Manipulating Bits and Bitstrings

Version 2023.3
2024-05-16

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Manipulating Bits and Bitstrings
InterSystems IRIS Data Platform Version 2023.3 2024-05-16

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

List of Tables

Manipulating Bits and Bitstrings

There are times that you may want to store a sequence of related boolean values in an application built on InterSystems IRIS data platform. You could create a number of boolean variables, or you could store them in an array or a list. Or you could use a concept known as a “bitstring,” which can be defined as a sequence of bits, presented least-significant bit first. Bitstrings allow you to store such data in a manner that is very efficient, both in terms of storage space and processing speed.

Bitstrings can be stored in one of two ways, as compressed strings or as integers. If you hear the term “bitstring” used without context, it means that the sequence of bits is stored as a compressed string. This page introduces you to both types of bitstrings and then presents some techniques you can use to manipulate them.

1 Storing a Sequence of Bits as a Bitstring

The most common way of storing a sequence of bits is in a bitstring, which is a special kind of compressed string. In addition to saving storage space, bitstrings can be manipulated efficiently using ObjectScript system functions.

One such system function is **$factor**, which converts an integer into a bitstring. We can turn the integer 11744 into a bitstring by executing the statement:

```
set bitstring = $factor(11744)
```

To see a representation of the contents of a bitstring, you can use the **zwrite** command:

```
zwrite bitstring
bitstring=$zwc(128,4)_$c(224,45,0,0)/*$bit(6..9,11,12,14)*/
```

It looks cryptic at first, but at the end of the output, you will see a comment showing a list of the actual bits that are set: 6, 7, 8, 9, 11, 12, and 14. Bit 1 in a bitstring represents 2^0, bit 2 represents 2^1, and so forth. Adding all of the bits together, we get $2^5 + 2^6 + 2^7 + 2^8 + 2^{10} + 2^{11} + 2^{13} = 11744$.

To get a more pleasing visual representation, you can use another system function, **$bit**:

```
for i=1:1:14 {write $bit(bitstring, i)}
00000111101101
```

In this example, `$bit(bitstring, i)` returns the value of bit `i` of the bitstring.

Note: For more insight into how this sequence of bits is stored internally, take a closer look at the output of the **zwrite** command:

```
bitstring=$zwc(128,4)_$c(224,45,0,0)/*$bit(6..9,11,12,14)*/
```

This bitstring is stored in four chunks of 8 bits each. Evaluating each chunk separately gives you 224, 45, 0, 0.

If it helps you to think of a bitstring as a string, you can think of each chunk as an 8-bit character.

A common application of bitstrings is in the storage of bitmap indexes. A bitmap index is a special type of index that uses a series of bitstrings to represent the set of objects that correspond to a given value for a particular property. Each bit in the bitmap represents an object in the class.

As an example, say you are creating a database of animals and their characteristics and define a class as follows:

```
Class User.Animal Extends %Persistent
{
Property Name As %String [Required];
Property Classification As %String;
Property Diet As %String;
Property Swims As %Boolean;

Index ClassificationIDX On Classification [Type = bitmap];
Index DietIDX On Diet [Type = bitmap];
Index SwimsIDX On Swims [Type = bitmap];
}
```

After populating the database with a few animals, it might look like this:

Table 1: Sample Animal Database

ID	Name	Classification	Diet	Swims
1	Penguin	Bird	Carnivore	1
2	Giraffe	Mammal	Herbivore	0
3	Cheetah	Mammal	Carnivore	0
4	Bullfrog	Amphibian	Carnivore	1
5	Shark	Fish	Carnivore	1
6	Fruit Bat	Mammal	Herbivore	0
7	Snapping Turtle	Reptile	Carnivore	1
8	Manatee	Mammal	Herbivore	1
9	Ant	Insect	Omnivore	0
10	Rattlesnake	Reptile	Carnivore	0

The bitmap index `DietIDX` keeps track of the animals that have a specific value for the Diet property. Each row of the index can store a chunk representing up to 64,000 animals.

Table 2: Bitmap Index for the Diet Property

Diet	Chunk	Bitmap
Carnivore	1	1011101001
Herbivore	1	0100010100
Omnivore	1	0000000010

Internally, the bitmap index is stored in the following nodes of the global `^User.AnimalI`:

```
^User.AnimalI("DietIDX"," CARNIVORE",1)
^User.AnimalI("DietIDX"," HERBIVORE",1)
^User.AnimalI("DietIDX"," OMNIVORE",1)
```

The first subscript is the name of the index (`DietIDX`), the second subscript is that value of the property being indexed (for example, `CARNIVORE`), and the third subscript is the chunk number (`1`, in this example).

Likewise, the bitmap index `SwimsIDX` keeps track of the animals that have a specific value for the Swims property.

Table 3: Bitmap Index for the Swims Property

Swims	Chunk	Bitmap
True	1	1001101100
False	1	0110010011

This bitmap index is stored in the following nodes of `^User.AnimalI`:

```
^User.AnimalI("SwimsIDX",1,1)
^User.AnimalI("SwimsIDX",0,1)
```

To give you an idea of the power of bitstrings, you can count the number of carnivores in the database very easily by counting the number of ones in the bitmap, without having to examine the actual data. Just use the system function **$bitcount**:

```
set c = ^User.AnimalI("DietIDX"," CARNIVORE",1)

write $bitcount(c,1)
6
```

Similarly, you can count the number of animals that swim:

```
set s = ^User.AnimalI("SwimsIDX",1,1)

write $bitcount(s,1)
5
```

To count the number of carnivores that swim, use the **$bitlogic** function to find the intersection of the two sets:

```
set cs = $bitlogic(c&s)

write $bitcount(cs,1)
4
```

Note: Use **zwrite** again to examine how the bitmap for the animals that are carnivores is stored internally:

```
zwrite ^User.AnimalI("DietIDX"," CARNIVORE",1)
^User.AnimalI("DietIDX"," CARNIVORE",1)=$zwc(413,2,0,2,6,8,9)/*$bit(2,4..6,8,11)*/
```

Here, you can see the bits corresponding to all the animals where the Diet property is `CARNIVORE`. As you know, bitmap indexes are broken into chunks of 64,000 bits. The bit stored for an animal with a given ID is stored in chunk (ID\64000) + 1, position (ID#64000) + 1. Thus, the bit representing the animal with ID 1 is stored in chunk 1, position 2. So, in this bitstring, bit 2 represents `Penguin`, not `Giraffe`.

The SQL Engine includes a number of special optimizations that can take advantage of bitmap indexes, so you can reap the benefits whenever you write an SQL query.

For more examples on how to work with bitstrings, see Manipulating Bitstrings.

2 Storing a Sequence of Bits as an Integer

If you want to pass a sequence of boolean arguments to a method, one common method is to pass them as a sequence of bits encoded into a single integer.

For example, the method **Security.System.ExportAll()** is used to export security settings from an InterSystems IRIS instance. If you look at the class reference for this method, you will see that it is defined as follows:

```
classmethod ExportAll(FileName As %String = "SecurityExport.xml",
ByRef NumExported As %String, Flags As %Integer = -1) as %Status
```

The third argument, `Flags`, is an integer, where each bit represents a type of security record that can be exported.

```
Flags - What type of records to export to the file, -1 = ALL
Bit 0 - System
Bit 1 - Events
Bit 2 - Services
Bit 4 - Resources
Bit 5 - Roles
Bit 6 - Users
Bit 7 - Applications
Bit 8 - SSL Configs
Bit 9 - PhoneProvider
Bit 10 - X509Credential
Bit 11 - OpenAMIdentityService
Bit 12 - SQL privileges
Bit 13 - X509Users
Bit 14 - DocDBs
Bit 15 - LDAPConfig
Bit 16 - KMIPServer
```

Bit 0 in a bitstring stored as an integer represents 2^0, bit 1 represents 2^1, and so forth. If you want to export security records of the types corresponding to bits 5, 6, 7, 8, 10, 11, and 13, this is done by setting Flags to $2^5 + 2^6 + 2^7 + 2^8 + 2^{10} + 2^{11} + 2^{13} = 11744$.

In ObjectScript, this might look like:

```
set flags = (2**5) + (2**6) + (2**7) + (2**8) + (2**10) + (2**11) + (2**13)
set status = ##class(Security.System).ExportAll("SecurityExport.xml", .numExported, flags)
```

Some InterSystems APIs have macros defined to make your code easier to read. One such case is in the DataMove utility, where a DataMove object is created using the method **DataMove.Data.CreateFromMapEdits()**. Without getting too far into the details, this method is defined in the class reference as follows:

```
classmethod CreateFromMapEdits(Name As %String, ByRef Properties As %String,
ByRef Warnings As %String, ByRef Errors As %String) as %Status
```

It has the following arguments:

```
Parameters:
Name - A name for the DataMove object to create.
Properties - Array of properties used to create the object. The following properties may
optionally be specified. See the class definition for more information on each property.
Properties("Description") - Description of the Data Move operation, default ="".
Properties("Flags") - Flags describing the operation, default = 0.
Properties("LogFile") - Directory and filename of the logfile, default = \iris\mgr\DataMovename.log.
```

To make `Properties("Flags")` easier to define, these macros are available for your use:

```
Bit Flags to control Data Move.
$$$BitNoSrcJournal - Allow source database to be non-journaled
$$$BitNoWorkerJobs - Don't use 'worker' jobs during the copying of data
$$$BitBatchMode - Run Copy jobs in 'batch' mode
$$$BitCheckActivate - Call $$CheckActivate^ZDATAMOVE() during Activate()
```

These macros are defined as the calculated value for specific bits, allowing you to set the proper bits without having to memorize which bit represents which flag:

```
#;Definitions for DataMove Flags properties
#define BitNoSrcJournal 1
#define BitNoWorkerJobs 512
#define BitBatchMode 2048
#define BitCheckActivate 4096
```

In your code, you might use this code snippet to set the flags and create a DataMove object:

```
// Set properties("Flags") to 6657
set properties("Flags") = $$$BitNoSrcJournal + $$$BitNoWorkerJobs + $$$BitBatchMode + $$$BitCheckActivate
set status = ##class(DataMove.Data).CreateFromMapEdits("dm", .properties, .warnings, .errors)
```

For more examples on how to work with bitstrings-as-integers, see Manipulating Bitstrings Implemented as Integers.

3 Manipulating Bitstrings

3.1 Set Bits

To create a new bitstring, use the **$bit** function to set the desired bits to 1:

```
kill bitstring

set $bit(bitstring, 3) = 1

set $bit(bitstring, 6) = 1

set $bit(bitstring, 11) = 1
```

To set a bit to 1 in an existing bitstring using **$bit**:

```
set $bit(bitstring, 5) = 1
```

To set a bit to 0 in an existing bitstring using **$bit**:

```
set $bit(bitstring, 5) = 0
```

Since the first bit in a bitstring is bit 1, attempting to set bit 0 returns an error:

```
set $bit(bitstring, 0) = 1

SET $BIT(bitstring, 0) = 1
^
<VALUE OUT OF RANGE>
```

3.2 Test If Bits Are Set

To test whether a bit is set in an existing bitstring, also use the **$bit** function:

```
write $bit(bitstring, 6)
1
write $bit(bitstring, 5)
0
```

If you test a bit that is not explicitly set, **$bit** returns 0:

```
write $bit(bitstring, 4)
0
write $bit(bitstring, 55)
0
```

For more information on the **$bit** function, see $BIT.

3.3 Display Bits

To display the bits in a bitstring, use the **$bitcount** function to get a count of the number of bits in the bitstring, and then loop through the bits:

```
for i=1:1:$bitcount(bitstring) {write $bit(bitstring, i)}
00100100001
```

You can also use **$bitcount** to count the number of ones or zeroes in a bitstring:

```
write $bitcount(bitstring, 1)
3
write $bitcount(bitstring, 0)
8
```

For more information on the **$bitcount** function, see $BITCOUNT.

3.4 Find Set Bits

To find which bits are set in a bitstring, use the **$bitfind** function, which returns position of the next bit of a specified value, starting at a given position in the bitstring:

```
Class User.BitStr
{

ClassMethod FindSetBits(bitstring As %String)
{
   set bit = 0
   for {
      set bit = $bitfind(bitstring, 1, bit)
      quit:'bit
      write bit, " "
      set bit = bit + 1
   }
}

}
```

This method searches through the string and quits if **$bitfind** returns 0, indicating no more matches were found.

```
do ##class(User.BitStr).FindSetBits(bitstring)
3 6 11
```

Be very careful when you are testing bitstrings for equivalence.

For example, you could have two bitstrings `b1` and `b2`, which have the same bits set:

```
do ##class(User.BitStr).FindSetBits(b1)
3 6 11
do ##class(User.BitStr).FindSetBits(b2)
3 6 11
```

Yet if you compare them, you see that they, in fact, are not equal:

```
write b1 = b2
0
```

If you use **zwrite**, you can see that the internal representation of these two bitsrings is different:

```
zwrite b1
b1=$zwc(405,2,2,5,10)/*$bit(3,6,11)*/

zwrite b2
b2=$zwc(404,2,2,5,10)/*$bit(3,6,11)*/
```

In this case, `b2` has bit 12 set to `0`:

```
for i=1:1:$bitcount(b1) {write $bit(b1, i)}
00100100001
USER>for i=1:1:$bitcount(b2) {write $bit(b2, i)}
001001000010
```

Furthermore, there may be other internal representations, as well, such as the one created by **$factor**, which turns an integer into a bitstring:

```
set b3 = $factor(1060)

zwrite b3
b3=$zwc(128,4)_$c(36,4,0,0)/*$bit(3,6,11)*/

for i=1:1:$bitcount(b3) {write $bit(b3, i)}
00100100001000000000000000000000
```

To compare two bitstrings that may have different internal representations, you could use a the **$bitlogic** function to directly compare the set bits, as described in Perform Bitwise Arithmetic.

For more information on the **$bitfind** function, see $BITFIND.

3.5 Perform Bitwise Arithmetic

Use the **$bitlogic** function to perform bitwise logical operations on bitstrings.

In this example, assume you have two bitstrings `a` and `b`.

```
for i=1:1:$bitcount(a) {write $bit(a, i)}
100110111
for i=1:1:$bitcount(b) {write $bit(b, i)}
001000101
```

Use the **$bitlogic** function to perform a logical OR across the bits:

```
set c = $bitlogic(a|b)

for i=1:1:$bitcount(c) {write $bit(c, i)}
101110111
```

Use the **$bitlogic** function to perform a logical AND across the bits:

```
set d = $bitlogic(a&b)

for i=1:1:$bitcount(d) {write $bit(d, i)}
000000101
```

This example shows how you can use the **$bitlogic** function to perform a logical XOR to test if two bitstrings `b1` and `b3` have the same set bits, regardless of internal representation:

```
zwrite b1
b1=$zwc(405,2,2,5,10)/*$bit(3,6,11)*/

zwrite b3
b3=$zwc(128,4)_$c(36,4,0,0)/*$bit(3,6,11)*/

write $bitcount($bitlogic(b1^b3),1)
0
```

Logical XOR can quickly show that there are no differences in the set bits of the two bitstrings.

For more information on bitwise logical operations on bitstrings, see $BITLOGIC.

3.6 Convert to a Bitstring-as-Integer

To convert a regular bitstring to a bitstring stored as an integer, use the **$bitfind** function to find the set bits and sum up their powers of two. Remember to divide the result by two to shift the bits to the left because bit 1 in a regular bitstring corresponds to bit 0 in a bitstring-as-integer.

```
ClassMethod BitstringToInt(bitstring As %String)
{
   set bitint = 0
   set bit = 0
   for {
      set bit = $bitfind(bitstring, 1, bit)
      quit:'bit
      set bitint = bitint + (2**bit)
      set bit = bit + 1
   }
   return bitint/2
}
```

Convert `bitstring` to a bitstring-as-integer:

```
for i=1:1:$bitcount(bitstring) {write $bit(bitstring, i)}
00100100001
set bitint = ##class(User.BitStr).BitstringToInt(bitstring)

write bitint
1060
```

4 Manipulating Bitstrings Implemented as Integers

4.1 Set Bits

To create a new bitstring stored as an integer, sum up the powers of 2 for each bit:

```
set bitint = (2**2) + (2**5) + (2**10)

write bitint
1060
```

To set a bit to 1 in an existing bitstring-as-integer, use option 7 (arg1 ! arg2) of the **$zboolean** function (logical OR):

```
set bitint = $zboolean(bitint, 2**4, 7)

write bitint
1076
```

To set a bit to 0 in an existing bitstring-as-integer, use option 2 (arg1 & ~arg2) of the **$zboolean** function:

```
set bitint = $zboolean(bitint, 2**4, 2)

write bitint
1060
```

To toggle a bit in an existing bitstring-as-integer, use option 6 (arg1 ^ arg2) of the **$zboolean** function (logical XOR):

```
set bitint = $zboolean(bitint, 2**4, 6)

write bitint
1076
set bitint = $zboolean(bitint, 2**4, 6)

write bitint
1060
```

4.2 Test If Bits Are Set

To test whether a bit it set in an existing bitstring-as-integer, use option 1 (arg1 & arg2) of the **$zboolean** function (logical AND):

```
write $zboolean(bitint, 2**5, 1)
32
write $zboolean(bitint, 2**4, 1)
0
```

Bit 5 is set, so **$zboolean** returns the value of that bit.

Bit 4 is not set, so **$zboolean** returns 0.

4.3 Display Bits

To display the bits in a bitstring-as-integer, you can use a method like the one below, which loops over the bits and uses the **$zboolean** function:

```
Class User.BitInt{
{
ClassMethod LogicalToDisplay(bitint as %Integer)
{
   for i = 0:1 {
      quit:((2**i) > bitint)
      if $zboolean(bitint, 2**i, 1) {write 1} else {write 0}
   }
}
}
```

Running this method on `bitint` gives the following result:

```
do ##class(User.BitInt).LogicalToDisplay(bitint)
00100100001
```

4.4 Find Set Bits

This method finds which bits are set in a bitstring-as-integer using the **$zlog** function, which returns the base 10 logarithm value. The method lops off smaller and smaller chunks of the bitstring until none are left:

```
ClassMethod FindSetBits(bitint as %Integer)
{
   set bits = ""
   while (bitint '= 0) {
      set bit = $zlog(bitint) \ $zlog(2)
      set bits = bit _ " " _ bits
      set bitint = bitint - (2**bit)
   }
   write bits
}
```

Running this method on `bitint` gives the following result:

```
do ##class(User.BitInt).FindSetBits(bitint)
2 5 10
```

4.5 Perform Bitwise Arithmetic

Use the **$zboolean** function to perform bitwise logical operations on bitstrings stored as integers.

For this example, assume you have two bitstrings a and b, stored as integers, and a **LogicalToDisplay()** method, as defined in Display Bits, to display the bits.

```
do ##class(User.BitInt).LogicalToDisplay(a)
100110111
do ##class(User.BitInt).LogicalToDisplay(b)
001000101
```

Use option 7 of the **$zboolean** function to perform a logical OR across the bits:

```
set c = $zboolean(a, b, 7)

do ##class(User.BitInt).LogicalToDisplay(c)
101110111
```

Use option 1 of the **$zboolean** function to perform a logical AND across the bits:

```
set d = $zboolean(a, b, 1)

do ##class(User.BitInt).LogicalToDisplay(d)
000000101
```

For more information on bitwise logical operations, see $ZBOOLEAN.

4.6 Convert to a Regular Bitstring

To convert a bitstring stored as an integer to a regular bitstring, use the **$factor** function. For this example, assume you have a bitstring-as-integer `bitint` and a **FindSetBits()** method, as defined in Find Set Bits, to show which bits are set.

```
do ##class(User.BitInt).FindSetBits(bitint)
2 5 10
set bitstring = $factor(bitint)

zwrite bitstring
bitstring=$zwc(128,4)_$c(36,4,0,0)/*$bit(3,6,11)*/
```

Note that the bits in the regular bitstring appear shifted to the right by one because bitstrings do not have a bit 0. The first bit in a bitstring is bit 1.