

Part 1 Theory Fundamentals

2

Chapter 1 Information Representation

Learning objectives

By the end of this chapter you should be able to:

- show understanding of the basis of different number systems
- show understanding of, and be able to represent, character data in its internal binary form
- show understanding of how data for a bitmapped or vector graphic image is encoded
- show understanding of how sound is represented and encoded
- show understanding of the characteristics of video streams
- show understanding of how digital data can be compressed.

1.01 Number systems

As a child we first encounter numbers when learning to count. Specifically we learn to count using 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. These are natural numbers expressed in what can be described as the denary, decimal or base-10 system of numbers. Had we learned to count using 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 we would have more clearly understood that the number system was base-10 because there are 10 individual, distinct symbols or digits available to express a number.

A little later we learn that the representation of a number has the least significant digit at the right-hand end. For example, writing a denary number as 346 has the meaning:

$$3 \times 10^2 + 4 \times 10^1 + 6 \times 10^0$$

All computer technology is engineered with components that represent or recognise only two states. For this reason, familiarity with the binary number system is essential for an understanding of computing systems. The binary number system is a base-2 system which uses just two symbols, 0 and 1. These binary digits are usually referred to as ‘bits’.

All data inside a computer system are stored and manipulated using a binary code. However, if there is ever a need to document some of this binary code outside of the computer system it is not helpful to use the internal code.

Instead, it is far better to use a hexadecimal representation for documentation purposes. Whether or not a code represents a binary number, it can be treated as such and converted to the corresponding hexadecimal number. This makes the representation more compact and, as a result, more intelligible.

Hexadecimal numbers are in the base-16 system and therefore require 16 individual symbols to represent a number. The symbols chosen are 0–9 supplemented with A–F. A few examples of the hexadecimal representation of binary numbers represented by eight bits are shown in Table 1.01.

Binary	Hexadecimal	Denary
00001000	08	8
00001010	0A	10
00001111	0F	15
11111111	FF	255

Table 1.01 Hexadecimal representations of binary numbers and the denary values

Note that each grouping of four bits is represented by one hexadecimal symbol. Also note that it is common practice to include leading zeros in a hexadecimal number when used in this way.

Question 1.01

Does a computer ever use hexadecimal numbers?

Converting between binary and denary numbers

To convert a binary number to a denary number the straightforward method is to sum the individual position values knowing that the least significant bit represents 2⁰, the next one 2¹ and so on. This is illustrated by conversion of the binary number 11001 as shown in Figure 1.01.

Cambridge International AS and A level Computer Science

Position values	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
Binary digits	1	1	0	0	1

Figure 1.01 Position values for a binary number

Starting from the least significant bit, the denary equivalent is $1 + 0 + 0 + 8 + 16 = 25$.

An alternative method is to use the fact that 1×16 is equal to 2×8 and so on. To carry out the conversion you start at the most significant bit and successively multiply by two and add the result to the next digit:

$1 \times 2 = 2$
add 2 to 1, then $2 \times 3 = 6$
add 6 to 0, then $2 \times 6 = 12$
add 12 to 0, then $2 \times 12 = 24$
add 24 to 1 to give 25.

When converting a denary number to binary the procedure is successive division by two with the remainder noted at each stage. The converted number is then given as the set of remainders in reverse order.

This is illustrated by the conversion of denary 246 to binary:

$246 \div 2 \rightarrow 123$ with remainder 0
 $123 \div 2 \rightarrow 61$ with remainder 1
 $61 \div 2 \rightarrow 30$ with remainder 1
 $30 \div 2 \rightarrow 15$ with remainder 0
 $15 \div 2 \rightarrow 7$ with remainder 1
 $7 \div 2 \rightarrow 3$ with remainder 1
 $3 \div 2 \rightarrow 1$ with remainder 1
 $1 \div 2 \rightarrow 0$ with remainder 1

Thus the binary equivalent of denary 246 is 11110110. As a check that the answer is sensible, you should remember that you are expecting an 8-bit binary number because the largest denary number that can be represented in seven bits is $2^7 - 1$ which is 127. Eight bits can represent values from 0 to $2^8 - 1$ which is 255.

Converting hexadecimal numbers

To convert a hexadecimal number to binary, each digit is treated separately and converted into a 4-bit binary equivalent, remembering that F converts to 1111, E converts to 1110 and so on. Subsequent conversion of the resulting binary to denary can then be done if needed.

To convert a binary number to hexadecimal you start with the four least significant bits and convert them to one hexadecimal digit. You then proceed upwards towards the most significant bit, successively taking groupings of four bits and converting each grouping to the corresponding hexadecimal digit.

It is possible to convert a denary number directly to hexadecimal but it is easier to convert first to binary before completing the conversion.

TASK 1.01

- Convert the denary number 374 into a hexadecimal number.
- Convert the hexadecimal number 3A2C to a denary number.

1.02 Internal coding of numbers

The discussion here relates only to the coding of integer values. The coding of non-integer numeric values (real numbers) is considered in Chapter 16 (Section 16.03).

It is convenient at this point to emphasise that the coding used in a computer system is almost exclusively based on bits being grouped together with eight bits representing a **byte**. A byte, or a group of bytes, might represent a binary value but equally might represent a code. For either case, the right-hand bit is referred to as the least significant and the left-hand bit as the most significant or top bit. Furthermore, the bits in a byte are numbered right to left starting at bit 0 and ending at bit 7.



KEY TERMS

Byte: a group of eight bits treated as a single unit

Coding for integers

Computers have to store integer values for a number of purposes. Sometimes the requirement is only for an unsigned integer to be stored. However, in many cases a signed integer is needed where the coding has to identify whether the number is positive or negative.

An unsigned integer can be stored simply as a binary number. The only decision to be made is how many bytes should be used. If the choice is to use two bytes (16 bits) then the range of values that can be represented is 0 to $2^{16} - 1$ which is 0 to 65535.

If a signed integer is to be represented, the obvious choice is to use one bit to represent the + or – sign. The remaining bits then represent the value. This is referred to as ‘sign and magnitude representation’. However, there are a number of disadvantages in using this format.

The approach generally used is to store signed integers in **two’s complement** form. Here we need two definitions. The **one’s complement** of a binary number is defined as the binary number obtained if each binary digit is individually subtracted from 1 which, in practice, means that each 0 is switched to 1 and each 1 switched to 0. The two’s complement is defined as the binary number obtained if 1 is added to the one’s complement number.



KEY TERMS

One’s complement: the binary number obtained by subtracting each digit in a binary number from 1

Two’s complement: the one’s complement of a binary number plus 1

If you need to convert a binary number to its two’s complement form you can use the method indicated by the definition but there is a quicker method. For this you start at the least significant bit and move left ignoring any zeros up to the first 1 which is also ignored. Any remaining bits are then changed from 0 to 1 or from 1 to 0.

For example, expressing the number 10100100 in two’s complement form leaves the right-hand 100 unchanged then the remaining 10100 changes to 01011 so the result is 01011100.

The differences between a sign and magnitude representation and a two’s complement representation are illustrated in Table 1.02. For simplicity we consider only the values that can be stored in four bits (referred to as a ‘nibble’).

Cambridge International AS and A level Computer Science

Signed denary number to be represented	Sign and magnitude representation	Two's complement representation
+7	0111	0111
+6	0110	0110
+5	0101	0101
+4	0100	0100
+3	0011	0011
+2	0010	0010
+1	0001	0001
+0	0000	0000
−0	1000	Not represented
−1	1001	1111
−2	1010	1110
−3	1011	1101
−4	1100	1100
−5	1101	1011
−6	1110	1010
−7	1111	1001
−8	Not represented	1000

Table 1.02 Representations of signed integers

There are several points to note here. The first is that sign and magnitude representation has a positive and a negative zero which could cause a problem if comparing values. The second, somewhat trivial, point is that there is an extra negative value represented in two's complement.

The third and most important point is that the representations in two's complement are such that starting from the lowest negative value each successive higher value is obtained by adding 1 to the binary code. In particular, when all digits are 1 the next step is to roll over to an all-zero code. This is the same as any digital display would do when each digit has reached its maximum value.

It can be seen that the codes for positive values in the two's complement form are the same as the sign and magnitude codes. However, this fact rather hides the truth that the two's complement code is self-complementary. If a negative number is in two's complement form then the binary code for the corresponding positive number can be obtained by taking the two's complement of the binary code representing the negative number.

TASK 1.02

Take the two's complement of the binary code for −5 and show that you get the code for +5.

WORKED EXAMPLE 1.01

Converting a negative number expressed in two's complement form to the corresponding denary number.

Consider the two's complement binary number 10110001.

Method 1. Convert to the corresponding positive binary number then find the denary value

Converting to two's complement leaves unchanged the 1 in the least significant bit position then changes all of the remaining bits to produce 01001111.

Now using the ‘successive multiplication by two’ method we get (ignoring the 0 in the most significant bit position):

$2 \times 1 = 2$
add 2 to 0, then $2 \times 2 = 4$
add 4 to 0, then $2 \times 4 = 8$
add 8 to 1, then $2 \times 9 = 18$
add 18 to 1, then $2 \times 19 = 38$
add 38 to 1, then $2 \times 39 = 78$
add 78 to 1 to give 79

So the original number is –79 in denary.

Method 2. Sum the individual position values but treat the most significant bit as a negative value

From the original binary number 10110001 this produces the following:

$$\begin{aligned} & -2^7 + 0 + 2^5 + 2^4 + 0 + 0 + 0 + 1 = \\ & -128 + 0 + 32 + 16 + 0 + 0 + 0 + 1 = -79. \end{aligned}$$

Discussion Point:

What is the two’s complement of the binary value 1000? Are you surprised by this?

One final point to make here is that the reason for using two’s complement representations is to simplify the processes for arithmetic calculations. The most important example of this is that the process used for subtracting one signed integer from another is to convert the number being subtracted to its two’s complement form and then to add this to the other number.

TASK 1.03

Using a byte to represent each value, carry out the subtraction of denary 35 from denary 67 using binary arithmetic with two’s complement representations.

Binary coded decimal (BCD)

One exception to grouping bits in bytes to represent integers is the binary coded decimal (BCD) scheme. If there is an application where single denary digits are required to be stored or transmitted, BCD offers an efficient solution. The BCD code uses four bits (a nibble) to represent a denary digit. A four-bit code can represent 16 different values so there is scope for a variety of schemes. This discussion only considers the simplest BCD coding which expresses the value directly as a binary number.

If a denary number with more than one digit is to be converted to BCD there has to be a group of four bits for each denary digit. There are, however, two options for BCD; the first is to store one BCD code in one byte leaving four bits unused. The other option is packed BCD where two 4-bit codes are stored in one byte. Thus, for example, the denary digits 8503 could be represented by either of the codes shown in Figure 1.02.

One BCD digit per byte	00001000	00000101	00000000	00000011
Two BCD digits per byte	10000101	00000011		

Figure 1.02 Alternative BCD representations of the denary digits 8503

Cambridge International AS and A level Computer Science

There are a number of applications where BCD can be used. The obvious type of application is where denary digits are to be displayed, for instance on the screen of a calculator or in a digital time display. A somewhat unexpected application is for the representation of currency values. When a currency value is written in a format such as \$300.25 it is as a fixed-point decimal number (ignoring the dollar sign). It might be expected that such values would be stored as real numbers but this cannot be done accurately (this type of problem is discussed in more detail in Chapter 16 (Section 16.03). One solution to the problem is to store each denary digit in a BCD code.

It is instructive to consider how BCD arithmetic might be performed by a computer if fixed-point decimal values were stored as BCD values. Let's consider a simple example of addition to illustrate the potential problem. We will assume a two-byte representation. The first byte represents two denary digits for the whole part of the number and the second byte represents two denary digits for the fractional part. If the two values are \$0.26 and \$0.85 then the result should be \$1.11. Applying simple binary addition of the BCD codes will produce the result shown in Figure 1.03.

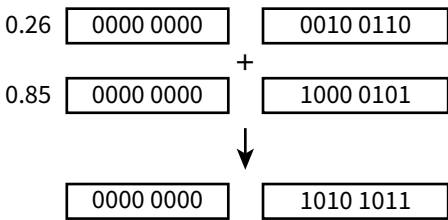


Figure 1.03 Erroneous addition using BCD coding

In the first decimal place position, the 2 has been added to the 8 to get 10 but the BCD scheme only recognises binary codes for a single-digit denary number so the addition has failed. The same problem has occurred in the addition for the second decimal place values. The result shown is 'point ten eleven', which is meaningless in denary numbers. The 'carry' of a digit from one decimal place to the next has been ignored.

To counteract this in BCD arithmetic, the processor needs to recognise that an impossible value has been produced and apply a method to remedy this. We will not consider the recognition method. The remedy is to add 0110 whenever the problem is detected.

Starting with the least significant nibble (see Figure 1.04), adding 0110 to 1011 gives 10001 which is a four-bit value plus a carry bit. The carry bit has to be added to the next nibble as well as adding the 0110 to correct the error. Adding 1 to 1010 and then adding 0110 gives 10001. Again the carry bit is added to the next nibble to give the correct result of \$1.11 for the sum of \$0.26 and \$0.85.

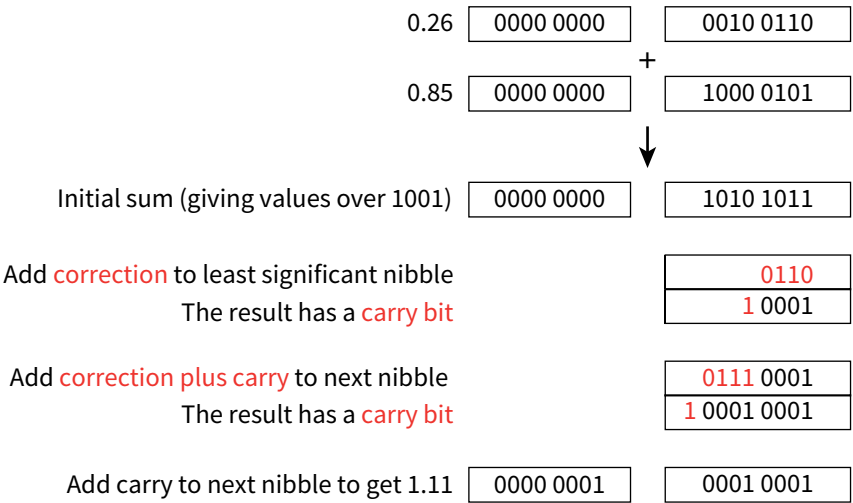


Figure 1.04 Correct representation of the BCD code for 1.11

In Chapter 5 (Section 5.02) there is a brief discussion of how a processor can recognise problems arising from arithmetic operations using numbers coded as binary values.

1.03 Internal coding of text

ASCII code

If text is to be stored in a computer it is necessary to have a coding scheme that provides a unique binary code for each distinct individual component item of the text. Such a code is referred to as a character code. There have been three significant coding schemes used in computing. One of these, which is only mentioned here in passing, is the EBCDIC code used by IBM in their computer systems.

The scheme which has been used for the longest time is the ASCII (American Standard Code for Information Interchange) coding scheme. This is an internationally agreed standard. There are some variations on ASCII coding schemes but the major one is the 7-bit code. It is customary to present the codes in a table for which a number of different designs have been used.

Table 1.03 shows an edited version with just a few of the codes. The first column contains the binary code which would be stored in one byte, with the most significant bit set to zero and the remaining bits representing the character code. The second column presents the hexadecimal equivalent as an illustration of when it can be useful to use such a representation.

Binary code	Hexadecimal equivalent	Character	Description
00000000	00	NUL	Null character
00000001	01	SOH	Start of heading
00000010	02	STX	Start of text
00100000	20		Space
00100001	21	!	Exclamation mark
00100100	24	\$	Dollar
00101011	2B	+	Plus
00101111	2F	/	Forward slash
00110000	30	0	Zero
00110001	31	1	One
00110010	32	2	Two
01000001	41	A	Uppercase A
01000010	42	B	Uppercase B
01000011	43	C	Uppercase C
01100001	61	a	Lowercase a
01100010	62	b	Lowercase b
01100011	63	c	Lowercase c

Table 1.03 Some examples of ASCII codes

The full table shows the 2^7 (128) different codes available for a 7-bit code. You should not try to remember any of the individual codes but there are certain aspects of the coding scheme which you need to understand.

Firstly, you can see that the majority of the codes are for printing or graphic characters. However, the first few codes represent non-printing or control characters. These were introduced to assist in data transmission or in entering data at a computer terminal. It is fair to say that these codes have very limited use in the modern computer world so they need no further consideration.

Cambridge International AS and A level Computer Science

Secondly, it can be seen that the obvious types of character that could be expected to be used in a text based on the English language have been included. Specifically there are upper- and lower-case letters, punctuation symbols, numerals and arithmetic symbols in the coding tables.

It is worth emphasising here that these codes for numbers are exclusively for use in the context of stored, displayed or printed text. All of the other coding schemes for numbers are for internal use in a computer system and would not be used in a text.

There are some special features that make the coding scheme easy to use in certain circumstances. The first is that the codes for numbers and for letters are in sequence in each case so that, for example, if 1 is added to the code for seven the code for eight is produced. The second is that the codes for the upper-case letters differ from the codes for the corresponding lower-case letters only in the value of bit 5. This makes conversion of upper case to lower case, or the reverse, a simple operation.

Unicode

Despite still being widely used, the ASCII codes are far from adequate for many purposes. For this reason new coding schemes have been developed and continue to be developed further. The discussion here describes the Unicode schemes but it should be noted that these have been developed in tandem with the Universal Character Set (UCS) scheme; the only differences between these schemes are the identifying names given to them. The aim of Unicode is to be able to represent any possible text in code form. In particular this includes all languages in the world. However, Unicode is designed so that once a coding set has been defined it is never changed. In particular, the first 128 characters in Unicode are the ASCII codes.

Unicode has its own special terminology. For example, a character code is referred to as a ‘code point’. In any documentation there is a special way of identifying a code point. An example is U+0041 which is the code point corresponding to the alphabetic character A. The 0041 are hexadecimal characters representing two bytes. The interesting point is that in a text where the coding has been identified as Unicode it is only necessary to use a one-byte representation for the 128 codes corresponding to ASCII. To ensure such a code cannot be misinterpreted, the codes where more than one byte is needed have restrictions applied. Figure 1.05 shows the format used for a two-byte code.

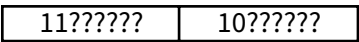


Figure 1.05 Unicode two-byte code format


The most significant bit for an ASCII code is always 0 so neither of the two-byte representations here can cause confusion.

1.04 Images

Images can be stored in a computer system for the eventual purpose of displaying the image on a screen or for presenting it on paper, usually as a component of a document. Such an image can be created by using an appropriate drawing package. Alternatively, when an image already exists independently of the computer system, the image can be captured by using photography or by scanning.

Vector graphics

It is normal for an image that is created by a drawing package or a computer-aided design (CAD) package to consist of a number of geometric objects. The outcome is then usually for the image to be stored as a **vector graphic** file.



KEY TERMS

Vector graphic: a graphic consisting of components defined by geometric formulae and associated properties, such as line colour and style

We do not need to consider how an image of this type would be created. We do need to consider how the data is stored after the image has been created. A vector graphic file contains a drawing list. The list contains a command for each object included in the image. Each command has a list of attributes that define the properties of the object. The properties include the basic geometric data such as, for a circle, the position of the centre and its radius. In addition properties such as the thickness and style of a line, the colour of a line and the colour that fills the shape, if that is appropriate, are defined. An example of what could be created as a vector graphic file is shown in Figure 1.06.

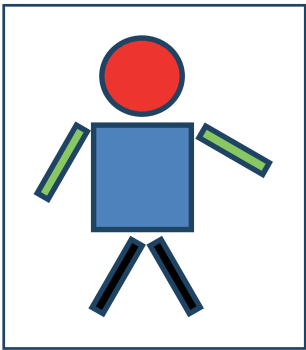


Figure 1.06 A simple example of a vector graphic image

The most important property of a vector graphic image is that the dimensions of the objects are not defined explicitly but instead are defined relative to an imaginary drawing canvas. In other words, the image is scalable. Whenever the image is to be displayed the file is read, the appropriate calculations are made and the objects are drawn to a suitable scale. If the user then requests that the image is redrawn at a larger scale the file is read again and another set of calculations are made before the image is displayed. This process cannot of itself cause distortion of the image.

TASK 1.04

Construct a partial drawing list for the graphic shown in Figure 1.06. You can take measurements from the image and use the bottom left corner of the box as the origin of a coordinate system. You can invent your own format for the drawing list.

A vector graphic file can only be displayed directly on a graph plotter, which is an expensive specialised piece of hardware. Otherwise the file has to be converted to a bitmap before presentation.

Bitmaps

Most images do not consist of geometrically defined shapes so a vector graphic representation is inappropriate. The general purpose approach is to store an image as a bitmap. Typical uses are when capturing an existing image by scanning or perhaps by taking a screen-shot. Alternatively, an image can be created by using a simple drawing package.

The fundamental concept underlying the creation of a bitmap file is that the **picture element (pixel)** is the smallest identifiable component of a bitmap image. The image is stored as a two-dimensional matrix of pixels. The pixel itself is a very simple construct; it has a position in the matrix and it has a colour.