

## Chapter 2 - Embedded programming

### Section 2.1 - C in embedded systems

**C** is a popular programming language in embedded systems due to the language's simplicity and efficiency.

Table 2.1.1: Embedded systems programming languages.

| Language       | % of embedded programmers with current project mostly in this language (2013) |
|----------------|---|
| C              | 60%   |
| C++            | 21%   |
| Assembly       | 5%  |
| Java           | 3%  |
| C#             | 2%  |
| MATLAB/LabView | 4%  |
| Python         | 1%  |
| .NET           | 1%  |
| Other          | 4%  |

Source: [www.embedded.com](http://www.embedded.com), 2013 Embedded Market Study.

C was not originally created for embedded systems, but rather was created in 1972 for mainframe and desktop computers, which typically manipulate data in files such as integer or character data. In contrast, embedded systems commonly manipulate bits, in addition to other data types. This chapter describes C's built-in data types and discusses how to manipulate bits in C. Most of the discussion applies equally to the **C++** language, which added object-oriented concepts, especially useful in large programming projects .

The table below shows programming language usage for all types of systems. One sees that C and its variants still dominate. Many people argue that, because C does not hide many low-level details from a programmer, mastering C makes for a stronger programmer in any language.

Table 2.1.2: Worldwide programming language usage (not just embedded systems).

| Language    | "Popularity rating"<br>(see below source for definition) |
|-------------|--|
| C           | 18%  |
| Java        | 16%  |
| Objective-C | 10%  |
| C++         | 9%   |
| PHP         | 7%   |
| C#          | 6%   |
| Basic       | 4%   |
| Python      | 4%   |
| Other       | 4%   |

Source: [TIOBE Programming Community Index \(July 2013\)](#).



| # | Question  | Your answer |
|---|---|-------------|
| 1 | C and the related C++ language represented the main language for over 80% of embedded programmers (from 2013 survey). | True        |
|   |   | False       |
| 2 | C was originally designed for embedded systems.   | True        |
|   |   | False       |
| 3 | Assembly language is the main language for nearly half of embedded programmers.                                       | True        |
|   |   | False       |
| 4 | Fluency in C is rarely useful outside embedded systems.   | True        |
|   |   | False       |

Exploring further:

- [Wikipedia: C language](#)
- [Wikipedia: C++ language](#)
- [cplusplus.com](#): Excellent resource for C++ as well as C
- [cprogramming.com](#)

## Section 2.2 - C data types

Several C data types are commonly used to represent integers in embedded system

programs:

Table 2.2.1: Data types.

| Type                         | Width | Range                     | Notes  |
|------------------------------|-------|---------------------------|--|
| <b><i>signed char</i></b>    | 8     | -128 to 127               |  |
| <b><i>unsigned char</i></b>  | 8     | 0 to 255                  |  |
| <b><i>signed short</i></b>   | 16    | $-2^{15}$ to $2^{15}-1$   | $2^{16}$ is 65,356 (aka 64k)                                       |
| <b><i>unsigned short</i></b> | 16    | 0 to $2^{16}-1$           |  |
| <b><i>signed long</i></b>    | 32    | $-2^{31}$ to $2^{31}-1$   | $2^{32}$ is about 4 billion (aka 4 Gig)                            |
| <b><i>unsigned long</i></b>  | 32    | 0 to $2^{32}-1$           |  |
| <b><i>signed int</i></b>     | N     | $-2^{N-1}$ to $2^{N-1}-1$ | <i>Though commonly used, we avoid these due to undefined width</i> |
| <b><i>unsigned int</i></b>   | N     | 0 to $2^N-1$              |  |

Thus, a variable whose value may only range from 0 to 100 might be best declared as: unsigned char. A variable whose value may only range from -999 to 999 might be best declared as: signed short. Using the smallest possible data type ensures that the limited space in typical embedded system designs is conserved. Note that unsigned data types represent positive integers, while signed types represent negative and positive integers. If a variable is used to represent a series of bits (rather than a number), then an unsigned type should be used.

## P

Participation  
Activity

## 2.2.1: In embedded systems, the smallest data type is preferred.

Start

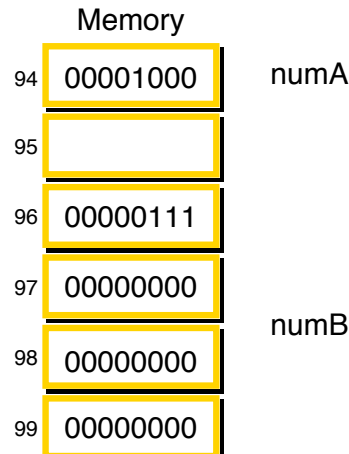
```

unsigned char numA = 7;
unsigned long numB = 7;

numA = numA + 1;
numB = numB + 1;

```

char: Only 1 byte  
long: 4 bytes



char: Only 1 b  
long: 4 byte o  
(move bytes  
do adds, m)

Embedded systems commonly deal with **1-bit** data items. C does not have a 1-bit data type, which is unfortunate. Thus, 1-bit items are typically represented using an unsigned char, as in: unsigned char myBitVar. The programmer only assigns the variable with either 0 or 1, e.g., myBitVar = 1, even though the variable could be assigned integers up to 255. Checking whether such a variable is 1 or 0 is typically done without explicit comparison to 1 or 0, and is instead done as if (myBitVar) or as if (!myBitVar).

Below are some example variable declarations:

Figure 2.2.1: Variable declarations.

```

unsigned char ucI1;
unsigned short usI2;
signed long slI3;
unsigned char bMyBitVar;

```

C also has types like **float** and **double** for floating-point numbers like 98.6 or  $6.02 \times 10^{23}$ . Many embedded programmers avoid using floating-point types. The reason is that microcontrollers commonly lack floating-point hardware, to stay small, low cost, and low power. Thus, floating-point arithmetic must be done in software, requiring hundreds of

instructions even for operations like addition or multiplication. Thus, we omit further discussion from this section.

A common practice is to name variables with a lower-case prefix indicating the data type, as above — `uc`, `us`, and `ul` for unsigned char, short, and long; `sc`, `ss`, and `sl` for signed char, short, and long; or `b` for bit — to help ensure that larger constants or variables aren't assigned to smaller variables. This book often skips variable naming practices, for the clarity of short examples. Programmers of larger projects should consider using such a variable naming convention.

`char` is called such because it is commonly used in desktop programming to represent the integer value of an 8-bit ASCII *character*. Note however that `char` is actually an integer. Also, 8-bits is sometimes called a **byte**.

In C, the word "signed" is optional for a signed type, so "**`char i1;`**" is the same as "**`signed char i1;`**". However, for program clarity, we avoid that shortcut. Also, the word "int" may follow the words short or long, but that word is superfluous so we usually omit it.


Unfortunately, although the above widths are quite common, C actually defines the above widths as *minimum widths*, so a compiler could for example create a long as 64 bits. Thus, a programmer should never assume an exact width, e.g., a program should not increment an "unsigned char" and expect it to roll over from 255 to 0, because the char could be 16 bits. Another unfortunate fact is that C allows a variable to be declared merely as type "int", where the width is compiler dependent. *Due to the unpredictability of int, we avoid using the int type almost entirely.* Following these conventions improves code **portability**, which is the ability to recompile code for a different microprocessor without undesirable changes in program behavior.

The underlying representation of each data type is binary. For an 8-bit unsigned char `uc1`:

Figure 2.2.2: Bit representation.

```
ucI1 = 1;    // underlying bits will be 00000001
ucI1 = 12;   // underlying bits will be 00001100
ucI1 = 127;  // underlying bits will be 01111111
ucI1 = 255;  // underlying bits will be 11111111
```

In binary, the rightmost bit has weight  $2^0$ , the next bit  $2^1$ , then  $2^2$ , etc. In other words, from left to right, the 8 bits have weights 128, 64, 32, 16, 8, 4, 2, and 1. 0001100 is thus  $8 + 4 = 12$ .



2.2.2: Binary number tool.

---

|       |       |       |       |       |       |       |       |                   |
|-------|-------|-------|-------|-------|-------|-------|-------|-------------------|
| 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0 (decimal value) |
| 128   | 64    | 32    | 16    | 8     | 4     | 2     | 1     |                   |
| $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |                   |

Signed data types in C use two's complement representation. At the bit level, a variable of type *char* set to 127 would have an internal representation of 01111111, while -128 would be 10000000, and -1 would be 11111111. For the curious reader -- the binary representation of a negative number in two's complement can be obtained by representing the number's magnitude in binary, complementing all the bits, and adding 1. For example, -1 is 00000001 (magnitude is 1) --> 11111110 (complement all bits) --> 11111110+1 = 11111111 (add 1). -128 is 10000000 (magnitude is 128) --> 01111111 (complement all bits) --> 01111111+1 = 10000000 (add 1). Note that the eighth bit will always be 1 for a negative 8-bit number and is thus called the sign bit. The programmer generally need not deal directly with the binary representations of signed numbers, because compilers/assemblers automatically create the proper constants (e.g., `myVar = -1;` would store the two's complement representation in `myVar`). However, knowing whether an item is signed or unsigned is important when assigning values, and for determining a variable's range. ([Wikipedia: Two's complement](https://en.wikipedia.org/wiki/Two's_complement))

The following provides some examples of choosing an appropriate data type for a variable based on the variable's intended purpose:

Table 2.2.2: Choosing appropriate type.

| Purpose   | Variable declaration  |
|---|---|
| Store a person's age in years                         | <code>unsigned char age; // Not &lt; 0, not &gt; 255</code>           |
| Store an airplane's speed                             | <code>unsigned short speed; // Not &lt; 0, not &gt; 64k</code>        |
| Store the remaining joules of energy in a car battery | <code>unsigned long energy; // Not &lt; 0, not &gt; 4Gig</code>       |
| Store feet of elevation above/below sea level of land | <code>signed short elevation; // Could be &lt; 0, not &gt; 32K</code> |

## P

### Participation Activity

### 2.2.3: C data types.

| # | Question  | Your answer          |
|---|---|----------------------|
| 1 | How many bits is an unsigned char?                          | <input type="text"/> |
| 2 | How many bits is a signed char?                             | <input type="text"/> |
| 3 | What is 15 in 8-bit binary?                                 | <input type="text"/> |
| 4 | What are the bits stored in memory for unsigned char x = 5? | <input type="text"/> |
|   | What are the bits stored in                                 | <input type="text"/> |



|    |   |                      |
|----|---|----------------------|
| 5  | memory for unsigned char x = 199?   | <input type="text"/> |
| 6  | Define a variable "volts" that can range from -100 to +100 (integers only). End with ;.                       | <input type="text"/> |
| 7  | Define a variable "height" that will hold a human's height in inches (integers only). End with ;.             | <input type="text"/> |
| 8  | Define a variable "birthYear" that will hold the year a person's birth occurred, range is 1. A.D. to today.   | <input type="text"/> |
| 9  | Define a variable "distMoon" that holds the distance in miles that the moon is from the earth on a given day. | <input type="text"/> |
| 10 | Define a variable "button" that will indicate whether a button is pressed or not.                             | <input type="text"/> |

## Section 2.3 - RIMS implicitly defined I/O variables

In RIMS, each microcontroller input and output (I/O) is implicitly defined as a global variable: `unsigned char A0;`, `unsigned char A1;`, ..., `unsigned char B0;`, etc. An item intended to represent a single bit, such as B0 in RIMS, should be set to only 0 or 1, as in: `B0 = 1`.

RIMS implicitly defines two additional global variables: A represents the 8-bit input as a decimal number, and B the 8-bit output:

Figure 2.3.1: RIMS implicit variables.

```
unsigned char A; // Built-in variable A, representing RIMS' 8 input
                  // pins as a single 8-bit variable

unsigned char B; // Built-in variable B, representing RIMS' 8 output
                  // pins as a single 8-bit variable
```

Implicitly-defined global variables are commonplace in microcontroller programming environments, enabling access to I/O pins as well as other microcontroller resources. In the case of RIMS, the RIMS.h file contains the definitions of A and B, and a programmer can enable their use via the line `#include "RIMS.h"` at the top of a program.

Built-in **grouped bits** like A and B enable the programmer to treat RIMS' 8 input bits or 8 output bits as an 8-bit decimal number. For example, A might represent a number like 12 coming from a temperature sensor as an 8-bit binary number 00001100. The programmer might use A in an arithmetic comparison as in `A > 15`, or an arithmetic calculation as in `newTemp = A + 5`. The programmer can write a number to B, as in `B = 15`, which will cause 00001111 to appear on RIMS' eight output pins. Setting RIM's outputs to RIM's inputs is achieved with just: `B = A`.

Because B is a global variable, a program can write as well as read that variable. However, input A is automatically written by the microcontroller and should never be written by a program, only read. In RIMS, writing to A results in a runtime error, causing execution to terminate. Caution: Microcontrollers differ in how they treat input and output variables, especially because most microcontrollers allow each pin to be configured as either input or output, so programmers should read a microcontroller's datasheet or other instructions carefully.

### Try 2.3.1: RIMS B outputs and A inputs.

Write a C program in RIMS that repeatedly executes: `B = 7`. Note that outputs B2, B1, and B0 become 1s, because 7 is 00000111 in binary (note that 7 appears under the output pins in RIMS). Next, set the input switches such that A3=1, A2=0, A1=0, and A0=1, with the other inputs 0, and note that 9 appears below RIMS' input pins (because 00001001 is 9 in binary).

### Try 2.3.2: Set RIMS B output to A plus 1.

Write and execute a C program for RIMS that sets B equal to A plus 1.

### Try 2.3.3: RIMS B output overflow.

Write a C program for RIMS that sets  $B = 300$ . Note that the value actually output on B is not 300, because an unsigned char has a range of 0 to 255.

C variables can be *initialized* when declared, as in: `unsigned char i1 = 5;`

The keyword **const**, short for constant, can precede any variable declaration, as in: `const unsigned char i1 = 5;`. A **constant variable**'s value cannot be changed by later code, and thus can help to prevent the introduction of future errors. A constant variable must therefore be initialized when declared. Above, 5 is a constant, and i1 is a constant variable.

### Try 2.3.4: Airbag system.

A car has a sensor that sets A to the passenger's weight (e.g., if the passenger weighs 130 pounds, A7..A0 will equal 10000010). Write a RIM C program that enables the car's airbag system ( $B0=1$ ) if the passenger's weight is 105 pounds or greater. Also, illuminate an "Airbag off" light (by setting  $B1=1$ ) if  $\text{weight} > 5$  pounds but  $\text{weight} < 105$  pounds.



## 2.3.1: RIMS inputs and outputs.

Write a statement, ending with a semicolon, for each desired behavior (unless otherwise specified). Write "not possible" if appropriate.

| # | Question   | Your answer  |
|---|--|--|
| 1 | Set RIMS' first output pin to 1.   | <input type="text"/>                                     |
| 2 | Set RIMS' 8 output bits to decimal 0.  | <input type="text"/>                                     |
| 3 | Set RIMS' first output to RIMS' last input.                                    | <input type="text"/>                                     |
| 4 | Set RIMS' outputs to RIMS's inputs minus 1, treating each as a decimal number. | <input type="text"/>                                     |
| 5 | Set A7 to 1.   | <input type="text"/>                                     |
| 6 | Set variable x to B7's current value.  | <input type="text"/>                                     |
| 7 | Write an expression (without parentheses) that evaluates to true if A5 is 1.   | <pre> if ( <input type="text"/> ) {     x = B0; } </pre> |

## Section 2.4 - Hexadecimal

Commonly an 8-bit unsigned item is not used as a number but rather just as eight distinct bits. For example, if RIM's eight outputs connected to eight light bulbs and the programmer

wanted to light all the bulbs, the programmer could write  $B = 255$  (because 255 is 11111111 in binary). However, 255 does not directly convey the programmer's intent. Ideally, the programmer could write  $B = \text{b}11111111$  or something similar, but C unfortunately has no binary constant support. But fortunately, C does support hexadecimal constants, which are closer to the ideal.

**Hexadecimal**, or **hex**, is a base 16 number, where each digit can have the value of 0, 1, ..., 8, 9, A, B, C, D, E, or F. A is ten, B is eleven, C is twelve, D is thirteen, E is fourteen, and F is fifteen.

Table 2.4.1: Hex/binary representations.

| Hexadecimal | Binary |
|-------------|--------|
| 0           | 0000   |
| 1           | 0001   |
| 2           | 0010   |
| 3           | 0011   |
| 4           | 0100   |
| 5           | 0101   |
| 6           | 0110   |
| 7           | 0111   |
| 8           | 1000   |
| 9           | 1001   |
| A           | 1010   |
| B           | 1011   |
| C           | 1100   |
| D           | 1101   |
| E           | 1110   |
| F           | 1111   |



Start

| dec | bin  | hex |
|-----|------|-----|
| 0   | 0000 | 0   |
| 1   | 0001 | 1   |
| 2   | 0010 | 2   |
| 3   | 0011 | 3   |
| 4   | 0100 | 4   |
| 5   | 0101 | 5   |
| 6   | 0110 | 6   |
| 7   | 0111 | 7   |
| 8   | 1000 | 8   |
| 9   | 1001 | 9   |
| 10  | 1010 | A   |
| 11  | 1011 | B   |
| 12  | 1100 | C   |
| 13  | 1101 | D   |
| 14  | 1110 | E   |
| 15  | 1111 | F   |

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
|---|---|---|---|

Check

Next

In the C language, a hex constant is preceded by 0x (the first character 0 is a zero, not the letter O). Thus, 0xFF represents 11111111 in binary. Each hex digit corresponds to four bits (four bits is called a **nibble**). 0xff may also be used; hex constants are not case sensitive. ([Wikipedia: Hexadecimal](#))

Thus, B = 0xFF sets all RIM outputs to 1s. The intent of 0xFF is clearer than 255. Likewise, B = 0xAA sets the outputs to 10101010, having much clearer intent than B = 170. Good practice is to always use hex rather than decimal when intending to write a bit pattern, even when the bit-pattern equivalent for a decimal number is known, to make the programming intent clear and prevent future confusion. For example, if intending to set all outputs to 0s, use B = 0x00 rather than B = 0. Likewise, to set B0 to 1 and B1-B7 to 0s, use B = 0x01 rather than B = 1.

## P

Participation  
Activity

## 2.4.2: Hex represents desired bit patterns more directly than decimal..

Start

binary    0   0   1   1        1   1   1   0    (62)

|       |       |       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|-------|-------|
| $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
| 128   | 64    | 32    | 16    | 8     | 4     | 2     | 1     |

Range: 0-15  
0000-1111Range: 0-15  
0000-1111

Des

11:

B

240 doe  
des

hex            3                    E                    (62)

 $16^1$  $16^0$ 

16

1

Range: 0-15  
0-FRange: 0-15  
0-F

111:

F

1 hex digit for every 4 binary digits  
(because  $2^4$  is 16)

## P

Participation  
Activity

## 2.4.3: Choose binary based on hex.

Start

Match the **hex value** in binary.

F2

0 0 0 0 0 0 0 0

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|

Check

Next





## 2.4.4: RIMS and hex.

When setting outputs, use a statement ending with a semicolon. Use hex when appropriate, and use uppercase (0xFF, not 0xff).

| # | Question  | Your answer          |
|---|---|----------------------|
| 1 | Set RIMS' first output pin to 1.  | <input type="text"/> |
| 2 | Set RIMS' first output pin to 1 and all other pins to 0.                                    | <input type="text"/> |
| 3 | Set RIMS' last output pin to 1 and all other pins to 0.                                     | <input type="text"/> |
| 4 | Set RIMS' outputs to all 1s.  | <input type="text"/> |
| 5 | Set RIMS' outputs to 01010101.  | <input type="text"/> |
| 6 | Fill in the blank of this expression to detect that all 8 RIMS inputs are 1s:<br>A == ____. | <input type="text"/> |
| 7 | What expression detects that RIMS' first input pin is 1?                                    | <input type="text"/> |
| 8 | What expression detects that A0 is the only RIMS input that is 1?                           | <input type="text"/> |

### Try 2.4.1: RIMS and hex.

Write a single statement for RIMS that sets B7-B4 to 1s and B3-B0 to 0s, using a hex constant. Test in RIMS.

### Try 2.4.2: Using hex again.

Write a single statement for RIMS that sets B0 to 1 if all eight A inputs are 1s, using a hex constant. Test in RIMS.

### Example 2.4.1: Simple hexadecimal example.

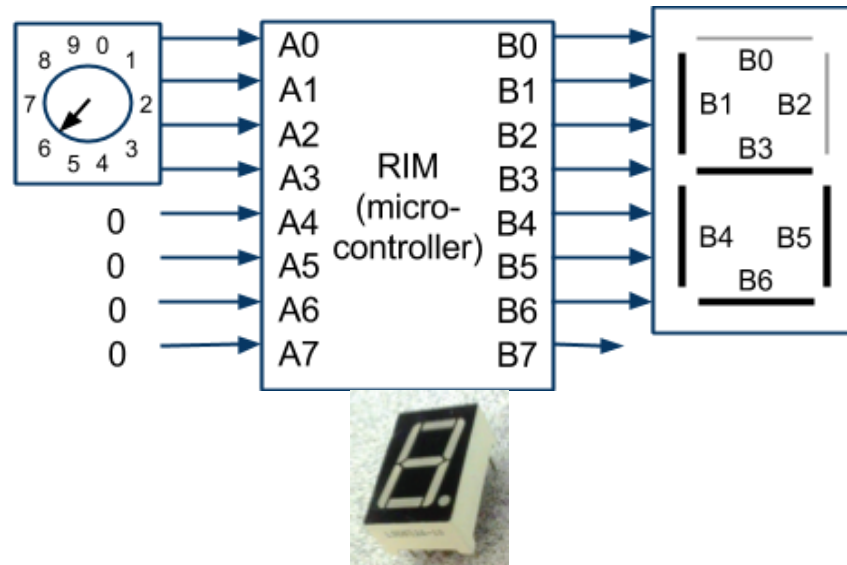
The following program sets B to 00000000 when A1A0=00, to 01010101 when A1A0=01, to 10101010 when A1A0=10, and to 11111111 when A1A0=11.

```
#include "RIMS.h"

// Set B to 00000000 when A1A0=00, to 01010101 when A1A0=01,
// to 10101010 when A1A0=10, and to 11111111 when A1A0=11
void main()
{
    while (1) {
        if (!A1 && !A0) {
            B = 0x00; // 0000 0000
        }
        else if (!A1 && A0) {
            B = 0x55; // 0101 0101
        }
        else if (A1 && !A0) {
            B = 0xAA; // 1010 1010
        }
        else if (A1 && A0) {
            B = 0xFF; // 1111 1111
        }
    }
}
```

### Example 2.4.2: 7-segment display.

Consider the following embedded system with a dial that can set A3..A0 to binary 0 to 9, and a **7-segment display** ([Wikipedia: 7-Segment Display](https://en.wikipedia.org/wiki/7-segment_display)) connected to B6..B0 as shown:



Below is a (partial) RIM C program that appropriately sets the display for the given dial position:

```
#include "RIMS.h"

void main()
{
    while (1) {
        switch( A )
        {
            case 0 : B = 0x77; break; // 0111 0111 (0)
            case 1 : B = 0x24; break; // 0010 0100 (1)
            case 2 : B = 0x5D; break; // 0101 1101 (2)
            //...
            case 9 : B = 0x6F; break; // 0110 1111 (9)
            default: B = 0x5B; break; // 0101 1011 (E for Error)
        }
    }
}
```

Participation  
Activity

## 2.4.5: 7-segment display.

| # | Question  | Your answer          |
|---|---|----------------------|
| 1 | What B_ outputs should be set to 1 for case 3? List in ascending order separated by spaces, i.e., B0 B2 ... | <input type="text"/> |
| 2 | To what should B be set for case 3? B = ____; Use uppercase letters for the hex literal.                    | <input type="text"/> |

## Try 2.4.3: RIMS 7-segment display.

Complete the above 7-segment display program using hex constants to produce the correct display for dial settings 3..8. Test in RIMS.

## Section 2.5 - Bitwise operators

An important programming skill for an embedded C programmer is manipulating bits within an integer variable. C's bitwise operators enable such bit manipulation.

- **& : bitwise AND** — 1 if both bit-operands are 1s.
- **| : bitwise OR** — 1 if either or both bit-operands are 1s.
- **^ : bitwise XOR** — 1 if exactly one of the two bit-operands is 1 (eXclusive OR)
- **~ : bitwise NOT** — 1 if the bit-operand is 0; 0 if bit-operand is 1.

Bitwise operators operate on the operands' corresponding bits, as shown below.

## P

## Participation Activity

### 2.5.1: Bitwise operations.

Start

1  
and & 1  
1

```
0x0E      0 0 0 0 1 1 1 0
0x18      & 0 0 0 1 1 0 0 0
           0 0 0 0 1 0 0 0
```

$$\begin{array}{ccc} & 1 & 1 & 0 \\ \text{or} & | & | & | \\ & 1 & 0 & 1 \\ & 1 & 1 & 1 \end{array}$$

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
|   | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| I | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
|   | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |

## P

## Participation Activity

### 2.5.2: Bitwise operations.

Enter the result of the given bitwise operation.

| # | Question   | Your answer          |
|---|--|----------------------|
| 1 | $00001111 \& 10101010$   | <input type="text"/> |
| 2 | $00001111 \mid 10101010$   | <input type="text"/> |
| 3 | $00001111 \wedge 10101010$   | <input type="text"/> |
| 4 | $\sim 00001111$  | <input type="text"/> |
| 5 | Type a statement that sets B's bits to the opposite of A's bits, so if A is 11110000, B will be 00001111. End with ; | <input type="text"/> |

**Boolean operators** `&&`, `||`, and `!` (there is no Boolean XOR operator) treat operands as zero (false) or non-zero (true). So while `0x0F & 0xF0` (bitwise AND) evaluates to 0 because each AND of corresponding operand bits evaluates to 0, in contrast `0x0F && 0xF0` (Boolean AND) evaluates to 1 because both operands are non-zero.

Recall that, because C does not have a Boolean or single-bit data type, a programmer uses an unsigned char to represent a Boolean data type. Good practice is to only use Boolean operators with variables representing Boolean (single-bit) data, even though a bitwise operator may yield the same result. For example, a programmer should use `A0 && A1` rather than `A0 & A1`, even though both yield the same result if A0 and A1 can each only be 1 or 0 (e.g., `00000001 && 00000000` and `00000001 & 00000000` both yield `00000000`). However, using a bitwise operator may mislead someone in believing the intent is to really compare each bit individually, which is not the case. Note, that `!A0` yields a different result than `~A0` (e.g., `!00000001` becomes `00000000`, while `~00000001` becomes `11111110`), illustrating a danger of being careless with use of Boolean versus bitwise operators. A common error is to accidentally type a bitwise operator like `|` when intending to type a Boolean operator like `||`, and vice-versa.

P

Participation Activity

2.5.3: Bitwise operations.

Choose the correct output. Select an output value as bX if the output depends on value of the bX input bit.

Start

|                    | b0 | b1 | b2 | b3 |
|--------------------|----|----|----|----|
| <code>&amp;</code> | 1  | 0  | 1  | 1  |
|                    |    |    |    |    |
|                    | 0  | 0  | 0  | 0  |

1

2

3

4

5

6

Check
Next

Embedded programmers make extensive use of masks. A **mask** is a constant pattern of 0s and 1s, as in `0x0F`, used with bitwise operators to manipulate a value. For example, setting B

to A but with B1 and B0 forced to 1s is accomplished with  $B = A | 0x03$ . Masks are based on the following bitwise operation features:

- $0 \& x$  yields 0 — Used to set a particular bit to 0.
- $1 | x$  yields 1 — Used to set a particular bit to 1.
- $0 | x$  yields  $x$ , or  $1 \& x$  yields  $x$  — Used to pass a bit through unchanged.

Different combinations of masks and bitwise operators achieve specific goals. Below are some examples of using bitwise operators along with RIMS I/O.

Figure 2.5.1: RIMS I/O and bitwise operator examples.

```
B = A | 0x01; // Sets B to A, except forces B0 to 1
B = A | 0x04; // Sets B to A, except forces B2 to 1
B = A & 0xF7; // Sets B to A, except that B3 is cleared to 0
B = A | 0xF0; // Sets B7..B4 to 1111, and B3..B0 to A3..A0
B = A & 0x0F; // Sets B7..B4 to 0000, and B3..B0 to A3..A0
B = B | 0x0F; // Keeps B7..B4 the same, forces B3..B0 to 1s
B = B & 0xF0; // Keeps B7..B4 the same, forces B3..B0 to 0s
```

The term *mask* comes from the role of letting some parts through while blocking others, like a mask someone wears on his face letting the eyes and mouth through while blocking other parts.

As a rule: To set bits to 1, think  $|$ ; to set bits to 0, think  $\&$ . Then, to pass bits through, use 0s if you used  $|$ , use 1s if you used  $\&$ .



## 2.5.4: Bit masking tool.

Select the masking bits to get the correct output.

Start

|       | b0 | b1 | b2 | b3 |   |   |
|-------|----|----|----|----|---|---|
| &     | 0  | 0  | 0  | 0  |   |   |
| <hr/> |    |    |    |    |   |   |
|       | 0  | 0  | 0  | 0  |   |   |
| <hr/> |    |    |    |    |   |   |
|       | 1  | 2  | 3  | 4  | 5 | 6 |

Check

Next





### Participation Activity

## 2.5.5: RIMS I/O and bitwise operations.

Given A is 00001111, type the 8-bit result of each operation (e.g., 11110000).

| # | Question | Your answer          |
|---|----------|----------------------|
| 1 | A & 0x03 | <input type="text"/> |
| 2 | A   0xF0 | <input type="text"/> |
| 3 | A & 0x18 | <input type="text"/> |
| 4 | A0 && A1 | <input type="text"/> |
| 5 | A0 & A1  | <input type="text"/> |



### Participation Activity

## 2.5.6: RIMS I/O applications.

Suppose expressions given below are used in an if statement as in `if (expression) { ... }`. Recall that a non-zero value is considered true, while zero is considered false.

| # | Question   | Your answer |
|---|--|-------------|
| 1 | The expression (A & 0x0F) evaluates to true if any of A3, A2, A1, or A0 is 1.  | True        |
|   |  | False       |
| 2 | The expression (A && 0xF0) evaluates to true if any of A7, A6, A5, or A4 is 1. | True        |
|   |  | False       |

|   |  |       |
|---|--|-------|
|   |  |       |
| 3 | The expression <code>(A == 0x0F)</code> is a correct way to check if A3, A2, A1, and A0 are all 1s, and A7 through A4 don't matter.              | True  |
|   |  | False |
| 4 | The expression <code>(A == 0xF)</code> is a correct way to check if A3, A2, A1, and A0 are all 1s, and A7 through A4 don't matter.               | True  |
|   |  | False |
| 5 | The expression <code>((A &amp; 0x0F) == 0x0F)</code> is a correct way to check if A3, A2, A1, and A0 are all 1s, and A7 through A4 don't matter. | True  |
|   |  | False |
| 6 | The expression <code>(A == 0xFC)</code> is a correct way to check if A7 through A2 are all 1s, and A1 and A0 are 0s.                             | True  |
|   |  | False |
| 7 | The expression <code>(!A1 &amp;&amp; !A0)</code> is a correct way to check if A7 through A2 are all 1s, and A1 and A0 are 0s.                    | True  |
|   |  | False |
| 8 | <code>B = A   0xC0</code> sets B's bits to A's bits except that B7 and B6 are forced to 1s.  | True  |
|   |  | False |

### Try 2.5.1: Setting RIMS output.

Write a single C statement for RIMS that sets B to A except that B1 and B0 are always 0. Hint: Use bitwise AND. Test in RIMS.

Exploring further:

- [Wikipedia: Bit Manipulation](#)

- [Wikipedia: Bitwise Operation](#)
- [Wikipedia: Mask](#)

## Section 2.6 - Shift operators

Embedded programmers commonly use two more operators when manipulating bits:

- *bitpat* << *amt* : **Left shift** *bitpat* by *amt* positions
- *bitpat* >> *amt* : **Right shift** *bitpat* by *amt* positions

For unsigned integer types, a shift operator moves the first operand's (the bit pattern) bits left or right by the shift amount indicated by the second operand. The shift amount can be 0, 1, ..., N, where N is the bit pattern's number of bits. The following figure illustrates.

Figure 2.6.1: Bitwise shift operators.

|            |            |
|------------|------------|
| 0x0F << 2: | 0x0F >> 3: |
| 00001111   | 00001111   |
| << 2       | >> 3       |
| -----      | -----      |
| 00111100   | 00000001   |

Note that vacated positions have 0s shifted in. Below are some examples of using shift operators and RIMS I/O.

Figure 2.6.2: Bitwise shift examples with RIMS I/O.

```

B = A << 1; // Sets B7 to A6, B6 to A5, ..., B1 to A0, and B0 to 0
B = A >> 4; // Sets B7..B4 to 0000, and B3..B0 to A7..A4
B = B << 1; // Sets B7 to B6, B6 to B5, ..., and B0 to 0

```

### Example 2.6.1: 4-bit addition using shifts.

Consider a system with a sound sensor with a 4-bit output connected to A3..A0, 0 (0000) meaning no sound, 15 (1111) meaning loud sound. A second sensor connects with A7..A4. A system should add those two 4-bit values, outputting the sum on B. The following example uses a mask to isolate A3..A0 in variable sound1, and a shift to isolate A7..A4 in variable sound2, before adding sound1 and sound2.

```
#include "RIMS.h"

void main()
{
    unsigned char sound1 = 0;
    unsigned char sound2 = 0;

    while (1) {
        sound1 = A & 0x0F; // 0000A3A2A1A0
        sound2 = A >> 4;   // A7A6A5A4A3A2A1A0 --> 0000A7A6A5A4
        B = sound1 + sound2;
    }
}
```

A common error is to fail to shift a binary number's bits to start with the rightmost bit, instead using just a mask. For the above example, `sound2 = A & 0xF0` yields A7A6A5A40000, which has A7..A4 in the wrong positions. If A3..A0 is 0001 (1) and A7..A4 is 0011 (3), B should be 00000001 (1) + 00000011 (3) which is 00000100 (4). Failing to shift would instead set B to 00000001 (1) + 00110000 (48), which is 00110001 (49).



Participation  
Activity

### 2.6.1: Shifting.

Assume A and B refer to RIMS' 8-bit inputs and outputs, respectively.

| # | Question  | Your answer          |
|---|---|----------------------|
| 1 | Given A is 01100111, what is A << 1?  | <input type="text"/> |
| 2 | Given A is 01100111, what is A >> 3?  | <input type="text"/> |
| 3 | Type a shift expression that results in bit A4 being in the rightmost bit. Type answer in this form: A >> 7 | <input type="text"/> |

|   |  |                      |
|---|--|----------------------|
| 4 | Type a single statement (including the ending semicolon) that sets B3..B0 to A7..A4, while setting B7..B4 to 0s. Use a shift operator and no other bitwise operators.                              | <input type="text"/> |
| 5 | Suppose A3..A0 hold a 4-bit binary number, and A7..A4 are used for another purpose. Fill in the blank so that this statement sets B to A3..A0 plus 5: B = (____) + 5;                              | <input type="text"/> |
| 6 | Suppose A7..A4 hold a 4-bit binary number, and A3..A0 are used for another purpose. Fill in the blank so that this statement sets B to A7..A4 plus 5: B = (____) + 5;                              | <input type="text"/> |
| 7 | Type a statement (including the ending semicolon) that sets B7..B4 to the value in variable num, which is an unsigned char that only holds values from 0 to 15. B3..B0 can be set to 0. End with ; | <input type="text"/> |

### Try 2.6.1: RIMS I/O with shifting.

Write a single C statement for RIMS that sets B3-B0 to A5-A2 and sets other output bits to 0s. Test in RIMS.

### Try 2.6.2: Parking lot sensors.

A parking lot has eight spaces, each with a sensor connected to RIM input A7, A6, ..., or A0. A RIM input being 1 means a car is detected in the corresponding space. Spaces A7 and A6 are reserved handicapped parking. Write a RIM C program that: (1) Sets B0 to 1 if both handicapped spaces are full, and (2) Sets B7..B5 equal to the number of available non-handicapped spaces.

Shifting can be performed on signed integer types too, but we do not recommend such use. Such shifting was previously popular because shifting a binary number left or right is equivalent to multiplying or dividing by 2, respectively (just as shifting a decimal number left or right is equivalent to multiplying or dividing by 10), and shifting could result in faster code execution than the slower `*` and `/` operations on some processors. However, modern compilers automatically replace `*` and `/` by shifts when possible, so today the programmer can emphasize understandable code rather than such low-level speedup attempts. For the curious reader, shifting a signed number performs an "arithmetic" shift that preserves the number's sign, rather than a "logical" shift that merely shifts all bits. This material will never shift signed types.

Exploring further:

- [Wikipedia: Arithmetic Shift](#)
- [Wikipedia: Logical Shift](#)

## Section 2.7 - Bit access functions

Defining C functions that get or set a particular bit of a variable can make programs easier to read, especially on microcontrollers that don't have variables for specific input and output pins like RIMS' A0..A7 and B0..B7 variables.

The following expression returns a value in which the  $k^{\text{th}}$  bit (rightmost bit being the 0th bit) of an unsigned char `x` is set to 1:

Figure 2.7.1: Expression to set a particular bit to 1.

```
x | (0x01 << k) // Evaluates to x but with k'th bit set to 1
```

The expression shifts mask 0x01 left  $k$  positions to get a 1 bit into the  $k^{\text{th}}$  position. The expression then bitwise ORs the result with  $x$ , forcing the  $k^{\text{th}}$  bit to 1 while passing through  $x$ 's other bits. For example, if  $k$  is 2, the mask will be shifted to become 00000100. Similar expressions can be created for unsigned short or long types.

Similarly, the following expression returns a value in which the  $k^{\text{th}}$  bit of an unsigned char  $x$  is set to 0:

Figure 2.7.2: Expression to set a particular bit to 0.

```
x & ~(0x01 << k) // Evaluates to x but with k'th bit set to 0
```

The expression shifts mask 0x01 left  $k$  positions to get a 1 bit into the  $k^{\text{th}}$  position. The expression then bitwise complements the result using  $\sim$  to yield a mask with a 0 in the  $k^{\text{th}}$  bit and 1s in the other bits. For example, when  $k$  is 1, the shifted mask will be 00000010, which will then be complemented into 11111101. The expression then bitwise ANDs the resulting mask with  $x$ , so that in the result the  $k^{\text{th}}$  bit is forced to 0, while  $x$ 's bits pass through to the remaining bits.

The above expressions can be called by a function that can set any bit to either 0 or 1:

Figure 2.7.3: SetBit() function to set a particular bit to 0 or 1.

```
// x: 8-bit value.    k: bit position to set, range is 0-7.    b: set bit to 0 or 1
unsigned char SetBit(unsigned char x, unsigned char k, unsigned char b) {
    return (b ? (x | (0x01 << k)) : (x & ~(0x01 << k)));
    //      Set bit to 1           Set bit to 0
}
```

The function uses C's **ternary conditional operator** (**?:**). The operator checks the first operand: If non-zero, the expression evaluates to the second operand, else the expression evaluates to the third operand. For example, for  $m = (n < 5) ? 44 : 99$ , if  $n < 5$  then  $m$  will be assigned 44, else  $m$  will be assigned 99.

## P

Participation  
Activity

## 2.7.1: SetBit returns the given unsigned char with a single flipped bit.

Start

```

        x = b11110101          k = 5          b
unsigned char SetBit(unsigned char x, unsigned char k, unsigned c
    return (b ? (x | (0x01 << k)) : (x & ~(0x01 << k)) );
}
    0          (x & ~(0x01 << k))
...
B = 0xF1;          11110101
B = SetBit(B, 2, 1);    & 11011111
                                  
B = SetBit(B, 5, 0);    B = 11010101

```

An example of using the SetBit function involves setting B's lowest four bits to the value of A0, without changing B's highest four bits (for RIMS).

Figure 2.7.4: Example using SetBit() function.

```

unsigned char i, val;
val = A0;
for (i=0; i<4; i++) {
    B = SetBit(B, i, val);
}

```





| # | Question  | Your answer          |
|---|---|----------------------|
| 1 | Call SetBit() to return RIMS' A input but with the lowest bit set to 1.                         | <input type="text"/> |
| 2 | Call SetBit() to return RIMS' A input but with the highest bit set to 0.                        | <input type="text"/> |
| 3 | The expression $x \mid (0x01 \ll k)$ sets a specific bit to what value?<br>Answer: 1 or 0.      | <input type="text"/> |
| 4 | In the expression $x \& \sim(0x01 \ll k)$ does &'s right operand have a 1 or a 0 in position k? | <input type="text"/> |
| 5 | If A is 00001111, what value will x get for:<br>$x = (A \& 0x04) ? 1 : 0;$                      | <input type="text"/> |

Similarly, a function can be created that gets (rather than sets) the value of a particular bit in an integer variable:

Figure 2.7.5: GetBit() function to get a particular bit.

```
unsigned char GetBit(unsigned char x, unsigned char k) {
    return ((x & (0x01 << k)) != 0);
}
```

The function creates a mask containing a 1 in position  $k$  and 0s in all other positions, then performs a bitwise AND to pass the  $k^{\text{th}}$  bit of  $x$  through, resulting either in a zero result if the  $k^{\text{th}}$  bit was 0, or a non-zero result if the  $k^{\text{th}}$  bit was 1. The function compares the result with 0, thus returning either a 1 or a 0.

# P

## Participation Activity

### 2.7.3: GetBit returns a 0 or 1 indicating the value of a single bit.

Start

```

x = b11001100      k = 3

unsigned char GetBit(unsigned char x, unsigned char k) {
    return ((x & (0x01 << k)) != 0);
} return

...
B = 0xCC;           1
unsigned char i = GetBit(B, 3);

                11001100
                & 00001000
                -----
                00001000  != 0

```

### Example 2.7.1: Parking lot example using bit access functions.

A parking lot has eight parking spaces, each with a sensor connected to input A. The following program sets B to the number of occupied spaces, by counting the number of 1s using the GetBit() function.

```
#include "RIMS.h"

unsigned char GetBit(unsigned char x, unsigned char k) {
    return ((x & (0x01 << k)) != 0);
}

void main()
{
    unsigned char i;
    unsigned char cnt;
    while (1) {
        cnt=0;
        for (i=0; i<8; i++) {
            if (GetBit(A, i)) {
                cnt++;
            }
        }
        B = cnt;
    }
}
```

In RIMS, the sum could also be obtained as:  $B = A_0 + A_1 + A_2 + A_3 + A_4 + A_5 + A_6 + A_7$ . However, some microcontrollers don't provide variables for a port's individual bits, but rather only for the grouped bits.

Note that the above bit access functions do not perform error checking (e.g., a call can attempt to set the 9th bit of a variable).

The examples using the SetBit and GetBit functions may seem inefficient due to computing the mask, but today's optimizing compilers handle these very efficiently. Furthermore, the **inline** keyword can be prepended to each function declaration (e.g., "inline unsigned char GetBit(...)") to encourage compilers to inline the function calls (though many compilers would do so anyways). **Inlining** means to replace a function call by the function's internal statements. Compiler optimizations may then eliminate most of the statements within the GetBit and SetBit functions.

A programmer may wish to copy the bit-access functions to the top of a file and then call those functions in a program, as shown below.

Figure 2.7.6: Example using bit access functions to set B as the reverse of A.

```
#include "RIMS.h"

// Bit-access functions
inline unsigned char SetBit(unsigned char x, unsigned char k, unsigned char
    return (b ? (x | (0x01 << k)) : (x & ~(0x01 << k)) );
}
inline unsigned char GetBit(unsigned char x, unsigned char k) {
    return ((x & (0x01 << k)) != 0);
}

void main(){
    unsigned char i;
    while(1){
        for (i=0; i<8; i++) {
            B = SetBit(B, 7-i, GetBit(A,i));
        }
    }
}
```

Try 2.7.1: Consecutive 1s.

Write a C program for RIMS that sets B0 = 1 if a sequence of three consecutive 1s appears anywhere on input A (e.g., 11100000 and 10111101 have such sequences, while 11001100 does not), using a C for loop and the GetBit() function.

P

Participation Activity

2.7.4: Bit access functions.

For the following, assume RIMS only has variables A and B available, and *not* the individual bit variables like A0, A1, ... and B0, B1, ... — as is common in some microcontrollers.

| # | Question  | Your answer |
|---|---|-------------|
| 1 | GetBit(A, 0) is a correct way to determine whether the bit in A's position 0 is 1.    | True        |
|   |   | False       |
| 2 | GetBit(A, 7, 1) is a correct way to determine whether the bit in A's position 7 is 1. | True        |
|   |   | False       |

|   |  |       |
|---|--|-------|
| 3 | A == 0x01 is a correct way to determine whether the bit in A's position 0 is 1.          | True  |
|   |  | False |
| 4 | (A & 0x04) == 0x04 is a correct way to determine whether the bit in A's position 2 is 1. | True  |
|   |  | False |
| 5 | B = SetBit(B, 7, GetBit(B, 0)) sets B's highest bit to B's lowest bit.                   | True  |
|   |  | False |
| 6 | If B is originally 11110000, then after SetBit(B, 2, 1), B is updated to 11110100.       | True  |
|   |  | False |
| 7 | If B is originally 00001111, then after B = SetBit(B, 7, 1), B is updated to 10000000.   | True  |
|   |  | False |
| 8 | Trying to set a bit to 1 that is already 1 using SetBit is an error.                     | True  |
|   |  | False |

Exploring further:

- [Wikipedia: ?: \(Ternary\) Operator](#)

## Section 2.8 - Rounding and overflow

## Rounding during integer division

Expressions involving integer division should be treated with extra care due to error that is introduced from **rounding during integer division**, wherein any fraction is truncated.

Consider the formula for converting Celsius to Fahrenheit:  $F = (9/5)*C + 32$ . Suppose a RIMS program is coded as follows:

Figure 2.8.1: Rounding error in a Celsius to Fahrenheit program.

```
#include "RIMS.h"

unsigned char C2F_uc(unsigned char C) {
    unsigned char F;
    F = (9/5)*C + 32;
    return F;
}

void main() {
    while (1) {
        B = C2F_uc(A);
    }
}
```

The table below shows the actual Fahrenheit values for Celsius values from 0 to 9, followed by those values when rounded to integers, followed by values obtained from the above program:

Table 2.8.1: Celsius to Fahrenheit values, showing rounding error.

| Celsius | Fahrenheit (actual) | Fahrenheit (integer) | Fahrenheit (from above program) | Fahrenheit (altered to do division later) |
|---------|---------------------|----------------------|---------------------------------|---|
| 0       | 32                  | 32                   | 32                              | 32  |
| 1       | 33.8                | 34                   | 33                              | 33  |
| 2       | 35.6                | 36                   | 34                              | 35  |
| 3       | 37.4                | 37                   | 35                              | 37  |
| 4       | 39.2                | 39                   | 36                              | 39  |
| 5       | 41.0                | 41                   | 37                              | 41  |
| 6       | 42.8                | 43                   | 38                              | 42  |
| 7       | 44.8                | 45                   | 39                              | 44  |
| 8       | 46.4                | 46                   | 40                              | 46  |
| 9       | 48.2                | 48                   | 41                              | 48  |

The values obtained from the above program are wrong due to rounding. Because all values in the C2F\_uc function's expression are integers, the term  $9/5$  is computed as an integer; the value is 1.8 but is rounded to 1 because C rounds by truncating the fraction. Thus, the computation actually being carried out is:  $F = 1 * C + 32$ .

The rounding problem can be partially addressed by doing the division as late as possible. For the above program, the line that computes F can be changed to:

```
F = (9*C)/5 + 32;
```

The values obtained after that change are shown in the last column of the above table. Those values are much closer to the desired values; the differences are due to the C language truncating any fractional part, rather than rounding up if the fractional part is 0.5 or greater.

Declaring items as floating-point type rather than integer type can reduce rounding error, but embedded programmers commonly (but not always) avoid floating-point types due to slow execution, especially on small microcontrollers.



| # | Question   | Your answer          |
|---|--|----------------------|
| 1 | Given x is 50, what is the result of the expression $(3/2) * x$ ?                                      | <input type="text"/> |
| 2 | Rewrite the expression $(3/2) * x$ to reduce the impact of rounding. Use one set of parentheses.       | <input type="text"/> |
| 3 | Rewrite the expression to reduce rounding error: $(a/2) + (b/2) + (c/2)$ . Use one set of parentheses. | <input type="text"/> |

## Overflow

On the other hand, when computing expressions, care must also be taken to avoid overflow during intermediate calculations. **Overflow** occurs when a value is too large to fit into a variable's storage, causing high-order bits to be lost. The following animation illustrates.



## P

## Participation Activity

### 2.8.2: Overflow.

Start

```
long totalHours;
```

...

```
totalHours = 4294967297;
```

[illegible]

**long** totalHours

32 bits wide

[illegible]

1

Because embedded programmers commonly use the smallest possible data types to conserve memory and improve execution speed on resource-limited microcontrollers, embedded programmers must be even more attentive to overflow than traditional programmers.

One way to reduce overflow is to compute divisions earlier. For example, computing the average of three variables might be done as:

$$\text{avg} = (a + b + c) / 3$$

Assume all variables are unsigned short type. If the values in a, b, and c are potentially large (e.g., 10,000 or more), then the sum could overflow (exceed 65,535), yielding an incorrect value that would then be divided by 3. Rewriting the expression as follows eliminates the possibility of overflow:

$$\text{avg} = (a/3) + (b/3) + (c/3)$$

Note that doing division earlier is the opposite solution as was proposed for reducing rounding error, for which doing division later was suggested. Thus, a tradeoff is involved. The best expression depends on the data that the programmer expects. For example, if unsigned short variables will store numbers in the hundreds, reducing rounding may dominate a programmer's concerns. But if those short variables will store numbers in the tens of thousands, avoiding overflow may dominate. No general rule exists; the embedded programmer must think carefully about a program's data.

Another way to reduce overflow within a calculation is to temporarily cast to a larger data type. A function dealing with a particular data type might first cast smaller types to larger ones, compute a result, and then cast back to a smaller type. The function can also explicitly check for overflow, and report an error, before returning a result. For example, the Celsius to Fahrenheit function might be rewritten as follows.

Figure 2.8.2: Function that converts to larger data types to detect and/or reduce overflow.

```
unsigned char C2F_uc(unsigned char C) {
    unsigned char F;
    unsigned short Csi, Fsi;

    Csi = C;
    Fsi = (9*Csi)/5 + 32; // Can't possibly overflow

    if (Fsi <= 255) {
        F = (char)Fsi;
    }
    else {
        // Print error message or indicate error if possible
        F = 0; // Set F to a unique "error" value if possible
    }
    return F;
}
```

A further improvement that can reduce rounding error, made possible by the larger data type, would be to set  $Csi = 10 * C$ , compute  $Fsi$  as above, round  $Fsi$  to the 10s place (if the 1s column is 5 or greater, increment the 10s column), and then divide  $Fsi$  by 10, before checking for overflow.



| # | Question   | Your answer |
|---|--|-------------|
| 1 | Postponing division, as in $(a * b * c) / d$ , can reduce occurrences of overflow.   | True        |
|   |  | False       |
| 2 | Given unsigned char variables a, b, and c, if each may range from 0-100, then $(a+b+c)/3$ may overflow.  | True        |
|   |  | False       |
| 3 | Given unsigned char variables a and b, if each may range from 0-100, then $(a + b) / (2 * a)$ may overflow.  | True        |
|   |  | False       |
| 4 | If a function's parameters p1 and p2 and return type are all unsigned short types, casting parameters first to unsigned long types may reduce overflow.                                  | True        |
|   |  | False       |
| 5 | If a function has a parameter that has been cast to a larger type to prevent overflow, multiplying the parameter by 10, then dividing a final result by 10, can further reduce overflow. | True        |
|   |  | False       |