Binary and Octal

Introduction

This document introduces the binary (base two) and octal (base eight) 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
3217_{8}	is a base eight (octal) 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

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.

How octal (base 8) 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, programmers typically use a convenient "shorthand" for binary.

Two of these shorthand notations are the *octal* (base 8) number system and *hexadecimal* (base 16) number system. Hexadecimal is actually used more often than octal today. Octal was more commonly used in the early days of computing, when the binary widths of data values and memory addresses were smaller. But both octal and hexadecimal are still in use. (For more information on the hexadecimal number system, see the Binary and Hexadecimal document.) Octal is considered a shorthand for binary because it allows us to express three binary digits using only one octal digit.

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 octal number system, provides eight digits (0 through 7).

What are the decimal values of octal digits? Octal digits have the same values as the first eight decimal digits, 0 through 7, are thus very easy to remember. Here are the octal digits and their equivalent decimal values:

Octal Digit	Equivalent Decimal Value
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7

Just as in other number systems, in octal we place multiple digits together to represent whole numbers larger than 7, where each position in the number is given a place value – in this case, some multiple of eight.

Where do these octal place values come from? Recall that place values in decimal (base 10) use powers of ten. Similarly, place values in octal use powers of 8:

Ones place $8^0 = 1$

Eights place $8^1 = 8_{10}$

64s place $8^2 = 64_{10}$

512s place $8^3 = 512_{10}$

and so on. Given an octal value of 6273, 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 octal is that we're dealing with powers of 8:

$6 \times 8^3 + 2 \times 8^2 + 7 \times 8^1 + 3 \times 8^0$

$6 \times 512 + 2 \times 64 + 7 \times 8 + 3 \times 1$

$3072 + 128 + 56 + 3$

3259_{10}

This may all seem a bit cumbersome and esoteric, until we see how binary values can be expressed in octal notation, and how we can easily convert back and forth between binary and octal. Using the same octal value, 6273, let's see what the equivalent binary value is:

6273_8 is equivalent to 110010111011_2

Clearly, octal 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 3259 in decimal instead of 110010111011 in binary. Unfortunately, converting between decimal and octal requires the multiplication mess shown above. In contrast, converting between binary and octal is far easier. Here are a few more examples to consider:

101111100001110_2 can be expressed as 57416_8

001010101010101010_2 can be expressed as 125252_8

010100011001101_2 can be expressed as 24315_8

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

Octal Digit	Equivalent Binary Value
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Here's how to convert from a binary value to an octal value:

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

To see how this works, let's convert the number 1100011001101_2 into its octal equivalent. First, we notice that the number of digits is not a multiple of three, so we fill in two leading zeros on the left, to get 001100011001101_2. This binary number has the same value as the original. Now, looking at each group of three bits in turn, we convert each group into a single octal digit:

First group 001_2 is equivalent to 1_8

Second group 100_2 is equivalent to 4_8

Third group 011_2 is equivalent to 3_8

Fourth group 001_2 is equivalent to 1_8

Fifth group 101_2 is equivalent to 5_8

Putting the octal digits together, we obtain the octal value 14315_8. Converting from octal to binary is similarly straightforward. Going in this direction, we look at each octal digit and convert it into its equivalent sequence of three binary digits:

First digit 1_8 is equivalent to 001_2

Second digit 4_8 is equivalent to 100_2

Third digit 3_8 is equivalent to 011_2

Fourth digit 1_8 is equivalent to 001_2

Fifth group 5_8 is equivalent to 101_2

Putting all the groups of three binary digits together, we end up with 001100011001101_2. At first, we use the table to perform the conversion between an octal digit and its sequence of three 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 Octal

The C programming language does not directly support the expression of binary values, but it does provide good support for octal values. To express an octal value in C, place a 0 immediately before the sequence of octal digits, with no intervening spaces. Here are some examples of expressing octal values in C:

```
0377          055347        037777735255  024315
027253370015  033653370015  027253337357  067545336
```

C also supports hexadecimal (base 16) values, using a prefix of 0x or 0X before a sequence of hexadecimal digits. You can find out more about hexadecimal in Binary and Hexadecimal.

The printf library function (and related functions) supports printing integer values in octal and hexadecimal. (Refer to The C printf Fact Sheet for details.) Likewise, the scanf library function supports reading octal and hexadecimal values. (Refer to The C scanf Fact Sheet for details.)