Data Link Layer, Part 1

CSC 343·643

WAKE FOREST
UNIVERSITY

Department of Computer Science

Fall 2013

Data Link Layer

Provide *reliable* and *efficient* communications between to adjacent machines (physically connected via a channel)

- Providing well-defined service interface to the network layer
- Design issues
  - Determining how bits are grouped into *frames*
  - Error detection (possibly correction)
  - Line discipline
  - Flow control (between adjacent machines)

Were any of these items provided by the physical layer?

Services Provided to the Network Layer

Three possible services provided

1. **Unacknowledged connectionless service**
   - Source sends independent frames to destination
   - Destination machine does not acknowledge receipt
   - No connection is established or released
   - If frame lost, then no attempt to recover/retransmit

   What types of traffic can benefit from this service?

2. **Acknowledged connectionless service**
   - No connection is used
   - Each frame is acknowledged by destination

   Is there a need for acknowledgements at this layer?

- Acknowledgements (ACKs) are **not** a requirement...
  - *ACKs can be handled in higher layers (network and/or transport)*
    Any disadvantage to higher layer ACKs

3. **Acknowledged connection-oriented service**
   - Source and destination establish a connection
   - Each frame is numbered
   - Service guarantees each frame received
   - Each frame is received only once and in the right order
   - *Reliable bit stream*

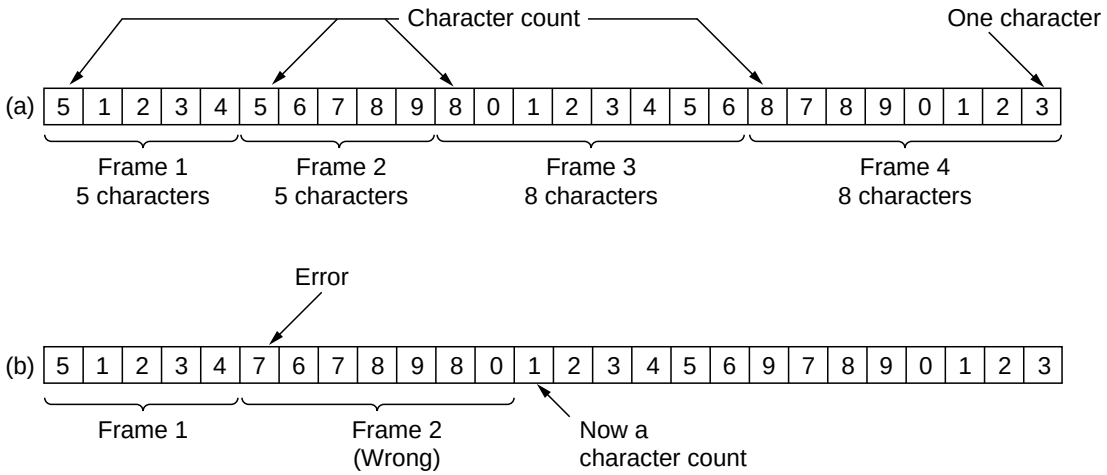
Framing

Physical layer only delivers a *raw* bit stream (**no guarantees**)

- Data link layer must detect and (if possible) correct any errors
- Data link layer breaks the bit stream into *frames*
  - Data link layer calculates the *checksum* per frame
  - Frame and checksum are transmitted
- Before discussing checksums, consider four framing methods
  1. Character count
  2. Starting and ending characters, with character stuffing
  3. Starting and ending flags, with bit stuffing
  4. Physical layer coding violations

Character Count Framing

- Field in the header specifies the number of characters in the frame
  - Destination data link layer then knows the number of characters, thus the end of the frame
- Problem occurs when the count is lost or in error



- Very difficult to resynchronize after count loss/error

Start/End Character Framing

Use *special* characters to delineate frame

- Each frame starts with the ASCII sequence DLE (Data Link Escape) STX (Start of TeXt)
- Each frame ends with the sequence DLE ETX (End of TeXt)

DLE	STX	P	L	U	F	L	I	V	E	S	DLE	ETX

- If the destination losses track of boundary, it just needs to look for the next boundary

  What is a possible problem with the starting/ending sequences?

- It is possible that the start/stop sequence may appear in the actual data stream (consider binary data)
- One solution is *character stuffing*
  - Insert an extra DLE before any *accidental* DLE
  - Receiver must remove the additional DLE

Original data	A | DLE | B
Stuffed framed data	DLE | STX | A | DLE | DLE | B | DLE | ETX

- Disadvantage - Closely tied to 8 bit characters

Start/End Flag Framing

Use *flag* byte to delineate frame

- Each frame begins/ends with 01111110
- Bit stuffing is required
  - When the sender sees five consecutive 1's, it inserts (stuffs) a 0
  - Receiver must remove stuffed bit
  - Similar to character stuffing

Original	01101111111111111110010
Stuffed framed	01111110011011111011111011111010010011111110

What does 0111110 look like stuffed?

Physical Layer and Redundant Framing

Digital encoding that provide synchronization can be used for framing

What are examples of such encoding methods?

- If a 0 is represented by a low-to-high state and 1 by a high-to-low
  - Then high-to-high and low-to-low is not used
  - Use these *non-transitions* as boundaries
  - Part of the 802 LAN standards

Most data link protocols employ redundant framing

- Use character counts and frame flags
- Then receiver knows the *correct* location of the end flag

Error Detection and Correction

How do we know if the frame data is correct?

- Two types of errors
  - Single-bit or burst *What is more common?*

- Two strategies for error handling
  1. **Error detection and correction**
     - Include redundant information with a frame to detect and correct error
  2. **Error detection and retransmission**
     - Include redundant information with a frame to detect error
     - If error occurred request retransmission

     What are the advantages and disadvantages?

Error Detection

- A simple idea is to send each bit twice (high redundancy)
  - Doubles the transmission time
  - Too inefficient; therefore, need to reduce the redundancy
- Four types of redundancy checks
  1. Vertical Redundancy Check (VRC or parity check)
  2. Longitudinal Redundancy Check (LRC)
  3. Cyclical Redundancy Check (CRC)
  4. Checksum
- VRC, LRC, and CRC are typically in the physical layer

Vertical Redundancy Check (VRC)

Also called a *parity check*

- An extra bit (parity-bit) added to the end of a message
- The extra bit is added to cause the number of 1 bits to be even (or odd)

Data	Even Parity	Odd Parity
10110101	101101011	101101010

- Performance
  - Can detect single-bit errors
  - Can detect burst-errors, with **odd lengths**

Data	Even Parity	Error Frame
10110101	101101011	100111011

Hamming Distance

- Assume a message has m bits and r redundant (check) bits
  - The total length is $n = m + r$
  - The n bit sequence of bits is the **codeword**
  - On the previous slide $m = 7$, $r = 1$, and $n = 8$ bits
- Given any two *valid* codewords, determine how bits differ
  - XOR the codewords and count the number of 1's
  - The minimum number is called the **Hamming distance**
  - If two codewords have a distance of d, then it will require d errors to convert one into the other $\Rightarrow$ **undetected error**

What is the Hamming distance of VRC?

Longitudinal Redundancy Check (LRC)

- Message is arranged as a table (instead of a single row)
- Assume we are sending a 32 bit message

11100111 11011101 00111001 10101001

- Organize as a table with 4 rows and 8 columns

	11100111
	11011101
	00111001
	10101001
LRC $\rightarrow$	10101010

- LRC (column parity bits) is calculated based on the columns, then added to the end of the message

11100111 11011101 00111001 10101001 10101010

LRC Performance

- Suppose the following message is sent

  10101001 00111001 11011101 11100111 10101010

- Suppose it is received in error as follows

  10100011 10001001 11011101 11100111 10101010

- LRC detects the error

  10100011 10001001 11011101 11100111 10101010

- Can handle burst-errors

  What error pattern can cause a problem?

Checksum

Used by the transport layer (Internet checksum)

- Checksum generator (sender side)
  - Checksum generator divides the data into equal n bit segments
  - Segments are added together using *one's complement* arithmetic (segments are added modulo $2^n - 1$)

$$(s_0 + s_1 + ... + s_{n-1}) \text{mod} (2^n - 1) = x$$

  - Result is complemented then added to the end of the message
- Checksum checker (receiver side)
  - Divide data into n bit segments
  - Add segments together and complement result
  - Add the checksum field, result is zero if *no* errors

- Example - suppose `10101001 00111001` is to be sent using a checksum of 8 bits

	10101001
+	00111001
	11100010
checksum	00011101

  - Data sent is `10101001 00111001 00011101`

- Performance
  - Detects all errors involving an odd number of bits
  - If one or more bits of a segment are damaged and the corresponding bit(s) of opposite value in a second segment are also damaged, the receiver will not detect

Why is a checksum used by the transport layer, while stronger error checking (CRC) is performed in the link layer?

```
/* Compute Internet checksum for count bytes, beginning at
*  location addr
*/
unsigned short checksum(unsigned short *addr, int count)
{
   register long sum = 0;
   while(count > 1)
   {
      sum += *addr++;
      count -= 2;
   }
   /* Add any left-over byte */
   if(count > 0)
      sum += *addr;
   /* Fold 32 bit sum into 16 bits */
   while(sum >> 16)
      sum = (sum & 0xffff) + (sum >> 16);
   return -sum;
}
```

Cyclic Redundancy Check (CRC)

- Represent the message as a n degree polynomial
  - Where the highest-order term is x^n
  - Value of each bit is the coefficient of each term, either 1 or 0

$$[10011010] \Rightarrow 1 \cdot x^7 + 0 \cdot x^6 + 0 \cdot x^5 + 1 \cdot x^4 + 1 \cdot x^3 + 0 \cdot x^2 + 1 \cdot x^1 + 0 \cdot x^0$$
$$= x^7 + x^4 + x^3 + x$$

- Sender and receiver must agree on a *divisor*, $c(x)$
  - Another polynomial of degree k, for example $x^3 + x^2 + 1$
- Sender sends message $m(x) \Rightarrow n + 1$ data and k redundant bits
  - Create $m(x)$ so it is completely divisible by $c(x)$

CRC Rules for Polynomial Math

Assume $b(x)$ and $c(x)$ are polynomials

- $b(x)$ can be divided by $c(x)$ if $b(x)$ has a higher degree than $c(x)$
- $b(x)$ can be divided once by $c(x)$ if $b(x)$ and $c(x)$ have the same degree
- Remainder of $\frac{b(x)}{c(x)}$ is obtained from subtracting $c(x)$ from $b(x)$
- To subtract $c(x)$ from $b(x)$, XOR matching coefficients

Examples

- $x^3 + 1$ can be divided by $x^3 + x^2 + 1$
  - The remainder would be $0 \cdot x^3 + 1 \cdot x^2 + 0 \cdot x^1 + 0 \cdot x^0$
- Easier to see using bit patterns, [1001] divided by [1101] $\Rightarrow$ [0100]

Rules for Generating CRC Message

- Want to create a message that consists of original data $m(x)$ and redundant bits, that is exactly divisible by $c(x)$
- Rules for message generation
  1. Multiply $m(x)$ by x^k (add k zeroes to end) call this $t(x)$
  2. Divide $t(x)$ by $c(x)$ and find the remainder
     - Remainder is called the **CRC**
  3. Subtract remainder from $t(x)$ and transmit
- Rules for error detection
  1. Divide received message by $c(x)$
  2. If remainder is zero, then no errors, otherwise error occured

In Other Words...

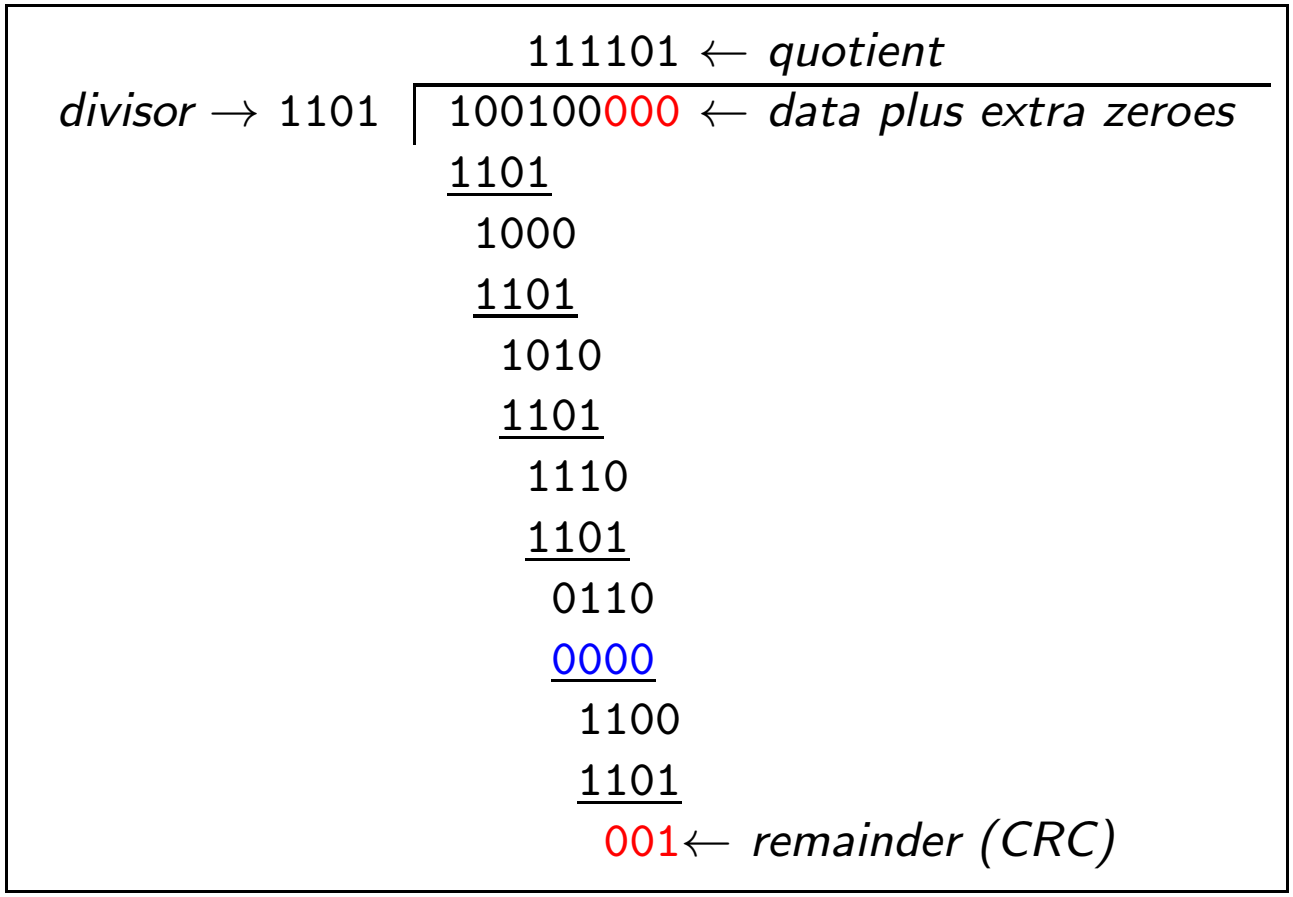
- View the data bits as coefficients to a polynomial, $m(x)$

$$[10011010] \Rightarrow 1 \cdot x^7 + 0 \cdot x^6 + 0 \cdot x^5 + 1 \cdot x^4 + 1 \cdot x^3 + 0 \cdot x^2 + 1 \cdot x^1 + 0 \cdot x^0$$
$$= x^7 + x^4 + x^3 + x$$

- Have the sender and receiver agree to a generator, $g(x)$
- Given a message $m(x)$ make it divisible by $g(x)$ and send
- Once received, make certain message is divisible by $g(x)$, if so remove original message

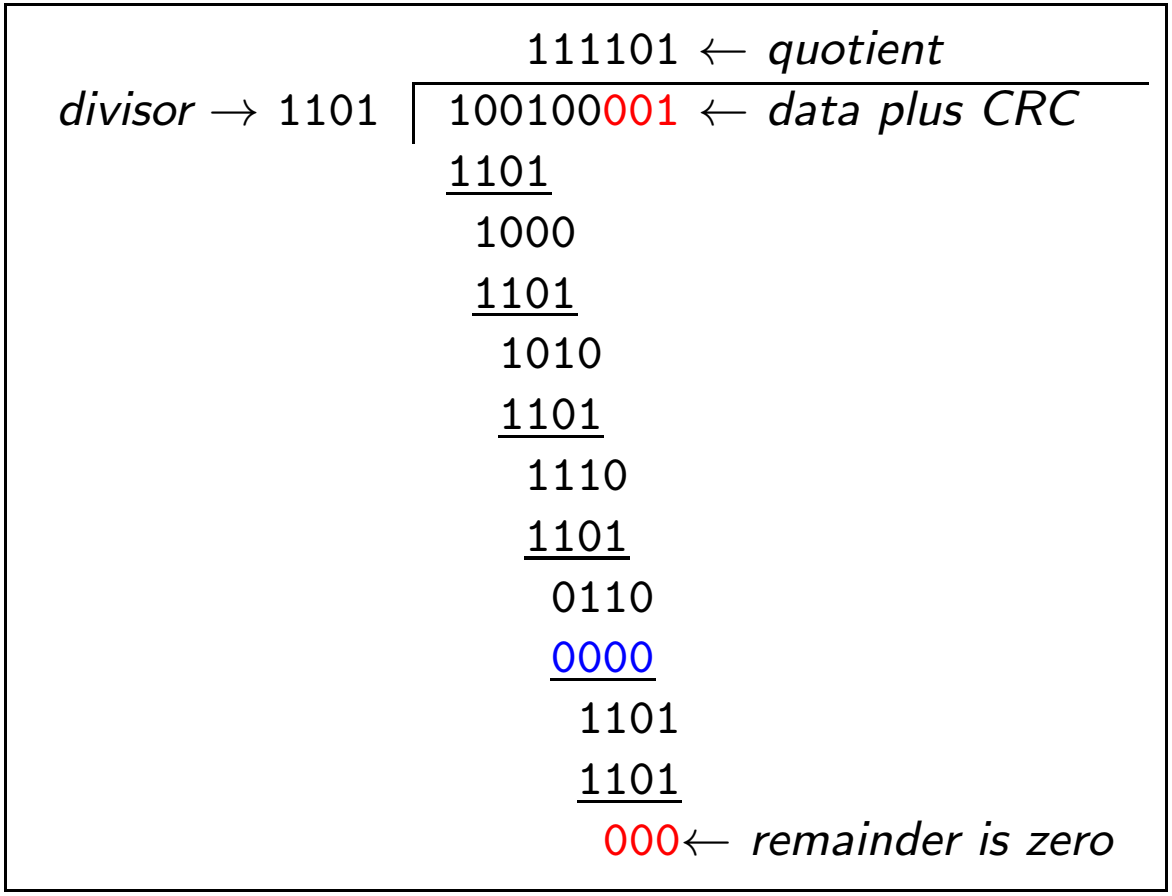
CRC Example

- Given the message 100100 and the divisor 1101, the CRC is



What is sent?

- To check, use the received message and divisor



  - Remainder should be zero for *no* error

CRC Performance

- Standard polynomials include

CRC-12	$x^{12} + x^{11} + x^3 + x + 1$
CRC-16	$x^{16} + x^{15} + x^2 + 1$
CRC-ITU-T	$x^{16} + x^{12} + x^5 + 1$

What is the divisor polynomial on the previous slide?

- CRC performance
  - Can detect all burst errors that affect an odd number of bits
  - Can detect all burst errors of length less than or equal to the degree of the polynomial
  - Can detect with a high probability burst errors of length greater than the degree of the polynomial

CRC Performance

- Standard polynomials include

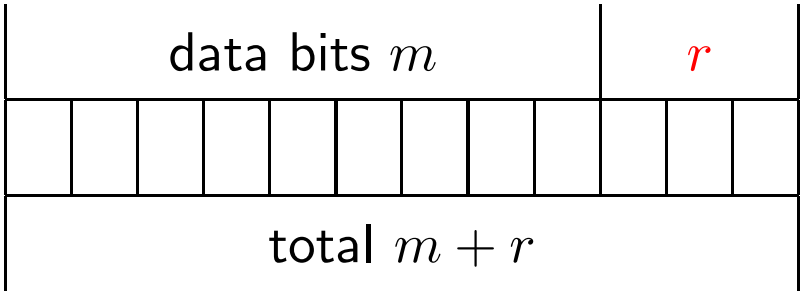
CRC-12	$x^{12} + x^{11} + x^3 + x + 1$
CRC-16	$x^{16} + x^{15} + x^2 + 1$
CRC-ITU-T	$x^{16} + x^{12} + x^5 + 1$

What is the divisor polynomial on the previous slide?

- CRC performance
  - Can detect all burst errors that affect an odd number of bits
  - Can detect all burst errors of length less than or equal to the degree of the polynomial
  - Can detect with a high probability burst errors of length greater than the degree of the polynomial

Redundancy Bits

To calculate the number of redundancy bits r required to correct a given number of data bits m, must find a relationship between m and r



- $m + r$ bits $\Rightarrow$ r must indicate at least $m + r + 1$ different states
  - *Each state identifies a specific bit*
  - One of the states must indicate *no error*
- Therefore, if $m + r + 1$ states are needed and r can identify 2^r states

$$2^r \geq m + r + 1$$

- The value of r can be determined by using the value of m
- If $m = 7$ then the smallest r is 4

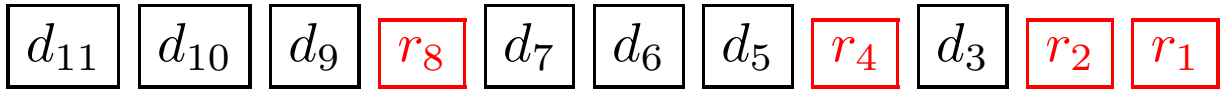
$$2^4 \geq 7 + 4 + 1$$

Data bits (m)	Redundancy bits (r)	Total bits ($m + r$)
1	2	3
2	3	5
3	3	6
4	3	7
5	4	9
6	4	10
7	4	11

Hamming Code

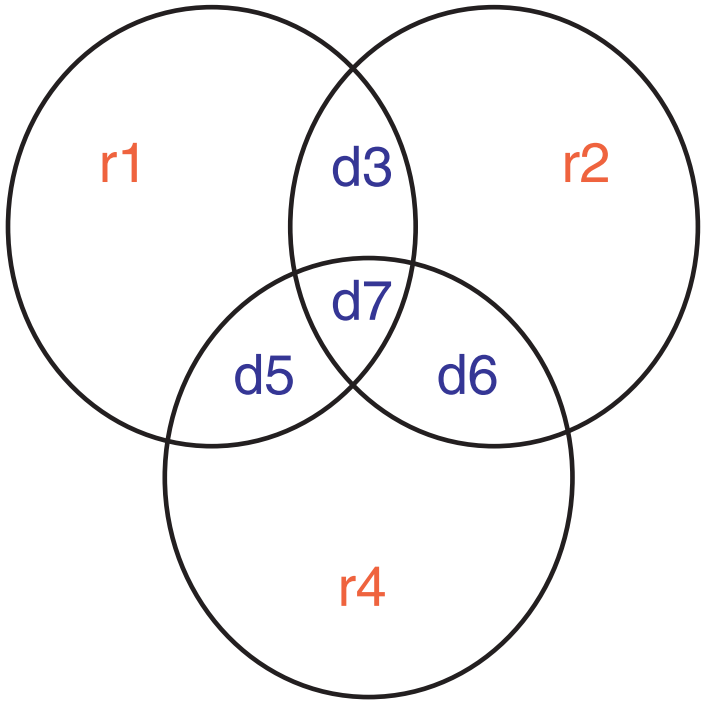
We know the number of redundancy bits required to find all possible error states. How do we use this information to discover which state?

- Positioning the redundancy bits
  - Redundancy bits are added among the data bits



  - Redundancy bits are r_1, r_2, r_4, and r_8
  - Each r bit is the VRC for a combination of d bits

r_1	bits 1, 3, 5, 7, 9, 11
r_2	bits 2, 3, 6, 7, 10, 11
r_4	bits 4, 5, 6, 7
r_8	bits 8, 9, 10, 11



What is the pattern for r_i ?

Hamming Code

- For the VRC - Data bits are included at lease twice, while redundancy bits are included once
- What is the pattern? Look at the binary value of each bit *position*
  - r_1 is calculated using all bit positions whose binary representation includes a 1 in the right-most position

1011		1001		0111		0101		0011		0001
d_{11}	d_{10}	d_9	r_8	d_7	d_6	d_5	r_4	d_3	r_2	r_1

  - r_2 is calculated using all bit positions whose binary representation includes a 1 in the second right-most position

1011	1010			0111	0110			0011	0010	
d_{11}	d_{10}	d_9	r_8	d_7	d_6	d_5	r_4	d_3	r_2	r_1

  - This repeats for the remaining r bits

- Once the the message is at the receiver
  - Calculate new VRC's based on the message (use same method)
  - Assemble *new* parity bits as binary number $[r_8, r_4, r_2, r_1]$
  - Number identifies the location of the error
- Example, assume the data is [1001101]

d_{11}	d_{10}	d_9	r_8	d_7	d_6	d_5	r_4	d_3	r_2	r_1
1	0	0	?	1	1	0	?	1	?	?

d_{11}	d_{10}	d_9	r_8	d_7	d_6	d_5	r_4	d_3	r_2	r_1
1	0	0	?	1	1	0	?	1	?	1

d_{11}	d_{10}	d_9	r_8	d_7	d_6	d_5	r_4	d_3	r_2	r_1
1	0	0	?	1	1	0	?	1	0	1

d_{11}	d_{10}	d_9	r_8	d_7	d_6	d_5	r_4	d_3	r_2	r_1
1	0	0	?	1	1	0	0	1	0	1

d_{11}	d_{10}	d_9	r_8	d_7	d_6	d_5	r_4	d_3	r_2	r_1
1	0	0	1	1	1	0	0	1	0	1

– The resulting message [10011100101] is sent

– Assume the received message is [10010100101], determine which bit is in error

d_{11}	d_{10}	d_9	d_8	d_7	d_6	d_5	d_4	d_3	d_2	d_1
1	0	0	1	0	1	0	0	1	0	1

r_8	r_4	r_2	r_1
?	?	?	?

d_{11}	d_{10}	d_9	d_8	d_7	d_6	d_5	d_4	d_3	d_2	d_1
1	0	0	1	0	1	0	0	1	0	1

r_8	r_4	r_2	r_1
?	?	?	1

d_{11}	d_{10}	d_9	d_8	d_7	d_6	d_5	d_4	d_3	d_2	d_1
1	0	0	1	0	1	0	0	1	0	1

r_8	r_4	r_2	r_1
?	?	1	1

d_{11}	d_{10}	d_9	d_8	d_7	d_6	d_5	d_4	d_3	d_2	d_1
1	0	0	1	0	1	0	0	1	0	1

r_8	r_4	r_2	r_1
?	1	1	1

d_{11}	d_{10}	d_9	d_8	d_7	d_6	d_5	d_4	d_3	d_2	d_1
1	0	0	1	0	1	0	0	1	0	1

r_8	r_4	r_2	r_1
0	1	1	1

- The resulting code is [0111] = 7
- Invert the 7th bit and the message is correct

Is there a pattern for the position of r bits?

Can you make Hamming codes handle multiple errors?