Chapter 9: Data Representation

In this chapter you will learn:

- why computers need data in binary form
- how different data is converted into binary form
- how denary number are represented as hexadecimal
- how to add together binary numbers
- how files are compressed

Why do computers need data in binary form?

OCR specification reference:

- ☑ why data is represented in computer systems in binary form
- ☑ how data needs to be converted into a binary format to be processed by a computer

As humans the data that we process is called **analogue** data. Analogue data is a continuous stream of data, like a sound wave or a light wave. Everything we see or hear is a continuous stream of data to our senses. Computers are not able to process analogue data; they need data to be in a different, **digital** form. This means that any data that we want a computer to process must first be converted to digital form. Digital data is made up of **binary** digits.

Any input into a computer that is analogue will need to be converted to digital to be processed by the computer. The conversion is carried out by an **analogue-to-digital converter (ADC)**. For example, if we plug a microphone into a computer an ADC will convert the analogue sound that is input by our voice and the microphone into a digital form to be processed by the computer.

If we then want to hear the sound that we have input, a digital-to-analogue converter (DAC) is needed to convert the digital data back to analogue, so that we can hear it through a speaker or headphones.

Analogue – the continuous stream of data that our senses process as humans

Digital – data that is binary and represented as 1s and 0s

Binary – a base-2 number system that uses the values 0 and 1

ADC – a device that converts analogue data to digital data

How are numbers converted from denary to binary?

OCR specification reference

- ☑ how to convert positive denary whole numbers (0–255) into 8-bit binary numbers and vice versa

The number system that we use in our daily lives is called the **denary** number system. This is a number system that uses the numbers 0–9. This means that it uses 10 numbers overall. For this reason it is referred to as a base-10 system. In a base-10 system each different unit that is used increases by the power of 10.

Denary – a base-10 number system that uses the values 0 to 9

For example, the number 111 is made up of 1 unit, 1 ten and 1 hundred. A ten is 10 units and a hundred is 10 tens; they increase by the power of 10 each time:

	Increase by power of 10		Increase by power of 10		
Hundreds	←	**Tens**	←	**Units**	
1 hundred		1 ten		1 unit	
1 × 100	+	1 × 10	+	1 × 1	= 111

The number system that computers use is called the binary number system. This is a number system that uses the numbers 0 and 1. This means that it uses two numbers overall. For this reason it is referred to as a base-2 system. In a base-2 system each different unit that is used increases by the power of 2.

Let us look at the binary number 1001:

	Increase by power of 2		Increase by power of 2		Increase by power of 2	
	8	←	4	←	2	← 1
Binary number	1	+	0	+	0	+ 1

To covert the binary number 1001 into denary we need to observe the binary numbers that are used. The number 1 means TRUE and the number 0 means FALSE. When calculating the denary number we need to note which units are TRUE and add them together:

	8		4		2		1
Binary number	1		0		0		1
Denary number	8	+	0	+	0	+	1

Therefore, the binary number 1001 when converted to denary is 8 + 1 = 9

1001 is a 4-bit binary number. We can apply the same principle to an 8-bit binary number. Let us look at the binary number 11001001:

Increase by power of 2		Increase by power of 2		Increase by power of 2		Increase by power of 2		Increase by power of 2		Increase by power of 2		Increase by power of 2	
128	←	64	←	32	←	16	←	8	←	4	←	2	← 1
1		1		0		0		1		0		0	1

Therefore, the binary number 11001001 when converted to denary is 128 + 64 + 8 + 1 = 201.

This is how we convert from binary to denary. We also need to be able to convert from denary to binary. To do this we need to decide which binary units will need to be 1 (TRUE) to create the denary number. There is a method we can use to do this. Let's convert the number 202 into binary.

First of all we write down the eight binary units we might need to use. It is easier to see if we put these into a small table:

128	64	32	16	8	4	2	1

Then we need to work out which of these units are needed. We can do this by starting with the highest unit, 128, and comparing it to our denary number that we are converting. If our denary number is greater than or equal to this unit, then we will need it. 202 is greater than 128 so we can put a 1 below this unit:

128	64	32	16	8	4	2	1
1							

As we have used the unit 128, we can now deduct this from our denary number 202 and we are left with 74. Then we can move to the next unit and compare it to our denary number again, that is now 74. The next unit is 64.

74 is greater than 64 so we can also put a 1 below this unit and then deduct it from our denary number:

128	64	32	16	8	4	2	1
1	1						

74 – 64 is 10; this now becomes our denary number. We can repeat the process by looking at the next unit, which is 32. 10 is not greater than or equal to 32, so we do not use this unit. This means we put a 0 below it:

128	64	32	16	8	4	2	1
1	1	0					

We can now repeat this with the next unit 16. 10 is not greater than or equal to 16, so again we put a 0 below it:

128	64	32	16	8	4	2	1
1	1	0	0				

We move to the next unit. 10 is greater than 8, so we do need this unit and we can put a 1 below it. We then deduct 8 from 10 making our denary value 2:

128	64	32	16	8	4	2	1
1	1	0	0	1			

We move to the next unit. 2 is not greater than or equal to 4, so we put a 0 below it:

128	64	32	16	8	4	2	1
1	1	0	0	1	0		

We move to the next unit. 2 is equal to 2, so we can put a 1 below it and deduct the 2, leaving us with 0:

128	64	32	16	8	4	2	1
1	1	0	0	1	0	1	

As we have now got the denary value to 0, we know we do not need the unit 1. We can put a 0 below it:

128	64	32	16	8	4	2	1
1	1	0	0	1	0	1	0

We now know that the denary number 202 when converted to binary is 11001010.

If we add all the binary units together we would get the denary number:
128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255

This means that the biggest denary number that can be converted using 8-bit binary is 255. If we wanted to convert a number bigger than this we would need to use 12-bit or 16-bit binary. That is beyond what you need to know.

Quick task

Convert the following binary numbers to denary:

1. 10100011
2. 01101001
3. 00100100

Convert the following denary numbers to binary:

1. 99
2. 150
3. 234

How are numbers converted from denary to hexadecimal?

OCR specification reference

☑ how to convert positive denary whole numbers (0–255) into two-digit hexadecimal numbers and vice versa

Hexadecimal – a base-16 number system that uses the characters 0 to 9 and A to F

We have looked at a denary number system and a binary number system so far. There is also a number system called **hexadecimal**. Hexadecimal is a number system that uses the numbers 0 to 9 and the letters A to F. This means that it uses 16 characters overall. For this reason it is referred to as a base-16 system. In a base-16 system each different unit that is used increases by the power of 16. You only need to know hexadecimal to two digits, so the units used will be 1 and 16. Hexadecimal uses 16 characters; numbers remain the same but letters are equal to a denary value:

0 = 0	4 = 4	8 = 8	12 = C
1 = 1	5 = 5	9 = 9	13 = D
2 = 2	6 = 6	10 = A	14 = E
3 = 3	7 = 7	11 = B	15 = F

The reason numbers above 9 need to be represented by a letter is because only one character can be used for each unit in hexadecimal. For example, the number 12 has two characters 1 and 2, so we need to replace it with one character and this is C. Let us look at converting the denary number 40 into hexadecimal. To do this we need to find out how many of the unit 16 and how many of the unit 1 we need:

Increase by power of 16 ←

16	1
1 × 16 = 16	
2 × 16 = 32	

We cannot go any higher with our workings as 3 × 16 = 48 and this is greater than 40. Therefore we know we need 2 of the unit 16. 2 × 16 is 32, so we can deduct 32 from our denary number, leaving us with 8. This means that we know we need 8 of the unit 1:

16	1
2	8

This means that when we convert the number 40 into hexadecimal, we get the hexadecimal value 28.

Let's look at converting the denary number 62 into hexadecimal:

16	1
1 × 16 = 16	
2 × 16 = 32	
3 × 16 = 48	

We cannot go any higher with our workings as 4 × 16 = 64 and this is greater than 62. Therefore we know we need 3 of the unit 16 and can deduct 48 from 62, leaving us with 14. This means we know we need 14 of the unit 1, but we do not write this as 14 as it is greater than 9 and has two characters. We need to use one of the characters we looked at earlier. We said that 14 is represented by the character E, so we use this:

16	1
3	E

This means that when we convert the number 62 into hexadecimal we get the hexadecimal value 3E.

We also need to be able to convert hexadecimal into denary. To do this we need to multiply the unit by the number of each unit needed. Let us convert the hexadecimal value 4B:

16		1
4 × 16 = 64		11 × 1 = 11
64	+	11

Looking at this calculation we can see that 64 + 11 = 75. Therefore we know that the hexadecimal value when converted into denary is 75.

Computers do not actually process hexadecimal, they use binary. Computer scientists use hexadecimal because it is quicker to read than binary. Lots of 1s and 0s are needed to represent a large amount of data; this can be reduced by using hexadecimal.

Hexadecimal is used in a few ways by programmers. One application that you may know of is its use in Hypertext Markup Language (HTML). It is used in HTML to represent colour codes. For example, the colour code for black is #000000 and the colour code for white is #FFFFFF. These are hexadecimal codes that are made up of six digits.

Another application of hexadecimal is in Media Access Control (MAC) addresses. A MAC address is a unique address given to each device on a network. It is used to identify the device. A MAC address is a 12-digit hexadecimal code, for example 001C626454E6.

Try converting that to denary!

MAC addresses use 12-digit hexadecimal codes

How do we carry out a binary shift?

OCR specification reference

☑ binary shifts

A binary shift can be performed on a binary number. It can be performed to the left or to the right on a byte of data.

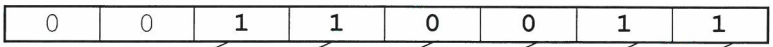
In a binary shift all the numbers in the binary number are shifted to the left or to the right by a number of places. Zeros are added to fill the places in the byte left by the shift. The numbers that have been shifted out of the byte are discarded.

Example of a left shift

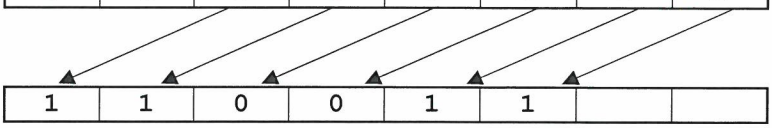
Let's use the binary number 00110011. We will perform a left shift of two places on this binary number.

If we perform a left shift of two places, each digit in the binary number gets shifted two places to the left. The two digits at the left of the number are discarded. Two zeros are placed in the empty spaces left by the shift. We can show this in stages:

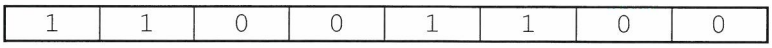
Original binary number:



Each digit shifted two places left. The digits 00 shifted out are discarded:



Zeros replace empty spaces:



A binary shift affects the denary value of a binary number. If we convert the first binary number to denary we get:
00110011 = 51

After we have performed the two-places-left shift, the value of the denary number changes:
11001100 = 204

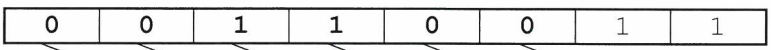
Although you don't need to know this, the notation used for a left shift is '<<'. So we have just calculated 51 << 2 = 204.

Example of a right shift

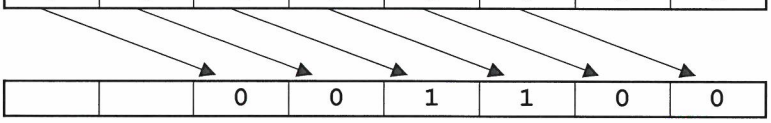
Let's use the binary number 00110011 again. We will perform a right shift of two places on this binary number.

If we perform a right shift of two places, each digit in the binary number gets shifted two places to the right. The two digits at the right of the number are discarded. Two zeros are placed in the empty spaces left by the shift. We can show this in stages:

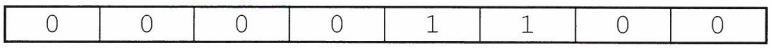
Original binary number:



Each digit shifted two places right. The digits 11 shifted out are discarded:



Zeros replace empty spaces:



If we convert the first binary number to denary we get:
00110011 = 51

After we have performed the two-places-right shift, the value of the denary number changes:
00001100 = 12

Right shifts also have their own notation '>>'; in this case 51 >> 2 = 12.

Quick task

Perform a three-place binary shift to the left on the binary number 10110111.

What effect has this had on the denary value?

How do we add two binary numbers together?

OCR specification reference

☑ how to add two 8-bit binary integers and explain overflow errors which occur

Adding together two binary numbers is not the same as adding together two denary numbers. There are different rules that we need to follow.

There are four rules that we need to remember when adding binary numbers; these are:

0 + 0 = 0	This is a familiar calculation that you already know
1 + 0 = 1	This is a familiar calculation that you already know
1 + 1 = 10	10 is the binary number for the denary value 2
1 + 1 + 1 = 11	11 is the binary number for the denary value 3

Let's try adding together two binary numbers using these rules:

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

We start adding at the right-hand side. We know from our rules that 0 + 0 = 0, so we write 0 in the answer section:

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

We move one place left to the next sum. We know from our rules that 0 + 1 = 1, so we write 1 in the answer section:

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

We move to the next sum. We know from our rules that 1 + 1 = 10. We can only write one digit in the answer section; the other digit we need to carry over to the next sum. We write the right-hand number in the answer section, which is the 0. We carry the left-hand number over to the next sum, which is the 1:

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

We move to the next sum. We know from our rules that 1 + 1 + 1 = 11. We write 1 in the answer section and carry 1 to the next sum:

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


We move to the next sum. We have a different situation now. As we have a carry that creates a sum of 1 + 1, we do not use the 0 in the calculation. We only use all three numbers when we have a carry, if all three numbers are 1. If a number is a 0 in the calculation it is discarded. If both numbers are 0 in the calculation we use 0 and the carry of 1. We know from our rules that 1 + 1 = 10. We write 0 in the answer section and carry 1 to the next sum:

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

We move to the next sum. We have two 0s in the calculation so we use the carry and one 0. We know from our rules that 1 + 0 = 1. We write 1 in the answer section:

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

We move to the next sum. We know from our rules that 0 + 0 = 0. We write 0 in the answer section:

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

We move to the next sum. We know from our rules that 1 + 0 = 1. We write 1 in the answer section:

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

So we can see that the answer to our binary addition question is 10101010. We can check whether our addition is correct by converting all the binary numbers to their denary value:

```
  1 0 0 1 1 1 0 0      = 156
+ 0 0 0 0 1 1 1 0      = 14
-----------------
1 0 1 0 1 0 1 0        = 170
```

We can see that our addition is correct as 156 + 14 = 170.

Let's try adding two more binary numbers, 11000110 and 10101111:

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

We can use the rules we know and add each sum together and we will get:

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


We get a problem though when we get to the last sum. We need to add together 1 and 1; this will give us 10. We write 0 in the answer section, but we still have 1 to carry. We cannot carry the one to another sum though as we do not have any left. This 1 becomes what is known as an **overflow error**:

Overflow error – an error that occurs when the total from adding binary numbers cannot be stored within a byte of data

```
Overflow error ──► 1      1 1 0 0 0 1 1 0
                   +      1 0 1 0 1 1 1 1
                   ------------------------
                          0 1 1 1 0 1 0 1
```

An overflow error occurs because our number after addition becomes too big to be stored in one byte. We stated earlier that the largest number that could be stored in an 8-bit byte is 255. If we convert our two binary numbers to their denary value and add them together, we will see that they equal a number greater than 255:

```
      1 1 0 0 0 1 1 0      =    198
+     1 0 1 0 1 1 1 1      =    175
-------------------------
      0 1 1 1 0 1 0 1
```

We can see that 198 + 175 = 373. This is larger than 255 so it creates the overflow error.

What is a check digit?

OCR specification reference

☑ check digits

When data is stored or transmitted from one device to another, we often check to see whether any errors have occurred in entering, storing or transmitting the data. One way of detecting errors is by using a **check digit**.

Check digit – an error detection method that can be used to check the data entry of numbers

A check digit is used to detect errors in numerical data such as barcodes, bank account numbers and ISBN numbers. It is used to check for human error when entering the data. A calculation is performed on the data entered, and a check is made to see whether the result is equal to the result stored of the same calculation performed on the correct data by the computer. If the two results match, there are almost certainly no errors entered in the data. If the two results do not match, there must be an error in the data entered.

An example of a check digit calculation is the Modulo-11 system. This system is often used for ISBN numbers.

Let's use the ISBN number 020103801-3. This is the ISBN number for a book called *The Art of Computer Programming*. The last digit in the ISBN number is the check digit; in this case the check digit is 3. There is a calculation performed on the ISBN number to create the check digit. We can carry out this calculation in stages. This calculation is called the Modulo-11 system.

First of all we write down all the numbers in the ISBN apart from the check digit:

0	2	0	1	0	3	8	0	1

Then we place a 10 under the first number, a 9 under the second, an 8 under the third, etc.:

0	2	0	1	0	3	8	0	1
10	9	8	7	6	5	4	3	2

We then multiply the top numbers by the bottom numbers:

0	2	0	1	0	3	8	0	1
10	9	8	7	6	5	4	3	2
0	18	0	7	0	15	32	0	2

We now add together the results of the multiplication:

$$0 + 18 + 0 + 7 + 0 + 15 + 32 + 0 + 2 = 74$$

We divide this total by 11 and take note of the remainder:

$$74 \div 11 = 6 \text{ remainder } 8$$

We now subtract the remainder from 11:

$$11 - 8 = 3$$

The result of this calculation becomes the check digit. The result of our calculation is 3. We can see the original check digit is 3; this means that our check digits match and our data was entered correctly. If we had made an error when entering our ISBN our check digits would not match; for example:

If we entered 020113801 rather than 020103801, when the Modulo-11 calculation is performed the result would be:

0	2	0	1	1	3	8	0	1
10	9	8	7	6	5	4	3	2
0	18	0	7	6	15	32	0	2

0 + 18 + 0 + 7 + 6 + 15 + 32 + 0 + 2 = 80
80 ÷ 11 = 7 remainder 3
11 - 3 = 8

Therefore the check digit would be 8 and would not match the original check digit of 3.

How is binary used to represent characters?

OCR specification reference

☑ the use of binary codes to represent characters

Computers can only process binary; this means that all data we input into a computer needs to be converted to binary. We have looked at converting number into binary; we also need to understand how text, images and sound are converted into binary.

If we had to write in binary to create all the documents we needed to create, such as our homework or messages to our friends, it would take us a great deal of time. Typing out the 1s and 0s that are needed for each character would take much longer than typing any character the binary numbers represent. This is why binary codes were created to represent each character, so that we can just press the character on our keyboard and it is converted into binary, so the computer understand which character we have pressed. The character will then appear in our document or message.

Each character is assigned a different binary code. Capital letters and lower-case letters will also have different binary codes, as will punctuation and numbers. All other keys on a keyboard will also have a binary code assigned to them, such as ALT and SHIFT.

What is a character set?

OCR specification reference

- ☑ the term 'character set'
- ☑ the relationship between the number of bits per character in a character set and the number of characters which can be represented (for example ASCII, Extended ASCII and Unicode)

The binary code that is used to represent each character is called a **character set**. There are two main character sets that are used: ASCII and Unicode.

Character set – a list of characters used to create documents and their binary codes

ASCII stands for American Standard Code for Information Interchange. It was designed back in the 1960s and based upon the English language. In ASCII, each character is 1 byte of data. The character code itself is made up of 7 bits and there is a 0 present at the start of each to create the byte of data. There are 128 characters in ASCII; these are the numbers 0 to 9, the lower-case letters a to z, the capital letters A to Z, punctuation symbols, the space bar and other keys present on a standard keyboard.

You can easily search for an ASCII conversion chart on the Internet. It will show you the binary codes for each character; for example: a = 01100001 A = 01000001

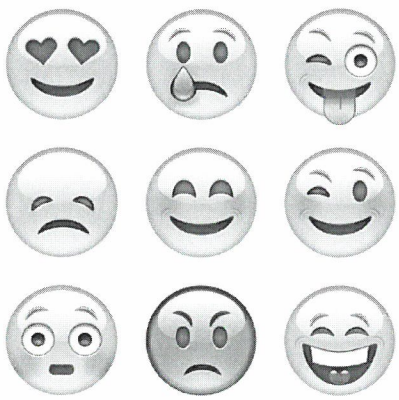
Quick task

Search for an ASCII table on the Internet and decode this:

01000011 01101111 01001101 01110000 01110101 01110100 01100101 01110010
01010011 01100011 01101001 01100101 01101110 01100011 01100101

As each character is equal to 1 byte of data, it would actually be possible to have 256 characters, but as each character is a 7-bit code with a 0 at the front, it uses only 128. This works well for the English language, but other languages, such as French and German, need more characters than the 128 that are available. These languages can make use of the character set Extended ASCII. This character set uses the 128 original characters from ASCII as well as making available additional characters used in other languages such as those with accents and umlauts. This is done by making use of the whole 8 bits available in the byte, rather than the 7 bits used in standard ASCII.

There are some languages that use a completely different alphabet to the English language, such as Russian. It would not be possible to use ASCII or Extended ASCII to create the additional characters needed for this kind of alphabet. Therefore another character set was formed called Unicode. Unicode was developed in the late 1980s when it was recognised the ability to assign a lot more characters to have a universal character set was needed. The 1-byte system used in ASCII can only represent up to 256 characters. Unicode uses a 2-byte system for each character, which means that more than 65,000 characters can be represented. This is more than enough for every language to be represented in one character set. Due to this, there is space for so called 'emoji' characters – icons that convey various emotions, and some icons that are just plain silly or entertaining!



How is an image represented as binary?

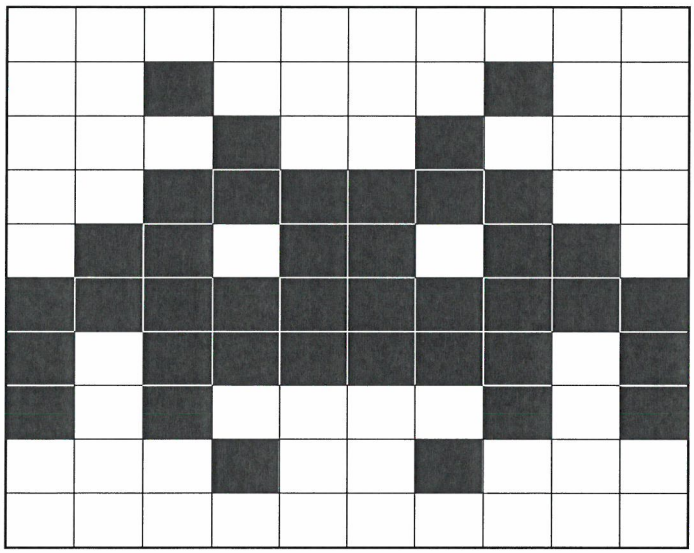
OCR specification reference

- ☑ how an image is represented as a series of pixels represented in binary
- ☑ metadata included in the file

We regularly take photographs with our digital devices, but how are these images processed and stored by our computers? We see the images as analogue, but as we have stated previously a computer needs them to be converted to binary to be able to process and store them.

Images are made up of **pixels**; these are tiny squares that appear on a screen. Each of these pixels has a binary representation. If an image is simply black and white in colour, then each pixel would be a 1 or a 0; for example:

Pixels – a small dot in an image; many of them together create an image



In this image 1 represents black.
The binary code for this image would be:

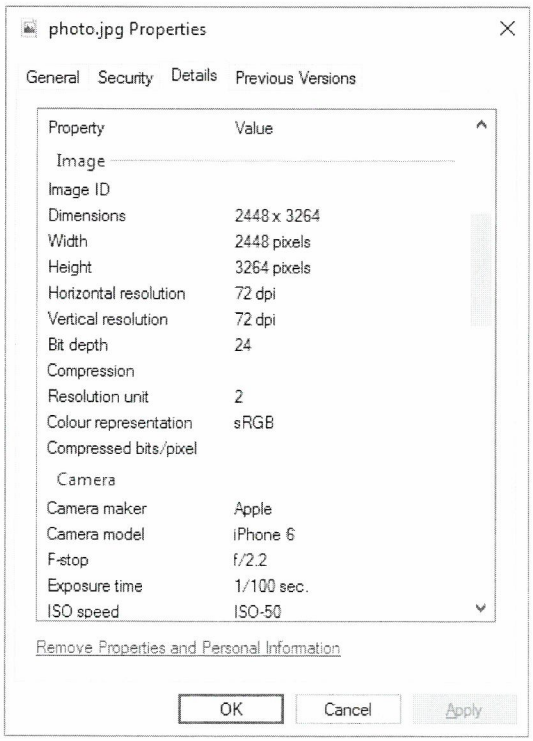
0000000000
0010000100
0001001000
0011111100
0110110110
1111111111
1011111101
1010000101
0001001000
0000000000

The dimensions (the height and the width) of the image are 10 pixels by 10 pixels. If we did not define the dimensions then we would not know how to create the image properly.

Metadata – data that is stored with a file that provides information about how the file needs to be structured

For example, if we set the image to 12 pixels across rather than 10 and used the binary code, we would not end up with the same image. The data that sets the dimensions of the image for the computer is called **metadata.**

Metadata is included with each image to set the structure of the image. As well as the dimensions of the image, it will also store details about the colour depth.



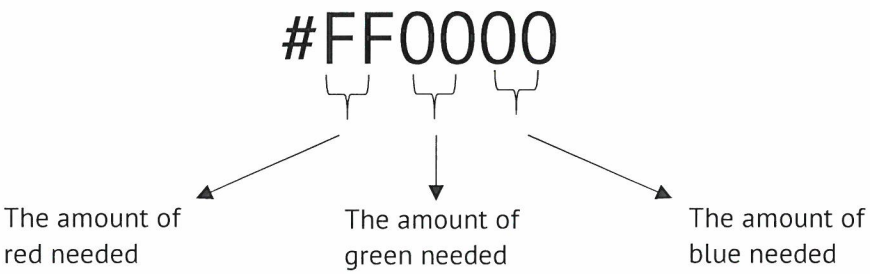
Some of the metadata stored in a photo taken by an iPhone

How can colour depth and resolution affect the size of an image?

OCR specification reference

☑ the effect of colour depth and resolution on the size of an image file

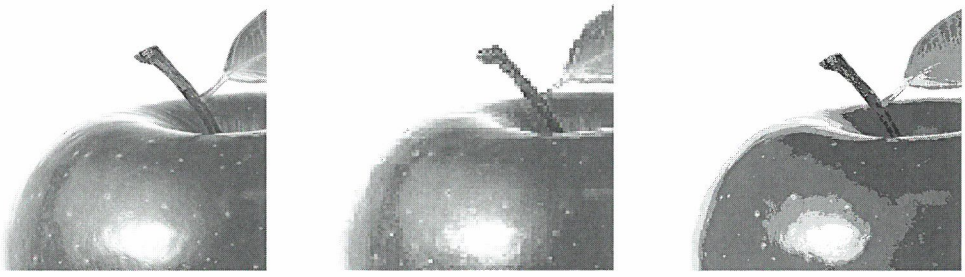
If we want our image to be more than just two colours, we need further binary data to be included. This binary data will instruct the computer on how to create the colour needed for each pixel. Most colours are made up of red, green and blue; this is called the RGB colour system. The amount of red, green and blue needed to make up the colour will be reflected in the binary data for each pixel. These colour codes are shortened to their hexadecimal form so they are easier for us to use. An example of a colour code is #FF0000. The code is broken up into the following data:



This is the colour code for red; as you can see, it needs the full amount of red but no green or blue. If we converted this hexadecimal number to binary it would be 111111110000000000000000; each colour needed, red, green and blue, is represented by a byte of data. All this data is needed to create one coloured pixel in an image. Thousands of pixels are normally needed to create an image, therefore if each needs at least 24 bits (3 bytes) of data, that's a lot of data to create an image!

The size of an image file can be affected by the amount of data that needs to be stored. The amount of data that needs to be stored is affected by the quality and size of the image. If an image has a 2-bit colour depth, it would allow for four different values to be used, 00, 01, 10 and 11, giving four possible colours. The greater the number of bits that are used per pixel, the more colours can be made available. An 8-bit colour depth would allow 256 colours to be used. The system we spoke of above uses a 24-bit colour depth, giving over 16 million possible colours per pixel. Therefore the greater the colour depth, the more data needs to be stored to create the image.

The resolution of an image can also affect the size of an image file. The resolution of an image is measured in dots per inch (dpi). This is the number of pixels that are used in each inch on a screen or a hardcopy. A good-quality image will normally have a resolution of 300 dpi, whereas a low-quality image will normally have a resolution of 72 dpi. This means that a low-quality image needs 72 × 72 dpi in an inch square, needing 5,184 pixels to create each square inch. For an image to be of a good quality, 228 more pixels per inch need to be stored; 300 × 300 gives 90,000 pixels needed for each square inch. This will increase the size of the image by a very large amount.



Normal | Low-resolution | 16 colours

How is sound represented as binary?

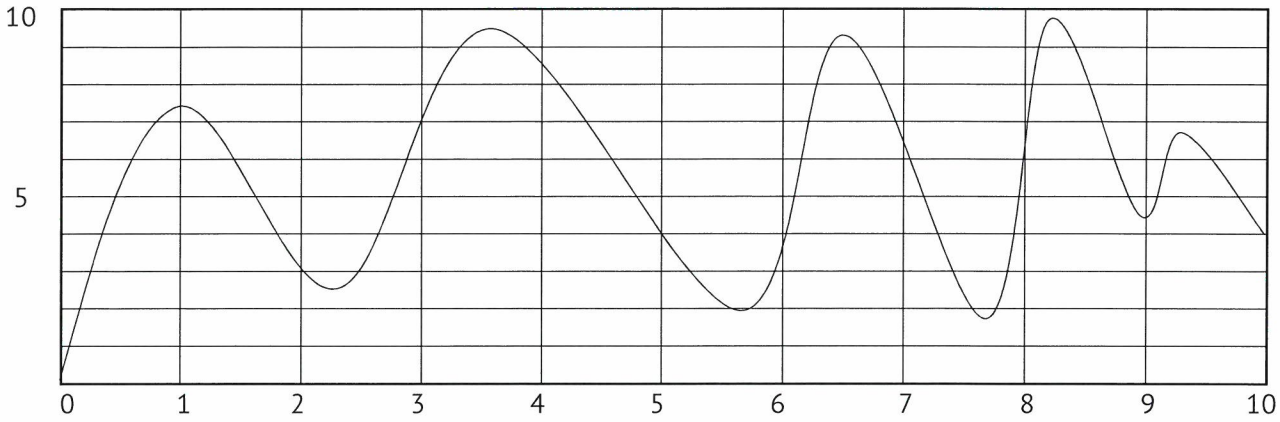
OCR specification reference

☑ how sound can be sampled and stored in digital form

We also need to convert sound into binary to be able to process and store it using a computer. When we record sound we do this at set time intervals; this is called **sampling**. The sample size is measured in hertz; 1 hertz means 1 sample taken per second. Most music CDs are sampled at 44,100 hertz.

Sampling – a sound measurement taken at a given point in a sound file

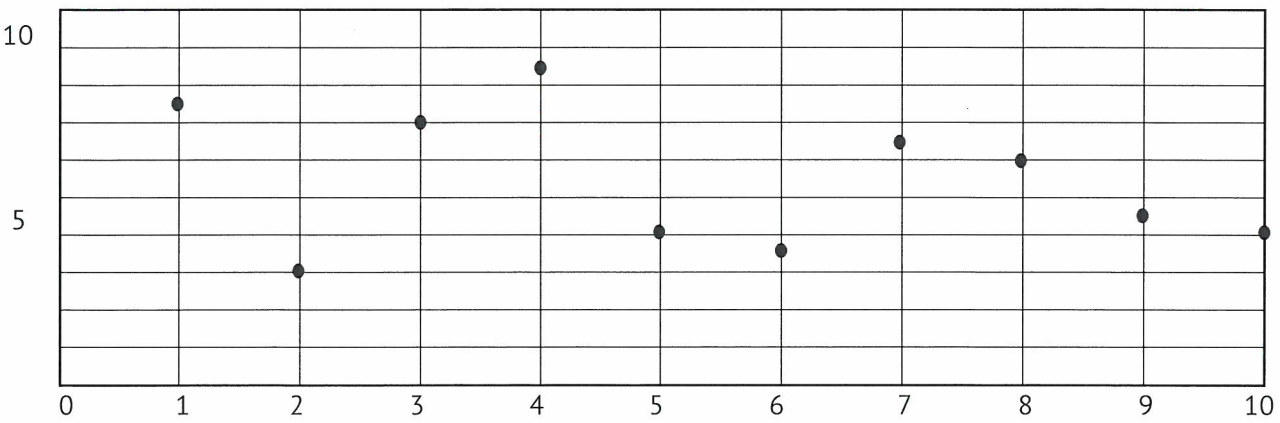
We can look at how a simple sound wave is sampled:



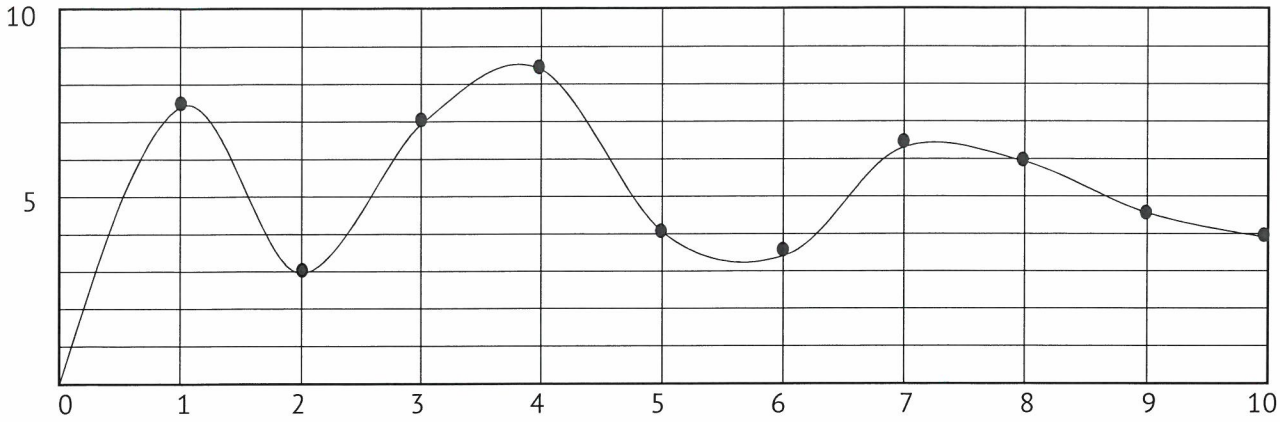
We can record the sound value at each sample taken:

Sound value	7.5	3	7	8.5	4	3.5	6.5	6	4.5	4
Time sample	1	2	3	4	5	6	7	8	9	10

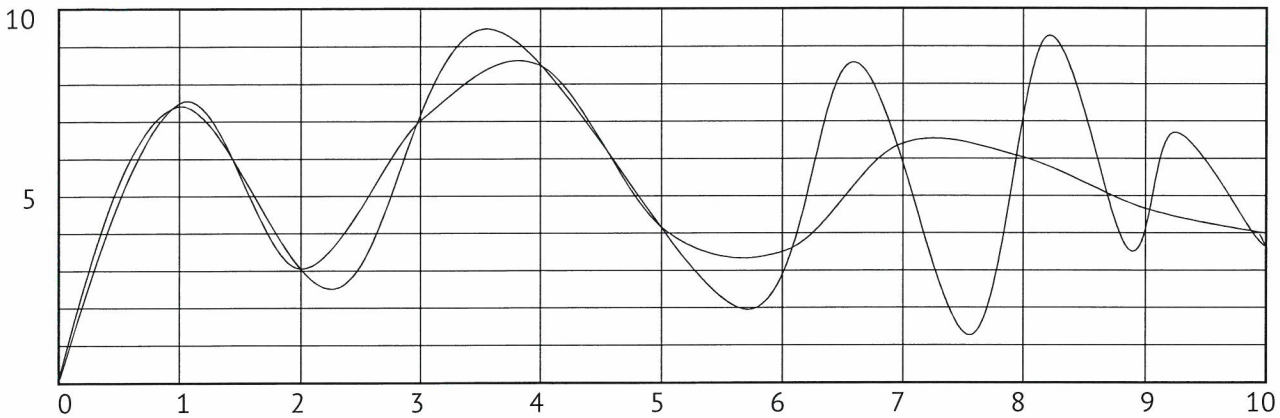
These values are converted to binary and stored in the sound file. When the sound file is played back, the binary values are used to recreate the analogue sound. We can use the samples we took to recreate the sound wave:



We can then create the sound wave using these sample points:



If we overlay the original sound wave with the one played back after recording, we can see there is a large difference between the two:



This difference occurs because the computer can only recreate and playback the sound wave from the samples. It doesn't know what has happened between each sample as it hasn't recorded this. If you look at the end of the sound wave in particular, it fluctuates up and down quite a bit in between each time sample. The computer hasn't recorded this so it just interprets it as a steady decline in the sound wave.

How can sampling frequency and bit rate affect the output of sound?

OCR specification reference

☑ how sampling intervals and other factors affect the size of a sound file and the quality of its playback:
- sample size
- bit rate
- sampling frequency

Sample frequency – the frequency at which a sample in a sound file is taken

As we have just seen, the frequency at which the sound is sampled in a sound recording can affect how the sound is output. This is called the **sample frequency**. If a small sample frequency is used, like the one above, then fluctuations in the original sound may not be recorded at all. In order to increase the accuracy of the sound that is output we can increase the sample frequency. If we look at the sound wave above and double the number of time samples taken, we will get a more accurate sound wave when output. Try it and see!

The closer together the time samples, the more accurate the sound that will be output. This is why CDs are recorded with a sample size of 44,100 hertz. It is important that the output is as close as possible to the original sound recorded. Telephone conversations are sampled at a lower rate of 8,000 hertz.

Discussion point: Why are telephone conversations sampled at a lower rate than music CDs?

The higher the sample frequency for a sound file, the larger the file will be. This is simply because it has more samples to store.

The sample size can also affect the size of a sound file. The sample size refers to how many values can be measured on it. A 2-bit sample size would give four values to measure; an 8-bit sample size would give 256 values to measure. Most CDs use a 16-bit sample size. The more values that can be measured, the more data that can be recorded and the larger the file size.

Another aspect that can affect the size of a sound file is the **bit rate**. The bit rate is simply how much data per second is required to output the sound file. It is measured in megabits per second (Mb/s). A 3-minute song at a sample frequency of 44,100 hertz, a sample size of 16 bits and a bit rate of 1.35 Mb/s would create a file of size 30.3MB.

Bit rate – the amount of data per second that is required to output a sound file

How can we compress a file and why might we need to do this?

OCR specification reference

- ☑ bit, nibble, byte, kilobyte, megabyte, gigabyte, terabyte, petabyte
- ☑ need for compression
- ☑ types of compression: lossy and lossless

The size of a digital file can be measured in a number of different units. The measurements for files are:

Unit of measurement	Abbreviation	Conversion
nibble	-	4 bits
byte	B	8 bits
kilobyte	kB	1024 bytes
megabyte	MB	1024 kB
gigabyte	GB	1024 MB
terabyte	TB	1024 GB
petabyte	PB	1024 TB

Discussion point: What is the biggest unit of storage that exists? Why does storage need to be this big?

Sometimes a file may be too big for us to store, or too big for us to download or transmit to someone else. We may want to reduce the size of a file or a number of files to reduce the amount of storage space used on our hard drive, or to make downloading or transmitting a file quicker. We can do this by using file compression.

When we compress data, we manipulate the structure of the data to make the size of the file smaller. A compression algorithm is used to do this and there are two main types of algorithm used: **lossy** and **lossless**.

Lossy – a file compression method that permanently removes redundant data

Lossless – a file compression method that does not remove any data, but compresses data by looking for repeating patterns

In lossy compression, data that is redundant is removed from the file to reduce the size. In a sound file this redundant data might be sounds that the human ear is unable to hear, therefore the quality of the output will not be affected for a human if these sounds are removed. The data is permanently removed from the file when the file is compressed using lossy compression. This means that the sound will not be exactly as it was when it was recorded, but it will be a close representation and the human ear will not really notice the difference. The file format MP3 is a common lossy compression format for sound and a JPEG is a common lossy compression format for images.

In lossless compression no data is removed and discarded in the compression process. Lossless compression looks for patterns in the data. When it finds repeating patterns in the data it takes the pattern and stores how many times the pattern occurs. When the data is restored to its uncompressed form, the file uses the number of times each pattern is stored in the reconstruction process. We can look at this in a simple form with text.

Consider the text: **I love the subject Computer Science. Computer Science is my favourite subject of all.**

If we focus on the characters of this message, without full stops and spaces it has 70 characters. We would need at least 70 bytes of data to store this message. We can see that the message has some repeating words in it, so we could use a lossless compression method to reduce the size of the file by recognising these repeating words. We would also need to store the position of the word in the message to make sure it could be reconstructed. Therefore, using a lossless compression method we could create the following file:

Word	Position
I	1
love	2
the	3
subject	4, 12
Computer	5, 7
Science	6, 8
is	9
my	10
favourite	11
of	12
all	13
Total bytes	
48	14

We can see that by simply recording these repeating words and their positions we have reduced our file size from 70 bytes to 62 bytes. This may seem like a small reduction, but this is only a small message. Imagine this on a larger scale where many more repeating patterns could be identified. A common lossless file format is .ZIP.

Chapter Summary

- We process analogue data but computers process digital (binary) data. Therefore all data needs to be converted to binary to be processed by a computer.
- There are different number systems that we need to know to convert data from denary to binary and to hexadecimal. Denary is a base-10 system, binary is a base-2 system and hexadecimal is a base-16 system.
- A binary shift can be performed on a binary number. The shift can either be left or right. A binary shift will affect the denary value of the number.
- There are four rules we need to remember when adding binary numbers.
- If the addition of our binary number is greater than 255 this will create an overflow error.
- A check digit is an error detection method that can be used to check the data entry of numbers is correct. It is a calculation that is carried out on the data and a check digit is added to the data from the calculation. If the result of the calculation when the data has been entered is the same as the check digit, the data entered is correct.
- The text-based characters that we use every day need to be represented as binary to be processed by a computer. The computer knows what the character selected is by the use of a character set.
- Images are made up of pixels. The amount of data needed for each pixel depends on whether it is a colour image or black and white. If the colour depth and resolution of an image are increased, this increases the size of the image file.
- Sound is sampled at set time intervals when it is processed by a computer. This is called the sample frequency. If the sample frequency is increased, this will increase the size of the sound file. A sound file will also be increased if the sample size and the bit rate of the sound file are increased.
- Files can be compressed. A compression algorithm can be used to do this.
- There are two main types of compression algorithm: lossy and lossless. Lossy permanently removes redundant data from a file; lossless looks for repeating patterns to use to reduce the file size and does not remove any data.

Practice Questions

1. Convert the binary number 10110001 into denary. Show your working. [2]
2. Convert the binary number 10110001 into hexadecimal. Show your working. [2]
3. Explain what is meant by an overflow error. [1]
4. Explain two ways in which the quality of the playback of a sound file can be improved. [4]
5. Abdul has a video file that he wants to send to Cindy. He wants to use the Internet to share the file with Cindy for their homework project. Explain which data compression method he should use to send the file and why. [3]

