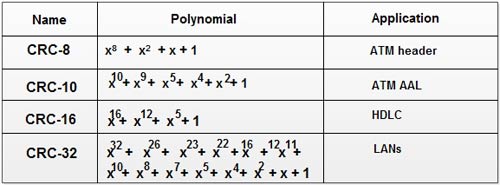
# Experiment 2: Error Detection using CRC-CCITT

**Aim:** To apply CRC (CCITT Polynomial) for error detection

**Objective:** After carrying out this experiment, students will be able to:

* Apply CRC CCITT to develop codes for error detection
* Analyse how this CRC is able to detect bit errors irrespective of their length and position in the data

**Problem statement:** You are required to write a program that uses CRC to detect burst errors in transmitted data. Initially, write the program using the CRC example you studied in class. Your final program should ask the user to input data and choose a generator polynomial from the list given in the figure below. Your program is required to calculate the checksum and the transmitted data. Subsequently, the user enters the received data. Applying the same generator polynomial on the received data should result in a remainder of 0.



**Analysis:** While analyzing your program, you are required to address the following points:

* How is this method different from 2D parity scheme that you have implemented previously?
* What are the limitations of this method of error detection?

**MARKS DISTRIBUTION**

|  |  |  |
| --- | --- | --- |
| **Component** | **Maximum Marks** | **Marks Obtained** |
| Preparation of Document | 7 |  |
| Results | 7 |  |
| Viva | 6 |  |
| **Total** | **20** |  |

Submitted by: Subhendu Maji

Register No: 18ETCS002121

1. Algorithm/Flowchart

**checkReceivedData**(char\* gp)

STEP 1: Start

STEP 2: receivedData from user

STEP 3: temp receivedData[:len(gp)]

STEP 4: \_crc(receivedData, gp, temp)

STEP 5: if temp is 0, display pass message, else display fail message

STEP 6: Stop

**crc**(char\* src, char\* gp)

STEP 1: Start

STEP 2: dataLen len(src) + len(gp) – 1

STEP 3: data empty list

STEP 4: appendZeros(src, data, dataLen)

STEP 5: temp = data[:len(gp)]

STEP 6: \_crc(data, gp, temp)

STEP 7: display results

\_**crc**(char\* data, char\* poly, char\* temp)

STEP 1: Start

STEP 2: for i=0 to len(data) – len(poly) + 1, do

2.1: if temp[0] is ‘1’, then xor(temp, poly, temp)

2.2: leftShift(temp)

2.3: temp[len(poly)-1] = data[i+len(poly)]

STEP 3: Stop

**xor**(char\* a, char\* b, char\* temp)

STEP 1: Start

STEP 2: for i=0 to len(b), do

2.1: x a[i] as int

2.2: y b[i] as int

2.3: temp[i] (x ^ y) as char

STEP 3: Stop

**leftShift**(char\* input)

STEP 1: Start

STEP 2: for i=0 to len(input), do

2.1: input[i-1] = input[i]

STEP 3: input[end] = 0

STEP 4: Stop

**appendZeros**(char\* src, char\* dest, int dataLen)

STEP 1: Start

STEP 2: dest[:len(src)] = src

STEP 3: for i=len(src) to dataLen, do

3.1: dest[i] = ‘0’

STEP 4: Stop

1. Program

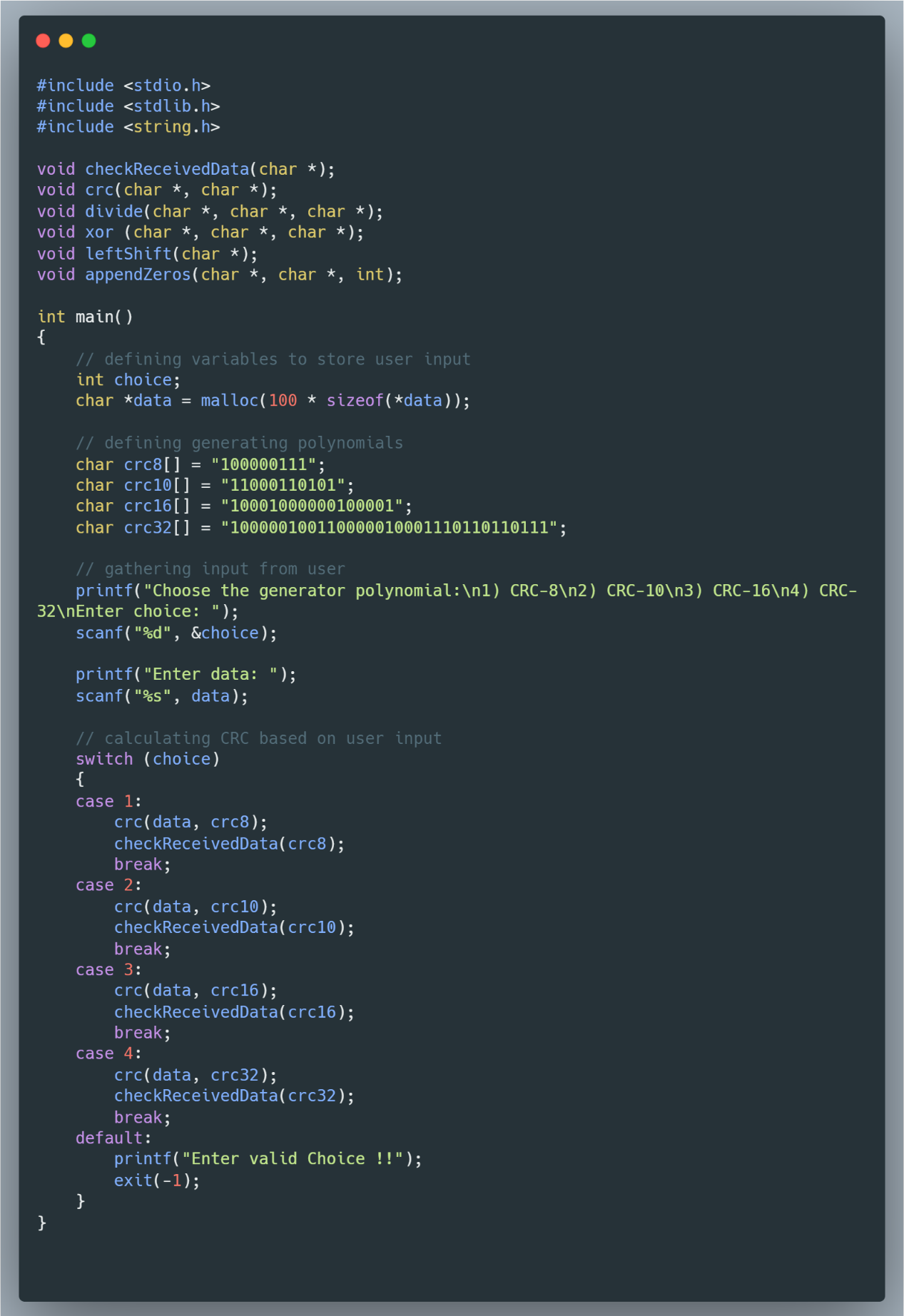


Figure Driver Program of CRC



Figure Snippet of Divide function

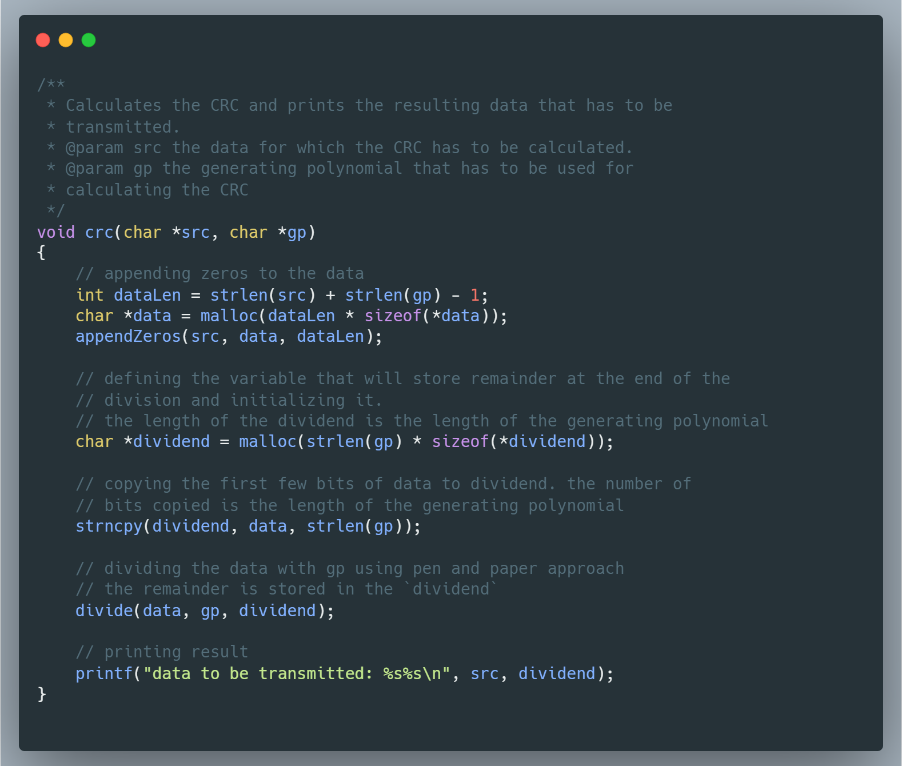


Figure snippet of crc function

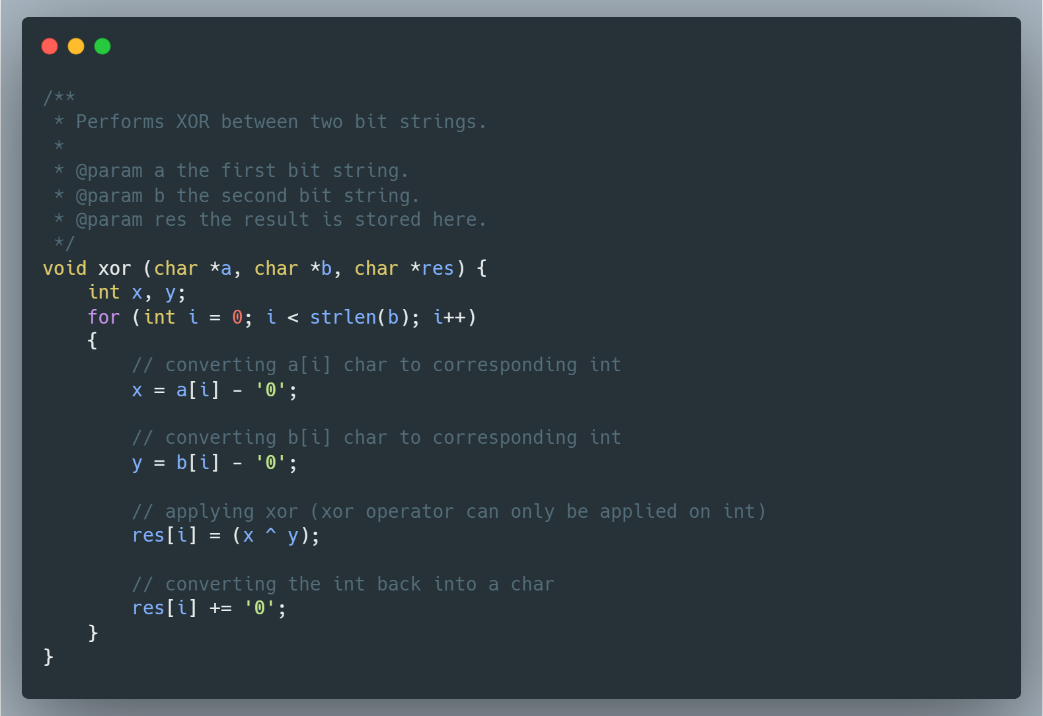


Figure snippet of crc function



Figure snippet of leftShift and appendZeros function



Figure snippet of checkReceivedData function

1. Results



Figure makefile

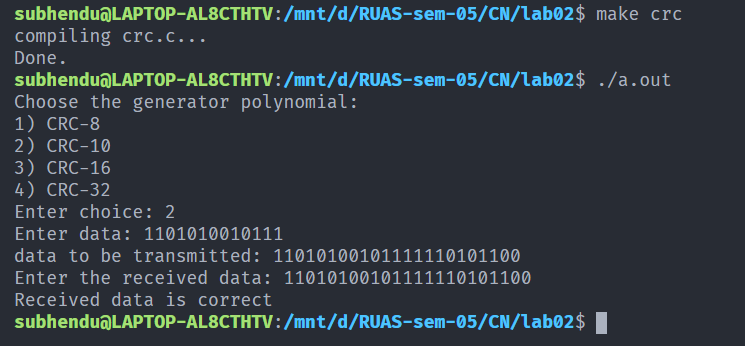


Figure CRC execution

1. Analysis and Discussions

Cyclic codes are special linear block codes wherein if a code word is cyclically shifted, the result is another code word. A cyclic redundancy check (CRC) is an error detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data.

In this method, the data blocks get a check value appended to it. This check value is based on the remainder of a polynomial division of the block’s contents. When the transmitted block is received, the calculation is repeated. If the result of the calculation is 0, then the received data is the same as the original data. If it is not 0, it indicates that the received data is corrupted in some form.

Cyclic redundancy check is a very good technique to detect single bit and burst errors. Since the method involves polynomial division of binary numbers which can be implemented using XOR and bit shift operations, this method is very fast when implemented in hardware.

1. Conclusions

Cyclic redundancy check is a technique that uses cyclic codes to detect single bit and burst errors. Due to its mathematical formulation, the technique is extremely fast when implemented in hardware. In this experiment, the method to implement cyclic redundancy check in C was learned.

1. Comments
   1. Limitations of the experiment

Cyclic redundancy check is not suitable for protection against intentional alteration of data, and overflow of data is a possibility in cyclic redundancy check.

* 1. Learning

The way to implement cyclic redundancy check in software was learned. Furthermore, this implementation was learned using C language.