The `Bitstream` C++ library

John P. Costella

3 Sabo Place, Mount Martha, Victoria 3934, Australia
jpcostella@hotmail.com; assassinationscience.com/johncostella

(April 25, 2008)

Abstract

This document describes the `Bitstream` C++ library, its programming interface, and the accompanying unit test program.

1. Introduction

The `Bitstream` C++ library is a thin wrapper around the `iostream` library of the C++ Standard Template Library (STL) that provides convenient functionality for bitwise stream operations.

The `Bitstream` library is one of the core libraries in the new JPEG-Clear image format (`assassinationscience.com/johncostella/jpegclear`). As those libraries are converted from C to C++, I am making them individually available, in case they are of general usefulness to other developers.

The `Bitstream` library requires nothing more than standard C++ (ISO C++98), and is made available under the "MIT" license at the end of this document, so that it can be used for any purpose.

The following files comprise the `Bitstream` library package:

```
Costella/Bitstream.h
Costella/Bitstream.imp.h
Bitstream.Test.cpp
Bitstream.Test.h
Makefile
```

Sec. 2 of this document describes in detail the unit test program for the library, which gives the quickest introduction to its functionality and interface. Sec. 3 documents the interface in detail. Finally, Sec. 4 contains the copyright notice.

2. Unit test program

This section describes the unit test program, `Bitstream.Test`, included within the library package.

2.1. Building `Bitstream.Test`

To build `Bitstream.Test`, you should simply need to run `make` with the supplied `Makefile`:

```
$ make
g++ -Wall -pedantic -O3   -c -o Bitstream.Test.o Bitstream.Test.cpp
g++ -Wall   Bitstream.Test.o   -o Bitstream.Test.
```

Running the resulting program should yield the output

```
$ Bitstream.Test
All tests successful!
```

If either of these fails to work as advertized, then please contact me directly.

2.2. Detailed analysis of `Bitstream.Test`

Let us now examine the code of `Bitstream.Test` in detail.

Note that `Bitstream.Test.h` contains

```
using namespace std;
using namespace Costella::Bitstream;
```

for convenience, which hides the explicit namespace requirements in the following.

To perform the tests, all of `main()` is contained within a `try` block, with a simple `catch` block that reports the first error encountered (if any) and exits. Method calls that are *supposed* to fail are contained within their own `try` blocks, which throw an exception to the outer `catch` block if an exception is *not* thrown by the said method.

The program uses a `std::stringstream` (memory-resident stream) for convenience:

```
stringstream ss;
```

This class acts as both a `std::ostream` and a `std::istream`.

The program contains two scoping blocks. In the first, a `Bitstream::Out` is constructed, with `ss` specified as the `std::ostream`:

```
Out<> out( ss );
```

The `<>` tells the templated class `Out` to use the default type for keeping track of the bit position within the bitstream, which is `int`. This can be changed if your requirements are heavier. For example, if `int`s are 32 bits on your compiler, and you want to support bitstreams more than 256 MiB in length, then you need to specify a larger type. For example, if `long`s are 64 bits, then you can specify

```
Out< long > out( ss );
```

(or, if you really needed to squeeze out that 64th bit, `unsigned long`).

The program then starts by writing out three Boolean flags to `out`—not just to test the `Out::boolean()` method, but also to ensure that the bitstream is not aligned with a byte boundary for the test to follow:

```
out.boolean( true );
out.boolean( false );
out.boolean( -2 );
```

After these method calls, the bitstream should contain the bits

```
101x:xxxx
```

where bits will be represented in nybbles and bytes in the usual manner. (The third method call tests that a nonzero value is taken to be `true`; it has been chosen to be an even number to provide assurance that `boolean()` is *not* simply writing the least significant bit of the value.)

The program then sets up a `std::vector` of four bytes:

```
vector< unsigned char > vec;
vec.push_back( 0xfe );
vec.push_back( 0xdc );
vec.push_back( 0x5b );
vec.push_back( 0xa3 );
```

which in binary is

```
1111:1110 1101:1100 0101:1011 1010:0011
```

It then writes out the first 27 bits of this vector of bytes:

```
out.bits( vec, 27 );
```

where "first" is taken from the most significant bit of each byte, *i.e.*, it is writing the 27 bits

```
1111:1110 1101:1100 0101:1011 101
```

The bitstream should now contain, in total,

```
1011:1111 1101:1011 1000:1011 0111:01xx
```

or

```
bf db 8b 0111:01xx
```

where complete bytes will be written in hex rather than binary.

The program next tries to write out a negative number of bits, which should fail:

```
out.bits( vec, -1 );
```

It then clears the vector, and writes out no bits from the empty vector, which is legal:

```
vec.clear();
out.bits( vec, 0 );
```

It then tries to write one bit from the empty vector, which should fail:

```
out.bits( vec, 1 );
```

It then puts the byte `0xd3` into the vector, and writes it out:

```
vec.push_back( 0xd3 );
```

It then tries to write nine bits from the vector, which should fail:

```
out.bits( vec, 9 );
```

The program now begins its tests of `Out::fixed()`, which writes out a specified fixed number of least significant bits from any non-negative integer value. It first writes out the bottom eleven bits of `0xa7b` = `1010:0111:1011`, namely, `010:0111:1011` = `0x37b`:

```
out.fixed( 0xa7b, 11 );
```

It then writes out no bits from the value `0`, which is legal:

```
out.fixed( 0, 0 );
```

It then tries to write a negative number of bits, which should fail:

```
out.fixed( 0, -1 );
```

and one bit from a negative value, which should also fail:

```
out.fixed( -1, 1 );
```

It then tries to write out nine bits from a byte, which should likewise fail:

```
out.fixed( static_cast< unsigned char >( 0 ), 9 );
```

(Note that the type of the argument is crucial here: the call

```
out.fixed( 0, 9 );
```

would *not* fail, because the value `0` defaults to being an `int`, which is guaranteed to have more than nine bits available for positive values.) The program then tries to write eight bits from a signed byte, which should also fail:

```
out.fixed( static_cast< signed char >( 0 ), 8 );
```

The program now tests the method `Out::position()`, which returns the current bit position in the output bitstream. At this stage of the proceedings, 49 bits have been written out:

```
if( out.position() != 49 )  // ... throw exception
```

It then tests the method `Out::flush()`, which flushes out any bits that are "waiting" to be written to the `std::ostream`, *i.e.*, from zero to seven bits that do not, as yet, form a complete byte. It does this by padding out the least significant bits of the byte with zero bits, and writing it out. Thus, in this example, a flush should bring us up to position 56; moreover, a second call to `flush()` should do nothing:

```
out.flush();
out.flush();
if( out.position() != 56 )  // ... throw exception
```

The program now tests the method `Out::variable()`, which allows positive integers to be written out with a variable number of bits, such that small values require fewer bits. The strategy employed is basically as follows: write out the bit-width of the integer in question, followed by the bits of the integer in question. Actually, since the method requires the integer to be strictly positive, then the leading '1' bit need never be written; and indeed the bit-width will always be at least 1, so in the first step we need simply write out *one less* than the actual bit-width. The only catch is that we still need to know how many bits will be required for this first write, *i.e.*, the writing out of "one less than the bit-width" is (internally) carried out using `Out::fixed()`. (There are many ways that one could in turn encode this value, but in practice they are less efficient than making this decision up front, as will soon become clear.)

The name of this argument is `ldMaxWidth`, as it turns out that it is simply the logarithm to base 2 of the maximum bit-width that one wishes to accommodate. For example, to allow values from 1 to 256 (*i.e.*, the maximum bit-width is 8 bits), use `ldMaxWidth` $= \log_2 8 = 3$; for 16-bit values, use `ldMaxWidth` $= \log_2 16 = 4$; for 32-bit values, use `ldMaxWidth` $= \log_2 32 = 5$; for 64-bit values, use `ldMaxWidth` $= \log_2 64 = 6$; and so on. The practical significance of `ldMaxWidth` is that each and every value written out by `Out::variable()` requires at least `ldMaxWidth` bits; the number of additional bits required for a given value is one less than the actual bit-width of the value. Thus `ldMaxWidth` is effectively a "fixed cost" per value; but each increment in its value squares the maximum value that can be accommodated. In practice, there are very few values in any application that are truly unbounded in principle (for example, the dimensions of an image, for the specific example of JPEG-Clear); once these global parameters are set, all internal values will usually have natural bounds (for example, run-length encoding on the pixels of an image cannot have a run length that it greater than the total number of pixels in the image).

Returning to `Bitstream.Test`, the program first tries to write out a value of zero (with `ldMaxWidth` $= 4$), which should fail:

```
out.variable( 0, 4 );
```

and a negative value, which should likewise fail:

```
out.variable( -1, 4 );
```

as should a negative `ldMaxWidth`:

```
out.variable( 1, -1 );
```

The program then tries the legal but useless case of using `Out::variable()` to write a value of 1 with `ldMaxWidth` $= 0$. This is useless because `ldMaxWidth` $= 0$ implies that the value is only one bit wide; but, since the value zero is forbidden for `Out::variable()`, this means that the value must be precisely 1. However, despite being "useless" (in the sense of having no information content at all), it is very possible that this case may be required in a real-world application, if it were only determined at run-time that a positive quantity

could only have the value 1. It is therefore heartening to see that this singular case doesn't require that any bits be written at all: since `ldMaxWidth` = 0, no bits are written out for the bit-width; and since the leading '1' bit is known in advance to be the *only* bit, there is nothing left to write out of the value either.

In fact, the return value of the method `Out::variable()` is the number of bits required for the value written. The program therefore checks that the return value is zero in this case:

```
if( out.variable( 1, 0 ) )  // ... throw exception
```

It then tries to write out a value of 2 with `ldMaxWidth` = 0:

```
out.variable( 2, 0 );
```

This should fail, because for `ldMaxWidth` = 0 the maximum value that can be accommodated is, as we have just seen, 1.

The program now writes out a value of 1, but this time with `ldMaxWidth` = 1. This is no longer trivial, because `ldMaxWidth` = 1 can accommodate any positive dibit (two-bit value), namely, the values 1, 2, and 3. Now, in the case of the value 1, the method determines that the bit-width of 1 is 1; one less than this is 0. Therefore, this is the one-bit (*i.e.*, in general, `ldMaxWidth`-bit) value that is first written out. Since there are no more bits of the value itself that need to be written out, the total number of bits required is just this one bit:

```
if( out.variable( 1, 1 ) != 1 )  // ... throw exception
```

Of course, this is a general property: to write out the value 1 for any `ldMaxWidth`, the method need simply write out the value 0 with a bit-width of `ldMaxWidth`; there are then no more bits of the value that need to be written out, so the total number of bits required is just `ldMaxWidth`.

The program now writes out the remaining possible values 2 and 3 for `ldMaxWidth` = 1. Each of these require two bits in total: they are encoded as `10` and `11` respectively:

```
if( out.variable( 2, 1 ) != 2 )  // ... throw exception
if( out.variable( 3, 1 ) != 2 )  // ... throw exception
```

It then tries to write out a value of 2 with `ldMaxWidth` = 0, which should fail:

```
out.variable( 4, 1 );
```

The program now writes out the value 1 with `ldMaxWidth` = 2, which, by the above, requires two bits (being encoded as `00`):

```
if( out.variable( 1, 2 ) != 2 )  // ... throw exception
```

then the values 2, 3, and 4 with `ldMaxWidth` = 2 (encoded as `010`, `011`, and `1000` respectively):

```
if( out.variable( 2, 2 ) != 3 )  // ... throw exception
if( out.variable( 3, 2 ) != 3 )  // ... throw exception
if( out.variable( 4, 2 ) != 4 )  // ... throw exception
```

It then moves to the corner-case of writing out the value of `0xf` with `ldMaxWidth` = 2, which should take five bits (being encoded as `11111`):

```
if( out.variable( 0xf, 2 ) != 5 )  // ... throw exception
```

It then tries writing `0x10` with `ldMaxWidth` = 2, which should fail (since `ldMaxWidth` = 2 specifies a nybble):

```
out.variable( 0x10, 2 );
```

It then moves on to the value 1 with `ldMaxWidth` = 3, which should require three bits:

```
if( out.variable( 1, 3 ) != 3 )  // ... throw exception
```

and the maximum byte value `0xff` with `ldMaxWidth` = 3, which should require ten bits (encoded as `111` followed by the seven bits `1111111`):

```
if( out.variable( 0xff, 3 ) != 10 )  // ... throw exception
```

Again, a value of `0x100` with `ldMaxWidth` = 3, should fail (since `ldMaxWidth` = 3 specifies a byte):

```
out.variable( 0x100, 3 );
```

Finally, the program tests 16-bit, 32-bit, and 64-bit integers using essentially randomly-chosen values:

```
if( out.variable( 0x37b5, 4 ) != 17 )  // ... throw exception
if( out.variable( 0x005e37b5, 5 ) != 27 )  // ... throw exception
if( out.variable( 0xdef5389a582c190f, 6 ) != 69 )  // ... throw exception
```

and checks that the final bit position is 204:

```
if( out.position() != 204 )  // ... throw exception
```

The program now closes the scoping block for the object `out`, which flushes out the final four bits in its destructor. It then opens a new scoping block and creates an input bitstream, using the same `std::stringstream` as its `std::istream`:

```
In<> in( ss );
```

It then reads in all the values written by `out` to `ss`, using the corresponding methods, and testing those corner cases that are applicable, with the following two exceptions:

Firstly, after calling `In::fixed()` to read in the eleven-bit value `0x27b`, the program then calls `In::unread()` to "unread" the last seven bits:

```
in.unread( 7 );
```

Up to eight bits can be unread using `unread()`; this can provide a useful improvement in speed and efficiency when it is not known in advance how many bits will be required (such as when reading in a Huffman code), by allowing the input to be processed a byte at a time, rather than a bit at a time, where any "overshoot" is returned to the input bitstream. In the case of `Bitstream.Test`, these seven bits need to be re-read, before progressing to the next test:

```
in.fixed( i, 7 );
```

Secondly, instead of reading in the value of `0xff` with `ldMaxWidth` = 3, the program instead calls `skip()` to skip the ten bits involved, to test the `skip()` method:

```
in.skip( 10 );
```

Finally, if all the tests pass without throwing an exception, the program reports this fact to standard output:

```
cout << "All tests successful!" << endl;
```

3. Library interface

This section describes in detail the interface to the `Bitstream` library.

3.1. The `Bitstream` library

This library provides a thin wrapper around the `iostream` library of the C++ STL that allows streams of bits (rather than bytes or larger objects) to be handled conveniently.

Dependencies

The `Bitstream` library does not depend on any other library.

Namespace

All classes of the `Bitstream` library are contained within the `Bitstream` namespace, which is itself contained within the `Costella` namespace.

Interface classes

The `Bitstream` library has the following interface classes:

`Out< PositionType >()`

This class wraps the `ostream` class of the STL. Its class declaration is

```
template< typename PositionType = int >
class Out
```

`PositionType`: The integral type to be used for measuring bit positions within the output bitstream. By default `int` is used, but there are many situations in which the user may wish to select a more appropriate type. For example, if bitstreams longer than 256 MiB are to be supported, then 32-bit `int`s are insufficient; a suitable 64-bit type can instead be specified. The `Out` class throws an exception if a write operation is attempted that would overflow the bit position counter.

`In< PositionType >()`

This class wraps the `istream` class of the STL. Its class declaration is

```
template< typename PositionType = int >
class In
```

`PositionType`: The integral type to be used for measuring bit positions. The `In` class throws an exception if a read operation is attempted that would overflow the bit position counter.

Interface methods

The `Bitstream` library has the following interface methods:

`Out::Out()`

This constructor has prototype

```
Out( std::ostream& outstream = std::cout );
```

`outstream`: The STL `ostream` to which the bitstream will be written. The default stream is `cout`.

Note that the bitstream is written to `outstream` one byte at a time.

`Out::~Out()`

This destructor has prototype

```
~Out();
```

Before the object is destroyed, any unwritten bits are flushed out, *i.e.*, padded out to a byte using zero bits for the appropriate number of least significant bits, and the byte written out to the `std::ostream` specified in the constructor.

`Out::bits()`

This method writes an arbitrary number of bits from a `std::vector` of bytes, starting with the most signficant bit of the first byte. (*I.e.*, if the number of bits is not a multiple of eight, then the final bits are taken from the most significant end of the last byte used.) Its prototype is

```
template< typename NumBitsType >
void bits( const std::vector< unsigned char >& data, NumBitsType
    numBits );
```

`NumBitsType`: The type of `numBits`.

`data`: Vector of bytes containing the bits to be written.

`numBits`: The number of bits to write. If this value is negative, or greater than eight times the size of the vector of bytes, then the method throws an exception.

`Out::fixed()`

This method writes a fixed bit-width non-negative integral value. Its prototype is

```
template< typename ValueType >
void fixed( ValueType value, int width );
```

`ValueType`: The type of `value`.

`value`: The non-negative integral value whose least significant bits are to be written. If `value` is negative, the method throws an exception.

`width`: The bit-width desired. Note that this can be less than or greater than the true bit-width of `value`, but it cannot be negative nor wider than the maximum possible value of type `ValueType`; in either case, the method throws an exception.

`Out::variable()`

This method writes out a positive integral value using a variable number of bits, such that smaller values require fewer bits. It is useful when the dynamic range of possible values is large, but bit efficiency for small values is important; it is essentially optimal if the *logarithm* of the value is relatively uniformly distributed between 0 and some (natural or arbitrarily imposed) upper bound. Its prototype is

```
template< typename ValueType >
int variable( ValueType value, int ldMaxWidth );
```

`ValueType`: The type of `value`.

`value`: The positive integral value to be written. If `value` is zero or negative, the method throws an exception.

`ldMaxWidth`: The logarithm to base 2 of the maximum bit-width possible, rounded up, as shown in Table 1. If `ldMaxWidth` is negative or greater than 8, the method throws an exception.

Return value: The number of bits required to write the value, equal to `ldMaxWidth` plus one less than the actual bit-width of `value`.

If `value` is larger than the maximum value specified by Table 1 for the given value of `ldMaxWidth`, then the method throws an exception.

`Out::boolean()`

This method writes out a Boolean flag, *i.e.*, a single bit. Its prototype is

```
void boolean( bool b );
```

ldMaxWidth	Bits	Range of possible values	comment
0	1	[0x1, 0x1]	useless
1	2	[0x1, 0x3]	dibit
2	4	[0x1, 0xf]	nybble
3	8	[0x1, 0xff]	byte
4	16	[0x1, 0xffff]	word
5	32	[0x1, 0xffff:ffff]	doubleword
6	64	[0x1, 0xffff:ffff:ffff:ffff]	quadword
7	128	$[1, 2^{128}-1]$	octword
8	256	$[1, 2^{256}-1]$	hexword

Table 1: Values of `ldMaxWidth` for the method `variable()`. Note that the value `0` is not permitted, so a dibit carries only three possible values, a nybble only fifteen possible values, and so on.

`b`: The Boolean flag to be written.

`Out::flush()`

This method flushes out any bits waiting in the object, if necessary, by writing out a complete byte (with the appropriate number of least significant bits set to zero) to the `std::ostream` specified in the object's constructor. Its prototype is

```
void flush();
```

The bit position is appropriately updated; *i.e.*, after a call to `flush()`, it will be a multiple of eight.

If there are no bits waiting to be written, the method has no effect. Thus, any number of consecutive calls to `flush()` is equivalent to a single `flush()`.

The object can continue to be used after a `flush()` command, and any number of `flush()` commands can be issued.

The object's destructor calls `flush()` to ensure that the last bits are written out before the object is destroyed.

`Out::position()`

This is the accessor for the current bit position in the bitstream. Its prototype is

```
PositionType position() const;
```

`PositionType`: The integral type used for position values, as specified in the object's constructor.

Return value: The current bit position.

Note that, at any point in time, the number of bytes that have been written to the `std::ostream` specified in the object's constructor will be the value of `position()`

divided by eight, rounded down; *i.e.*, there may be up to seven bits that have not yet been written to the `std::ostream`, but they are still counted in the value returned by `position()`.

`In::In()`

This constructor has prototype

```
In( std::istream& instream = std::cin );
```

`instream`: The STL `istream` from which the bitstream will be read. The default stream is `cin`.

Note that the bitstream is read from `instream` one byte at a time. Also note that the `In` class allows one extra byte to be read when `instream` returns true for `eof()`; this allows for up to eight "extra" bits to be read, and subsequently unread.

`In::bits()`

This method reads an arbitrary number of bits into a `std::vector` of bytes, corresponding to method `Out::bits()`. Its prototype is

```
template< typename NumBitsType >
void bits( std::vector< unsigned char >& data, NumBitsType numBits );
```

`NumBitsType`: The type of `numBits`.

`data`: Vector into which the bits will be read.

`numBits`: The number of bits to read. If this value is negative, the method throws an exception.

Note that the vector `data` is cleared before the bits are read in.

`In::skip()`

This method skips (reads and ignores) an arbitrary number of bits. Its prototype is

```
template< typename NumBitsType >
void skip( NumBitsType numBits );
```

`NumBitsType`: The type of `numBits`.

`numBits`: The number of bits to skip. If this value is negative, the method throws an exception.

`In::fixed()`

This method reads a fixed bit-width non-negative integral value, usually written by the method `Out::fixed()`. Its prototype is

```
template< typename ValueType >
void fixed( ValueType& value, int width );
```

`ValueType`: The type of `value`. This can be any integral type that is wide enough to hold non-negative values of bit-width `width`.

`value`: Reference to where the read value will be stored.

`width`: The bit-width required. If this value is too large for the type `ValueType`, the method throws an exception.

`In::variable()`

This method reads a positive integral value written with variable bit-width by the method `Out::variable()`. Its prototype is

```
template< typename ValueType >
int variable( ValueType& value, int ldMaxWidth );
```

`ValueType`: The type of `value`. This can be any integral type that is wide enough to hold the *actual* value read in.

`value`: Reference to where the read value will be stored.

`ldMaxWidth`: The logarithm to base 2 of the maximum bit-width possible, as specified in the original call to `Out::variable()` (see Table 1).

Return value: The total number of bits read from the bitstream to extract `value`.

If the method determines that the value to be read in will be larger than the maximum value specified by Table 1 for the given value of `ldMaxWidth`, then the function throws an exception, and unreads the bits that were read to make this determination. (In other words, if this exception is caught, the value can still be read in by issuing a subsequent call to this method with a suitably wide `ValueType`.)

`In::boolean()`

This method reads a single bit as a Boolean flag. Its prototype is

```
void boolean( bool& b );
```

`b`: Reference to where the read Boolean flag will be stored.

`In::unread()`

This method unreads some or all of the last bits read from the bitstream, up to a maximum of eight bits. Its prototype is

```
void unread( int numBits );
```

If `numBits` is negative, greater than the number of bits last read from the object, or greater than eight, then the method throws an exception.

`In::flush()`

This method flushes the input bitstream to a byte boundary, *i.e.*, skips from zero to seven bits, so that the bit position is a multiple of eight. Its prototype is

```
void flush();
```

Note that this method only flushes to the nearest byte boundary from the current bit position, regardless of whether bits past this boundary have been previously read and then unread using `unread()`.

For example, if five bits are read from a new `In` object, then six more bits are read, the bit position will be 11. If four bits are then unread, the bit position will then be 7. If `flush()` is then called, only one bit will be skipped, leaving the bit position at 8.

This behavior means that `In::flush()` is an exact counterpart to `Out::flush()`; a call to the latter on output can always be matched precisely by a call to the former on input. It does, however, mean that the `In` object is not actually "flushed" in the sense of "clearing its buffer"; if unreading has occurred, it is possible that eight bits will still remain "cached" in the object after a call to `flush()`.

`In::position()`

This is the accessor for the current bit position in the bitstream. Its prototype is

```
PositionType position() const;
```

Return value: The current bit position.

Note that the number of bytes read from the `std::istream` specified in the constructor will be the value of `position()` divided by eight, rounded up, or one greater than this value (if `unread()` has been called); *i.e.*, there may be up to fifteen bits that have been read from the `std::istream`, but not read from the bitstream; these bits are not counted in the value returned by `position()`.

`width< ValueType >()`

This function returns the bit-width of any non-negative integer value. Its prototype is

```
template< typename ValueType >
int width( ValueType value );
```

`ValueType`: The type of `value`.

`value`: The integer value whose bit-width is desired. If `value` is negative, the function throws an exception.

Return value: The bit-width of `value`.

`variableBits< ValueType >()`

This function computes the number of bits that would be required to encode a variable-width positive integer using `Out::variable()`. Its prototype is

```
template< typename ValueType >
int variableBits( ValueType value, int ldMaxWidth );
```

`ValueType`: The type of `value`.

`ldMaxWidth`: The logarithm to base two of the maximum bit-width of `value`; see Table 1.

Return value: The number of bits that would be required.

In terms of its return value, this function behaves the same as `Out::variable()`. The only difference is that nothing is actually written out to any output bitstream.

4. Copyright notice