Binary and Hexadecimal

Introduction

This document introduces the binary (base two) and hexadecimal (base sixteen) number systems as a foundation for systems programming tasks and debugging. In this document, if the base of a number might be ambiguous, we use a base subscript to indicate the base of the value. For example:

453_{10}	is a base ten (decimal) value
1101_{2}	is a base two (binary) value
8217_{16}	is a base sixteen (hexadecimal) value

Why do we need binary?

The binary number system (also called the base two number system) is the native language of digital computers. No matter how elegantly or naturally we might interact with today's computers, phones, and other smart devices, all instructions and data are ultimately stored and manipulated in the computer as binary digits (also called *bits*, where a bit is either a 0 or a 1). CPUs (central processing units) and storage devices can only deal with information in this form. In a computer system, absolutely everything is represented as binary values, whether it's a program you're running, an image you're viewing, a video you're watching, or a document you're writing. Everything, including text and images, is represented in binary.

In the "old days," before computer programming languages were available, programmers had to use sequences of binary digits to communicate directly with the computer. Today's high level languages attempt to shield us from having to deal with binary at all. However, there are a few reasons why it's important for programmers to understand the binary number system:

1. **Debugging** - Sometimes during the debugging process, we need to examine the contents of memory. Knowledge of the binary number system helps us correctly interpret what we're seeing, allowing us to make more informed decisions about what might be going wrong in our code.
2. **Systems Programming** - When writing or reading software that deals directly with controlling hardware (e.g., embedded systems, operating systems, device drivers, etc.), we are often dealing with code that operates on data at the individual bit level.
3. **Efficient Data Manipulation** – An understanding of binary and how to manipulate data at the bit level can allow us to write code that performs better and more logically. Without this understanding, we might manipulate data in less-efficient or less-intuitive ways.

How decimal (base 10) works

Before diving head first into binary, it's useful to first analyze the base ten (decimal) number system that we use every day. This analysis will help us take the leap to binary in the next section.

The base ten number system provides us with ten available digits: 0 through 9. In base ten, we put multiple digits together to represent whole numbers larger than 9, where each position in the number is given a place value – some multiple of ten. For example, the decimal value 425 has a 5 in the ones place, a 2 in the tens place, and a 4 in the hundreds place. We're using only the ten digits that the decimal number system allows, but with those digits, we can express whole numbers that

Copyright © 2017 Bytellect LLC. All rights reserved.

are arbitrarily large; just put the appropriate digits into the appropriate places, and you have your value.

Where do these place values come from? As children, we likely memorized them as the ones, tens, hundreds, thousands, etc. They are all powers of ten:

Ones place $10^0 = 1$

Tens place $10^1 = 10$

Hundreds place $10^2 = 100$

Thousands place $10^3 = 1000$

and so on. So, our sample decimal number 425, when analyzed in terms of place values is:

$4 \times 10^2 + 2 \times 10^1 + 5 \times 10^0$

$4 \times 100 + 2 \times 10 + 5 \times 1$

$400 + 20 + 5$

How binary (base 2) works

By contrast, the binary number system (base two) provides only two available digits to express values: 0 and 1. In binary, we put multiple digits together to represent whole numbers larger than 1, where each position in the number is given a place value – some multiple of two.

Where do these binary place values come from? Recall that place values in decimal (base 10) use powers of ten. Similarly, place values in binary (base two) use powers of two:

Ones place $2^0 = 1$

Twos place $2^1 = 2_{10}$

Fours place $2^2 = 4_{10}$

Eights place $2^3 = 8_{10}$

and so on. For example, the binary value 1101 has a 1 in the ones place, a 0 in the twos place, a 1 in the fours place, and a 1 in the eights place. We're using only the two digits that the binary number system allows, but with those digits, we can express whole numbers that are arbitrarily large; just put the appropriate digits into the appropriate places, and you have your value.

To convert our sample binary number 1101 into its equivalent decimal value, we analyze the binary digits as follows:

$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

$1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$

$8 + 4 + 1$

13_{10}

We can say, then, that:

1101_2 is equal to 13_{10}

In the world of computers, it is convenient to refer to a sequence of eight bits as a *byte*. Storage space is typically measured in terms of bytes (e.g., a hard disk might have a size of 500GB, or 500 gigabytes, which is 500 billion bytes). Historically, the term byte has not always meant eight bits. But nowadays, the term byte is widely understood as meaning a sequence of eight bits. In many computer architectures, a byte is the smallest addressable unit of storage.

Copyright © 2017 Bytellect LLC. All rights reserved.

How hexadecimal (base 16) works

While binary is indeed the number system used internally by computer systems for both data and instructions, it can be very cumbersome and error-prone to deal with long sequences of binary digits. In fact, many high-level languages don't have a facility for handling binary values directly in source code. To make dealing with such values somewhat easier for programmers, we typically use a convenient "shorthand" for binary – the *hexadecimal* (or base 16) number system. It is considered a shorthand because it allows us to express the value of an eight-bit byte, for example, using only two digits instead of eight binary digits.

Recall that the decimal number system provides ten digits (0 through 9), and the binary number system provides just two digits (0 and 1). The hexadecimal number system (sometimes called *hex*, for short), provides a whopping sixteen digits to express values: 0 through 9 and A through F. So, how did letters creep into this number system? Well, in our everyday world, decimal numbers use up all of the ten digits we know about. Any number system with a base higher than ten needs some additional symbols to represent the higher-valued digits. Base eleven would need one more digit symbol beyond 0 through 9; base twelve would require two additional digit symbols. So, base sixteen (hexadecimal) requires 6 additional digit symbols. And the symbols for these higher-valued digits are A, B, C, D, E, and F.

What are the decimal values of hexadecimal digits? The first ten hex digits, 0 through 9, are easy to remember – they have the decimal values 0 through 9. That's a relief. The remaining six digits A through F have the decimal values ten through fifteen. Here are the conversions between hexadecimal digits and their equivalent decimal values:

Hexadecimal Digit	Equivalent Decimal Value
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

Just as in other number systems, in hexadecimal we place multiple digits together to represent whole numbers larger than 15, where each position in the number is given a place value – in this case, some multiple of sixteen.

Where do these hexadecimal place values come from? Recall that place values in decimal (base 10) use powers of ten. Similarly, place values in hexadecimal use powers of 16:

Ones place $16^0 = 1$

Sixteens place $16^1 = 16_{10}$

256s place $16^2 = 256_{10}$

Copyright © 2017 Bytellect LLC. All rights reserved.

4096s place $16^3 = 4096_{10}$

and so on. Given a hexadecimal value of 5AE7, let's see how these place values are used to arrive at the equivalent decimal value (all values are decimal, unless otherwise noted). We use the exact same technique we used to evaluate decimal and binary values. The only difference in hexadecimal is that we're dealing with powers of 16, and we need to translated the hexadecimal digits A through F into their equivalent decimal values:

$5 \times 16^3 + A_{16} \times 16^2 + E_{16} \times 16^1 + 7 \times 16^0$

$5 \times 4096 + 10 \times 256 + 14 \times 16 + 7 \times 1$

$20480 + 2560 + 224 + 7$

23271_{10}

This may all seem a bit cumbersome and esoteric, until we see how binary values can be expressed in hexadecimal notation, and how we can easily convert back and forth between binary and *hex* (short for hexadecimal). Using the same hexadecimal value, 5AE7, let's see what the equivalent binary value is:

$5AE7_{16}$ is equivalent to 0101101011100111_2

Clearly, hex is much more concise than binary. But why not just use the somewhat-more-concise decimal number system that we already know and love? Certainly, decimal is more concise than binary, since we can say 23271 in decimal instead of 0101101011100111 in binary. Unfortunately, converting between decimal and hex requires the multiplication mess shown above. In contrast, converting between binary and hex is far easier. Here are a few more examples to consider:

0101111100001110_2 can be expressed as $5F0E_{16}$

1010101010101010_2 can be expressed as $AAAA_{16}$

0010100011001101_2 can be expressed as $28CD_{16}$

Effectively, every four binary digits (bits) of a binary value translate directly into an equivalent single hexadecimal digit. Consider the following table that translates between hex digits and sequences of four bits:

Hexadecimal Digit	Equivalent Binary Value	Equivalent Decimal Value
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
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Here's how to convert from a binary value to a hexadecimal value:

Copyright © 2017 Bytellect LLC. All rights reserved.

1. If the number of binary digits is not a multiple of four, fill in leading zeros on the left-most end of the binary value to make the number of binary digits a multiple of four. This doesn't alter the binary value at all, but makes the subsequent steps easier.
2. Now, consider a group of four binary digits, and using the table above, convert that group of four bits into a single hexadecimal digit.
3. Repeat step 2 for each remaining group of four binary digits.
4. The resulting sequence of hexadecimal digits is the hexadecimal equivalent of the binary value.

To see how this works, let’s convert the number 10100011001101_2 into its hexadecimal equivalent. First, we notice that the number of digits is not a multiple of four, so we fill in two leading zeros on the left, to get 0010100011001101_2. This has the same value as the original. Now, looking at each group of four bits in turn, we convert each group into a single hexadecimal digit:

First group 0010_2 is equivalent to 2_{16}

Second group 1000_2 is equivalent to 8_{16}

Third group 1100_2 is equivalent to C_{16}

Fourth group 1101_2 is equivalent to D_{16}

Putting the hex digits together, we obtain the hex value $28CD_{16}$. Converting from hex to binary is similarly straightforward. Going in this direction, we look at each hex digit and convert it into its equivalent sequence of four binary digits:

First digit 2_{16} is equivalent to 0010_2

Second digit 8_{16} is equivalent to 1000_2

Third digit C_{16} is equivalent to 1100_2

Fourth digit D_{16} is equivalent to 1101_2

Putting all the groups of four binary digits together, we end up with 0010100011001101_2. At first, we use the table to perform the conversion between a hex digit and its sequence of four bits. After a while, we start recognizing the patterns and performing the conversions in our head – a task that would be much more difficult if we had to convert between decimal values and binary values.

C Support for Hexadecimal

The C programming language does not directly support the expression of binary values, but it does provide good support for hexadecimal values. To express a hex value in C, place a 0x or 0X immediately before the sequence of hex digits, with no intervening spaces. (By convention, most real-world C code uses the lower-case x). The letter digits A through F may be either uppercase or lowercase – just be consistent to aid in readability. Here are some examples of expressing hex values in C:

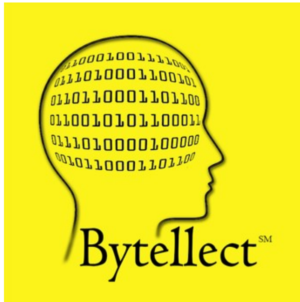
```
0xFF         0x5ae7       0xFFFFBAAD   0x28CD
0xBAADFOOD   0xDEADF00D   0xBAADBEEF   0xdecade
```

Incidentally, C also supports octal (base eight) values. While some existing code contains octal values, use of hexadecimal values is much more common today. Octal values begin with a zero, and contain the digits 0 through 7. One octal digit represents exactly three bits of binary. So, octal is another shorthand for binary, but it's not as good of a shorthand as hexadecimal is.

The printf library function (and related functions) supports printing integer values in hexadecimal. We can control whether the A through F print in uppercase or lowercase, and whether leading zeros

Copyright © 2017 Bytellect LLC. All rights reserved.

appear in the output. (Refer to The C printf Fact Sheet for details.) Likewise, the scanf library function supports reading hexadecimal values. (Refer to The C scanf Fact Sheet for details.)



Bytellect LLC provides professional training and consulting services, including software training for developers and end users, both online and onsite. Bytellect also designs and develops custom software and firmware solutions for our clients. Visit our web site at www.bytellect.com for more details and contact information.

Copyright © 2017 Bytellect LLC. All rights reserved.