Bits, Bytes and Logic Operations

1 Introduction

Embedded systems use a modern CPU to implement the desired tasks for a system. These tasks are thus implemented in a digital form and operate on digital data. This document first reviews digital data in the form of binary numbers and its relationship to other number systems and then reviews the methods used to manipulate binary numbers.

2 Bits and Bytes Review

This section reviews how numbers (i.e. data) are represented in computers. This includes:

- Binary numbers
- Fractions
- Fixed and floating point numbers
- Negative numbers

2.1 Decimal numbers

We use them every day without thinking about them or the processes we use to manipulate them. In practice, it is a number representation to the base 10, which gives us tens, hundreds, thousands, etc. multiplied by one of the ten numbers in the range 0 to 9.

For example the number 43958 represents:

4×10000	$4x10^4$
3×1000	$3x10^3$
9×100	$9x10^2$
5×10	$5x10^1$
8×1	$8x10^0$

Addition and subtraction are implemented by a simple carry and borrow method:

For example;

Addition:

$$\begin{array}{rrrrr}
 & 3 & 8 & 7 & 5 \\
+ & 2 & 7 & 5 & 3 \\
\multicolumn{5}{r}{=======} \\
 & & & & \underline{8} \\
 & & 1 & \underline{2} & \\
 & 1 & \underline{6} & & \\
 & \underline{6} & & & \\
 & \underline{6} & \underline{6} & \underline{2} & \underline{8}
\end{array}$$

Subtraction:

$$\begin{array}{rrrrr}
 & 3 & 2 & 5 & 8 \\
- & 2 & 7 & 6 & 3 \\
\multicolumn{5}{r}{=======} \\
 & & & & \underline{5} \\
 & & -1 & \underline{9} & \\
 & -1 & \underline{4} & & \\
 & \underline{0} & & & \\
 & & \underline{4} & \underline{9} & \underline{5}
\end{array}$$

Multiplication can be implemented as a series of additions (e.g. $4 \times 3 = 3 + 3 + 3 + 3$) and division can be implemented as a series of multiplications and subtractions (long division).

2.2 Binary numbers

Internally a computer uses binary notation to represent numbers. Binary numbers are simply numbers represented to the base 2, i.e. numbers can only take the values 0 or 1, compared with decimal (base 10) where numbers can take the values 0 to 9. For example the number 131 in decimal is 10000011 in binary:

$131 = 1\times100 + 3\times10 + 1\times1 = 1\times128 + 0\times64 + 0\times32 + 0\times16 + 0\times8 + 0\times4 + 1\times2 + 1\times1$

One of the reasons behind the use of binary is that it can be easily used to represent the state of a device, for example Logic gates are either ON or OFF (1 or 0), typically 5V or 0V.

Computer arithmetic at the carry level is very simple and fast (A+B):

A	0	0	1	1
B	0	1	0	1
Carry	0	0	0	1
Result	0	1	1	0

A binary joke:

"There are 10 types of people in the world, those who understand binary and those who don't".

To make it clear that a number is binary we often use the post-fix $_2$, i.e. 10_2

2.3 Hexadecimal Numbers

Numbers to the base 16, called hexadecimal numbers, are another type of number used in computer systems. Numbers can take the values of 0 to 15 where the numbers 10 through 15 are represented by the digits A, B, C, D, E and F;

i.e. numbers take the values of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

For example, the number D4CF represents $13 \times 16^3 + 4 \times 16^2 + 12 \times 16^1 + 15 \times 16^0 = 54479$. Hexadecimal is commonly used for describing 16 and 32-bit numbers. Each hexadecimal digit has four binary bits. The number is often preceded by the prefix 0x which indicates that it is a hexadecimal number.

For example: 0xD4CF

D (13) in binary is	1101
4	0100
C	1100
F	1111

D4CF = 1101 0100 1100 1111 = 54479

Therefore a 4 digit hexadecimal number represents a 16 bit binary number. Long 32-bit memory addresses are given by 8 hexadecimal digits.

2.4 Binary Arithmetic

2.4.1 Addition/Subtraction

Carry and borrow, such as is used for the addition and subtraction of decimal numbers, applies to the addition and subtraction of binary numbers, but using base 2.

For example;

```
     1 1 0 1 0 1  (53)                 1 1 0 1 0 1  (53)
  +  0 1 1 1 1 0  (30)               – 0 1 1 1 1 0  (30)
  =============                      ============
               1                                1
             1                              * 1
         1 0                              * 1
       1 0                              * 0
     1 1                              * 1
   1 0                                0

   1 0 1 0 0 1 1  (83)                0 1 0 1 1 1  (23)  (* indicates borrow)
```

Remember that logic gates store each bit, that is, each 0 or 1. If *n* bits are used to store a number the range is 0 to (2n - 1), thus a 8-bit number is limited to the range 0-255 and a 16-bit number is limited to the range 0-65535.

This leads to a problem if we try to add together numbers with a result that is greater than the range. Clearly we could use more digits but that requires more logic gates. For example;

```
    1 0 1 0 1 0 1 0  (170)
  + 1 1 0 0 1 0 0 0  (200)
   =====================
  1 0 1 1 1 0 0 1 0  (370)
```

Thus the 8-bit result above is 114 (written in decimal – i.e. 114_{10}). However, if we look at the carry bit form the last operation, we know if there has been an overflow. This can be interpreted as an error or used in subsequent stages.

2.4.2 Negative numbers

So far we have considered positive numbers – what about negative numbers? Recall that an 8-bit integer takes the range 00000000_2 to 11111111_2 (0_{10} to 255_{10}) . A negative number must satisfy the identify: A + (-A) = 0. A simple method to find the negative value of a positive integer is to take the 1's complement (the logical inverse, i.e. apply the bitwise NOT operator – see section 3) and then increment it. For example;

5_{10} = 00000101. So -5 is:

```
   00000101
   11111010  (1's complement)
 + 00000001
   ========
   11111011
```

Consider the number $1_{10} = 00000001_2$

```
  00000001
+ 11111111 (1's complement of 00000001 plus 00000001)
  ========
  00000000
```

In other words, the value 11111111_2 corresponds to the value -1 or to the unsigned integer 255.

In 2s compliment arithmetic, negative numbers start at 11111111_2 and run backward towards 00000000_2, i.e. $-1_{10} = 11111111_2$, $-2_{10} = 11111110_2$, etc.. Clearly, 11111011_2 can also be 251_{10} if we are considering unsigned (non-negative) numbers as well as -5_{10} for signed numbers. We just have to be careful not mix the two conventions, since the computer does not care – it just applies arithmetic to bit patterns and it is thus down to the programmer to decide how the numbers are interpreted.

In signed numbers the most significant bit (usually defined as the bit with the highest value, which in our case is the left most bit, which is the case in most binary numbers) is used as the sign bit, 0 for positive and 1 for negative. For an 8-bit number the range is -128_{10} to 127_{10} (i.e. 0_{10} to 127_{10} positive and -1_{10} to -128_{10} negative).

2.4.3 Multiplication

Multiplication is successive addition ($5_{10}\times4_{10} = 5_{10} + 5_{10} + 5_{10} + 5_{10}$). Although computers use faster methods than this it is important to consider range implications. The addition of two n bit binary numbers requires $n+1$ bits to store the answer. The multiplication of two n bit numbers requires $2n$ bits to store the answer. Note that shifting binary numbers to the left is the same as multiplying by 2 and shifting to the right is equivalent to division by 2. We must be aware of overflow and underflow with multiplication and division.

2.4.4 Fractions

We have considered operations on signed and unsigned integers (often referred to as fixed point arithmetic, by implication that the decimal point is after the least significant bit). However, we could treat binary values as a fraction for example take the value 010110_2 which is 22_{10}.

010110 = 0×25 + 1×24 + 0×23 + 1×22 + 1×21 + 0×20

If we take the decimal point as after the most significant bit we can interpret it as 0.6875;

$010110 = 0\times2^0 + 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3} + 1\times2^{-4} + 0\times2^{-5}$

For multiplication we still generate a product of 2n bits, e.g. ¼ + ¼ = 1/16, which in binary is 00100000 × 00100000 = 00001000 00000000.

Note that if we treated it as an integer we would have the same arithmetic.

2.4.5 Fixed point arithmetic

Using the idea of fixing the location of the decimal point we can represent real numbers using fixed-point notation. For example, for a 16 bit number use the 6 most significant bits for the integer part and the remaining 10 for the fractional part then 1.0 is given by 00000100 00000000_2. With this choice of decimal point location we have a maximum positive value of 31.9990234375 (01111111 11111111_2) and smallest positive value of 1×2^{-10} (00000000 00000001_2) – for signed numbers.

There are still problems with overflow when exceeding the range and can lead to values which are scaled before use in computations and then rescaled after a calculation.

Fixed point arithmetic is used in embedded systems where there is no floating point processor or software emulation of floating point is too slow.

2.4.6 Floating point arithmetic

Most modern processors support floating point arithmetic. In a floating point number there are exponent and fractional parts and numbers are represented in standard form (i.e. $Ax10^{B}$). Like with most engineering systems or concepts, there are standards defined by particular bodies, which engineers are either obliged or recommended to work to. This ensures some form of commonality between work conducted by different people in different industries, countries, etc.. In the case of floating point numbers, the IEEE defines a single precision (32-bit) floating point number as 1 sign, 8 exponent and 23 fraction bits:

```
S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
0 1      8 9                     31
```

Double precision (64-bits) is 1 sign, 11 exponent and 52 fraction bits:

```
S EEEEEEEEEEE   FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
0 1         11 12                                                 63
```

From the left: the first bit is the sign bit, followed by the exponent bits and finally the fraction bits.

Compilers or a run-time library for the chosen processor and programming language handle transformations between differing exponents.

Exercise
Typical values for 32-bit floating point arithmetic are 8.43×10^{-37} to 3.37×10^{38} with a resolution of 10^{-7}. Using a calculator input a number in the range [0-100] and square root it 20 times. Now square the answer 20 times. Why is the result a different value than you started with?

Answer

Truncation/over/under flow errors leading to a 20 times squared rooted value accurate only to a fixed number of decimal places and the rest ignored. When it is squared twenty times, the truncated number is squared and not the true value, leading to an error – see the Patriot Missile example in section 4.

3 Bitwise Logic

This section looks at programming on the “bit” level. Even though information data is usually composed of multiple bits it is often important to consider the individual bits in a longer string of bits. We isolate and analyse these bits through bitwise functions. *Bitwise programming* is particularly important for accessing and manipulating ports on a device. This is because device registers are commonly memory mapped to an I/O port and manipulating bits on a register provides fine control (bit level, rather than byte) over I/O.

There are several reasons why bit control is useful for I/O control:

- Individual bits may set a device into a particular operating mode
- Setting individual bits on a control register may make a pin I or O exclusive
- Data may not span all bits
- Device status information may be on the bit level, rather than as a byte
- MEMORY EFFICIENT

As an example, the 1 byte status register of an infrared sensor might be split into bits to provide the status information listed below:

- Bit 1 LOW_BATTERY = bit set
- Bit 2 SENSOR_DETECT {positive detection}
- Bit 3 DEVICE_ERROR
- Bit 4 ALARM_ACTIVE {associated alarm is active}
- Bit 5 SENSOR_THRESHOLD_LEVEL {2 bits, therefore 4 levels}
- Bit 6 SENSOR_THRESHOLD_LEVEL
- Bit 7 NULL {no operation}
- Bit 8 NULL

In C, bits are manipulated using bitwise operators. These are used to Set, Test or Shift bits in a byte/word. This only works on char/int variety data types (not floats etc…). Bitwise operations work just like logical equivalents (AND, OR, NOT), except they work at the bit level, rather than “expression” level.

Below is a list of C bitwise operators:

Bitwise Operator	Action
&	AND
|	OR
^	XOR
~	NOT (1’s complement)
>>	Shift right
<<	Shift left

The following is a list of the truth tables for the four logic functions:

AND (&) truth table

Input 1	Input 2	Output
0	0	0
1	0	0
0	1	0
1	1	1

OR (|) truth table

Input 1	Input 2	Output
0	0	0
1	0	1
0	1	1
1	1	1

XOR (^) truth table

Input 1	Input 2	Output
0	0	0
1	0	1
0	1	1
1	1	0

NOT (~) truth table

Input	Output
0	1
1	0

If you are unfamiliar with logic functions/gates then you should refer to a standard electronics text book that covers these.

A few examples that illustrate the use of these bitwise functions are given below.

3.1.1 Bitwise &

```
byte num8bit = 12;  //  00001100

num8bit & 1;        //  00001100
                    // &00000001
                        ========
                        00000000 (12 & 1)

// another number
byte num8bit = 127; //  01111111

num8bit & 48;       //  01111111
                    // &00110000
                        ========
                        00110000 (127 & 48)
```

This example demonstrates that the & operator is used to clear bits.

3.1.2 Bitwise |

```
byte num8bit = 12;  //  00001100

num8bit | 1;        //  00001100
                    // |00000001
                        ========
                        00001101 (12 | 1)
```

This example demonstrates that the | operator can be used to set individual bits, i.e. the 1 bits in either operand get set to 1 in the output.

Bitwise XOR, “^”, is like bitwise |, but only dissimilar bits give a 1 bit in the output. Bitwise ~ (ones complement) reverses the value of each bit.

3.1.3 Bitwise <<

```
byte num8bit = 12;  // 00001100

num8bit << 2;       // 00110000 (12 << 2)
```

The shift functions shifts all of the numbers the specified digits. Digits disappear when they are shifted out of the number (i.e. they disappear from the left when using <<) and 0’s appear into the number on the opposite side to replace the lost digits (i.e. 0’s appear on the right when using <<).

3.1.4 Convenience functions

Raw bitwise operations on numbers can sometimes be hard to decipher. As a consequence convenience functions for setting bits are often used. The convenience functions for setting bits can then be defined in the following manner, where the macros are used as bit identifiers (`unsigned int BIT`) and the operation is done on the variable `unsigned int num`:

```
//Sets the bit identified by BIT to 1 in the number num
unsigned int BitSet(unsigned int num, unsigned int BIT) {
    unsigned int rv;
    rv = num | BIT;
    return rv;
}


//Sets the bit identified by BIT to 0 in the number num
unsigned int BitUnSet(unsigned int num, unsigned int BIT) {
    unsigned int rv;
    rv = num & (~BIT);
    return rv;
}


int BitTest(unsigned int num, unsigned int BIT) {
    if(num & BIT)
        return 1;
    else
        return 0;
}
```

4 Data types for Embedded Systems

When choosing data types for embedded software you will usually have the choice of several different types. For example the STM32F407 supports signed and unsigned 8-bit, 16-bit (half-word), 32-bit (full-word) and 64-bit (double-word) data types. It also supports floating point data types.

When defining variables in your programme you will need to select a particular data types. When doing so, there are several issues need to be considered:

- The use of different data types often results in different answers (round-off, overflow, etc.) - larger data types generally lead to greater precision.

- It is well known that inappropriate data types may cause serious software defects, for example precision errors – leading to accumulation errors (see the Case Study on the Patriot missile below).

- For real-time applications, data types also significantly affect the processing time – especially mathematical operations (floating point arithmetic is generally slower than fixed point arithmetic).

- Some embedded systems suffer significant constraints in memory and processing power. Choosing appropriate data types is thus critical when developing embedded software. For example, for systems with severe processing limitations, lower size data types (e.g. char, int or float instead of double) should be used whenever it is possible.

4.1 Case Study – Patriot missile

This case study highlights the importance of selecting the correct data types and the implications of arithmetic functions when designing a solution for a particular application.

An America Patriot Missile battery failed to track and intercept an incoming Iraqi Scud missile on Feb 25, 1991, Gulf War. 28 soldiers killed and ~100 people injured when the Scud struck an Army barracks.

The cause was the inaccurate calculation of the time since boot due to computer arithmetic errors. The time in tenths of a second from the systems internal clock was multiplied by 1/10 to produce the time in seconds. However 1/10 was represented by a non-terminating binary expansion, which was stored as a 24 bit fixed point number, thus the number is truncated/abbreviated. The 24 bit representation of 1/10 when multiplied by a large number, i.e. the time in tenths of a second after a long period of operation leads to a significant error, e.g. after the estimated 100 hours of continuous use of the patriot missile battery the error would be:

$1/10 = 1/2^4+1/2^5+1/2^8+1/2^9+1/2^{12}+1/2^{13}+....$

Which in binary is	0.0001100110011001100110011001100....
Stored as a 24 bit number this is	0.00011001100110011001100

This gives an error of 0.0000000000000000000000011001100... in binary, or about 0.000000095 in decimal.

Therefore the error after 100 hours = $0.000000095\times100\times60\times60\times10=0.34$ seconds

Since a Scud travels at ~ 1676 ms^{-1} it travels more the 0.5km in 0.34 seconds. This was sufficient to place the Scud outside of the “range gate” tracked by the Patriot battery.

http://www.ima.umn.edu/~arnold/disasters/patriot.html

5 An important final note

Further information on Binary, Hexadecimal, Data Types, Logic Operations, etc. with particular reference to the ARM Cortex-M processor can be found in the following document in the “General Resources” folder of the supporting information for the STM32F4Discovery:

“Embedded Software in C for ARM Cortex M”