Chapter 1:
Information representation

Learning objectives

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

- show understanding of binary magnitudes and the difference between binary prefixes and decimal prefixes
- show understanding of the basis of different number systems
- perform binary addition and subtraction
- describe practical applications where Binary Coded Decimal (BCD) and Hexadecimal are used
- show understanding of and be able to represent character data in its internal binary form, depending on the character set used
- show understanding of how data for a bitmapped image are encoded
- perform calculations to estimate the file size for a bitmap image
- show understanding of the effects of changing elements of a bitmap image on the image quality and file size
- show understanding of how data for a vector graphic are encoded
- justify the use of a bitmap image or a vector graphic for a given task
- show understanding of how sound is represented and encoded
- show understanding of the impact of changing the sampling rate and resolution
- show understanding of the need for and examples of the use of compression
- show understanding of lossy and lossless compression and justify the use of a method in a given situation
- show understanding of how a text file, bitmap image, vector graphic and sound file can be compressed.

1.01 Number systems

Denary numbers

As a child we first encounter the numbers that we use in everyday life when we are learning to count. Specifically, we learn to count using 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. This gives us ten different symbols to represent each individual digit. This is therefore a base-10 number system. Numbers in this system are called **denary numbers** or, more traditionally, decimal numbers.

When a number is written down the value that it represents is defined by the place values of the digits in the number. This can be illustrated by considering the denary number 346 which is interpreted as shown in Table 1.01.

Place value	$10^2 = 100$	$10^1 = 10$	$10^0 = 1$
Digit	3	4	6
Product of digit and place value	300	40	6

Table 1.01 Use of place values in the representation of a denary number

You can see that starting from the right-hand end of the number (which holds the least significant digit), the place value increases by the power of the base number.

Binary numbers

The binary number system is base-2. Each binary digit is written with either of the symbols 0 and 1. A binary digit is referred to as a **bit**.

As with a denary number, the value of a binary number is defined by place values. For example, see Table 1.02 for the binary number 101110.

Place value	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
Digit	1	0	1	1	1	0
Product of digit and place value	32		8	4	2	0

Table 1.02 Use of place values in the representation of a binary number

By adding up the values in the bottom row you can see that the binary number 101110 has a value which is equivalent to the denary number 46.

You must be able to use the binary number system in order to understand computer systems. This is because inside computer systems there is no attempt made to represent ten different digits individually. Instead, all computer technology is engineered with components that represent or recognise only two states: ‘on’ and ‘off’. To match this, all software used by the hardware uses binary codes which consist of bits. The binary code may represent a binary number but this does not have to be the case.

Binary codes are most often based on the use of one or more groups of eight bits. A group of eight bits is called a **byte**.

Hexadecimal numbers

These are base-16 numbers where each hexadecimal digit is represented by one of the following symbols: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. The symbols A through to F represent the denary values 10 through to 15. The value of a number is defined by place values. For example, see Table 1.03 for the hexadecimal number 2A6.

Place value	$16^2 = 256$	$16^1 = 16$	$16^0 = 1$
Digit	2	A	6

Product of digit and place value	512	160	6

Table 1.03 Use of place values in the representation of a hexadecimal number

Adding up the values in the bottom row shows that the equivalent denary number is 678.

In order to explain why hexadecimal numbers are used we need first to define the **nibble** as a group of four bits.

A nibble can be represented by one hexadecimal digit. This means that each byte of binary code can be written as two hexadecimal digits. Two examples are shown in Table 1.04 together with their denary equivalent.

Binary	Hexadecimal	Denary
00001010	0A	10
11111111	FF	255

Table 1.04 Examples of a byte represented by two hexadecimal digits

Note here that if you were converting the binary number 1010 to a hexadecimal number as an exercise on a piece of paper you would not bother with including leading zeros. However, a binary code must not include blanks; all positions in the byte must have either a 0 or a 1. This is followed through in the hexadecimal representation.

One example when you will see hexadecimal representations of bytes is when an error has occurred during the execution of a program. A memory dump could be provided which has a hexadecimal representation of the content of some chosen part of the memory. Another use is when the bytes contain binary numbers in the charts that define character codes. This is discussed later in this chapter.

In the character code charts and in other online sources you may see references to octal numbers which are base-8. You can ignore these.

Converting between binary and denary numbers

One method for converting a binary number to a denary number is to add up the place values for every digit that has a value 1. This was illustrated in Table 1.02.

An alternative method is shown in Worked Example 1.01.

WORKED EXAMPLE 1.01

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. The following shows the method being used to convert the binary number 11001 to the denary number 25:

	1	×	2	=	2
add 2 to 1, then	2	×	3	=	6
add 6 to 0, then	2	×	6	=	12
add 12 to 0, then	2	×	12	=	24

add 24 to 1 to give 25.

To convert a denary number to binary begin by identifying the largest power of 2 that has a value less than the denary number. You can then write down the binary representation of this power of 2 value. This will be a 1 followed by the appropriate number of zeros.

Now subtract the power of two value from the denary number. Then identify the largest power of 2 value that is less than the remainder from the subtraction. You can now replace a zero in the binary representation with a 1 for this new power of 2 position.

Repeat this process until you have accounted for the full denary number.

For example, for the denary number 78 the largest power of two value less than this is 64 so you can start by writing down 1000000. The remainder after subtracting 64 from 78 is 14. The largest power of two value less than this is 8 so the replacement of a zero by 1 gives 1001000. Repeating the process finds values of 4 then 2 so the final answer is 1001110.

An alternative approach is shown in Worked Example 1.02.

WORKED EXAMPLE 1.02

A useful way to convert a denary value to its binary equivalent is the procedure of successive division by two with the remainder written down at each stage. The converted number is then given as the set of remainders in reverse order.

This can be illustrated by the conversion of denary 246 to binary:

246	÷	2	→	123	with remainder 0
123	÷	2	→	61	with remainder 1
61	÷	2	→	30	with remainder 1
30	÷	2	→	15	with remainder 0
15	÷	2	→	7	with remainder 1
7	÷	2	→	3	with remainder 1
3	÷	2	→	1	with remainder 1
1	÷	2	→	0	with remainder 1

Thus, the binary equivalent of denary 246 is 11110110.

TIP

To check that an answer with eight bits is sensible, remember that the maximum denary value possible in seven bits is $2^7 - 1$ which is 127 whereas eight bits can hold values up to $2^8 - 1$ which is 255.

Conversions for hexadecimal numbers

It is possible to convert a hexadecimal number to denary by using the method shown in Table 1.03. However, if there are more than a few digits, the numbers involved in the conversion become very large. Instead, the sensible approach is to first convert the hexadecimal number to a binary number which can then be converted to denary.

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.

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.

TASK 1.01

Convert each of the denary numbers 96, 215 and 374 into hexadecimal numbers.

Convert each of the hexadecimal numbers B4, FF and 3A2C to denary numbers.

Question 1.01

Does a computer ever use hexadecimal numbers?

1.02 Numbers and quantities

There are several different types of numbers within the denary system. Examples of these are provided in Table 1.05.

Type of number	Examples	Comments
Integer	3 or 47	A whole number used for counting
Signed integer	−3 or 47	The positive number has an implied + sign
Fraction	2/3 or 52/17	Rarely used in computer science
A number with a whole number part and a fractional number part	−37.85 or 2.83	The positive number has an implied + sign
A number expressed in exponential notation	-3.6×10^{8} or 4.2×10^{-9}	The value can be positive or negative and the exponent can be positive or negative

Table 1.05 Different ways to express a value using the denary number system

We will focus on how large values are represented. If we have a quantity that includes units of measurement, it can be written in three different ways. For example, a distance could be written in any one of these three ways:

- 23 567 m
- 23.567×10^{3} m
- 23.567 km

The second example has used an exponential notation to define the magnitude of the value. The third example has added a prefix to the unit to define this magnitude. We read this as 23.567 kilometres.

The ‘kilo’ is an example of a **decimal prefix**. There are four decimal prefixes commonly used for large numbers. These are shown in Table 1.06.

Decimal prefix name	Symbol used	Factor applied to the value
kilo	k	10^{3}
mega	M	10^{6}
giga	G	10^{9}
tera	T	10^{12}

Table 1.06 The decimal prefixes

Unfortunately, for a long time the computing world used these prefix names but with a slightly different definition. The value for 2^{10} is 1024. Because this is close to 1000, computer scientists decided that they could use the kilo prefix to represent 1024. So, for example, if a computer system had the following values quoted for the processor speed and the size of the memory and of the hard disk:

Processor speed	1.6 GHz
Size of RAM	8 GB
Size of hard disk	400 GB

The prefix G would represent 10^{9} for the processor speed but would almost certainly represent $1024 \times 1024 \times 1024$ for the other two values.

This unsatisfactory situation has now been resolved by the definition of a new set of names which can be used to define a **binary prefix**. A selection of these is shown in Table 1.07.

Binary prefix name	Symbol used	Factor applied to the value
kibi	Ki	2^{10}
mebi	Mi	2^{20}
gibi	Gi	2^{30}
tebi	Ti	2^{40}

Table 1.07 Some examples of binary prefixes

When a number or a quantity is presented for a person to read it is best presented with either one denary digit or two denary digits before the decimal point. If a calculation has been carried out, the initial result found may not match this requirement. A conversion of the presented value will be needed by choosing a sensible magnitude factor. For example, consider the following two answers calculated for the size of a file:

a 34 560 bytes

Here, a conversion to kibibytes would be sensible using the calculation:

344560B=345601024KiB=33.75 KiB

b 3 456 000 bytes

Here, a conversion to mebibytes would be sensible using the calculation:

3456000B=(34560001024)1024MiB=3.296 MiB

If a calculation is to be performed with values quoted with different magnitude factors there must first be conversions to ensure all values have the same magnitude factor. For example, if you needed to know how many files of size 2.4 MiB could be stored on a 4 GiB memory stick there should be a conversion of the GiB value to the corresponding MiB value.

The calculation would be:

(4×1024)MiB2.4MiB=1076

1.03 Internal coding of numbers

The discussion in this chapter 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).

Coding for integers

Computers need to store integer values for a number of purposes. Sometimes only a simple integer is stored, with the understanding that it is a positive number. This is 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 65 535.

However, in many cases we need to identify whether the number is positive or negative, so we use a signed integer. A signed integer can just have the binary code for the value with an extra bit to define the sign. This is referred to as 'sign and magnitude representation'. For this the convention is to use a 0 to represent + and a 1 to represent –. A few examples of this are shown in Table 1.08.

However, there are a number of disadvantages to using this format, so signed integers are usually 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. This 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.

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 you also ignore. Finally you change any remaining bits 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.

To represent a positive denary integer value as the equivalent two's complement binary form, the process is as follows.

- Use one of methods from Section 1.01 to convert the denary value to a binary value.
- Add a 0 in front of this binary value.

To represent a negative denary integer value as the equivalent two's complement binary form the process is as follows.

- Disregard the sign and use one of methods from Section 1.01 to convert the denary value to a binary value.
- Add a 0 in front of this binary value.
- Convert this binary value to its two's complement form.

A few simple examples of two's complement representations are shown in Table 1.08.

To convert a two's complement binary number representing a positive value into a denary value, the leading zero is ignored and one of the methods in Section 1.01 is applied to convert the remaining binary.

There are two alternative methods for converting a two's complement binary number representing a negative number into a denary value. These are illustrated in Worked Example 1.03.

WORKED EXAMPLE 1.03

Methods for 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 convert to denary before adding the minus sign

- Converting 10110001 to two's complement leaves unchanged the 1 in the least significant bit position then changes all of the remaining bits to produce 01001111.
- You ignore the leading zero and apply one of the methods from Section 1.01 to convert the remaining binary to denary which gives 79.
- You add the minus sign to give −79.

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

You follow the approach illustrated in Table 1.02 to convert the original binary number 10110001 as follows:

Place value	-2^7 = −128	2^6 = 64	2^5 = 32	2^4 = 16	2^3 = 8	2^2 = 4	2^1 = 2	2^0 = 1
Digit	1	0	1	1	0	0	0	1
Product	−128	0	32	16	0	0	0	1

You now add the values in the bottom row to get −79.

Some points to note about two's complement representation are as follows.

- There is only one representation of zero.
- 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.
- Just adding a leading zero to an unsigned binary value converts it to the two's complement representation of the corresponding positive number
- You use a two's complement conversion to change the sign of a number from positive to negative or from negative to positive. We say that the two's complement values are self-complementary.
- You can add any number of leading zeros to a representation of a positive value without changing the value.
- You can add any number of leading ones to a representation of a negative value without changing the value.

Signed denary number to be represented	Sign and magnitude representation	Two's complement representation
7	0111	0111
1	0001	0001
0	0000	0000
–0	1000	Not represented
–1	1001	1111
–7	1111	1001
–8	Not represented	1000

Table 1.08 Representations of signed integers

TIP

If you are converting a negative denary number into two's complement you begin by converting the denary value to a binary value. Then you must not forget to add a leading zero before taking the two's complement to convert the positive value to a negative value.

TASK 1.02

Take the two's complement of the binary code for –7 and show that you get the code for +7.

TASK 1.03

Convert the two's complement number 1011 to the denary equivalent. Then do the same for 111011 and convince yourself that you get the same value.

Discussion Point:

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

Binary arithmetic

Before considering the addition of binary numbers it is useful to recall how we add two denary numbers. Two rules apply. The first rule is that the process is carried out starting with addition of the two least significant digits and then working right to left. The second rule is that if an addition produces a value greater than 9 there is a carry of 1. For example in the addition of 48 to 54, the first step is adding 8 to 4 to get 2 with a carry of 1. Then 5 is added to 4 plus the carried 1 to give 0 with carry 1. The rules produce 102 for the sum which is the correct answer.

For binary addition, starting at the least significant position still applies. The rules for the addition of binary digits are:

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 1 = 0 with a carry of 1
- 1 + 1 + 0 = 0 with a carry of 1
- 1 + 1 + 1 = 1 with a carry of 1

The last two rules are used when a carried 1 is included in the addition of two digits.

As an example, the addition of the binary equivalent of denary 14 to the binary equivalent of denary 11 can be examined.

```
      1  0  1  1
+     1  1  1  0
-----------------
   1  1  0  0  1
```

The steps followed from right to left are:

- 1 + 0 = 1 with no carry
- 1 + 1 = 0 with carry 1
- 0 + 1 + carried 1 = 0 with carry 1
- 1 + 1 + carried 1 = 1 with carry 1

The rules have correctly produced the 5-bit answer which is the binary equivalent of 25. In a paper exercise like this these rules for addition will always produce the correct answer.

Again for subtraction we can first consider how this is done for denary numbers. As for addition the process starts with the least significant digits and proceeds right to left. The special feature of subtraction is the “borrowing” of a 1 from the next position when a subtracting digit is larger than the digit it is being subtracted from.

For example in subtracting 48 from 64 the first step is to note that 8 is larger than 4. Therefore 1 has to be borrowed as 10. The 10 added to 4 gives 14 and 8 subtracted from this gives 6. When we proceed to the next digit subtraction we first have to reduce the 6 to 5 because of the borrow. So we have subtraction of 4 from 5 leaving 1. The answer for the subtraction is 16.

For binary subtraction, starting at the least significant position still applies. The rules for the subtraction of binary digits are:

- 0 – 0 = 0
- 0 – 1 = 1 after a borrow
- 1 – 0 = 1
- 1 – 1 = 0

As an example, the subtraction of the binary equivalent of denary 11 from the binary equivalent of denary 14 can be examined.

	1	1	1	0
−	1	0	1	1
	0	0	1	1

The steps followed from right to left are:

- 1 is larger than 0 so 1 is borrowed giving subtraction of 1 from 10 leaving 1
- Because of the borrow the 1 is reduced to 0 so that 1 is to be subtracted from 0. This requires a further borrow giving subtraction of 1 from 10 leaving 1
- Because of the borrow the 1 is reduced to 0 leaving subtraction of 0 from 0
- 1 – 1 gives 0

The answer is the binary value for denary 3.

When binary addition is carried out by a computer using internally stored numbers there is a major difference. This arises from the fact that the storage unit will always have a defined number of bits. For example, in the above addition, if binary values were limited to being stored in a nibble the result of the addition would be incorrectly stored as 1001. This is an example of an **overflow**. The value produced is too large to be stored.

When the values in a computer system are stored in two’s complement form this problem has a characteristic behaviour.

In the following addition where +63 is added to +63 there is no problem; the answer is correctly obtained as +126:

	0	0	1	1	1	1	1	1
+	0	0	1	1	1	1	1	1
	0	1	1	1	1	1	1	0

However, if the binary for +96 is added to +96 the result is as follows:

	0	1	1	0	0	0	0	0
+	0	1	1	0	0	0	0	0
	1	1	0	0	0	0	0	0

The overflow means that the answer has a leading 1, which causes a computer system to interpret the answer as a negative number.

A similar problem can occur when two negative values are added. For example the addition of −96 to the same value results in the following:

		1	0	1	0	0	0	0	0
+		1	0	1	0	0	0	0	0
	(1)	0	1	0	0	0	0	0	0

This time there has been a carry when the most significant bits were added and the result obtained is a positive number.

Clearly we need the processor to detect overflow and output an error message. There is a discussion of how a processor can detect overflow in Chapter 6 (Section 6.07).

One of the advantages of using two's complement representations is that it simplifies the process of subtracting one number from another. The number being subtracted is converted to its two's complement form, which is then added to the other number.

TASK 1.04

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. This is useful in applications that require single denary digits to be stored or transmitted. The BCD code uses a nibble to represent a denary digit. We consider the simple scheme where the digits are coded as the binary values from 0000 to 1001. The remaining codes 1010 to 1111 do not have any meaning.

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 1.01.

One BCD digit per byte	00001000	00000101	00000000	00000011

Two BCD digits per byte	10000101	00000011

Figure 1.01 Alternative BCD representations of the denary digits 8503

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 as a BCD code.

Let's consider how BCD arithmetic might be performed by a computer if fixed-point decimal values for currency were stored as BCD values. Here is an example of addition.

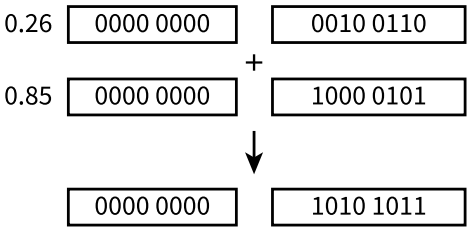

Figure 1.02 Incorrect addition using BCD coding

We will assume a two-byte packed BCD 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 of the addition should be $1.11. This would involve a carry from the first decimal place to the whole number 1. However, applying simple binary addition of the BCD codes would produce the result shown in Figure 1.02.

The additions for the fractional parts have produced values corresponding to the denary values 10 and 11 but a BCD value is supposed to be a single digit. The error has resulted in no carry to the whole number column.

We need the processor to recognise that an impossible value has been produced and apply a method to correct this. The solution is to add 0110 whenever the problem is detected. This is illustrated in Figure 1.03.

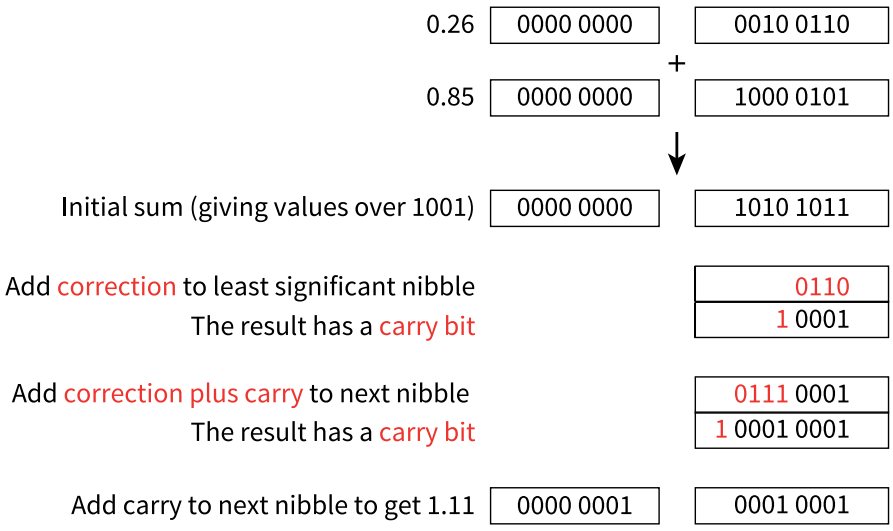

Figure 1.03 Use of the correction value to perform BCD addition

The steps shown in Figure 1.03 are as follows.

- Starting with the least significant nibble, adding 0110 to 0101 gives 1011 which is recognised as being incorrect.
- The 0110 correction value is added to produce 10001.
- The 0001 is stored and the leading 1 is carried to the next nibble.
- In the first decimal position adding 0100 to 1000 then adding the carry bit 1 gives 1011 which is recognised as being incorrect.
- The 0110 correction is added to produce 10001.
- The 0001 is stored and the leading 1 is carried to the next nibble.

In this example the two whole number nibbles have zero values so adding these has no effect.

1.04 Internal coding of text

To store text in a computer, we need 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 many different examples of character coding schemes throughout the history of computing.

ASCII code

The scheme which has been used for the longest time is the ASCII (American Standard Code for Information Interchange) coding scheme. The 7-bit version of the code (often referred to as US ASCII) was standardised many years ago by ANSI (American National Standards Institute). The codes are always presented in a table. Table 1.09 shows an edited version of a typical table. 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 shows the hexadecimal equivalent.

Binary code	Hexadecimal equivalent	Character	Description
00000000	00	NUL	Null character
00000001	01	SOH	Start of heading
00100000	20		Space
00100001	23	#	Number
00110000	30	0	Zero
00110001	31	1	One
01000001	41	A	Uppercase A
01000010	42	B	Uppercase B
01100001	61	a	Lowercase a
01100010	62	b	Lowercase b

Table 1.09 Some examples of ASCII codes stored in one byte with the remaining, most significant bit set to zero

A full table would show the 2^7 (128) different codes available for a 7-bit code.

TIP

Do not try to remember any of the individual codes

You need to remember these key facts about the ASCII coding scheme.

- A limited number of the codes represent non-printing or control characters; these were introduced to assist in data transmission or for data handling at a computer terminal.
- The majority of the codes are for characters that would be found in an English text and which are available on a standard keyboard.
- These include upper- and lower-case letters, punctuation marks, denary digits and arithmetic symbols.
- The codes for numbers and for letters are in sequence so that, for example, if 1 is added to the code for seven, the code for eight is produced.
- The codes for the upper-case letters differ from the codes for the corresponding lower-case letters only in the value of bit 5, which allows a simple conversion from upper to lower case or the reverse. (Don’t forget that the least significant bit is bit 0.)

Note that this coding for numbers is 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.

Although a standard version of ASCII has been created, different versions of 7-bit ASCII are tailored to different software or different countries. Mostly, the coding for the printable characters has remained unchanged. A notable exception was the use in some countries of the code 00100001 to represent a currency symbol rather than #. However, because most of the control characters became of limited use, there were versions of ASCII that used these codes to produce small graphic icons. For example, the code 00000001 would show ☺.

Extended ASCII is a code that uses all eight bits in a byte. The most used standardised version is often referred to as ISO Latin-1. The name Latin-1 reflects the fact that many of the new character definitions are for accented or otherwise modified alphabetic characters found in European languages, for example Ç or ü. As with the 7-bit code, there are many variations of the standard code.

Question 1.02

Many years ago, a byte was defined as six bits. If a character was to be represented by one byte, which characters would you expect to be representable and which ones would you expect to be unavailable?

Unicode

Although ASCII codes are widely used, they do not cover all the characters needed for some uses. For this reason, new coding schemes have been developed and continue to be developed further. The discussion here describes one of the Unicode schemes. It should be noted that Unicode codes have been developed in tandem with the Universal Character Set (UCS) scheme, standardised as ISO/IEC 10646.

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. The most popular version of Unicode which is discussed here is named UTF-8. The inclusion of 8 in the name indicates that this version of the standard includes codes defined by one byte in addition to codes using two, three and four bytes.

Figure 1.04 shows the structure of the codes. The 1 byte code reproduces 7-bit ASCII. Because the byte has the most significant bit set to 0 there can be no confusion with any byte which is part of a multiple byte code. Note that for the two-byte, three-byte and four-byte representations all continuing bytes have the two most significant bits set to 10. Whenever a byte has the most significant bits set to 11 there will be at least one continuation byte following.

0???????			
110?????	10??????		
1110????	10??????	10??????	
11110???	10??????	10??????	10??????

Figure 1.04 Byte formats for Unicode UTF-8

The number of codes available is determined by the number of bits that are not pre-defined by the format. For example, there are eleven bits free to identify codes in the 2-byte format. This allows 2^{11} = 2048 different codes.

Unicode has its own special terminology and symbolism. A character code is referred to as a ‘code point’. In any documentation a code point is identified by U+ followed by a 4-digit hexadecimal number. The code points U+0000 to U+00FF define characters which are a duplicate of those in the standard Latin-1 scheme. The binary codes corresponding to U+0000 to U+007F use one byte only and range from 00000000 through to 01111111. Then the binary codes for U+0080 to U+00FF require two bytes and range from 11000000 for the first byte followed by 10000000 for the second byte through to 11000001 followed by 10111111.

1.05 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 graphics 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

In an image that is created by a drawing package or a computer-aided design (CAD) package each component is an individual **drawing object**. The image is then stored, usually as a **vector graphic** file.

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, each attribute defines a **property** 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 are defined such as the thickness and style of a line, the colour of a line and the colour that fills the shape. An example of what could be created as a vector graphic file is shown in Figure 1.05.

TASK 1.05

Construct a partial drawing list for the graphic shown in Figure 1.05. 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.

Figure 1.05 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 avoids image distortion, such as the image appearing squashed or stretched.

Note that a vector graphic file can only be displayed directly on a graph plotter, which is an expensive specialised piece of hardware. For the image to appear correctly on other types of display, the vector graphic file often has to be converted to a bitmap.

Bitmaps

Most images do not consist of geometrically defined shapes, so a vector graphic representation is inappropriate. Instead, generally an image is stored 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 graphics 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. It does not matter whether each pixel is a small rectangle, a small circle or a dot.

The scheme used to represent the colour has to be defined. The simplest option is to use one bit to represent the colour, so that the pixel is either black or white. Storage of the colour in four bits allows simple greyscale colouring. At least eight bits per pixel are necessary to provide a sufficient range of colours to provide a reasonably realistic representation of any image. The number of bits per pixel is sometimes referred to as the **colour depth**.

An alternative definition is the **bit depth**. Although these terms are sometimes used interchangeably, bit depth is best defined as the number of bits used to store each of the red, green and blue primary colours in the RGB colour scheme.

A colour depth of 8 bits per pixel provides 256 different colours. A bit depth of 8 bits per primary colour provides 256 × 256 × 256 = 16 777 216 different colours. The eye cannot distinguish this number of different colours. However, this many are needed if an image contains areas of gradually changing colour such as in a picture of the sky. If a lower bit depth is used the image will show bands of colour.

We also need to decide which resolution to use for the image, which can be represented as the product of the number of pixels per row times the number of rows. When considering resolution it is important to distinguish between an **image resolution**, as defined in a bitmap file, and a **screen resolution** for a particular monitor screen that might be used to display the image. Both of these have to be considered if a screen display is being designed.

A bitmap file does not define the physical size of a pixel or of the whole image. When the image is scaled the number of pixels in it does not change. If a well-designed image is presented on a suitable screen the human eye cannot distinguish the individual pixels. However, if the image is magnified too far the individual pixels will be seen. This is illustrated in Figure 1.06 which shows an original small image, a magnified version of this small image and a larger image created with a more sensible, higher resolution.

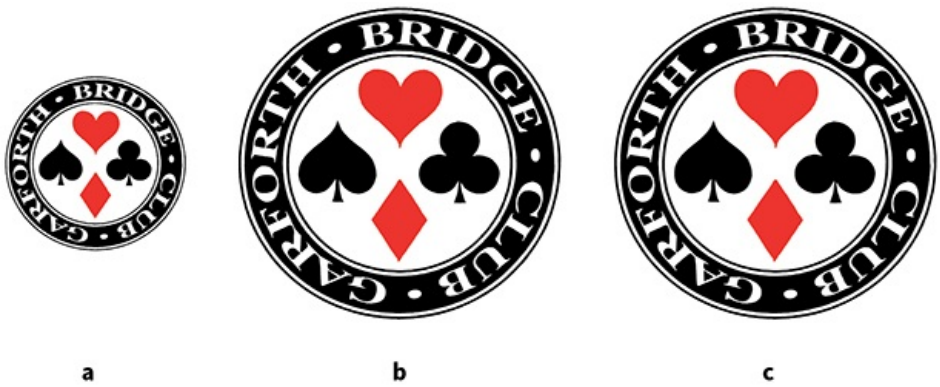

Figure 1.06 (a) a bitmap logo; (b) an over-magnified version of the image; (c) a sensible larger version

File size is always an issue with an image file. A large file occupies more memory space and takes longer to display or to be transmitted across a network. Usually, a vector graphic file uses considerably less memory space than a corresponding bitmap file.

You can calculate the size of a bitmap graphic knowing the resolution and the colour depth. As an example, consider that a bitmap graphic is needed to fill a laptop screen where the resolution is 1366 by 768. If we want colour depth of 24 then the number of bits we need is:

$$1366 \times 768 \times 24 = 25\ 178\ 112 \text{ bits}$$

The result of this calculation shows the number of bits, but a size is always quoted as a number of bytes or multiples of bytes. For our bitmap graphic:

$$25\ 178\ 112 \text{ bits} = 25\ 178\ 112 \div 8 = 3\ 147\ 264 \text{ bytes}$$

$$= 3\ 147\ 264 \div 1024 = 3073.5 \text{ kibibytes (3073.5 KiB)}$$

= 3073.5 ÷ 1024 = approximately 3 MiB

Note that this calculation has assumed that the colour depth specifies the total number of bits used to define each pixel. If the information given was that the bit depth was eight, then the calculation would use 8 + 8 + 8 for the number of bits per pixel.

WORKED EXAMPLE 1.04

You have been asked to calculate a value for the minimum size of a bitmap file. The bitmap is to use a bit depth of 8 and the bitmap is to be printed with 72 dpi (dots per inch) and to have dimensions 5 inches by 3 inches.

We use the information provided about the colour depth or the bit depth to give the number of bits per pixel. In this case the bit depth is 8, which means 8 bits for each of the RGB components, so 24 bits are needed for one pixel.

Let's state that 72 dpi means 72 pixels per inch.

So, the number of pixels per row is 5 × 72 = 360

And the number of pixels per column is 3 × 72 = 216

Therefore, the total number of pixels is 360 × 216 = 77 760

The total number of bits is this value multiplied by 24. However, we want the size in bytes not bits, so we multiply by 3 because there are 8 bits in a byte. So, we get:

77 760 × 3 = 233 280 bytes.

We can quote this in kibibytes by dividing by 1024:

233 280 / 1024 = 227.8 KiB

A bitmap file has to store the pixel data that defines the graphic, but the file must also have a **file header** that contains information on how the graphic has been constructed. Because of this, the bitmap file size is larger than the size of the graphic alone. At the very least the header will define the colour depth or bit depth and the resolution.

The following are considerations when justifying the use of either a bit map or a vector graphic for a specific task.

- A vector graphic is chosen if a diagram is needed to be constructed for part of an architectural, engineering or manufacturing design.
- If a vector graphic file has been created but there is a need to print a copy using a laser or inkjet printer the file has first to be converted to a bitmap.
- A digital camera automatically produces a bitmap.
- A bitmap file is the choice for insertion of an image into a document, publication or web page.

1.06 Sound

Natural sound consists of variations in pressure which are detected by the human ear. A typical sound contains a large number of individual waves, each with a defined frequency. The result is a wave form in which the amplitude of the sound varies in a continuous but irregular pattern.

If we want to store sound or transmit it electronically the original analogue sound signal has to be converted to a binary code. The measured sound values are input to a sound encoder which has two components. The first is a band-limiting filter. This is needed to remove high-frequency components. A human ear cannot detect these very high frequencies and they could cause problems for the coding if not removed. The other component in the encoder is an analogue-to-digital converter (ADC) which converts the **analogue data** to **digital data**.

Figure 1.07 shows the **sampling** operation of the ADC. The amplitude of the wave (the red line) has to be sampled at regular intervals. The blue vertical lines indicate the sampling times. The amplitude cannot be measured exactly; instead the amplitude is approximated by the closest of the defined amplitudes represented by the horizontal lines. In Figure 1.07, sample values 1 and 4 will be an accurate estimate of the actual amplitude because the wave is touching an amplitude line. In contrast, samples 5 and 6 will not be accurate because the actual amplitude is approximately half way between the two closest defined values.

To code sound, we need to make two decisions. The first is the number of bits we will use to store the amplitude values, which defines the **sampling resolution**. If we use only three bits then eight levels can be defined as shown in Figure 1.07. If too few are used there will be a significant error when the closest amplitude in the scale of values dictated by the sampling resolution is used as the approximation for the real value. In practice, 16 bits provides reasonable accuracy for most digitised sound.

We also need to choose the **sampling rate**, which is the number of samples taken per second. This should be in accordance with Nyquist's theorem which states that sampling must be done at a frequency at least twice the highest frequency of the sound in the sample.

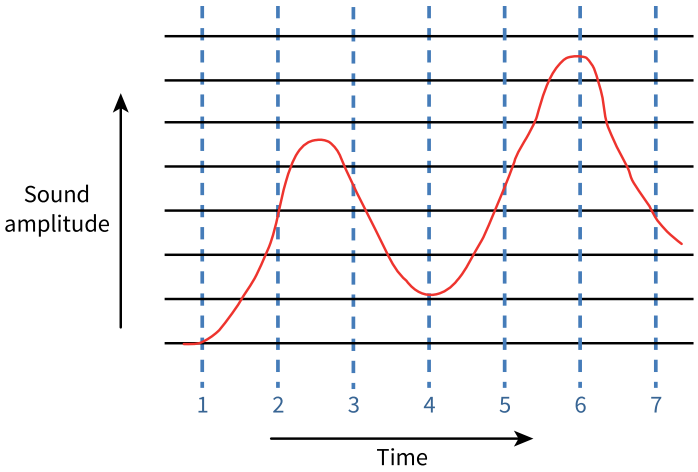

Figure 1.07 ADC sampling

Once again file size can be an issue. An increased sampling rate and an increased sampling resolution will both cause an increase in file size.

1.07 Compression techniques

Larger files require larger storage capacity but more importantly, larger files have lower transmission or download rates. For this reason, compression techniques are often used to reduce file size.

There are two categories of compression. The first is **lossless compression** where the file size is reduced but no information is lost. The process can be reversed to re-create the original file. The second is **lossy compression** where the file size is reduced with some loss of information and the exact original file can never be recovered. In many applications a combination of lossless and lossy methods are used.

We could use the same type of lossless file compression for everything, because all files contain binary codes. A good compression application will recognise patterns in files that it can compress, without any knowledge of what file type the code represents. However, most compression techniques have been developed to work with a particular type of file.

A common lossless compression technique is run-length encoding. This works particularly well with a bitmap file. The idea is that compression converts sequences of the same byte value into a code that defines the byte value and the number of times it is repeated (the count).

For example, the sequence of the same four bytes:

01100110 01100110 01100110 01100110

could be replaced by:

00000100 01100110

which says that there is a run of four of the bytes.

However, this is not the full story because in this simple form it is not obvious which byte represents the number (count) in the sequence. There are a number of methods used to distinguish the count byte from a data byte, but we do not need to go into the details.

If a file contains text, then compression must be lossless because any loss of information would lead to errors in the text. One possible compression method is called Huffman coding. The procedure used to carry out the compression is quite detailed, but the principle is straightforward. Instead of having each character coded in one byte, the text is analysed to find the most often used characters. These are then given shorter codes. The original stream of bytes becomes a bit stream.

A possible set of codes if a text contained only eight different letters is shown in Table 1.10. The important point to note here is the prefix property. None of the codes begins with the sequence of bits representing a shorter code. This means that there can be no ambiguity when the transmitted compressed file has to be converted back to the original text.

Code	Character
10	e
01	t
111	o
110	h
0001	l
0000	p
0011	w
0010	z

Table 1.10 An example of Huffman coding

Huffman coding can also be used for compressing a sound file. This is effective because some values for

the amplitude occur far more often than others do.

If a vector graphic file needs to be compressed it is best converted to a Scalable Vector Graphics format. This uses a markup language description of the image which is suitable for lossless compression.

Lossy compression can be used in circumstances where a sound file or an image file can have some of the detailed coding removed or modified. This can happen when it is likely that the human ear or eye will hardly notice any difference. One method for lossy compression of a sound file takes advantage of the fact that the successive sampled values are unlikely to change very much. The file of individual sample amplitudes can be converted to a file of amplitude differences. Compression is achieved by using a lower sample resolution to store the differences. An alternative is to convert the sampled amplitudes that represent time domain data and transform them to a frequency domain representation. The values for frequencies that would be barely audible are then re-coded with fewer bits before the data is transformed back to the original time domain form.

For a bitmap a simple lossy compression technique is to establish a coding scheme with reduced colour depth. Then for each pixel in the original bitmap the code is changed to the one in the new scheme which represents the closest colour.

Extension Question 1.01

Graphic files can be stored in a number of formats. For example, JPEG, GIF, PNG and TIFF are just a few of the possibilities. What compression techniques, if any, do these use?

Reflection Point:

Can you recall the different possibilities for what one byte might be coded to represent?

Summary

- A binary code or a binary number can be documented as a hexadecimal number.
- Internal coding of signed integers is usually based on a two’s complement representation.
- Binary addition can cause overflow.
- BCD is a convenient coding scheme for single denary digits.
- ASCII and Unicode are standardised coding schemes for text characters.
- An image can be stored either in a vector graphic file or in a bitmap file.
- An ADC works by sampling a continuous waveform.
- Lossless compression allows an original file to be recovered by a decoder; lossy compression irretrievably loses some information.

Exam-style Questions

1 A file contains binary coding. The following are two successive bytes in the file: 10010101 and 00110011

a One possibility for the information stored is that the two bytes together represent one unsigned integer binary number.

i Calculate the denary number corresponding to this. [2]

ii Calculate the hexadecimal number corresponding to this. [2]

b Give **one** example of when a hexadecimal representation is used. [1]

c Another possibility for the information stored is that the two bytes individually represent two signed integer binary numbers in two's complement form.

i State which byte represents a negative number and explain the reason for your choice.

ii Calculate the denary number corresponding to each byte. [3]

d Give **two** advantages of representing signed integers in two's complement form rather than using a sign and magnitude representation. [2]

e Give **three** different examples of other options for the types of information that could be represented by two bytes. For each example, state whether a representation requires two bytes each time, just one byte or only part of a byte each time. [3]

2 A designer wishes to include some multimedia components on a web page.

a If the designer has some images stored in files there are two possible formats for the files.

i Describe the approach used if a graphic is stored in a vector graphic file. [2]

ii Describe the approach used if a graphic is stored in a bitmap file. [2]

iii State which format gives better image quality if the image has to be magnified and explain why. [2]

b The designer is concerned about the size of some bitmap files.

i If the resolution is to be 640 × 480 and the colour depth is to be 16, calculate an approximate size for the bitmap file. State the answer using sensible units. [2]

ii Explain why this calculation only gives an approximate file size. [1]

c The designer decides that the bitmap files need compressing.

i Explain how a simple form of lossless compression could be used. [2]

ii Explain **one** possible approach to lossy compression that could be used. [2]

3 An audio encoder is to be used to create a recording of a song. The encoder has two components.

a One of the components is an analogue-to-digital converter (ADC).

i Explain why this is needed. [2]

ii Two important factors associated with the use of an ADC are the sampling rate and the sampling resolution. Explain the two terms. Sketch a diagram if this will help your explanation. [5]

b The other component of an audio encoder has to be used before the ADC is used.

i Identify this component. [1]

ii Explain why it is used. [2]

4 **a** **i** Using two's complement, show how the following denary numbers could be stored in an 8-bit register:

124

–77 | | | | | | | | | [2]

ii Convert the two numbers in **part (a) (i)** into hexadecimal. [2]

b Binary Coded Decimal (BCD) is another way of representing numbers.

i Write the number 359 in BCD form. [1]

ii Describe a use of BCD number representation. [2]

Cambridge International AS & A level Computer Science 9608 paper 13 Q1 June 2015

5 **a** Sound can be represented digitally in a computer.

Explain the terms sampling resolution and sampling rate. [4]

b The following information refers to a music track being recorded on a CD:

- music is sampled 44 100 times per second
- each sample is 16 bits
- each track requires sampling for left and right speakers.

i Calculate the number of bytes required to store one second of sampled music. Show your working. [2]

ii A particular track is four minutes long.

Describe how you would calculate the number of megabytes required to store this track. [2]

c When storing music tracks in a computer, the MP3 format is often used. This reduces file size by about 90%.

Explain how the music quality is apparently retained. [3]

Cambridge International AS & A level Computer Science 9608 paper 12 Q4 November 2015