*Section 11*

*Bit Operations*

NOTE 11.1

## Internal Representation of Integer Types

**Positive/Unsigned Values**

The internal computer representation of positive integer types is normally binary, wherein each bit position has a power of 2 value associated with it. The decimal equivalent of the binary number is determined by adding the values of each position containing a 1. If a **signed** type is being represented, the most significant bit is used as a sign bit rather than a value (0 => positive and 1 => negative).

**8 Bit Binary Representation Of The Decimal Value 62 (hex 0x3e, octal 076)**

$2^7$ **is Sign or Value Bit**

| $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | power of 2 of each position |
|---|---|---|---|---|---|---|---|---|
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | decimal value of each position |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | binary representation of $62_{10}$ |

**Negative Values**

Signed magnitude is the format normally used by humans represent numbers, but it is not practical for computer use. The most common internal computer representations of negative integer values are known as binary 1's and binary 2's complements, with 2's being the most common by far. To find these representations, one need only express the number in binary as if it were a **signed** positive number, then take the 1's or 2's complement of that representation. In both complement methods, re-complementing returns to the original number.

**Signed Magnitude**

To negate a number, simply complement the sign bit. In signed magnitude arithmetic:

- Both a positive zero (all 0s) and a negative zero (all 0s except sign bit) exist;
- Extra hardware is required to detect both 0s;
- Extra hardware and extra time are required to process negative values.

**1's Complement**

To obtain the 1's complement of any binary number, simply change all 1s to 0s and all 0s to 1s. In 1's complement arithmetic:

- Both a positive zero (all 0s) and a negative zero (all 1s) exist;
- Extra hardware is required to detect both 0s;
- Extra hardware and extra time are required during subtraction.

**2's Complement**

There are two ways to obtain the 2's complement of a binary number:

1. Take the 1's complement of the number then add 1 to the LSD, or
2. Moving right-to-left, leave unaltered all initial 0s and the first 1, then take the 1's complement the remaining bits.

In 2's complement arithmetic:

- There is only one zero, represented by all 0s. By definition, it is its own 2's complement;
- There is always one more negative number than there are positive numbers (because one of the positive numbers is zero). The most negative number is represented by a 1 in the sign bit and 0s in all other bits. The 2's complement of this number cannot be taken since there is no corresponding positive number.

**8 Bit Signed Magnitude, 1's Complement, and 2's Complement Representations Of The Decimal Value −62 (hex −0x3e, octal −076)**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | binary representation of +62 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | signed magnitude representation of −62 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1's complement representation of −62 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 2's complement representation of −62 |

NOTE 11.2

Integer Conversions – Promotions and Demotions

**Signed and Unsigned Integers**

1 “When a value with integer type is converted to another integer type other than **_Bool**, if the value can be represented by the new type, it is unchanged.” (ISO/IEC 9899:2011 section 6.3.1.3.1)

Example of conversion to “wider” integer type:

a. If the original value is positive and the new data type has more bits, the new value is created by filling with zero bits on the left up to the new width. For example the signed or unsigned 8-bit value $00111010_2$ becomes $00000000\ 00111010_2$ when converted to a 16 bit integer type.
b. If the original value is negative and the new data type has more bits, the new value is created in a way dependent upon how negative numbers are represented. For the most common schemes (1's or 2's complement) the new value is created by filling with one bits on the left up to the new width. For example the signed 8-bit value $10111010_2$ becomes $11111111\ 10111010_2$ when converted to a 16 bit integer type.

2 “Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.” (ISO/IEC 9899:2011 section 6.3.1.3.2)

3 “Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.” (ISO/IEC 9899:2011 section 6.3.1.3.3)

Bitwise Operators

Bitwise operators treat their integer operands as groups of individual binary bits rather than as single arithmetic values, thereby permitting selected bits to be independently set to 1s, cleared to 0s, or tested. Bitwise operations are most often encountered in embedded applications where hardware registers are mapped into the CPU address space and accessed using pointers to those addresses. Such registers frequently contain “fields” of one or more bits whose functions are unrelated to the functions of adjacent fields and must, therefore, be manipulated without disturbing the contents of the adjacent fields. For example, consider the following communications controller register whose bits are used as indicated:

$2^{15}$ $2^0$

| 15 - 11 | 10 - 6 | 5 – 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|
| unused | Protocol Code | Bit Rate | Stop Bits | Parity Type | Check Parity |

**Definition:** A *mask* is an expression used as a pattern to set, clear, test, or extract specific bits from another expression.

In most bitwise operations involving two operands, one of those operands is treated as a *mask* and contains 1s (or sometimes 0s) in the bit positions of interest.

**signed/unsigned** operands

Although bitwise operations work equally well on **signed** and **unsigned** types, mixing the two can result in subtle portability problems associated with **signed** to **unsigned** conversions. Using only **unsigned** types (**unsigned char**, **unsigned short**, **unsigned int**, **unsigned long**, **unsigned long long**) usually provides the fewest number of “surprises”.

 

NOTE 11.3

## The ~ Bitwise 1's Complement Operator

The single integer operand of the unary bitwise 1's complement operator first undergoes integer promotion. A value is then produced that is the 1's complement of the promoted operand (regardless of how the implementation represents negative integer values). This operator is often used in masking operations and to ensure portable, type-width independent code.

For the following illustrations arbitrarily assume that all variables are 16 bit type **unsigned**.

• The basic 1's complement operation:

```
x = 0x5dadu;       /* 0101 1101 1010 1101   (0x5dadu)   into x */
y = ~x;            /* 1010 0010 0101 0010   (0xa252u)   into y */
x = ~x;            /* 1010 0010 0101 0010   (0xa252u)   into x */
x = ~0u;           /* 1111 1111 1111 1111   (~0u)   into x */
x = ~~0u;          /* 0000 0000 0000 0000   (~~0u)   into x */
```

• Setting *x* to a pattern with all bits set:

The following succeeds if *x* is 16 bits wide:

```
x = 0xffffu;       /* 1111 1111 1111 1111   (0xffffu)   into x */
```

...but fails if *x* is more than 16 bits wide:

```
x = 0xffffu;       /* ......0000 1111 1111 1111 1111   (0xffffu)   into x */
```

The following always succeeds because the compiler determines the width of the right operand:

```
x = ~0u;           /* ......1111 1111 1111 1111 1111   (~0u)   into x */
```

Similarly, **~0uL** to produce a type **unsigned long** value with all bits set.

• Setting *x* to a pattern with bits $2^{15}$, $2^{1}$, and $2^{0}$ cleared and all other bits set:

The following succeeds if *x* is 16 bits wide:

```
x = 0x7ffcu;       /* 0111 1111 1111 1100   (0x7ffcu)   into x */
```

...but fails if *x* is more than 16 bits wide (bits $2^{16}$ and beyond get cleared):

```
x = 0x7ffcu;       /* ......0000 0111 1111 1111 1100   (0x7ffcu)   into x */
```

The following always succeeds because the compiler determines the width of the right operand:

- Create a value having 1s in the bit positions to be cleared and 0s elsewhere
  0x8003**u** (......0000 1000 0000 0000 0011 binary) is that value
- Use the 1's complement of that value

```
x = ~0x8003u;      /* ......1111 0111 1111 1111 1100   (~0x8003u)   into x */
```

NOTE 11.4

## The **&** Bitwise AND Operator

The two integer operands of the bitwise AND operator first undergo the usual arithmetic conversions. Then the corresponding bits in each are compared and a new value is produced that contains a 1 in every bit position in which *both* operands contain a 1. A 0 is produced for all other bit positions. The bitwise AND operator is commonly used to:

- clear selected bits;
- test the state of selected bits.

The following illustrations arbitrarily assume that all variables are 16 bit type **unsigned**. Although all mask values are shown as constants, they may be represented by any expression of the appropriate type.

- The basic AND operation:

```
                         /* 0101 1101 1010 1101  (0x5dadu)  value of one operand */
                         /* 1011 1010 0100 0111  (0xba47u)  value of other operand */
x = 0x5dadu & 0xba47u;   /* 0001 1000 0000 0101  (0x1805u)  result of & into x */
```

- Clearing selected bits:

PROBLEM: Produce a value having 0s in bit positions 15, 14, 13, 12, 7, 1, and 0, and having the same bit pattern as some arbitrary integer expression in all other bit positions.

SOLUTION: Bitwise AND the arbitrary expression with the 1's complement of a mask containing:
- a 1 in each bit position where a 0 is desired;
- a 0 in each bit position where the bit value of the arbitrary expression is to be reproduced.

```
x = 0x5dadu;             /* 0101 1101 1010 1101  (0x5dadu)  arbitrary value */
                         /* 1111 0000 1000 0011  (0xf083u)  value of mask */
x & ~0xf083u;            /* 0000 1101 0010 1100  (0x0d2cu)  no side effects */
x &= ~0xf083u;           /* 0000 1101 0010 1100  (0x0d2cu)  update x */
x &   0x0f7cu;           /* 0000 1101 0010 1100  (0x0d2cu)  no side effects (problem here?) */
x &= 0x0f7cu;            /* 0000 1101 0010 1100  (0x0d2cu)  update x (problem here?) */
```

- Testing for set bits:

PROBLEM: Determine if there are 1s in bit positions 15, 14, 13, 12, 7, 1, and 0 of some arbitrary integer expression. Ignore the other bit positions.

SOLUTION: Bitwise AND the arbitrary expression with a mask. If the result equals the mask value, the bits of interest are indeed 1s. The mask must contain:
- a 1 in each bit position of interest;
- a 0 in each bit position to be ignored.

```
x = 0x5dadu;                  /* 0101 1101 1010 1101  (0x5dadu)  arbitrary value */
                              /* 1111 0000 1000 0011  (0xf083u)  value of mask */
if ((x & 0xf083u) == 0xf083u) /* TRUE if all 1 bits in the mask are also set in x */
```

- Testing for odd/even:

PROBLEM: Determine if the value of an arbitrary integer expression is odd or even.

SOLUTION: Bit $2^0$ will be set for all odd positive integer values but will be implementation-defined for negative values. Therefore, bitwise AND the positive equivalent of the arbitrary expression with a mask in which only bit $2^0$ is set.

```
if (x & 1u)                /* TRUE if arbitrary positive integer expression x is odd but...
                              ...result is implementation-defined for negative values. */
if ((x < 0 ? -x : x) & 1u) /* TRUE if arbitrary integer expression x is odd.  Works for...
                              ...negative values also and is implementation-independent. */
```

NOTE 11.5

## The | Bitwise OR Operator

The two integer operands of the bitwise OR operator first undergo the usual arithmetic conversions. Then the corresponding bits in each are compared and a new value is produced that contains a 1 in every bit position in which *either or both* operands contain a 1. A 0 is produced for all other bit positions. The bitwise OR operator is commonly used to:

- set selected bits.

The following illustrations arbitrarily assume that all variables are 16 bit type **unsigned**. Although all mask values are shown as constants, they may be represented by any expression of the appropriate type.

- The basic OR operation:

```
                               /* 0101 1101 1010 1101   (0x5dadu)   value of one operand */
                               /* 1011 1010 0100 0111   (0xba47u)   value of other operand */
    x = 0x5dadu | 0xba47u;     /* 1111 1111 1110 1111   (0xffefu)   result of | into x */
```

- Setting selected bits:

PROBLEM: Produce a value having 1s in bit positions 15, 14, 13, 12, 7, 1, and 0, and having the same bit pattern as some arbitrary integer expression in all other bit positions.

SOLUTION: Bitwise OR the arbitrary expression with a mask containing:

- a 1 in each bit position where a 1 is desired;
- a 0 in each bit position where the bit value of the arbitrary expression is to be reproduced.

```
    x = 0x5dadu;               /* 0101 1101 1010 1101   (0x5dadu)   arbitrary value */
                               /* 1111 0000 1000 0011   (0xf083u)   value of mask */
    x |   0xf083u;             /* 1111 1101 1010 1111   (0xfdafu)   no side effects */
    x |= 0xf083u;              /* 1111 1101 1010 1111   (0xfdafu)   update x */
```

 

NOTE 11.6

## The ^ Bitwise Exclusive-OR Operator

The two integer operands of the bitwise exclusive-OR operator first undergo the usual arithmetic conversions. Then the corresponding bits in each are compared and a new value is produced that contains a 1 in every bit position in which *the bits differ*. A 0 is produced for all other bit positions. This leads to the obvious observations that any bit exclusive-OR'ed with:

- a 0 produces that same bit: 1 ^ 0 is 1 and 0 ^ 0 is 0
- a 1 produces the complement of that bit: 1 ^ 1 is 0 and 0 ^ 1 is 1
- itself produces a 0 bit: 1 ^ 1 is 0 and 0 ^ 0 is 0
- its complement produces a 1 bit: 1 ^ 0 is 1 and 0 ^ 1 is 1

The bitwise exclusive-OR operator is of limited use compared to the bitwise AND and OR operators and is primarily used to:

- 1's complement selected bits.

The following illustrations arbitrarily assume that all variables are 16 bit type **unsigned**. Although all mask values are shown as constants, they may be represented by any expression of the appropriate type.

- The basic exclusive-OR operation:

```
                              /* 0101 1101 1010 1101   (0x5dadu)   value of one operand */
                              /* 1011 1010 0100 0111   (0xba47u)   value of other operand */
x = 0x5dadu ^ 0xba47u;        /* 1110 0111 1110 1010   (0xe7eau)   result of  ^  into x */
```

- Producing the 1's complement of selected bits:

PROBLEM: In bit positions 15, 14, 13, 12, 7, 1, and 0, produce the 1's complement of the bits in some arbitrary integer expression. In all other positions preserve the bits in that expression.

SOLUTION: Bitwise exclusive-OR the arbitrary expression with a mask containing:
- a 1 in each bit position where a complement is desired;
- a 0 in each bit position where the bit value of the arbitrary expression is to be reproduced.

```
x = 0x5dadu;                  /* 0101 1101 1010 1101   (0x5dadu)   arbitrary value */
                              /* 1111 0000 1000 0011   (0xf083u)   value of mask */
x ^   0xf083u;                /* 1010 1101 0010 1110   (0xad2eu)   no side effects */
x ^= 0xf083u;                 /* 1010 1101 0010 1110   (0xad2eu)   update x */
```

NOTE: To produce the 1's complement of an entire expression *don't* use the exclusive-OR operator at all but instead use the unary ~ 1's complement operator.

- Exchanging two integer values without a temporary variable:

With a temporary variable --- exchange *x* and *y* --- (temp = x; x = y; y = temp;)

Without a temporary variable --- using bitwise exclusive-OR --- exchange *x* and *y*

```
y = 0xad2eu;       /*                                 y  is  1010 1101 0010 1110 */
x = 0x5dadu;       /*                                 x  is  0101 1101 1010 1101 */
x ^= y;            /* x  gets  1111 0000 1000 0011,   y  is  1010 1101 0010 1110 */
y ^= x;            /* y  gets  0101 1101 1010 1101,   x  is  1111 0000 1000 0011 */
x ^= y;            /* x  gets  1010 1101 0010 1110,   y  is  0101 1101 1010 1101 */
```

NOTE 11.7

## The << and >> Bit Shift Operators

The bit shift operators provide a way of moving the entire group of bits comprising an integer value to the left or right by a specified number of bit positions. These operators implement a shift operation, not a rotate, meaning that any bits shifted off either end are lost. Shift operations are commonly used to:

- position bit patterns to specific locations for interfacing with embedded hardware such as control and status registers in peripheral devices;

- efficiently multiply or divide positive values by a power of 2.

Both shift operands must be integer values. They undergo the usual *unary* conversions (integer promotion) and the expression data type is that of the left operand after promotion. If the value of the right operand (the shift count) is negative or is greater than the width of the promoted left operand, the behavior is undefined.

**Left Shift** syntax: expr1 << expr2

The left shift operator produces a value whose bit pattern is that of *expr1* shifted left by the number of bit positions specified by *expr2*. 0s are always shifted in from the right:

```
int shiftCount = 2;
unsigned  x = 0x7a17;        /* 0111 1010 0001 0111   (0x7a17u)   31255 decimal */

x << shiftCount              /* 1110 1000 0101 1100   (0xe85cu)   59484 decimal   no side effects */
x <<= shiftCount             /* 1110 1000 0101 1100   (0xe85cu)   59484 decimal   update x */
```

For each bit position that a positive value is left shifted, its value is multiplied by two provided the data type can represent the new value. Shifts are typically more efficient than multiplications:

**Since $6236_{10}$ is $185C_{16}$:** /* 0001 1000 0101 1100 (0x185c**u**) 6236 decimal */

```
6236 << 1                    /* 0011 0000 1011 1000   (0x30b8u)   12472 decimal (6236 * 2) */
6236 << 2                    /* 0110 0001 0111 0000   (0x6170u)   24944 decimal (6236 * 4) */
6236 << 3                    /* 1100 0010 1110 0000   (0xc2e0u)   49888 decimal (6236 * 8) */
6236 << 4                    /* 1000 0101 1100 0000   (0x85c0u)   34240 decimal (Overflow!) */
```

**Right Shift** syntax: expr1 >> expr2

The right shift operator produces a value whose bit pattern is that of *expr1* shifted right by the number of bit positions specified by *expr2*. 0s are shifted in from the left end if *expr1* is positive, but if negative it is implementation-defined whether 0s or 1s are shifted in:

```
int shiftCount = 2;
unsigned  x = 0x7a17;        /* 0111 1010 0001 0111   (0x7a17)    31255 decimal */

x >> shiftCount;             /* 0001 1110 1000 0101   (0x1e85u)    7813 decimal   no side effects */
x >>= shiftCount;            /* 0001 1110 1000 0101   (0x1e85u)    7813 decimal   update x */
```

For each bit position a positive or unsigned value is right shifted its value is divided by two. For negative values the results are implementation dependent. Shifts are typically more efficient than divisions:

**Since $6236_{10}$ is $185C_{16}$:** /* 0001 1000 0101 1100 (0x185c**u**) 6236 decimal */

```
6236 >> 1                    /* 0000 1100 0010 1110   (0x30b8u)   3118 decimal (6236 / 2) */
6236 >> 2                    /* 0000 0110 0001 0111   (0x6170u)   1559 decimal (6236 / 4) */
6236 >> 3                    /* 0000 0011 0000 1011   (0x030bu)   779 decimal (6236 / 8) */
6236 >> 4                    /* 0000 0001 1000 0101   (0x185bu)   389 decimal (6236 / 16) */
```

 

NOTE 11.8

## Using Bitwise Operators

Bitwise operators treat their integer operands as combinations of individual binary bits rather than as the numeric values the bits may represent. While some bitwise operations may provide a more efficient alternative to mathematical operations such as multiplication and division, they are most often used when writing programs that must interface to hardware registers in which various bits must be manipulated independently of each other.

As an illustration, assume an I/O device is configured in such a way that we can access its control register as a 16 bit **unsigned int** at an absolute hardware memory address, that is, by using an **unsigned** pointer. If the register is configured with the following independent “fields”, we must be able to read or write any field of bits without disturbing any others.

$2^{15}$ $2^{0}$

| 15 - 11 | 10 - 6 | 5 – 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|
| unused | Protocol Code | Bit Rate | Stop Bits | Parity Type | Check Parity |

| | | |
|---|---|---|
| Check Parity | (1 bit flag) | 1 => check parity, 0 => don't check parity |
| Parity Type | (1 bit flag) | 1 => odd parity, 0 => even parity |
| Stop Bits | (1 bit flag) | 1 => 2 stop bits, 0 => 1 stop bit |
| Bit Rate | (3 bit code) | select 1 of 8 possible transmission rates |
| Protocol Code | (5 bit code) | select 1 of 32 possible protocols |
| unused | (5 bits) | writing has no effect, read value is undefined |
| | ————————— | |
| | 16 bits total | |

The following illustration provides some examples of reading and writing various fields of bits without disturbing the others. By using “bit-fields”, to be discussed later, this same program can be written in a much more programmer-friendly way.

```
/* define a constant mask for each "field" of bits */

#define CHECK_PARITY     0x01u      /* 0000 0000 0000 0001  (bit 2^0) */
#define PARITY_TYPE      0x02u      /* 0000 0000 0000 0010  (bit 2^1) */
#define STOP_BITS        0x04u      /* 0000 0000 0000 0100  (bit 2^2) */
#define BIT_RATE         0x38u      /* 0000 0000 0011 1000  (bits 2^5–2^3) */
#define PROTOCOL_CODE    0x07c0u    /* 0000 0111 1100 0000  (bits 2^10–2^6) */
#define UNUSED           0xf800u    /* 1111 1000 0000 0000  (bits 2^15–2^11) */

void ManipulateFields(void)
{
    unsigned code, *registerAddress = (unsigned *)0x1234;   /* I/O device is at memory address 0x1234 */

    /* Only one "field" is manipulated by each of the following statements */

    *registerAddress |= CHECK_PARITY;                           /* set the check parity flag */
        Same as:  *registerAddress = *registerAddress | CHECK_PARITY;

    *registerAddress &= ~STOP_BITS;                             /* clear the stop bits flag */
        Same as:  *registerAddress = *registerAddress & ~STOP_BITS;

    *registerAddress = (*registerAddress & ~BIT_RATE) | (4u << 3);    /* set the bit rate to 4 */
    code = (*registerAddress & PROTOCOL_CODE) >> 6;             /* read the protocol code */
}
```

NOTE 11.9A

## Bit-fields

Bit-fields are **int**, **unsigned int**, or **signed int** members of structures, classes, or unions that are declared to represent a specific number of bits. Some implementations also permit other integer/Boolean/enumeration bit-field types. Bit-fields are primarily used to:

- access I/O devices that require bits in particular positions;
- eliminate the need for using bitwise operators in some situations;
- conserve storage by packing small pieces of data into common allocation units.

### Bit-field Syntax

```
struct Tag   (or class Tag or union Tag)
{
        unsigned control:4;            /* field 4 bits wide named control */
        unsigned   :5;                 /* skip an unused field 5 bits wide */
        unsigned status:2;             /* field 2 bits wide named status */
        unsigned   :0;                 /* skip remainder of bits in this allocation unit */
        unsigned feedback:13;          /* field 13 bits wide named feedback in next allocation unit */
        ...
};
```

### Bit-field Characteristics

- Non-portable: Some implementations allocate bit-fields right-to-left while others allocate left-to-right. The order of allocation is not necessarily related to the “endian” architecture of the machine;
- Inefficient: Since the underlying mechanism can involve shifting and masking, bit-fields should not be used to conserve storage unless that is more important than speed;
- Restricted Width: A field cannot be wider than its type or portably straddle allocation units;
- No Arrays: There can be no arrays of bit-fields, that is, no *name:2[6]*;
- No Addresses: The address of a bit-field cannot be taken;
- “Signedness”: The signedness of types declared without **signed** or **unsigned** is implementation dependent.

Assume we wish to access an I/O device that has the following bit definitions in a 16-bit word, and we need to be able to read or write any “field” of bits without disturbing any others.

$2^{15}$ $2^{0}$

| 15 - 11 | 10 - 6 | 5 – 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|
| unused | Protocol Code | Bit Rate | Stop Bits | Parity Type | Check Parity |

| | | |
|---|---|---|
| Check Parity | (1 bit flag) | 1 => check parity, 0 => don't check parity |
| Parity Type | (1 bit flag) | 1 => odd parity, 0 => even parity |
| Stop Bits | (1 bit flag) | 1 => 2 stop bits, 0 => 1 stop bit |
| Bit Rate | (3 bit code) | select 1 of 8 possible transmission rates |
| Protocol Code | (5 bit code) | select 1 of 32 possible protocols |
| unused | (5 bits) | writing has no effect, read value is undefined |
| | ————— | |
| | 16 bits total | |

...............CONTINUED

NOTE 11.9B **...............CONTINUATION**

Bit-fields, Cont'd.

Assume that the 16-bit register shown on the previous page, which reads as written, has been hard wired into memory address 0x1234 on a machine that uses 16-bit **int**s. Assume also that the compiler will not try to use address 0x1234 for its own purposes.

```
/*  Version 1 - Without bit-fields  -  repeated from a previous example */

/* define a mask for each "field" of bits – use const int declarations in C++ */
#define CHECK_PARITY       0x01u                  /* 0000 0000 0000 0001  (bit 2^0) */
#define PARITY_TYPE        0x02u                  /* 0000 0000 0000 0010  (bit 2^1) */
#define STOP_BITS          0x04u                  /* 0000 0000 0000 0100  (bit 2^2) */
#define BIT_RATE           0x38u                  /* 0000 0000 0011 1000  (bits 2^5–2^3) */
#define PROTOCOL_CODE      0x07c0u                /* 0000 0111 1100 0000  (bits 2^10–2^6) */
#define UNUSED             0xf800u                /* 1111 1000 0000 0000  (bits 2^15–2^11) */

void ManipulateFields(void)
{
        unsigned code, *registerAddress = (unsigned *)0x1234;   /* I/O device is at memory address 0x1234 */

        /* Only one "field" is manipulated by each of the following statements */

        *registerAddress |= CHECK_PARITY;                                   /* set the check parity flag */
        *registerAddress &= ~STOP_BITS;                                     /* clear the stop bits flag */
        *registerAddress = (*registerAddress & ~BIT_RATE) | (4u << 3);      /* set the bit rate to 4 */
        code = (*registerAddress & PROTOCOL_CODE) >> 6;                     /* read the protocol code */
}



/*  Version 2 - Uses bit-fields  -  Assumes right-to-left field allocation */
typedef struct                                    /* structure containing bit-fields */
{
        unsigned checkParity:1;
        unsigned parityType:1;
        unsigned stopBits:1;
        unsigned bitRate:3;
        unsigned protocolCode:5;
} MESSAGE;

void ManipulateFields(void)
{
        unsigned code;
        MESSAGE *registerAddress = (MESSAGE *)0x1234;      /* I/O device is at memory address 0x1234 */

        /* Only one "field" is manipulated by each of the following statements */

        registerAddress->checkParity = 1u;                 /* set the check parity flag */
        registerAddress->stopBits = 0u;                    /* clear the stop bits flag */
        registerAddress->bitRate = 4u;                     /* set the bit rate to 4 */
        code = registerAddress->protocolCode;              /* read the protocol code */
}
```

## Section 11 Practice Quiz (not for submission or grading)

This is a theoretical "paper only" quiz in which you must assume a perfectly implemented ANSI/ISO C/C++ compiler. How any particular program runs on your computer only indicates how your computer runs that program and not necessarily how it should run or how portable it is.

1. Predict the output from:
   *cout << hex << ~0x3a5c;*
   A. 0x3a5c
   B. 3a5c
   C. ffffc5a3
   D. c5a3
   E. The output is implementation dependent.

2. Predict the output assuming 8 bit **char**s, 16 bit **int**s, 32 bit **long**s, and two's complement. (The character between the *%* and the *x* in the first three conversion specifications is the letter *ell*.)
   ```
   int z = 0x7fff;
   printf("%lx %lx %lx %x %x",
         (long)(z << 4),
         (long)((long)z << 4),
         (long)(z >> 4),
         z ^ 0xaa,
         (z >> 4) << 4 );
   ```
   A. fffffff0  fffffff0  ffffffff  7f55  7ff0
   B. 7fff0  7fff0  7ff  7f55  7ff0
   C. fffffff0  fffffff0  7ff  7f55  7ff0
   D. fffffff0  7fff0  7ff  7f55  7ff0
   E. 7ffffff0  7fff0  7fff  7faa  7ff

3. Predict the output from:
   *cout << dec << (−2 >> 1);*
   A. −1
   B. −2
   C. 32767
   D. 2147483647
   E. The output is implementation dependent.

4. Assuming two's complement, predict the output from: *printf("%d", −2 << 1);*
   A. −1
   B. −2
   C. −4
   D. −8
   E. The output is implementation dependent.

5. Assuming 16 bit **int**s and two's complement, predict the output from:
   ```
   typedef struct
   {
         unsigned controlBit:1;
   } MESSAGE;
   unsigned fakeRegister = 0u;
   MESSAGE *regAdr =
         (MESSAGE *)&fakeRegister;
   regAdr−>controlBit = 1u;
   printf("%x", fakeRegister);
   ```
   A. 8000 *or* 1
   B. 8000
   C. 1
   D. 6684165
   E. Some other implementation dependent value.

6. On a machine using 1's complement negative integers and 16 bit **int**s, what is the bit pattern for −2?
   A. 1111 1111 1111 1111
   B. 1111 1111 1111 1110
   C. 1111 1111 1111 1101
   D. 1000 0000 0000 0010
   E. implementation dependent

7. If an **int** is 16 bits and a **char** is 8 bits, the values in *sch* and *uch* after ***signed char*** *sch = 256;* and ***unsigned char*** *uch = 256*; are:
   A. *sch* is 256 and *uch* is 256
   B. *sch* is implementation defined and *uch* is 256
   C. *sch* is implementation defined and *uch* is 0
   D. *sch* is 0 and *uch* is 0
   E. The results of both are undefined.

8. Assuming a 16 bit **int** and 2's complement, predict the value of *−17 >> 1*
   A. −9 or 0x7FF7
   B. −8
   C. 17
   D. 8
   E. other implementation dependent values

## Section 11 Practice Exercises (not for submission or grading)

11-1. Define and test macro *ShiftPattern* that takes two arguments of any integer type and produces the value of the first argument shifted (not rotated) by the number of bit positions specified by the second argument. If the second argument is positive, a right shift will occur while a negative argument produces a left shift. Do not concern yourself with whether the right shift is arithmetic or logical.

11-2. The number of bits in type **char** (1 **char** = 1 byte) is an implementation-dependent value greater than or equal to 8 while the number of bits in type **int** is an implementation-dependent value greater than or equal to 16. Write both a macro and a function version of ***int*** *CountIntBits(**void**);* that return the number of bits in type **int** on <u>any and every implementation</u> on which they are called. Right-shifts are not permitted and the function version may not use any information from <limits.h>/<climits>. Your macro/function may be used in the exercises that follow.

11-3. Write a function named *RotatePattern* with the following syntax:

**unsigned** RotatePattern(**unsigned** patternToRotate, **int** count);

*RotatePattern* will rotate the bits in *patternToRotate* to the right or left by the number of bit positions specified by *count* and return the resulting bit pattern. If *count* is positive the bits will rotate to the right while a negative value of *count* will cause a left rotation.

The definition of a bit rotation requires that the least significant bit (lsb) and the most significant bit (msb) of a value be treated as if they are connected. That is, when a bit gets right shifted out of the lsb, it gets shifted into the msb rather than getting thrown away. Conversely, when a bit gets left shifted out of the msb, it gets shifted into the lsb rather than getting thrown away. For example, if a 16 bit **unsigned** with a value of *0xA701* is rotated one position to the right, the resultant value will be *0xD380*.

In order to operate correctly on any machine, your function must make <u>no assumptions</u> about the number of bits in the data type of *patternToRotate*. Of course, rotating on a machine with 16 bit integers may yield a different, but equally correct result from rotating on a machine with 32 bit integers. Make sure your function can handle values of *count* that are greater than the number of bits in the data type of *patternToRotate*. Use the following 4 cases as well as some of your own to test your function:

RotatePattern(0x5, 1); RotatePattern(0x5, −1); RotatePattern(0x5, 64); RotatePattern(0x8765, −64);

11-4. Write a function called *SearchForBitPattern* that looks for the occurrence of a specified pattern of bits inside an **int**. The function syntax is

**int** SearchForBitPattern(**int** source, **int** patternToFind, **int** bitsInPattern);

The function will search *source*, starting at the leftmost bit, to see if the rightmost *bitsInPattern* bits of *patternToFind* occur in *source*. If those bits are found, the function will return number of the bit at which they begin, where the leftmost bit is bit number 0. If they are not found, the function will return −1. For example, the call *SearchForBitPattern(0x70fa, 0x5, 3)* will cause the *SearchForBitPattern* function to search the number *0x70fA* ( 0111 0000 1111 1010 binary ) for the occurrence of the three-bit pattern *0x5* ( 101 binary ). The function would return 12 for 16 bit **int**s and 28 for 32 bit **int**s. In order to operate correctly on any machine, your function must make <u>no assumptions</u> about the number of bits in an **int**. Test your function with at least the following arguments:

0xe1f4, 0x1, 1    0xe1f4, 0xe1f4, 16    0xe1f4, 0x5, 3    0xe1f4, 0x5, 4