Dr SABER Takfarinas
takfarinas.saber@dcu.ie

CA169
Networks & Internet

# Link Layer
# 2- Errors Control

# Errors Control

- What the physical layer does is accept a raw bit stream and attempt to deliver it to the destination
- If the channel is noisy, the physical layer will add some redundancy to its signals to reduce the bit error rate
- However, the bit stream received by the data link layer is not guaranteed to be error free
- Some bits may have different values and the number of bits received may differ from the number of bits transmitted
- It is up to the data link layer to **detect** and, if necessary, **correct** errors

# Detecting Errors

- The usual approach is for the data link layer to break up the bit stream into discrete frames, compute a short token called a checksum for each frame, and include the checksum in the frame when it is transmitted
- When a frame arrives at the destination, the checksum is recomputed
- If the newly computed checksum is different from the one contained in the frame, the data link layer knows that an error has occurred and takes steps to deal with it

# Error Control

- As we know when a signal is transmitted by the physical layer attenuation and distortion can cause the signal to be read incorrectly

- There are a number of things we can do
  - Use codes to detect if errors have occurred
  - Use codes to correct errors when they occur
  - Retransmit any lost frames

- Reliability is a concern that appears in multiple layers

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

# Adding Redundancy

- If we add extra information to the frame we can use these to help detect or correct errors
- Error Detection
  - Add **check bits** to messages that let some errors be detected
- Error Correction
  - Add more **check bits** to messages that let some errors be corrected
- The hardest problem is to structure the codes to detect as many errors with a **small amount** of extra bits and not too much calculation

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

# Simple Error Detection Example

- A simple code for detecting errors
  - Send two copies, if they are different there has been an error
- How good is this code?
  - How many errors can it detect/correct?
  - How many errors will make it fail?
- How much overhead does it add ? In other words: how much data do we need to send without requiring it to communicate?

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

# Using Error Codes

- Codeword consists of D data bits plus R check bits
  - Called systematic block code
- Number of check bits depends on the size of the data
- Check bits are computed based on the data and then added to the end
- When the receiver gets the package it recomputes the check bits based on the data bits
  - If there are no errors the check bits should match
- When an error is detected it can be difficult to tell if the error is in the data or in the check bits

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

# Single Parity Check

- Use a single check bit such that the number of 1s in every codeword is **even**
  - If the number of 1s in the data is even we add a 0
  - If the number of 1s in the data is odd we add a 1
- This is called **single parity check**
- If receiver gets a codeword with an odd number of 1's, it knows error(s) occurred
  - can **detect** any odd number of bit errors
  - but: can't tell how many errors, or which bits are in error
  - even worse: any even number of bit errors is *undetectable*

# Single Parity Check - Example

- Data to be transmitted: 10110101
  - There are 5 1s in the data so the parity bit is 1
- We transmit: 101101011
  - If receiver gets 101101011 parity check is ok
  - If receiver gets 101100011 parity check fails
  - If receiver gets 101110011 parity check is ok but codeword is incorrect
  - If receiver gets 001100011 parity check is ok but codeword is incorrect
- Data to be transmitted: 10110001
  - There are 4 1s in the data so the parity bit is 0

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

# 2 Dimensional Parity Check

- In this scheme we form data into a 2-dimensional array and add single parity check bits to each row and each column
- If we have the data 1110001 1000111 0011001
- We form a 3x7 array and add row and column parity bits
  - 1 1 1 0 0 0 1 **0**
  - 1 0 0 0 1 1 1 **0**
  - 0 0 1 1 0 0 1 **1**
  - **0 1 0 1 1 1 1 1**

# 2 Dimensional Parity Check

- The receiver knows to form received bit string into 4x8 array, then check the row and column parity bits

- This scheme can **detect** any odd number of bit errors in a row or column
  - It can also detect an even number of bit errors if they're in a single row (using the column parity checks)
  - It can also detect an even number of bit errors if they're in a single column (using the row parity checks)

- This scheme can **correct** any single bit error

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

# Example 1 - 1 Bit Error

- If we have the data 1110001 1000111 0011001
- And the receiver gets
  - 1 1 0 0 0 0 1 0
  - 1 0 0 0 1 1 1 0
  - 0 0 1 1 0 0 1 1
  - 0 1 0 1 1 1 1 1
- The receiver can detect that the bit in position (1, 3) was in error and can correct it

# Example 2 - 2 Bit Errors

- If we have the data 1110001 1000111 0011001
- And the receiver gets
  - 0 1 0 0 0 0 1 0
  - 1 0 0 0 1 1 1 0
  - 0 0 1 1 0 0 1 1
  - 0 1 0 1 1 1 1 1
- The receiver can detect that the bit errors have occurred but it cannot correct them

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

# Example 3 - 2 Bit Errors

- If we have the data 1110001 1000111 0011001
- And the receiver gets
  - 0 1 1 0 0 0 1 0
  - 1 0 1 0 1 1 1 0
  - 0 0 1 1 0 0 1 1
  - 0 1 0 1 1 1 1 1
- The receiver can detect that the bit errors have occurred but it cannot correct them
- The same parity checks would be received if the errors were in positions (1, 3) and (2, 1)

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

# Example 4 - 3 Bit Errors

- If we have the data 1110001 1000111 0011001
- And the receiver gets
  - 0 0 0 0 0 0 1 0
  - 1 0 0 0 1 1 1 0
  - 0 0 1 1 0 0 1 1
  - 0 1 0 1 1 1 1 1
- The receiver can detect that the bit errors have occurred but it cannot correct them
- Two of the bit errors could be on a single row with a correct parity so we cannot correct them

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

# Example 5 - 4 Bit Errors

- If we have the data 1110001 1000111 0011001
- And the receiver gets
  - 0 1 0 0 0 0 1 0
  - 1 0 1 0 1 1 1 0
  - 0 0 1 1 0 0 1 1
  - 1 1 0 1 1 1 1 1
- The receiver can detect that the bit errors have occurred but cannot correct them
- Errors in (1,1), (1,3), (2,1) and (4,3) would have the same parity check

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

# Codewords

- Within any encoding scheme there are a number of **valid** codewords
- This means that whatever check that is performed must be able to determine if an error has occurred by evaluating the codeword.

# Codewords

- We would like to send the data: 1110001 1000111 0011001
- If we use **single parity** check what is the sent Codeword?
  - 1110001 1000111 00110011
- If we using **2D parity** what is the sent Codeword?
  - 11100010 10001110 00110011 01011111

# Codewords

- Single Parity example:
  - Receiver: 1111001 1000111 00110011
    - Counts the ones
    - Finds problem with codeword
    - ➢ Drops the packet as it cannot fix it
- 2D Parity example:
  - Receiver: 11110010 10001110 00110011 01011111
    - Places codeword into array
    - Finds problem with codeword
    - Fixes the error
    - Keeps fixed codeword -> Data: 1110001 1000111 0011001

error

Why does this occur?

# Hamming Distance

- The Hamming distance between two codewords W1 and W2 is the **number of bits** that must be flipped to change one to the other
- We write this as d(W1, W2)
- Example:
  - W1 =10001001
  - W2 =10110001
  - d(W1, W2) = 3

# Hamming Distance

- For any error detection/correction scheme, we can define the minimum Hamming distance **D** (or “minimum distance”) of the scheme as the ***smallest number*** of bit errors that changes one **valid** codeword into another.

- Each codeword is made up of **m** data bits and **r** check bits
  - Therefore **n = m + r** total bits

- For data all possible $\mathbf{2^m}$ data strings are usually valid
- But because the check bits are calculated based on the data bits not all $\mathbf{2^n}$ possible codewords are valid

# Hamming Distance

- If the method of computing the check bits is known, the list of **all valid codewords** can be calculated and stored at the receiver
- When a word **W** is received the receiver finds the closest valid codeword to **W** using the Hamming distance and takes this as the transmitted codeword
- The **minimum Hamming distance** of a **scheme** is the smallest Hamming distance between all possible pairs in a set of words in the scheme.
- This is called General Parity Check

# General Parity Check

- If the minimum distance of an error-handling scheme is D, this scheme:
  - can **detect** any combination of **≤ D-1** bit errors
  - can **correct** any combination of **< D/2** bit errors

- Alternatively:
  - If you want to **detect** B bit errors, use a scheme with minimum distance (D) at least B+1;
  - If you want to **correct** B bit errors, use a scheme with minimum distance (D) at least 2B+1

# General Parity Check Examples

- Example - Single parity check
  - This scheme has a minimum Hamming distance of D = 2
  - We noted that it could **detect** only any single bit error but **cannot correct** any bit error
    - Detect ≤ D–1 = 1
    - Correct < D/2 = 1

- Example 2-dimensional parity check
  - This scheme has a minimum Hamming distance of D = 4
  - We noted that it can ***detect*** any combination of ≤ 3 bit errors and can only ***correct*** any single bit error
    - Detect ≤ D–1 = 3
    - Correct < D/2 = 2

# General Parity Check Examples

- Suppose we have only 4 valid codewords
  - C1 = 00000 00000
  - C2 = 00000 11111
  - C3 = 11111 00000
  - C4 = 11111 11111
- The minimum hamming distance is 5
  - Any combination of ≤ 4 bit errors can be detected
  - Any combination of < 2.5 bit errors can be corrected

C1 = 00000 00000
C2 = 00000 11111
C3 = 11111 00000
C4 = 11111 11111

# General Parity Check Example

- If we transmit 00000 00000 but receive X = 00000 00011
  - We know it is not a valid codeword
  - Therefore, we compute the hamming distance between X and all the valid codewords

- d(X, C1)=2, d(X, C2)=3, d(X, C3)=7, and d(X, C4)=8

- The receiver takes C1 = 00000 00000 as transmitted codeword ***(errors corrected)***

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

C1 = 00000 00000
C2 = 00000 11111
C3 = 11111 00000
C4 = 11111 11111

# General Parity Check Example

- If we transmit 00000 00000 but receive X = 00000 00111
  - We know it is not a valid codeword
  - Therefore, we compute the hamming distance between X and all valid code words

- d(X, C1)=3, d(X, C2)=2, d(X, C3)=8, and d(X, C4)=7

- The receiver takes C2 = 00000 11111 as transmitted codeword ***(in this case, error correction fails)***

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

# General Parity Check

- What is the problem with the schemes we have discussed:

- Single Parity Check:
  - Cannot Correct any bit error

- 2D parity Check:
  - Can correct 1 bit error, but uses too many bits to achieve this!!

- There are schemes that can do this more efficiently. E.g.,
  - Hamming Codes
  - Cyclic Redundancy Check (CRC)

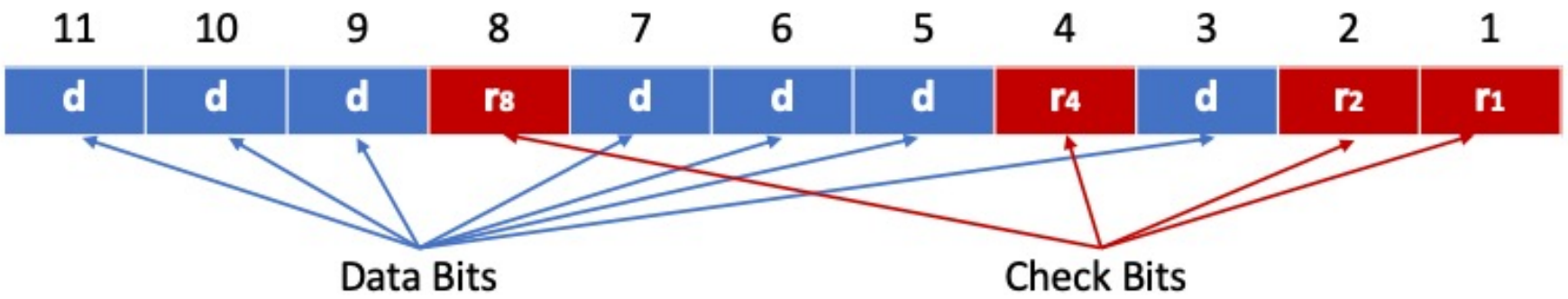
# Hamming Code

- Like before, the codeword is generated from the data
  - Each check bit in the codeword is **generated** based on the data
- The codeword is sent to receiver.
- At the receiver, the codeword is checked for errors using the scheme
  - Based on the check the code word is accepted or not.

# Hamming Code

- Each redundancy bit is the parity bit for a different combination of data bits

- Each data bit may be included in more than one parity check



# Hamming Code Calculation

- By writing a data bit's position in binary, we can tell which redundancy bit(s) checks this bit
  - The data bit in position **11** is checked by $r_1$ , $r_2$ , and $r_8$
    - because 11=1+2+8;
  - The data bit in position **6** is checked by $r_2$ and $r_4$
    - because 6=2+4;
  - ...
- Which data bits does each redundancy bit check?
  - $r_1$ checks data in **positions** whose **binary** representations have a: **1** in the **same binary position** as $r_1$ (1st position)
    - E.g., **1**, 1**1**, 10**1**, 11**1**, ...
  - $r_2$ checks data in positions whose binary representations have a: **1** in the **same binary position** as $r_2$ (2nd position)
    - E.g., **1**0, **1**1, 1**1**0, 1**1**1, ...
  - $r_4$ checks data in positions whose binary representations have a: **1** in the **same binary position** as $r_4$ (3th position)
    - E.g., 0**1**00, 0**1**01, 0**1**10, 0**1**11 ,...

# Hamming Code Calculation

- Steps involved:
  1. Create a codeword array

# Hamming Code Calculation

- Steps involved:
  1. Create a codeword array
  2. Number the positions in the codeword

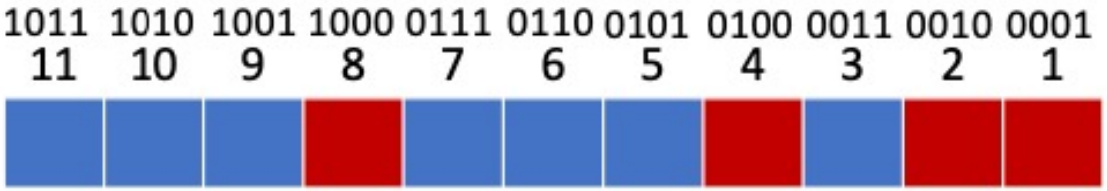

DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

# Hamming Code Calculation

- Steps involved:
  1. Create a codeword array
  2. Number the positions in the codeword
  3. Put in check bit placeholders

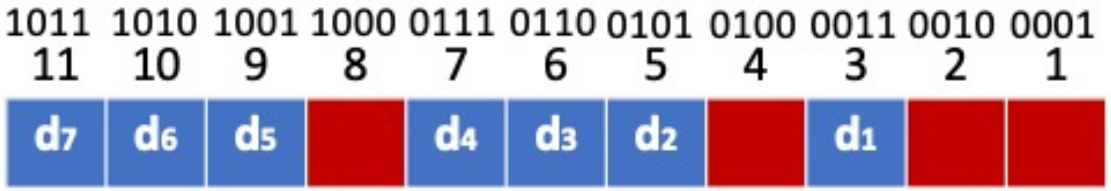
# Hamming Code Calculation

- Steps involved:
  1. Create a codeword array
  2. Number the positions in the codeword
  3. Put in check bit placeholders
  4. Turn numbers into binary representations



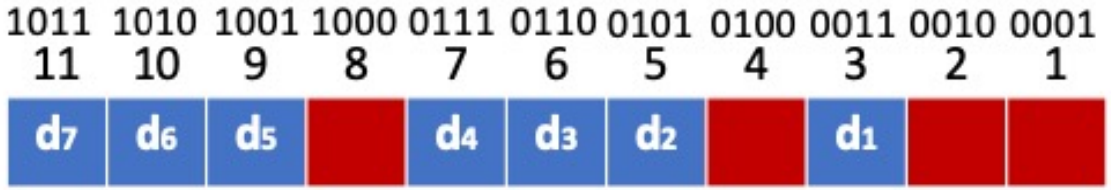
# Hamming Code Calculation

- Steps involved:
  1. Create a codeword array
  2. Number the positions in the codeword
  3. Put in check bit placeholders
  4. Turn numbers into binary representations
  5. Put data into the codeword



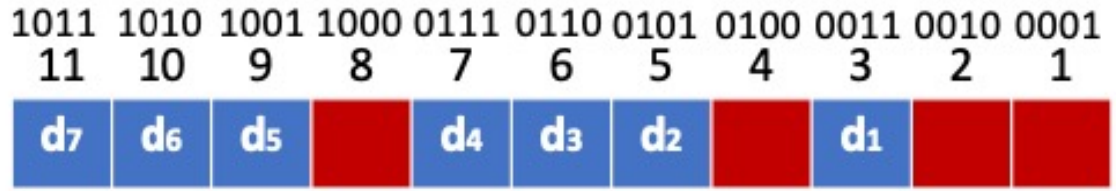
# Hamming Code Calculation

- Steps involved:
  1. Create a codeword array
  2. Number the positions in the codeword
  3. Put in check bit placeholders
  4. Turn numbers into binary representations
  5. Put data into the codeword
  6. Calculate the check bits individually



# Hamming Code Calculation

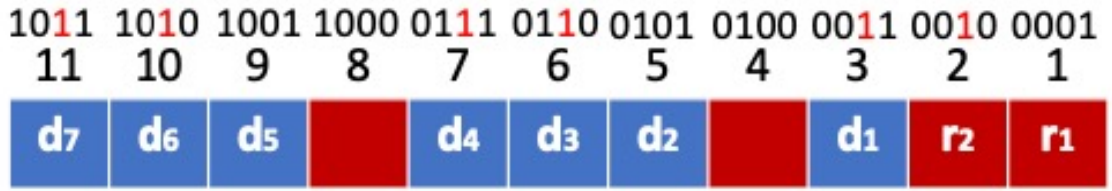
- Steps involved:
  1. Create a codeword array
  2. Number the positions in the codeword
  3. Put in check bit placeholders
  4. Turn numbers into binary representations
  5. Put data into the codeword
  6. Calculate the check bits individually



# Hamming Code Calculation

- Steps involved:
  1. Create a codeword array
  2. Number the positions in the codeword
  3. Put in check bit placeholders
  4. Turn numbers into binary representations
  5. Put data into the codeword
  6. Calculate the check bits individually

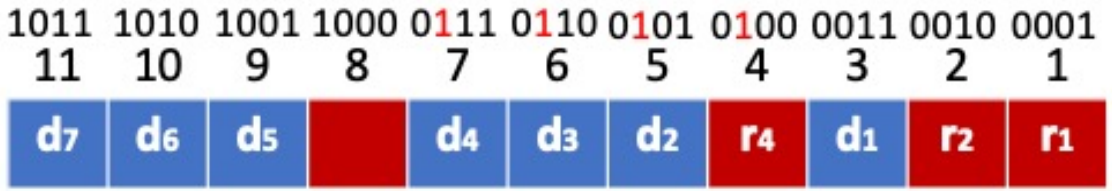
$r_2$ will take care of bits in positions 2, 6, 7, 10, and 11



# Hamming Code Calculation

- Steps involved:
  1. Create a codeword array
  2. Number the positions in the codeword
  3. Put in check bit placeholders
  4. Turn numbers into binary representations
  5. Put data into the codeword
  6. Calculate the check bits individually

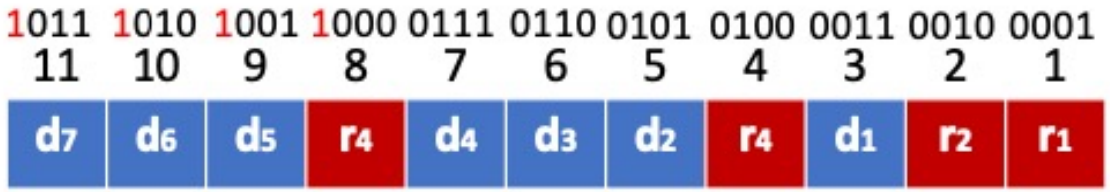
$r_4$ will take care of bits in positions 4, 5, 6, and 7



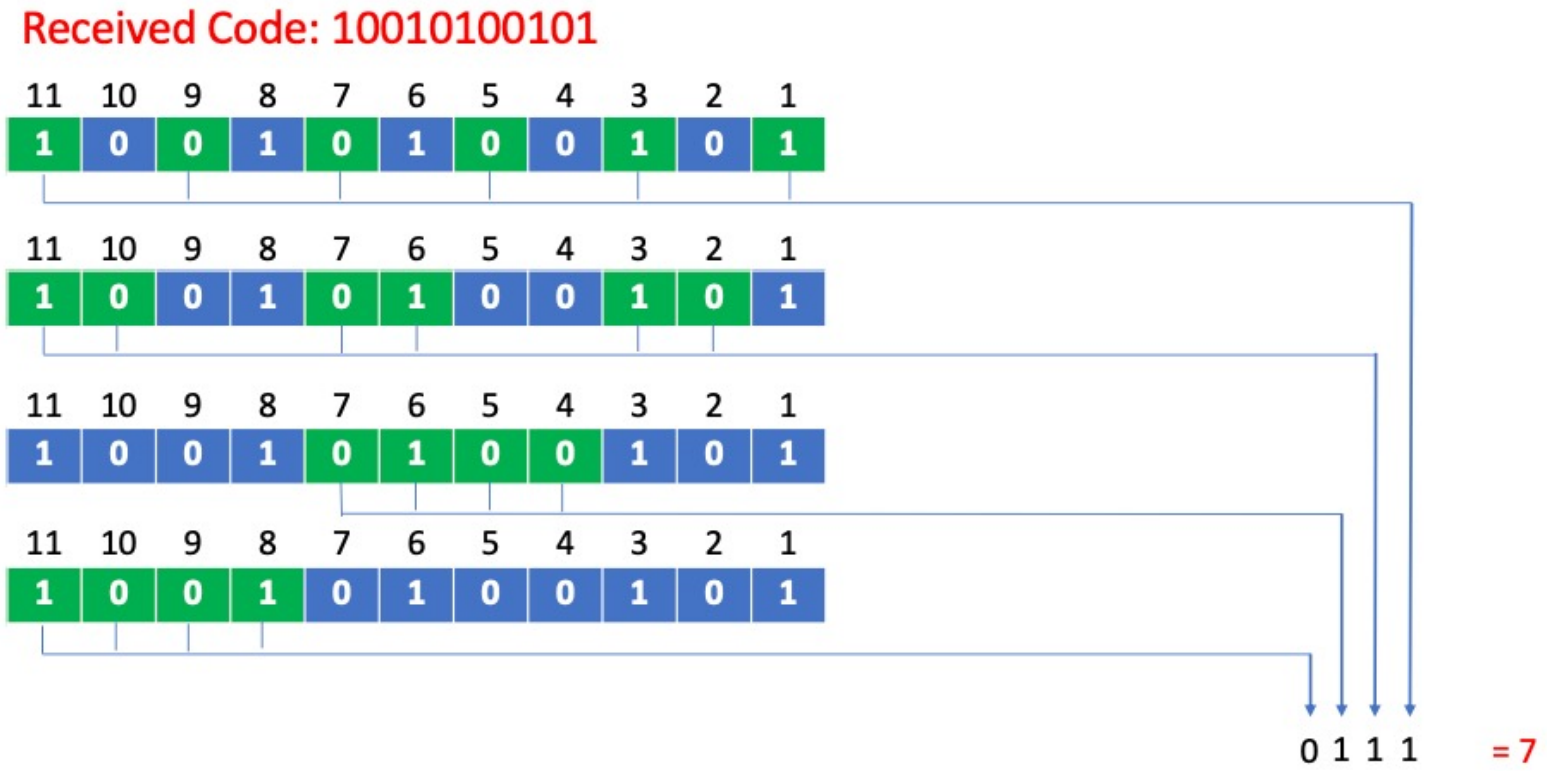
# Hamming Code Calculation

- Steps involved:
  1. Create a codeword array
  2. Number the positions in the codeword
  3. Put in check bit placeholders
  4. Turn numbers into binary representations
  5. Put data into the codeword
  6. Calculate the check bits individually

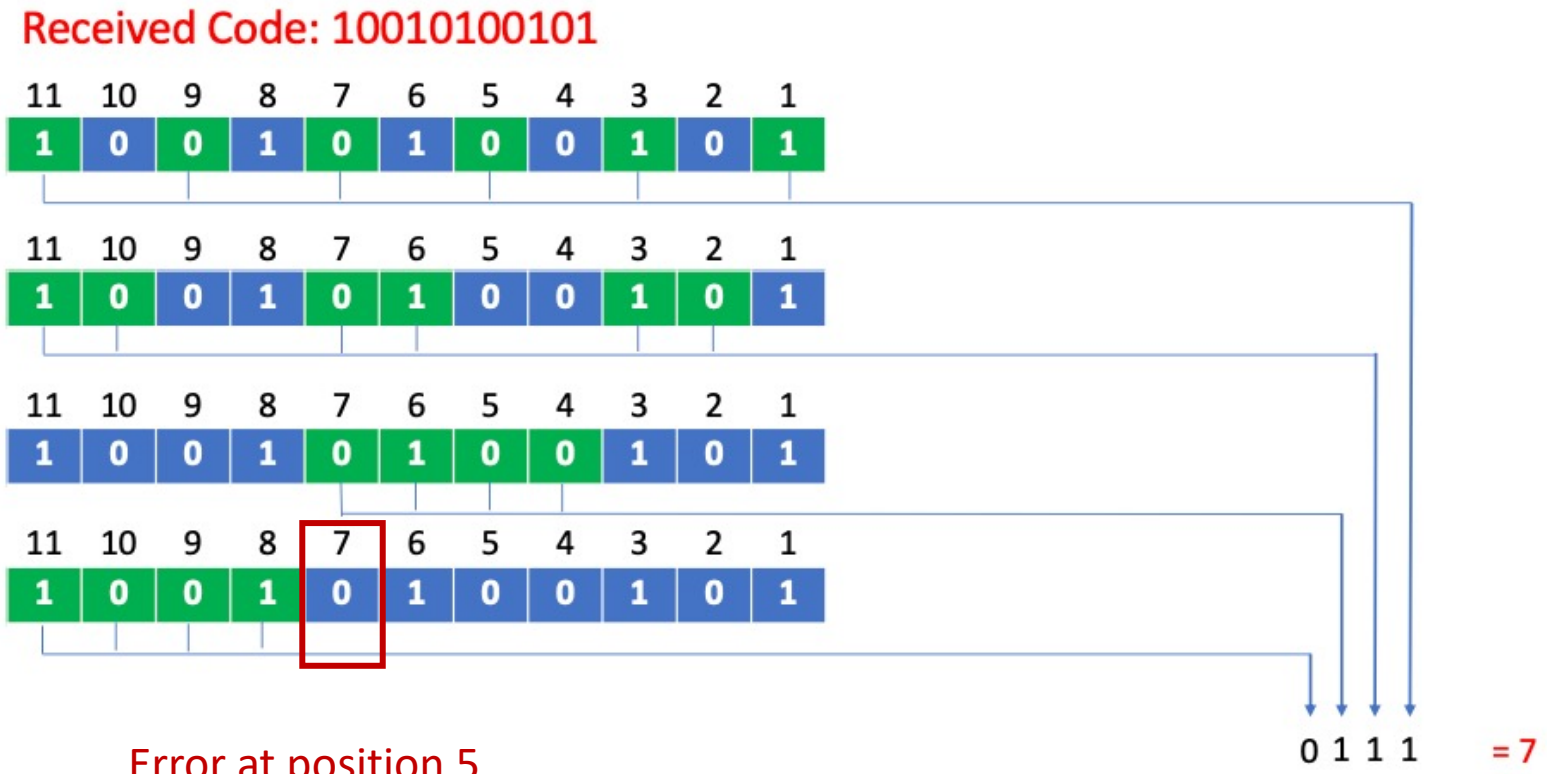
$r_8$ will take care of bits in positions 8, 9, 10, and 11



# Checking Received Hamming Code



# Checking Received Hamming Code




Error at position 5
(from the left)

# Hamming Code Checking

- This simple form of Hamming code can be used to provide some protection against burst errors
  - by transmitting 1st bit from every codeword to be transmitted, then 2nd bit from every one of these codewords, and so on...
  - In some cases, burst errors can be corrected
- However, for better error protection it is necessary to add more redundancy bits, and the number of extra redundancy bits rises dramatically from that required to correct single bit errors
- In cases where better **error detection** is required, more efficient methods exist (e.g., checksum or Cyclic Redundancy Check).
  - If these mechanisms detect errors, re-transmission is requested (where it is feasible)– more efficient than error correction

# Cyclic Redundancy Check (CRC)

- The cyclic redundancy check is in widespread use in the in the link layer
  - It is also known as a polynomial code
- Polynomial codes are based upon treating bit strings as representations of polynomials with coefficients of 0 and 1 only
- A *k*-bit frame is regarded as the coefficient list for a polynomial with *k*
- Such a polynomial is said to be of degree $k - 1$. The high-order (leftmost) bit is the coefficient of $x^{k-1}$, the next bit is the coefficient of $x^{k-2}$, and so on

# Cyclic Redundancy Check (CRC)

- For example, 110001 has 6 bits and thus represents a six-term polynomial with coefficients.
  - The polynomial code for 110001 is $1x^5 + 1x^4 + 0x^3 + 0x^2 + 0x^1 + 1x^0$
- When the polynomial code method is employed, the sender and receiver must agree upon a **generator polynomial**, $G(x)$, in advance
- Both the high-order (here $x^5$) and low-order (here $x^0$) bits of the generator **must be 1**

# Cyclic Redundancy Check (CRC)

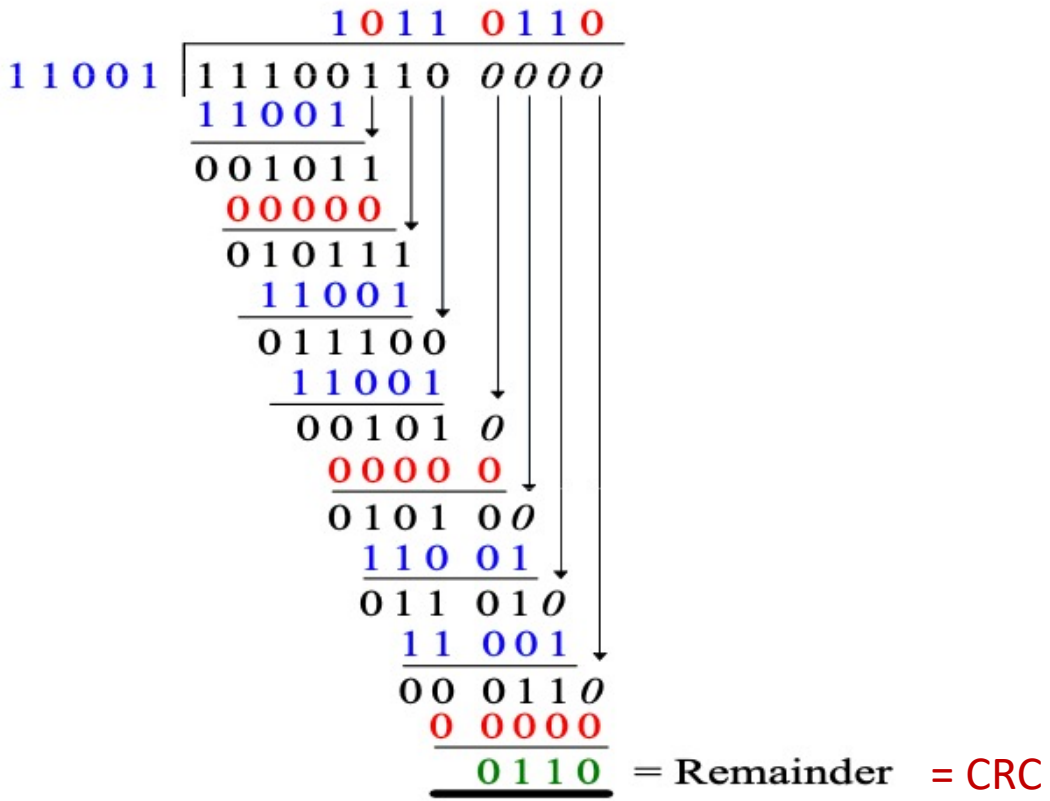
- Let M be the frame contents that are to be protected, usually from everything except the flags and the bit stuffing. Assume it to be k bits long.
- Let the CRC be n bits long
- Let G, the Generator Polynomial, be n+1 bits long
- Divide M (frame) with n zeros appended to it, by G (generator) and the remainder is the CRC
- This CRC, which is n bits long, is sent along with the frame to the receiver

# Cyclic Redundancy Check (CRC)

- You transmit the bit sequence, M, appended by the CRC, and any other unprotected bits like the flags.
- The receiver takes the unprotected bits off, and then **divides** the k+n bits of the bit stream by the Generator G.
- If the remainder is zero then there are no detected errors. A non-zero remainder will indicate that errors have been detected.

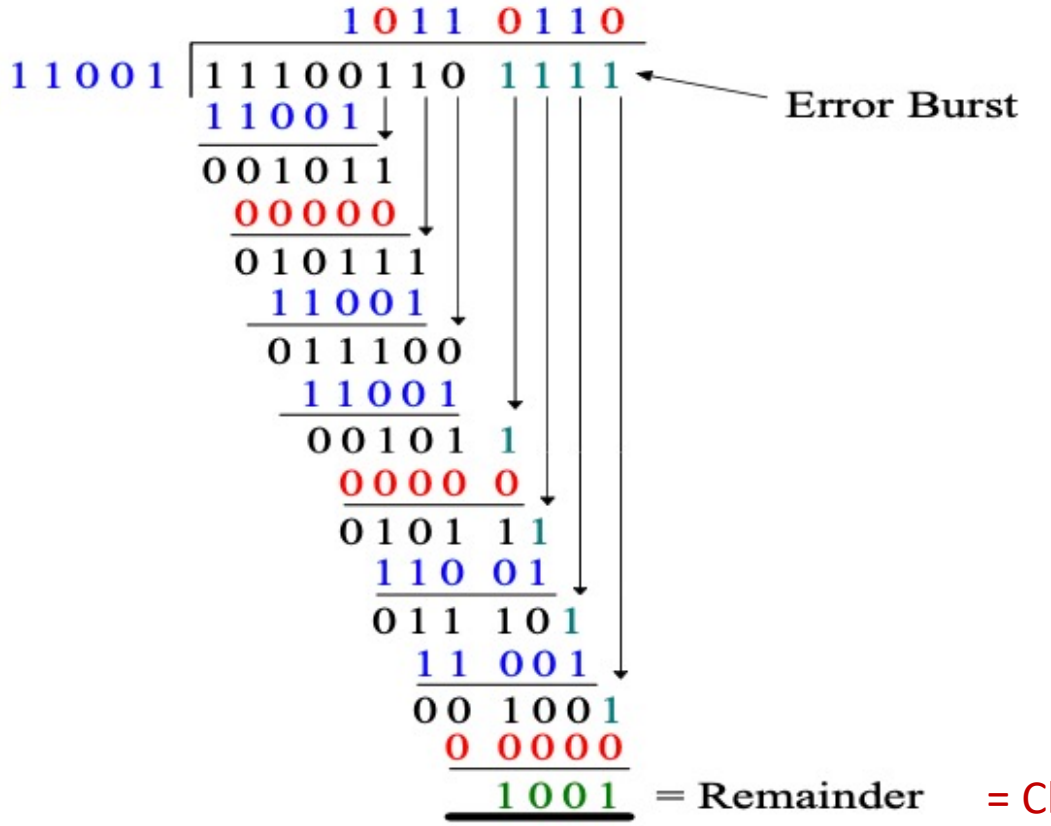
# Example 1 – At Sender

**Note:**
**M**: 11100110
(k=8)
**M with zeros** :
11100110 *0000*
(n=4)
**G**: 11001
**CRC**: 0110
**Transmitted bits**:
111001100110




For more information about how to divide two binary numbers see this link: https://www.wikihow.com/Divide-Binary-Numbers

# Example 1 – At Receiver




= Checksum Value

Remainder ≠ 0, ⇒ Error Detected

# CRC Operation

- Only an error pattern that is identical, or has a factor identical to the generator polynomial, will produce the same CRC bits remaining undetected
- For this reason, a polynomial which is prime, in the modulo-2 sense, is chosen as the generator
- The standard way of representing a generator polynomial is to show those bit positions in the number that are binary 1 as powers of X.
  - A standard 16-bit CRC generator polynomials is:
    - CRC-16 = $X^{16}+X^{15}+X^2+1$
      = 1 1000 0000 0000 0101

# Summary

- Single Parity Check
  - Can detect 1 single bit error. Can correct 0 bit errors
- 2-Dimensional Parity Check
  - Can detect and correct 1 single bit error
  - But requires many extra check bits
- General Parity Check
  - Min Distance D => detect ≤ D-1 bit errors
    => correct < D/2 bit errors
- Hamming Code
  - Can detect and correct 1 bit error
  - Uses the minimum number of extra check bits
- CRC:
  - Can detect any error pattern that is not identical or a factor of the generator. Can correct 0 bit errors
  - Requires the sender and receiver to know the generator. The sender must add the remainder (which has the same size as generator - 1)