

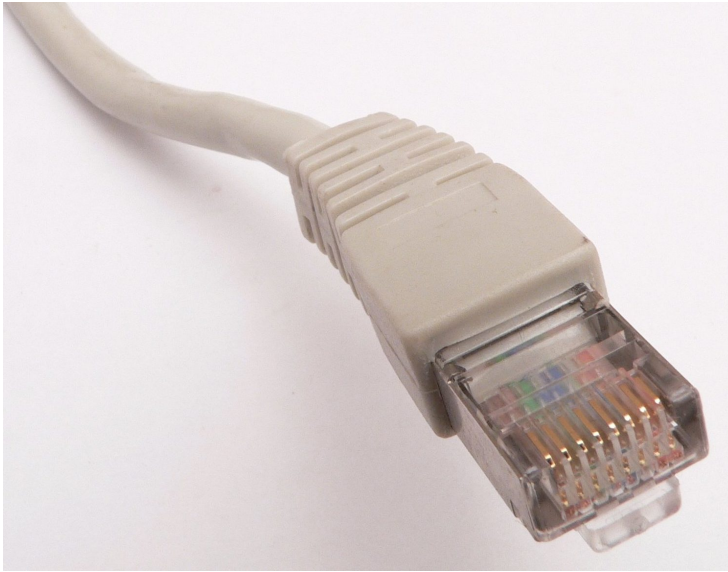
Data transmission

IGCSE Computer Science

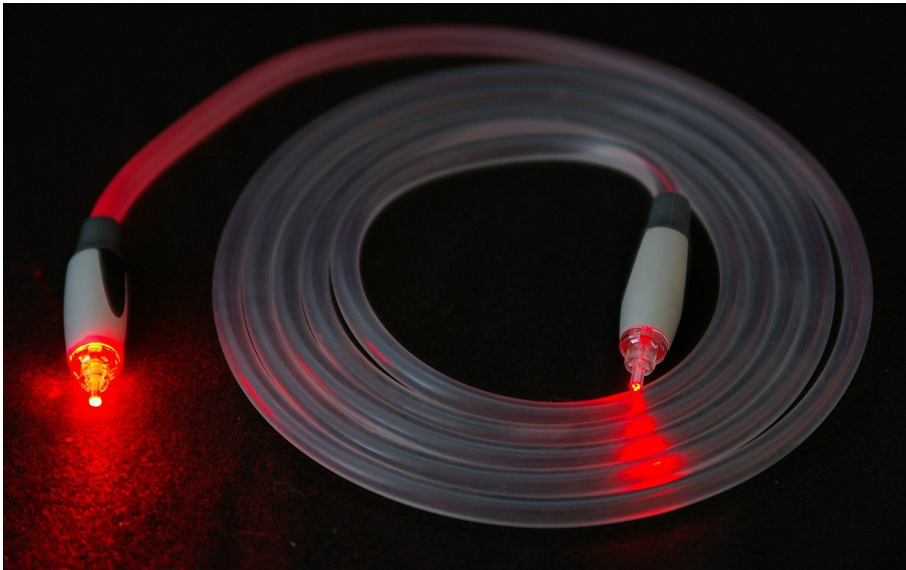
What is data transmission?

Data transmission means moving data from one place to another – for example from your computer to a website, or from a phone to a printer.

Data often travels along a cable. Two common ones are shown below.



An Ethernet cable with an RJ45 plug, often used to carry data on a wired network



A fibre-optic cable carries data as pulses of light along thin glass fibres

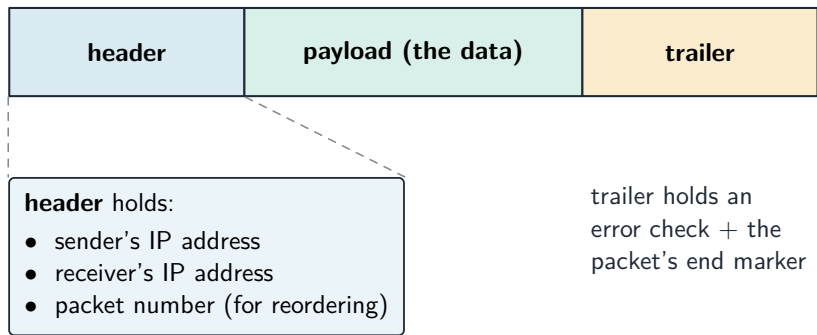
Before it is sent, the data is broken down into small, equal-sized pieces called **packets**. Each packet travels separately and they are put back together at the other end.

Using packets has benefits:

- if one packet is lost or damaged, only that packet is sent again, not the whole file;
- packets can take different routes, so a busy or broken route can be avoided;
- many users can share the same connection at the same time.

The structure of a packet

Every packet has three parts.



A packet has a header, a payload (the data) and a trailer; the header carries the addresses and packet number

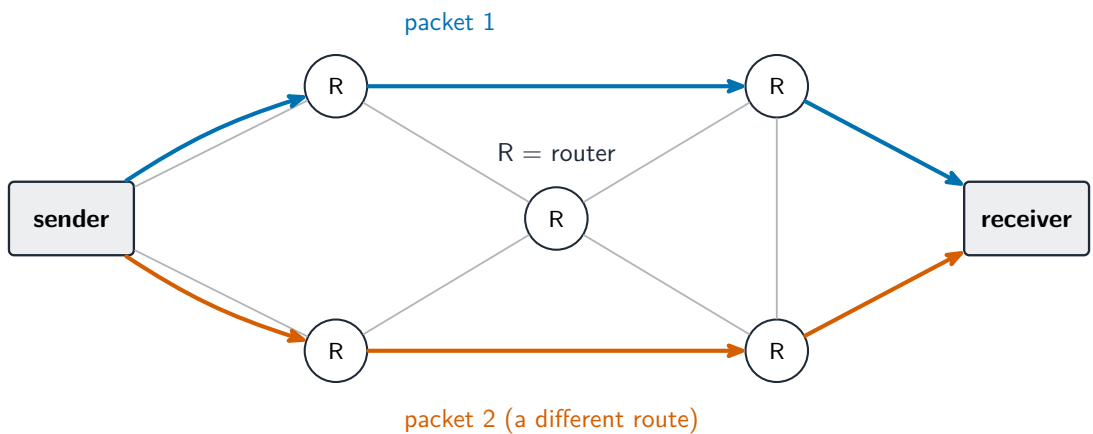
Part	What it holds
packet header	control information (see below)
payload	the actual data being carried
trailer	shows where the packet ends, plus an error check

The **packet header** holds:

- the sender's IP address (where the packet came from),
- the receiver's IP address (where it is going),
- the **packet number** (so the packets can be put back in the right order).

Packet switching

Packet switching is the way packets are sent across a network.



Routers forward packets, and different packets can take different routes to the same receiver

- Special devices called **routers** read each packet's header and choose the best route for it.
- Each packet may take its **own path**, so packets can travel different ways.
- Packets may **arrive out of order**.
- When the last packet has arrived, the packets are **reordered** using their packet numbers.

Methods of data transmission

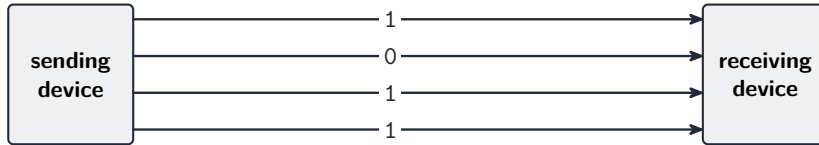
■ Serial and parallel

	Serial	Parallel
How	one bit at a time, down one wire	several bits at once, down several wires
Distance	good over long distances	good over short distances only
Cost	cheaper (fewer wires)	more expensive (more wires)
Errors	bits stay in order	bits can arrive at slightly different times over long wires

Serial sends one bit after another along a single wire. **Parallel** sends several bits at the same time along several wires, so it is faster over short distances.



serial: one bit at a time, one wire



parallel: several bits at once, several wires

Serial sends one bit at a time down one wire; parallel sends several bits at once down several wires

■ Simplex, half-duplex and full-duplex

These describe the *direction* data can travel.



Simplex is one way only; half-duplex is both ways but one at a time; full-duplex is both ways at once

Method	Direction
simplex	one direction only (e.g. computer → printer)
half-duplex	both directions, but only one at a time (e.g. walkie-talkie)
full-duplex	both directions at the same time (e.g. phone call)

■ Choosing a method

The choice depends on the **distance**, the **speed** needed, the **cost**, and whether data must travel both ways at once.

USB

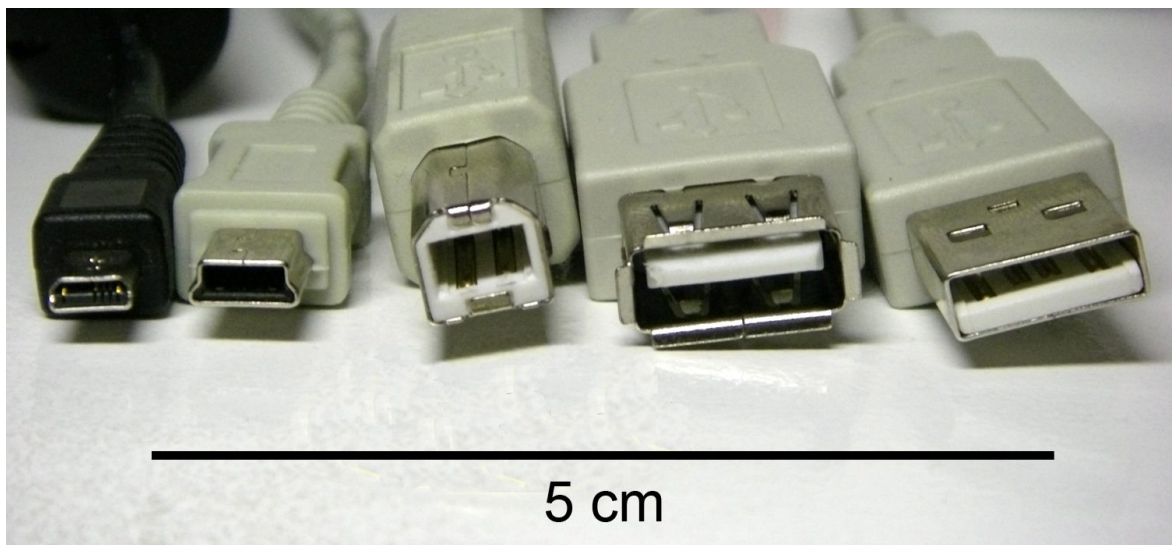
The **universal serial bus** (USB) is a common **interface** (a connection point) for joining devices to a computer.

Benefits:

- the same connector is used by many devices (one standard);
- the device is detected automatically and the correct driver is loaded;
- it carries power, so it can charge or run a device;
- the connector only fits one way, so it is hard to plug in wrongly.

Drawbacks:

- the cable can only be a limited length (a few metres);
- older USB versions are slower than newer ones;
- transfer speeds can be lower than some other connection types.



A few of the USB connector shapes. All use the same USB standard, so devices are detected automatically.

Why we check for errors

While data travels, **interference** can change it. Three things can go wrong:

- **data loss** – some bits do not arrive;
- **data gain** – extra bits are added;

- **data change** – some bits are flipped.

So the receiver checks whether the data arrived correctly.

Error detection methods

■ Parity check

A **parity check** adds one extra bit, the **parity bit**, to each byte. There are two types:

data: 1 0 1 1 0 0 1

count the 1s: four (even)

parity bit = 0 (keep it even)

sent: 1 0 1 1 0 0 1 0

A parity bit is added to make the number of 1s even (or odd)

- **even parity** – the total number of 1s in the byte (including the parity bit) must be **even**;
- **odd parity** – the total number of 1s must be **odd**.

Example (even parity): the data 1011001 has four 1s, which is already even, so the parity bit is 0:

parity bit + data
0 1011001

To find an error: the receiver counts the 1s. If even parity was agreed but a byte arrives with an **odd** number of 1s, an error has happened during transmission.

A parity check cannot find every error. If two bits flip, the count may still look correct.

Worked example. A single parity bit tells you *that* an error happened, but not *where*. A **parity block** can find the exact bit. Several bytes are sent together, each with a row parity bit, and one extra **parity byte** at the bottom holds a parity bit for each column. Even parity is used below, and one bit was flipped during transmission. Find it.

	b1	b2	b3	b4	b5		parity	
byte 1	1	0	1	1	0		1	
byte 2	0	1	1	0	1		1	
byte 3	1	1	1	1	0		1	<- row: odd number of 1s
byte 4	0	0	1	1	0		0	

parity	0	0	1	1	1		1	
			^ column b3: odd number of 1s					

Row *byte 3* breaks even parity, and column *b3* breaks even parity. The flipped bit is where they cross: **byte 3, bit b3**. Changing it back from 1 to 0 repairs the data. A single parity bit only says an error happened; a parity block says exactly where, so the receiver can even fix it.

■ Checksum

A **checksum** is a value worked out from all the data before sending. The receiver works out the checksum again from the data it received. If the two values do not match, an error has occurred and the data is resent.

■ Echo check

In an **echo check**, the receiver sends the data back to the sender. The sender compares it with the original. If they differ, an error happened. (This needs the data to be sent twice, so it is slow.)

■ Automatic Repeat reQuest (ARQ)

Automatic Repeat reQuest (ARQ) uses messages called **acknowledgements**.

The syllabus calls this automatic repeat query; automatic repeat request is another name for the same method. The sender starts a **timeout** timer when it sends a packet. The receiver uses an error check on the received packet. If the check passes, it sends a **positive acknowledgement**. If an error is detected, it sends a **negative acknowledgement**.

Try it – packet 4. In three separate cases, packet 4 (A) passes the receiver's error check, (B) fails the check, or (C) never arrives. For each case, choose the receiver's response and the sender's next action.

Worked trace.

Case	Receiver	Sender
A	Sends a positive reply	Receives it; stops timer; can send next packet
B	Sends a negative reply	Receives it; resends packet 4; restarts timer
C	Has no packet to check	Timeout; resends packet 4; restarts timer

A negative reply can cause a repeat before the timer expires. If no positive acknowl-

edgement arrives before timeout, the sender repeats the transmission. The timer handles silence, including a lost packet or a lost reply.

Transfer – a lost reply. Packet 4 arrives and passes its check. The receiver sends a positive acknowledgement, but this reply is lost. What does the sender do when its timer expires? Does timeout prove that packet 4 was damaged?

Answer. The sender resends packet 4 and restarts its timer. The receiver knows the packet arrived successfully, but the sender has no confirmation. Timeout does not tell the sender whether the packet or the reply was lost or delayed. ARQ requests another transmission; it does not itself repair damaged bits. It uses an error check to decide whether to accept a received packet.

■ Check digit

A **check digit** is an extra digit placed at the end of a number, worked out from the other digits. It is used to find mistakes when a number is **typed in**.

When the number is entered, the check digit is calculated again and compared. Depending on the method, it can detect some of these errors; no check digit guarantees that every entry error is found:

- an **incorrect digit** entered,
- a **transposition error** (two digits swapped, e.g. 21 typed as 12),
- a digit **left out or an extra digit** added,
- a **phonetic error** (a digit that sounds similar, e.g. 13 and 30).

Check digits are used in **barcodes** and ISBNs (book numbers).

Encryption

■ Why we need encryption

Encryption scrambles data so that it cannot be understood if it is **intercepted** (caught) by the wrong person. The readable data is called **plaintext**; after encryption it becomes **ciphertext**.

Encryption does **not** stop data from being intercepted. It only stops the data from being understood.

■ Symmetric encryption

Symmetric encryption uses a single **key** to both encrypt and decrypt the data. The sender and receiver must share this same secret key. The risk is that the key itself could be stolen while being shared.

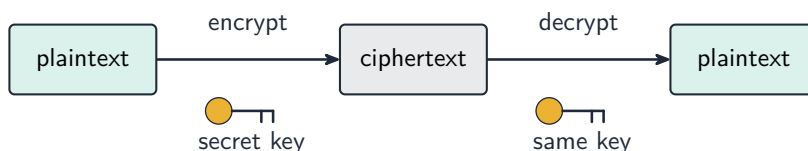
■ Asymmetric encryption

Asymmetric encryption uses two different keys:

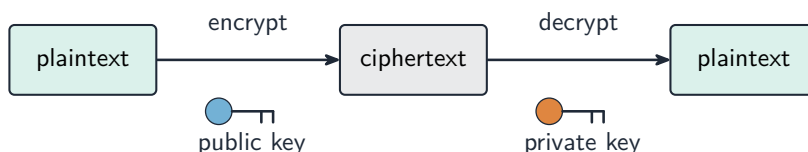
- a **public key** that anyone can have, used to encrypt;
- a **private key** that is kept secret, used to decrypt.

Data encrypted with the public key can only be decrypted with the matching private key, so the key never has to be shared. This is safer than sharing one secret key.

Symmetric: one shared secret key



Asymmetric: a public key and a private key



Symmetric uses one shared key; asymmetric uses a public key to encrypt and a private key to decrypt

Exam tips

- Data is sent in **packets** (a header with the addresses and packet number, a payload, and a trailer). Packets can take different routes and arrive out of order, then are reordered by their packet numbers.
- Match the transmission types: **serial** (one bit at a time, good over long distances) vs **parallel** (several bits at once, short distances); **simplex** / **half-duplex** / **full-duplex** describe the direction.
- A single **parity bit** only shows *that* an error happened; a **parity block** locates the exact wrong bit – the row and the column that both fail.
- Learn the error-detection methods and what each does: parity check, checksum, echo check, ARQ, and the check digit (for typed-in numbers).
- **Symmetric** encryption uses one shared key; **asymmetric** uses a public key to encrypt and a private key to decrypt, so no secret key is ever shared.