

RADIX 95^n : BINARY-TO-TEXT DATA CONVERSION

James G. Jones, B.S.

APPROVED:

Major Professor

Committee Member

Committee Member

Dean of the Robert B. Toulouse School of
Graduate Studies

Jones, James G., Radix 95ⁿ: Binary-to-Text Data Conversion.

Master of Science (Interdisciplinary Studies), June, 1991, 37 pp, 19 figures, bibliography.

This paper presents Radix 95ⁿ, a binary to text data conversion algorithm. Radix 95ⁿ (base 95) is a variable length encoding scheme that offers slightly better efficiency than is available with conventional fixed-length encoding procedures. Radix 95ⁿ advances previous techniques by allowing a greater pool of 7-bit combinations to be made available for 8-bit data translation. Since 8-bit data (i.e. binary files) can prove to be difficult to transfer over 7-bit networks, the Radix 95ⁿ conversion technique provides a way to convert data such as compiled programs or graphic images to printable ASCII characters and allows for their transfer over 7-bit networks.

Research has been completed and published on Radix 95¹, but no work has been done on the more complex combinations of Radix 95ⁿ. Established data conversion techniques are outlined. Radix 95ⁿ will be discussed with examples of overhead and how efficiency varies through greater combinations. Radix 95ⁿ is specifically a conversion technique, not a protocol, and discussion is limited to the conversion technique. A protocol, similar to KERMIT, could be developed in the future using a Radix 95ⁿ conversion scheme.

RADIX 95ⁿ: BINARY-TO-TEXT DATA CONVERSION

THESIS

Presented to the Graduate Council of The
University of North Texas in Partial
Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

By

James G. Jones, B.S.

Denton, Texas

August, 1991

Copyright by

James G. Jones

1991

TABLE OF CONTENTS

	Page
LIST OF FIGURES	v
Chapter	
I. INTRODUCTION	1
Conversion Techniques	
Overhead	
Radix 95^1 - Radix 95^n	
II. ESTABLISHED DATA CONVERSION TECHNIQUES	8
Hex Encoding	
Character Prefixing	
Radix 64 Encoding	
Radix 85 Encoding	
ABE Encoding	
III. RADIX 95^1	15
Radix 95^1 Encoding/Decoding	
Radix 95^1 Benchmarks	
IV. RADIX 95^n	22
Variable Length Encoding	
Radix 95^n Encoding/Decoding	
Radix 95^n Overhead	
Radix 95^{1-3} Benchmarks	
V. CONCLUSION	32
BIBLIOGRAPHY	33

LIST OF FIGURES

1.1	Equation for Overhead Calculation	6
2.1	Hex Encoding Scheme	9
2.2	Character Prefixing Encoding Scheme	11
2.3	UUENCODE Encoding Scheme	13
2.4	Benchmarks of ABE1 and ABE2	14
3.1	Radix 95^1 Encoding Scheme	16
3.2	Theoretical Overhead of Radix 95^1 without Parity	17
3.3	Theoretical Overhead of Radix 95^1 with Parity	17
3.4	Diagram of Encoding Process for Radix 95^1	19
3.5	Benchmark Comparison of Radix 64 vs. Radix 95^1	21
4.1	Radix 95^{1-5} Variable Length Encoding	23
4.2	Bit Patterns unused in Radix 95^n Representation	25
4.3	Parameters for Radix 95^n Encoding	25
4.4	Diagram of Encoding Process for Radix 95^3	27
4.5	Theoretical Overhead for Radix 95^n	28
4.6	Calculation for $R95^{1-5}$ with and without Parity	39
4.7	Graph of Overhead for $R95^{1-5}$ with and without Parity	39
4.8	Overhead for Radix 95^{1-21} without Parity	30
4.9	Benchmarks of Radix 95^{1-3}	31

CHAPTER I

INTRODUCTION

8-bit data (i.e. binary files) can prove to be difficult to transfer over 7-bit networks. The Radix 95ⁿ conversion technique provides a way to convert data such as compiled programs or graphic images to printable ASCII characters and allow their transfer over 7-bit networks. 8-bit networks, capable of transparent binary data transfer, are becoming the standard; however, there are still many networks that use 7-bit data transfer. Some examples of these networks are UNIX UUCP (UCB 1980), the amateur packet radio network (Fox 1984), MCI Mail (Mefford 1989), and the Texas Education Association TENET system (Stout 1991). By using a text data conversion program before transmission on these networks, potential problems can be avoided. Radix 95ⁿ is a variable length encoding scheme that offers better efficiency than is available with conventional fixed-length encoding procedures.

Radix 95¹, based on the 95 printable characters found in the first 127 ASCII characters, was researched by Abel and Knezek under partial funding from Moore Business Systems in 1986 (Abel and Knezek 1986). A formulated statistical overhead calculation for Radix 95¹ (Renka 1987) and benchmarking of Radix 95¹ were both completed in 1987 (Yu, Knezek, Carruth 1987). This thesis will build upon research into Radix 95, by investigating R95ⁿ and its issues as first described in Abel's conclusion on Radix 95¹. (Abel and Knezek 1986)

Binary to ASCII conversion is relevant to many existing technologies, such as mainframe connectivity, personal computer connectivity, and certain types of data networks. Connectivity between mainframe and personal computers using modems is an important issue for many businesses. Several methods exist for supporting these connections, such as Hayes AutoSync, UDS SyncUp, protocol converters, and others. Protocol converters are still the dominant means of providing connectivity and are used between these systems to handle the conversion between different communications protocols.

A binary to text data conversation technique can be used to transfer files between a mainframe and personal computer without problems. (da Cruz 87) IBM mainframes support EBCDIC instead of ASCII, thus a protocol converter must handle the conversion between EBCDIC and extended ASCII. A protocol converter does certain transformations to make the two systems compatible. Protocol converters typically support a sub-set of standard ASCII character set; therefore, the characters “!`[\]^|~ SPACE” must be avoided for transparent transfer. (Adler 1991) This makes Radix 95 incompatible with this type of transfer. Radix 85 handles compatibility issues between systems that have this problem. This is because the additional 10 printable ASCII characters in excess of those in Radix 85 can be used to avoid the characters (“!`[\]^|~ SPACE”) that cause problems.

MS-KERMIT is an example of a personal computer connectivity program that does printable ASCII conversion. MS-KERMIT grew from the KERMIT file transfer program that embodied a simple terminal emulator

into a complex and powerful communications program. (Gianone, da Cruz, and Douppnik 1989) MS-KERMIT performs two major functions — terminal emulation and file transfer. MS-KERMIT can be executed on a mainframe, personal computer, or network port to allow transparent flow of characters that any of the three might have difficulty transferring.

The amateur packet radio network, based on the AX.25 protocol, is an example of a network that is 8-bit based, but has other difficulties with 8-bit data exchange. The AX.25 protocol allows for 8-bit data transfer, but a TNC (Terminal Node Controller) does not provide a simple interface for a user to send and receive 8-bit data. The converse mode (7-bit data) is supported by all amateur networks and applications. The user interface of a TNC allows for the simple use of 7-bit data transfer instead of 8-bit data transfer. No special characters need to be avoided when using the amateur radio network, but the STREAMSW command on the TNC must be set to a non-printable ASCII character like TAB (\$09). This is to prevent the TNC from dropping the corresponding STREAMSW character and the following channel number, thus losing two characters. (Jones and Knezek 1988; Jones and Saengrussamee 1989) The gradual appearance of packet radio programs (i.e. DED, YAPP) replacing standard modem programs (i.e. PROCOMM, X-TALK) is allowing users to send binary files more effectively, but this trend has been slow. Protocols requiring end-to-end acknowledgments, like XMODEM, do not function well over current amateur packet radio networks, because these telecommunication based protocols require end-to-end responses faster than can be supported by the amateur network. Phil Karn's KA9Q TCP/IP package (Karn 1985; Karn 1987) has been gaining

widespread usage by the amateur community. This package is designed for the slower speed radio links and has been optimized to work more efficiently than standard TCP/IP packages. The KA9Q package allows for transparent 8-bit data operations and provides some unique interconnections to established commercial/educational networks.

Another example where the conversion of binary data to ASCII data would be beneficial is certain types of data compression methods used by modems. Level 5 of the Microcom Network Protocol (MNP-5) implements a data compression method based on a modified run-length Huffman encoding, which produces best performance on English language-based data and worst performance on compressed binary data. (TIA 1988) Some modems, which do not implement anti-compressing code, increase the amount of data being sent. A text data conversion done before transmission could show better performance for binary transfers.

Proof-of-concept testing of Radix 95¹ has taken place over the amateur packet radio network (Jones and Saengrussamee 1989), NASA's ATS-3 satellite (Mukaisa, Jones, et al. 1989; Knezek and Jones 1989), UNIX mail, BITNET mail, and the Japanese Fuji-Oscar 20 satellite. These tests were accomplished using a program developed by the author. (Jones and Knezek 1988) No errors were encountered in these tests, but known problems do exist over certain networks. For example, InterNet mailers are known to corrupt certain text sequences if they are in specific areas of a mail message. The benchmarking done in 1987 showed that Radix 95¹ had additional savings in overhead as compared to more traditional fixed-length methods. (Yu, Knezek, and Carruth 1987)

Conversion Techniques

The basic rationale for any text data conversion method is that a file transfer might fail or the file might be corrupted before the file reaches its destination, whenever the transfer assumes that the 8th-bit of a byte is available for data. There are three common steps involved in converting 8-bit into 7-bit data and then transferring the data from one point to another. The first step is to translate a sequence of bits into printable ASCII data. The next step is to transmit the ASCII data to the system at the other end. The final step is for the system at the other end to convert the ASCII data back into its original binary form. (Abel and Knezek 1986)

The problems facing 8-bit data transmission are many. These problems include how to transmit streams of eight or more binary digits through a network that might have problems handling the data, how to pass certain 7-bit sequences through without having the sequence interpreted as control characters, and how to pass flow control (DC1, DC3), padding (NULL, DEL), transfer of control (ESC), or any of the other non-printable 7-bit characters across a link. This class of difficulties can be summarized by saying that any sequence of data bits that could represent a control character should not be sent unless some special provision to mask it as an ASCII character is made. (Stone 1985; Brown and Wilcox 1984; da Cruz and Catchings 1984a)

Other issues involved with efficient conversion techniques include how to avoid significant amounts of overhead that may be generated during the conversion phase and how to avoid creating excessive computational time during the conversion process. Common data conversion methods

overcome the transmission of binary data and avoid sending special control characters, but introduce significant amounts of overhead when doing so.

(Abel and Knezek 1986)

Overhead

Overhead is the measure of how many extra bits are generated after text data encoding occurs. Figure 1.1 contains the equation for overhead. The major concern with data conversion overhead is the amount of time required to transmit and store the additional overhead generated. With the increase of data transmission speeds, concern for conversion overhead is becoming relatively small. Overhead remains constant at any speed of data transmission; however, speed does affect the amount of time it takes to complete the data transfer. A file transmitted at 300 bit-per-second takes considerably longer to transfer than the same file transferred at 10-Megabits. As significant amounts of overhead are added to the file being transferred, the amount of time at slower speeds increases.

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

O_e = % encoding overhead.

b_s = number of bits in the source.

b_e = number of bits after encoding.

Figure 1.1: Equation for Overhead Calculation

Radix 95¹ - Radix 95ⁿ

Radix 95¹ uses 95 of the 128 characters in the standard ASCII set to represent data. Control characters and the escape character are excluded. Radix 95ⁿ uses combinations of the standard printable ASCII set (32-126) to make more combinations possible for the translation of source data into encoded text data. The use of Radix 95ⁿ was first suggested in the conclusion of Abel's Radix 95¹ research. (Abel and Knezek 1986) The premise of Radix 95ⁿ is that the more combinations available for data representation, the more it should allow the overhead of the encoded output to be lowered. This research has shown that Radix 95ⁿ produces no appreciable increase in efficiency over R95¹. Radix 95ⁿ has some unique qualities that will be described in Chapter IV.

CHAPTER II

ESTABLISHED DATA CONVERSION TECHNIQUES

Several methods have been conceived for converting 8-bit data into a form that can be transferred successfully. In most cases, the reason for converting the data into a 7-bit representation is that the medium used for data transfer is not capable of transferring the data in its original form. Several conversion methods will be discussed to create a basis for comparison of Radix 95ⁿ to these established methods. These methods include Hex encoding (BINHEX, DEBUGSCR), Character Prefixing (KERMIT), Radix 64 encoding (UUENCODE), Radix 85 encoding (BTOA, ABE2, SHIP), and ABE, a modified Radix 94 encoding scheme.

Hex Encoding

Hex encoding views each binary octet (byte) as two contiguous 4-bit sequences (nibbles). Each nibble is translated into its corresponding hexadecimal character by taking the 4-bit sequence (nibble) and placing it into the lower half of its own byte. The value 30h is added to the values 0-9 and 37h is added to values 10-15 to transform the hex value into an ASCII character representation. The mathematical transformation is from Binary (base 2) to Hexadecimal (base 16). After transmission, the 4-bit sequences (nibbles) are recombined into the original data. Figure 2.1 shows the Hex Encoding scheme. (Mefford 1989)

Bit pattern	Shift	Hex Value
0000	+30h	0
0001	.	1
0010	.	2
0011	.	3
0100	.	4
0101	.	5
0110	.	6
0111	.	7
1000	.	8
1001	.	9
1010	+37h	A
1011	.	B
1100	.	C
1101	.	D
1110	.	E
1111	.	F

Figure 2.1: Hex Encoding Scheme

The overhead for Hex encoding is 75% $[(7-4)/4 = 3/4 \text{ bits}]$. With the addition of the accompanying parity bit, implementations use 16-bits to transmit 8-bits of information. The total data encoding plus parity-bit overhead is 100% $[(8-4)/4 = 4/4 \text{ bits}]$. (Abel and Knezek 1986; da Cruz and Catchings 1984a) BINHEX for the Apple Macintosh™ is a common program which implements Hex Encoding. (Lempereur 1985) DEBUGSCR, for personal computer use, converts files of up to 60K-bytes into ASCII data using Hex encoding. DEBUGSCR creates a script file with instructions that tells DOS' DEBUG utility how to recreate the original file. (Meffod 1989) Files created using most conversion methods generate higher overheads than their theoretical values, since these programs include CR/LFs and other information used when decoding.

Character Prefixing

Eight-bit Character Prefixing checks every sequence of eight bits and sends it unaltered if the bit stream corresponds to a printable ASCII text character. If the eighth bit is a 1, then the character “&” 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 that is printable (by complementing the seventh bit) and the transformed character is preceded by the character “#”. The special prefix characters are themselves prefixed for transmission. (da Cruz and Catchings 1984b; da Cruz 1987) Figure 2.2 shows the encoding scheme for eight-bit character prefixing.

The overhead for character prefixing is dependent on the type of file transmitted. Text files are efficient, since few characters need to be prefixed. Binary files consisting of randomly distributed 1's and 0's have an overhead of 83.5%. (Abel and Knezek 1986) KERMIT uses character prefixing for data conversion. KERMIT is one of the most common communication packages used for transferring files between computers. (Frاند 1990) The KERMIT protocol is the only popular 7-bit transfer method available, in contrast to the predominant 8-bit methods for file transmissions, like XMODEM, YMODEM, and Compuserve B. (Quarterman 1990) KERMIT is versatile, consisting of about 400 different programs to date, including implementations for Macintosh, Unix, Novell NetWare, and other computer operating systems. All told, KERMIT can enable up to 80,000 combinations of computers to transfer their files between one another. (Miles 1990)

	8-bit Pattern	Numeric	7-bit Transformation	Transmission
# Prefixed	00000000	0	1000000	# @
.	00000001	1	1000001	# A
.
.	00011111	31	1011111	# -
Unaltered	00100000	32	0100000	SPACE
.	00100001	33	0100001	!
.	00100010	34	0100010	“
.	00100011	35	1100011	# #
.
.	00100110	38	1100110	# &
.	00100111	39	0100111	,
.
.	01111110	126	1111110	~
# Prefixed	01111111	127	0111111	# ?
&# Prefixed	10000000	128	1000000	& # @
.	10000001	129	1000001	& # A
.
& Prefixed	10100000	160	0100000	& SPACE
.	10100001	161	0100001	& !
.	10100010	162	0100010	& “
.	10100011	163	0100011	& # #
.
.	10100110	166	0100110	& # &
.	10100111	167	0100111	& ,
.
.	11111110	254	0111110	& ~
&# Prefixed	11111111	255	0111111	& # ?

Figure 2.2: Character Prefixing Encoding Scheme
Based on eight-bit prefix encoding. (da Cruz 1987)

Radix 64 Encoding

Radix 64 partitions three consecutive bytes (24-bits) into four 6-bit units. Each 6-bit sequence, with 32 added to avoid control characters, is converted to its corresponding ASCII text character. The mathematical transformation is from binary (base 2) to base 64. The overhead of Radix 64 is 16.7% $[(28-24)/24 \text{ bits}]$, because a 7-bit ASCII character is used to carry six meaningful bits. When the parity-bit is added, the total overhead is

33.3% [(32-24)/24 bits]. (Abel and Knezek 1986) UUENCODE, found on most UNIX systems, uses the Radix 64 encoding scheme. The SPACE character is avoided by using ASCII characters 33 (“!”) through 96 (“”) for compatibility. UUENCODE includes additional information for recreation and checking on the remote system. (UCB 1990) Figure 2.3 shows the UUENCODE encoding scheme generated from a modified UUENCODE program. The packet driver link protocol-g, supplied with standard UNIX-based systems, allows for 8-bit data transfer. (Chesson 1988) The UUCP (Unix to Unix Copy) program utilizes this protocol for transmitting data between UNIX systems. The problem of 8-bit data transfer using this protocol occurs when in-band signaling (XON/XOFF) is used between the system and telecommunication devices — typically modems. In these cases, data conversion must be used to avoid special in-band signaling characters.

Another example of Radix 64 encoding is XXENCODE, a modified UUENCODE encoding standard. XXENCODE modifies UUENCODE by changing the encoding character set to avoid special characters. This change makes it more compatible with existing software, like mailers and other network applications. The character set used by XXENCODE is “+”, “-”, “0 — 9”, “A — Z”, and “a — z”. (Camp and Howard 1985)

Radix 85 Encoding

Radix 85 partitions four consecutive bytes (32-bits) into five 6-bit units (30-bits). Each 6-bit sequence, with 32 added to avoid control characters, is converted to its ASCII equivalent. (Orost 1991) The two remaining bits (32-30=2) are then shifted and placed with the next sequence of four bytes read. The mathematical transformation is from binary (base 2)

<u>6 Bit Pattern</u>	<u>Num</u>	<u>ASCII</u>	<u>6 Bit Pattern</u>	<u>Num</u>	<u>ASCII</u>	<u>6 Bit Pattern</u>	<u>Num</u>	<u>ASCII</u>
000000	0	`	010110	22	6	101100	44	L
000001	1	!	010111	23	7	101101	45	M
000010	2	"	011000	24	8	101110	46	N
000011	3	#	011001	25	9	101111	47	O
000100	4	\$	011010	26	:	110000	48	P
000101	5	%	011011	27	;	110001	49	Q
000110	6	&	011100	28	<	110010	50	R
000111	7	'	011101	29	=	110011	51	S
001000	8	(011110	30	>	110100	52	T
001001	9)	011111	31	?	110101	53	U
001010	10	*	100000	32	@	110110	54	V
001011	11	+	100001	33	A	110111	55	W
001100	12	,	100010	34	B	111000	56	X
001101	13	-	100011	35	C	111001	57	Y
001110	14	.	100100	36	D	111010	58	Z
001111	15	/	100101	37	E	111011	59	[
010000	16	0	100110	38	F	111100	60	\
010001	17	1	100111	39	G	111101	61]
010010	18	2	101000	40	H	111110	62	^
010011	19	3	101001	41	I	111111	63	_
010100	20	4	101010	42	J			
010101	21	5	101011	43	K			

Figure 2.3: UUENCODE Encoding Scheme

to base 85. The overhead of Radix 85 is 9.4% [(35-32)/32 bits] and when parity is added the overhead with parity-bit is 25% [(40-32)/32 bits]. BTOA (Binary to ASCII), found on most UNIX systems, uses a Radix 86 encoding scheme. BTOA uses “!” through ”u” and “z”. The “z” character is used to represent a 32-bit zero. BTOA includes a header and checksum to the file. (Orost 1991) ABE2, uses a modified Radix 86 conversion method. ABE2 does an analysis of the file before conversion and determines the most frequently occurring sequences and uses these sequences to define the character set and their patterns to be converted. (Templeton 1989) Overhead for ABE2 will be discussed with ABE encoding. SHIP also provides a base 85 encoding method with checksum capability and a mail split routine. (Adler 1991)

ABE Encoding

ABE (ASCII to Binary) is an encoding program that uses a Radix 94 or Radix 86 encoding scheme, depending on the option selected. In the standard ABE1 (Radix 94) encoding, 256 bytes are broken into 3 sets of 86, 86 and 84 bytes. The most frequently found bytes in the file go into sets 0,1, and 2. 86 of the printable ASCII characters are used to encode the members of each set. SPACE is excluded in ABE1, creating 94 character combinations. ABE provides header information that defines information about the encoded files, block headings, sizes, and checksums. A three-character header is attached to every line and can be used to sort the file if it is damaged. The ABE2 encoding (Radix 86) splits 256 bytes into 4 sets of 64 bytes each. ABE2 avoids certain special characters, similar to BTOA. (Templeton 1989)

Benchmarks generated by the author found that ABE's method of pre-analysis for reducing overhead works well for organized patterns of data, but produces higher overhead on more random data, like compressed files. Figure 2.4 contains benchmarks for ABE1 and ABE2, using Radix 95ⁿ benchmark data.

TEST FILE	Original Size	<u>ABE 1 (Radix 94)</u>		<u>ABE 2 (Radix 86)</u>	
		Result Size	Overhead Percentage	Result Size	Overhead Percentage
objcode 1	28,246	36,019	27.52%	37,110	31.38%
objcode 2	310,480	381,646	22.92%	392,155	26.31%
objcode 3	294,470	322,413	9.49%	322,477	9.51%
source 1	168,276	190,353	13.12%	192,948	14.66%
source 2	59,119	65,321	10.49%	65,385	10.60%
random 1	40,000	60,130	50.33%	61,064	52.66%
random 2	200,000	299,406	49.70%	303,449	51.72%

Figure 2.4: Benchmarks of ABE1 and ABE2.

CHAPTER III

RADIX 95¹

Radix 95¹ was researched in 1986 by Abel (Abel and Knezek 1986) and then later in 1987 by Yu. (Yu, Knezek, and Carruth 1987) These works concentrated on theoretical aspects and benchmarking Radix 95¹ respectively. This chapter will discuss the previous research on Radix 95¹ that will provide a foundation from which Radix 95ⁿ will then be discussed.

Radix 95¹ uses the 31 printable characters available beyond those used in Radix 64 to represent designated 7-bit sequences. A sequence, when read from a file, can be either 6-bits or 7-bits in length. The 7th bit is included only if the first 6-bits represent an integer value in the range of 0 to 30. When the integer value is in the range of 0 to 30, a zero (0) or one (1) will be read as the 7th-bit creating two sets of 7-bit sequences and one set of 6-bit sequences. Figure 3.1 shows the encoding scheme used for Radix 95ⁿ. 62 (2 x 31) characters can be used to represent 7-bit segments, with 31 characters being used to represent 6-bit segments. (Abel and Knezek 1986)

The theoretical overhead for Radix 95¹, as originally defined by Renka (Renka 1987), is shown in Figure 3.2. The theoretical overhead for Radix 95ⁿ with parity-bit added is shown in Figure 3.3. As can be seen by comparing figures 3.2 (7.95% overhead) and 3.3 (23.37% overhead), the impact upon encoding efficiency due to the parity-bit is extensive. The addition of the parity-bit in later Radix 95ⁿ combinations will also prove to be significant.

<u>7 Bit Pattern</u>	<u>Num</u>	<u>ASCII</u>	<u>6 Bit Pattern</u>	<u>Num</u>	<u>ASCII</u>	<u>7 Bit Pattern</u>	<u>Num</u>	<u>ASCII</u>
0000000	0	SPACE	0111111	31	?	1000000	64	`
0000001	1	!	1000000	32	@	1000001	65	a
0000010	2	"	1000001	33	A	1000010	66	b
0000011	3	#	1000010	34	B	1000011	67	c
0000100	4	\$	1000011	35	C	1000100	68	d
0000101	5	%	1001000	36	D	1000101	69	e
0000110	6	&	1001001	37	E	1000110	70	f
0000111	7	'	1001010	38	F	1000111	71	g
0001000	8	(1001011	39	G	1001000	72	h
0001001	9)	1010000	40	H	1001001	73	i
0001010	10	*	1010001	41	I	1001010	74	j
0001011	11	+	1010010	42	J	1001011	75	k
0001100	12	,	1010011	43	K	1001100	76	l
0001101	13	-	1010100	44	L	1001101	77	m
0001110	14	.	1010101	45	M	1001110	78	n
0001111	15	/	1010110	46	N	1001111	79	o
0010000	16	0	1010111	47	O	1010000	80	p
0010001	17	1	1100000	48	P	1010001	81	q
0010010	18	2	1100001	49	Q	1010010	82	r
0010011	19	3	1100010	50	R	1010011	83	s
0010100	20	4	1100011	51	S	1010100	84	t
0010101	21	5	1100100	52	T	1010101	85	u
0010110	22	6	1100101	53	U	1010110	86	v
0010111	23	7	1100110	54	V	1010111	87	w
0011000	24	8	1100111	55	W	1011000	88	x
0011001	25	9	1110000	56	X	1011001	89	y
0011010	26	:	1110001	57	Y	1011010	90	z
0011011	27	;	1110010	58	Z	1011011	91	{
0011100	28	<	1110011	59	[1011100	92	
0011101	29	=	1111000	60	\	1011101	93	}
0011110	30	>	1111001	61]	1011110	94	~
			1111010	62	^			
			1111011	63	_			

Figure 3.1: Radix 95¹ Encoding Scheme

Let N6 = The number of 6-bit strings in a random binary data file. (Values of range 31-63)

Let N7 = The number of 7-bit strings in a random binary data file.

(These have lower 6-bit values in the range 0-31 and either 0 or 1 as the 7th bit)

$$\begin{aligned} \text{Then :} \quad bs &= 6(N6) + 7(N7) \\ be &= 7(N6 + N7) \\ Oe &= \frac{be - bs}{bs} \end{aligned}$$

$$\text{Therefore :} \quad \frac{7 [N6 + N7] - [6 (N6) + 7 (N7)]}{6 (N6) + 7 (N7)} = \frac{N6}{6 (N6) + 7 (N7)} = \frac{1}{6 + 7 (N7/N6)}$$

Let p = The probability that a random 6-bit string has a value in the range 0-30. $p = 31/64$

Let q = 1-p. $q = 33/64$

For a theoretical random binary file

$$N7/N6 = p/q = [(31/64) / (33/64)] = 31/33$$

By Substituting 31/33 for N7/N6 then:

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

Figure 3.2: Theoretical overhead of Radix 95¹ without Parity (Renka 1987)

Let N6 = The number of 6-bit strings in a random binary data file. (Values of range 31-63)

Let N7 = The number of 7-bit strings in a random binary data file.

(These have lower 6-bit values in the range 0-31 and either 0 or 1 as the 7th bit)

$$\begin{aligned} \text{Then :} \quad bs &= 6(N6) + 7(N7) \\ bep &= 8(N6 + N7) \quad (8 \text{ replaces } 7 \text{ for parity when encoded}) \\ Oep &= \frac{bep - bs}{bs} \end{aligned}$$

$$\text{Therefore :} \quad \frac{8 [N6 + N7] - [6 (N6) + 7 (N7)]}{6 (N6) + 7 (N7)} = \frac{2N6 + N7}{6 (N6) + 8 (N7)} = \frac{2 + (N7/N6)}{6 + 7 (N7/N6)}$$

Let p = The probability that a random 6-bit string has a value in the range 0-30. $p = 31/64$

Let q = 1-p. $q = 33/64$

For a theoretical random binary file

$$N7/N6 = p/q = [(31/64) / (33/64)] = 31/33$$

By Substituting 31/33 for N7/N6 then

$$Oe^P = \frac{2 + (N7/N6)}{6 + 7 (N7/N6)} = \frac{2 + (31/33)}{6 + 7 (31/33)} = 23.37\%$$

Figure 3.3: Theoretical overhead of Radix 951 with Parity
Derived from Figure 3.2 (Renka 1987)

Radix 95¹ Encoding/Decoding

Every file can be thought of as a binary file composed entirely of bytes of binary ones and zeros. Common practice is to view a text file as a collection of bytes or characters. The values of bytes in a text file are meant to be interpreted using standard printable ASCII text (32-126). The difference between binary and text files is that a text file is interpreted as having characters in the standard ASCII set, while a binary file may represent machine codes, binary values, and other non-ASCII text information. By interpreting any input file as a continuous string of bits, it is possible to break the file into segments of fixed or variable lengths for encoding. Since the segments are kept in proper order during encoding and decoding, the transfer of the encoded data can take place without error. (Abel and Knezek 1987)

Six bits are read from the input file for encoding. The six bits are assigned place values in ascending order from left to right (1,2,4,8) to create a number. If the number is in the range of 0 to 30, the next contiguous bit is added to the number creating a number in the range of 0 to 30 or 64 to 94. 32 is then added to the number to create an ASCII value, which is then written to the output file. (Abel and Knezek 1987) Figure 3.4 shows a diagram of this process.

When the end-of-file is encountered, there will be between zero and five bits remaining to be encoded. This is an effect of the variable length encoding process. The bit sequence (0-5) is written as a full ASCII character; however, only a certain portion of the byte contains meaningful data. One additional character is written after the last encoded character, representing the number of bits to be extracted from the preceding character.

Example with 7th bit as a 0:

001010 0...

								7th bit
Data:	0	0	1	0	1	0		0
Value	1	2	4	8	16	32		64
<hr/>								
Count			4	+	16		=	20
Since value is between 0 and 31 take 7th bit								20 + 0 = 20
								+ 32 = 52 '4'

Example with 7th bit as a 1:

001010 1...

								7th bit
Data:	0	0	1	0	1	0		1
Value	1	2	4	8	16	32		64
<hr/>								
Count			4	+	16		=	20
Since value is between 0 and 31 take 7th bit								20 + 64 = 84
								+ 32 = 116 't'

Figure 3.4: Diagram of Encoding Process for Radix 95¹

This prevents additional bits from being added during decoding. An example of this process would be when 01101 (value 13) remains to be encoded. The character '-' (13+32) is written, followed by the character '%' (5+32) to indicate that only five of the seven bits carried by '-' are to be extracted upon decoding. (Abel and Knezek 1986)

When decoding, characters are read from the encoded file one at a time. 32 is subtracted from each character and a number is generated. If the number is in the range of 0-30 or 64-94, then the seven bits of the number from the least to most significant are written. If the number is between 31 and 63 then six bits are written. 32 is subtracted from the last character decoded, and the number that results is used to determine how many bits are to be extracted from the next-to-last character. (Abel and Knezek 1986)

Radix 95¹ Benchmarks

Figure 3.5 shows benchmarks for Radix 95¹ vs. Radix 64. (Yu, Knezek, and Carruth 1987) Radix 95¹ encoding for random binary data produced the greatest overhead of the three types of test data. As the figure shows, Radix 95¹ produced less overhead than did Radix 64 in all three test groups. These benchmarks also discovered that the parity-bit is present during storage, due to the byte boundary imposed for ASCII text storage in most computer systems. (Yu, Knezek, and Carruth 1987)

BENCHMARKS for Radix 95

TEST FILE	Original Size	Result Size†	Overhead
objcode	22,924	26,830	17.04%
objcode	45,848	53,658	17.03%
objcode	91,696	107,314	17.03%
source	41,796	50,798	21.54%
source 2	56,040	68,162	21.63%
random 1	40,000	49,343	23.36%
random 2	200,000	246,826	23.41%

BENCHMARKS Radix for 64

TEST FILE	Original Size	Result Size†	Overhead
objcode	22,924	30,565	33.33%
objcode	45,848	61,130	33.33%
objcode	91,696	122,261	33.33%
source	41,796	55,727	33.33%
source 2	56,040	74,719	33.33%
random 1	40,000	53,333	33.33%
random 2	200,000	266,666	33.33%

objcode - Object Code generated by C compiler.

source - C source Code. Source 2 is used in the R95n Benchmark.

random - random binary digits. Random 1 and 2 are used in the R95n Benchmark.

† - Forced by Disk Storage Method. Based on Byte Boundaries.

Figure 3.5: Benchmark Comparison of Radix 64 vs. Radix 95¹
(Yu, Knezek, and Carruth 1987)

CHAPTER IV

RADIX 95^n

Radix 95^n (base 95) is a variable length encoding scheme that offers improved efficiency over that which is available with conventional fixed-length encoding procedures. Radix 95^n advances previous techniques by allowing a greater pool of 7-bit combinations to be made available for 8-bit data translation. Radix 95^n creates longer bit stream patterns by using combinations of the 95 printable ASCII characters that are the basis for Radix 95^1 . This chapter will describe the characteristics of Radix 95^n .

Variable Length Encoding

The encoding lengths used by Radix 95^n vary according to the value of n . Figure 4.1 shows how the number of 2^k -bit combinations are related to the combinations generated by 95^n . This figure contains the theoretical variable length encoding, the actual variable length encoding, the number of combinations formed by $2^k + (95^n - 2^k)$, and the number of combinations not used in $2^{k+1} - 95^n$ and $2^{7n} - 95^n$. As shown in this figure, the maximum theoretical encoding length is never reached, since 95^n is always smaller than the highest theoretical length [$95^n < (2^{7n} - 95^n)$].

For example, with Radix 95^3 , the number of combinations created is 857,375. The theoretical variable length encoding for Radix 95^3 is 18 [$6(3) = 18$] through 21 [$7(3) = 21$] bits, creating a total of 2,097,152 theoretical

2^k bits	2^k bit Combinations	$R95^n$ Type	$R95^n$ Combinations	VLE	$2^k + (95^n - 2^k)$ Characters	$2^{k+1} - 95^n$ (A) $2^{7n} - 95^n$ (B)
1	2					
2	4					
3	8					
4	16					
5	32					
6	64					
	<---	$R95^1$	95	<u>6,7</u>	$2^6 + 31$	33 (A) 33 (B)
7	128					
8	256					
9	512					
10	1,024					
11	2,048					
12	4,096					
13	8,192					
	<---	$R95^2$	9,025	<u>12,13,14</u>	$2^{13} + 833$	7,359 7,359
14	16,384					
15	32,768					
16	65,536					
17	131,072					
18	262,144					
19	524,288					
	<---	$R95^3$	857,375	<u>18,19,20,21</u>	$2^{19} + 333,087$	191,201 1,239,777
20	1,048,576					
21	2,097,152					
22	4,194,304					
23	8,388,608					
24	16,777,216					
25	33,554,432					
26	67,108,864					
	<---	$R95^4$	81,450,625	<u>24,25,26,27,28</u>	$2^{26} + 14,341,761$	52,767,103 186,984,831
27	134,217,728					
28	268,435,456					
29	536,870,912					
30	1,073,741,824					
31	2,147,483,648					
32	4,294,967,296					
	<---	$R95^5$	7,737,809,375	<u>30,31,32,33,34,35</u>	$2^{32} + 3,442,842,079$	852,125,217 26,621,928,993
33	8,589,934,592					
34	17,179,869,184					
35	34,359,738,368					

Figure 4.1: $R95^{1-5}$ Variable Length Encoding
VLE - Variable Length Encoding. Actual VLE is underlined.

combinations. Figure 4.1 shows that the largest 2^k -bit combination that can fit within $R95^3$ is 2^{19} . This leaves some number of 2^{20} -bit combinations that can be represented by the remainder of unused $R95^3$ combinations. The number of 2^{19} -bit combinations plus 2^{20} -bit combinations that make up $R95^3$ is $2^{19} + (2^{20} - 95^3) = 857,375$ [$524,288 + (1,048,576 - 857,375)$]. Thus 524,288 2^{19} -bit combinations and 333,087 2^{20} -bit combinations can be represented by $R95^3$ (857,375). This leaves 1,239,777 combinations ($2^{21} - 95^3$) of the theoretical variable length combinations not used in $R95^3$ encoding. This example shows that the Radix 95^n encoding algorithm has to define only k and $k+1$ to successfully encode and decode data [$2^k \leq 95^n \leq 2^{k+1}$].

The potential combinations not used represent one type of inefficiency found in Radix 95^n . This inefficiency will be further amplified when the parity-bit is introduced. Figure 4.2 shows a graph of the unused combinations of Radix 95^{1-5} . The percentage of characters not used is derived from the number of total combinations that could be used, but which are not used by the Radix 95^n encoding technique. The percentage of bit patterns unused by the actual VLE of $R95^3$ is 44% ($7,359/16,384$). Figure 4.3 shows the equation for the values displayed in Figure 4.1.

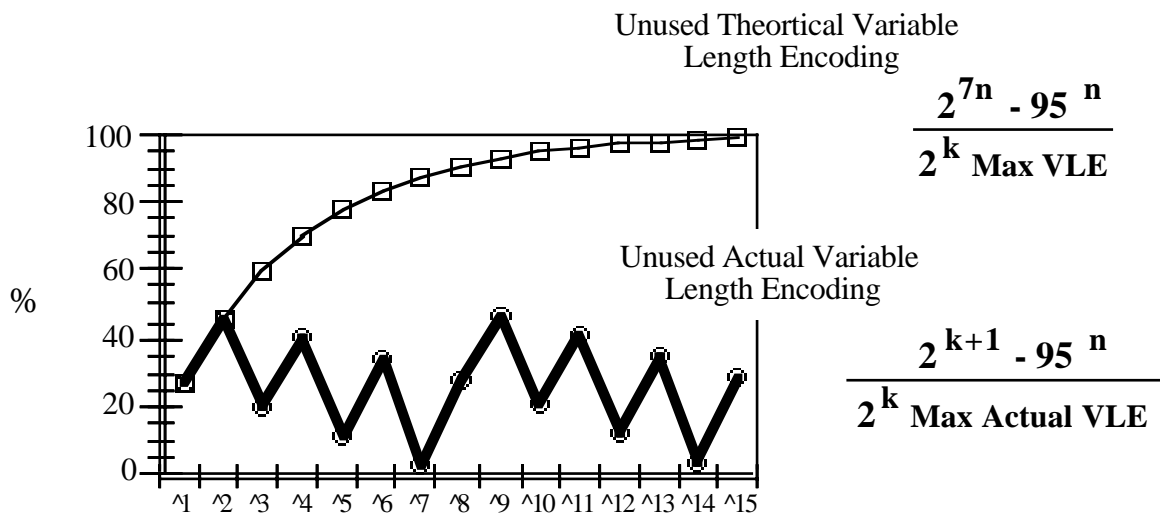


Figure 4.2: Bit Patterns Unused in Radix 95ⁿ Representation.

R95 ¹	R95 ²	R95 ³	R95 ⁴	R95 ⁵	
6bit 64	13bit 8,192	19bit 524,288	26bit 67,108,864	32bit 4,294,967,296	2^k bits
7bit 31	14bit 833	20bit 333,087	27bit 14,341,761	33bit 3,442,842,079	$95^n - 2^k$ bits
[95]	[9,025]	[857,375]	[81,450,625]	[7,737,809,375]	95^n
33	7,359	191,202	52,767,103	852,135,217	$2^{k+1} - 95^n$ bits
33	7,359	1,239,777	186,984,831	26,621,928,993	$2^{7n} - 95^n$ bits

Figure 4.3: Parameters for Radix 95ⁿ Encoding

Radix 95^n Encoding/Decoding

As with the Radix 95^1 example, a file can be considered as a string of 1's and 0's. Radix 95^n requires only two of the variable length encoding bits to be determined for encoding to occur successfully. The first variable length number is the largest 2^k -bit combination that falls within 95^n , with the second being defined as 2^{k+1} . 2^k and 2^{k+1} bits define the variable length encoding bits to be used [$2^k \leq 95^n \leq 2^{k+1}$]. k bits are read from the file and are assigned place values from left to right in ascending order to create a number. If the number lies in a range of 0 through $95^n - 2^k$ then the next contiguous bit is taken and added to the number in the 2^{k+1} place value. The defined number is converted to its Radix 95^n character sequence by taking modulo $95^n - 1$ down to 95^0 for each character. 32 is added to each generated character to place the character in the standard ASCII set (32-126). Figure 4.4 contains an example of this process.

0 through $k-1$ bits will remain after the end-of-file is encountered. The remaining number of bits are encoded in the same manner as $R95^1$, except using an n -character sequence. $R95^n$ could define the ending sequence to be one character instead of an n -character sequence, since one character can hold up to 94 place values.

To decode, n -characters are read. 32 is subtracted from each character and then multiplied from right to left with increasing factors (95^0 , 95^1 , 95^2 , 95^3). If the number is in the range of 0 through $(95^n - 2^1)$ or 2^k through 95^n then $k+1$ bits are written; otherwise k -bits are written. The last character combination in the encode file contains the number of bits to be extracted from the previous set of characters.

Example for $R95^3$ with 20th bit as a 0:

Data: 001010 001010 001010 0...0

Value: 83,220

Since the value is between 0 and 333,087 take the 20th bit, which is 0 - giving 83,220.

$83,220 \bmod 95^{3-1} = 9$ with 1995 remaining = $9 + 32 = 41$	“)”
$1995 \bmod 95 = 21$ with 0 remaining = $21 + 32 = 53$	“5”
$0 = 0 + 32 = 32$	SPACE

Example for $R95^3$ with 20th bit as a 1:

Data: 001010 001010 001010 0...1

Value: 83,220

Since the value is between 0 and 333,087 take the 20th bit, which is 1 - giving 607,508.

$607,508 \bmod 95^{3-1} = 67$ with 2833 remaining = $67 + 32 = 99$	“c”
$2833 \bmod 95 = 29$ with 78 remaining = $29 + 32 = 61$	“=”
$78 = 78 + 32 = 110$	“n”

Figure 4.4: Diagram of Encoding Process for Radix 95^3

Radix 95^n Overhead

Since the premise of Radix 95^n is based on Radix 95^1 , the theoretical equation in Figure 3.2 can be used for all Radix 95^n types. Figure 4.5 shows the theoretical overhead of Radix 95^n with and without parity. By taking the equation in Figure 4.5 and applying it to the numbers listed in Figure 4.1 and Figure 4.3, Figure 4.6 is created showing the overhead calculations of Radix 95^{2-5} . Figure 4.7 shows a graph of the comparison of the calculations made in Figure 4.6 based on percentage overhead. The gain in overhead percentage on each successive type is very low.

Let n = The power of the Radix encode. $[R95^n]$

Let k = The variable length encode bit defined by $2^k \leq 95^n < 2^{k+1}$.

Let N_k = The number of k bit strings in a random binary data file.

Let N_{k+1} = The number of $k+1$ bit strings in a random binary data file.

$$\begin{aligned} \text{Then :} \quad \text{bs} &= k N_k + (k + 1) N_{k+1} \\ \text{be} &= 7n (N_k + N_{k+1}) \\ \text{Oe} &= \frac{\text{be} - \text{bs}}{\text{bs}} \end{aligned}$$

$$\begin{aligned} \text{Therefore:} \quad & \frac{[7n (N_k + N_{k+1})] - [k N_k + (k + 1) N_{k+1}]}{k N_k + (k + 1) N_{k+1}} \\ &= \frac{7n - k + (7n - k - 1) [N_{k+1} / N_k]}{k + (k + 1) [N_{k+1} / N_k]} \end{aligned}$$

If p = The probability that a random N_k bit string is in the range of 0 to $95^n - 2^k - 1$ which is $(95^n - 2^k) / 2^k$

$$\begin{aligned} \text{Then: } N_{k+1} / N_k &= p / (1-p) = (95^n - 2^k) / (2^{k+1} - 95^n) \\ &= \frac{7n - k + [(7n - k - 1) [(95^n - 2^k) / (2^{k+1} - 95^n)]]}{k + [(k + 1) [(N_{k+1} + 1) / N_k]]} \end{aligned}$$

The equation for parity becomes, by substituting 8 for 7 in the equation:

$$\text{Oe}^D = \frac{8n - k + [(8n - k - 1) [(95^n - 2^k) / (2^{k+1} - 95^n)]]}{k + [(k + 1) [(N_{k+1} + 1) / N_k]]}$$

Figure 4.5: Theoretical Overhead for Radix 95^n

$k, k+1$	$R95^n$	95^n	$95^n - 2^k$	$2^{k+1} - 95^n$	$\frac{95^n - 2^k}{2^{k+1} - 95^n}$	Oe	Oe ^P
6,7	1	95	31	33	0.939393939	7.95%	23.37%
13,14	2	9,025	833	7,359	0.113194728	6.86%	22.12%
19,20	3	857,375	333,087	191,201	1.742077709	6.95%	22.23%
26,27	4	81,450,625	14,341,761	52,767,103	0.271793602	6.81%	22.07%
32,33	5	7,737,809,375	3,442,842,079	852,125,217	4.040300663	6.70%	21.95%

Figure 4.6: Calculation for $R95^{1-5}$ With (Oe) and Without Parity (Oe^P).

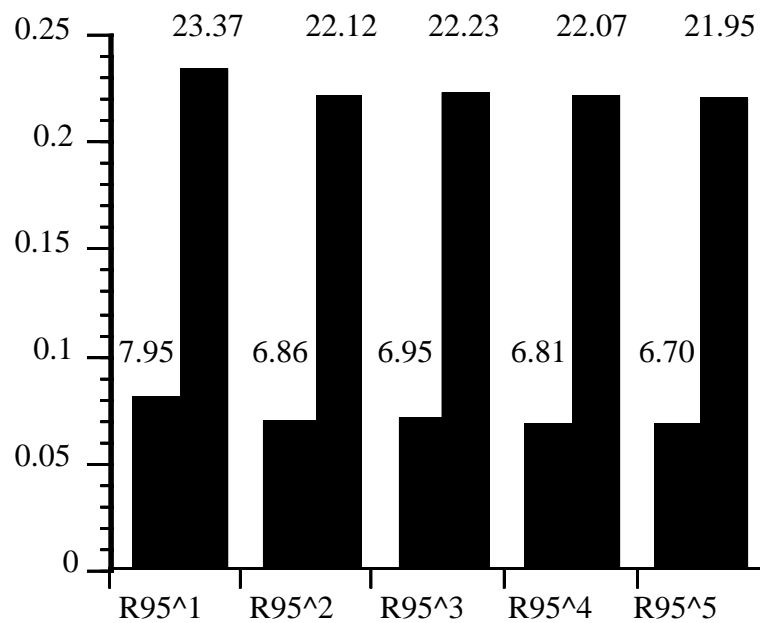


Figure 4.7: Graph of Overhead for $R95^{1-5}$ With and Without Parity.

Legend: Black - Without Parity / Gray - With Parity.

An asymptotic analysis of overhead, provided by Renka, shows that Radix 95^n approaches 6.55% overhead without parity, and 21.77% overhead with parity. Figures 4.6 and 4.7 show that the overhead does not drop in a linear manner, but in a sinewave like manner as it approaches the asymptote. A spreadsheet analysis revealed that the overhead calculation for Radix 95^1 through Radix 95^{50} only approached 6.56% and 21.78% overhead. These optimum overhead figures were produced every 7th power (7,14,21,28...). Figure 4.8 contains a graph of Radix 95^1 through Radix 95^{21} for overhead without parity, showing the cyclic nature of the overhead as it approaches 6.55%. Note that the pattern is similar to the one in Figure 4.2.

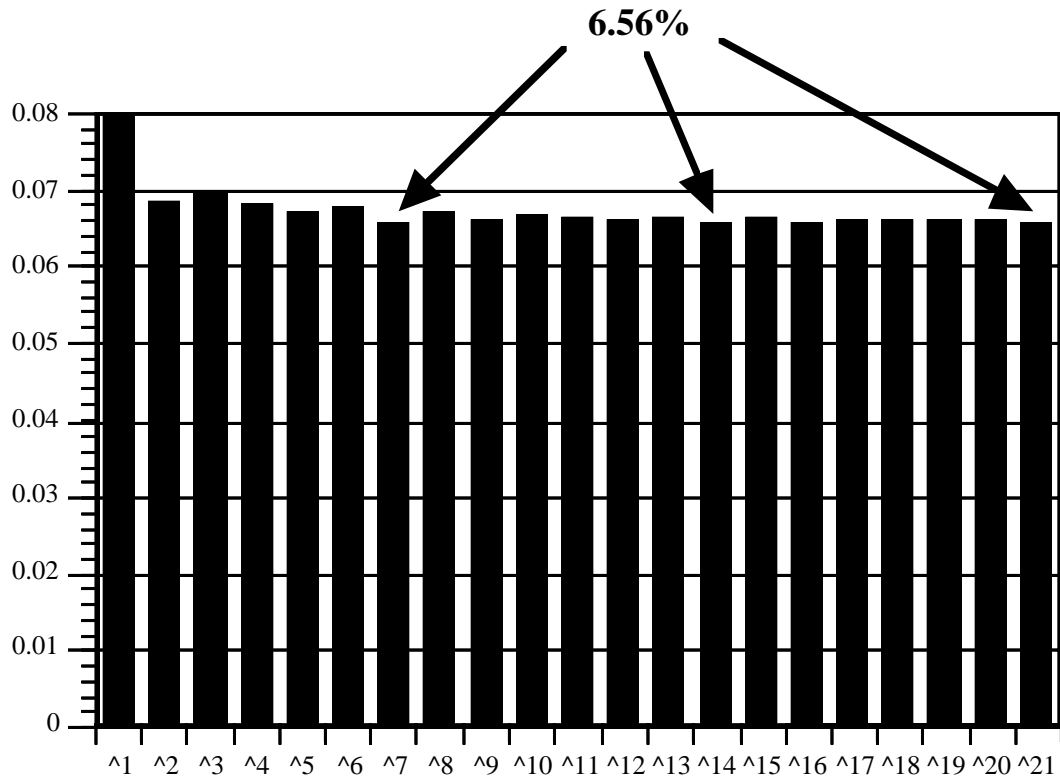


Figure 4.8: Overhead for Radix 95^{1-21} without Parity.

Radix 95ⁿ Benchmarks

The accuracy of the theoretical overhead calculations has been confirmed through encoding software developed for this thesis. Original data files used in benchmarking Radix 95¹ (Figure 3.5) have also been used in benchmarking Radix 95¹⁻³. These examples are source 2, random 1, and random 2. Figure 4.9 shows the benchmarks of Radix 95¹ through Radix 95³. These benchmarks reconfirm the information provided in Figure 4.6.

TEST FILE	Original Size	<u>Radix 95¹</u>		<u>Radix 95²</u>		<u>Radix 95³</u>	
		Result† Size	Overhead %	Result† Size	Overhead %	Result† Size	Overhead %
objcode 1	28,246	34,230	21.19%	34,004	20.00%	34,383	21.70%
objcode 2	310,480	372,332	19.92%	370,466	19.00%	376,143	21.15%
objcode 3	215,600	258,478	19.88%	257,250	19.31%	261,189	21.15%
source 1	168,276	204,196	21.35%	204,248	21.37%	204,219	21.36%
source 2	59,119	71,861	21.55%	71,740	21.35%	71,730	21.33%
random 1	40,000	49,343	23.36%	48,856	22.14%	48,912	22.28%
random 2	200,000	246,826	23.41%	244,250	22.13%	244,458	22.23%

objcode - Object Code generated by C compiler.

source - C source Code. Source 2 is used in the R95¹ Benchmark.

random - random binary digits. Random 1 and 2 are used in the R95¹ Benchmark.

† - Forced by Disk Storage Method. Based on Byte Boundaries.

Figure 4.9: Benchmark of Radix 95¹⁻³

CHAPTER V

CONCLUSION

Radix 95¹ substantially reduces encoding overhead over several other commonly available fixed-length encoding techniques. Radix 95ⁿ offers only slightly better efficiency than simple Radix 95¹. Since encoding overhead is just one of the factors that determines communication efficiency, the other factors must also be kept in mind. At some point the reduction in file size overhead will be offset by the amount of additional computation time required to produce higher powers of Radix 95ⁿ.

In this thesis, Radix 95ⁿ was researched to discover if additional savings in overhead could be created by encoding larger combinations of bits through collections of contiguous ASCII character sets. This investigation has shown that Radix 95ⁿ does not outperform Radix 95¹ to an extent great enough to justify the extra code and processing time required for encoding and decoding. Further reduction of processing time could be shown if the high-level software routines written were replaced by assembly level versions.

Additional reduction in overhead for all conversion programs can be gained by compressing the source data before the conversion process takes place. With the use of available compression programs, the size of the converted source file can be lower than its original size. For example, a 65K

binary file might typically be compressed at 40% to 39K, then encoded for transmission using Radix 95¹, which would increase the size to 46.8K. The transmittable ASCII version of the file would still be 18.2K (28%) smaller than its original form.

Current work on data conversion techniques is becoming less important with the introduction of higher bandwidth services yielding greater data transmission speeds for end users. Concurrent with the general trend toward higher bandwidth services is the rapid increase in low-bandwidth mobile data communications using radio channels. In these low-bandwidth applications, generated overhead is a serious problem when communicating to systems that do not handle 8-bit data properly. Although R95ⁿ did not drastically reduce overhead significantly, this work has shown the benefits and associated shortcomings of using grouped sets of characters for encoding text data. Radix 95 techniques will continue to offer noticeable reduction in transmission times for systems that have difficulty with 8-bit data transmission.

BIBLIOGRAPHY

- Abel, J. Alex, and Gerald Knezek. 1986. Binary to Text File Conversion Using RADIX 95: Department of Computer Science, North Texas State University. photocopied.
- Adler, Mark. 1991. SHIP: A Compression Program. 18 March. comp.compression, USENET.
- Brown, Eric, and Art Wilcox. 1984. Communications Features Explained. PC World, September , 170-177.
- Camp, David J., and Phil Howard. 1985. XXENCODE. UNIX Programmer's Manual. Berkeley, California: University of California, Computer Science Division, Department of Electrical Engineering and Computer Science.
- Chesson, Greg. 1988. Packet Driver Protocol. 5 May. comp.std.unix, vol. 14, no. 13, USENET.
- da Cruz, Frank, and Bill Catchings. 1984. Kermit: A File-Transfer Protocol for Universities. Part 1 : Design Considerations and Specifications. Byte, June, 225-278.
- _____. 1984. Kermit : A File-Transfer Protocol for Universities. Part 2: States and Transitions, Heuristic Rules, and Example. Byte, July, 143-145, 400-403.
- da Cruz, Frank. 1987. KERMIT: A File Transfer Protocol. Bedford MA: Digital Press.
- Fox, T. 1984. AX.25 Amateur Packet Radio Link-Layer Protocol Version 2.0. Newington, CT: American Radio Relay League.

- Frاند, Jason L. 1990. Sixth annual UCLA survey of school computer usage. Communications of the ACM, May, vol. 33 no. 5, 544.
- Gianone, C., F. da Cruz, and J.R. Douplik. 1989. MS-KERMIT User Guide Version 2.32/A. New York, NY: Columbia University.
- Jones, James G., and Gerald Knezek. 1988. Radix 95: Binary to Text Data Conversion for Packet Radio. In The Proceedings of the ARRL 7th Computer Networking Conference held in Columbia, Maryland, October 1982, edited by Paul Rinaldo. Newington, CT: American Radio Relay League.
- Jones, Greg, and D. Saengrussamee. 1989. R95 - Utility for Transferring 8-bit data across the Amateur Network. Texas Packet Radio Society Quarterly Report, February, Vol. 5 Issue 3.
- Jones, James G., Masato Hata, and Gerald Knezek. 1989. Packet Radio Prospects for Pacific Basin Data Communications. In The Proceedings of the Eleventh Annual Pacific Telecommunications Council Conference, in Honolulu, Hawaii, January 1989, edited by L.S. Harms and Dan J. Wedemeyer. Honolulu, HI: Pacific Telecommunications Council.
- Karn, Phil. 1985. TCP/IP: A Proposal for Amateur Radio Levels 3 and 4. In The Proceedings of the ARRL 4th Computer Networking Conference held in San Francisco, California, March, 1985, edited by Paul Rinaldo. Newington, CT: American Radio Relay League.
- _____. The KA9Q Internet (TCP/I) Package: A Progress Report, In The Proceedings of the ARRL 4th Computer Networking Conference held in Redondo Beach, California, August 1987, edited by Paul Rinaldo. Newington, CT: American Radio Relay League.

- Knezek, Gerald, and Greg Jones. 1989. ATS-3 : Packet Experiments. The Potential Impact of Packet Radio upon Pacific Basin Communications. In The Proceedings of the ARRL 7th Computer Networking Conference held in Colorado Springs, Colorado, October 1989, edited by Paul Rinaldo. Newington, CT: American Radio Relay League.
- Lempereur, Yves. 1985. BINHEX. Mainstay Software, Agoura Hills, California.
- Mefford, Michael J. 1989. Easy binary file transfer through ASCII text files. PC Magazine, 28 November, vol. 8 no. 20, 319.
- Miles, J.B. 1990. Like puppet, Kermit Software is friendly, easy to manipulate. Government Computer News, 30 April, vol. 9, no. 9, 29.
- Mukaida, Lori, Greg Jones, et al. 1989. Pacific Island Interactive Data Base Network Access : A PEACESAT, PICHTR, University of Hawaii Library Pilot Project. In The Proceedings of the Eleventh Annual Pacific Telecommunications Council Conference, in Honolulu, Hawaii, January 1989, edited by L.S. Harms and Dan J. Wedemeyer. Honolulu, HI: Pacific Telecommunications Council.
- Orost, Joe. 1991. BTOA Version 4.0. 26 March. comp.compression, USENET.
- Quarterman, John. 1990. The Matrix: Computer Network and Conferencing Systems Worldwide. Bedford MA: Digital Press.
- Racal-Vadic and Adaptive Computer Technologies. 1988. Comparative Benchmarks of high-performance data compress algorithms. EIA / Telecommunications Industry Association, TR-30.1 Committee, 13 December, TR-30.1/88-12072. photocopied.

Renka, Robert. 1987. Theoretical Overhead for Radix 95 Encoding in Binary Encoding Benchmarks: Radix 64 vs Radix 95 (Yu, Knezek, Carruth 1987) presented at the 11th Annual Computer Science Conference, Federation of North Texas Area Universities. Department of Computer Science, North Texas State University, Denton, Texas.

Renka, Robert. 1991. Radix 95 Encoding Notes to Greg Jones, March.

Stone, M. David. 1985. Picking the Proper Protocol. PC Magazine, 11 June, vol. 7, no. 4, 355-360.

Stout, Connie. TENET: Texas Education Network for Texas K-12 Educators, Electric Pages, May 1991.

Templeton, Brad. 1989. ABE Reference Manual. UUNET.UU.NET, Clairnet.

Camp, David J., and Phil Howard. 1985. XXENCODE. UNIX Programmer's Manual. Berkeley, California: University of California, Computer Science Division, Department of Electrical Engineering and Computer Science.

UNIX™ Programmer's Manual 7th Ed. 1980. Berkeley, California: University of California, Computer Science Division, Department of Electrical Engineering and Computer Science.

UUENCODE. 1990. UNIX Programmer's Manual. Berkeley, California: University of California, Computer Science Division, Department of Electrical Engineering and Computer Science.

Yu, Carol, Gerald Knezek, and Jeff Carruth. 1987. Binary Encoding Benchmarks: Radix 64 vs Radix 95 presented at the 11th Annual Computer Science Conference, Federation of North Texas Area Universities. Department of Computer Science, North Texas State University, Denton, Texas.