Flexible, Portable Bit Fields

1. Introduction

This document describes a bit field facility provided by the header file bitfield.h. The facility is more portable and flexible than the bit field facility provided by the base language. Bit field structures can be laid out from most to least or least to most significant bits, under programmer control. Bit field structures can be located in device registers that are not memory mapped. With good compiler optimization, the performance is comparable.

For requirements for template parameters given below, the “as if” rule applies.

This facility is in compliance with all ISO C++ Standards (beginning with C++98). Nominally, the facility is not portable under (any version of) the Standard. Its implementation depends on the use of `reinterpret_cast`. However, the de facto portability is extremely high.

This facility reserves all identifiers in all scopes that begin with the prefix `BITF_`. It also reserves the namespace `Bitfield_impl`.

2. A Quick Example

Instead of:

```
struct A
  {
    union
      {
        struct
          {
            unsigned x : 5;
            unsigned a0 : 2;
```

```
          unsigned a1 : 2;
          unsigned a2 : 2;
        }
      u;

      struct
        {
          unsigned x : 5;
          unsigned all_a : 6;
        }
      v;
    };
  };

…

A x;
x.v.all_a = 0x3f;
x.u.a1 = 0;
```

write:

```
typedef Bitfield<Bitfield_traits_default<> > Bf;

struct A : private Bitfield_fmt
  {
    F<5> x;
    F<2> a[3];
  };

typedef Bitfield_w_fmt<Bf, A> Bwf;

…

Bwf::Format::Define::T x;
BITF(Bwf, x, a) = 0x3f;
BITF(Bwf, x, a[1]) = 0;
```

3. Bit Field Structures

This facility allows a set of M storage locations (M a positive integer) for unsigned values to be manipulated as a structure of bit fields. The bit precision of the unsigned values must be the same as some unsigned C++ type (the *storage type*). Each storage location must be identified by an integer from 0 to M – 1.

The structure of the bit fields is defined by a bit field *format* structure. Format structures are not instantiated. Each byte in a format structure corresponds to a bit in the bit structure format it defines (so the `sizeof` the format structure has to be less than or equal to M * P, where P is the number of bits of precision of the storage type). Format structures are built up from fields whose types are instantiations of this class template:

```
template <unsigned BITF_WIDTH> struct F { char x[BITF_WIDTH]; };
```

The properties of a bit field format structure are:

- It defines the class template F. Typically, this can be done by (privately) inheriting from the class `Bitfield_format`, which defines F.

- There are no function members.
- There are no private members.
- An instantiation of the F class template is a valid type for a data member.
- Another format structure is a valid type for a data member.
- An array of elements of another valid data member type is also a valid data member type.
- A union of fields of other valid data member types is also a valid data member type.
- Other than `Bitfield_format`, base classes of a format structures must be other format structures and must be publicly inherited.

It is important that the empty base optimization (http://en.cppreference.com/w/cpp/language/ebo) causes no bytes to be allocated for the `Bitfield_format` subclass. Because of this, if the type of the first data member is not an instantiation of F, the format structure must have F as a direct member class template, and not have `Bitfield_format` as a base classs. bitfield.h defines the following macro to provide a shorthand way of defining F:

```
#define BITF_DEF_F \
template <unsigned BITF_WIDTH> struct F { char x[BITF_WIDTH]; };
```

For the facility to work properly, the compiler must not insert “pad” bytes or reorder format structure components in the memory layout. The rules governing class memory layouts are not consistent among the various versions of the C++ Standard. De facto, however, I don't think there is any compiler that would pad or reorder the memory layout of a format structure as defined here.

A single bit field may map to multiple storage locations with consecutive identification numbers.

3.1. An Example

This is the format structure for the register for the Texas Instruments PC16550D Universal Asynchronous Receiver/Transmitter (http://www.ti.com/lit/ds/symlink/pc16550d.pdf).

```
struct Fmt_pc16550d : private Bitfield_format
  {
    F<8> scratch;

    struct Modem_status
      {
        F<1> dcd, ri, dsr, cts, ddcd, teri, ddsr, dcts;
      }
    modem_status;

    struct Line_status
      {
        F<1> err_rx_fifo, temt, thre, bi, fe, pe, oe, dr;
      }
    line_status;

    struct Moden_control
```

```
    {
      F<3> zero;
      F<1> loop, out2, out1, rts, dtr;
    }
  modem_control;

  struct Line_control
    {
      F<1> dlab, set_break, stick_parity, eps, pen, stb;
      F<2> data_bits_minus_5;
    }
  line_control;

  union
    {
      // On write.
      struct Fifo_control
        {
          F<2> rx_trigger, reserved;
          F<1> dma_mode, tx_fifo_reset, rx_fifo_reset, fifo_enable;
        }
      fifo_control;

      // On read.
      struct Intr_ident
        {
          F<2> fifos_enabled, zero;
          F<3> intr_id;
          F<1> comp_intr_pending;
        }
      intr_ident;
    };

  union
    {
      // DLAB in line control must be 1 to access this.
      F<16> divisor_latch;

      // DLAB in line control must be 0 to access this.
      struct
        {
          struct Int_enable
            {
              F<4> zero;
              F<1> edss, els, etbe, erbf;
            }
              int_enable;

          // Tx on write, Rx on read.
          F<8> tx_rx_byte;
        }
      dlab0;
    };
};
```

Assumptions

- The storage type is `uint8_t`.

- The identification numbers for the storage locations are the register addresses given in the data sheet.
- The offsets for the bit fields will be calculated from the end of the format structure, and the offsets will run from low to high address numbers, and then from least to most significant bit.

4. The `Bitfield` Class Template

4.1. Template Parameters

4.1.1. `class Traits`

For bit field structures that are stored in (successive locations in) main memory, the class template `Bitfield_traits_default` is provided (at global scope) for use as the `Traits` parameter to B`itfield`. `Bitfield_traits_default` has two template parameters. The first is the value type (which defaults to `unsigned`), which should be some unsigned C++ integral type, The second is the storage type (which defaults to the value type), which should also be some unsigned C++ integral type. (To make the bit fields read-only, the storage type should be `const`) Instantiations of `Bitfield_traits_default` have a copy constructor whose only parameter is a storage type pointer. The address of the storage location with the least address should be passed as the constructor parameter.

4.1.1.1. Required Public Type Members

4.1.1.1.1. `Value_t`

An unsigned C++ integral type. The number of bits of precision of this type must be greater than or equal to the bit width of any bit field that will be accessed using the instantiation of `Bitfield`. For `Bitfield_traits_default`, this will be the same as the value type parameter.

4.1.1.1.2. `Storage_t`

An unsigned C++ integral type. The storage type discussed in section 2. The number of bit of precision of `Value_t` must be divisible the number of bits of precision of this type. For `Bitfield_traits_default`, this will be the same as the storage type parameter.

4.1.1.1.3. `Storage_access_t`

This must be a class. Its internal state must select a particular storage location (see section 2). In addition to a copy constructor, it has the following required public member functions and operator.

4.1.1.1.3.1. `void operator += (unsigned offset)`

If n is the numeric identifier for the storage location selected by the instance, calling this operator will change the selected storage location to n + offset.

4.1.1.1.3.2. `Storage_t read()`

Returns the current value of the selected storage location. For a particular bit field operation, this function will be call at most once for a particular storage location.

4.1.1.1.3.3. `void write(Storage_t t)`

Changes the current value of the selected storage location to t. If bit field values will only be read, but not altered, using this instantiation of `Bitfield`, this member function can be omitted. For a particular bit field

modification operation, this function will be call at most once for a particular storage location.

4.1.1.2. Required Boolean Static Constants

Alll of these default to true in `Bitfield_traits_default`.

4.1.1.2.1. `Storage_ls_bit_first`

If this is true, bit offsets run from the least to the most significant bit in the storage locations. If it is false, bit offsets run the most to the least significant bit in the storage locations. (Bit offsets start in storage location 0 and run to the higher-numbered locations.)

4.1.1.2.2. `Fmt_offset_from_start`

If this is true, offsets for fields in format structures are byte offsets from the start. Otherwise, they are from the end (with the ending field having offset zero).

4.1.1.2.3. `Fmt_align_at_zero_offset`

If this is true, the offset for a bit field is the same as the offset of the corresponding field in the format structure. Otherwise, there is a non-negative offset from the format field offsets to the bit field offsets, so that the highest offset bit in the highest numbered storage location is part of the bit field with the highest offset.

4.1.1.3. Example Traits Class

In this hypothetical scenario, the bit fields are in 16-bit registers at consecutive offsets in a device. The registers have the opposite endiance from the processor. The registers are not directly mapped. To access a register, you first do an 8-bit write to address 0xE0000400. You then read or write the register value by doing a 16-bit read or write to the address 0xE0000402. The processor architecture guarantees strong memory ordering for accesses to these addresses. (That is the order of the actual accesses to these addresses is the same as the order of access in the nominal execution order of instructions.) The device registers are only accessed from a single execution thread and not from any interrupt service routine. Thus, no mutual exclusion protection of the 2-step access process is necessary. The largest bit field in the device registers is 27 bits wide, so `uint32_t` is chosen as the value type. The documentation lists the device registers bit fields from high to low offset, most to least significant, so the traits class is defined to allow the fields in the bit format structure to be in the same order. Here is the traits class:

```
class Example_bitfield_traits
  {
  public:

    typedef uint32_t Value_t;

    typedef uint16_t Storage_t;

    class Storage_access_t
      {
      private:

        static volatile uint8_t & reg_select()
          { return(*reinterpret_cast<volatile uint8_t *>(0xE0000400)); }

        static volatile uint16_t & reg_buffer()
          { return(*reinterpret_cast<volatile uint16_t *>(0xE0000402)); }

        uint16_t reg_offset;
```

```
    public:

      Storage_access_t() : reg_offset(0) { }

      void operator += (unsigned offset) { reg_offset += offset; }

      Storage_t read()
        {
          reg_select() = reg_offset;
          Storage_t reg_val = reg_buffer();
          return((reg_val >> 8) | (reg_val << 8));
        }

      void write(uint8_t reg_val)
        {
          reg_select() = reg_offset;
          reg_buffer() = (reg_val >> 8) | (reg_val << 8);
        }
    };

  static const bool Storage_ls_bit_first = true;

  static const bool Fmt_offset_from_start = false;

  static const bool Fmt_align_at_zero_offset = true;

};
```

4.1.3. `class Err_act`

The required public members of this class are two static functions:

```
void field_too_wide(unsigned field_width)
void value_too_big(Value_t v, unsigned field_width)
```

The first is called if the width of a bit field is detected to be bigger than the number of bits of precision in the value type. The second is called if the value to be used to modify a bit field is detected to be need more bits of precision than the width of the bit field to be modified.

In the default class for this parameter, the functions do nothing. You can create a class to use as this parameter that throws exceptions, logs the error, resets the board, etc.

4.2. Public Members That Duplicate Traits Class Parameter Members

`Value_t`

`Storage_t`

`Storage_access_t`

`Storage_ls_bit_first`

`Fmt_offset_from_start`

`Fmt_align_at_zero_offset`

4.3. Other Public Members

4.3.1. Types

4.3.1.1. `class Bf`

None of the member functions of `Bf` validate parameters or data members, except as noted below. (Validation failures result in calls to the member functions of the `Err_act` template class parameter.) This class has the following functions and operators as its public members (as well as a trivial destructor):

4.3.1.1.1. Constructors

```
Bf(Storage_access_t base, unsigned short first_bit, unsigned short
field_width)
```

`base` should initially be selecting storage location 0. `first_bit` is the offset of the first bit in the bit field. `field_width` is the width is bits of the bit field. This constructor is typically only used indirectly by invoking the `BITF` macros described below. If the offset corresponds to a bit in a storage location whose number is out of range, the resulting behavior is undefined.

```
Bf(const Bf &)
```

Copy constructor.

4.3.1.1.2. `unsigned short width()`

Returns the bit field width (passed to the constructor).

4.3.1.1.3. `unsigned short offset()`

Returns the bit field offset (passed to the constructor).

4.3.1.1.4. `bool is_width_invalid()`

Returns true if the bit field width (stored in a private member) is zero or greater than the bits of precision of the value type.

4.3.1.1.5. Reading

```
Value_t read()

operator Value_t ()

Value_t read_sign_extend()
```

These members return the current value of the bit field. They validate the bit field width (stored in a private member). They return `~Value_t(0)` if the width is invalid (and the `Err_act` member does not throw an exception).

`read_sign_extend()` copies the most significant bit of the bit field value to the more significant bits of the return value. It is up to the calling code to cast the return value to the signed type corresponding to `Value_t`.

4.3.1.1.6. Writing

```
bool write(Value_t val)

Value_t operator = (Value_t val)

bool write_nvc(Value_t val)
```

These members change the current value of the bit field to the value of the parameter. They validate the bit field width (stored in a private member). (Note that, if the current value of the bit field is known to be zero, it will typically require fewer instructions to or the new value into the field rather than writing it in.)

`Operator =` returns the value parameter.

`write()` and `operator =` validate that the value parameter does not exceed the precision of the bit field. (The `_nvc` suffix indicates no value checking.)

`write()` and `write_nvc()` return false if any validation they perform fails.

4.3.1.1.7. `bool zero()`

Sets the current value of the bit field to zero. Validates the bit field width (stored in a private member), returns false if it is invalid.

4.3.1.1.8. `bool b_comp()`

Sets the current value of the bit field to its bitwise compliment. Validates the bit field width (stored in a private member), returns false if it is invalid.

4.3.1.1.9. Anding

```
bool b_and(Value_t val)

Bf & operator &= (Value_t val)

bool b_and_nvc(Value_t val)
```

These members change the current value of the bit field by bitwise-anding it with the parameter. They validate the bit field width (stored in a private member).

`Operator &=` returns a reference to the bit field object.

`b_and()` and `operator &=` validate that the value parameter does not exceed the precision of the bit field. (The `_nvc` suffix indicates no value checking.)

`b_and()` and `b_and_nvc()` return false if any validation they perform fails.

4.3.1.1.10. Oring

```
bool b_or(Value_t val)

Bf & operator |= (Value_t val)

bool b_or_nvc(Value_t val)
```

These members change the current value of the bit field by bitwise-oring it with the parameter. They validate

the bit field width (stored in a private member).

`Operator |=` returns a reference to the bit field object.

`b_or()` and `operator |=` validate that the value parameter does not exceed the precision of the bit field. (The `_nvc` suffix indicates no value checking.)

`b_or()` and `b_or_nvc()` return false if any validation they perform fails.

4.3.1.1.11. Exclusive-oring

```
bool b_xor(Value_t val)

Bf & operator ^= (Value_t val)

bool b_xor_nvc(Value_t val)
```

These members change the current value of the bit field by bitwise-exclusive-oring it with the parameter. They validate the bit field width (stored in a private member).

`Operator ^=` returns a reference to the bit field object.

`b_xor()` and `operator ^=` validate that the value parameter does not exceed the precision of the bit field. (The `_nvc` suffix indicates no value checking.)

`b_xor()` and `b_xor_nvc()` return false if any validation they perform fails.

4.3.1.1.12. Generic Modification

```
template <class Modifier> bool modify(Modifier m)

template <class Modifier> bool modify_nvc(Modifier m)
```

These members change the current value of the bit field in a manner determined by the parameter. They validate the bit field width (stored in a private member).

The template class parameter is required to have this member:

```
void operator () (
  Storage_access_t s, unsigned storage_shift,
  unsigned value_shift, unsigned storage_width)
```

`s` selects the storage location to be modified by the call to the operator. `storage_shift` is the bit shift between the position of the least significant bit of the storage location, and the least significant bit of the bit field in the storage location to be modified by the call to the operator. `storage_width` is the width of the bit field in the storage location to be modified by the call to the operator. `value_shift` is the bit shift between the position of the least significant bit in a `Value_t` instance, and the least significant bit in the bit field in the value that corresponds to the bit field that the call to the operator is modifying.

If the bit field in storage location being modified does not include the most significant bit in the storage location, the corresponding bit field in the value *does* include the most significant bit in the overall bit field in the value.

The template class parameter is also required to have this member function:

```
Value_t value()
```

`modifier()` will validate that the value returned by `value()` does not exceed the precision of the (overall) bit field.

These members are used to implement the other bit field modifying members. It might have been better to make them private or protected members.

4.3.1.2. Utility Modifier Base Classes

```
class Modifier_base
  {
  public:

    static Value_t value() { return(0); }

  protected:

    // Zeros the specified bit field in the given storage location.
    static Storage_t clear(
      Storage_access_t s, unsigned left_shift, unsigned width) { … }
  };

class Value_modifier_base : public Modifier_base
  {
  public:

    Value_t value() const { return(val); }

    Value_modifier_base(Value_t v) : val(v) { }

  private:

    const Value_t val;
  };
```

4.3.1.3. `template<class Format> struct Define`

The parameter to this class template must be a bit field format class.

`T` is a public member type of this structure. It is an array of `Storage_t`, just large enough to hold a bit field structure defined by `Format`.

`Dimension` is an unsigned static const member, which is the dimension of `T`.

4.3.2. Functions

4.3.2.1. Function Returning Bf Instance From Offset, Width

```
static Bf fn(
  Storage_access_t base, unsigned offset, unsigned width)
```

Same parameters as `Bf` constructor.

4.3.2.2. Function Returning Bf Instance From Format Member Pointer

```
template<class Format, typename Mbr_type>
static Bf f(
  Storage_access_t base, Mbr_type Format::*field,
  unsigned offset = 0)
```

`base` must select storage location 0 for the (full) bit field structure. `field` must point to the member of the bit format structure `Format`,corresponding to the bit field. (Template parameter deduction works for this function.) If `Format` is actually the format of a substructure within the full bit field structure, then `offset` must provide the (byte) offset from the start of the full format structure to the start of the format substructure. (If an offset is given, it is presumed to include any necessary alignment offset.)

`field` can be a pointer to a nested structure (to be access as a single large bit field) as long as its size does not exceed the bit precision of the value type.

Use of this function is limited by the limitations of C++ function pointer syntax. Syntax such as `&Fmt::x[3]` and `&Fmt::x.y` is not accepted. The `BITF` macros described below provide a more flexible way of generating `Bf` instances.

4.3.2.3. Function Returning Bf Instance From Format Member Pointer

```
template<class Format, typename Mbr_type>
static unsigned field_width(Mbr_type Format::*field)

template<class Format, typename Mbr_type>
static unsigned field_offset(
  Mbr_type Format::*field, unsigned base_offset = 0)
```

`base` must select storage location 0 for the (full) bit field structure. `field` must point to the member of the bit format structure `Format`,corresponding to the bit field. (Template parameter deduction works for these functions.) If `Format` is actually the format of a substructure within the full bit field structure, then `offset` must provide the (byte) offset from the start of the full format structure to the start of the format substructure. (If an offset is given, it is presumed to include any necessary alignment offset.) `field` can be a pointer to a nested structure (to be treated as a single large bit field).

4.3.3. `Storage_bits`

An unsigned static constant, giving the number of bits of precision of the storage type.

5. The `Bitfield_w_fmt` Class Template

```
template <class Bitfield, class Fmt>
struct Bitfield_w_fmt : public Bitfield
…
```

The template parameter `Bitfield` has the same requirements as the traits parameter to the `Bitfield` template, except that the `Storage_access_t` type is not required. This means that a class that is a specialization of the `Bitfield` template can be use as the `Bitfield` parameter. `Fmt` is a bit field format structure. This class has a public typedef `Format` that is a synonym for the `Fmt` parameter. The class's public member `Storage_bits` is the same as `Bitfield::Storage_bits`.

This is also a member function template:

```
template <class Base_fmt>
static unsigned base_offset(unsigned derived_offset = 0)
```

`Base_fmt` must be a (direct or indirect) base class of `Fmt`. It returns the offset of the base class within `Fmt`. If the bit structure corresponding to `Fmt` is at an offset from the first bit of storage location 0, this offset must be provided as the parameter. (If an offset is given, it is presumed to include any necessary alignment offset.)

6. `BITF` Macros

6.1. Shared Parameters

`BWF` – An instantiation of the `Bitfield_w_fmt` template.

`BASE` – An instance of `BWF::Storage_access_t` that selects the base storage location (storage location 0).

`FIELD_SPEC` – A field specifier for a field within `BWF::Format`. For example, for the field `x.y.z` in the structure `x`, `y.z` is the field specifier. The specified field can be a substructure containing other fields, in which case it will be treated as one large field.

`OFS` – When the bit structure corresponding to `BWF::Format` is contained within a larger bit structure, this value is the offset of the bit structure within the larger bit structure. It is assumed to include any offset required if `BWF::Fmt_align_at_zero_offset` is false.

6.2. `BITF_STD(BWF, BASE, FIELD_SPEC)`

Evaluates to an instance of `BWF::Bf` for the specified bit field.

6.3. `BITF_W_OFS_STD(BWF, BASE, FIELD_SPEC, OFS)`

Evaluates to an instance of `BWF::Bf` for the specified bit field.

6.4. `BITF_OFFSET(BWF, FIELD_SPEC)`

Evaluates to the offset of the specified bit field.

6.5. `BITF_WIDTH(BWF, FIELD_SPEC)`

Evaluates to the width of the specified bit field.

6.6. `BITF_CAT_STD(BWF, BASE, FIELD_SPEC1, FIELD_SPEC2)`

Evaluates to an instance of `BWF::Bf` for the bit field which is the concatenation of the first and second specified bit fields, and any bit fields between them.

6.7. BITF_CAT_W_OFS_STD(BWF, BASE, FIELD_SPEC1, FIELD_SPEC2, OFS)

Evaluates to an instance of `BWF::Bf` for the specified concatenated bit field.

6.8. BITF_CAT_OFFSET(BWF, FIELD_SPEC1, FIELD_SPEC2)

Evaluates to the offset of the specified concatenated bit field.

6.9. BITF_CAT_WIDTH(BWF, FIELD_SPEC1, FIELD_SPEC2)

Evaluates to the width of the specified concatenated bit field.

6.10. Alternate Macros

For each (standard) macro above whose name ends with `_STD`, there is a corresponding wrapper macro with the same name, but with `ALT` substituted for `STD`. In place of the parameters `BWF` and `BASE`, there is instead the parameter `SEL`. Each alternname macro invokes the corresponding standard macro, using `BITF_U_`*`SEL`*`_BWF` for the `BWF` parameter, and `BITF_U_`*`SEL`*`_BASE` for the `BASE` parameter. For example, the invocation:

```
BITF_ALT(MYSEL, abc.arr[3])
```

will expand to:

```
BITF_STD(BITF_U_MYSEL_BWF, BITF_U_MYSEL_BASE, abc.arr[3])
```

The programmer would be responsible for providing definitions for the symbols `BITF_U_MYSEL_BWF` and `BITF_U_MYSEL_BASE`.

6.11. Macro Name Abbreviations

By default, the macro names above ending with `_STD` can be abbreviated by dropping the `_STD`. For example, `BITF_STD` can be abbreviated as `BITF`.

If the preprocessor symbol `BITF_USE_ALT` is defined before including bitfield.h, then macro names ending with `_STD` are not abbreviated. Instread, the macro names above ending with `_ALT` can be abbreviated by dropping the `_ALT`.

7. Efficiently Writing a Sequence of Bit Fields

Suppose you are configuring a device whose register map has the format given in the following format structure:

```
class Fmt : private Bitfield_format
  {
  public:
```

```
    F<5> f1;
    F<15> f2;
    F<8> f3;
    F<18> f4;
    F<22> f5;
    F<4> f6;
    F<8> f7;
  };
```

Suppose that the value type is `uint32_t`, the storage type is `uint16_t`, and the storage access type is `My_sa_t`, and `Fmt_offset_from_start` is `true`. To configure the device, we have to write fields `f2`, `f3`, `f5`, and `f6`. Fields `f1`, `f4`, and `f7` are don't-cares, it does not matter what is or isn't written to them. We could accomplish the configuration with the following code:

```
struct My_traits : public Bitfield_traits_default<uint32_t, uint16_t>
  {
    typedef My_sa_t Storage_access_t;
  };

typedef Bitfield_w_fmt<My_traits, Fmt> Bwf;

My_sa_t my_sa;

BITF(Bwf, my_sa, f2) = 5;
BITF(Bwf, my_sa, f3) = 10;
BITF(Bwf, my_sa, f5) = 15;
BITF(Bwf, my_sa, f6) = 13;
```

This would result in the following sequence of reads and writes to the device:

Read storage location (at offset) 0
Write storage location 0
Read storage location 1
Write storage location 1
Read storage location 1
Write storage location 1
Read storage location 2
Write storage location 2
Read storage location 3
Write storage location 3
Read storage location 4
Write storage location 4
Read storage location 4
Write storage location 4

The issue is that accesses to device registers are generally much slower than conventional memory accesses. So one could be tempted to instead write lower-level, error-prone, bulky, harder-to-port code that would configure the device using just 5 writes, one to each of the device's storage locations.

A third alternative is to use additional facilities in bitfield.h, like this:

```
struct My_traits : public Bitfield_traits_default<uint32_t, uint16_t>
```

```
  {
    typedef Bitfield_seq_storage_write_t<My_sa_t> Storage_access_t;
  };

typedef Bitfield_w_fmt<My_traits, Fmt> Bwf;

My_sa_t my_sa_;

{
  Bitfield_seq_storage_write_buf<My_sa_t> my_sa(my_sa_);

  BITF(Bwf, my_sa, f2) = 5;
  BITF(Bwf, my_sa, f3) = 10;
  BITF(Bwf, my_sa, f5) = 15;
  BITF(Bwf, my_sa, f6) = 13;
}
```

This code will also configure the device with just 5 writes, one to each of the device's storage locations. The first parameter to the `Bitfield_seq_storage_write_t` and `Bitfield_seq_storage_write_buf` templates is a valid storage access type, with one additional requirement: it must provide `Storage_t` as a public typedef. A `Bitfield_seq_storage_write_t` instance can be implicitly constructed from an instance of `Bitfield_seq_storage_write_t`. The fields in the sequence must be written in order of ascending offset of the first bit in each field. In this example, it's important that `my_sa` is destroyed after the last field is assigned, as its destructor will do a write if there is still buffered data that needs to be flushed. `Bitfield_seq_storage_write_buf` has a member function `flush()` (with void return value) that will flush any buffered data. It also has a member function `discard()` (no return value) that will discard an buffered data.

The `Bitfield_seq_storage_write_t` and `Bitfield_seq_storage_write_buf` templates accept a second parameter, which defaults to:

```
class Bitfield_seq_storage_write_default_traits
  {
  public:

    static const bool Flush_on_destroy = true;

    static const unsigned First_bit = 0;

    static const unsigned End_bit = ~unsigned(0);

    static void handle_backwards(
      unsigned /* previous_offset */, unsigned /* next_offset */)
      { }
  };
```

`First_bit` should be `0` if the first field to be written starts at a storage location boundary, or if all the bit at lower bit offsets between it and the storage location boundary are don't-cares. Otherwise, `First_bit` should be bit offset from the preceding storage location boundary to the start of the first field to write. `End_bit` should be `~unsigned(0)` if the highest-offset bit in the last field to write precedes a storage location boundary, or all the bits between this bit and the next storage location boundary are don't-cares. Otherwise, `End_bit` should be one more than the offset of the highest-offset bit in the last field to write.

In the example above suppose instead that the fields `f1` and `f7` were *not* don't-cares, that is to say, their

values had to be preserved. Then it would be necessary to change the code to:

```
typedef Bitfield_w_fmt<Bitfield_traits_default<uint32_t, uint16_t>, Fmt>
Bwf_def;

struct My_ssw_traits : public Bitfield_seq_storage_write_default_traits
  {
    static const unsigned First_bit = BITF_OFFSET(Bwf_def, f1);

    static const unsigned End_bit = BITF_OFFSET(Bwf_def, f7);
  };

struct My_traits : public Bitfield_traits_default<uint32_t, uint16_t>
  {
    typedef Bitfield_seq_storage_write_t<My_sa_t, My_ssw_traits>
Storage_access_t;
  };

typedef Bitfield_w_fmt<My_traits, Fmt> Bwf;

My_sa_t my_sa_;

{
  Bitfield_seq_storage_write_buf<My_sa_t, My_ssw_traits> my_sa(my_sa_);

  BITF(Bwf, my_sa, f2) = 5;
  BITF(Bwf, my_sa, f3) = 10;
  BITF(Bwf, my_sa, f5) = 15;
  BITF(Bwf, my_sa, f6) = 13;
}
```

The sequence of storage location accesses needed to configure the device would now be:

Read storage location 0
Write storage location 0
Write storage location 1
Write storage location 2
Write storage location 3
Read storage location 4
Write storage location 4

If `Flush_on_destroy` is changed to `false`, buffered data will no longer be flushed when the `Bitfield_seq_storage_write_buf` is destroyed. It would then be necessary for the code to call the `flush()` member function after setting the last field in the sequence to ensure the final write occurred.

The `handle_backwards` member function (which must be static) is called if the code erroneously sets the bit fields out of offset order. The first parameter is the offset of the buffered storage location. The second parameter is the offset of the storage location that is less than the offset of the buffered location, and is erroneously being accessed. This function, as shown, does nothing by default. It can throw an exception, log the error, do a reset or exit, etc.

8. Object Code Observations

These observations are based on assembler output from GNU G++, with a 64-bit x86 target.

When the bit field offset and bit field are constants known at compile time, the number of instructions generated using this facility and base language bit fields seem to be very comparable (with optimization enabled).

If either the offset or the width are determined by variables (which are not resolved to constants by optimization), inlined bit field operations result in hundreds of machine instructions.

The C++ Standard allows the compiler to make inlined functions effectively out-of-line. One would suspect that the compiler, seeing that an instantiation of `Bitfield::Bf::f()` for variable parameters was called multiple times in the code, would make it out-of-line. But this does not seem to happen. This facility was designed for use with format structures, which generally will result in constant field offsets and widths. An exception is when the bit field is an element of an array, and the array index is a variable. Another exception is when the bit field is specified as relative to an offset, and that offset is a variable.

This facility provides the ability to bypass validation of value bit precision (when changing a bit field value). But validation of bit field width (to be non-zero and not exceed value type precision) cannot be bypassed. My thinking was that bit field widths typically resolve to constants at compile time, which will be valid. Thus (due to inlining), the compiler will typically not generate any object code to validate the width at run time.

I have assumed that efforts to propagate `noexept` or empty throw modifiers from the traits storage access class's member functions to `Bitfield::Bf` member functions are unnecessary, based on https://waltsgeekblog.quora.com/G++-inline-and-noexcept . I expect that the compiler will not generate unnecessary exception handling code.

9. C Language Counterpart

I have written a similar facility to this that can be used in straight C. It is implemented using macros. It is proprietary to Nokia Networks. I put a copy of it in the Alcatel-Lucent ACOS Forge (no access from outside the Nokia intranet).