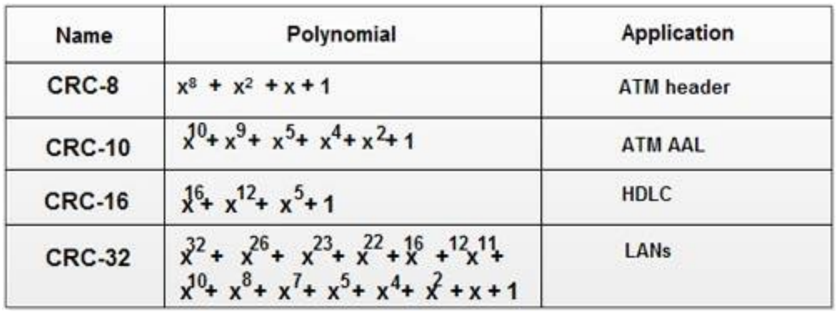
Experiment 2: Error Detection using CRC-CCITT

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.



Name	Polynomial	Application
CRC-8	$x^8 + x^2 + x + 1$	ATM header
CRC-10	$x^{10} + x^9 + x^5 + x^4 + x^2 + 1$	ATM AAL
CRC-16	$x^{16} + x^{12} + x^5 + 1$	HDLC
CRC-32	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$	LANs

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

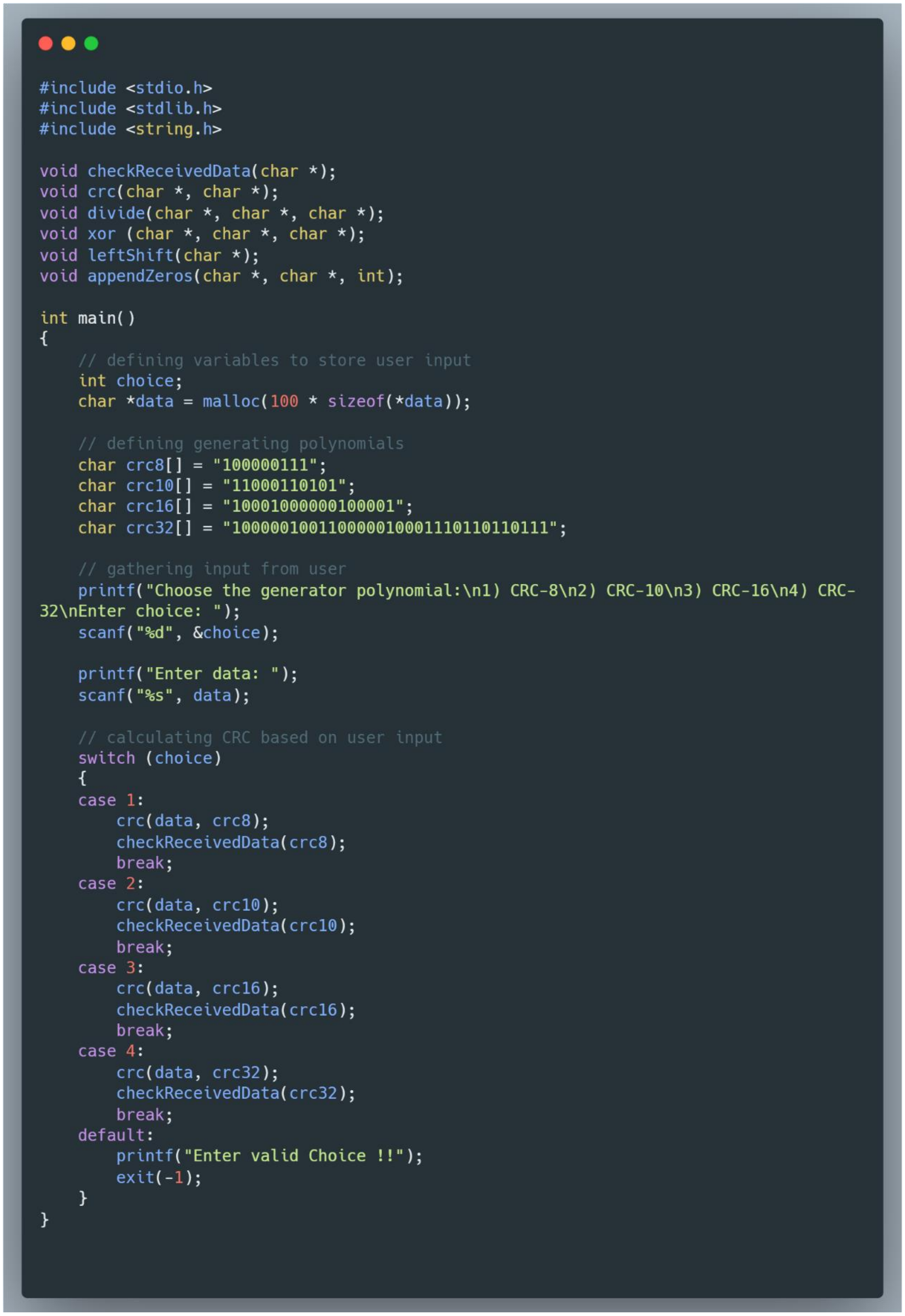
Register No: 18ETCS002121

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
```

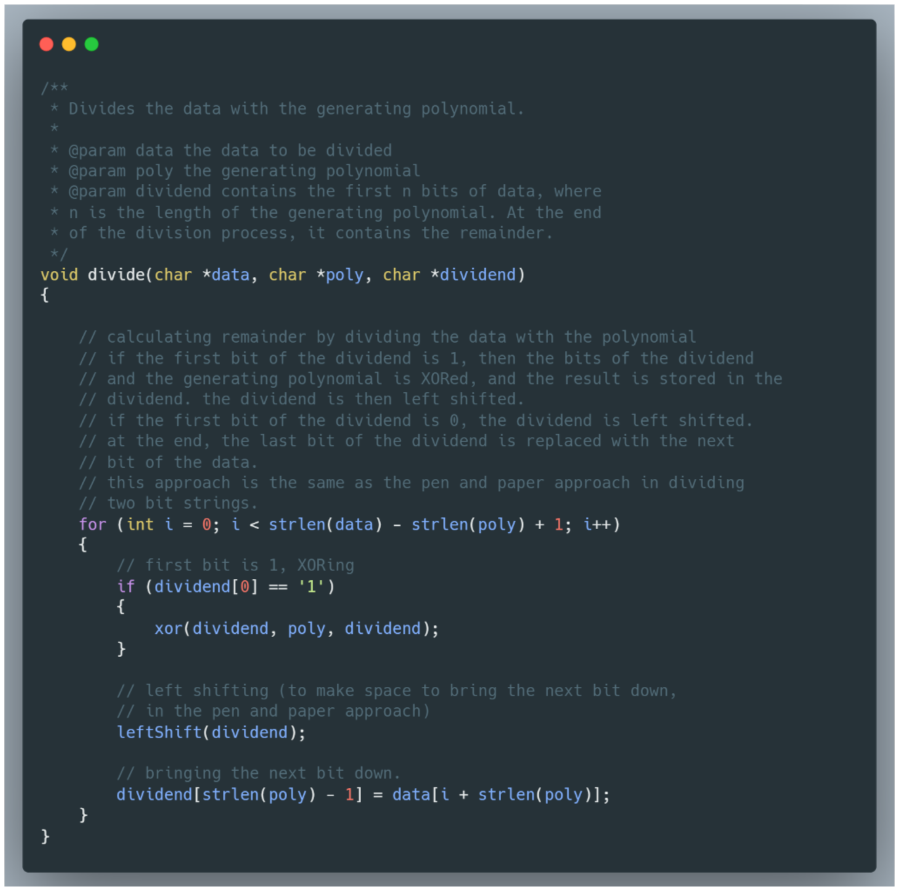
```
    2.3: temp[i] ← (x ^ y) as char
STEP 3: Stop

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
```

2. Program



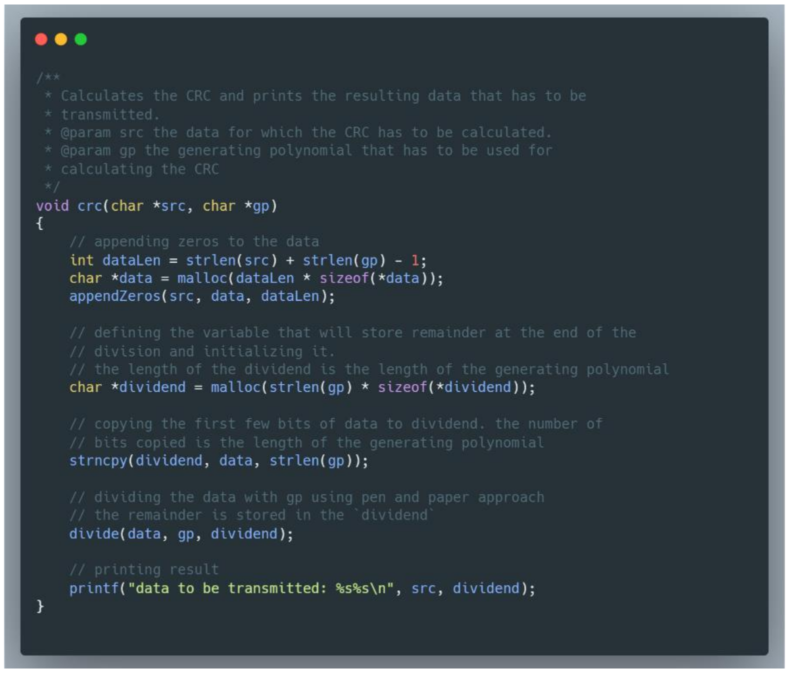
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void checkReceivedData(char *);
void crc(char *, char *);
void divide(char *, char *, char *);
void xor (char *, char *, char *);
void leftShift(char *);
void appendZeros(char *, char *, int);

int main()
{
    // defining variables to store user input
    int choice;
    char *data = malloc(100 * sizeof(*data));

    // defining generating polynomials
    char crc8[] = "100000111";
    char crc10[] = "11000110101";
    char crc16[] = "10001000000100001";
    char crc32[] = "100000100110000010001110110110111";

    // gathering input from user
    printf("Choose the generator polynomial:\n1) CRC-8\n2) CRC-10\n3) CRC-16\n4) CRC-
32\nEnter choice: ");
    scanf("%d", &choice);

    printf("Enter data: ");
    scanf("%s", data);

    // calculating CRC based on user input
    switch (choice)
    {
    case 1:
        crc(data, crc8);
        checkReceivedData(crc8);
        break;
    case 2:
        crc(data, crc10);
        checkReceivedData(crc10);
        break;
    case 3:
        crc(data, crc16);
        checkReceivedData(crc16);
        break;
    case 4:
        crc(data, crc32);
        checkReceivedData(crc32);
        break;
    default:
        printf("Enter valid Choice !!");
        exit(-1);
    }
}
```

Figure 1 Driver Program of CRC



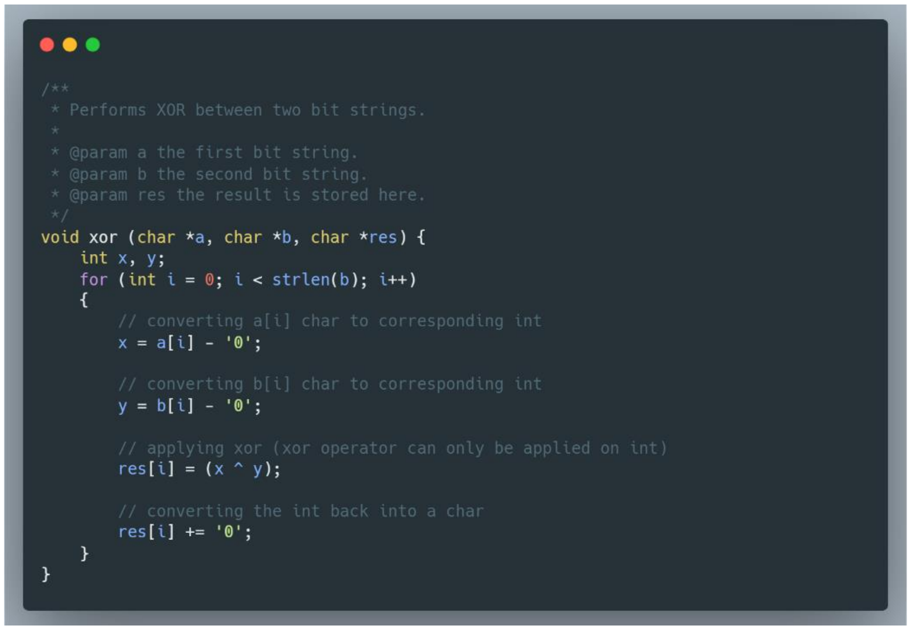
```c
/**
 * Divides the data with the generating polynomial.
 *
 * @param data the data to be divided
 * @param poly the generating polynomial
 * @param dividend contains the first n bits of data, where
 * n is the length of the generating polynomial. At the end
 * of the division process, it contains the remainder.
 */
void divide(char *data, char *poly, char *dividend)
{

    // calculating remainder by dividing the data with the polynomial
    // if the first bit of the dividend is 1, then the bits of the dividend
    // and the generating polynomial is XORed, and the result is stored in the
    // dividend. the dividend is then left shifted.
    // if the first bit of the dividend is 0, the dividend is left shifted.
    // at the end, the last bit of the dividend is replaced with the next
    // bit of the data.
    // this approach is the same as the pen and paper approach in dividing
    // two bit strings.
    for (int i = 0; i < strlen(data) - strlen(poly) + 1; i++)
    {
        // first bit is 1, XORing
        if (dividend[0] == '1')
        {
            xor(dividend, poly, dividend);
        }

        // left shifting (to make space to bring the next bit down,
        // in the pen and paper approach)
        leftShift(dividend);

        // bringing the next bit down.
        dividend[strlen(poly) - 1] = data[i + strlen(poly)];
    }
}
```

Figure 2 Snippet of Divide function



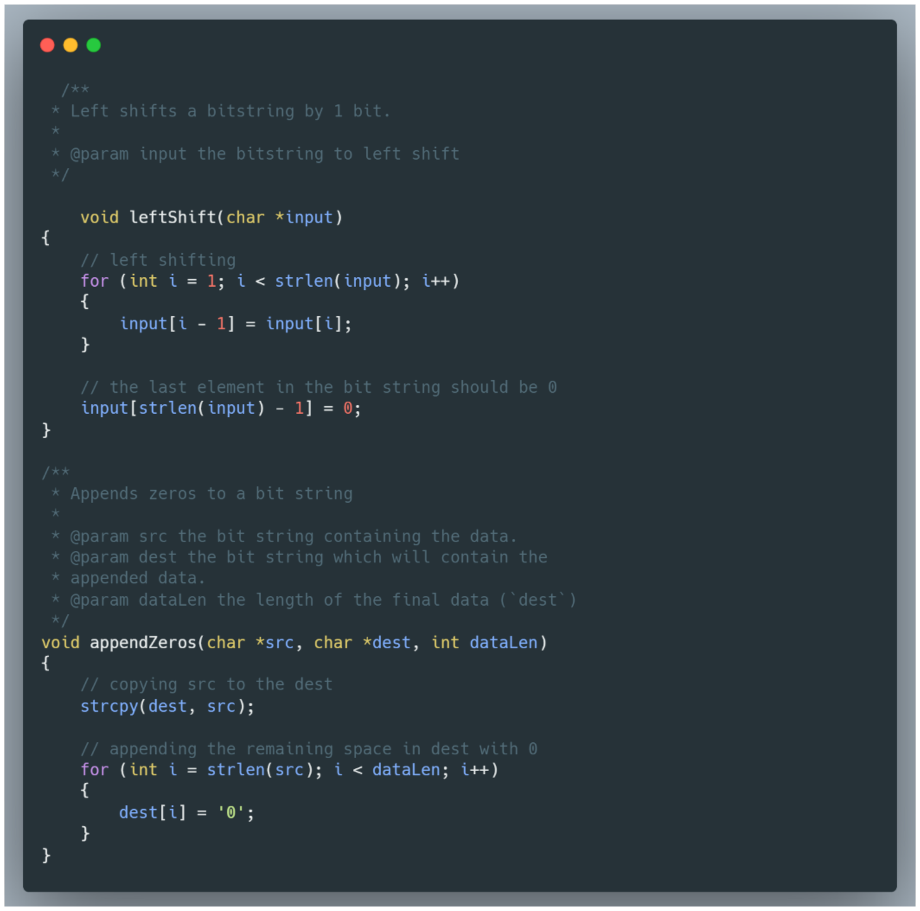
```c
/**
 * Calculates the CRC and prints the resulting data that has to be
 * transmitted.
 * @param src the data for which the CRC has to be calculated.
 * @param gp the generating polynomial that has to be used for
 * calculating the CRC
 */
void crc(char *src, char *gp)
{
    // appending zeros to the data
    int dataLen = strlen(src) + strlen(gp) - 1;
    char *data = malloc(dataLen * sizeof(*data));
    appendZeros(src, data, dataLen);

    // defining the variable that will store remainder at the end of the
    // division and initializing it.
    // the length of the dividend is the length of the generating polynomial
    char *dividend = malloc(strlen(gp) * sizeof(*dividend));

    // copying the first few bits of data to dividend. the number of
    // bits copied is the length of the generating polynomial
    strncpy(dividend, data, strlen(gp));

    // dividing the data with gp using pen and paper approach
    // the remainder is stored in the `dividend`
    divide(data, gp, dividend);

    // printing result
    printf("data to be transmitted: %s%s\n", src, dividend);
}
```

Figure 3 snippet of crc function



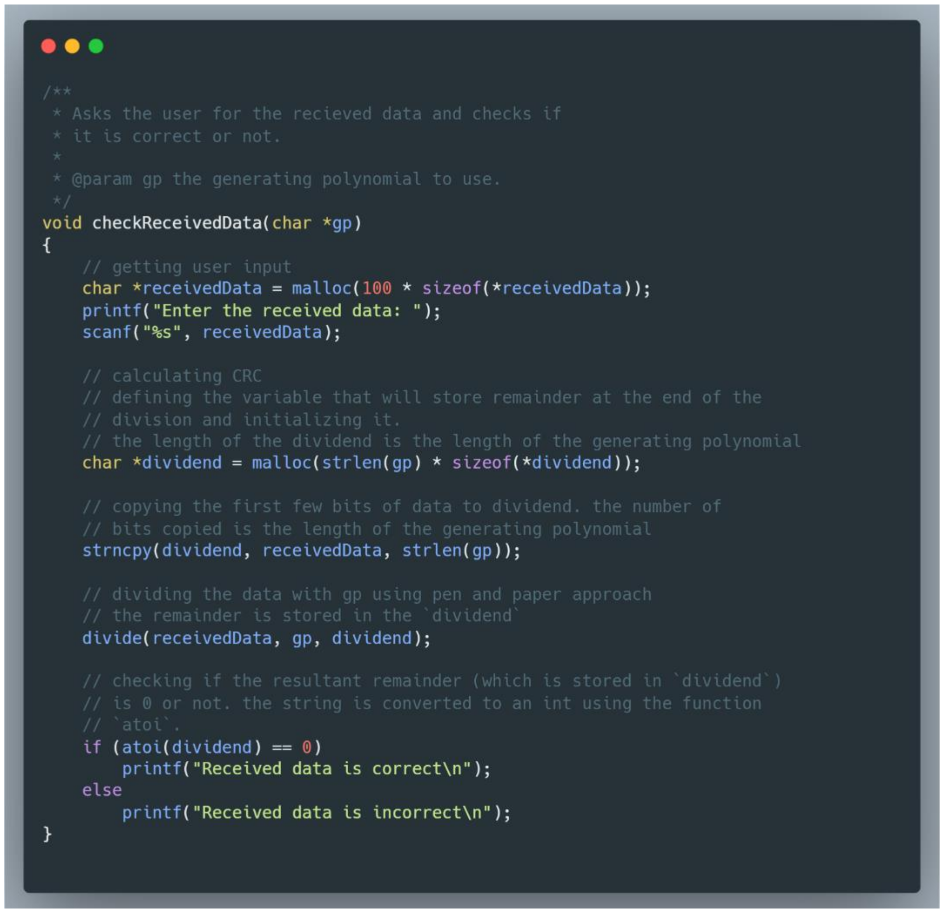
```c
/**
 * Performs XOR between two bit strings.
 *
 * @param a the first bit string.
 * @param b the second bit string.
 * @param res the result is stored here.
 */
void xor (char *a, char *b, char *res) {
    int x, y;
    for (int i = 0; i < strlen(b); i++)
    {
        // converting a[i] char to corresponding int
        x = a[i] - '0';

        // converting b[i] char to corresponding int
        y = b[i] - '0';

        // applying xor (xor operator can only be applied on int)
        res[i] = (x ^ y);

        // converting the int back into a char
        res[i] += '0';
    }
}
```

Figure 4 snippet of crc function



```c
  /**
 * Left shifts a bitstring by 1 bit.
 *
 * @param input the bitstring to left shift
 */

    void leftShift(char *input)
{
    // left shifting
    for (int i = 1; i < strlen(input); i++)
    {
        input[i - 1] = input[i];
    }

    // the last element in the bit string should be 0
    input[strlen(input) - 1] = 0;
}

/**
 * Appends zeros to a bit string
 *
 * @param src the bit string containing the data.
 * @param dest the bit string which will contain the
 * appended data.
 * @param dataLen the length of the final data (`dest`)
 */
void appendZeros(char *src, char *dest, int dataLen)
{
    // copying src to the dest
    strcpy(dest, src);

    // appending the remaining space in dest with 0
    for (int i = strlen(src); i < dataLen; i++)
    {
        dest[i] = '0';
    }
}
```

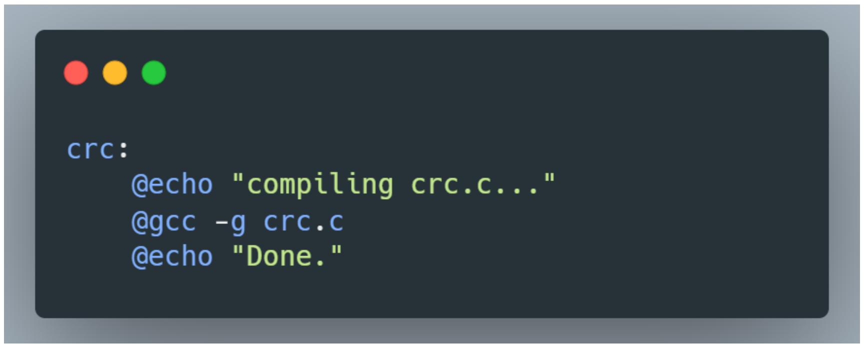
Figure 5 snippet of leftShift and appendZeros function



```c
/**
 * Asks the user for the recieved data and checks if
 * it is correct or not.
 *
 * @param gp the generating polynomial to use.
 */
void checkReceivedData(char *gp)
{
    // getting user input
    char *receivedData = malloc(100 * sizeof(*receivedData));
    printf("Enter the received data: ");
    scanf("%s", receivedData);

    // calculating CRC
    // defining the variable that will store remainder at the end of the
    // division and initializing it.
    // the length of the dividend is the length of the generating polynomial
    char *dividend = malloc(strlen(gp) * sizeof(*dividend));

    // copying the first few bits of data to dividend. the number of
    // bits copied is the length of the generating polynomial
    strncpy(dividend, receivedData, strlen(gp));

    // dividing the data with gp using pen and paper approach
    // the remainder is stored in the `dividend`
    divide(receivedData, gp, dividend);

    // checking if the resultant remainder (which is stored in `dividend`)
    // is 0 or not. the string is converted to an int using the function
    // `atoi`.
    if (atoi(dividend) == 0)
        printf("Received data is correct\n");
    else
        printf("Received data is incorrect\n");
}
```

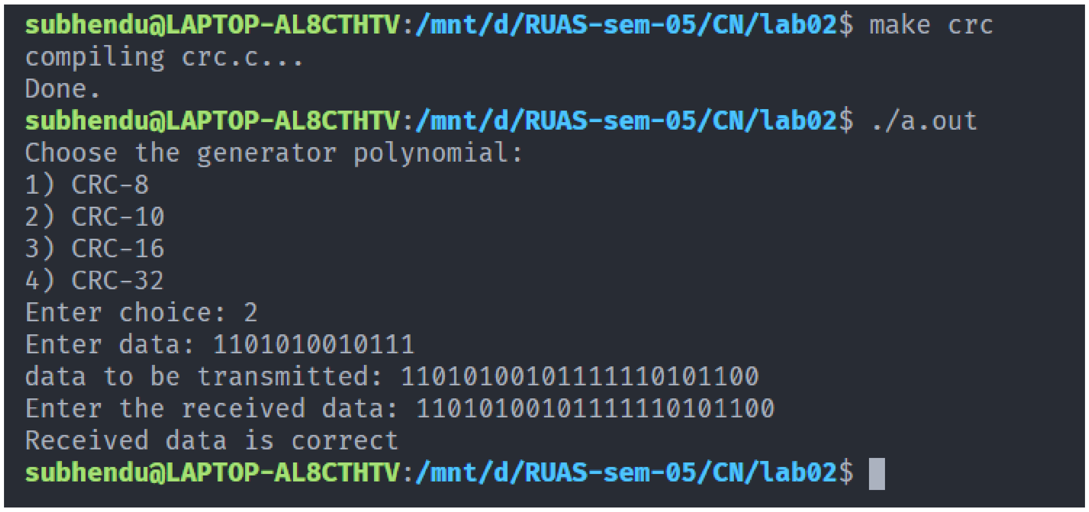
Figure 6 snippet of checkReceivedData function

3. Results



```make
crc:
	@echo "compiling crc.c..."
	@gcc -g crc.c
	@echo "Done."
```

Figure 7 makefile



```
subhendu@LAPTOP-AL8CTHTV:/mnt/d/RUAS-sem-05/CN/lab02$ make crc
compiling crc.c...
Done.
subhendu@LAPTOP-AL8CTHTV:/mnt/d/RUAS-sem-05/CN/lab02$ ./a.out
Choose the generator polynomial:
1) CRC-8
2) CRC-10
3) CRC-16
4) CRC-32
Enter choice: 2
Enter data: 1101010010111
data to be transmitted: 11010100101111110101100
Enter the received data: 11010100101111110101100
Received data is correct
subhendu@LAPTOP-AL8CTHTV:/mnt/d/RUAS-sem-05/CN/lab02$
```

Figure 8 CRC execution

4. 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.

5. 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.

6. Comments

   a. 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.

   b. Learning

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