RADIX 95 :
Binary to Text Data Conversion for Packet Radio

James G. Jones, WD5IVD
University of North Texas
Department of Computer Science

Gerald A. Knezek, KB5EWV
University of North Texas
Department of Computer Education and Cognitive Systems
Denton, Texas 76203

Abstract

Binary files can prove to be difficult to transfer over the current amateur packet radio network. Radix 95 provides a way to convert data such as compiled programs or graphic images to printable ASCII characters and allow their transfer in Converse Mode. Radix 95 (base 95) is a simple variable length encoding scheme which offers greater efficiency than is available with conventional fixed-length encoding procedures.

Introduction

Transfer of data across the amateur packet radio network usually takes the form of point-to-point **computer** connections or store-and-forward BBS's. In the case of point-to-point data transfer, the TNC (Terminal Node Controller) can be configured in ways to allow the transfer of 8-bit data (converse mode [8BITCONV ON, AWLEN 8, XFLOW OFF] or simply use transparent mode). However, the transfer of 8-bit data through the current store-and-forward BBS (Bulletin Board Sysetm) network is unreliable due to the use of certain 8-bit characters for control. In this case, a message that **assumes** the eighth bit is available for data can cause the transfer to fail.

With the use of available compression programs, 8-bit to 7-bit conversion techniques, and file splitting, it should be possible to transfer 8-bit data across the amateur packet radio network with minimal negative impact upon total network operations.

This paper will focus on Radix 95, a base 95 8-bit to 7-bit file conversion method which offers greater efficiency than is available with conventional fixed-length encoding procedures, and its possible use for sending 8-bit data across the network.

8-bit to 7-bit Conversion

Three common steps are involved in converting 8-bit to 7-bit data and transferring it from one point to another :

1. Translate a sequence of bits into printable ASCII code.
2. Transmit the data
3. Convert the ASCII code back to original form.

The problems facing the transmission of 8-bit data are :

1. How to transmit streams of eight or more binary digits through a network that might have problems handling the data.

2. How to pass certain 7-bit sequences through without having the sequence interpreted as a control character. These characters include flow control (DC1, DC3), padding (NULL, DEL), transfer of control (ESC), or any of the other non-printable 7-bit characters. Any sequence of data bits which could represent a control character should not be sent unless some special provision is made to mask it as a printable ASCII character. [Stone 1984; Brown 1984; Da Cruz 1984-1]

3. How to avoid significant amounts of transmission overhead.

Most data conversion methods achieve 1 and 2, but introduce significant amounts of transmission overhead. Radix 95 produces less overhead than most conversion methods now commonly used.

Overhead

Overhead is the measure of how many extra bits must be utilized to convey meaningful information. Encoding overhead is defined as :

$$O_e = \frac{b_e - b_s}{b_s}$$

O_e = encoding overhead.
b_s = number of bits of meaningful data in the source representation.
b_e = number of bits occupied by the meaningful data after encoding.

Current Conversion Methods

Hex Encoding

Hex encoding views each binary octet as two contiguous 4-bit sequences. Thus each group of four bits is translated into its corresponding hexadecimal character.

The mathematical transformation is from Binary (base 2) to Hexadecimal (base 16). After transmission, the 4-bit sequences are recombined into the original data. Table 1 shows the Hex Encoding scheme.

The encoding overhead for Hex Encoding is 75% [(14-8)/8 = 6/8 bits]. With the addition of the accompanying parity bit, most implementations actually use 16-bits to transmit eight bits of information. The total data encoding plus parity-bit overhead is 100%.

BINHEX for the macintosh is a common program which implements Hex Encoding.

TABLE 1 :
Binary Encoding with Hexadecimal

Bit pattern	Representation
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Character Prefixing

Character Prefixing checks every sequence of eight bits and sends it unaltered if the bit stream corresponds to a printable ASCII character. If the eighth bit is a 1, then a special prefix character (typically &) is sent preceding the character corresponding to the remaining seven bits. If the remaining seven bits represent an ASCII control character,

then the character is transformed to one which is printable (by complimenting the seventh bit) and the transformed character is also preceded by a second special prefix character (typically #). The special prefix characters are themselves prefixed for transmission. Table 2 shows the encoding scheme.

TABLE2 :
Binary Encoding with Character Prefixing

Bit Pattern	Numeric Equivalent	Transformation	Transmission
00000000	0	1000000	# @
00000001	1	1000001	# A
.	.	.	.
.	.	.	.
oooilllll	31	lolilll	# -
00100000	32	0100000	SPACE
00100001	33	0100001	!
00100010	34	0100010	"
00100011	35	1100011	& c
00100100	36	0100100	$
00100101	37	0100101	%
00100110	38	1100110	& f
00100111	39	0100111	'
.	.	.	.
.	.	.	.
01111110	l i 6	1111110	~
01111111	127	0111111	# ?
10000000	128	1000000	& # @
10000001	129	1000001	& # A
.	.	.	.
.	.	.	.
111~1111	255	0111111	& # ?

The overhead for character prefixing is dependent on the type of file being transmitted. Text files are efficient, since few characters need to be prefixed. For a binary file consisting of randomly distributed l's and O's, the overhead is 83.5%. [Abel 1986] Kermit is a popular program using Character Prefixing for binary file transfer. [Da Cruz 1984-2]

RADIX 64

Radix 64 partitions three consecutive bytes (24-bits) into four 6-bit units. Each 6-bit sequence (with 32 is added to avoid control characters) is converted to its corresponding ASCII character. The mathematical transformation is from binary (base 2) to base 64. Table 3 shows the encoding scheme.

The overhead of Radix 64 is 16.7% [(7-6)/6 bits], because a 7-bit ASCII character is used to carry six meaningful bits. When parity is added, the total overhead is 33.3% [(8-6)/6 bits].

Uuencode [UCB 1980], found on most UNIX systems, uses the Radix 64 encoding scheme.

TABLE3 :
Binary Encoding with Radix 64

Bit Pattern	Numeric Equivalent	Representation
000000	0	SPACE
000001	1	!
000010	2	"
.	.	.
.	.	.
111101	61	}
111110	62	^
111111	63	

RADIX 95

Radix 95 uses all printable ASCII characters to carry meaningful code. The 31 printable characters in excess of Radix 64 requirements are used to represent designated 7-bit sequences. In addition, since there are exactly two 7-bit combinations such that the first six bits are the same, there are no first six bits of the extra 31 characters. Each 6-bit segment becomes a 7-bit combination when a 0 or 1 is appended to it. 62 (2x31) characters can actually be used to represent 7-bit segments. Table 5 shows the encoding scheme used for Radix 95. The theoretical overhead for Radix 95 is shown in Table 4.

Table 4 : Theoretical Overhead of Radix 95 [Renka 1987]

Let N6 = Number of 6-bit strings in a random binary data file. (Values of range 31-63)
Let N7 = Number of 7-bit strings in a random binary data file.
(These have lower 6-bit values in the range O-30 and either 0 or 1 as the 7th bit)

then : $bs = 6(N6) + 7(N7)$

$be = 7(N6 + N7)$

$Oe = \frac{be - bs}{bs}$

therefore : $\frac{7[N6 + N7] - [6(N6) + 7(N7)]}{6(N6) + 7(N7)}$

$= \frac{N6}{6(N6) + 7(N7)} = \frac{1}{6 + 7(N7/N6)}$

Let p = Probability that a random 6-bit string has a value in the range O-30 **p = 31164**
Let q = 1 - p **q = 33164**

For a theoretical random binary file :
$N7/N6 = p/q = [(31/64) / (33/64)] = 31/33$

By Substituting :
$Oe = \frac{1}{6 + 7(N7/N6)} = \frac{1}{6 + 7(31/33)} = 7.95\%$

If begun with be = 8 (N6 + N7 **),** to allow for parity, **then a similar computation yields 23.23 % overhead.**

TABLE5 :
Binary Encoding with Radix 95

Bit Pattern	Numeric Eauivalent	Representation
0000000	0	**SPACE**
0000001	1	!
.	.	.
0011110	30	>
011111	31	?
100000	32	@
100001	33	A
.	.	.
111111	63	-
1000000	64	`
1000001	65	a
1000010	66	b
.	.	.
1011101	93	}
1011110	94	~

RADIX 95 Encoding/Decoding [Abel 1986]

Common practice is to view a file as a collection of bytes of characters. By taking the input file as a continuous string of bits, it is possible to break the file into segments of fixed or variable lengths. As long as the segments are kept in proper order during encoding and decoding, the transfer takes place correctly .

First take all 64 6-bit combinations as 6-bit binary numbers, such that all combinations are enumerated by their binary value (000000 = 0, **000001 = 1, etc.).** To make use of the remaining 31 printable character possibilities, the first 31 of the 6-bit numbers are replaced by 7-bit numbers **(62 total)** created by including a 0 or 1 as the seventh bit.

To encode, six bits are collected from the input file and are assigned place values in ascending order (1,2,4,8) to make a number. If the number is O-30 inclusive, then the next contiguous bit (0 or 1, with a place value of 2^7) is added to the number. The number is then added to 32 to give the ASCII value of a printable character, which is then written to the output file.

Since it is not possible to determine beforehand the number of bits that will be encoded, there will be between zero and five bits remaining at the end of the file when end-of-file is read. This last (short) bit sequence will be written as a full ASCII character, so a special provision must be made to prevent the decoding program from appending extra O's to the output file.

One extra character is written after the last transmission, representing data from the file. This last character specifies how many bits are to be extracted from the preceding character. Example : if the four bits 0110 (value 6) remain at the end of a file, then the character & (6+32) is written to carry the data, followed by the character $ (4+32) to indicate that only four of the seven bits carried by & are to be extracted upon decoding.

To decode, characters are read from the encoded file one at a time, and converted to a number by subtracting 32 from each. If the number is O-30 or 64-94, then the seven bits of the number from the least to most significant are written. 32 is subtracted from the last character decoded, and the number which results is used to determine how many bits (least to most significant) are to be extracted from the next-to-last character.

Benchmarks

The following are benchmarks for Radix 95 vs Radix 64 [Yu 1987]

BENCHMARKS for Radix 95 (including parity")

TEST FILE	Oriainal Size	Result Size	Overhead
objcode	22924	26830	17.04%
objcode	45848	53658	17.03%
objcode	91696	107314	17.03%
source	41796	50798	21.54%
source	56040	68162	21.63%
random	40000	49340	23.35%
random	80000	98712	23.39%

BENCHMARKS Radix for 64 (including parity")

TEST FILE	Oriainal Size	Result Size	Overhead
objcode	22924	30565	33.33%
objcode	45848	61130	33.33%
objcode	91696	122261	33.33%
source	41796	55727	33.33%
source	56040	74719	33.33%
random	40000	53333	33.33%
random	200000	266666	33.33%

objcode - Object Code generated by C compiler.
source - C source Code.
random - random binary digits.
† - Forced by Disk Storage Method.

Radix 95 encoding for random binary data produced the greatest overhead of the three types of test data included for this scheme. Also as the two tables show, Radix 95 produced less overhead than did Radix 64 in all three test groups.

Discussion

Radix 95 offers greater efficiency than does other commonly available conversion met hods However, encoding overhead is just one of the factors that determines the overall efficiency. Processing time for the conversion must be kept in mind. At some point the cost in file size overhead will be offset by the amount of computation time required. It has been suggested that Radix 95^2 = 9025 paired

printable characters or 95^3 = 857,375 triplets **(even more time consuming)** be investigated for further reduction in transmission overhead.

Recommended File Formats

RADIX 95

The following is a proposed standard format for Radix 95 files after encoding :

1. Top line of a Radix 95 file will read :
   (RADIX 95 - [FILENAME : DATE])
   Filename is the first 8 characters of file name
   Date is - 00/00/00 [Month/Date/Year]

2. Each line of Radix produced will be 70 characters long. No other information should appear after the last character on the line.

3. The file ends with :
   (RADIX 95 - END FILENAME).
   Filename is the first 8 characters of file name

4. A Radix 95 conversion program should be designed to search a file's first 10 lines **(one that is to be converted back to 8-bit data)** for the **"(RADIX 95"** before attempting a file conversion. If the program cannot find the correct line within the first 10 lines, then the program should abort.

File Splitting

The following is a proposed standard form for splitting Radix 95 files.

1. The first line of each split file will be :
   (FILENAME.# of #)
   'Filename' is the first 8 characters of the actual file name to be split.
   '.#' is the part of the whole file.
   'of #' is how many different sections.

2. The file ends with :
   (END - FILENAME.# of #)

This will allow the user to specify the first file of a number for the file splitter program to read. Then the file splitter will attempt to use the FILENAME.# convention to reconstruct the file for the user.

User Protocols

In this example, let's suppose KB5EWV wishes to send a data file to WD5IVD.

1. Start with a 65K binary file.
2. Compress the 65K binary file = 39K
   (Assuming 40% compression)
3. Encode the 39K file = 46.8K
   (Assuming a 20% Overhead with Radix 95)
   At this point you have saved 18.2K and you have a completely ASCII file.
4. Determine if the file needs to be split.
   a. **HF SKIP-NET forwarding - break into 5K** or smaller **segments. That would leave 10 files to be transmitted over a period of days. Remember that the major flow of** SKIPNET **is 300 baud on HF. This accounts for the small size of messages.**
   b **VHF forwarding - break into 30K** or smaller segments. **Local users will want to consult with their local BBS system operator to determine better message sizes. High-speed networks would be able to handle larger messages than lower speed network connections.**
   c. **Point-to-Point - No need to split a file you intend to send all at one time.**
5. Send File(s).
   Type : Private to KB5EWV @ BBS
   Title of : **RDX - FILENAME.# of # TYPE**
   Type is the data compression program used.
   Examples : SIT - Mac Stuff-It
   ARC - IBM ARC
   PIT - Mac Packet-It
6. Recipient receives file(s).
7. Recipient joins file(s).
8. Recipient Decodes **(Radix 95)** file.
9. Recipient removes compression.

Summary

The usage of Radix 95, data **compression,** and file splitting would allow amateurs greater flexibility in sending information involving 8-bit data across the amateur packet radio network. Radix 95 would allow an amateur to load data files onto existing BBS's for local reading or for message forwarding while reducing the amount of traffic over the network compared to if the same files were sent in their original state. Currently there is no agreement on the transfer of 8-bit data. Radix 95 would be an optimal choice in 8-bit to 7-bit conversion for data transfers that require the data be in 7-bit format.

Bibliography

Abel 1986 Abel, J. Alex and Gerald Knezek. Binary to Text File Conversion Using RADIX 95. Department of Computer Science, North Texas State University, 1986.

Brown 1984 Brown, Eric and Art Wilcox. Communications Features Explained. PC World. September 1984, pp. 170-l 77.

Da Cruz 1984-1 Da Cruz, Frank and Bill Catchings. Kermit: A File-Transfer Protocol for Universities. Part 1 : Design Considerations and Specifications. Byte, June 1984, pp. 225-278.

Da Cruz 1984-2 Da Cruz, Frank and Bill Catchings. Kermit : A File-Transfer Protocol for Universities. Part 2: States and Transitions, Heuristic Rules, and Example. Byte, July 1984, pp. 143-l 45, 400-403.

Renka 1987 Renka, Robert. Theoretical Overhead for Radix 95 Encodina. Excerpt from : Binary Encoding Benchmarks Radix 64 vs Radix 95 [Yu1987]. 11 th Annual Computer Science Conference, Federation of North Texas Area Universities. Department of Computer Science, North Texas State University, 1987.

Stone 1984 Stone, M. David. Picking the Proper Protocol. PC Maaazine, vol. 7, no. 4, June 11, 1985, pp. 355360.

UCB 1980 UNIX™ Programmer's Manual 7th Ed, Virtual VAX-l 1 version, November 1980, Computer Science Division, Department of Electrical Engineering and Computer Science. Berkeley, California: University of California, 1980.

Yu 1987 Yu, Carol, Gerald Knezek, and Jeff Carruth. Binary Encodina Benchmarks: Radix 64 vs Radix 95, North Texas State University Department of Computer Science, 1987.

RADIX 95 Source

```
/* RADIX 95 ENCODE    Jeff Carruth & Greg Jones 1988 ● */
#include <stdio.h>

main()
{
      unsigned long buffer = 0;
      int           temp;
      short         bits = 0;

      while ((temp = getchar()) != EOF) {
            buffer = (buffer << 8 ) | temp;
            bits t= 8;
            while (bits >= 7) {
                  if ((temp = buffer >> (bits - 6)) > 30) {
                        putchar(temp  +  32);
                        bits -= 6;
                  }
                  else {
                        temp |= ((buffer >> (bits - 7)) & 1) << 6;
                        putchar(temp + 32);
                        bits -= 7;
                  }
                  buffer &= ~(~0L << bits);
            }
      }
      putchar(buffer t 32);
      putchar(bits  +  32);
}
```

```
/* RADIX 95 DECODE    Jeff Carruth & Greg Jones 1988 ● */
#include <stdio.h>
#define getbyte(a,b,c) (a = b, b = c - 32, c = getchar())

main()
{
      char          current,  next;
      int           i, next2;
      unsigned long bit_buf = 0;
      short         in_buf = 0, in-current;

      if ((getbyte(current, next, next2)) == EOF) {
            fprintf(stderr,
                  "decode: not enough bytes in input.\n");
            return(-1);
      }
      if ((getbyte(current, next, next2)) == EOF) {
            fprintf(stderr,
                  "decode: not enough bytes in input.\n");
            return(-1);
      }

      while (getbyte(current, next, next2) != EOF) {
            in-current = ((current > 30) && (current <  64)) ? 6 : 7;

            bit-buf = (bit-buf << 6) | (current & ~(~0L << 6));
            in-buf += 6;

            if (in-current == 7) {
                  bit_buf = (bit-buf << 1) | (current >> 6);
                  ++in_buf;
            }
```