[3.23] Use single bits as flags to save status information

Sometimes a program needs to save and change the status of several conditions. For example, we may need to keep track of which operations the user has performed. If each of these operations requires a lengthy calculation and the user has already performed the operation, we want to skip the calculation. A typical method to accomplish this record-keeping is to use variables with boolean values *true* and *false*. This is effective if there are only a few status items we need to modify and record, but it becomes unwieldy if there are many items. If we had 30 status items, we would need to create and initialize 30 separate variables. Each boolean variable requires three bytes of RAM storage, so our 30 status items would use 90 bytes.

A variable that is used to indicate the status of a condition is called a *flag*. We can use RAM more efficiently if we save each status flag as a single bit in an integer. Since we will use the *and*, *or*, and *xor* operators to manipulate the flags, we can have, at most, 32 flags in a single integer. If you need more than 32 flags, you can use lists of integers. This method of using a single bit for a flag has these advantages, compared to using a single true/false variable for each status item:

- Less RAM is used, if the program requires many status flags.
- Large groups of flags can be quickly initialized.
- Fewer variable names may be needed, since each group of flags has only one name, and the bits are specified by a number.
- You can combine related status flags into single variables or lists, which makes it easier to manipulate them.
- You can quickly perform status tests on groups of flags with a *mask*.

There are four basic operations we need to perform on flags: set a flag, clear a flag, invert the status of a flag, and test to determine if a flag is set or cleared. While these operations can be performed most efficiently with C or assembly programs, it is possible to code these operations in TIBasic, as these four functions show:

Function	Results
Set a bit: bitset(VarOrList,bit#)	Set bit# in VarOrList to 1
Clear a bit: bitclr(VarOrList,bit#)	Clear bit# in VarOrList to 0
Invert a bit: bitnot(VarOrList,bit#)	Invert bit# in VarOrList: if bit# is 0, set it to 1; if 1, set it to zero
Test a bit: bittst(VarOrList,bit#)	Return *true* if bit# in VarOrList is set, otherwise return *false*.

For all four routines, *VarOrList* is a 32-bit integer variable or constant, or a list of 32-bit variables or constants. *bit#* is the number of the bit to manipulate or test. Bits are numbered starting with zero, so the bits in a 32-bit integer are numbered from zero to 31. Bit number 0 is the least significant bit, and bit number 31 is the most significant bit.

The 89/92+ use a prefix convention to specify 32-bit integers. The two prefixes are '0b' and '0h'. '0b' specifies a binary (base-2) integer consisting of digits 0 and 1. '0h' specifies a hexadecimal (base-16) integer consisting of digits 0 to 9 and A to F. If no prefix is used, the integer is represented as a base-10 number.

To demonstrate the functions, suppose our program needs to keep track of eight status flags, which we store in a variable called *status*. To clear all the flags, use

```
0→status
```

To set all the flags, use

```
0hFF→status
```

Note that I use the *0h* prefix to indicate that *FF* is a hexadecimal (base-16) integer. You could also use

```
255→status
```

since, in binary (base-2) representation, both numbers are 0b11111111.

Suppose that all the flags are cleared, or set to zero. To set bit 0, use

```
bitset(status,0)→status
```

then *status* would be, in binary, 0b00000001.

Next, suppose that *status* is 0b00000111. To clear bit 1, use

```
bitclr(status,1)→status
```

then *status* would be 0b00000101.

We may need to reverse the status of a flag, and *bitnot()* is used to do that. Suppose that *status* is 0b00000000, then

```
bitnot(status,7)→status
```

results in *status* = 0b10000000, and bit 7 has been changed from 0 to 1.

Our program will need to decide what actions to take if certain flags are set or cleared. To perform a block of operations if flag 5 is set, use

```
if bittst(status,5) then
 ... {block} ...
endif
```

Or, to perform a block of operations if flag 7 is cleared, use

```
if not bittst(status,7) then
 ... {block} ...
endif
```

Note that the *not* operator is used to invert the result of *bittst()*.

All the examples shown so far have used a single 32-bit integer to hold the status flags. If you need more than 32 flags, you can use the same functions, but use a list of 32-bit integers instead of a single integer. For example, if we need 96 flags, we would use a list of three integers. The following example shows how to initialize such a list, then perform various operations on the flags in the list.

```
...
© Clear all the flags
{0,0,0}→statl
...
© Set bit 22
bitset(statl,22)→statl
...
© Clear bit 87
bitclr(statl,87)→statl
...
© Execute {block} if bit 17 is set
if bittst(statl,17) then
...{block}...
endif
```

For the purposes of the functions, you can consider the flag list to be a single integer which is 32*n bits, where *n* is the number of list elements. In reality, the flag bits are not numbered consecutively in the list. The bit numbers start at the least significant bit of the first element, and end at the most significant bit of the last element. For our example above, the bit numbers of the three integers are

{ |31|30|29|...|2|1|0|, |63|62|61|...|34|33|32|, |95|94|93|...|66|65|64| }

We can simultaneously test several status flags by using a mask. A mask is simply an integer with certain bits set or cleared, as needed to perform some desired operation. We will use the built-in boolean operators *and*, *or*, *xor* and *not* to perform mask operations. For example, suppose we want to execute a block of code if flags 0, 4 and 7 are all set. The mask to test these flags would be 0b10010001, and the code would look like this:

```
if status and 0b10010001=0b10010001 then
 ... {block} ...
endif
```

The result of and-ing the mask with *status* will only equal the mask if the mask bits are set. This method also provide more flexibility than the corresponding test with separate boolean variables, because the mask can be a variable, which can be changed depending on program requirements.

We can also use masks to perform other manipulations and tests. First, these truth tables show the operation of the built-in boolean operators.

a	b	a and b
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a or b
0	0	0
0	1	1
1	0	1
1	1	1

a	b	a xor b
0	0	0
0	1	1
1	0	1
1	1	0

You can use these truth tables to determine the mask values and boolean operators needed to perform different tests and manipulations, as shown:

Description	Mask bit description	Operation	Example
Set a group of flags	0: the corresponding flag bit is unchanged. 1: The corresponding flag bit is set.	*flags* or *mask*	Set flags 2 and 3: *flags* = 0b0001 *mask* = 0b1100 result = 0b1101
Clear a group of flags	0: the corresponding flag bit is cleared. 1: the corresponding flag bit is unchanged	*flags* and *mask*	Clear flags 0 and 3: *flags* = 0b0111 *mask* = 0b1001 result = 0b0001
Invert a group of flags	0: the corresponding flag bit is unchanged. 1: the corresponding flag bit is inverted	*flags* xor *mask*	Invert flags 1 and 2: *flags* = 0b0011 *mask* = 0b0110 result = 0b0101
Invert all the flags	(none)	not *flags*	*flags* = 0b1010 result = 0b0101
Return *true* if all of a group of flags are set	0: ignore the flag bit in the test. 1: include the flag bit in the test	*flags* and *mask* = *mask*	*flags* = 0b0111 *mask* = 0b0110 *flags* and *mask* = 0b0110 result = true
Return *true* if any of a group of flags are set	0: ignore the flag bit in the test. 1: include the flag bit in the test	*flags* and *mask* ≠ 0	*flags* = 0b0101 *mask* = 0b0110 *flags* and *mask* = 0b0100 result = true

Masks can be used with lists of integers, as well as with individual integers. For example, suppose that *flags* = {0b0110,0b1100} and *mask* = {0b1111,0b1000}, then

flags and *mask* = {0b0110,0b1000}

All the previous examples have shown the bit number argument passed as a number. It may, of course, be a variable instead, and you can improve program readability by choosing appropriate variable names for the flags. As an example, suppose that we have a program which may or may not calculate four sums, three limits and two integrals. We want to keep track of which operations have been done. We need a total of 9 flags, so we can save them all in one integer. At a slight expense in RAM usage, we can assign the flag numbers to variables like this:

```
0→sum1stat   ©Status flag names for sum1,
1→sum2stat   ©...sum2,
2→sum3stat   ©...sum3
3→sum4stat   ©...and sum4
4→lim1stat   ©Status flag names for limit1,
5→lim2stat   ©...limit2
6→lim3stat   ©...and limit 3
7→int1stat   ©Status flag names for integral1
8→int2stat   ©...and integral2
```

Now when we refer to one of the status flags, we use its name instead of the number, for example:

```
bitset(status,lim1stat)→status
```

would set flag 4 in *status*. To test whether or not the second integral has been calculated, we would use

```
if bittst(status,int2stat) then
 ... {block} ...
endif
```

All four of the bit manipulation functions do some simple error checking:

- The flags argument must be an integer or a list of integers. Any other type will return an error message.
- The bit number must be greater than -1, and one less than the total number of flags. If the flags argument is an integer, the bit number must be less than 32.

The same error message is used regardless of which error occurs. The error message is a string of the form "*name*() err", where *name* is the name of the function. Since one error message is used for all errors, there is some ambiguity as to which condition caused the error. This is not a serious flaw, because there are only two possible error causes. As usual, the calling routine can can use *gettype()* to determine the type of the returned result; if it is a string, then an error occurred.

Some possible errors are not detected. If the flags are passed as a list, the type of the list elements are not checked, that is, you must ensure that the list elements are integers. The functions do not determine if the argument is an integer, only that it is a number. Both integers and floating-point numbers have a type of "NUM", so the functons cannot distinguish between them.

A warning message is shown in the status line, when the bit number is 31 (for a single integer flag argument), or is the most significant bit of an integer in a list argument. The warning message is "Warning: operation requires and returns 32 bit value". This warning does not indicate a flaw in the functions, nor does it affect the operation.

The code for the four bit manipulation functions is shown below, and some explanation follows the listings.

Code for *bitset*() - set a bit

```
bitset(x,b)
Func
©(wordOrList,bit#) set bit#
©18may01/dburkett@infinet.com

local k,msg,type
"bitset() err"→msg
gettype(x)→type

when(type="NUM",when(b<0 or b>31,msg,exact(x or 2^b)),when(type="LIST",when(b<0
or b>32*dim(x)-1,msg,augment(left(x,k-1),augment({exact(x[k] or
2^(b-(k-1)*32))},right(x,dim(x)-k)))|k=1+intdiv(b,32)),msg))

EndFunc
```

Code for *bitclr*() - clear a bit

```
bitclr(x,b)
Func
©(wordOrList,bit#) clear bit#
©21may01/dburkett@infinet.com

local k,msg,type

"bitclr() err"→msg
gettype(x)→type

when(type="NUM",when(b<0 or b>31,msg,exact(x and not 2^b)),when(type="LIST",
when(b<0 or b>32*dim(x)-1,msg,augment(left(x,k-1),augment({exact(x[k] and not
2^(b-(k-1)*32))},right(x,dim(x)-k)))|k=1+intdiv(b,32)),msg))
```

```
EndFunc
```

Code for *bitnot*() - invert a bit

```
bitnot(x,b)
Func
©(wordOrList,bit#) invert bit#
©21may01/dburkett@infinet.com

local k,msg,type

"bitnot() err"→msg
gettype(x)→type

when(type="NUM",when(b<0 or b>31,msg,exact(x xor 2^b)),when(type="LIST",when(b<0
or b>32*dim(x)-1,msg,augment(left(x,k-1),augment({exact(x[k] xor
2^(b-(k-1)*32))},right(x,dim(x)-k)))|k=1+intdiv(b,32)),msg))

EndFunc
```

Code for *bittst*() - test bit status

```
bittst(x,b)
Func
©(wordOrList,bit#) test bit#
©22may01/dburkett@infinet.com

local k,msg,type
"bittst() err"→msg
gettype(x)→type

when(type="NUM",when(b<0 or b>31,msg,exact((x and 2^b)≠0)),
when(type="LIST",when(b<0 or b>32*dim(x)-1,msg,exact((x[k] and
2^(b-(k-1)*32))≠0)|k=1+intdiv(b,32)),msg))

EndFunc
```

All of the calculation for each function is performed with a single nested *when()* function: even though the listings above show three lines, each *when()* is a single TIBasic line. If the argument is a single integer, the function tests the bit number argument to ensure that it is between 0 and 31, since an integer has 31 bits. If the flags argument is a list, the functions test to ensure that the bit number argument is greater than zero, and less than or equal to the total number of bits less one, which is found by the expression 32*dim(x)-1.

For each function, a bit mask of the form 2^b is used to perform the required function. The bit mask is all zeros except for a 1 in the bit number argument position. For example, if the bit number is 4, then the bit mask is 0b10000. If the flags argument is a list, the bit mask is in the form $2^{b-(k-1)*32}$, which chooses the correct bit (from 0 to 31) in list element *k*. The list element *k* is found from the expression 1+intdiv(b,32), where *b* is the bit number input argument. In either case, the desired bit is set, cleared or inverted with a simple boolean expression:

bitset() uses x or 2^b
bitclr() uses x and 2^b
bitnot() uses x xor 2^b

where *x* is the flags integer. From the boolean function truth tables shown above, you can verify that these operations result in the desired effect for the different functions. Note that each boolean operation is executed as the argument to *exact()*, which ensure proper results if the mode is set to Auto or Approx when the function is called.

If the flags argument is a list, then *bitset(), bitclr()* and *bitnot()* use a nested *augment()* function to assemble the original list elements, along with the changed element, into the returned list.

bittst() works a little differently, since it must return a *true* or *false* result. *bittst()* uses the boolean expression

```
exact((x and 2^b)≠0)
```

where *x* is the flags integer, and 2^b is the bit mask described above. This expression returns *true* if bit number *b* is set, otherwise it returns *false*.