# 28 Hardware and software

**SPECIFICATION COVERAGE**

3.6.1 Hardware and software

**LEARNING OBJECTIVES**

In this chapter you will learn:

- the definition of hardware and software
- that hardware is classified into internal and external components
- that there are many different types of software
- many programs are created to make a computer system work, such as operating systems, library and utility programs
- specific software exists to convert programming code into machine code
- how the operating system manages resources.

**KEYWORDS**

**Hardware**: a generic term for the physical parts of the computer, both internal and external.

**Software**: a generic term for any program that can be run on a computer.

**INTRODUCTION**

The term 'system' is often used to refer to the various physical components of your computer – the monitor, keyboard, etc. A computer system has two main elements: **hardware** and **software**. It is only when the two are combined that you create a fully working system. There is no point in having one without the other. Some definitions of a computer system add in a third vital element – the user. In this chapter we will look at the main elements of hardware and software that make up a computer system.

## Hardware

Computer hardware is the physical components of the computer. Sometimes this is described as 'the parts you can touch'. This is not particularly helpful as many elements of hardware are contained inside your computer and can only be seen (or touched) by taking off the case. Therefore, it is important to distinguish between the internal components, which are the processing and storage devices, and external components, normally referred to as peripherals.

- **External components (peripherals):** The external components of hardware are the parts that you can touch, for example the monitor, mouse, keyboard, and printer. The external components are used either to get data into or out of the system. Consequently, they are referred to as input and output (I/O) devices. Some storage devices are also external, for example DVD and flash drives may be added as peripherals. Input devices include the keyboard, mouse, scanner, digital camera and microphone. Output devices include the monitor, printer and speakers.

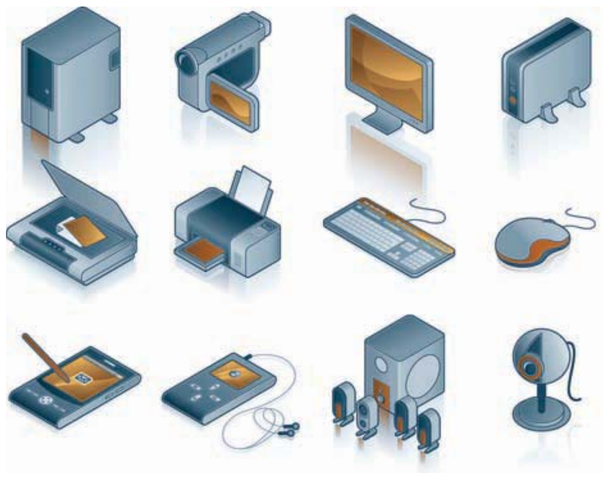


**Figure 28.1** External hardware devices

- **Internal components (processing and storage):** the internal hardware components are housed within the casing of the computer and include the processor, the hard disk, memory chips, sound cards, graphics cards and the circuitry required to connect all of these devices to each other and to the I/O devices.

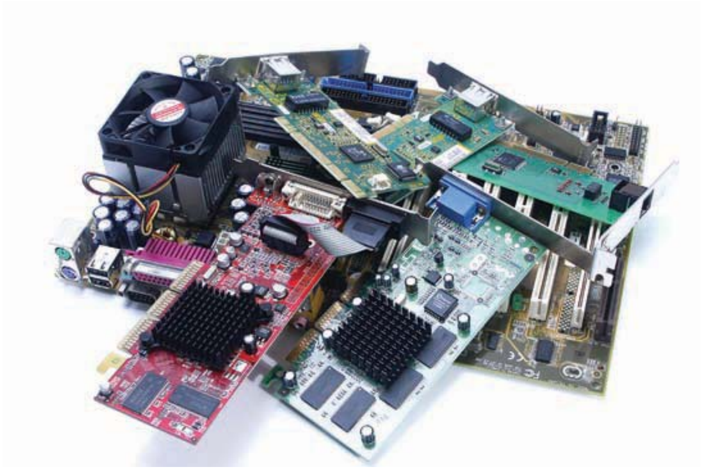


**Figure 28.2** Internal hardware components

## Software

Hardware is useless on its own unless we have some programs to run on it. Software is the general term used to describe all of the programs that we run on our computers. These programs contain instructions that the processor will carry out in order to complete various tasks.

This covers an enormous range of possibilities from standard applications, such as word processors, spreadsheets and databases, to more specific applications, such as web-authoring software and games. It also includes programs that the computer needs in order to manage all of its resources, such as file management and virus-checking software. As users, we tend to be aware of the software that we use on a regular basis, yet this is only one part of the software that is on our computers.

The range of software is so great now that some classification is needed in order to make sense of it all. A first level of distinction is made between application software and system software.

**KEYWORD**

**Application software**: programs that perform specific tasks that would need doing even if computers didn't exist, e.g. editing text, carrying out calculations.

### Application software

**Application software** refers to all of the programs that the user uses in order to complete a particular task. In effect, it is what users use their computers for. People do not buy computers for the sake of it, they buy them because they have a need to do something: write essays, email, manage a business, create web pages, etc. To carry out any of these, application software is needed that has the necessary features.

There is a wide range of application software available and, in most cases, a number of different applications to choose from that complete the same task. For example, you need application software to access web pages on the Internet – it is called a browser. There are three main ones to choose from: Internet Explorer, Google Chrome and Mozilla Firefox; and many more less well known ones. All of them do the same thing although there are subtle differences between them.

### System software

Whereas application software is what we use our computers for, system software covers a range of programs that are concerned with the more technical aspects of setting up and running the computer. Many aspects of system software are invisible to the user which means that they will not even realise that they have system software on their computer. System software exists to support the applications software.

There are four main types: utility programs; library programs; compilers, assemblers and interpreters; and operating system software.

**KEYWORD**

**Utility programs**: programs that perform specific common task related to running the computer, e.g. zipping files.

#### Utility programs

This covers software that is written to carry out certain housekeeping tasks on your computer. They are normally supplied with the operating system though they can be purchased separately. **Utility programs** are often made available as free downloads. Utility programs are designed to enhance the use of your computer and programs though your computer will still work without them.

A common example of a utility program is compression software which allows you to compress files, making them much smaller. Other examples include anti-virus software, back-up software and registry cleaners.

**KEYWORD**

**Library programs**: code, data and resources that can be called by other programs.

## Library programs

**Library programs** are similar to utility programs in that they are written to carry out common tasks. The word library indicates that there will be a number of software tools available to the users of the system.

Whereas some utility programs are non-essential, library programs tend to be critical for the applications for which they were built. For example, the Windows operating system uses Dynamically Linked Library (DLL) files, which contain code, data and resources. These are similar to executable files and are loaded dynamically by Windows as they are required. There are hundreds of DLL files that carry out a wide range of actions including controlling dialog boxes, managing memory, displaying text and graphics and configuring device drivers.

The Python programming languages also has an extensive library that contains built-in modules that provide various standard system functions and solutions. For example, the library contains code modules for handling common data types, displaying fonts and graphics and performing mathematical and functional operations.

**KEYWORDS**

**Translators**: software that converts programming language instructions into 0s and 1s (machine code). There are three types – compilers, assemblers and interpreters.

**Compiler**: a program that translates a high-level language into machine code by translating all of the code.

**Assembler**: a program that translates a program written in assembly language into machine code.

**Interpreter**: a program for translating a high-level language by reading each statement in the source code and immediately performing the action.

**Operating system software**: a suite of programs designed to control the operations of the computer.

**Virtual machine**: the concept that all of the complexities of using a computer are hidden from the user and other software by the operating system.

## Translators: compilers, assemblers and interpreters

**Translators** are software used by programmers to convert programs from one language to another. There are three types: **compilers**, **assemblers** and **interpreters**. At some point, every piece of software, whether it is application software or system software, has to be written by a programmer. A program is simply a series of instructions written by a programmer that the computer's processor must carry out.

In order to write software, programmers use programming languages which allow them to write code in a way that is user-friendly for the programmer. However, the processor will not understand the programmers' code, so it has to be translated into machine code, that is, 0s and 1s. Compilers, assemblers and interpreters are used to carry out this translation process. There is more detail on how these work in Chapter 29.

## Operating system software

An operating system is a collection of software designed to act as an interface between the user and the computer and manages the overall operation of the computer. It links together the hardware, the applications and the user, but hides the true complexity of the computer from the user and other software – a so-called **virtual machine**.

When you are using a computer you are obviously aware of the applications software you are using, whether it is an Internet browser, a spreadsheet or a game of some sort, but you are much less aware of the software that is running in the background. The systems software is dominated by the operating system (OS).

The OS in a modern computer is very large. For example, Microsoft Windows 8 needs a minimum of 1 GB of RAM and 16 GB of hard disk space when the computer is in use. This gives you some idea of the complexity of the OS. This is because the OS carries out many tasks. For example it:

- controls the start-up configuration of your computer, including what icons to put on your desktop and what backdrop to use
- recognises when you have pressed a mouse button and then decides what, if any, action to take
- sends signals to the hard disk controller, telling it what program to transfer to memory
- decides which sections of memory to allocate to the program you are intending to use and manages memory to ensure all of the programs you want to run are allocated the space they need
- attempts to cope with errors as and when they occur. For example, if a printer sends an 'out of paper error' or you fail to save a file correctly, it is the operating system that displays the appropriate message
- makes sure that your computer shuts down properly when you have finished
- controls print queues
- manages the users on a network – it maintains the lists of usernames and passwords and controls which files and resources users have access to.

## Resource management

Computers are capable of running many programs, seemingly at the same time. It is the job of the operating system to make sure that each program is allocated enough memory to operate efficiently.

In a computer with only one **processor**, only one program can actually be live at any one moment in time. In order to allow more than one program to appear to run simultaneously, the operating system has to allocate access to the processor and other resources such as peripherals and memory. One of the main tasks that an operating system has to do is to make sure that all these allocations make the best possible use of the available resources.

Usually the most heavily used resource in a computer is the processor. The process of allocating access to the processor and other resources is called **scheduling**. The simplest way that an operating system can schedule access to the processor is to allocate each task a time slice. This means that each task is given an equal amount of processor time. This process of passing access to the processor from one task to the next is also known as 'round-robin' scheduling.

Time slicing is a crude system because a particular task might not need all or even any of its allotted time slice with the processor. For example, a word processor might be waiting for the slowest part of any computer system, the user, to press a key. Waiting for this to happen is a waste of processor time and so a more sophisticated scheduler might pass the time slice on to the next task before the word processor's time slice has expired.

**KEYWORDS**

**Resource management**: how an operating system manages hardware and software to optimise the performance of the computer.

**Processor**: a device that carries out computation on data by following instructions, in order to produce an output.

**Scheduling**: a technique to ensure that different users or different programs are able to work on the same computer system at the same time.

### Managing input/output devices

The operating system also controls the way in which the various input and output devices are allocated, controlled and used by the programs that are using them. Common examples would include:

- allocating print jobs to printers
- rendering the windows, frames and dialog boxes to the screen
- controlling the read/write access to the hard disk.

Accessing some devices is a relatively slow process compared to the speed at which the processor can handle requests. At the same time there are likely to be competing requests where several processes are waiting for the same device. For example, reading and writing to a standard hard disk is relatively slow. Rather than wait for each process to end before it can continue, the OS can effectively create a queue of commands that are waiting for the device and then handle each request in sequence or based on priority.

Every input/output device has a device driver, which is a piece of software that enables the device to communicate with the OS. Device drivers are often built in to the OS or installed when new devices are attached. When the OS starts up it loads the various drivers for all of the input/output devices it detects.

There is more on how input/output devices connect to the computer in Chapter 32.

## Memory management

In the same way that the operating system of a computer controls the way files are stored on a secondary storage system, such as a hard disk, the operating system also controls how the primary memory or RAM is used.

The operating system stores details of all the unallocated locations in a section of memory known as the heap. When an application needs some memory, this is allocated from the heap, and once an application has finished with a memory location or perhaps an application is closed, the now unneeded memory locations are returned to the heap.

**KEYWORD**

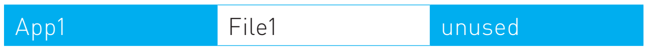
**Memory management**: how the operating system uses RAM to optimise the performance of the computer.

When a user asks for a file to be loaded, it is the job of the **memory management** routines to check to see if enough memory is available and then allocate the appropriate memory and load the file in to those locations.

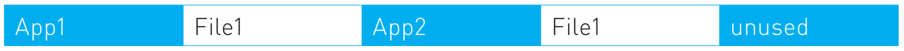
The operating system controls the use of main memory by creating a memory map, which shows which blocks of memory have been allocated to each task. This way an operating system can control more than one task in the RAM at any one time. The amount of memory needed by each task is dependent on the size of the program itself, the variables that will be needed and any files that might be generated by the task, but it is up to the operating system to decide how much space can be allocated.

When the operating system processes a request to load an application or file from the hard disk it is the job of the memory management system to decide whereabouts in the RAM the file will be stored.

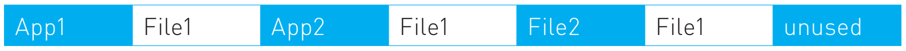
As you add to a file it is highly likely that the work will need more than one block of RAM. In this case, a type of linked list is used to show where each subsequent block is stored.



In this stylised example of a memory map, the application, `App1` was loaded first and work started on `File1`.



After a while the application `App2` is loaded, then work carried on with `File1`. Because `App2` is now in the memory, `File1` is now spread across two sections of RAM.



A little work is done using `App2` which generates `File2`, and then work recommences on `File1` which is now spread across three sections of RAM and so on.

| | | | | | |
|---|---|---|---|---|---|
| App1 | unused | App2 | unused | File2 | unused |

`File1` is now saved and then deleted from RAM, the memory map now looks something like this.

| | | | |
|---|---|---|---|
| App1 | App2 | File2 | unused |

The operating system might now decide to rationalise this and move the applications and existing files so that all the unused space is put together. Even with this simple example you can see that managing the memory can be a complex task for the operating system to control.

## Virtual memory and paging

In some cases, the application or file you are trying to work with will be too big to fit in the available RAM. In this case a process called virtual memory can be used. This involves using secondary storage such as a hard disk to store code or files that would normally be held in RAM. The operating system then treats that part of the hard disk as if it is part of the RAM, hence the name virtual memory.

An alternative method is to hold a kernel or central block of the code in RAM. Other sections of code known as 'pages' are loaded from the secondary storage as and when they are needed. Using this method allows very large applications to run in a small section of RAM. This in turn frees up memory for other applications to use.

A word processor will hold instructions about how to cope with the text itself in the kernel but if the user selects a less commonly used task, such as spell checking or mail merging, the appropriate page of the program will be loaded into RAM from the hard disk. This downloading often causes a noticeable delay in the operation of the application.

KEYWORD

**File management**: how an operating system stores and retrieves files.

## File management

One of the many tasks the operating system has to deal with is managing files – this includes controlling the structures that are used to store the files.

The hard disk on a home computer is likely to contain many thousands of files and without some sort of logical storage structure it would be impossible to find a specific file amongst so many. The operating system on a home computer uses folders or directories. These allow the user to group similar files together so that, for example, all the files for your Computing course might be kept in one folder whilst all the photos from your digital camera would be kept in another. In the case of the photos, it is highly likely that there will be folders within folders. The way folders like these are arranged is known as a hierarchical structure – it looks rather like an inverted tree and indeed the start or base folder is normally referred to as the root folder.

Each file has a filename but because of the folder structure it is possible to use the same filename for different files. In the coursework/photo example there might be a folder called `miscellaneous` and a file called `latest` in each area. Obviously this is not particularly good practice, especially if you want to move files between folders and you lose track of which folders you are working with. It is a good practice to use folder and file names that indicate what they are being used for.

As hard disks get larger and larger, it is becoming increasingly common to split up or partition a hard disk. This means that although you actually only have one hard disk in your computer, the operating system splits it up into a number of partitions or logical drives. This means your computer seems to be fitted with more than one hard drive. You might use this system to store your applications on one logical drive and your data on another.

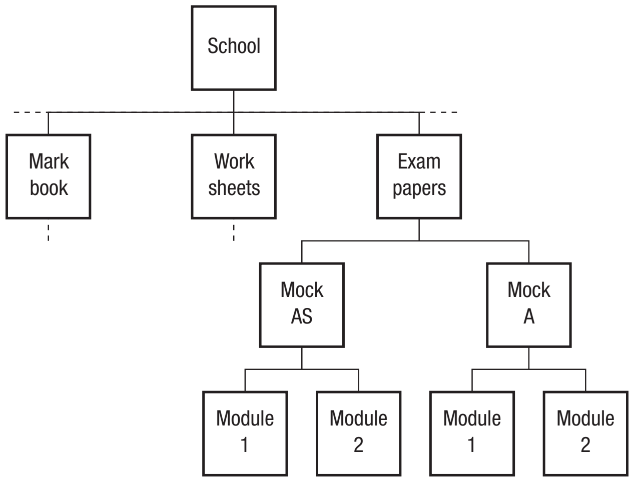


**Figure 28.3** Hierarchical file structure

**Practice questions can be found at the end of the section on page 264.**

## TASKS

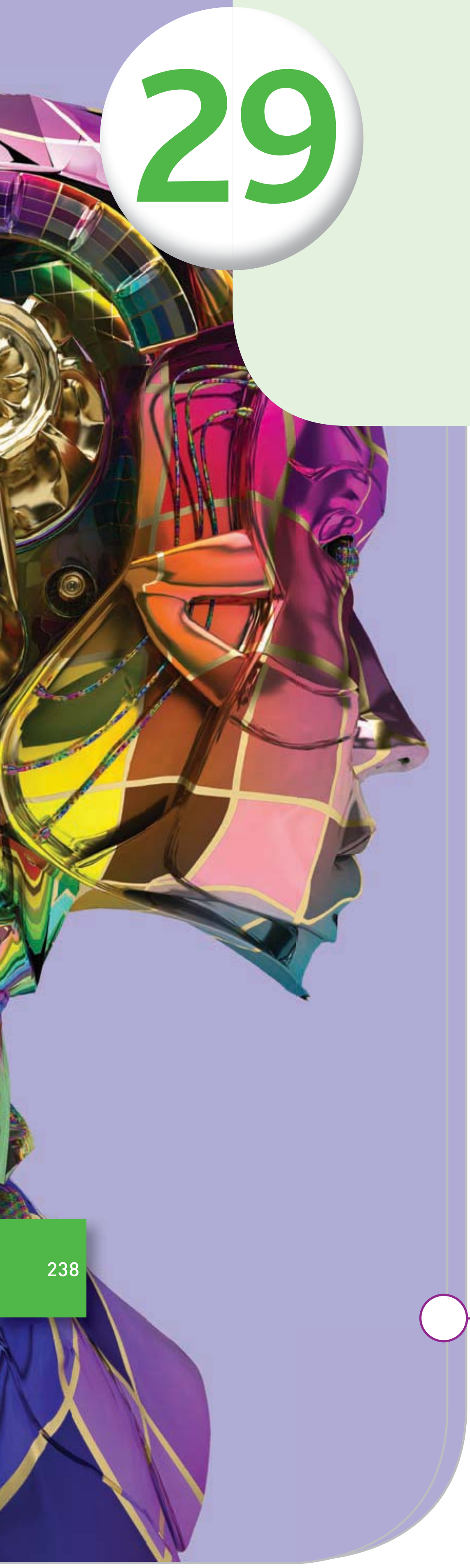
1 Explain the terms hardware and software, differentiating between different types of hardware and software.
2 Identify two utility programs that are not normally part of an operating system.
3 Explain the term virtual machine in the context of an operating system. This should include details of the main tasks performed by the OS.
4 Explain how it is possible for two programs to apparently be running at the same time.
5 Explain how it is possible for two files with the same name to be stored in a file structure.
6 Explain the difference between resource and file management.

## STUDY / RESEARCH TASKS

1 What type of software is a DLL and what is its purpose?
2 What are the most common operating systems in use for mobile devices?
3 Compare two commonly used operating systems. Why would you choose to use one over the other?
4 Open source operating systems such as Linux are not as widely used as Windows or Mac OS. Why not?
5 In a computer system with multiple processors, what implications does this have for how the OS handles resources?

## KEY POINTS

- A computer system is made up of hardware and software.
- Hardware is usually classified in terms of internal and external components.
- System software includes the operating system, library and utility programs.
- Compilers, interpreters and assemblers are programs that convert high-level programming language into executable instructions.
- The operating system plays a critical role in managing resources.



# 29 Classification of programming languages and translation

**SPECIFICATION COVERAGE**

3.6.2 Classification of programming languages

3.6.3 Types of translator

**LEARNING OBJECTIVES**

In this chapter you will learn:

- how programming can be performed in machine code, assembly language and high-level languages
- that there are different methods of programming known as paradigms
- how interpreters and compilers are used to turn programming code into machine code (0s and 1s)
- what bytecode is.

**INTRODUCTION**

Most programming is now done using high-level programming languages. In this chapter we will look at how programming languages have developed over time, in particular looking at the main types of programming languages and the different methods (or paradigms) of programming. Linked with this is the way in which the code that you write as a programmer is converted into 0s and 1s so that it can be understood and processed by the processor.

## Types of programming languages

Although computers as we know them have only been around since the mid-1940s they have changed beyond all recognition in that short time span. Programming the early computers was a very tedious business as programs were entered as a sequence of switch settings. Each switch was either on or off so by combining enough switches the programmer could represent different things.

Thankfully, programming computers has become a much simpler process, though the ever-increasing complexity of the computers themselves brings its own problems.

As computers have developed then so have the programming languages that can be used on them. There are three main types of languages:

- Machine code
- Assembly language
- High-level language

} These two are known as low-level languages

## Machine code

**KEYWORD**

**Machine code**: the lowest level of code made up of 0s and 1s.

The processor of a computer can only work with binary digits (bits). This means that all the instructions given to a computer must, ultimately, be a pattern of 0s and 1s. Programs written in this format are said to be written in **machine code**. Having to write everything down as a sequence of 0s and 1s means that the programs are going to be very long-winded, and because everything is entered as bits there is a high risk of making a mistake. One way to make these bits easier to understand is to show a sequence of bits as either a decimal or hexadecimal number.

To compound the problem further, it is also going to be very difficult to track down any errors or bugs that exist in the code. One final problem is that because machine code is written for a specific processor, it is unlikely to be very portable. This means that it will only work on a computer with the same type of processor as it was written for.

Nevertheless because you are writing instructions that can be used directly by the processor, they will be executed very quickly and the processor will do exactly what you tell it to do.

## Assembly language

**KEYWORDS**

**Assembly language**: a way of programming that involves writing mnemonics.

**Mnemonics**: short codes that are used as instructions when programming, e.g. LDR, ADD.

The need to make programs more programmer-friendly led to the development of **assembly languages**. Rather than using sequences of 0s and 1s, an assembly language allows programmers to write code using words. There is a very strong connection between machine code and assembly language. This is because assembly language is basically machine code with words. The number of words that can be used in an assembly language is generally small. Each of these commands translates directly into one command in machine code. This is called a one-to-one relationship.

Most, but not all, assembly codes use a system of abbreviated words called **mnemonics**. In the small section of assembly code below, the mnemonic `LDR` stands for `Load Register` and `STR` means `Store Register`. `ADD` and `SUB` should be rather more obvious. In this simplified example all the numbers following each mnemonic refer to memory addresses – the place in memory where numbers are written to or read from.

```
LDR   20
ADD   43
STR   20
SUB   41
STR   45
```

This code does the following:

- loads the accumulator with the contents of address 20
- adds the value held in memory location 43 to the value held in the accumulator
- stores the contents of the accumulator to address 20
- subtracts the value held in memory location 41 from the value held in the accumulator
- stores the contents of the accumulator to address 45.

Although assembly language uses words, some of which you might recognise, the code is not particularly easy to understand.

Before any code written in assembly language can be executed it has to be converted into machine code. This is because the processor can only understand binary digits (0s and 1s) – the words are meaningless to the computer, so the words must be converted into these binary digits. This conversion process is carried out by an assembler. The assembler will be able to identify some, but not all, of the errors that are likely to be hidden somewhere in the code. The code that the programmer creates is known as the **source code**. The **assembler** takes this source code and translates it into a machine code version that is known as the **object code**. As with machine code, assembly languages are based on one processor so they are generally not very portable.

**KEYWORDS**

**Source code**: programming code that has not yet been compiled into an executable file.

**Assembler**: a program that translates a program written in assembly language into machine code.

**Object code**: compiled code that can be run as an executable on any computer.

Assembly language is still used in certain situations today as it has a number of advantages over high-level languages:

- Programs are executed quickly as a compiler does not optimise the machine code that it produces as effectively as a programmer who codes in an assembly language that maps directly to machine code.
- Program code is relatively compact for the same reason.
- Assembly language allows direct manipulation of the registers on the processor, giving high levels of control.

Its main uses are where direct manipulation of hardware is required, for example in embedded systems where a low level of interaction is required with hardware. The processors in these systems may be relatively slow and have limited memory in which case the efficiency of an assembly language is needed. This may mean that very few commands are needed to operate the device. Therefore, these instructions can be programmed directly to the processor using an assembly language.

Real-time applications may also use assembly language as they need to respond very quickly to inputs. This is particularly relevant where it is an embedded system.

Assembly language does have other uses where only low-level coding is required. For example, some device drivers are written in assembly language, particularly for customised hardware and software.

**KEYWORDS**

**Low-level language**: machine code and assembly language.

**High-level language**: a programming language that allows programs to be written using English keywords and that is platform independent.

## High-level languages

Machine code and assembly languages are known collectively as **low-level languages**. The increasingly complex demands made on computers meant that writing programs in assembly code was too slow and cumbersome. **High-level languages** were developed to overcome this problem.

Whereas low-level languages are machine-oriented, high-level languages are problem-oriented. This means that the commands and the way the program is structured are based on what the program will have to do rather than the components of the computer it will be used with. This means that a program written in a high-level language will be portable. It can be written on one type of computer and then executed on other types of computer.

There are many different high-level languages available, each being written to cope with the demands of a specific type of problem. For example, some high-level languages are specifically designed for scientific applications, others for manipulating databases, whilst others are used to create web pages. The language the programmer chooses will depend to a large extent on the nature of the problem they are trying to solve.

**KEYWORDS**

**Imperative language**: a language based on giving the computer commands or procedures to follow.

**Object-oriented language**: a programming paradigm that encapsulates instructions and data together into objects.

**Declarative languages**: languages that declare of specify what properties a result should have, e.g. results will be based on functions.

**Functional language**: a programming paradigm that uses mathematical functions.

High-level languages are often classified into three groups based on the programming paradigm we saw in Chapter 5. A paradigm is a concept for the way something works:

- **Imperative languages**: Also known as procedural languages, these work by typing in lists of instructions (known as subroutines or procedures) that the computer has to follow. Every time the program is run, it follows the same set of instructions.
- **Object-oriented languages**: These work by creating objects where the instructions and data required to run the program are contained within a single object. Objects can be further grouped into classes.
- **Declarative languages**: These work by describing what the program should accomplish rather than how it should accomplish it. One type of declarative language is logic programming, which is used widely in the fields of artificial intelligence and works by programming in facts and rules, rather than instructions. The program then uses these facts and rules to interrogate the data and provide results. Another type of declarative language is a **functional language**, which works by treating procedures more like mathematical functions. The building blocks of the program therefore are functions rather than lists of instructions.

The main characteristics of a high-level language are:

- It is easier for a programmer to identify what a command does as the keywords are more like natural language.
- Like assembly languages, high-level languages need to be translated.
- Unlike assembly code, one command in a high-level language might be represented by a whole sequence of machine code instructions. This is called a one-to-many relationship.
- They are portable.
- They make use of a wide variety of program structures to make the process of program writing more straightforward. As a result they are also easier to maintain.

## Translating high-level languages

One of the main features of high-level languages is that they are programmer-friendly. Unfortunately this means the computer will not understand any of the high-level language source code, so it will have to be translated in some way. This process is called translation and in order to carry it out a special piece of systems software called a **translator** is needed.

The assembler is the translator used for low-level languages and there are two types of translator for high-level languages: an interpreter and a compiler.

**KEYWORDS**

**Translator**: the general name for any program that translates code from one language to another, for example translating source code into machine code.

**Interpreter**: a program for translating a high-level language by reading each statement in the source code and immediately performing the action.

### Interpreter

An **interpreter** works by reading a statement of the source code and immediately performing the required action. It may do this by interpreting the syntax of each statement, or by calling predefined routines. In some cases, the source code is translated into an intermediate format before it is executed. Interpreters may examine every statement, which can reduce efficiency. For example, in an iterative statement, the interpreter will read the same statement once for each iteration.

Typically an interpreter only converts what needs to be translated.
For example when the line of code:

```
If Age<17 Then Output = "Cannot drive a car"
```

needs to be translated, only the first section of the code `If Age<17 Then` is translated and executed. The second part of the code is only translated if the condition is true. This saves time as only the code that is needed is converted. Some interpreters translate an entire line of code before they execute it. These are known as line interpreters.

Benefits of using an interpreter:

- You do not need to compile the whole program in order to run sections of code. You can execute the code one statement at a time.
- As the code is translated each time it is executed, program code can be run on processors with different instruction sets.
- Because of this, an interpreter is most likely to be used whilst a program is being developed.

Drawbacks of using an interpreter:

- No matter how many times a section of code is revisited in a program it will need translating every time. This means that the overall time needed to execute a program can be very long.
- The source code can only be translated and therefore executed on a computer that has the same interpreter installed.
- The source code must be distributed to users, whereas with a compiled program, only the executable code is needed.

**KEYWORD**

**Compiler**: a program that translates a high-level language into machine code by translating all of the code.

### Compiler

A **compiler** converts the whole source code into object code before the program can be executed. The good thing about this is that once you have carried out this process you will have some object code that can be executed immediately every time, so the execution time will be quick. This is ideal once you have sorted out all the bugs in your program.

Benefits of using a compiler:

- Once the source code has been compiled you no longer need the compiler or the source code.
- If you want to pass your object code on to someone else to use they will find it difficult to work out what the original source code was. This process of working out what the source code was is known as reverse engineering.

Drawbacks of using a compiler:

- Because the whole program has to be converted from source code to object code every time you make even the slightest alteration to your code, it can take a long time to debug.
- The object code will only run on a computer that has the same platform.

## Bytecode

**KEYWORD**

**Bytecode**: an instruction set used for programming that can be executed on any computer using a virtual machine.

Some programming languages use **bytecode**, which is an instruction set that can be executed using a virtual machine. The virtual machine can emulate the architecture of a computer, meaning that the source code written using bytecode can be translated into a format that can be executed on any platform.

For example, Java bytecode is compiled using either one or two bytes to define the instruction and then any number of bytes to pass the parameters. This code can then be executed on any computer that is running the Java Virtual Machine regardless of what type of processor or operating system is being used.

Microsoft Common Intermediate Language (CIL) works on a similar basis where, as the name suggests, rather than translating source code into machine code that is specific to a platform, the source code is translated into an intermediate code. This intermediate code can then be executed by the virtual machine.

**Practice questions can be found at the end of the section on page 264.**

## KEY POINTS

- There are three main types of programming languages: machine code, assembly language and high-level languages.
- Machine code and assembly language are known as low-level languages.
- Machine code uses 0s and 1s.
- Assembly languages use mnemonics.
- High-level languages use natural language keywords.
- Assembly language needs to be converted to machine code using an assembler.
- High-level languages need to be converted to machine code using an interpreter or a compiler.
- There are three main programming paradigms: imperative (procedural), declarative and object-oriented.
- Bytecode is an instruction set that can be implemented using a virtual machine and is therefore platform-independent.

## TASKS

1. Under what circumstances would you compile a high-level computer program?
2. Explain the benefits of using a high-level language compared to an assembly language.
3. Explain the differences between a compiler and an interpreter.
4. Why are machine code and assembly languages said to be machine-oriented?
5. Explain why there are so many different high-level languages.
6. Explain why some programmers still write programs in an assembly language.
7. Explain why assembly language and machine code are said to have a 'one-to-one' relationship.
8. Why must assembly language programs be assembled before they can be executed?

## STUDY / RESEARCH TASKS

1. What is meant by the term artificial intelligence and what type of programming language might you use if working in this area?
2. What features would you expect to find in a programming language that was designed to work in a control environment?
3. There are dozens of programming languages available. Select one particular language, e.g. Java, C#, Python or Visual Basic, and explain why the language was originally developed.