echo("D: $d<br />");
```

up
down
2

Adam ¶

13 years ago

Be careful of order of operations.

for example, you may want to check if the second bit is set:

```
<?php
if ($x & 2 == 2) {
/* code */
}
?>
```

is different than

```
<?php
if (($x & 2) == 2) {
/* code */
}
?>
```

and the latter of the two should be used.

up
down
1

erich at seachawaii dot com ¶

11 years ago

Just a note regarding negative shift values, as the documentation states each shift is an integer multiply or divide (left or right respectively) by 2. That means a negative shift value (the right hand operand) effects the sign of the shift and NOT the direction of the shift as I would have expected.

FE. 0xff >> -2 results in 0x0

and 0xff << -2 result in 0xFFFFFFFFC0000000 (dependant on PHP_INT_MAX)

up
down
1

Core Xii ¶

13 years ago

Be very careful when XOR-ing strings! If one of the values is empty (0, '', null) the result will also be empty!

```
<?php
var_dump(1234 ^ 0); // int(1234)
var_dump(1234 ^ ''); // int(1234)
var_dump(1234 ^ null); // int(1234)
var_dump('hello world' ^ 0); // int(0)
var_dump('hello world' ^ ''); // string(0) ""
```

```
var_dump('hello world' ^ null); // int(0)
?>
```

This seems rather inconsistent behavior. An integer XOR'd with zero results the original integer. But a string XOR'd with an empty value results an empty value!

My password hashing function was always returning the same hash... Because I was XOR-ing it with a salt that was sometimes empty!

up
down
1

sag at ich dot net ¶

10 years ago

me reimplement for bitwise NOT (~)

```
protected function flipBin($number) {
$bin = str_pad(base_convert($number, 10, 2), 32, 0, STR_PAD_LEFT);
for ($i = 0; $i < 32; $i++) {
switch ($bin{$i}) {
case '0' :
$bin{$i} = '1';
break;
case '1' :
$bin{$i} = '0';
break;
}
}
return bindec($bin);
}
```

the benefit is, it works with numbers greater MAX_INT

up
down
1

Anonymous ¶

12 years ago

To make very clear why ("18" & "32") is "10".
1) they they are both strings ,
2) "&" operator works on strings by taking each !Character! from each string and make a bit wise & between them and add this value to the resulting string

So:
"18" is made up of two characters: 0x31, 0x38
"32" is made up of two characters: 0x33, 0x32
----RESULT-----
0x31 & 0x33 = 0x31 => "1"
0x38 & 0x32 = 0x30 => "0"

and the result is "10" which is 100% correct.

＋add a note