**NETWORK LAB REPORT**

**NAME:** ANURAN CHAKRABORTY

**ROLL NO.:** 20

**CLASS:** BCSE-III

**SECTION:** A1

**ASSIGNMENT NUMBER:** 1

**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 data word 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.

(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.]

**DEADLINE:** 24TH JANUARY, 2019

**SUBMITTED ON:** 24TH JANUARY, 2019

**REPORT SUBMITTED ON:** 31ST JANUARY, 2019

**DESIGN**

The program implements the four error-detection mechanisms namely Checksum, LRC, VRC, CRC. The program consists of four files which handles the various aspects of the program.

1. **errorchecker.py**

This file contains the implementation of all the above algorithms in different functions.

1. **sender.py**

This file contains the code to perform the work of the sender. Read from the input file, form codeword from the data word, inject error and write to the output file.

1. **receiver.py**

This code reads from the file written to by the sender, checks if any error exists and accordingly gives an output.

1. **main.py**

As a wrapper to run all the above modules

|  |
| --- |
|  |
| ***Fig. 1.*** *A brief outline of the program design* |

**Fig. 1** gives a brief outline of the program. The sender program reads input from a file, splits it into frames, converts data word to codeword, introduces some random error, and then writes the codeword to the respective files as shown. The receiver then reads from these files, splits it into frames and then applies the algorithms to find whether there is any error introduced or not.

Some important parameters for the design of the program are:

**CRC Polynomial:** The CRC polynomial which has been considered is CRC-7 ().

**Frame size:** Based on the CRC polynomial chosen, the frame size is chosen to be 8 as it has the same number of terms as in the CRC polynomial.

**Assumption:** During the design one assumption that has been made is that the number of bits in the input file is a multiple of 8.

**Input format:** The input for the program is a text file consisting of a string of only 0s and 1s.

**Output format:** The program outputs whether any error is detected by the respective methods.

**IMPLEMENTATION**

The assignment has been implemented in Python3. The detailed description is given below.

***errorchecker.py***

This file has all the error detection algorithms implemented in it.

*# Customisable parameters*

generator\_poly='10001001'

no\_of\_bits=len(generator\_poly)

The above code snippet defines the CRC polynomial and the frame size.

The following methods are present

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

This function takes a list of frames and the frame size as a parameter and return the checksum of all the frames within the list.

*# 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)

csum=csum[2:]

**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

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

This function takes a list of frames and the frame size as a parameter and return the lrc value of all the frames within the list.

*# 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

*vrc(list\_of\_frames):*

This function takes a list of frames as a parameter and then modifies the elements of the list i.e. a frame by adding one redundant bit at the end by making sure the entire frame has an even number of 1s (even parity).

*# 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

*crc(list\_of\_frames, generator, no\_of\_bits):*

This function takes a list of frames and the frame size as a parameter a the creates the CRC codeword for each frame by taking help of the ***modulo2div(dataword, generator)*** function. The ***modulo2div(dataword, generator)*** function takes a dataword and the generator polynomial as input and returns the codeword for the dataword after performing CRC division.

*# 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

*# Return the codeword for crc*

**def** crc(list\_of\_frames, generator, no\_of\_bits):

codewords=[]

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

*# For every dataword perform crc division*

dataword=list\_of\_frames[i]

*# Append length of generator-1 bits to dataword*

aug\_dataword=dataword+'0'\*(len(generator)-1)

*# Now perform the modulo 2 division*

checkword=modulo2div(aug\_dataword,generator)

*# Append the remainder*

codeword=dataword+checkword

codewords.append(codeword)

**return** codewords

***sender.py***

This module is responsible for performing the task of the sender by taking help of the errorchecker module.

*readfile(filename, no\_of\_bits):*

The function reads the input file which is passed as argument, splits it into frames, and then returns a list of frames to be used for various error checking algorithms.

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

**def** readfile(filename, no\_of\_bits):

*# Open the file for reading*

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

*ins\_error(list\_of\_frames, frame\_no, list\_of\_bit):*

This function takes a list of frames, the frame number and the bit position where error is to be introduced and then returns a new list of frames with the required bits in the required frame toggled. Overall, this function is for inserting error into a particular frame.

*# Function to introduce error*

**def** ins\_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

*write\_chksum(list\_of\_frames, no\_of\_bits, error\_list\_frames, error\_bit\_list):*

This function takes as input list of frames, frame size, the positions where any error is to be introduced, and the respective bit position for every frame where error is to be introduced. Then the checksum is calculated for the frames and the frames (including the checksum frame) are written to the respective file (csum\_op.txt). The function ***ins\_error()*** is used to insert the errors.

*# Function to write the checksum frames to file*

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

*# error\_list\_frames contains the list of frames to introduce the error into*

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

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)

*# Printing the frames*

**print**('Codeword frames sent:')

**print**(list\_of\_frames2)

*# Inserting error*

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

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

**with** open('csum\_op.txt', 'w') **as** f:

**for** item **in** list\_of\_frames2:

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

f.write("**%s**" % item)

*write\_lrc(list\_of\_frames, no\_of\_bits, error\_list\_frames, error\_bit\_list):*

This function takes as input list of frames, frame size, the positions where any error is to be introduced, and the respective bit position for every frame where error is to be introduced. Then the LRC value is calculated by using the respective function in the ***errorchecker*** module for the frames and the frames (including the LRC value frame) are written to the respective file (lrc\_op.txt). The function ***ins\_error()*** is used to insert the errors.

*# Function to write the lrc frames to file*

**def** write\_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)

*# Printing the frames*

**print**('Codeword frames sent:')

**print**(list\_of\_frames2)

*# Inserting error*

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

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

**with** open('lrc\_op.txt', 'w') **as** f:

**for** item **in** list\_of\_frames2:

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

f.write("**%s**" % item)

*write\_vrc(list\_of\_frames, no\_of\_bits, error\_list\_frames, error\_bit\_list):*

This function takes as input list of frames, frame size, the positions where any error is to be introduced, and the respective bit position for every frame where error is to be introduced. Then the VRC frames are obtained by using the respective function in the ***errorchecker*** module and the frames are written to the respective file (vrc\_op.txt). The function ***ins\_error()*** is used to insert the errors.

*# Function to write the vrc frames to file*

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

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

*# Printing the frames*

**print**('Codeword frames sent:')

**print**(list\_of\_frames2)

*# Inserting error*

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

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

**with** open('vrc\_op.txt', 'w') **as** f:

**for** item **in** list\_of\_frames2:

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

f.write("**%s**" % item)

*write\_crc(list\_of\_frames, generator, no\_of\_bits, error\_list\_frames, error\_bit\_list):*

This function takes as input list of frames, the generator polynomial, frame size, the positions where any error is to be introduced, and the respective bit position for every frame where error is to be introduced. Then the CRC frames are obtained by using the respective function in the ***errorchecker*** module and the frames are written to the respective file (vrc\_op.txt). The function ***ins\_error()*** is used to insert the errors.

*# Function to write the crc frames to file*

**def** write\_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)[:]

*# Printing the frames*

**print**('Codeword frames sent:')

**print**(list\_of\_frames2)

*# Inserting error*

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

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

**with** open('crc\_op.txt', 'w') **as** f:

**for** item **in** list\_of\_frames2:

f.write("**%s**" % item)

*dataword\_to\_codeword(list\_of\_frames, no\_of\_bits, error\_list\_frames, error\_bit\_list):*

This function is a wrapper for calling all the above functions.

*# Coverts dataword to codeword and wrote to the appropriate file*

**def** dataword\_to\_codeword(list\_of\_frames, no\_of\_bits, error\_list\_frames, error\_bit\_list):

**global** no\_of\_errors

**print**('Writing to checksum file')

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

**print**('Writing to lrc file')

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

**print**('Writing to vrc file')

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

**print**('Writing to crc file')

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

*andop(a,b):*

A utility function to perform and operation between two strings.

*# Function to and two numbers*

**def** andop(a,b):

andstr=''

**for** i **in** range(len(a)):

**if**(a[i]=='0' **or** b[i]=='0'):

andstr+='0'

**else**:

andstr+='1'

**return** andstr

***receiver.py***

This module is responsible for performing the task of the receiver by taking help of the errorchecker module.

*readfile(filename, no\_of\_bits):*

This function reads from the output files generated by sender and splits it into frames and returns a list of frames for later error checking.

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

**def** readfile(filename, no\_of\_bits):

*# Open the file for reading*

**print**('Reading file '+filename)

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)]

*# Printing the frames*

**print**('Codeword frames received:')

**print**(list\_of\_frames)

*# list\_of\_frames=data.split('\n')*

*# list\_of\_frames=list\_of\_frames[0:-1]*

**return** list\_of\_frames

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

This function checks if there is any error in the list of frames passed as a parameter by using checksum. The checksum method of the errorchecker module is used to calculate the checksum and if it returns zero there is no error.

*# 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):

*# Case of no error extract dataword*

**print**('No error in data detected by checksum')

**print**('Dataword frames are')

**print**(list\_of\_frames[0:-1])

**else**:

**print**('\*\*\* Error detected by checksum')

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

This function checks if there is any error in the list of frames passed as a parameter by using LRC. The lrc method of the errorchecker module is used to calculate the LRC value and if it returns zero there is no error.

*# 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**(int(lrcval,2)==0):

**print**('No error in data detected by LRC')

**print**('Dataword frames are')

**print**(list\_of\_frames[0:-1])

**else**:

**print**('\*\*\* Error detected by LRC')

*check\_vrc(list\_of\_frames):*

This function checks if there is any error in the list of frames passed as a parameter by using VRC. The parity of 1 for every frame is checked and if it 0 then there is no error in that frame else there is an error in the frame.

*# 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 detected in frame '+str(i+1)+' by VRC')

flag=False

**if**(flag):

*# No error extract dataword*

**print**("No error detected in data 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)

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

This function checks if there is any error in the list of frames passed as a parameter by using CRC. For every frame the ***modulo2div***() method of errorchecker module is applied with the generator polynomial passed as an argument. If the remainder is 0 then there is no error in the frame else there is an error in the frame.

*combiner():*

Used as a wrapper method for calling all the functions.

*# Function which combines all module*

**def** combiner():

no\_of\_bits=err.no\_of\_bits

**print**('===========================================================')

list\_of\_frames=readfile('csum\_op.txt',no\_of\_bits+len(err.generator\_poly)-1)

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

list\_of\_frames=readfile('lrc\_op.txt',no\_of\_bits+len(err.generator\_poly)-1)

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

list\_of\_frames=readfile('vrc\_op.txt',no\_of\_bits+len(err.generator\_poly)-1)

check\_vrc(list\_of\_frames)

list\_of\_frames=readfile('crc\_op.txt',no\_of\_bits=no\_of\_bits+len(err.generator\_poly)-1)

check\_crc(list\_of\_frames,generator=err.generator\_poly)

***main.py***

This module combines all the above modules and gives a suitable output for all the cases.

**import** **sender** **as** **se**

**import** **receiver** **as** **re**

**import** **errorchecker** **as** **err**

*# Main module to show all 3 cases*

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

list\_of\_frames=(se.readfile('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, error\_list\_frames=[0, 1], error\_bit\_list=[[6], [4]])

re.combiner()

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

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

list\_of\_frames=(se.readfile('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, error\_list\_frames=[0], error\_bit\_list=[[0, 4, 7]])

re.combiner()

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

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

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

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

se.dataword\_to\_codeword(list\_of\_frames,no\_of\_bits=err.no\_of\_bits, error\_list\_frames=[1], error\_bit\_list=[[0, 4, 7]])

re.combiner()

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

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

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

**print**('Case4: Error detected by VRC not by LRC')

se.dataword\_to\_codeword(list\_of\_frames,no\_of\_bits=err.no\_of\_bits, error\_list\_frames=[1,2], error\_bit\_list=[[0],[0]])

re.combiner()

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

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

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

**print**('Case5: Error detected by LRC not by VRC')

se.dataword\_to\_codeword(list\_of\_frames,no\_of\_bits=err.no\_of\_bits, error\_list\_frames=[1], error\_bit\_list=[[0, 4]])

re.combiner()

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

**TEST CASES**

For the 3 cases errors have been inserted manually in order to get the required output. The input file used consists of the following sequence of 0s and 1s

**Input:** 100100011100111100111100101010111100111011010001

With a frame size of 8 bits the given input file has a total of 6 frames.

For the different cases errors have been introduced in different bit positions (0 based indexing from left hand side) as described below

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

Error introduced in:

Frame 0; Bit position 6

Frame 1; Bit position 4

Output:

|  |
| --- |
|  |

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

Error introduced in:

Frame 0; Bit positions 0,4,7

Output:

|  |
| --- |
|  |

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

Error introduced in:

Frame 0; Bit positions 0,4,7

Output:

|  |
| --- |
|  |

**Case 4: Error is detected by VRC but not by LRC.**

Error introduced in:

Frame 1; Bit position 0

Frame 2; Bit position 0

Output:

|  |
| --- |
|  |

**Case 5: Error is detected by LRC but not by VRC.**

Error introduced in:

Frame 1; Bit positions 0, 4

Output:

|  |
| --- |
|  |

**RESULTS**

The performance metric for the evaluation of the above methods is robustness i.e., how well can an algorithm detect errors. For robustness the algorithms were run 10 times with random error injection. Out of the 10 times, CRC failed to detect the error only once, checksum failed about 3 times, LRC about 3 times and VRC about 3 times. For about 4 runs all 4 schemes could detect the error. Thus, we can say CRC is very robust compared to others as the polynomial used has no common factors. Table 1 shows a summary of the results.

|  |  |
| --- | --- |
| **Algorithm** | **Number of times error detected** |
| CRC | 9 |
| LRC | 7 |
| VRC | 7 |
| Checksum | 7 |

**ANALYSIS**

Overall the implementation of the assignment is more or less correct. Some possible bugs can arise due to the assumption that the input size is a multiple of the frame size. However, this can easily be overcome by padding the last frame of the input data with 0s so that it is a multiple of the frame size. Also, the program may be modified to work on inter process communication or communication over an unreliable network.

**COMMENTS**

Overall the lab assignment was a great learning experience as we got to implement the well-known error detection algorithms ourselves. The assignment can be rated as moderately difficult.