CRC 16-CCITT: Software Implementation and Testing

By Angel Solis, CPE 405

Reliability is a must in modern technology. From cars to phones to the internet, all things are expected to work on demand. So, when transmissions, be they wired, or wireless are corrupted, how can we tell? This is where redundancy systems like CRC are used to detect errors. CRC is essentially doing long division on the message to be transmitted with a specific devisor. The resulting remainder is the CRC code that is appended to the message for transition. The receiving side then does the same long division on the message with the appended CRC code. If the remainder is zero, then no errors were detected.

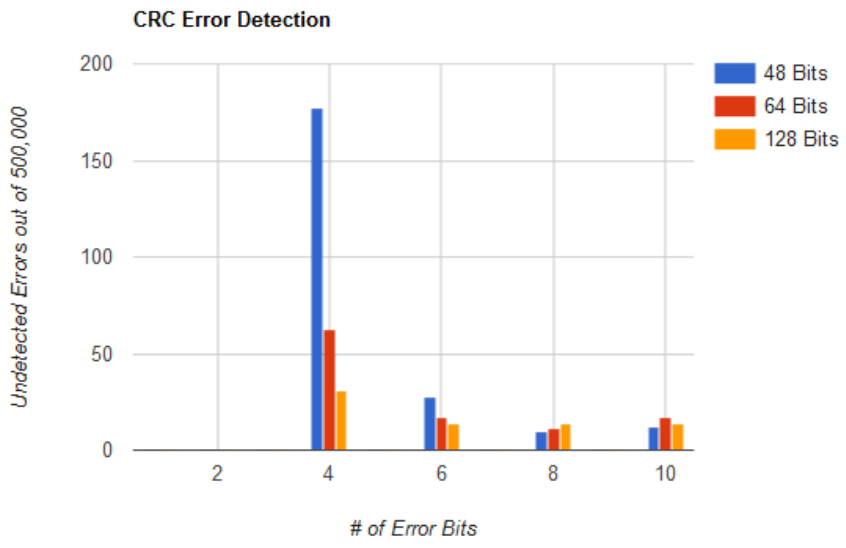
The CRC software implementation was written in python. The modules consist of a CRC generator which will output the remainder. A CRC decode which given the message and remainder will return the syndrome. A random string generator which uses a pool of all capital and lowercase letters as well as all digits to pull from. This algorithm is non-replacing meaning that a taken letter will not appear again. This ensures that each string is unique. The last module is part of the code that calls the other modules and automates the testing process, 500,000 messages were generated for each test.

The CRC generation code uses the generator polynomial 0x1021 which was chosen as one it is one version of CRC 16-CCITT [1]. The initial value used for the CRC buffer was all zeros. The algorithm then runs through the message byte by byte. If the current bit is one an XOR operation will take place after the shift. The CRC buffer is then shifted once to the left. If the current byte and current bit are both one, then one is added to the CRC buffer. This simulates shifting a one into the buffer. This loop continues until the end of the message. Then 16 zeros are

appended to the message, this is done to fully flush out the buffer. The resulting CRC values is then ANDed with 0xFFFF to mask off the upper bits.

The CRC decode follows a similar approach to the algorithm written above. The differences are that an existing CRC is expected. The appending process takes place inside this algorithm. The appending is done though a series of type conversion to change the numerical CRC into a hex string that must then have the “0x” removed. This is finally appended to the message and the same division process takes place with the exception of appending the 16 zeros. This no longer needs to be done as the old CRC is appended instead.

The code for testing the CRC ended up being rather complicated. Because random errors needed to be introduced this meant that I needed to do three version of the code: one for single random errors, one for burst errors, and one for errors in the CRC transmission. The problem with this is that the XOR operation is not supported for strings. This meant that to apply the error I had to convert the string into bytes and then XOR it with a concatenation of all randomly selected values. This then had to be rebuilt into a string again to be sent for testing. The burst errors were created by randomly selecting one bit in the string and then corrupting bits next to it until the desired length of corrupted bits is reached. The CRC errors were created with a similar approach to the single random



errors except that the original CRC was corrupted instead.

Figure 1

The results of the single error test are summarized in figure 1. Any numbers of errors generated below 4 are always caught. At 4 however errors begin to seep through. This gives us our experimental Hamming distance of 4. While all message lengths had errors, it seems the longer the message the fewer errors go undetected. In the worst case which was 4 error bits with a 48-bit message, 99.97% of the messages were found. The errors generated in the CRC code were all detected. This is because adding anything, but the correct remainder will never result in zero when dividing. For the burst errors all errors were once again found. This result surprised me as I never expected CRC to be more resilient toward burst errors rather than random errors. However, the university of Hawaii did a similar study and they too found that burst errors are more easily detected [2]. This is because CRC was designed to detect burst errors as they are the most detrimental to data transmission. In conclusion CRC is extremely resilient to burst errors but its weakness lies in single random errors if they occur to frequently.

[1] http://srecord.sourceforge.net/crc16-ccitt.html

[2] http://www2.hawaii.edu/~tmandel/papers/CRCBurst.pdf

Code (Python 2.7 Used):

Random Errors:

```
import  datetime, numpy, random, string,binascii

# Generates the CRC16- CCITT remainder
def crc16(data):
```

```
    data = bytearray(data)                          # convert the data to an
array of bytes
    poly = 0x1021                                   # used for CCITT-16
    #crc = 0xFFFF                                   # crc should start with
all ones in some standards
    #crc = 0x1D0F                                   # CRC value when not
appending zeros
    crc = 0x0000                                    # CRC value when using
XModem
    mask = [0x80,0x40,0x20,0x10,                    # set of possible current
bit values
            0x08,0x04,0x02,0x01]
    for curr_byte in data:                          # go though all the data

        for curr_bit in mask:                       # go though all 8 bits,
curr_bit is current bit
            if (crc & 0x8000):                      # grab only first bit if
1
                xor = 1                             # set xor flag for later
            else:                                   # if not 1
                xor = 0                             # set xor flag for later

            crc<<=1                                 # shift left crc

            if (curr_byte & curr_bit ):             # if current byte &&
current bit
                crc+=1                              # append 1
            if (xor):
                crc^=poly                           # if flag xor crc with
polynomial

    for curr_byte in xrange(0,16):
        if (crc & 0x8000):                          # if first bit is 1
            xor = 1                                 # set xor flag
        else:
            xor = 0                                 # else clear xor flag
        crc <<=1                                    # shift crc one left
        if xor:                                     # if xor flag was set
            crc ^= poly                             # xor crc and poly

    crc = crc & 0xFFFF                              # mask all but last two
bytes

    return crc                                      # send remainder
polynomial

# Check if given remainder matches expected remainder
def crc16_check(data,code):
    data = bytearray(data)                          # convert the data to an
array of bytes
    tempstr = hex(code)[2:]                         # create string from code
remove 0x
    if (tempstr[-1] == 'L'):                        # if python decided to
add an L at the end remove it
        tempstr = tempstr[0:-1]
    if (len(tempstr)<4):                            # if the CRC Hex value is
less than 4 digits pad with zeros
```

```
        while(len(tempstr)<4):                          # pad with zeros until
desired length
            tempstr = "0" + tempstr

    for i in xrange(0,2):                               # get both bytes
        data.append(int(tempstr[i*2:2+i*2],16))         # append and convert the
string value into an int

    poly = 0x1021                                       # used for CCITT-16
    #crc = 0xFFFF                                       # crc should start with
all ones in some standards
    #crc = 0x1D0F                                       # CRC value when not
appending zeros
    crc = 0x0000                                        # CRC value when using
XModem
    mask = [0x80,0x40,0x20,0x10,                        # set of possible current
bit values
            0x08,0x04,0x02,0x01]
    for curr_byte in data:                              # go though all the data
        for curr_bit in mask:                           # go though all 8 bits,
curr_bit is current bit
            if (crc & 0x8000):                          # grab only first bit if
1
                xor = 1                                 # set xor flag for later
            else:                                       # if not 1
                xor = 0                                 # set xor flag for later

            crc<<=1                                     # shift left crc

            if (curr_byte & curr_bit ):                 # if current byte &&
current bit
                crc+=1                                  # append 1
            if (xor):
                crc^=poly                               # if flag xor crc with
polynomial
    crc = crc & 0xFFFF                                # mask all but last two bytes
    #print crc                                          # skip the added zeros
part
    return crc                                          # send remainder
polynomial

# Generate Random array of strings without repetition
def rand_str(count, size=8, chars=string.ascii_uppercase +
string.ascii_lowercase + string.digits ):
    limit = len(chars) ** size - 1                      # selected limit length
    start = 0                                           # number so far generated
    choices = []                                        # list for chosen strings
    for i in range(0,count):                            # run until desired
number of strings generated
        start = random.randint(start, start + (limit-start) // (count-i))
        digits = []
        temp = start
        while len(digits) < size:                       # whlie string not long
enough
            temp, i = divmod(temp, len(chars))
            digits.append(chars[i])
        choices.append(''.join(digits))
```

```
        start += 1
    return choices                                      # return list of strings
def errors_not_detected(count_str,len_str,no_err):

    err_list = []                                       # List of Possible
errors
    for i in range (0,8*len_str):                       # generate errors
change to incorperate all lengths
        if (len_str ==6):
            if   (i<1*8):
                err_list.append(bytearray([0,0,0,0,0,1<<i%8]))
            elif (i<2*8):
                err_list.append(bytearray([0,0,0,0,1<<i%8,0]))
            elif (i<3*8):
                err_list.append(bytearray([0,0,0,1<<i%8,0,0]))
            elif (i<4*8):
                err_list.append(bytearray([0,0,1<<i%8,0,0,0]))
            elif (i<5*8):
                err_list.append(bytearray([0,1<<i%8,0,0,0,0]))
            elif (i<6*8):
                err_list.append(bytearray([1<<i%8,0,0,0,0,0]))

        elif (len_str == 8):
            if   (i<1*8):
                err_list.append(bytearray([0,0,0,0,0,0,0,1<<i%8]))
            elif (i<2*8):
                err_list.append(bytearray([0,0,0,0,0,0,1<<i%8,0]))
            elif (i<3*8):
                err_list.append(bytearray([0,0,0,0,0,1<<i%8,0,0]))
            elif (i<4*8):
                err_list.append(bytearray([0,0,0,0,1<<i%8,0,0,0]))
            elif (i<5*8):
                err_list.append(bytearray([0,0,0,1<<i%8,0,0,0,0]))
            elif (i<6*8):
                err_list.append(bytearray([0,0,1<<i%8,0,0,0,0,0]))
            elif (i<7*8):
                err_list.append(bytearray([0,1<<i%8,0,0,0,0,0,0]))
            elif (i<8*8):
                err_list.append(bytearray([1<<i%8,0,0,0,0,0,0,0]))

        elif (len_str == 16 ):
            if   (i<1*8):

err_list.append(bytearray([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1<<i%8]))
            elif (i<2*8):

err_list.append(bytearray([0,0,0,0,0,0,0,0,0,0,0,0,0,0,1<<i%8,0]))
            elif (i<3*8):

err_list.append(bytearray([0,0,0,0,0,0,0,0,0,0,0,0,0,1<<i%8,0,0]))
            elif (i<4*8):

err_list.append(bytearray([0,0,0,0,0,0,0,0,0,0,0,0,1<<i%8,0,0,0]))
            elif (i<5*8):

err_list.append(bytearray([0,0,0,0,0,0,0,0,0,0,0,1<<i%8,0,0,0,0]))
            elif (i<6*8):
```

```
err_list.append(bytearray([0,0,0,0,0,0,0,0,0,1<<i%8,0,0,0,0,0]))
            elif (i<7*8):

err_list.append(bytearray([0,0,0,0,0,0,0,0,1<<i%8,0,0,0,0,0,0]))
            elif (i<8*8):

err_list.append(bytearray([0,0,0,0,0,0,0,0,1<<i%8,0,0,0,0,0,0]))
            elif (i<9*8):

err_list.append(bytearray([0,0,0,0,0,0,0,1<<i%8,0,0,0,0,0,0,0]))
            elif (i<10*8):

err_list.append(bytearray([0,0,0,0,0,0,1<<i%8,0,0,0,0,0,0,0,0]))
            elif (i<11*8):

err_list.append(bytearray([0,0,0,0,0,1<<i%8,0,0,0,0,0,0,0,0,0]))
            elif (i<12*8):

err_list.append(bytearray([0,0,0,0,1<<i%8,0,0,0,0,0,0,0,0,0,0]))
            elif (i<13*8):

err_list.append(bytearray([0,0,0,1<<i%8,0,0,0,0,0,0,0,0,0,0,0]))
            elif (i<14*8):

err_list.append(bytearray([0,0,1<<i%8,0,0,0,0,0,0,0,0,0,0,0,0]))
            elif (i<15*8):

err_list.append(bytearray([0,1<<i%8,0,0,0,0,0,0,0,0,0,0,0,0,0]))
            elif (i<16*8):

err_list.append(bytearray([1<<i%8,0,0,0,0,0,0,0,0,0,0,0,0,0,0]))
        #print binascii.hexlify(err_list[i])
    # begin generating error strings-----------------------------------------
-------
    start =datetime.datetime.now()                      # used to measure
delta time
    err_count = 0                                       # used to count the
amount of errors not detected
    code = rand_str(count_str,len_str)                  # generate random
strings
    result =[]                                          # Correct CRCs
    check =[]                                           # output from
rechecking crc
    random_err =[]                                      # save random errors

    for i in range (0,count_str):
        result.append(crc16(code[i]))                   # append results of
code
        temp = random.sample(err_list,no_err)           # create temporary
list of error bits
        random_err.append (code[i])                     # make error and
correct value the same
        if (no_err>0):                                  # if errors are
supposed to occour corrupt values
            for k in range (0,no_err):                  # for number of
desired errors
```

```
                random_err[i] =''.join(chr(ord(a) ^ ord(b)) for a,b in
zip(random_err[i],str(temp[k])))
                                                        # xor binary of
characters and then reform stirngs
        check.append(0)
        check[i]=(crc16_check(random_err[i],result[i]))
        if (check[i] == 0 ):
            err_count+=1

            print "{3}: text: {0}  text w/ err: {4}  CRC16: {1} Error on:
{2}\n".format(code[i],

hex(result[i])[2:-1].rjust(4,'0'),

hex(check[i])[2:-1].rjust(4,'0'),

format(i+1,'06'),

random_err[i],)
            for k in range (0,no_err):                      # for number of
desired errors
                print "{0}".format( binascii.hexlify(temp[k]))

    stop = datetime.datetime.now()
    print "{0}/{1} errors were not detected".format(err_count,count_str)
    delta = stop-start
    print "Runtime = {0}\n".format(delta)



# constants ------------------------------------------------------------------
----------
#count_str = 15                                     # Number of random
strings
count_str = 500000                                  # Number of random
strings
len_str = 6                                         # Length of the strings
'''
for i in range (1,11):                              # Simulate and test for
Strings length 6
    print "Number of errors generated: {0} String lengths: {1}".format(i,
len_str)
    errors_not_detected(count_str,len_str,i)

len_str = 8
for i in range (1,11):                              # Simulate and test for
Strings length 8
    print "Number of errors generated: {0} String lengths: {1}".format(i,
len_str)
    errors_not_detected(count_str,len_str,i)
'''
len_str = 16
for i in range (10,11):                             # Simulate and test for
Strings length 16
    print "Number of errors generated: {0} String lengths: {1}".format(i,
len_str)
    errors_not_detected(count_str,len_str,i)
```

Burst Errors:

```
import  datetime, numpy, random, string,binascii

# Generates the CRC16- CCITT remainder
def crc16(data):
    data = bytearray(data)                          # convert the data to an
array of bytes
    poly = 0x1021                                   # used for CCITT-16
    #crc = 0xFFFF                                   # crc should start with
all ones in some standards
    #crc = 0x1D0F                                   # CRC value when not
appending zeros
    crc = 0x0000                                    # CRC value when using
XModem
    mask = [0x80,0x40,0x20,0x10,                    # set of possible current
bit values
            0x08,0x04,0x02,0x01]
    for curr_byte in data:                          # go though all the data

        for curr_bit in mask:                       # go though all 8 bits,
curr_bit is current bit
            if (crc & 0x8000):                      # grab only first bit if
1
                xor = 1                             # set xor flag for later
            else:                                   # if not 1
                xor = 0                             # set xor flag for later

            crc<<=1                                 # shift left crc

            if (curr_byte & curr_bit ):             # if current byte &&
current bit
                crc+=1                              # append 1
            if (xor):
                crc^=poly                           # if flag xor crc with
polynomial

    for curr_byte in xrange(0,16):
        if (crc & 0x8000):                          # if first bit is 1
            xor = 1                                 # set xor flag
        else:
            xor = 0                                 # else clear xor flag
        crc <<=1                                    # shift crc one left
        if xor:                                     # if xor flag was set
            crc ^= poly                             # xor crc and poly

    crc = crc & 0xFFFF                              # mask all but last two
bytes

    return crc                                      # send remainder
polynomial

# Check if given remainder matches expected remainder
def crc16_check(data,code):
```

```
    data = bytearray(data)                          # convert the data to an
array of bytes
    tempstr = hex(code)[2:]                         # create string from code
remove 0x
    if (tempstr[-1] == 'L'):                        # if python decided to
add an L at the end remove it
        tempstr = tempstr[0:-1]
    if (len(tempstr)<4):                            # if the CRC Hex value is
less than 4 digits pad with zeros
        while(len(tempstr)<4):                      # pad with zeros until
desired length
            tempstr = "0" + tempstr

    for i in xrange(0,2):                           # get both bytes
        data.append(int(tempstr[i*2:2+i*2],16))     # append and convert the
string value into an int

    poly = 0x1021                                   # used for CCITT-16
    #crc = 0xFFFF                                   # crc should start with
all ones in some standards
    #crc = 0x1D0F                                   # CRC value when not
appending zeros
    crc = 0x0000                                    # CRC value when using
XModem
    mask = [0x80,0x40,0x20,0x10,                    # set of possible current
bit values
            0x08,0x04,0x02,0x01]
    for curr_byte in data:                          # go though all the data
        for curr_bit in mask:                       # go though all 8 bits,
curr_bit is current bit
            if (crc & 0x8000):                      # grab only first bit if
1
                xor = 1                             # set xor flag for later
            else:                                   # if not 1
                xor = 0                             # set xor flag for later

            crc<<=1                                 # shift left crc

            if (curr_byte & curr_bit ):             # if current byte &&
current bit
                crc+=1                              # append 1
            if (xor):
                crc^=poly                           # if flag xor crc with
polynomial
    crc = crc & 0xFFFF                          # mask all but last two bytes
    #print crc                                      # skip the added zeros
part
    return crc                                      # send remainder
polynomial

# Generate Random array of strings without repetition
def rand_str(count, size=8, chars=string.ascii_uppercase +
string.ascii_lowercase + string.digits ):
    limit = len(chars) ** size - 1                  # selected limit length
    start = 0                                       # number so far generated
    choices = []                                    # list for chosen strings
```

```
    for i in range(0,count):                            # run until desired
number of strings generated
        start = random.randint(start, start + (limit-start) // (count-i))
        digits = []
        temp = start
        while len(digits) < size:                       # whlie string not long
enough
            temp, i = divmod(temp, len(chars))
            digits.append(chars[i])
        choices.append(''.join(digits))
        start += 1
    return choices                                      # return list of strings
def errors_not_detected(count_str,len_str,no_err):

    err_list = []                                           # List of Possible
errors
    for i in range (0,8*len_str):                           # generate errors
change to incorperate all lengths
        if (len_str ==6):
            if    (i<1*8):
                err_list.append(bytearray([0,0,0,0,0,1<<i%8]))
            elif (i<2*8):
                err_list.append(bytearray([0,0,0,0,1<<i%8,0]))
            elif (i<3*8):
                err_list.append(bytearray([0,0,0,1<<i%8,0,0]))
            elif (i<4*8):
                err_list.append(bytearray([0,0,1<<i%8,0,0,0]))
            elif (i<5*8):
                err_list.append(bytearray([0,1<<i%8,0,0,0,0]))
            elif (i<6*8):
                err_list.append(bytearray([1<<i%8,0,0,0,0,0]))

        elif (len_str == 8):
            if    (i<1*8):
                err_list.append(bytearray([0,0,0,0,0,0,0,1<<i%8]))
            elif (i<2*8):
                err_list.append(bytearray([0,0,0,0,0,0,1<<i%8,0]))
            elif (i<3*8):
                err_list.append(bytearray([0,0,0,0,0,1<<i%8,0,0]))
            elif (i<4*8):
                err_list.append(bytearray([0,0,0,0,1<<i%8,0,0,0]))
            elif (i<5*8):
                err_list.append(bytearray([0,0,0,1<<i%8,0,0,0,0]))
            elif (i<6*8):
                err_list.append(bytearray([0,0,1<<i%8,0,0,0,0,0]))
            elif (i<7*8):
                err_list.append(bytearray([0,1<<i%8,0,0,0,0,0,0]))
            elif (i<8*8):
                err_list.append(bytearray([1<<i%8,0,0,0,0,0,0,0]))

        elif (len_str == 16 ):
            if    (i<1*8):

err_list.append(bytearray([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1<<i%8]))
            elif (i<2*8):

err_list.append(bytearray([0,0,0,0,0,0,0,0,0,0,0,0,0,0,1<<i%8,0]))
```

```
            elif (i<3*8):
err_list.append(bytearray([0,0,0,0,0,0,0,0,0,0,0,0,0,1<<i%8,0,0]))
            elif (i<4*8):
err_list.append(bytearray([0,0,0,0,0,0,0,0,0,0,0,0,1<<i%8,0,0,0]))
            elif (i<5*8):
err_list.append(bytearray([0,0,0,0,0,0,0,0,0,0,0,1<<i%8,0,0,0,0]))
            elif (i<6*8):
err_list.append(bytearray([0,0,0,0,0,0,0,0,0,0,1<<i%8,0,0,0,0,0]))
            elif (i<7*8):
err_list.append(bytearray([0,0,0,0,0,0,0,0,0,1<<i%8,0,0,0,0,0,0]))
            elif (i<8*8):
err_list.append(bytearray([0,0,0,0,0,0,0,0,1<<i%8,0,0,0,0,0,0,0]))
            elif (i<9*8):
err_list.append(bytearray([0,0,0,0,0,0,0,1<<i%8,0,0,0,0,0,0,0,0]))
            elif (i<10*8):
err_list.append(bytearray([0,0,0,0,0,0,1<<i%8,0,0,0,0,0,0,0,0,0]))
            elif (i<11*8):
err_list.append(bytearray([0,0,0,0,0,1<<i%8,0,0,0,0,0,0,0,0,0,0]))
            elif (i<12*8):
err_list.append(bytearray([0,0,0,0,1<<i%8,0,0,0,0,0,0,0,0,0,0,0]))
            elif (i<13*8):
err_list.append(bytearray([0,0,0,1<<i%8,0,0,0,0,0,0,0,0,0,0,0,0]))
            elif (i<14*8):
err_list.append(bytearray([0,0,1<<i%8,0,0,0,0,0,0,0,0,0,0,0,0,0]))
            elif (i<15*8):
err_list.append(bytearray([0,1<<i%8,0,0,0,0,0,0,0,0,0,0,0,0,0,0]))
            elif (i<16*8):
err_list.append(bytearray([1<<i%8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]))
        #print binascii.hexlify(err_list[i])
    # begin generating error strings-----------------------------------------
-------
    start =datetime.datetime.now()                          # used to measure
delta time
    err_count = 0                                           # used to count the
amount of errors not detected
    code = rand_str(count_str,len_str)                      # generate random
strings
    result =[]                                              # Correct CRCs
    check =[]                                               # output from
rechecking crc
    random_err =[]                                          # save random errors

    for i in range (0,count_str):
```

```
        result.append(crc16(code[i]))                    # append results of
code


        #temp = random.sample(err_list,no_err)           # create temporary
list of error bits
        temp = [];

        m = random.randint(0,len(err_list)-no_err)       # pick a random error
        for k in range (0,no_err):                       # create need number
of burst errors
            temp.append(err_list[m+k])                   # append errors


        random_err.append (code[i])                      # make error and
correct value the same

        if (no_err>0):                                   # if errors are
supposed to occour corrupt values
            for k in range (0,no_err):                   # for number of
desired errors
                random_err[i] =''.join(chr(ord(a) ^ ord(b)) for a,b in
zip(random_err[i],str(temp[k])))
                                                         # xor binary of
characters and then reform stirngs
        check.append(0)

        check[i]=(crc16_check(random_err[i],result[i]))
        if (check[i] == 0 ):
            err_count+=1

            print "{3}: text: {0}  text w/ err: {4}  CRC16: {1} Error
on:{2}\n".format(code[i],

hex(result[i])[2:-1].rjust(4,'0'),

hex(check[i])[2:-1].rjust(4,'0'),

format(i+1,'06'),

random_err[i],)
            for k in range (0,no_err):                        # for number of
desired errors
                print "{0}".format( binascii.hexlify(temp[k]))

    stop = datetime.datetime.now()
    print "{0}/{1} errors were not detected".format(err_count,count_str)
    delta = stop-start
    print "Runtime = {0}\n".format(delta)



# constants ---------------------------------------------------------------
----------
#count_str = 15                                  # Number of random
strings
```

```
count_str = 500000                                  # Number of random
strings
len_str = 6                                         # Length of the strings

for i in range (1,11):                              # Simulate and test for
Strings length 6
    print "Number of errors generated: {0} String lengths: {1}".format(i,
len_str)
    errors_not_detected(count_str,len_str,i)

len_str = 8
for i in range (1,11):                              # Simulate and test for
Strings length 8
    print "Number of errors generated: {0} String lengths: {1}".format(i,
len_str)
    errors_not_detected(count_str,len_str,i)

len_str = 16
for i in range (1,11):                              # Simulate and test for
Strings length 16
    print "Number of errors generated: {0} String lengths: {1}".format(i,
len_str)
    errors_not_detected(count_str,len_str,i)
```

CRC Errors:

```
import  datetime, numpy, random, string,binascii

# Generates the CRC16- CCITT remainder
def crc16(data):
    data = bytearray(data)                          # convert the data to an
array of bytes
    poly = 0x1021                                   # used for CCITT-16
    #crc = 0xFFFF                                   # crc should start with
all ones in some standards
    #crc = 0x1D0F                                   # CRC value when not
appending zeros
    crc = 0x0000                                    # CRC value when using
XModem
    mask = [0x80,0x40,0x20,0x10,                    # set of possible current
bit values
            0x08,0x04,0x02,0x01]
    for curr_byte in data:                          # go though all the data

        for curr_bit in mask:                       # go though all 8 bits,
curr_bit is current bit
            if (crc & 0x8000):                      # grab only first bit if
1
                xor = 1                             # set xor flag for later
            else:                                   # if not 1
                xor = 0                             # set xor flag for later

            crc<<=1                                 # shift left crc

            if (curr_byte & curr_bit ):             # if current byte &&
current bit
                crc+=1                              # append 1
            if (xor):
```

```
                crc^=poly                                   # if flag xor crc with
polynomial

    for curr_byte in xrange(0,16):
        if (crc & 0x8000):                                  # if first bit is 1
            xor = 1                                         # set xor flag
        else:
            xor = 0                                         # else clear xor flag
        crc <<=1                                            # shift crc one left
        if xor:                                             # if xor flag was set
            crc ^= poly                                     # xor crc and poly

    crc = crc & 0xFFFF                                      # mask all but last two
bytes

    return crc                                              # send remainder
polynomial

# Check if given remainder matches expected remainder
def crc16_check(data,code):
    data = bytearray(data)                                  # convert the data to an
array of bytes
    tempstr = hex(code)[2:]                                 # create string from code
remove 0x
    if (tempstr[-1] == 'L'):                                # if python decided to
add an L at the end remove it
        tempstr = tempstr[0:-1]
    if (len(tempstr)<4):                                    # if the CRC Hex value is
less than 4 digits pad with zeros
        while(len(tempstr)<4):                              # pad with zeros until
desired length
            tempstr = "0" + tempstr

    for i in xrange(0,2):                                   # get both bytes
        data.append(int(tempstr[i*2:2+i*2],16))             # append and convert the
string value into an int

    poly = 0x1021                                           # used for CCITT-16
    #crc = 0xFFFF                                           # crc should start with
all ones in some standards
    #crc = 0x1D0F                                           # CRC value when not
appending zeros
    crc = 0x0000                                            # CRC value when using
XModem
    mask = [0x80,0x40,0x20,0x10,                            # set of possible current
bit values
            0x08,0x04,0x02,0x01]
    for curr_byte in data:                                  # go though all the data
        for curr_bit in mask:                               # go though all 8 bits,
curr_bit is current bit
            if (crc & 0x8000):                              # grab only first bit if
1
                xor = 1                                     # set xor flag for later
            else:                                           # if not 1
                xor = 0                                     # set xor flag for later

            crc<<=1                                         # shift left crc
```

```
            if (curr_byte & curr_bit ):              # if current byte &&
current bit
                crc+=1                               # append 1
            if (xor):
                crc^=poly                            # if flag xor crc with
polynomial
    crc = crc & 0xFFFF                             # mask all but last two bytes
    #print crc                                       # skip the added zeros
part
    return crc                                       # send remainder
polynomial

# Generate Random array of strings without repetition
def rand_str(count, size=8, chars=string.ascii_uppercase +
string.ascii_lowercase + string.digits ):
    limit = len(chars) ** size - 1                   # selected limit length
    start = 0                                        # number so far generated
    choices = []                                     # list for chosen strings
    for i in range(0,count):                         # run until desired
number of strings generated
        start = random.randint(start, start + (limit-start) // (count-i))
        digits = []
        temp = start
        while len(digits) < size:                    # whlie string not long
enough
            temp, i = divmod(temp, len(chars))
            digits.append(chars[i])
        choices.append(''.join(digits))
        start += 1
    return choices                                   # return list of strings
def errors_not_detected(count_str,len_str,no_err):

    err_list = []                                        # List of Possible
errors
    for i in range (1,16):                         # generate errors  change to
incorperate all lengths
        err_list.append(1<<i)

    start =datetime.datetime.now()                       # used to measure
delta time
    err_count = 0                                        # used to count the
amount of errors not detected
    code = rand_str(count_str,len_str)                   # generate random
strings
    result =[]                                           # Correct CRCs
    check =[]                                            # output from
rechecking crc
    random_err =[]                                       # save random errors

    for i in range (0,count_str):
        result.append(crc16(code[i]))                    # append results of
code
        temp = random.sample(err_list,no_err)            # create temporary
list of error bits
        random_err.append (0)                            # make error have a
value
```

```
        if (no_err>0):                                  # if errors are
supposed to occour corrupt values
            for k in range (0,no_err):                  # for number of
desired errors
                random_err[i] |= temp[k]                # or all errors
together
        check.append(0)
        check[i]=(crc16_check(code[i],result[i] ^ random_err[i]))
        if (check[i] == 0 ):
            err_count+=1
            print "{3}: text: {0}  CRC16: {1}  CRC16 w/ err: {4} Output:  {2}
Error on:\n".format(code[i],

hex(result[i])[2:-1].rjust(4,'0'),

hex(check[i])[2:-1].rjust(4,'0'),

format(i+1,'06'),

hex(result[i] ^ random_err[i])[2:-1].rjust(4,'0'))
            for k in range (0,no_err):                  # for number of
desired errors
                    print "{0}".format(hex(temp[k])[2:-1].rjust(4,'0'))
        '''
        print "{3}: text: {0}  CRC16: {1}  CRC16 w/ err: {4} Output:  {2}
Error on:\n".format(code[i],

hex(result[i])[2:-1].rjust(4,'0'),

hex(check[i])[2:-1].rjust(4,'0'),

format(i+1,'06'),

hex(result[i] ^ random_err[i])[2:-1].rjust(4,'0'))
        for k in range (0,no_err):                      # for number of
desired errors
                print "{0}".format(hex(temp[k])[2:-1].rjust(4,'0'))
        '''
    stop = datetime.datetime.now()
    print "{0}/{1} errors were not detected".format(err_count,count_str)
    delta = stop-start
    print "Runtime = {0}\n".format(delta)



# constants ------------------------------------------------------------------
----------
#count_str = 15                                         # Number of random
strings
count_str = 500000                                      # Number of random
strings
len_str = 6                                             # Length of the strings

for i in range (1,11):                                  # Simulate and test for
Strings length 6
    print "Number of errors generated: {0} String lengths: {1}".format(i,
len_str)
```

```
    errors_not_detected(count_str,len_str,i)

len_str = 8
for i in range (1,11):                              # Simulate and test for
Strings length 8
    print "Number of errors generated: {0} String lengths: {1}".format(i,
len_str)
    errors_not_detected(count_str,len_str,i)

len_str = 16
for i in range (1,11):                              # Simulate and test for
Strings length 16
    print "Number of errors generated: {0} String lengths: {1}".format(i,
len_str)
    errors_not_detected(count_str,len_str,i)
```