**Assignment 1: Design and implement an error detection module**

**Name: Sayantan Biswas**

**Class: BCSE 3rd year 1st sem**

**Roll: 001910501057**

**Group: A2**

**Problem Statement: Design and implement an error detection module which has four schemes namely LRC, VRC, Checksum and CRC. Please note that you may need to use these schemes separately for other applications (assignments). You can write the program in any language. The Sender program should accept the name of a test file (contains a sequence of 0,1) from the command line. Then it will prepare the data frame(decide the size of the frame) from the input. Based on the schemes, codeword will be prepared. Sender will send the codeword to the Receiver. Receiver will extract the dataword from codeword and show if there is any error detected. Test the same program to produce a PASS/FAIL result for following cases.**

**(a) Error is detected by all four schemes. Use a suitable CRC polynomial (list is given in next page).**

**(b) Error is detected by checksum but not by CRC.**

**(c) Error is detected by VRC but not by CRC.**

**[Note: Inject error in random positions in the input data frame. Write a separate method for that.]**

**Due on: 09-13 August 2021 (in your respective online lab classes)**

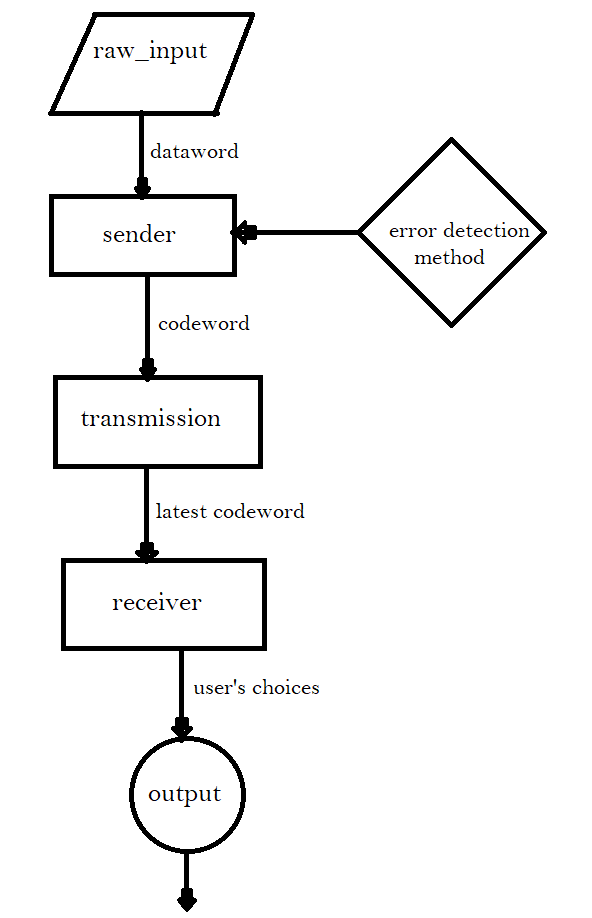
**Report submission due on: 15 August 2021**

DESIGN: The purpose of this program is to detect errors during transmission from sender to receiver using 4 different error detection techniques –

* **VRC (vertical redundancy check)**
* **lrc (longitudinal redundancy check)**
* **checksum**
* **crc (cyclic redundancy check)**

The code has been written in Python 3. There are mainly five files(.py) which execute the whole process –

* sender.py : This file performs the role of a sender; reads the frames of the input(raw\_input.txt file) which contains a sequence of 0,1 as datawords, then converts it into codewords and then sends them to transmission.py
* transmission.py : After applying the inject error method, this file sends the codeword(from ErrorDetection.py) to the receiver.py
* receiver.py : This file performs the role of a receiver; receives the codeword from transmission.py and rejects or accepts the codeword with the help of suitable error checking methods
* ErrorDetection.py : This file contains all the methods of error detection as different functions
* user.py : This file contains the choices that the user will make to run and check various cases



**Error Checking**

**Fig 1: Structure diagram of the program**

**Input Format:** The file ‘raw\_input.txt’ is the input. It contains 48 characters in a sequence of 0, 1.

**Output Format:** Error detected or not by those 4 error detection methods.

IMPLEMENTATION:

The program is written in Python.

Method description is written below with suitable Code snippets:

* **Error Detection:**

There are 4 techniques in this file (ErrorDetection.py) :

1) VRC (Vertical Redundancy Check):

This method takes a list of frames as a parameter, creates codeword by adding one redundant bit at the end of every frame so that the total number of 1s become even.

# Returns codeword for each dataword using vrc

def vrc(list\_of\_frames):

    codewords=[]

    for i in range(len(list\_of\_frames)):

        # For every frame check if the parity in even or odd

        dataword=list\_of\_frames[i]

        if(dataword.count('1')%2==0):

            dataword+='0'

        else:

            dataword+='1'

        codewords.append(dataword)

    return codewords

2) LRC (Longitudinal Redundancy Check):

This method takes a list of frames and a frame size (no\_of\_bits) as parameters. It returns the LRC code of all frames.

# Function to generate the LRC code for a list of frames

def lrc(list\_of\_frames, no\_of\_bits):

    lsum=0

    for frame in list\_of\_frames:

        lsum=lsum^int(frame,2)

    lsum=bin(lsum)[2:]

    # Stuffing

    while(len(lsum)<no\_of\_bits):

        lsum='0'+lsum

    return lsum

3) Checksum:

This method takes a list of frame and frame size as a parameter and returns

the checksum of all frames. the checksum is appended at the end of the

last frame.

# Function to find checksum of a number of frames

def checksum(list\_of\_frames, no\_of\_bits):

    chksum=0

    for frame in list\_of\_frames:

        chksum=chksum+int(frame,2) # Computing the sum

    # Wrapping the sum

    csum=bin(chksum) # binary form

    csum=csum[2:] # Adding csum starting from index 2 (0 based indexing)

    while(len(csum)>no\_of\_bits):

        first=csum[0:len(csum)-no\_of\_bits]

        second=csum[len(csum)-no\_of\_bits:]

        s=int(first,2)+int(second,2)

        csum=bin(s)

        csum=csum[2:]

    # Perform 1s complement

    while(len(csum)<no\_of\_bits):

        csum='0'+csum

    chksum=''

    for i in range(len(csum)):

        if(csum[i]=='0'):

            chksum+='1'

        else:

            chksum+='0'

    return chksum

4) CRC (Cyclic Redundancy Check):

This method takes a list of frame, generator (CRC polynomial) and a frame

size (no\_of\_bits) as parameters. In order to create the codeword for CRC;

modulo 2 division function is there. This modulo 2 division function takes

the dataword and the generator polynomial as parameters which performs

the CRC division to return the codewords.

Below code snippet defines the CRC polynomial and frame size (i.e. no\_of\_bits)

# CRC-4-ITU -> x^4 + x + 1

generator\_poly = '10011'

no\_of\_bits\_crc = len(generator\_poly) # n = 5

a xor function for modulo 2 division :

# xor function for two binary strings which is typecasted to integer

def xor(a,b):

    a=int(a,2) # returns the integer value which is equivalent to binary string

    b=int(b,2)

    a=a^b

    a=bin(a)[2:]

    return a

modulo 2 division function :

# Function to perform modulo 2 division

def modulo2div(dataword, generator):

    # Number of bits to be XORed a time

    l\_xor=len(generator)

    tmp=dataword[0:l\_xor]

    while (l\_xor<len(dataword)):

        if(tmp[0]=='1'):

            # If leftmost bit is 1 simply xor and bring the next bit down

            tmp=xor(generator,tmp)+dataword[l\_xor]

        else:

            # If leftmost bit is 0 then use all 0 divisor

            tmp=xor('0'\*len(generator),tmp)+dataword[l\_xor]

        tmp='0'\*(len(generator)-len(tmp))+tmp

        l\_xor+=1

    # For the last bit

    if(tmp[0]=='1'):

        tmp=xor(generator,tmp)

    else:

        tmp=xor('0'\*len(generator),tmp)

    tmp='0'\*(len(generator)-len(tmp)-1)+tmp

    checkword=tmp

    return checkword

* **Sender :**

This file (sender.py) performs the role of a sender; it creates the codewords from datawords with the help of ErrorDetection.py

1) This function reads the input file which passed as argument, splits into frames and then returns a list of frames.

# Function to read from the input file and convert it to a list of frames

def readfile(filename, no\_of\_bits):

    # Open the file in read mode

    f=open(filename,'r')

    data=f.read()

    # Now split the data into frames

    list\_of\_frames=[data[i:i+no\_of\_bits] for i in range(0, len(data), no\_of\_bits)]

    return list\_of\_frames

2) This function takes list of frames and the frame size as parameters and it creates the LRC value with the help of Error Detection methods and it appends at the end of the last frame. It also prints the list of final frames which contains codewords.

# Function to write the lrc frames to file

def write\_lrc(list\_of\_frames, no\_of\_bits):

    lrcval=err.lrc(list\_of\_frames=list\_of\_frames, no\_of\_bits=no\_of\_bits)

    list\_of\_frames2=list\_of\_frames[:]

    list\_of\_frames2.append(lrcval) #Append the resulted LRC at the last frame of datawords

    # Printing the frames

    print('Codeword frames sent:')

    print(list\_of\_frames2)

    with open('Original\_sender.txt', 'w') as fw:

        fw.write("\n\nOriginal LRC ---\n")

        for item in list\_of\_frames2:

            fw.write("%s | " % item)

3) This function takes list of frames and the frame size as parameters and it creates the VRC value with the help of Error Detection methods and it appends at the end of every frame and also prints the list of final frames which contains codewords.

# Function to write the vrc frames to file

def write\_vrc(list\_of\_frames, no\_of\_bits):

    list\_of\_frames2=err.vrc(list\_of\_frames=list\_of\_frames)[:]

    # Printing the frames

    print('Codeword frames sent:')

    print(list\_of\_frames2)

    with open('Original\_sender.txt', 'a') as fw:

        fw.write("\n\nOriginal VRC ---\n")

        for item in list\_of\_frames2:

            fw.write("%s | " % item)

4) This method is responsible for writing the checksum with the help of error detection methods from the checksum algorithm and then finally append at the last frame. Then it creates codewords.

# Function to write the checksum frames to the file

def write\_chksum(list\_of\_frames, no\_of\_bits):

    chksum=err.checksum(list\_of\_frames=list\_of\_frames, no\_of\_bits=no\_of\_bits)

    list\_of\_frames2=list\_of\_frames[:] # added all frames

    list\_of\_frames2.append(chksum) # put checksum at the end of last frame

    # Printing the frames

    print('Codeword frames sent:')

    print(list\_of\_frames2)

    with open('Original\_sender.txt', 'a') as fw:

        fw.write("\n\nOriginal checksum ---\n")

        for item in list\_of\_frames2:

            fw.write("%s | " % item)

5) This method is responsible for creating the CRC value with the help of Error Detection methods from the CRC algorithm. Then, it creates codewords.

# Function to write the crc frames to the file

def write\_crc(list\_of\_frames, generator):

    list\_of\_frames2=err.crc(list\_of\_frames=list\_of\_frames, generator=err.generator\_poly,

                            no\_of\_bits=err.no\_of\_bits\_crc)[:]

    # Printing the frames

    print('Codeword frames sent :')

    print(list\_of\_frames2)

    with open('Original\_sender.txt', 'a') as fw:

        fw.write("\n\nOriginal CRC ---\n")

        for item in list\_of\_frames2:

            fw.write("%s | " % item)

* **Transmission :**

This file (transmission.py) injects the error into the codewords, then transmits those codewords to receiver.py

1) This method is used to inject the errors.

# Function to inject error

def inject\_error(list\_of\_frames, frame\_no, list\_of\_bit):

    list\_of\_frames2=list\_of\_frames[:]

    frame=list\_of\_frames2[frame\_no]

    new=list(frame)

    # Inserting error in the given bit position here

    for i in range(len(list\_of\_bit)):

        if(new[list\_of\_bit[i]]=='0'):

            new[list\_of\_bit[i]]='1'

        elif (new[list\_of\_bit[i]]=='1'):

            new[list\_of\_bit[i]]='0'

    list\_of\_frames2[frame\_no]=''.join(new)

    return list\_of\_frames2

2) This method is responsible for sending the list of frames where the error was injected in the frame with the appropriate bit position.

# sending codewords through transmission.py

# error\_bit\_list is a list of lists containing bit posiiton of errors in each frame

def sending\_codeword(list\_of\_frames, no\_of\_bits, error\_list\_frames, error\_bit\_list):

    global no\_of\_errors

    medium\_lrc(list\_of\_frames, no\_of\_bits, error\_list\_frames, error\_bit\_list)

    medium\_vrc(list\_of\_frames, no\_of\_bits, error\_list\_frames, error\_bit\_list)

    medium\_chksum(list\_of\_frames, no\_of\_bits, error\_list\_frames, error\_bit\_list)

    medium\_crc(list\_of\_frames, no\_of\_bits, error\_list\_frames, error\_bit\_list)

3) This method writes the LRC frames to file.

# Function to write the lrc frames to file

def medium\_lrc(list\_of\_frames, no\_of\_bits, error\_list\_frames, error\_bit\_list):

    lrcval=err.lrc(list\_of\_frames=list\_of\_frames, no\_of\_bits=no\_of\_bits)

    list\_of\_frames2=list\_of\_frames[:]

    list\_of\_frames2.append(lrcval)

    # Inserting error

    for i in range(len(error\_list\_frames)):

        list\_of\_frames2=inject\_error(list\_of\_frames2, error\_list\_frames[i], error\_bit\_list[i])

        # Reciever will read this txt file

    with open('Transmission\_LRC.txt', 'w') as fw:

        for item in list\_of\_frames2:

            item='0'\*(len(err.generator\_poly)-1)+item # for list maintaining order

                                                      # without this, bug arises

            fw.write("%s" % item)

    with open('Latest\_reciver.txt', 'w') as fw:

        fw.write('\n\nReciever LRC ---\n')

        for i in list\_of\_frames2:

            fw.write("%s | " % i)

.

4) This method writes the VRC frames to file.

# Function to insert error toggling the vrc introduced frames to file

def medium\_vrc(list\_of\_frames, no\_of\_bits, error\_list\_frames, error\_bit\_list):

    list\_of\_frames2=err.vrc(list\_of\_frames=list\_of\_frames)[:]

    # Inserting error

    for i in range(len(error\_list\_frames)):

        list\_of\_frames2=inject\_error(list\_of\_frames2, error\_list\_frames[i], error\_bit\_list[i])

    with open('Transmission\_VRC.txt', 'w') as fw:

        for item in list\_of\_frames2:

            item='0'\*(len(err.generator\_poly)-2)+item

            fw.write("%s" % item)

    with open('Latest\_reciver.txt', 'a') as fw:

        fw.write('\n\nReciever VRC ---\n')

        for i in list\_of\_frames2:

            fw.write("%s | " % i)

5) This method writes the checksum frames to file.

# Function to insert error toggling the checksum frames to file

def medium\_chksum(list\_of\_frames, no\_of\_bits, error\_list\_frames, error\_bit\_list):

    chksum=err.checksum(list\_of\_frames=list\_of\_frames, no\_of\_bits=no\_of\_bits)

    list\_of\_frames2=list\_of\_frames[:]

    list\_of\_frames2.append(chksum)

    # Inserting error

    for i in range(len(error\_list\_frames)):

        list\_of\_frames2=inject\_error(list\_of\_frames2, error\_list\_frames[i], error\_bit\_list[i])

    with open('Transmission\_Checksum.txt', 'w') as f:

        for item in list\_of\_frames2:

            item=item='0'\*(len(err.generator\_poly)-1)+item

            f.write("%s" % item)

    with open('Latest\_reciver.txt', 'a') as fw:

        fw.write('\n\nReciever checksum ---\n')

        for i in list\_of\_frames2:

            fw.write("%s |" % i)

6) This method writes the CRC frames to file.

# Function to insert error toggling the CRC frames to file

def medium\_crc(list\_of\_frames, generator, error\_list\_frames, error\_bit\_list):

    list\_of\_frames2=err.crc(list\_of\_frames=list\_of\_frames,

        generator=err.generator\_poly, no\_of\_bits=err.no\_of\_bits\_crc)[:]

    # Inserting error

    for i in range(len(error\_list\_frames)):

        list\_of\_frames2=inject\_error(list\_of\_frames2, error\_list\_frames[i], error\_bit\_list[i])

    with open('Transmission\_CRC.txt', 'w') as f:

        for item in list\_of\_frames2:

            f.write("%s" % item)

    with open('Latest\_reciver.txt', 'a') as fw:

        fw.write('\n\nReciever CRC ---\n')

        for i in list\_of\_frames2:

            fw.write("%s | " % i)

* **Receiver :**

This file (receiver.py) performs the role of a receiver and it checks the error(s) is(are) present or not with the help of error checking methods.

1) In this method, it takes a list of frames and size of frames to detect the error for LRC codeword with the help of error checking method.

# Check for error by lrc

def check\_lrc(list\_of\_frames, no\_of\_bits):

    # Removing padding

    list\_of\_frames=[list\_of\_frames[i][len(err.generator\_poly)-1:]

     for i in range(len(list\_of\_frames))]

    lrcval=err.lrc(list\_of\_frames=list\_of\_frames, no\_of\_bits=no\_of\_bits)

    #if the appended value is zero

    if(int(lrcval,2)==0):

        print('No error is detected by LRC')

        print('Dataword frames are ')

        print(list\_of\_frames[0:-1])

    else:

        print('Error is detected by LRC')

2) In this method, it takes a list of frames to detect the error for VRC codeword with the help of error checking method.

# Check for error by vrc

def check\_vrc(list\_of\_frames):

    # Removing padding

    list\_of\_frames=[list\_of\_frames[i][len(err.generator\_poly)-2:]

     for i in range(len(list\_of\_frames))]

    flag=True

    for i in range(len(list\_of\_frames)):

        if(list\_of\_frames[i].count('1')%2!=0):

            print('Error is detected in frame '+str(i+1)+' by VRC')

            flag=False

    if(flag):

        # No error extract dataword

        print("No error is detected by VRC")

        list\_of\_frames=[list\_of\_frames[i][0:-1] for i in range(len(list\_of\_frames))]

        print('Dataword frames are ')

        print(list\_of\_frames)

3) In this method, it takes a list of frames and the frame size to detect the error for checksum codeword with the help of error checking method.

# Check for error by checksum

def check\_checksum(list\_of\_frames, no\_of\_bits):

    # Removing padding

    list\_of\_frames=[list\_of\_frames[i][len(err.generator\_poly)-1:]

    for i in range(len(list\_of\_frames))]

    chksum=err.checksum(list\_of\_frames=list\_of\_frames, no\_of\_bits=no\_of\_bits)

    if(int(chksum,2)==0):

        # In case of no error detected then dataword is printed

        print('No error is detected by checksum')

        print('Dataword frames are ')

        print(list\_of\_frames[0:-1])

    else:

        print('Error is detected by checksum')

4) In this method, it takes a list of frames and the CRC polynomial to detect the error for checksum codeword with the help of error checking method. For every frame, the **modulo2div()** method is applied and if the remainder is 0 then there is no error otherwise there is an error in the frame.

# Check for error by crc

def check\_crc(list\_of\_frames, generator):

    flag=True

    for i in range(len(list\_of\_frames)):

        if(int(err.modulo2div(list\_of\_frames[i],err.generator\_poly),2)!=0):

            print('Error is detected in frame '+str(i+1)+' by CRC')

            flag=False

    if(flag):

        list\_of\_frames=[list\_of\_frames[i][0:err.no\_of\_bits\_crc]

        for i in range(len(list\_of\_frames))]

        print('Dataword frames are ')

        print(list\_of\_frames)

        print("No error is detected by CRC")

* **User :**

This file (user.py) combines all the above files, functions and gives the output with the help of user choices for different cases.

def case1() :

    list\_of\_frames=(se.readfile('raw\_input.txt',no\_of\_bits=err.no\_of\_bits))

    print('Case1: All 4 schemes can detect the error')

    se.dataword\_to\_codeword(list\_of\_frames, no\_of\_bits=err.no\_of\_bits)

    tm.sending\_codeword(list\_of\_frames,no\_of\_bits=err.no\_of\_bits, error\_list\_frames=[0, 1], error\_bit\_list=[[2], [3]])

    re.modules()

    print('--------------------------------------------------------------------------')

def case2() :

    list\_of\_frames=(se.readfile('raw\_input.txt',no\_of\_bits=err.no\_of\_bits))

    print('Case2: Error detected by checksum but not by CRC')

    se.dataword\_to\_codeword(list\_of\_frames, no\_of\_bits=err.no\_of\_bits)

    #tm.sending\_codeword(list\_of\_frames,no\_of\_bits=err.no\_of\_bits, error\_list\_frames=[0], error\_bit\_list=[[0, 4, 7]])

    tm.sending\_codeword(list\_of\_frames,no\_of\_bits=err.no\_of\_bits, error\_list\_frames=[0], error\_bit\_list=[[0, 3, 4]])

    re.modules()

    print('--------------------------------------------------------------------------')

def case3() :

    list\_of\_frames=(se.readfile('raw\_input.txt',no\_of\_bits=err.no\_of\_bits))

    print('Case3: Error detected by VRC but not by CRC')

    se.dataword\_to\_codeword(list\_of\_frames, no\_of\_bits=err.no\_of\_bits)

    #tm.sending\_codeword(list\_of\_frames,no\_of\_bits=err.no\_of\_bits, error\_list\_frames=[1], error\_bit\_list=[[0, 4, 7]])

    tm.sending\_codeword(list\_of\_frames,no\_of\_bits=err.no\_of\_bits, error\_list\_frames=[0, 1], error\_bit\_list=[[0, 1, 5, 6, 8]])

    re.modules()

    print('--------------------------------------------------------------------------')

TEST CASES :

The input has been kept in a file (*raw\_input*). It consists of 48 bits of binary number (i.e. sequence of 0, 1).

**INPUT DATA:** 100100011100111100111100101010111100111011010001

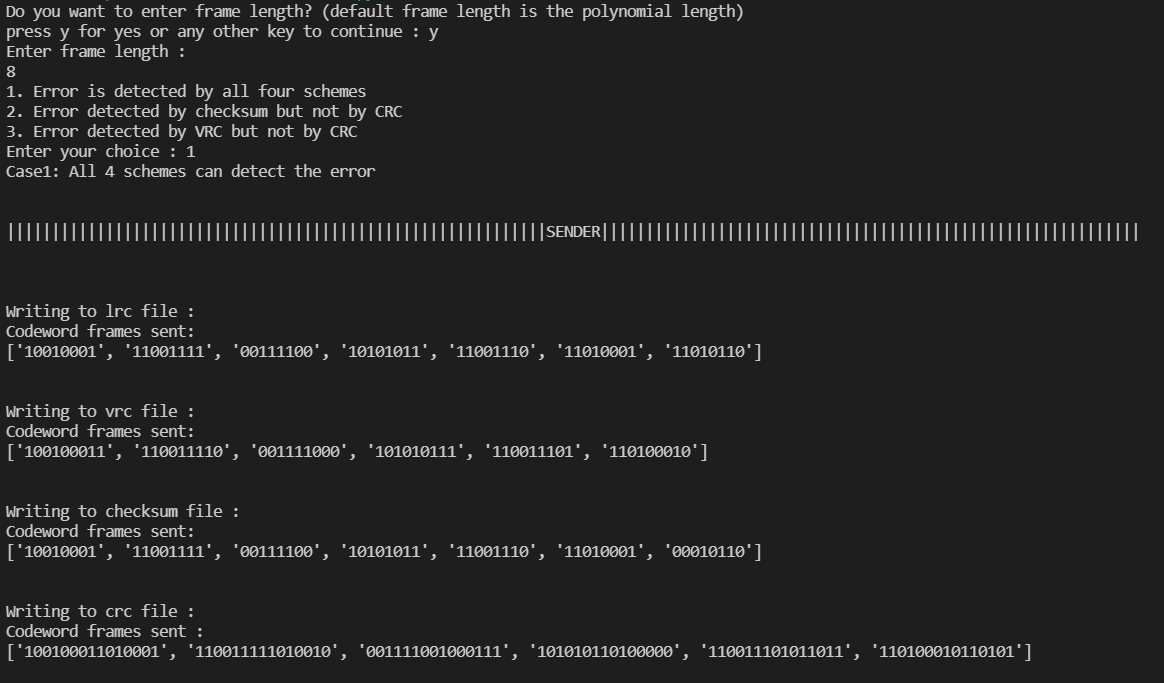
**CRC Polynomial:** 10001001 (CRC-7 -> *x*^7 + *x*^3 + 1)

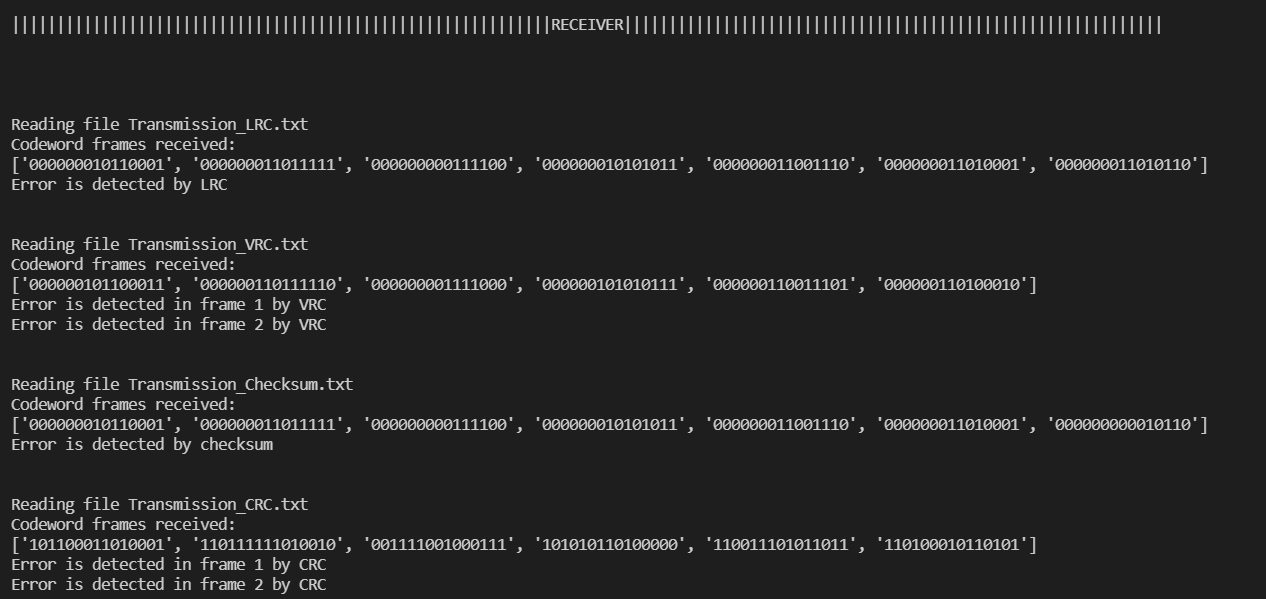
I have taken 8 bits frame size. Total number of frames is 6.

* Case 1: Error is detected by all four schemes.

In transmission.py errors have been injected in frame 0 (bit position 2) and frame 1 (bit position 3). All schemes detect error.

**OUTPUT:**

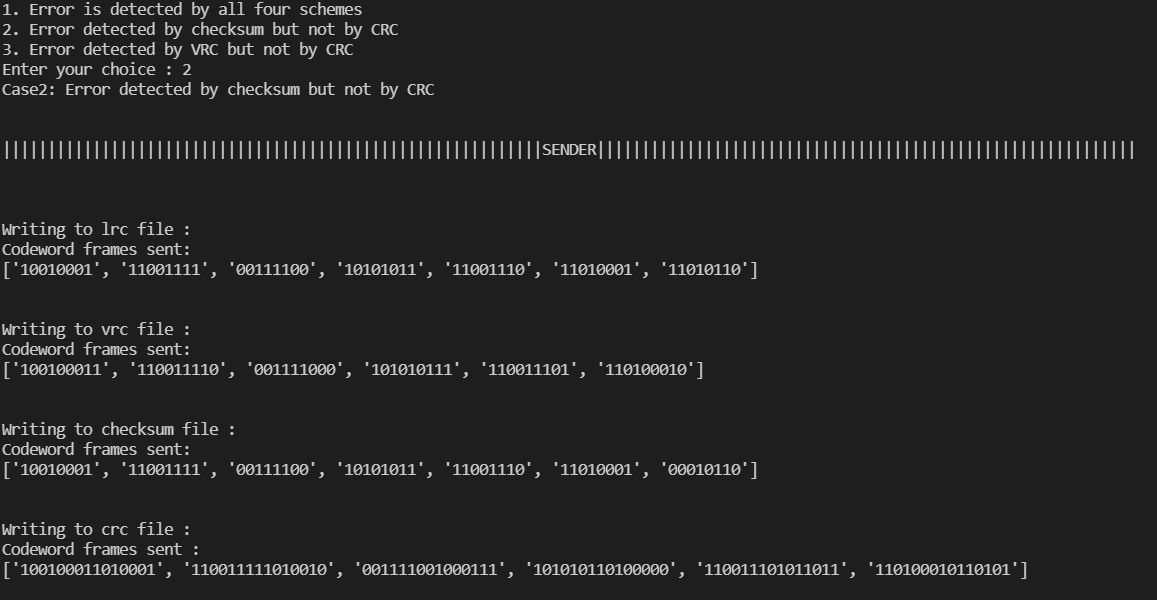


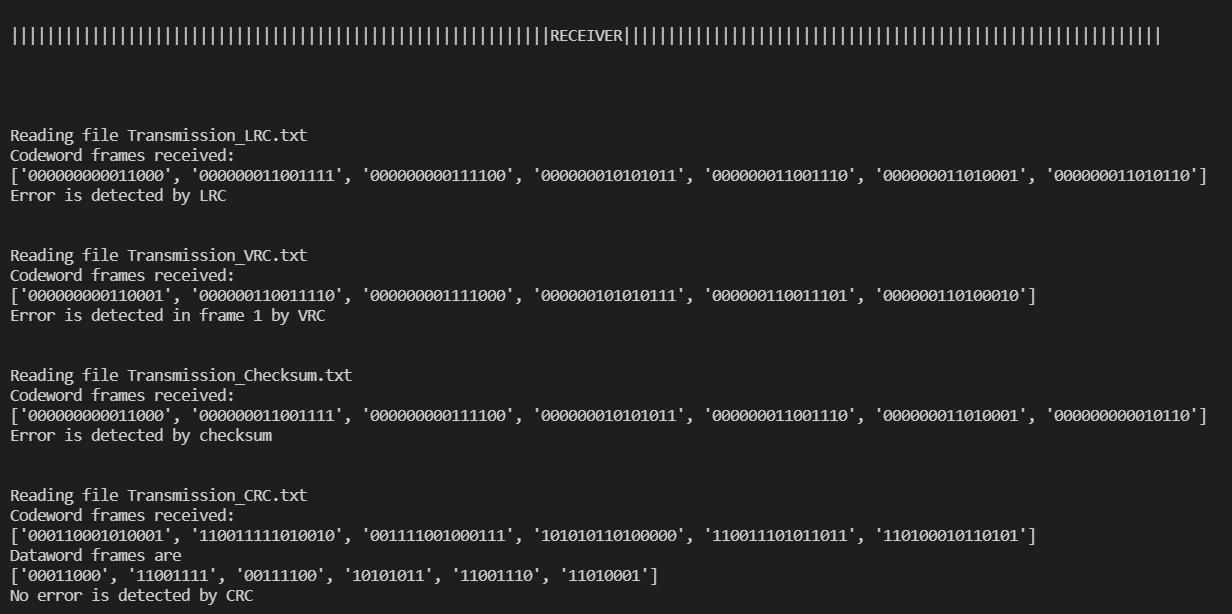


* Case 2: Error is detected by checksum but not by CRC.

In transmission.py errors have been injected in frame 0 (bit position 0, 4, 7). All schemes detect error except CRC.

**OUTPUT:**



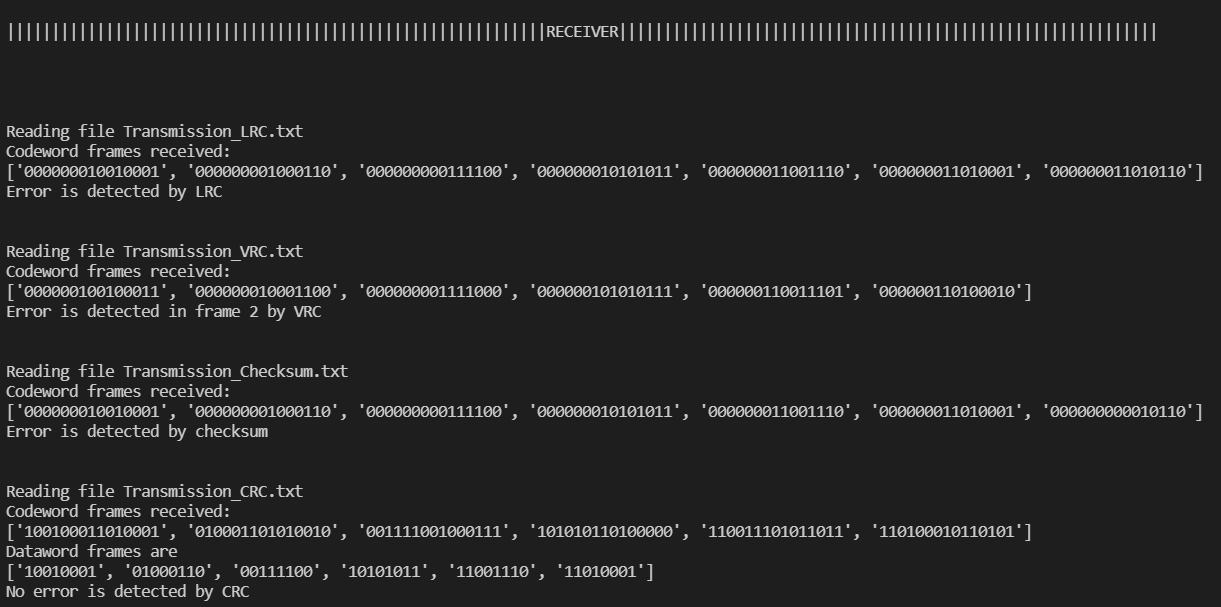


* Case 3: Error is detected by VRC but not by CRC.

In transmission.py errors have been injected in frame 1 (bit position 0, 4, 7). All schemes detect error except CRC.

**OUTPUT:**





Results :

**Performance Metric :**

Error Ratio = (False Positive) / (Frames Transmitted)

Where, False Positive means the error is injected (i.e. the frame has been changed) during the transmission but still accepted by scheme.

**Evaluation Table**:

**Methods Error Ratio**

LRC 0.5

VRC 0.5

Checksum 0.5

CRC 0.16

10 independent executions have been taken. Total number of frames (dataword) for each scheme per execution is 60 (6\*10) and frame size is 8 bit. Data is 48 bit.

Analysis :

* After evaluating the results, we can say CRC is sturdier and reliable than most other known schemes.
* Error detection capabilities of the code are increased significantly when all 4 schemes are used.

Comments :

The assignment has helped in understanding and implementing the various error detection schemes. I have analyzed the performance of each of the schemes after introducing random errors. The difficulty level was moderate.