Toward Boost.Bitfield 0.2.0

Vicente J. Botet Escriba

Table of Contents

Warning

Bitfield is not a part of the Boost libraries.

Overview

How to Use This Documentation

This documentation makes use of the following naming and formatting conventions.

- Code is in `fixed width font` and is syntax-highlighted.
- Replaceable text that you will need to supply is in *`italics`* .
- Free functions are rendered in the code font followed by (), as in free_function().
- If a name refers to a class template, it is specified like this: `class_template<>`; that is, it is in code font and its name is followed by <> to indicate that it is a class template.
- If a name refers to a function-like macro, it is specified like this: `MACRO()`; that is, it is uppercase in code font and its name is followed by `()` to indicate that it is a function-like macro. Object-like macros appear without the trailing `()`.
- Names that refer to *concepts* in the generic programming sense are specified in CamelCase.

Note

In addition, notes such as this one specify non-essential information that provides additional background or rationale.

Finally, you can mentally add the following to any code fragments in this document:

```
// Include all of Bitfield files
#include <boost/integer/bitfield.hpp>
using namespace boost::integer;
```

Motivation

In order for an application or a device driver to use the same source code base on both platforms, it must either be endian neutral, or use conditional compilation to select appropriate code modules. A program module is considered endian neutral if it retains its functionality while being ported across platforms of different endianness. In other words, there is no relation between its functionality and the endianness of the platform it is running on. Endianness issues become important when porting the pieces of the application that relate to client communication over a heterogeneous network, persistent data storage on the disk, product tracing (it is important that trace generated on SPARC gets formatted correctly on x86) and other related areas.

Boost.Endian handles with most of the issues related to the byte ordering, but do not manage the bit ordering. E.g. the following structure has a different layout in big and little endian machines.

```
struct X {
    unsigned int d00:1;
    unsigned int d01:7;
    unsigned int d02:2;
    unsigned int d03:6;
    unsigned int d04:4;
    unsigned int d05:4;
    unsigned int d06:1;
    unsigned int d07:3;
    unsigned int d08:4;
};
```

In order to make this structure endian neutral, we need to use conditional compilation, and swap on byte level of the fields

```
struct X {
#if __BYTE_ORDER == __BIG_ENDIAN
    unsigned int d00:1;
    unsigned int d01:7;
    unsigned int d02:2;
    unsigned int d03:6;
    unsigned int d04:4;
    unsigned int d05:4;
    unsigned int d06:1;
    unsigned int d07:3;
    unsigned int d08:4;
#else
    unsigned int d01:7;
    unsigned int d00:1;
    unsigned int d03:6;
    unsigned int d02:2;
    unsigned int d05:4;
    unsigned int d04:4;
    unsigned int d08:4;
    unsigned int d07:3;
    unsigned int d06:1;
#endif
}
```

The preceding technique can be applied as far as the bitfield is contained on a byte boundary. But for bitfields using several bytes as in

```
struct X {
    unsigned int d00:11;
    unsigned int d01:21;
};
```

there is no way using conditional compilation to adapt the structure and preserve the fields.

Two approaches can be considered:

Decompose the fields not defined on a byte boundary on several bitfield defined on a byte boundary

```
struct X {
#if __BYTE_ORDER == __BIG_ENDIAN
    unsigned int d00_b0:8;
    unsigned int d00_b1:3;
    unsigned int d01_b0:5;
    unsigned int d01_b1:8;
    unsigned int d01_b2:8;
#else
    unsigned int d01_b2:8;
    unsigned int d01_b1:8;
    unsigned int d01_b0:5;
    unsigned int d00_b1:3;
    unsigned int d00_b0:8;
#endif
};
```

and define a way to access the bitfield information of the ENDIAN_UNSAFE structure. This library do not use this apprach.

Replace the bitfields by the integer type

When it is possible, a suggestion would be to transform the bitfields into integers, as follows:

```
struct X {
#ifdef ATRIUM_PORTABLE
    unsigned int d0;
#else
    unsigned int d00:11; /* ENDIAN_UNSAFE */
    unsigned int d01:21; /* ENDIAN_UNSAFE */
#endif
};
```

and define a way to access the bitfield information.

When the structure is used by a C++ program, all the uses of the bit fields (d00 and d01) could be replaced by an inline function, so

```
y = x.d00;
```

could be replaced by

```
y = x.d00();
```

or

```
y = x.get_d00();
```

and x.d00= y;

could be replaced by

```
x.d00()= y;
```

or

```
x.set_d00(y)
```

where

```
struct X {
    typedef std::uint_32 storage_type;
    storage_type d0;
    typedef unsigned int value_type;
    BOOST_BITFIELD_DCL(storage_type, d0, unsigned int, d00, 0, 10);
    BOOST_BITFIELD_DCL(storage_type, d0, unsigned int, d01, 11, 31);
};
```

The goal of Boost.Bitfield is to make possible this simple portable translation.

If we want to write the storage type to a binary archive we need to precise which will be the endian format of the storage type. For this end, you can use any of the endian types provided by the Boost.Endian library. If the storage is big you can declare

```
struct X {
    typedef boost::ubig_32 storage_type;
    storage_type d0;
    typedef unsigned int value_type;
    BOOST_BITFIELD_DCL(storage_type, d0, unsigned int, d00, 0, 10);
    BOOST_BITFIELD_DCL(storage_type, d0, unsigned int, d01, 11, 31);
};
```

Description

Portable bitfields.

The main source of inspiration of this library was Boost.Bitfield from Emile Cormier.

Boost.Bitfield provides:

- a generic bitfield traits class providing generic getter and setter.
- a BOOST_BITFIELD_DCL macro making easier the definition of the bitfield helper type and the bitfield getter and setter functions.

Users'Guide

Getting Started

Installing Bitfield

Getting Boost.Bitfield

You can get the last stable release of Boost.Bitfield by downloading `bitfield.zip` from the Boost Vault

You can also access the latest (unstable?) state from the Boost Sandbox.

Building Boost.Bitfield

Boost.Bitfield is a header only library, so no need to compile anything.

Requirements

Boost.Bitfield depends on some Boost library. For these specific parts you must use either Boost version 1.38.0 or the version in SVN trunk (even if older version should works also).

Exceptions safety

All functions in the library are exception-neutral, providing the strong exception safety guarantee.

Thread safety

All functions in the library are thread-unsafe except when noted explicitly.

Tested compilers

Boost.Bitfield should work with an C++03 conforming compiler. The current version has been tested on:

Windows with

- MSVC 10.0

Cygwin 1.5

- GCC 3.4.4

Cygwin 1.7

- GCC 4.3.2

MinGW with

- GCC 4.4.0
- GCC 4.5.0

Note

Please let us know how this works on other platforms/compilers.

Note

Please send any questions, comments and bug reports to boost <at> lists <dot> boost <dot> org.

Tutorial

Declaring a storage variable

The first thing to do is to declare a storage type able to store all the bitfields. In the following example we want to stor bits for the RGB encoding, so 16 bits is enough.

```
struct Rgb565
{
    typedef ubig16_t storage_type;
    storage_type storage;
};
```

Bitfieds helper classes and bitfield accessors

Declaring a bitfield helper type

For each bitfields we need to declare a type that knows about the bitfield traits of the bitfield.

```
struct Rgb565
{
    //...
    typedef bitfield_traits<storage_type, 0, 4, unsigned char>    red_type;
};
```

bitfield getters

Now we are ready to define the getter of this bitfield respect to the storage variable. My prefered form is just to name the getter function as the type, others will prefer the get_field() form.

```
struct Rgb565
{
    //...
    red_type::value::value_type red() const { return red_type::value(storage).get(); }
    red_type::value::value_type get_red() const { return red_type::value(storage).get(); }
};
```

bitfield setters

There are two variants for the setter. My prefered form is just to name the setter function as the type and return a reference able to support assignations. The other is the traditional set_field() taking the new value as parameter.

```
struct Rgb565
{
    //...
    red_type::reference red() { return red_type::reference(storage); }
    void set_red(unsigned char val) { return red_type::reference(storage).set(val); }
};
```

Using macros to declare portable bitfields

All these stuf can be done at once using the BOOST_BITFIELD_DCL macro as follows.

```
struct Rgb565
{
    typedef volatile uint16_t storage_type;
    storage_type storage;
    typedef volatile uint16_t value_type;
    BOOST_BITFIELD_DCL(storage_type, storage, unsigned char, red, 0, 4);
    BOOST_BITFIELD_DCL(storage_type, storage, unsigned char, green, 5, 10);
    BOOST_BITFIELD_DCL(storage_type, storage, unsigned char, blue, 11,15);
};
```

Using bitfield getters and setters

Next follows some examples of how these bitfields can be used.

```
Rgb565 r;
r.storage= 0xffff;

// Write to a bitfield. Note that parenthesis are needed around the bitfield so
r.red() = 23;

// Read from a bitfield
Rgb565::value_type b = r.blue();

// Access a bit within a bit field
bool bit = r.blue()[3];
```

References

Boost.Endian general integer-like byte-holder binary types with explicit control over byte order, value type, size, and alignment from Beman Dawes

Reference

Header `<boost/integer/bitfield/bitfield.hpp>`

This is the main file of Boost.Bitfield, which includes the definition of the bitfield<> class.

```
namespace boost { namespace integer {
    template <typename T>
    struct bitfield_default_value_type;

    template <class, std::size, std::size, class, bool, class>
    class bitfield;

    template <typename STORAGE_TYPE, int F, int L
        , typename VALUE_TYPE=bitfield_default_value_type<typename STORAGE_TYPE::type>
    >
    struct bitfield_traits;
}
```

Metafunction `bitfield_default_value_type<>`

The default value type associated to an storage type is the storage type.

```
template <typename T>
struct bitfield_value_type {
    typedef T type;
};
```

The user could need to specialize this metafunction for some specific types, as for example for types comming from the Boost.Endian library.

Template class `bitfield<>`

The bitfield class is a wrapper to an storage type that allows to get/set a bitfield of a given value type defined by the first and last bit that the bitfield occupes in the strogae type. It defines the conversion to the bitfield value type as weel as all the arithmetic assignement operators.

This template class behaves as a reference to a bitfield. Note that C/C++ do not allows to take directly the address of a bit field.

```
template <typename STORAGE_TYPE, std::size F, std::size L
    , typename VALUE_TYPE=typename bitfield_value_type<typename STORAGE_TYPE::type>
    , typename REFERENCE_TYPE=STORAGE_TYPE&>
class bitfield {
public:
    //! Reference type of the word occupied by the bitfield
    typedef REFERENCE_TYPE reference_type;
    //! Value type of the bitfield itself
    typedef VALUE_TYPE value_type;
    //! storage type of the bitfield support
    typedef typename reference_traits<REFERENCE_TYPE>::value_type storage_type;

    //! Useful constants
    static const unsigned int FIRST;        //!< Position of the first bit
    static const unsigned int LAST;         //!< Position of the last bit
    static const unsigned int STS;
    static const unsigned int LASTD;
    static const unsigned int WIDTH;        //!< Width in bits of the bitfield
    static const unsigned int VAL_MASK;     //!< Mask applied against assigned values
    static const unsigned int SIGN_MASK;    //!< Sign Mask applied against assigned values
    static const value_type MIN_VAL;       //!< min value that can be represented with the bit↵
field
    static const value_type MAX_VAL;       //!< max value that can be represented with the bit↵
field
    static const unsigned int FIELD_MASK;   //!< Mask of the field's bit positions

    static std::size_t first();
    static std::size_t last();
    static std::size_t width();
    static storage_type val_mask();
    static storage_type field_mask();

    //! deleted because a reference is nedeed
    bitfield() = delete;
    //! explicit constructor from a reference
    explicit bitfield(reference_type field);

    //! Assignment from a value type
    bitfield& operator=(value_type val);

    //! value_type implicit conversion
    operator value_type() const;

    //! value_type explicit getter
    value_type get() const;

    //! Returns the negative of the bitfield value.
    // this must be modified on the case of signed value_type
    value_type operator~() const;

    bit_reference_type operator[](std::size_t index);

    bit_const_reference_type operator[](std::size_t index) const;

    //! Returns the current bitfield value as bit flags.
    /*! The returned bit flags can be ORed with other bit flags. */
    storage_type flags() const;

    //! Returns the current bitfield value as bit flags.
    /*! The returned bit flags can be ORed with other bit flags. */
    static storage_type flags(value_type val) const;
```

```
    //! Arithmetic-assign operators
    bitfield& operator++();
    value_type operator++(int);

    bitfield& operator--();
    value_type operator--(int);

    bitfield& operator+=(value_type rhs);
    bitfield& operator-=(value_type rhs);

    bitfield& operator*=(value_type rhs);
    bitfield& operator/=(value_type rhs);
    bitfield& operator%=(value_type rhs);

    bitfield& operator<<=(int rhs);
    bitfield& operator>>=(int rhs);

    bitfield& operator&=(value_type rhs);
    bitfield& operator|=(value_type rhs);
    bitfield& operator^=(value_type rhs);
};
```

Template class `bitfield_traits<>`

This traits class defies two traits:

- reference: used to modify a bitfield
- value: used to get the value of a bitset

```
template <typename STORAGE_TYPE, int F, int L
    , typename VALUE_TYPE=typename bitfield_value_type<typename STORAGE_TYPE::type>
>
struct bitfield_traits {
    typedef bitfield<STORAGE_TYPE, F, L, VALUE_TYPE> reference;
    typedef bitfield<const STORAGE_TYPE, F, L, VALUE_TYPE> value;
};
```

Header `<boost/integer/bitfield/bitfield_dcl.hpp>`

This file constains the macro making easier the definition of the bitfield helper type and the bitfield getter and setter functions.

```
#define BOOST_BITFIELD_DCL(STORAGE_TYPE, STORAGE_VAR, VALUE_TYPE, FIELD, F, L)
```

Macro `BOOST_BITFIELD_DCL`

```
#define BOOST_BITFIELD_DCL(STORAGE_TYPE, STORAGE_VAR, VALUE_TYPE, FIELD, F, L)
```

- Parameters
  - STORAGE_TYPE: the type to store the bitfield
  - STORAGE_VAR: the variable used to store the bitfield
  - VALUE_TYPE: the type for the bitfield

- FIELD: the name of the bitfield
- F: the first bit
- L: the last bit

- Effect This macro defines the bitfield traits and the functions necessary to make valid the following expressions:
  - str.FIELD() = value;
  - var= str.FIELD();
  - str.set_FIELD(var);
  - var= str.get_FIELD()

Two styles of getter/setter are provided: one using overloading on the field name, the other using clasical prefic get/set.

Header `<boost/integer/bitfield/endian_bitfield_value_type.hpp>`

This file includes the customization for the endian types for the bitfield_default_value_type metafunction. You need to include this file if you use endian types as storage for the bitfields.

```
namespace boost { namespace integer {
    template <endianness E, typename T, std::size_t n_bits, alignment A>
    struct bitfield_value_type<endian<E,T,n_bits,A> > {
        typedef typename endian<E,T,n_bits,A>::value_type type;
    };
}
```

Appendices

Appendix A: History

Version 0.2.0, Jan 29, 2011 *Bitfield porting to Windows and MSVC*

Bugs:

- Fix uses of specific POSIX files.
- Adapt to MSVC constraints.

Version 0.1.0, April 29, 2009 *Announcement of Bitfield*

Features:

- a generic bitfield traits class providing generic getter and setter for portable bitfields.
- a BOOST_BITFIELD_DCL macro making easier the definition of the bitfield helper type and the bitfield getter and setter functions.

Appendix B: Rationale

Why we can not declare portable C/C++ bitfields ?

Appendix C: Implementation Notes

FAQ

- Why bother with endian bitfields ?

External data portability and both speed and space efficiency. Availability of bit order representations is important in some applications.

- Why not just use Boost.Serialization?

Serialization involves a conversion for every object involved in I/O. Bitfields objects require no conversion or copying. They are already in the desired format for binary I/O. Thus they can be read or written in bulk.

- Do this type have any uses outside of I/O?

Probably not.

- Is there is a performance hit when doing arithmetic using these types?

Yes, for sure, compared to arithmetic operations on native bitfields integer types. However, these types are usually faster, and sometimes much faster, for I/O compared to stream inserters and extractors, or to serialization.

- These types are really just byte-holders. Why provide the arithmetic operations at all?

The first goal of the library is to be as close as possible of the usage of bitfields on C/C++. Providing a full set of operations reduces program clutter and makes code both easier to write and to read. Consider incrementing a variable in a record. It is very convenient to write:

```
++record.bf();
```

Rather than:

```
int temp( record.bf());
++temp;
record.vf() = temp;
```

Design considerations for Boost.Bitfield

- Must provide exactly the size and internal bit ordering specified.
- It is better software engineering if the same implementation works regardless of the CPU endianness. In other words, #ifdefs should be avoided where possible.

Appendix D: Acknowledgements

Thanks to Emile Cormier, the initiator of this library. And also to Beman Dawes; in a first version of the Boost.Bitfield the bitfield class tokes in addition an endian template parameter to make the needed endian conversions. With the Boost.Endian this is much more simpler and orthogonal.

Appendix E: Tests

aligned bitfields

Name	kind	Description	Result	Ticket
get	run	check getters	Pass	#
assign	run	check setters	Pass	#
flags	run	check flags	Pass	#
traits	run	check the traits of bitfield	Pass	#
invert	run	check the `operator~`	Pass	#
bit	run	check the `operator[]`	Pass	#
signed	run	check signed bitfields	Pass	#

Appendix F: Tickets

Appendix F: Future plans

Tasks to do before review

- Add test with the Boost.Endian types and binary archive.

For later releases

Add bitfields group

Used to easily manipulate groups of bitfields the same way as does C bitfields, but in a portable manner. Example declaration:

```
struct Rgb565
{
    struct red {};
    struct green {};
    struct blue {};
    typedef bitfields<mpl::vector<
            member<unsigned char, red,   5>,
            member<unsigned char, green, 6>,
            member<unsigned char, blue,  5>
    > > type;
};
```

Example usage:

```
Rgb565::type r = make_bitfields<Rgb565::type, 1,2,3>;

// Write to a bitfield.
r.get<Rgb565::red>() = 23;
//or
r.set<Rgb565::red>(23);

// Read from a bitfield
Rgb565::at<Rgb565::blue>::value_type b = r.get<Rgb565::blue>();
```

Other posibility could be to use unamed bitfields whic are accessed as tuples.

```
typedef bitfields_group<mpl::vector_c<5,6,5> > Rgb565;
Rgb565 r;

r.get<0>() = 23;
// or
r.set<0>(23);

// Read from a bitfield
Rgb565::at<2>::value_type b = r.get<2>();
```

Add pointer_plus_bits

Based on

- The article of Joaquin Optimizing red-black tree color bits,
- the implementation of Ion pointer_plus_bits from Boost.Intrusive , and
- Clang's QualType smart pointer

This class will allows to use the unsused bits of a pointer to reduce the size of the nodes containing pointers and bits and sometimes improving also the performances.

I have not reached yet the interface I would like to have. For the moment we can do

```
typedef pointer_plus_bits<int*,1,bool>::type pint_and_bool;

int i=0;
pint_and_bool v1;
ASSERT_EQUALS(v1.pointer(),0);
ASSERT_EQUALS(v1.small_int(),false);
pint_and_bool v2(&i, true);
ASSERT_EQUALS(v2.pointer(),&i);
ASSERT_EQUALS(v2.small_int(),true);
v1.pointer()=v2.pointer();
v1.small_int()=true;
ASSERT_EQUALS(v1.pointer(),&i);
ASSERT_EQUALS(v1.small_int(),true);

typedef pointer_plus_bits<
        pointer_plus_bits<int*,1,bool>::type
                         ,1, bool>::type  pint_and_bool_bool
pint_and_bool_bool v1;
ASSERT_EQUALS(v1.small_int(),false);
ASSERT_EQUALS(v1.pointer().get_pointer(),0);
ASSERT_EQUALS(v1.get_pointer().get_pointer(),0);
ASSERT_EQUALS(v1.get_pointer().small_int(),false);
```