Software

IGCSE Computer Science

Hardware and software

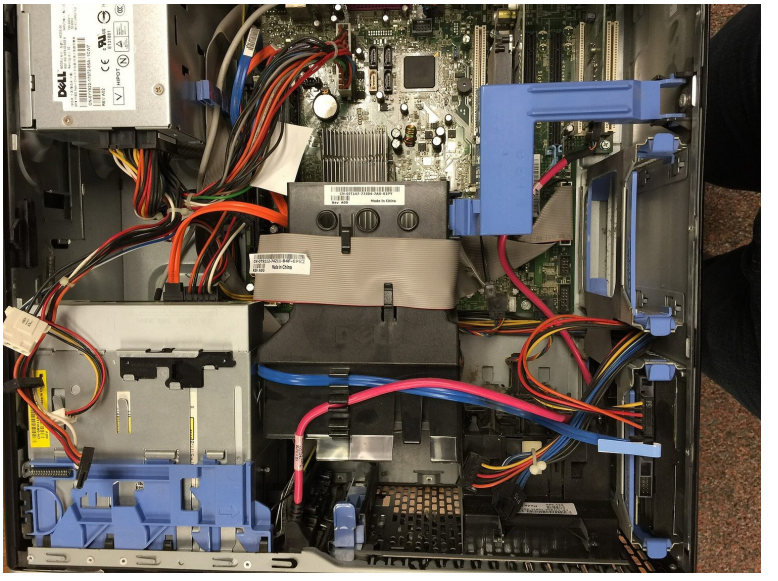
A computer system is made of two parts that work together.

hardware (physical parts)	software (programs)
CPU, keyboard, RAM	apps, the OS

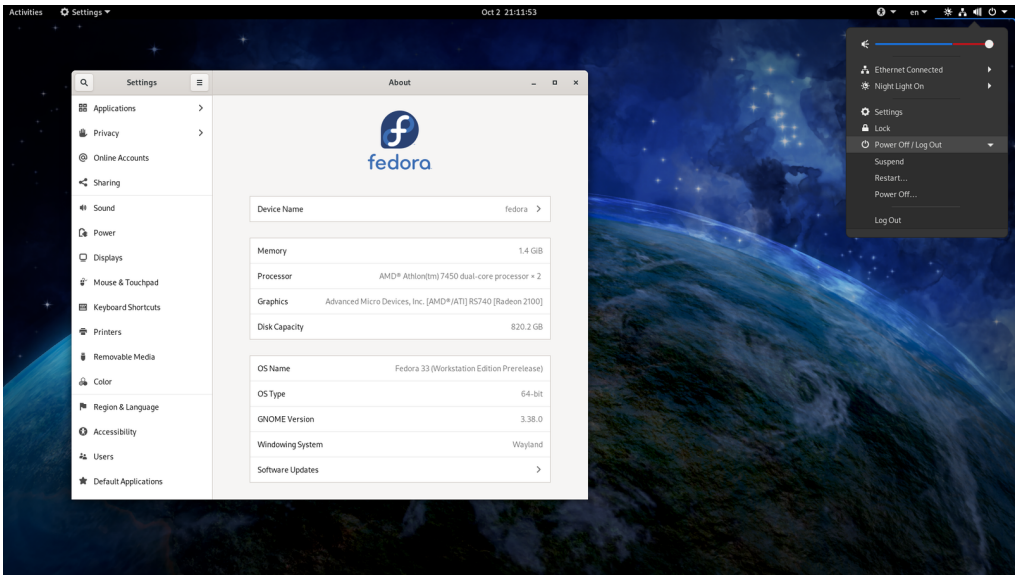
Hardware is the physical parts; software is the programs

- **Hardware** is the **physical** parts of the computer – the parts you can touch. Examples: the CPU, memory, keyboard, screen and hard disk.
- **Software** is the **programs** that control the computer and tell the hardware what to do.

Hardware cannot do anything useful on its own. Software needs hardware to run on. Both are needed.



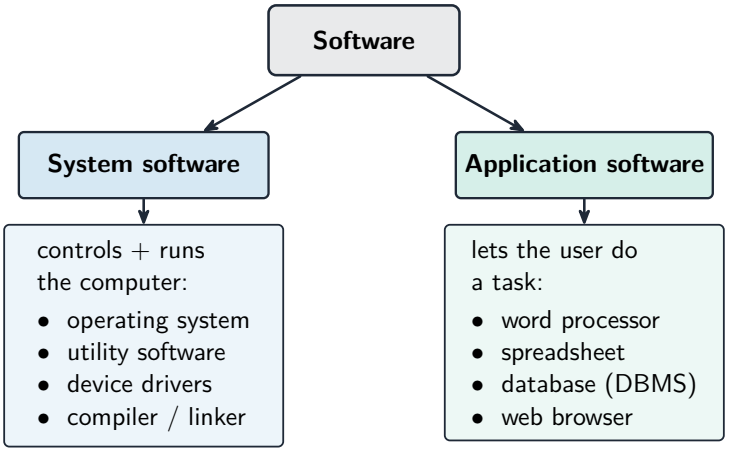
The motherboard and the parts attached to it are hardware – the physical parts you can touch



An operating system is software – the programs that tell the hardware what to do

Types of software

Software is split into two types.



Software divides into system software (runs the computer) and application software (lets the user do a task)

■ System software

System software controls the computer itself and gives a base for other programs to run on. Examples:

- **operating system** (see below);

- **compiler** and **linker** (tools that turn programs into a form the computer can run);
- **device driver** – software that lets the operating system control a piece of hardware;
- **utility software** – small tools that look after the computer (for example anti-virus, backup, disk clean-up).

Below the operating system sits **firmware** – software stored permanently on ROM or flash that runs the moment the computer is switched on. The **BIOS** and the **bootloader** are firmware: they test the hardware and then load the operating system. So the layers stack up **hardware → firmware → operating system → application software**, each running on the layer below.

■ Application software

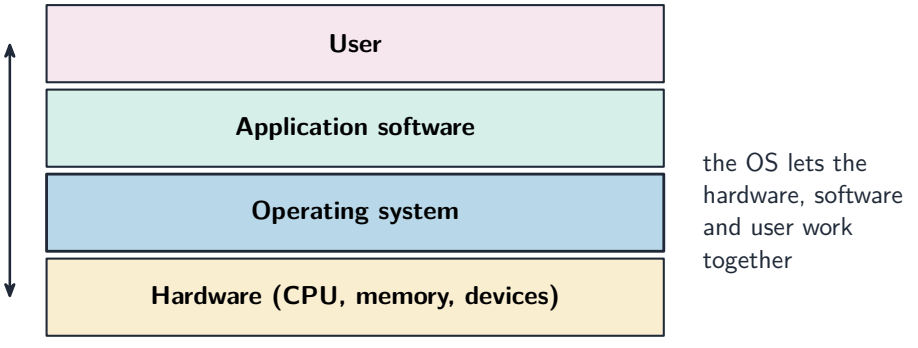
Application software lets the user do a useful task. Examples:

- **word processing** software (writing documents);
- **spreadsheet** software (working with numbers in a grid);
- **database management system** (storing and searching data);
- **web browser** (viewing web pages).

	System software	Application software
Job	runs and manages the computer	helps the user do a task
Used by	the computer (in the background)	the user (directly)
Examples	operating system, drivers, utilities	word processor, browser, games

The operating system

An **operating system** (OS) is the main system software. It controls the whole computer and lets the hardware, the application software and the user work together. Without an OS the computer is very hard to use.



The operating system sits between the hardware and the application software and user, letting them work together

The OS has many roles:

- **managing files** – saving, opening, naming, moving and deleting files;
- **handling interrupts** (see below);
- **providing a user interface** so the user can control the computer;
- **managing memory** – deciding what is kept in RAM and where;
- **managing peripherals** and their drivers – controlling devices like printers;
- **managing multitasking** – letting several programs run at once;
- **providing a platform** for running application software;
- **managing security** – user accounts, passwords and access rights.

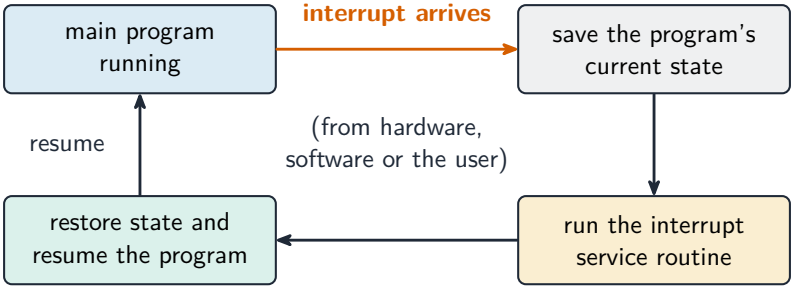
Interrupts

An **interrupt** is a signal sent to the **processor** to tell it that something needs attention now.

An interrupt can come from three sources:

- **hardware** – for example a key is pressed, or a printer runs out of paper;
- **software** – for example a program tries to divide by zero;
- **the user** – for example the user presses a "cancel" key.

When an interrupt arrives, the processor **stops** what it is doing, saves its place, and **services** (deals with) the interrupt. When that is done, it goes back to what it was doing before. This lets the computer react quickly to important events.



On an interrupt the CPU saves its place, runs the interrupt service routine, then restores its place and carries on

Programming languages

All computers process data using the **central processing unit** (CPU). To make the CPU do work, people write **programs** in a **programming language** . There are two kinds.

■ High-level languages

A **high-level language** is written in words that look a bit like English (for example Python). It is:

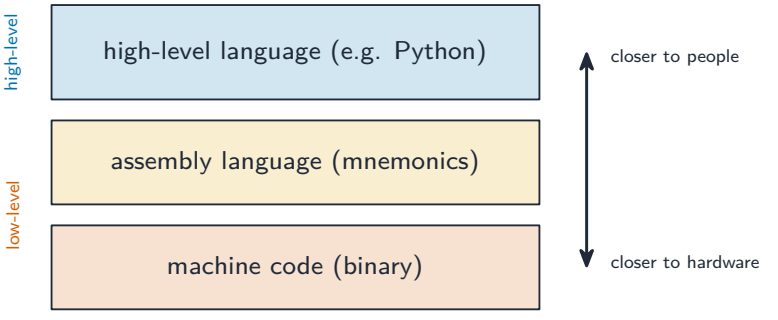
- **easy** for people to read, write and fix;
- **independent of the computer** – the same program can run on different types of computer.

■ Low-level languages

A **low-level language** is close to what the hardware actually uses. It includes:

- **machine code** – instructions written in binary (0s and 1s), which the CPU runs directly;
- **assembly language** – machine code written with short word codes (mnemonics) instead of binary.

A low-level language relates to the **specific** type of CPU, so a program written for one CPU may not run on another. It is used when the programmer needs full control of the hardware or very fast, small code.



High-level languages are close to human language and run on any computer; low-level languages (assembly and machine code) are close to the hardware and tied to one type of CPU

Translators

The CPU can only run machine code. A high-level program must first be changed into machine code. The software that does this is a **translator**. There are three types: a compiler and an interpreter (for high-level code), and an assembler (for low-level code).

Compiler

A **compiler** translates the **whole** program into machine code **before** it is run. After translating, the machine code can be run many times without translating again.

- translation happens once, in full;
- it provides an error report for the whole source, with the translation errors it detects;
- the finished program runs fast and does not need the compiler to run.

Interpreter

An **interpreter** translates and runs the program **one line at a time**.

- it stops at the **first error** it finds, which helps when testing;
- the program runs more slowly, because it is translated each time it runs;
- the interpreter must be present every time the program runs.

Use a **compiler** when you want to share a fast, finished program. Use an **interpreter** when you are writing and testing a program and want to find errors easily.

Worked comparison – when does execution happen?

> **Try it.** Use the syllabus's whole-program compiler and line-by-line interpreter models. Consider this short source description; it is not a program in a particular language.

Line	Source description
1	A valid instruction displays `Start`
2	An incomplete addition is missing a value after the plus sign
3	A valid instruction displays `Done`

Line 2 contains a translation error. What is displayed before that error is reported with each translator? Will line 3 execute? Explain the order of translation and execution.

Compiler trace. The whole source is translated before execution. The error in line 2 causes compilation to fail. An error report is provided for the whole source. The new program is not executed, so neither `Start` nor `Done` is displayed.

Interpreter trace. Line 1 is translated and executed, displaying `Start`. The interpreter encounters the error in line 2 and stops execution there. Line 3 is not reached, so `Done` is not displayed.

These traces illustrate the syllabus model; real language tools can combine approaches. A translator reports errors it can detect, not every mistake in the program's logic.

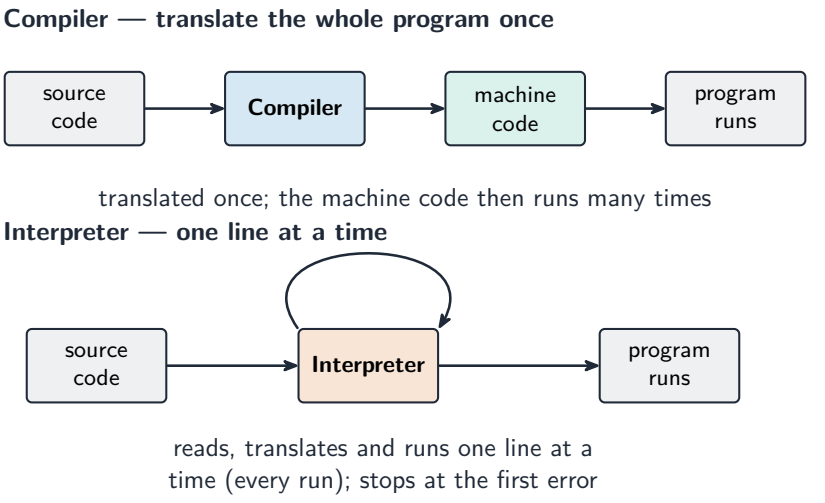
Transfer – development and release. A team wants to locate errors while testing short sections. Later, customers must run the finished program repeatedly without installing its translator. Choose a translator for each stage and explain why. In the **integrated development environment** (IDE), name one tool for

entering source and one for investigating a reported error. Finally, decide whether successful translation proves that a calculated total is correct.

Worked answer. In the syllabus model, an interpreter suits development: line-by-line translation and execution stop at an encountered error, helping the team locate it. A compiler suits release: successful compilation creates an executable file that can be run repeatedly without the compiler. Unchanged source does not need translation again for every run. A **code editor** enters and changes source; **error diagnostics** help locate and explain reported errors; the **run time environment** lets the team test the program. A wrong total can still result from faulty logic after successful translation. Test with known inputs, compare actual and expected results, and investigate the calculation.

■ Assembler

An **assembler** translates a **low-level assembly-language** program (written in mnemonics such as `LDA`, `ADD`) into machine code. Each mnemonic becomes **one** machine instruction – a direct one-to-one translation, unlike a compiler or interpreter, which work on high-level code.



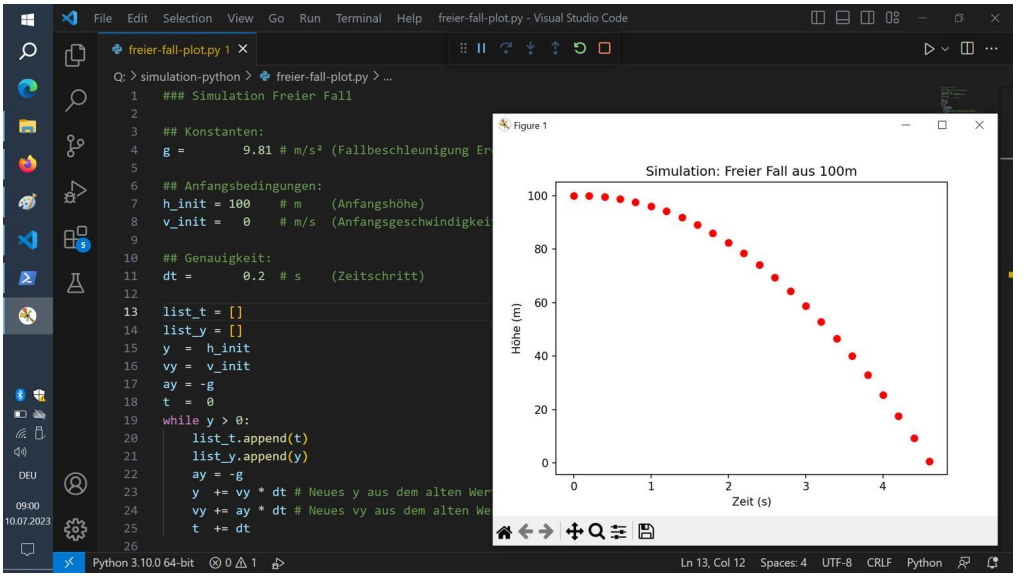
A compiler translates the whole program once into machine code; an interpreter translates and runs one line at a time

> **Worked example.** A team is writing a program. While developing it they want to find and fix errors quickly; when they ship it they want customers to run it fast without seeing the source code. Which translator suits each stage? While developing, use an **interpreter**: it translates and runs one line at a time and stops at the first error, so a mistake is easy to locate. To ship, use a **compiler**: it translates the whole program once into machine code, which then runs quickly with no translator present, and the customer receives only the executable rather than the source. The two are not rivals - name the **stage** and give the reason, because "a compiler is faster" on its own earns nothing.

Integrated development environment (IDE)

An **integrated development environment** (IDE) is software that helps you write programs. It puts many useful tools in one place:

- a **code editor** – for typing in your program;
- a **run time environment** – for running the program to test it;
- a **translator** – a compiler or interpreter to turn your code into machine code;
- **error diagnostics** – messages that help you find and understand mistakes;
- **auto-completion** and **auto-correction** – the IDE finishes or fixes code as you type;
- **prettyprinting** – laying out the code neatly with colour and indentation so it is easy to read.



Source code in an editor: an IDE combines editor, debugger and run tools in one place

Exam tips

- **Hardware** is the physical parts; **software** is the programs. **System software** runs the computer (OS, drivers, utilities); **application software** does a task for the user.
- A **compiler** translates the whole program at once (the finished program runs fast, and all errors are reported at the end); an **interpreter** translates line by line (it stops at the first error, runs slower, and is needed every time).
- **High-level** languages are close to English and run on any computer; **low-level** languages (assembly, machine code) are tied to one type of CPU.
- An **interrupt** is a signal that makes the CPU stop, save its place, service the event, then carry on where it left off.
- Learn the operating system's roles: managing files, memory, peripherals, multitasking, security and the user interface.