

**AQA Computer Science A-Level**  
**4.5.5 Information Coding Systems**  
Advanced Notes



## **Specification:**

### **4.5.5.1 Character form of a decimal digit:**

Differentiate between the character code representation of a decimal digit and its pure binary representation.

### **4.5.5.2 ASCII and Unicode:**

Describe ASCII and Unicode coding systems for coding character data and explain why Unicode was introduced.

### **4.5.5.3 Error checking and correction:**

Describe and explain the use of:

- parity bits
- majority voting
- checksums
- check digits



## Character form of a decimal digit

When computers need to represent a **character** like R, k or \$, an **information coding system** is used to match **characters** to **character codes**.

A character code is a **decimal digit** used to represent a character. For example, a primitive information coding system might assign the numbers **1 to 26** to the letters **A to Z** like so:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

In the above example, the numbers 1 to 26 are decimal digits that can be used to represent their corresponding character from A to Z. If the information coding system were to be used by a computer, the **binary** representation of the decimal digits would be used.

Character code		Character
Decimal	Binary	
13	1101	M

## ASCII and Unicode

ASCII (American Standard Code for Information Interchange, pronounced *ah-ski*) and Unicode are two **widely used** information coding systems.

Introduced in 1963, ASCII makes use of **7 bits** to represent **128** ( $= 2^7$ ) different characters including A to Z, a to z, 0 to 9 and various symbols.

As the Internet became widely used **throughout the world**, there was a requirement for an information coding system that could represent not just the Latin alphabet but also alphabets like Arabic, Cyrillic, Greek and Hebrew.

Unicode was introduced in 1991 to allow the representation of a **wide variety of alphabets** by computers. The standard uses **anywhere from 8 to 48 bits** (1 to 6 bytes) per character, allowing it to represent a **much wider range of different characters** than ASCII.



## Error checking and correction

When data is transmitted from computer to computer, **errors** can occur that cause the data to change during transmission which could have inconvenient or even catastrophic consequences. In order to **reduce the chances** of incorrect data being used, a number of error checking and correction principals have been created.

### Parity bits

A parity bit is a **single bit** added to a transmission that can be used to **check for errors** in the transmitted data. Its value is calculated **based on the transmitted data itself**.

There are two types of parity bit, **even** parity and **odd** parity.

In **even** parity, the value of the parity bit is chosen so as to make **the total number of 1s in the transmitted data even**. For example, if the data 01101110 (which contains 5 1s) were to be transmitted, the parity bit would be set to 1, so that the total number of 1s is even.

**Odd** parity works in a similar way to even parity, but adds a parity bit so that **the total number of 1s** in the transmitted data is **odd**.

When data is received, a **parity check** is carried out. If the value of the received parity bit conforms to the type of parity (odd or even) in use, then the received data is treated as correct. Otherwise, the computer will request that the sender **re-transmits** the data.

Data to transmit	Even parity applied		Data received	Parity check
1101	11011	→	11011	No error detected
0000	00000	→	00100	Error detected
0100	01001	→	11001	Error detected
1001	10010	→	11110	No error detected



If the first example, there is **no error** in transmission. When the parity check is applied, no error is found and so the transmitted data is treated as correct.

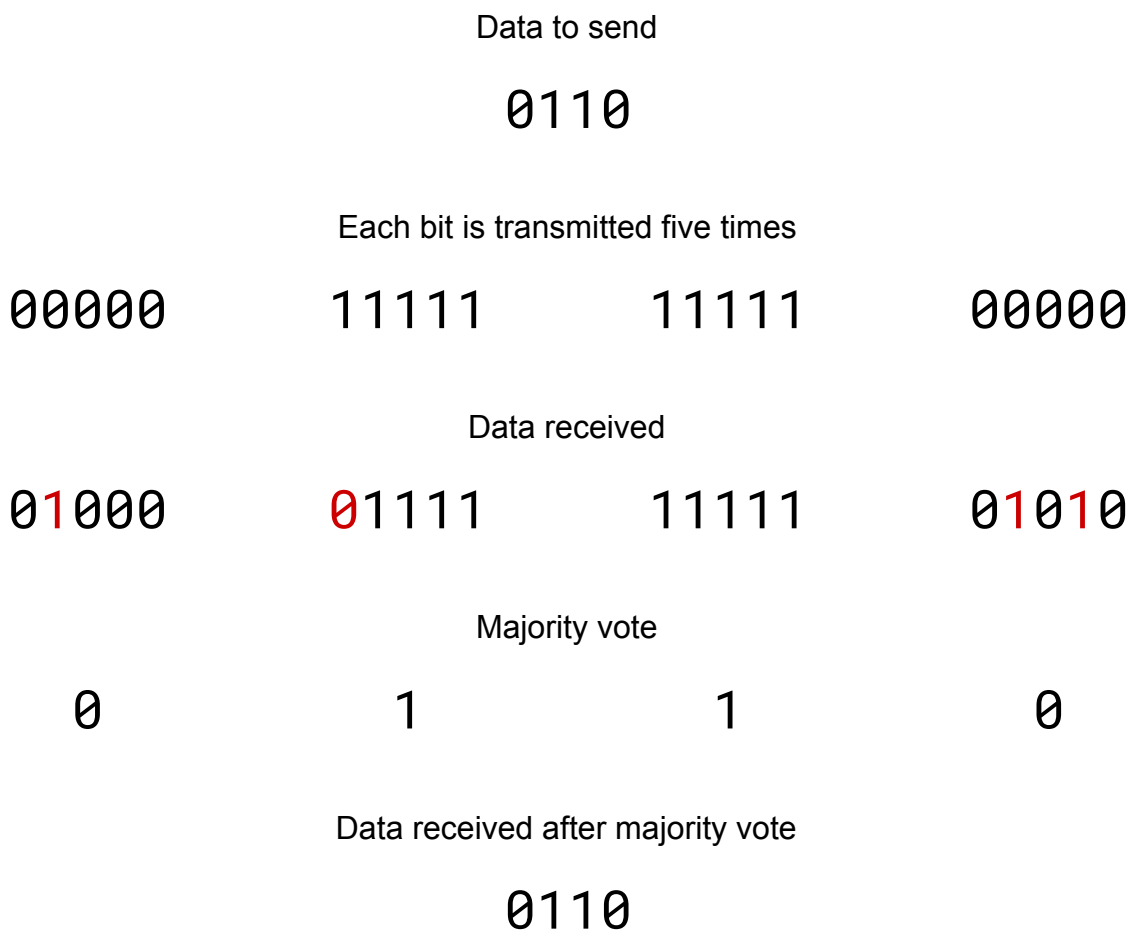
In the second and third examples, an error has resulted in the value of **1 bit being changed** (highlighted in red). After a parity check is applied, **the error is detected** and the computer would request that the data is retransmitted.

In the fourth example, an error has resulted in the values of **two bits changing**. However, when a parity check is applied, **no error is detected** as the total number of 1s in the data is still even.

This highlights the **major issue** with parity bits. Whether using odd or even parity, if an **even number of bits are changed** during transmission, the error **is not detected**.

### Majority voting

When using majority voting, each bit of the data is **transmitted multiple times**. When the data is received, the **most commonly occurring value** is taken to be correct.



As shown in the example, when an error occurs and the value of a bit is changed (shown in red), majority voting doesn't just detect the error but also **corrects the error**, meaning there's **no need for retransmission** like when using a parity bit.

The example also demonstrates majority voting's capability to correct errors when the values of **multiple bits** have changed, another advantage of majority voting over parity bits.

The primary **disadvantage** of majority voting is that the **volume of data being transmitted is increased** with the repetition of bits. In the example above, the data transmitted is **five times larger** than the original data. This would **significantly increase** the time taken to transmit data.

### Checksums

As with parity bits, checksums involve adding a value, **determined by the data itself**, to the transmitted data.

An algorithm is used to determine the value of a checksum based on the data being transmitted. There is **no agreed algorithm** for this and different systems will use their own solutions. A simple algorithm that could be applied is the **modulo** function, which returns the remainder after a division.

Data to send

$$46_{10} = 101110_2$$

Calculate value of checksum

$$46_{10} \text{ MOD } 8_{10} = 6_{10} = 110_2$$

Data transmitted

$$101110110_2$$

In the example above, the value of the checksum is calculated using the function **MOD 8** which returns the remainder when the value to send is divided by 8. This value is then **appended** to the original data in binary before being transmitted.



Once received, the recipient can **remove the checksum** and **apply the same algorithm** as was used when sending the data to ensure that the checksum matches the transmitted data. If the two do not match, the recipient **cannot correct the error itself** so must request that the sender **re-transmits** the data.

### Check digits

A check digit is a **type of checksum** in which only a **single digit** is added to the transmitted data. This **reduces the number of different algorithms** that could be used to calculate the value of the check digit and so **reduces the variety of errors** that the method can detect.

	<b>Can <i>detect</i> errors in transmission</b>	<b>Can <i>correct</i> errors in transmission</b>	<b>Efficiency</b>
<b>Parity bit</b>	Yes - but only if an odd number of bits are changed	No	Very efficient
<b>Majority vote</b>	Yes	Yes - as long as the majority of bits remain unchanged	Inefficient - each bit is sent multiple times
<b>Checksum</b>	Yes	No	Mostly efficient - a complex algorithm could make the process less efficient
<b>Check digit</b>	Yes	No	Efficient - the algorithms used to calculate the check digit are limited in complexity

