COIS 4310H – Networking
Assignment #1
Simon Willshire (0491272)

1.

$f(t) = t, \{0 \leq t \leq 1\}$

$a_0 = \frac{1}{2L}\int_{-L}^{L} f(x)dx = \frac{1}{2}\int_{-1}^{1} tdt$
$= \frac{1}{2}\left(\frac{t^2}{2}\right)|_{-1}^{1} = \frac{1}{2}$

$a_n = \frac{1}{L}\int_{-L}^{L} f(t)\cos\left(\frac{n\pi t}{L}\right)dt$
$= \int_0^1 tcos\left(\frac{n\pi t}{1}\right)dt$ Aside: $u = t, dv = \cos(n\pi t) = \frac{-\sin(n\pi t)}{n\pi}$
$= udv = uv - \int_0^1 vdu$
$= t\left(\frac{-\sin(n\pi t)}{n\pi}\right)|_0^1 - \int_0^1 \left(\frac{-\sin(n\pi t)}{n\pi}\right)$
$= t\left(\frac{-\sin(n\pi t)}{n\pi}\right)|_0^1 + \frac{1}{n\pi}(\cos(n\pi t))|_0^1$
$= \frac{-\sin(n\pi)}{n\pi} + \frac{\cos(n\pi)}{n\pi} - \frac{1}{n\pi}$
$= \boldsymbol{\frac{\sin(n\pi t)}{n^2\pi^2}}$

$b_n = \frac{1}{L}\int_{-L}^{L} f(t)\sin\left(\frac{n\pi t}{L}\right)dt$
$= \int_0^1 tsin\left(\frac{n\pi t}{1}\right)dt$ Aside: $u = t, dv = \sin(n\pi t) = \frac{\cos(n\pi t)}{n\pi}$
$= udv = uv - \int_0^1 vdu$
$= t\left(\frac{\cos(n\pi t)}{n\pi}\right)|_0^1 - \int_0^1 \left(\frac{\cos(n\pi t)}{n\pi}\right)$
$= t\left(\frac{\cos(n\pi t)}{n\pi}\right)|_0^1 - \frac{1}{n\pi}(\sin(n\pi t))|_0^1$
$= \boldsymbol{\frac{1}{n\pi}(\cos(n\pi t) - \sin(n\pi t))}$

2. TV Channels with 6Mhz width, what is the capable bit/sec rate?

Implement Nyquist Theorem, says that maximum rate:

$= 12\ million\ samples/sec$ (Doubling the Mhz)
$= 2H\log_2 V$ b/sec
$= 2(6Mhz)\log_2(4\ levels)$
$= 12(\log_2 2^2)$
$= 12(2)$
$= 24\ million\ b/sec = \boldsymbol{24Mb/s}$

3. Binary signal sent over 3Khz channel, signal noise ratio = 20dB (factor of 100), the max data rate is the minimum of signal to noise bandwidth (using Shannon's theorem), and Nyquist's theorem used in the previous question.

Shannon's Theorem:

$= H \log_2(1 + 100)$
$= 3 \log_2(101)$
$= 3(6.658211)$
$= 19{,}975\ Kb/s$

Therefore the binary signal sent over 3Khz channel with 20dB noise has a maximum data rate of $19{,}975\ Kb/s$

4. An upper layer packet is split into 10 frames, 20% error rate, how many times must it be resent for the message to come across on average?

$10 * 0.8 = 8$	Frames with damage typically.
$0.8x = 2$	Frames still needing to be sent
$x = \frac{2}{0.8} = 2.5 \cong \mathbf{3}$	Frames to be resent for a total of 13 frames on average with 20% damage

5. Maximum overhead in a byte stuffing algorithm: 100% (or 2x the size of the packet). However this varies depending on the algorithm of choice.

6.

No frame delay, 10ms process input from data, 2x10^8 (200,000Km/s, fibre optic speed), achieve 30fps and 60fps.

Delay of $30fps * 10ms = 300ms$, $700ms$ remain for data transfer
$350ms$ there, $350ms$ back.
$200{,}000km * 0.35 = 70{,}000km/s\ /\ 30fps$
$\cong$ **2,333.334**km distance from host location

Farthest identifiable city assuming the host location was in Peterborough: **Saskatoon Saskatchewan (CA).**

$60fps * 10ms = 600ms$, $400ms$ remain for data transfer
$200ms$ there, $200ms$ back.
$200{,}000km/s * 0.2 = 40{,}000km/s\ /\ 60fps$
$\cong$ **666.667**km distance from host location

Farthest identifiable city assuming the host location was in Peterborough: **Boston, Massachusetts (US)**

7.

Optimize streaming: Average customer is 78ms latency. Calculate how much data is stuck for:

1080p @ 8Mb/s + 5.1 Audio @ 512Kb/s

2160p @ 35Mb/s + 5.1 Audio @ 512Kb/s

Local Traffic: How much data lost in 78ms?

1080p: $0.078s(8Mb/s) + 0.078s(0.512) = 0.624Mb/s + 0.039936Mb/s$

$= \mathbf{0.663936}\boldsymbol{Mb/s}$ loss in transfer from latency.

2160p: $0.078s(35Mb/s) + 0.078(0.512) = 2.73Mb/s + 0.039936Mb/s$

$= \mathbf{2.769936}\boldsymbol{Mb/s}$ loss in transfer from latency.

International Traffic: How much data is lost in 266ms?

1080p: $0.266s(8Mb/s) + 0.266s(0.512) = 2.128Mb/s + 0.136192Mb/s$

$= \mathbf{2.264192}\boldsymbol{Mb/s}$ loss in transfer from latency.

2160p: $0.266s(35Mb/s) + 0.266(0.512) = 9.31Mb/s + 0.136192Mb/s$

$= \mathbf{9.446192}\boldsymbol{Mb/s}$ loss in transfer from latency.

Let's assume we use transfer rates of $0.12 TB USD from 'RackSpace', plug in our lost data rates, and we get:

Local Traffic (78ms @ 1080p): $0.000000663936 * 0.12 = \$0.00000007967232/user$

(78ms @ 2160p): $0.000002769936 * 0.12 = \$0.00000033239232/user$

International Traffic

(206ms @ 1080p): $0.000002264192 * 0.12 = \$0.00000027170304/user$

(206ms @ 2160p): $0.000009446192 * 0.12 = \$0.00000113354304/user$

Assume we have a big user base like Netflix (x40 million or so):

(78ms @ 1080p): $3.1868928

(78ms @ 2160p): $13.2956928

(206ms @ 1080p): $10.8681216

(206ms @ 2160p): $45.3417216

This solution is assuming all 40 million users had cut from the server from the same latency, where the cost is purely the lag time before we stop sending packets. Quite a drastic price increase!

8.

```
// COIS-4310H Networking
// Assignment #1
// Simon Willshire (0491272)
// 16bit (C-Short) Hamming Code
// 08-02-2015

#define DEBUG
#define MSG_LENGTH      16              // # bits in the message
#define ENC_LENGTH      21              // # bits in encoded message (+= 2^n increments in MSG_LENGTH)

#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <time.h>

void printEncoded(bool* encoded)
{
    for(int i = 0; i < ENC_LENGTH; i++)
        printf("%c", encoded[i] ? '1' : '0');
    printf("\n");
}

// Check starting at pos value for count pos value, skip by count pos value, repeat
bool getParityBit(int pos, bool* encoded)
{
    int count = 0, skipcount = 0;
    bool skip = false;
#ifdef DEBUG
    printf("Calculating parity bit at pos = %d\n", (pos+1));
#endif
    for(int i = pos; i < ENC_LENGTH; i++){
        if(!skip){
            if(encoded[i]){ count++; }
#ifdef DEBUG
            printf("Checked at %d = %c, count = %d\n", (i+1), encoded[i] ? '1' : '0', count);
#endif
        }
        if(skipcount == pos){
            skip = !skip;
            skipcount = 0;
#ifdef DEBUG
            printf("Skip counter reset at %d\n", (i+1));
#endif
        } else
            skipcount++;
    }
#ifdef DEBUG
    printf("Parity bit calculated with count = %d, hence it is %s\n", count, (count % 2) ? "odd" : "even");
    encoded[pos] = (count % 2);
    printEncoded(encoded);
#endif
    return (count % 2);         // Return True (1) for ODD parity! (# modulus 2 == 0 will return even)
}

int main()
{
    int c = 0;
    short message;
    bool encoded[ENC_LENGTH];

    srand(time(0));
    message = rand() % SHRT_MAX;

    // Print out short's binary to console as INPUT to Hamming Code
    // Example process in comments:
    //  1100011010010110
    for(int i = 0; i < MSG_LENGTH; i++)
        printf("%c", (message & (1 << i)) ? '1' : '0');

    // Dump in our message formatted by parity bit spacing
    //  When index is not of power 2, otherwise insert 0 to not be counted.
    //  __1_100_0110100_10110
    for(int i = 1; i < ENC_LENGTH+1; i++)
        encoded[i-1] = (i & (i - 1)) ? ((message & (1 << c++)) ? 1 : 0) : 0;

#ifdef DEBUG
    printf("\nBit Stuffing Commence!\n");
    printEncoded(encoded);
    printf("\n");
#endif
    // Calculate Parity Bits
    //  001110010110100010110
    for(int i=1; i < ENC_LENGTH+1; i++)
        if(!(i & (i - 1)))          // Index must be power of 2 to fill parity bit locations
            encoded[i-1] = getParityBit((i-1), encoded);

    printf(" = ");
    printEncoded(encoded);
    return 0;
}
```

Hamming Code Testing

Redid message by hand, compared by debug printlines in output. Here is sample output of the program with the debug preprocessor defined to walk through the parity bit logic:

```
root@alpha:~/Documents/HammingCode# g++ Ham.c
root@alpha:~/Documents/HammingCode# ./a.out
0100111111000110
Bit Stuffing Commence!
00001000111111000000110

Calculating parity bit at pos = 1
Checked at 1 = 0, count = 0
Skip counter reset at 1
Skip counter reset at 2
Checked at 3 = 0, count = 0
Skip counter reset at 3
Skip counter reset at 4
Checked at 5 = 1, count = 1
Skip counter reset at 5
Skip counter reset at 6
Checked at 7 = 0, count = 1
Skip counter reset at 7
Skip counter reset at 8
Checked at 9 = 1, count = 2
Skip counter reset at 9
Skip counter reset at 10
Checked at 11 = 1, count = 3
Skip counter reset at 11
Skip counter reset at 12
Checked at 13 = 1, count = 4
Skip counter reset at 13
Skip counter reset at 14
Checked at 15 = 0, count = 4
Skip counter reset at 15
Skip counter reset at 16
Checked at 17 = 0, count = 4
Skip counter reset at 17
Skip counter reset at 18
Checked at 19 = 1, count = 5
Skip counter reset at 19
Skip counter reset at 20
Checked at 21 = 0, count = 5
Skip counter reset at 21
Parity bit calculated with count = 5, hence it is odd
10001000111111000000110
Calculating parity bit at pos = 2
Checked at 2 = 0, count = 0
Checked at 3 = 0, count = 0
Skip counter reset at 3
Skip counter reset at 5
Checked at 6 = 0, count = 0
Checked at 7 = 0, count = 0
Skip counter reset at 7
Skip counter reset at 9
```

```
Checked at 10 = 1, count = 1
Checked at 11 = 1, count = 2
Skip counter reset at 11
Skip counter reset at 13
Checked at 14 = 1, count = 3
Checked at 15 = 0, count = 3
Skip counter reset at 15
Skip counter reset at 17
Checked at 18 = 0, count = 3
Checked at 19 = 1, count = 4
Skip counter reset at 19
Skip counter reset at 21
Parity bit calculated with count = 4, hence it is even
10001000111111000110
Calculating parity bit at pos = 4
Checked at 4 = 0, count = 0
Checked at 5 = 1, count = 1
Checked at 6 = 0, count = 1
Checked at 7 = 0, count = 1
Skip counter reset at 7
Skip counter reset at 11
Checked at 12 = 1, count = 2
Checked at 13 = 1, count = 3
Checked at 14 = 1, count = 4
Checked at 15 = 0, count = 4
Skip counter reset at 15
Skip counter reset at 19
Checked at 20 = 1, count = 5
Checked at 21 = 0, count = 5
Parity bit calculated with count = 5, hence it is odd
10011000111111000110
Calculating parity bit at pos = 8
Checked at 8 = 0, count = 0
Checked at 9 = 1, count = 1
Checked at 10 = 1, count = 2
Checked at 11 = 1, count = 3
Checked at 12 = 1, count = 4
Checked at 13 = 1, count = 5
Checked at 14 = 1, count = 6
Checked at 15 = 0, count = 6
Skip counter reset at 15
Parity bit calculated with count = 6, hence it is even
10011000111111000110
Calculating parity bit at pos = 16
Checked at 16 = 0, count = 0
Checked at 17 = 0, count = 0
Checked at 18 = 0, count = 0
Checked at 19 = 1, count = 1
Checked at 20 = 1, count = 2
Checked at 21 = 0, count = 2
Parity bit calculated with count = 2, hence it is even
10011000111111000110
 = 10011000111111000110
```

This is one example which can be tested by referencing above debug comments with paper solution.