L-Bit to M-Bit Code Mapping

To Avoid Long Consecutive Zeros in NRZ with Synchronization

Ruixing Li
Department of Electrical Engineering,
University of Nevada, Las Vegas,
Las Vegas, U.S.

Yun Lun
Department of Electrical Engineering,
University of Nevada, Las Vegas,
Las Vegas, U.S.

Shahram Latifi
Department of Electrical Engineering,
University of Nevada, Las Vegas,
Las Vegas, U.S.

Ming Lun
Department of Electrical Engineering,
University of Nevada, Las Vegas,
Las Vegas, U.S.

***Abstract*—we investigate codes that map L bits to m bits to achieve a set of codewords which contain no consecutive n "0"s. Such codes are desirable in the design of line codes which, in the absence of clock information in data, provide reasonable clock recovery due to sufficient state changes. Two problems are tackled- (i) we derive n_{min} for a fixed L and m and (ii) determine m_{min} for a fixed L and n. Results benefit telecommunication applications where clock synchronization of received data needs to be done with minimum overhead.**

Keywords—overhead; mapping; synchronization; consecutive "0"

I. Introduction and Background

In serial communications, data is transferred on a medium that carries a signal varying with time. For digital signal, each bit is represented as a high or low voltage for a fixed amount of time. We call this time period a clock cycle. The clock of the communication line is very important as it tells the transmitter when to transmit a new bit, and it tells the receiver when to read. For short distance transfer, such as communication within a digital system, we can have a clock signal between the transmitter and receiver to synchronize the clock; for example, the Serial Peripheral Interface (SPI) uses a clock signal for clock synchronization. However when it comes to long distance communication, adding signaling only for clock synchronization consumes part of the bandwidth. It is impossible to exactly match the clock speed for the transmitter and receiver. On the other hand, employing codes that contain explicit clock information (ex. Manchester coding) will waste half of the available bandwidth [1]. In practice, the clock information is embedded within the data so that, at the receiver end, the clock can be extracted and used to clock in the received data (using devices such as Phased Locked Loop or PLL). Nonetheless, having a long period of flat signal (which may correspond to consecutive "0"s) may cause the synchronization to be lost. For that purpose, the signal that carries the data must also have sufficient transitions or state changes to allow a PLL to lock onto the incoming data. In the event that a long sequence of "0"s is encountered, there will be a risk of losing synchronization.

As one of the scrambling techniques for data encoding, transmitter should provide sufficient amount of signal transitions for the receiver to maintain clock synchronization [2]. Line coding is applied on data before transmission especially in High Speed Serial Links to ensure a maximum Run Length (RL) to guarantee frequent transitions for Clock and Data Recovery (CDR) in asynchronous links [3], for example, B8ZS and HDB3, which substitute a long sequence with a code violation of the encoding rule. These types of techniques either require increase of signal rate for the same data rate, or require more than 2 signal levels to represent binary data. For Manchester and Differential Manchester, the signal rate is twice the data rate (50% overhead). For B8ZS and HDB3, having 3 signal levels to represent a single binary bit creates a 33% overhead. Though line codes can generate adequate timing information for clock recovery and error detection [5] [6], it usually comes at the cost of additional bits. In this paper, we will discuss how to minimize the overhead with the same clock recovery performance.

Another technique is to eliminate long sequence of zeros by encoding the data so that the transmitted data does not contain long sequences of *"0"*s. The 8b/10b encoding [4] which is widely used, adds 2 bits for every 8 bits resulting in 2/8 = 25% overhead while ensuring a maximum RL of 5. One other example would be mapping 4-bit data to 5-bit codes such that a sequence of *3 "0"*s is avoided ($L = 4, m = 5, n = 3$). There are total $2^4 = 16$ possible codes in 4-bit data.

In 5-bit code space, we have *24 (32 – 8)* codes without *"000"* sequence available. So mapping 4-bit all data with 5-bit codes is possible. Table I shows the 5-bit codes with the absent *"000"* patterns.

TABLE I. 5-Bit Codes with the Absent "000" Patterns

00100	00101	00110	00111	01001	01010
01011	01100	01101	01110	01111	10010
10011	10100	10101	10110	10111	11001
11010	11011	11100	11101	11110	11111

The overhead of this technique is *1/5 = 20%*[1], which is

[1] Calculated by (Total transmitted data size – Actual represented data size)/(Total transmitted data size)

lower than the *50%* of forced transition techniques and the *33%* of substitution techniques. We can try to reduce the overhead of code mapping techniques by mapping larger size data. Questions of interest might be - Can we do 9-bit to 10-bit mapping? If not, how about 9-bit to 11-bit, 61-bit to 64-bit, etc.

This paper proposes an empirical method of calculating the minimum overhead to avoid a given number of consecutive "0"s. The rest of this paper is organized as follows: Section 2 discusses the basic theory of avoid long sequence of "0"s. Section 3 introduces the methodology to achieve two empirical formulas for our concerns. The results and conclusion are given in Section 4 and 5 separately.

II. Theory

The research question here is- Given a specific size of a code, what is the smallest overhead to avoid a given number of consecutive *"0"*s.?

Example 1. We are given a 9-bit code and we want to avoid the sequence *"000".* First we will check if the 10-bit code has enough space to hold the 9-bit code and also avoid the sequence *"000".* If 10-bit is not possible, we will consider the 11-bit code and continue checking until we find the smallest size of code that can hold the 9-bit code and avoid the sequence *"000".*

To check if 10-bit code is enough, first we will enumerate the codes in the 10-bit code that has *"000"* sequence in. The calculations are depicted in Table II. Note that in the patterns given in the table, X can be 0 or 1 and each line must exclude the cases that had been counted in the previous lines.

TABLE II. Enumeration of 10-Bit Codes that Contain "000"

Pattern	The number of occurrences
XXXXX XX000	2^7=128
XXXXX X0001	2^6=64
XXXXX 0001X	$2^5 \times 2^1$=64
XXXX0 001XX	$2^4 \times 2^2$=64
XXX00 01XXX	$2^3 \times (2^3 - 1)$=56
XX000 1XXXX	$2^2 \times (2^4 - 3)$=52
X0001 XXXXX	$2^1 \times (2^5 - 8)$=48
0001X XXXXX	$2^0 \times (2^6 - 20) = 44$
Total	520

Note that for X's of length 3 or more on the right side, we have to exclude any codes that have the "000" sequence, because they were already covered in previous lines.

Subtracting the total number of *"000"* patterns- 520 from the total code space $2^{10} = 1024$, we get only 504 codes which is not enough for mapping all 9-bit codes to 10-bit codes.

Generalization

To answer the general question of if it is possible to map all *L*-bit codes to *m*-bit codes that avoid sequence of n consecutive zeros, we first have to find the number of codes without *n*-zero sequence by subtracting the number of codes with *n*-zero sequence from the total *m*-bit code space 2^m.

To answer another question of finding the smallest number of consecutive *"0"*s, we can start with 2 zeros and work upward. Say if we cannot avoid 2 zeros, test if we can avoid 3, 4, 5, etc. Repeat until we can find that smallest number of consecutives *"0"*s we can avoid through L-bit to m-bit mapping.

To find the number of codes with sequence(s) of n *"0"*s in m-bit space, use a similar step from the 9B-10B mapping example to obtain the solution (Table III).

TABLE III. Enumeration of M-Bit Codes that Contain N Consecutive "0"s

Pattern	The number of occurrences	Distribute the multiplication	k
XXXXXXXXX…X00 …0	2^{m-n}	2^{m-n}	k=0
XXXXXXXX…X00 …01	$2^{m-n-1} \times [2^0 - f(0,n)]$	$2^{m-n-1} - 2^{m-n-1} \times f(0,n)$	k = 1
XXXXXXX…X00… 01X	$2^{m-n-2} \times [2^1 - f(1,n)]$	$2^{m-n-1} - 2^{m-n-2} \times f(1,n)$	k = 2
XXXXXX…X00…0 1XX	$2^{m-n-3} \times [2^2 - f(2,n)]$	$2^{m-n-1} - 2^{m-n-3} \times f(2,n)$	k = 3
XXXXX…X00…01 XXX	$2^{m-n-4} \times [2^3 - f(3,n)]$	$2^{m-n-1} - 2^{m-n-4} \times f(3,n)$	k = 4
⋮			
X00…01XX…XXX XXX	$2^1 \times [2^{m-n-2} - f(m-n-2,n)]$	$2^{m-n-1} - 2^1 \times f(m-n-2,n)$	k= m-n-1
00…01XX…XXXX XXX	$2^0 \times [2^{m-n-1} - f(m-n-1,n)]$	$2^{m-n-1} - 2^0 \times f(m-n-1,n)$	k = m – n

We can see that calculating $f(m,n)$ requires recursive calculation of the number of codes with n-zero sequence in the code lengths less than m. We must define the basis for the recursive function, otherwise we will go to endless loop of calculations. We know that there cannot exist n-zero sequence in the code if the code length m is shorter than n bits. So $f(m,n) = 0$, for $m < n$.

For $m \geq n$, continue our generalization. Adding the terms and simplifying gives

$$2^{m-n} + (m-n) \times 2^{m-n-1} - \sum_{k=1}^{m-n} 2^{m-n-k} \times f(k-1,n)$$

Now that we obtain the piecewise recursive function $f(m,n)$:

$$f(m,n) = \begin{cases} 0, & m < n \\ 2^{m-n} + (m-n) \times 2^{m-n-1} \\ \quad - \sum_{k=1}^{m-n} 2^{m-n-k} \times f(k-1,n), & m \geq n \end{cases}$$

The calculation of the function $f(m,n)$ seems very complicated with the summation. We can make the calculation easier and more efficient by observing the following 2 special cases of m and n.

Case (i) if $m = n$, we know that there is only one code that has the n-zero sequence; that is the n-zero sequence itself.

Case (ii) if $n < m \leq 2n$, the last addition to the sum is $2^0 \times f(m-n-1,n)$. Let's look at the last largest $m = 2n$, we substitute m with 2n and obtain $2^0 \times f(n-1,n)$, the recursive call to the function returns *0* because $n-1<n$. Similarly, all m value between n and *2n* make recursive call to function f with first argument less than second argument n, which will ultimately give *0* as the result. So for $n<m\leq 2n$, the summation evaluates to *0.*

By separating the domain of the function, we produce a new formula with 4 pieces but is easier to calculate or more efficient to compute digitally.

$$f(m,n)=\begin{cases}0, & m<n\\ 1, & m=n\\ 2^{m-n}+(m-n)\times 2^{m-n-1}, & n<m\leq 2n\\ 2^{m-n}+(m-n)\times 2^{m-n-1} - \sum_{k=1}^{m-n} 2^{m-n-k}\times f(k-1,n), & m>2n\end{cases}$$

We can use the code in Appendix to calculate the function *f.* Let $m=10, n=3$, the function returns *520* which match our previous calculation for 9-bit to 10-bit mapping example. We can also check our result by counting the number of codes with “000” pattern by using a brute force checking program, created by Edgar Solorio (See Appendix) Our result matches the number counted by this checking program ($alpha=520$). Similarly when $m=11, n=3$, the function returns *1121,* that means there are *927* codes available for mapping. While this is not enough to map 10-bit, it is sufficient for 9-bit codes, giving *18.2%* overhead.

Generally speaking, for given m and n,

$$L_{max}=floor\{log_2[2^m-f(m,n)]\} \tag{1}$$

Floor is the greatest integer function, mapping a real number to the largest previous integer.

We define the minimum overhead bits

$$h=m-L_{max} \tag{2}$$

If we try *19*-bit to *22*-bit mapping, which is possible, there is only *13.6%* (*3* bits) overhead. Similarly *64*-bit codes has about $2^{56.4}$ codes without *“000”*, while we cannot map *61*-bit codes to *64*-bit codes, there is enough to map 56-bit codes. Also 8-bit to 9-bit mapping is possible, 9-bit space has *238* codes with *“000”,* leaving *274* codes available to map *8*-bit (*256* possible) codes. The overhead of for 8B9B is *11.1%.* Since 9B-10B is impossible, code mapping with *1*-bit overhead stops at with *8*-bit to *9*-bit mapping.

III. Methodology

The result from the 9B-10B example in the Work section shows that mapping from *9*-bit codes to *10*-bit codes cannot avoid all codes with 3 consecutive zeros. If we want to avoid *3* consecutive zeros, the minimum overhead to map *9*-bit code is *2* bits. The following 2 questions are our main concerns about avoiding consecutive zero level signal transmitted in regard to maintain synchronization.

A. *For a given pattern length L and mapping from L bits to $m=L+h$ bits, what is the minimum number of consecutive “0”s we can avoid? (Fixed L and m, find n_{min})*

The code in Appendix shows how to solve this question. Table IV shows the minimum avoidable zeros with *1* to *9* bits overhead for mapping data of lengths from $L=2$ to *24* bits. The jumps in n_{min} are highlighted and bold faced.

TABLE IV. Minimum Avoidable Zeros with 1 to 9 Bits Overhead for Mapping Data of Lengths from $L=2$ To **24** Bits

L	h=1	h=2	h=3	h=4	h=5	h=6	h=7	h=8	h=9
2	2								
3	2								
4	**3**	2							
5	3	2							
6	3	**3**	2						
7	3	3	2						
8	3	3	3	2					
9	**4**	3	3	2					
10	4	3	3	**3**	2				
11	4	3	3	3	2				
12	4	3	3	3	2				
13	4	3	3	3	**3**	2			
14	4	3	3	3	3	2	2		
15	4	3	3	3	3	**3**	2		
16	4	3	3	3	3	3	2		
17	4	**4**	3	3	3	3	**3**	2	
18	4	4	3	3	3	3	3	2	
19	4	4	3	3	3	3	3	**3**	2
20	4	4	3	3	3	3	3	3	2
21	**5**	4	3	3	3	3	3	3	2
22	5	4	3	3	3	3	3	3	**3**
23	5	4	3	3	3	3	3	3	3
24	5	4	**4**	3	3	3	3	3	3

If we rearrange and extend the data, we can get the following table:

TABLE V. Minimum L for Specific N_{MIN} and H

h	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9
1	2	4	9	21	43	88	177	355
2	4	6	17	38	82	171	348	702
3	6	8	24	56	122	254	519	
4	8	10	31	74	161	337	690	
5	10	13	38	92	201	420	861	
6	13	15	46	110	240	503		
7	15	17	53	127	279	586		
8	17	19	60	145	319	669		
9	19	22	68	163	358	753		
10	22	24	75	181	397	836		
11	24	26	82	199	437	919		
12	26	28	89	216				
13	28	31	97	234				
14	31	33	104	252				
15	33	35	111	270				
16	35	38	118	288				
17	38	40	126	305				
18	40	42	133	323				
19	42	44	140					
20	44	47	148					

We plot the data both in horizontal (Fig. 1) and vertical direction (Fig. 2),

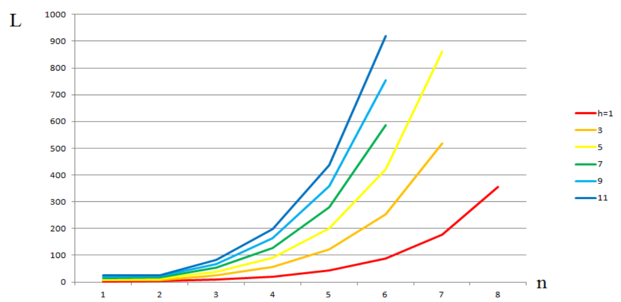


Fig. 1. Plotting using data from Table V (horizontal)

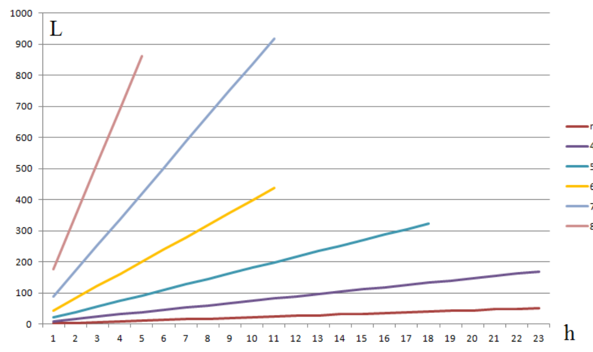


Fig. 2. Plotting using data from Table V (vertical)

Obviously, we can observe that L is approximating linear with h and is relatively exponential with n. This means for a fixed n, the overhead bits h should be proportional with L. We can assume that

$$L = \left(e^{(n-c)/a} - b\right) * h \qquad (3)$$

a, b, c in this formula is the coefficient to be determined.

We can choose any three points in Table V to determine the coefficient. For example, we substitute *(3, 20, 47)*, *(4, 20, 148)* and *(5, 11, 199)* [in *(n, h, L)* order] in (3), we can get $a = 1.333, b = 2.171, c = 0.988$. Therefore,

$$\left(\mathrm{e}^{\frac{\mathrm{n}-0.988}{1.333}} - 2.171\right) * \mathrm{h}$$

Thus,

$$n = 1.333 * \ln\left(\frac{L}{h} + 2.171\right) + 0.988 \qquad (4)$$

When L, m is given, we can determine n_{min} via (4) and $n_{min} = floor(n)$.

B. *For a given number of consecutive "0"s to avoid, how to minimize the overhead? Solve the problem for a specific case of avoiding two "0"s first. Under what conditions can we map "L" to "L + 1"? If not, what about "L" to "L + 2" or "L" to "L + 3"? (Fixed L, n, find h_{min})*

The code in Appendix shows how to solve question 2. Table VI shows the minimum overheads to avoid n zeros with $n = 2$ to 10 for mapping data of lengths from $L = 4$ to 24 bits. The jumps in overheads are highlighted and bold faced.

TABLE VI. MINIMUM OVERHEADS TO AVOID N CONSECUTIVE ZEROS

L	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9	n=10
4	2	1	1	0	0	0	0	0	0
5	2	1	1	**1**	0	0	0	0	0
6	**3**	1	1	1	**1**	0	0	0	0
7	3	1	1	1	1	**1**	0	0	0
8	**4**	1	1	1	1	1	**1**	0	0
9	4	**2**	1	1	1	1	1	**1**	0
10	**5**	2	1	1	1	1	1	1	**1**
11	5	2	1	1	1	1	1	1	1
12	5	2	1	1	1	1	1	1	1
13	**6**	2	1	1	1	1	1	1	1
14	6	2	1	1	1	1	1	1	1
15	**7**	2	1	1	1	1	1	1	1
16	7	2	1	1	1	1	1	1	1
17	**8**	**3**	1	1	1	1	1	1	1
18	8	3	1	1	1	1	1	1	1
19	**9**	3	1	1	1	1	1	1	1
20	9	3	1	1	1	1	1	1	1
21	9	3	**2**	1	1	1	1	1	1
22	**10**	3	2	1	1	1	1	1	1
23	10	3	2	1	1	1	1	1	1
24	**11**	**4**	2	1	1	1	1	1	1

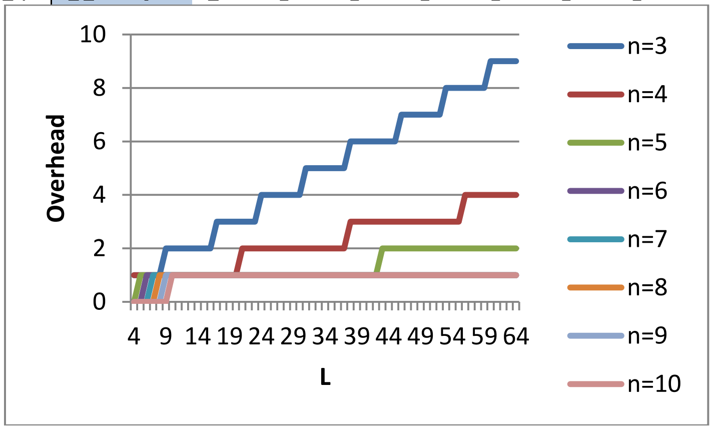


Fig. 3. Plotting using data from Table VI

If we rearrange and extend the data of TABLE VI, we can get the following table:

TABLE VII. MINIMUM L FOR SPECIFIC N AND H

h	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9
1	2	3	4	5	6	7	8	9
2	4	9	21	43	88	177	355	710
3	6	17	38	82	171	348	702	
4	8	24	56	122	254	519		
5	10	31	74	161	337	690		
6	13	38	92	201	420	861		
7	15	46	110	240	503			
8	17	53	127	279	586			
9	19	60	145	319	669			
10	22	68	163	358	753			
11	24	75	181	397	836			
12	26	82	199	437	919			
13	28	89	216					
14	31	97	234					
15	33	104	252					
16	35	111	270					
17	38	118	288					
18	40	126	305					
19	42	133	323					

This table is slightly different from Table V. We can use similar procedure and get

$$h = \frac{L}{1.009e^{0.75n} - 2.171} + 1 \qquad (5)$$

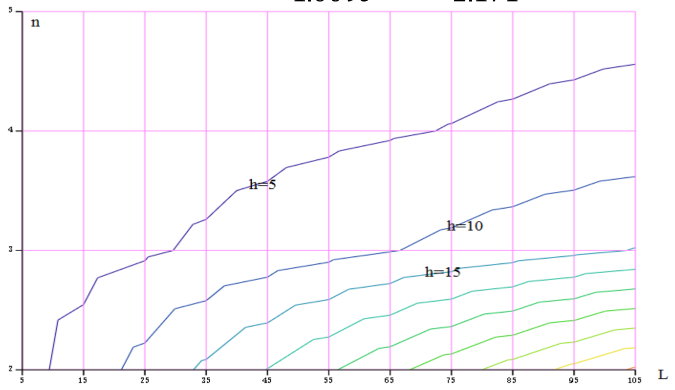


Fig. 4. Minimum overheads versus message length L and number of consecutive “0”s to be avoided

IV. Resulds

We can check the accuracy of (4) by different check point:

TABLE VIII. Comparison of Theoretical and Calculation N by Fixed L and H

Point(L, h)	(44, 19)	(60, 8)	(145, 8)	(437, 11)	(918, 11)	(861, 5)	(702, 2)
Actual n	3	4	5	6	7	8	9
Approximate According (4)	2.99	4.01	5.00	5.97	6.92	7.87	8.81
deviation	0.33 %	0.33 %	0.04 %	0.55 %	1.14 %	1.65 %	2.12 %

From this table, we can see, when n is between 3 and 9, (4) is accurate enough to determine the minimum consecutive *“0”s* can be avoided for fixed L and m.

We can also check the availability of (5) by different check points in TABLE IX:

From this table, we can see, when n is between 3 and 6, this formula is accurate enough to determine the minimum required overhead bits h to avoid n consecutive “0”s for fixed message length L.

TABLE IX. Comparison of Theoretical and Calculation N by Fixed L and H

L	88	82	56	74	92	46
n	6	5	4	4	4	3
Actual h	2	3	4	5	6	7
h from (5)	1.992597	3.013121	4.094725	5.089458	6.084191	7.214431
deviation	-0.37%	0.44%	2.37%	1.79%	1.40%	3.06%

V. Conclusion

We have considered the problem of L to m mapping to avoid a set of n consecutive “0”s. We derived two formulas to calculate (i) the minimum number of consecutive “0”s that can be avoided for fixed L and m (4) and (ii) the minimum overhead required to avoid a given number of consecutive “0”s with fixed L (5). We found the exact values for small values of L, m and n (Table IV and Table VII). For very long messages, we used the empirical results and combination of several tables to arrive at a formula that will give the desired answer with close approximation.

One may think of splitting a long code into smaller codes and using the results for small values to obtain the parameters for the long code. For example, the splitting of 56-bit code into 8*7-bit codes can simplify the calculation but will not work since a potential problem can occur: Even if all 8 7-bit codes have no “000”, when the frame size is more than 7 bits (e.g. 64 bits), there can exist consecutive “000” in the end of a 7-bit code and the start of another consecutive 7-bit code.

The results obtained can find applications in coding and communication where the synchronization of the transmitter and receiver is of primary concern.

Acknowledgment

This work was supported in part by the NSF EPSCoR grant under Award # EPS-IIA-1301726.

References

[1] V.Sneha Latha et al. “Performance Evaluation of Different Line Codes”.Indian Journal of Computer Science and Engineering (IJCSE). Vol. 2 No. 4, 2011

[2] Stallings, W. “Data and Computer Communications”. 10th Ed. 2013.

[3] J. Saadé et al. “Low Overhead, DC-Balanced and Run Length Limited Line Coding” IEEE 19th Workshop on Signal and Power Integrity (SPI), 2015.

[4] P.A. Franszek and A.X. Widmer, “A DC-Balanced, Paritioned-Block, 8B/10B Transmission Code”, IBM Journal of research and development, Volume 27, Number 5, September 1983.

[5] J. G. Proakis, Digital Communication, 4th ed., Mc-Graw Hill, New York, 2001.

[6] K. W. Cattermole, “Principles of digital line coding,” Int. J. Electron., vol. 55, no. 1, pp. 3–33, July 1983.

APPENDIX

```
int numOfOccurrences(int m, int n)
{
    if (m < n)          // m < n
        return 0;
    else if (m == n)    // m = n
        return 1;
    else if (m <= 2*n)  // n < m ≤ 2n
    {
        int result;
        result = twoToPowerOf(m-n) + (m-n) * twoToPowerOf(m-n-1);
        return result;
    }
    else                // m > 2n
    {
        int result;
        int *s = new int[m-n+1];
        s[0] = twoToPowerOf(m-n) + (m - n) * twoToPowerOf(m-n-1);
        result = s[0];
        for (int i = 1; i <= m-n; i++)
        {
            s[i] = numOfOccurrences(i-1, n) * twoToPowerOf(m-n-i);
            result -= s[i];
        }
        return result;
    }
}
```

C++ Code for function f (named numOfOccurrences)

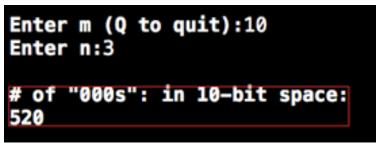
Note: The codes shown in this report emphasize on basic idea to implementation. The actual result from the code may be incorrect due to limitation in range in of int data type. Also, a cached table may be necessary to reduce repeated calculations and to improve performance.

twoToPowerOf is a simple function that returns the power of n without importing the C Math library.

```
int twoToPowerOf(int n)
{
    if (n < 0)
    {
        std::cout << "Cannot calculate negative or fractional powers"
                  << std::endl;
        exit(1);
    }
    else
    {
        int result = 1;
        result = result << n;
        return result;
    }
}
```

Code for function twoToPowerOf(n)

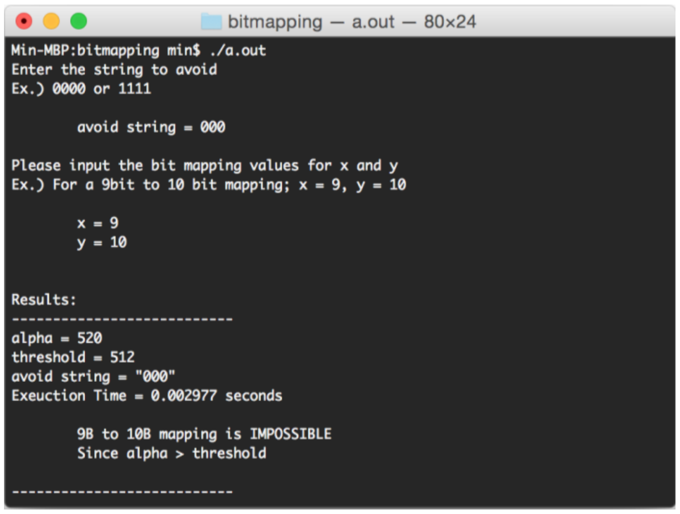
Programs to check number of codeword with n consecutive *"0"s*



```
Enter m (Q to quit):10
Enter n:3

# of "000s": in 10-bit space:
520
```

Program outputs for $m = 10, n = 3$



```
Min-MBP:bitmapping min$ ./a.out
Enter the string to avoid
Ex.) 0000 or 1111

        avoid string = 000

Please input the bit mapping values for x and y
Ex.) For a 9bit to 10 bit mapping; x = 9, y = 10

        x = 9
        y = 10


Results:
--------------------------
alpha = 520
threshold = 512
avoid string = "000"
Exeuction Time = 0.002977 seconds

        9B to 10B mapping is IMPOSSIBLE
        Since alpha > threshold

--------------------------
```

Program outputs using code provided by Edgar Solorio [8]

```
// this function calculates the minimum number of consecutive zeros
we can avoid
// by mapping l-bit codes to m-bit codes.
int minimumAvoidableConsecutiveZeros(int L, int m)
{
    if (L >= m)
    {
        return -1; // you cannot avoid any consecutive zeros
        // moreover, if m < L, you cannot even map from L-bit to m-bit
anyway.
    }
    int minZeros =L;
    // starting from 2 zeros "00"
    for (int i = 2; i < L; i++)
    {
        if (twoToPowerOf(m) - numOfOccurrences(m, i) >
twoToPowerOf(L))
        {
            minZeros = i;
break;
        }
    }
    return minZeros;
}
```

Code to solve question 1

```
#define MAXIMUM_ALLOWED_OVERHEAD 5
#define MAXIMUM_ALLOWED_PERCENT_OVERHEAD 30
// this function calculates the minimum overhead by mapping l-
bit code
int minimumOverheadToAvoid_n_Zeros(int L, int n)
{
    if (n < 2)
        return -1;  // error, n must be at least 2

    int maximum_m = L + MAXIMUM_ALLOWED_OVERHEAD;
    int max_MP = (m *
(100+MAXIMUM_ALLOWED_PERCENT_OVERHEAD) / 100);
    if (max_MP > maximum_m)
        maximum_m  = max_MP;

    ULL numOfLBitCodes = twoToPowerOf(L);

    for (int i = L+1; i < maximum_m; i++)
    {
        if (twoToPowerOf(i) - numOfOccurrences(i, n) >
numOfLBitCodes)
        {
            return (i-L);
        }
    }
    return -1;  // reached maximum allowed overhead
}
```

Code to solve Question 2