IMPLEMENTING COBS ZPE IN C#

KEERAT SINGH

.Net tool Developer, WIPRO Technologies.

Abstract- “Byte stuffing is a process that transforms a sequence of data bytes that may contain 'illegal' or 'reserved' values into a potentially longer sequence that contains no occurrences of those values. The extra length of the transformed sequence is typically referred to as the overhead of the algorithm. Consistent Overhead Byte Stuffing (COBS) is an algorithm for encoding data bytes that result in efficient, reliable, unambiguous packet framing regardless of packet content, thus making it easy for receiving applications to recover from malformed packets. Essentially what it does is change all zero bytes into bytes that indicate the length of the next block till a zero (including the zero itself in the length).” [che99]

This paper researches the COBS algorithm presented by Stuart Cheshire and Mary Baker, Member, IEEE and cites some examples as to show the encoding as well as decoding using COBS Algorithm Zero Pair Elimination method. In the end a programming implementation of the COBS algorithm in C# is given in order to decode a COBS encoded data using the ZPE variant.

Index Terms—*COBS, ZPE, Data Byte, Code Byte*

I. INTRODUCTION

The function of byte stuffing is to transform data packets into a form suitable for transmission over a serial medium like a telephone line. When the packets are to be sent over a serial medium there has to be a way to tell where one packet ends and the next begins, especially after errors, and this is typically done by using a reserved value to indicate packet boundaries called *Frame Delimiters*. *Byte stuffing* is a process that converts a sequence of data bytes that may contain 'reserved' values into a potentially longer sequence that contains no occurrences of those reserved values. The extra length of the transformed sequence is typically referred to as the *overhead*.

Consistent Overhead Byte Stuffing (COBS) algorithm can be used for encoding without being concerned too much of the data since all the packets with a length up to 254 bytes have only 1 byte of overhead and all the packets greater than 254 bytes in length have 1 byte overhead per 254 bytes. Which mathematically amounts to (1/254)*100 = 0.393 %. Rounding it off gives us an approximate figure of 0.4%.

The best thing about COBS algorithm is that is very less resource intensive as it can be implemented in any programming language with just a few lines of code. An example on how to decode a frame in C# is given at the end.

II. VARIOUS CONSISTENT OVERHEAD BYTE STUFFING ALGORITHMS

This section begins by describes the format used by COBS to encode the data and examples to help you understand the process of encoding and decoding better. Firstly it describes the basic COBS algorithm and then it goes on to describe a slightly different version of COBS algorithm called COBS/ZPE (Consistent Overhead Byte Stuffing/ Zero Pair Elimination Method). Then it goes on to show some bar charts comparing the real time experimental results and performance of both the versions of the algorithm.

A. COBS Algorithm

COBS algorithm can do a reversible conversion of data as it eliminates an octet from the data which is of size 1 byte and has a value of 00 most commonly since 0 and 1 are the most commonly used digits in binary transmission; hence the probability of occurrence of 00 is higher than other pairs. Performance of COBS is better when it eliminates more octets to remove it with a single encoded value.

After the elimination the octet can be used as a *frame delimiter* without any problems. COBS first reads the input packet and adds a zero octet called the *Phantom Octet* at the end. It is not compulsory to add this octet in the memory but the program has to consider a zero octet appended at the end.

Here is an example of what the packet looks like after appending the *Phantom Octet.*

Input:

72	00	4D	4C	79	00	00	40	06	4F	37	**00**

Fig.1. COBS encoded packet with a *Phantom Octet*

COBS then searches for all the zero octets in the packet (including the *Phantom Octet*), and separates the packet into various frames with each frame ending with a zero octet (00) and having exactly one zero octet. The frame may be as small as one byte or as big as the packet itself.

An Example shows how a packet is separated into various frames.

Input:

0	7	2	4	7	0	0	4	0	4	**0**
0	2	C	C	9	0	0	0	6	F	**0**

Output:

72	2C	4C	79	00
40	06	4F	00	

Fig.2. COBS encoded packet input with zero bytes and output with separated packets.

COBS encodes each frame using one or more variable length COBS code blocks. Frames which are less than 254 bytes in length are encoded as a single COBS code block and the ones which are longer are encoded using ((n)/254) bytes + 1 COBS code blocks. Refer to the example given below to have a better understanding of encoding of frames larger than 254 bytes.

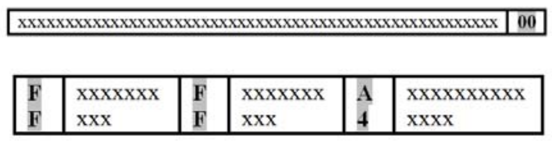

Fig.3. COBS code blocks. Each COBS code block begins with a single code byte (shown shaded), followed by zero or more data bytes.

After all the frames have been encoded with COBS code blocks they can now be joined together into one single packet and the entire packet does not contain any zero octet, hence 0x00 can be placed around each individual frame in a packet as frame delimiters so as to define the frame boundaries. A *COBS code block* consists of a single code byte as defined in Table 1, followed by zero or more data bytes. The number of data bytes is determined by the code byte.

The figure below shows some examples of valid COBS code blocks and the encoded frames with code bytes.

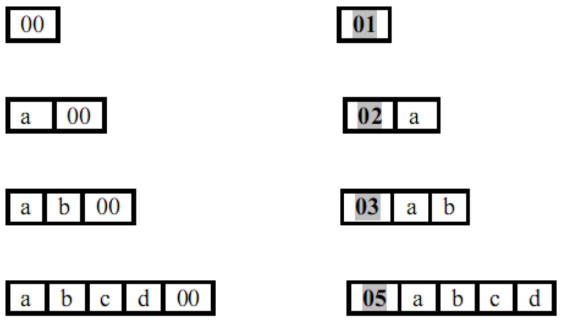

Fig.4. COBS code blocks and the encoded frames with data bytes

Code 0x01 means that there is no data byte after the code byte but just one 00 Zero octet which is actually the *Phantom Octet.*

Codes 0x02 to 0xFE mean that each COBS code block consists of N-1 data bytes followed by a 00 Zero octet which is again the *Phantom Octet* where N is the decimal value of the Hexadecimal Code.

Code 0xFF is used to denote the 254 data bytes not followed by a zero octet.

As it can be seen that these COBS code blocks don't add any additional overhead since the frame which is N+1 data bytes is encoded into a frame of 1+N data bytes which actually means that the code block had N data bytes followed by 1 byte zero octet is encoded into a frame which consists of 1 byte of code byte followed by N bytes of data.

This method is only valid for frames less than 254 bytes long. For frames longer than 254 bytes the code byte 0xFF is used to define a frame with 254 bytes of data and no zero octet in the end but the decoded frame does contain an implicit zero in the end.

B. How it actually works

The job of the COBS encoder is to translate the raw packet data into a series of COBS code blocks. The function of a COBS encoder is to convert the raw packet data into COBS code blocks. First separating the packet into different frames each ending with a trailing zero and then converting into an encoded packet is a good method to make one understand the working of the algorithm. But in actual the encoder directly converts the raw packet into an encoded one without breaking up into zero ending frames.

Code(Hex)	**Code(Decimal)**	**Followed by**	**Meaning**
0x**00**	**00**	(not applicable)	(not allowed)
0x**01**	**01**	no data bytes	A single zero byte
0x**02**-0x**FE** (0x**N**)	**02-254**	(*N*–1) data bytes	The (*N*–1) data bytes followed by a single zero
0x**FF**	**255**	254 data bytes	The 254 data bytes **not** followed by a zero

Table 1. Code values used by COBS encoder.

The encoder traverses through the first 254 bytes looking for the first zero octet. If no such octet is found then a code byte 0xFF is added to the starting of the frame followed by the 254 bytes of data. If a zero octet is found then the numbers of bytes before the octet are counted and the corresponding code byte is added to the starting of the frame. If the zero octet is found after N data bytes then the corresponding code would be N+1 converted into its Hexadecimal value. The code byte is added to the beginning of the frame, the data bytes are added after that and the zero octet is removed. This process takes place until the end of the packet is reached and a logical zero octet is appended to the end of the packet. Fig. 5 shows an example of packet encoding.

Input:

22	72	2C	00	79	4C	BD	37	2D	44	**00**

Output:

04	22	72	2C	**07**	79	4C	BD	37	2D	44

Fig.5. COBS encoding showing input with a phantom zero logically appended and the corresponding zero-free output.

The algorithm can be tweaked to cater to a particular case of a packet with a length of 254 bytes which does not need a zero octet to be added at the tail of the frame.

C. A COBS variant the Zero Pair Elimination Method

In the real world not only zero is common but a pair of adjacent zeros also occurs very frequently in the data transmitted across the internet especially in the headers of small TCP/IP packets. To make full use of this situation a new variant of the COBS algorithm was devised in which the maximum length which could be encoded was reduced and some new code bytes were introduced with new meanings. All the code bytes with their meanings can be seen in Table 2. Mostly the new code bytes are incorporated to represent the adjacent pair of zero octets.
In COBS Zero Pair Elimination method the byte codes from 0x**00** to 0x**CF** have the same meaning as the original COBS. Instead of 253 bytes now only 206 bytes can used to represent a sequence of non-zero data bytes followed by a zero octet. Earlier 0x**FF** was used to encode the maximum length of 255 bytes which is now denoted by 0x**D0** and the maximum length sequence has been reduced to 207 data bytes without the phantom octet. Code bytes 0x**D1** and 0x**D2** are unused and reserved for future use. 0x**D3** to 0x**DF** are used to denote a run of N zero octets. N = Decimal value of the code byte (0x**D3**–0x**DF**) – 208. Code bytes 0x**E0**–0x**FE** are used to denote N data bytes followed by a pair of Zero Octets. N = Decimal value of the code byte (0x**E0**–0x**FE**) – 223.0x**FF** is used to generally denote a Frame error.

Code(Hex)	**Code(Decimal)**	**Followed by**	**Meaning**
0x**00**	**00**	(not applicable)	(not allowed)
0x**01**-0x**CF**	**01-207**	(*n*–1) data bytes	The (*n*–1) data bytes followed by a single zero
0x**D0**	**208**	207 data bytes	The 207 data bytes **not** followed by a zero
0x**D1**	**209**	Unused	(resume preempted packet)
0x**D2**	**210**	Unused	(reserved for future use)
0x**D3**–0x**DF**	**211-223**	Nothing	a run of (n-D0) zeroes
0x**E0**–0x**FE**	**224-254**	n-E0 data bytes	The data bytes, plus two trailing zeroes
0x**FF**	**255**	Unused	(PPP error)

Table 2. Code values used by COBS ZPE encoder.

Input:

22	72	2C	00	00	4C	BD	00	00	44	**00**

Output:

E3	22	72	2C	**E2**	4C	BD	02	44

Fig.5. COBS ZPE encoding showing input with a phantom zero logically appended and the corresponding zero-free output.

III. EXPERIMENTAL RESULTS

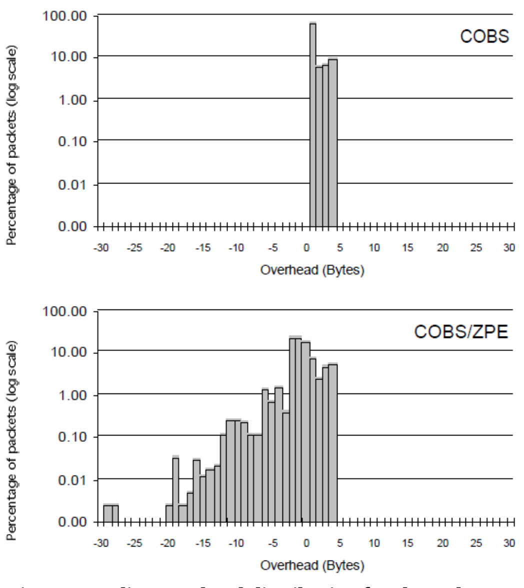

Fig. 7. Encoding overhead distribution for three-day trace.

Histograms showing, for each amount of overhead indicated on the horizontal axis, the percentage of packets that incur that overhead.

This experiment [Che99] is conducted by one of the colleague of the authors of the paper Consistent Overhead Byte Stuffing by Stuart Cheshire and Mary Baker, *Member, IEEE. "This trace was collected over a period of 3 days from home via a portable ISM-band packet radio attached to the computer. The trace contains 36,744 IP packets, totaling 10,060,268 bytes of data (including IP headers and higher layers; not including the link-level header). The MTU of the wireless interface in this case was 1024 bytes, giving worst-case COBS overhead for large packets of five bytes. However, most of the packets captured were not large; 69% of the packets were shorter than 254 bytes and necessarily incurred exactly one byte of overhead when encoded with COBS. Moreover, 41% of the packets were exactly 40 bytes long, which is just the length of a TCP acknowledgement containing no data. Another 10% of the packets were exactly 41 bytes long, which is the length of a TCP packet containing just one data byte. Taking these two numbers together, this means that over half the packets were 40 or 41 bytes long. Only 15% of the packets were maximum-sized 1024-byte packets. The three-day trace was a particularly challenging test case with which to evaluate COBS, because it contains so many small packets."*

IV. CONCLUSIONS

We can successfully conclude from the Fig 7 the COBS Zero Pair Elimination method has a far better compression rate as compared to the original COBS algorithm. Hence I have decided to go ahead and use the COBS/ZPE method in a program that I made which decodes the input string which is in encoded form. The program is written in a console application using C# language. A source code has been included after this section as on how the encoding is done according to my understanding of the COBS ZPE algorithm. As it was discussed earlier that the COBS algorithm is easy to implement which has been shown the core functionality can be achieved in only a few lines of code.

V. ACKNOWLEDGEMENT

I express my sincere thanks to Stuart Cheshire and Mary Baker for writing the wonderful paper about the COBS algorithm and explaining it beautifully.

VI. SOURCE CODE

C# Source Code for the implementation of COBS ZPE algorithm.

```
public static void COBSDecoder(string strInput)
{
String[] strFrameDelimitedString ;

/*Using Frame Delimiters to seperate Frame*/
strFrameDelimitedString = strInput.Split(new
string[1] { "00 00" },
StringSplitOptions.RemoveEmptyEntries);
int i = 0;
foreach (string line in strFrameDelimitedString)
{
        /*Removing the left over Frame Delimiters*/
        strFrameDelimitedString[i] =
        (line.Replace("00", "")).Trim();
        i++;
        }
        for(int j = 0; j <
strFrameDelimitedString.Length; j++)
        {
        OctetsDecoder(strFrameDelimitedString[j]);
        }
}

public static void OctetsDecoder(String
strFrameDelimitedString)
{
      List<String> lstFrameDelimitedString = new
      List<string>();
      /*Finding the position to insert 00 Octet or 00 00
      Pair Octet*/
      int i = 0;
      byte bytEncodedOctet;
      lstFrameDelimitedString=
      strFrameDelimitedString.Split(new string[1]{"
      "},
      StringSplitOptions.RemoveEmptyEntries).ToList
      <string>();
try
{
for (i=0; i<lstFrameDelimitedString.Count; i = i +
bytEncodedOctet)
{
    /*Conversion of HexaDecimal to Byte value*/
    bytEncodedOctet =
    byte.Parse(lstFrameDelimitedString[i],
    System.Globalization.NumberStyles.HexNumber)
    ;

    /*Searching for 01-CF (01-207) Octet and adding
    00 Octet*/
    if
    (Enumerable.Range(1,207).Contains((int)bytEnco
    dedOctet))
    {
      if(bytEncodedOctet.Equals((byte)lstFrameDeli
     mitedString.Count))
      {
      lstFrameDelimitedString.RemoveAt(i);
      }
      else
      {
```

```
  lstFrameDelimitedString.Insert(i +
 bytEncodedOctet, "00");
  lstFrameDelimitedString.RemoveAt(i);
  }
}

/*Searching for E0-FE(224-254) Octet and adding
00 00 Octet Pair */
else if
(Enumerable.Range(224,254).Contains((int)bytEn
codedOctet))
{
  lstFrameDelimitedString.Insert(i +
  bytEncodedOctet - 223, "00 00");
  lstFrameDelimitedString.RemoveAt(i);
  i -= 223;
}

/*Searching for D3-DF(211-223) Octet and
adding n 00 Octet pairs */
else if
(Enumerable.Range(211,223).Contains((int)bytEn
codedOctet))
{
  for (int j = 0; j < bytEncodedOctet - 208; j++)
  {
  lstFrameDelimitedString.Insert(i + 1, "00");
  }
  lstFrameDelimitedString.RemoveAt(i);
  i -= 208;
}

/*Searching for 'FF' frame*/
else if (bytEncodedOctet.Equals((byte)255))
{
  lstFrameDelimitedString.Clear();
  lstFrameDelimitedString.Add("Frame Error");

}
StringBuilder strOutput = new StringBuilder();
  /*Removing last 00 Octet also known as Phantom
  Octet*/
  if
  (lstFrameDelimitedString[lstFrameDelimitedStrin
  g.Count - 1].Contains("00"))
  {
    if
    (lstFrameDelimitedString[lstFrameDelimitedStr
    ing.Count - 1] == "00 00")
   {
   lstFrameDelimitedString[lstFrameDelimitedStrin
   g.Count - 1] = "00";
   }
   else
   {
   lstFrameDelimitedString.RemoveAt(lstFrameDel
   imitedString.Count - 1);
   }
   }
   foreach (string strOctet in
   lstFrameDelimitedString)
   {
   //Appending each octet with a space " "
   Console.Write(strOctet + " ");
   }
   Console.WriteLine();
 }
  catch
  {
  Console.WriteLine("Invalid Frame");
  }
}
```

x

[1] [Che99] Stuart Cheshire and Mary Baker, “Consistent Overhead Byte Stuffing”, *IEEE/ACM transactions on networking, vol.7, no. 2, April 1999.*

[2] [Che98] Stuart Cheshire and Mary Baker, “PPP Consistent Overhead Byte Stuffing (COBS)”, draft-ietf-pppext-cobs-00.txt November 1997

◈◈◈