IBM TECHNICAL INFORMATION EXCHANGE

6507-0173
August 30, 1965
19 Pages

EL

PACKING ALPHABETIC INFORMATION INTO FOUR BITS

Robert E. Hanson
IBM Corp.
1049 Asylum Avenue
Hartford, Conn.

FOR IBM INTERNAL USE ONLY

This paper is in the author's original form. The objective in providing this copy is to keep you informed in your field of interest. Please do not distribute this paper to persons outside the Company.

Distributed by
DPD Program Information Department
IBM Corporation
112 East Post Road
White Plains, New York

6507-0173

ABSTRACT

This paper proposes a new technique for alphabetic data representation. A reduction in storage space of 40 to 45 percent should be easily achieved.

A general program flow chart for encoding and decoding the data is also provided. A case is presented for hardware implementation of this method.

TABLE OF CONTENTS

ABSTRACT

I.	Introduction	1
II.	The Method	2
III.	Implementation via Software	5
IV.	Implementation via Hardware	6
V.	Application Areas Impacted	9
VI.	Conclusions and Recommendations	10
VII.	Illustrations (figures 1 - 5)	11 - 16
VIII.	Cited References	18

I. Introduction

The IBM System/360 introduced the eight bit byte as the standard unit of data representation. The reasons for its selection are noted elsewhere. This media is quite satisfactory for numeric data as two digits may be represented in a single byte. Each four bit half byte, which has a capacity of sixteen states, carries one decimal digit thereby wasting only 3/8 (16-10/16) of the theoretical information power of the byte.

Alphameric data for display purposes is another matter. The current line printers print about fifty graphics comprised of the alphabet, numbers, special characters, and blank. Therefore, this type of data could be contained in six bits, which will hold 64 different characters. This means that the byte is only 1/4 (64/256) utilized, an unsatisfactory factor.

This paper presents a method that permits the use of hexidecimal (4 bit) coding to represent fifty-five graphics. Theoretically, reductions of 50 per cent are possible, however, in practice 40 to 45 per cent are to be expected over one character per byte storage. Clearly there are profound implications on storage requirements on tape and direct access volumes, and on data transfer rates.

The method is unique in that by using four bit coding the stream processing instructions of System/360 are available to assist encoding and decoding the data. Further, the user is able to optimize the method for his own data sets.

II. The Method

To understand this method, one must not think of data as static, residing in storage or on a printed page, but rather as a dynamic stream flowing past an observation point. Using hexidecimal coding, we have only 16 possible codes to represent all the desired graphics. Therefore, it is clear that we must re-use some of these 16 symbols several times to achieve full data representation. So, we will reserve one character, the hexidecimal F, to designate that a new coding structure will be used for all the following characters until another hexidecimal F is encountered. We will call this hexidecimal F a shift code.

Shift codes do not signify what the new coding structure is to be so that this information must be taken from the hexidecimal character following the shift code. We will call this character the table designator code, as it designates the new coding table. This means that if, in the stream of data, we find a character not available in the current set of 15 (the 16 possible four bit codes minus the shift code), we must insert a shift code and table designator to put us in a set which does contain the desired character.

As our objective is to increase data packing efficiency and since shift and table designators do not carry useful information, we must try to reduce their frequency or find a way to make them carry significant information to increase packing efficiency.

To this end, it is well to study the usage of the characters of the alphabet in the construction of ordinary English language text. Figure 1 illustrates these relative usages. We see that the first thirteen characters of this table account for 85 percent of all usage. Therefore, we will call these the prime text characters, and the remaining 13, the residue characters.

If we combine the prime text characters with a blank and one of the 13 residue characters, we can create thirteen coding tables (using 15 codes each), that cover the entire alphabet in one table or another, each table containing the prime text characters. Further, we can make the hexidecimal code for the residue character contained in each table identical to the table designator. This will allow the table designator, as it appears in the data stream following a shift code, to also designate the actual graphic desired, thereby eliminating the loss of the table designator as a vehicle for information. Figure 2 is an illustration of this kind of table. Note that in this figure, 13 separate coding structures are indicated for the 15 available hexidecimal codes. The coding tables are arrayed vertically to correspond to the hexidecimal code indicated on the left axis.

II. The Method (continued)

If we are receiving a stream of alphabetic information coded in hexidecimal characters from some source and wish to decode it, we must know which table to use first. So, we will establish the following conventions:

a. The first character is always encoded or decoded from table zero.

b. If the first character is a residue character not contained in table zero, there will be a shift code followed by a table designator to shift us to a table containing the desired residue character.

To illustrate decoding, let us take the hexidecimal stream D3BB3F465F5. If we begin at table zero, we find that D decodes to F, 3 to alphabetic O, B to L, B to L, 3 to W. We then encounter a shift code which says that the next hexidecimal character is a table designator for a new coding structure, as well as the hexidecimal code for the next alphabetic character. Therefore, we shift to table 4 where 4 decodes to W, 6 to I, 5 to N. Then, finding another shift code, we shift to table 5 where 5 decodes to G, giving a decoded message of FOLLOWING. Note that this message would require nine bytes to store in EBCDIC but requires only 11 half bytes to store using this method, a 39 percent reduction.

Let us encode the message JOE. The J is not to be found in table zero, so we insert a shift code followed by the table designator A, which is the table containing J. The O and E are also on table A, so that the hexidecimal representation for messages is FA20.

We have now created a method capable of handling the alphabet and blanks, however, we still need to handle numerics and the special characters required for ordinary text or names and addresses. For this purpose, three additional tables may be provided using the three unused table designator codes. These table designators are special, in that they do not represent the next data character in the stream, so that it takes a whole byte to shift into one of these tables (shift code plus table designator). This is not a severe restriction, as numerics will usually occur as groups, and both numerics and special characters are relatively infrequent in text or names and addresses. Table D is used to contain the numbers and the three common special characters. Table E is composed of the prime text characters and a blank. This is provided to permit an exit from Table D or F where the first following text character

II. The Method (continued)

is not a residue character. That is, if the first character following a punctuation mark is a prime character or blank, there is no table designator code to permit a shift to a table containing the characters, so a shift is made to Table E (remember that table designators D, E and F do not convey information and are bypassed in text decoding).

F is used as a table designator, since it will not be confused with a shift code, because it is preceded by a shift code. Table F is used for additional special characters not found in Table D. The complete table structure is shown in Figure 3.

Let us try a sample data stream using numbers and special characters.

DATA: ST LOUIS, MO. 17542

Hexidecimal representation:

```
8 2 E B 3 0 6 8 F D A E F 1 3 F E C E 1 7 5 4 2
S T b L O U I S       , b   M O       . b 1 7 5 4 2
```

This message consists of 19 bytes of data and 24 hexidecimal half bytes (13 bytes) for an improvement of 37 percent. This is not bad if we consider that this is an exceptional case and that this data would probably be recorded as:

```
8 2 E B 3 0 6 8 E F 1 3 E F D 1 7 5 4 2
S T b L O U I S b   M O b       1 7 5 4 2
```

which has 17 bytes or 20 half bytes for an improvement of 41 percent.

The following text sample is probably fairly representative of the packing to be expected. Note the long streams that develop without shifting tables:

```
D 3 7 E 1 4 C 9 E 4 C C 1 8 8 E (F) 1 0 2 9 3 A (F D) A E
F O R b E A C H b A C C E S S b     M E T H O D        , b

(F) 7 3 B (F) 0 (F) 1 0 E A 6 8 (F) 2 3 8 6 1 6 [illegible] 5 E 6 8 E 8 2 0 C
    V O L     U      M E b D I S     P O S I T I              N b I S b S P E C
```

II. The Method (continued)

```
6 D 6 0 A E 4 8 E 0 6 1 9 0 7 E 3 D E 1 9 0 E D 3 B B 3(F)4 6 5(F)5
I F I E DbA S bE I T HE R bO F bT H E bF O L L O   WI N   G
```

This message is 82 bytes long in EBCDIC and 91 half bytes long in hexidecimal, a saving of 47 percent.

It would also be well to point out that a large volume of numeric data (particularly small fields) could be represented more efficiently with this method than with packed decimal, because the sign position would not be required for each byte.

This method provides a method of representing 54 graphics with a sixteen bit structure. This will handle most standard printer arrangements. However, some applications, such as text printing, may require both upper and lower case alphabet, and additional special characters. This could be provided if we were to let one of the special characters in table F tell us to shift to an entirely new set of coding tables, representing lower case alphabet and additional special characters (they could be the same as the original set if desired).

III. Implementation via Software

The stream of coded half bytes will not have a predictable length due to the variable packing percentage inherent in this system. Therefore, it is suggested that the first byte of each coded text be the length of the coded field in binary representation. This format will make it possible to use the length directly in instructions, or via an execute instruction with a general register.

To facilitate encoding and decoding data, the format of the coded bytes should be stored as follows. The first half of the coded message should be stored in the numeric portion of consecutive byte locations. The remaining half of the message could then be stored in the zone portion of the same bytes. The format is shown below with consecutive half bytes numbered starting at 1.

Bytes

Zone	Length	11	12	13	14	15	16	17	18	19	20
Numeric		1	2	3	4	5	6	7	8	9	10

= 20 in binary

Note that the length specified is the length of the stream of half bytes not including the length byte.

While these encode and decode routines have not been coded, a general flow chart for each is included as figures 4 and 5. These routines will make extensive use of the four bit data handling ability of System/360 to stack and unstack the coded stream with the move zones, move numeric and move with offset instructions. The stream may be searched for shift codes or characters not present in the current table with the translate and test instruction, and the actual translation performed by a translate instruction. Of course, the speeds of these routines are not determinate at this time, but the time necessary for translation might be more than compensated for by a reduction in I/O transfer time, particularly if inactive records were not decoded at all.

IV. Implementation via Hardware

The read only storage and microprogramming ability of the System/360 would suggest that instructions other than those in the current set could be implemented in the hardware. If this could be done in the case of an encode alpha and decode alpha instruction, the time necessary for the translation could be significantly reduced, as well as eliminating the subroutine storage requirement.

By using the suggested instruction formats, the user would supply the encode and decode tables for the instructions. This provides the user with the ability to analyze the frequency distribution of the characters of each data set, and thereby optimize the packing. Also, using separate tables could provide a measure of security for confidential data sets or in data transmission.

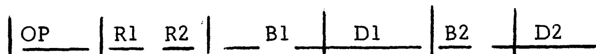
It would seem advantageous to change the stacking format for hardware implementation to an over and under arrangement as shown below:

Bytes

Zone	L	2	4	6	8	10	12	14	16	18	20
Numeric	G H	1	3	5	7	9	11	13	15	17	19

Suggested formats for these instructions are shown below:

Encode Alpha Instruction



SS Format

Operand 1 is the field to receive the packed information (length byte position).

Operand 2 is the field to be packed (length byte position).

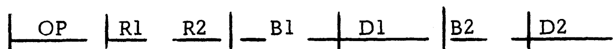
R1 is the register containing the address of the user supplied 256 byte encoding table.

Length of the data to be coded is taken from the first byte of operand 2. Output length is developed and inserted in the first byte of operand 1.

R2 not used.

The area to receive the packed data should be as long as the area containing the source data, as it is possible that no packing could take place.

Decode Alpha Instruction



Operand 1 is the area to receive the unpacked information (length byte position).

Operand 2 is the data to be unpacked (length byte position).

R1, the general register which contains the address of the user supplied 256 byte decode table, may be the same as the encode table.

Length of the coded data is supplied by the first byte of Operand 2. The length of the decoded alphameric information is placed in the first byte of Operand 1.

R2, not used.

The area to receive the unpacked data should be twice as long as the coded data area, as it is theoretically possible for this much expansion to take place.

Timing

If these instructions could be made to operate 1/2 as fast as a translate and test instruction (102 + 16N microseconds on a Model 30, where N is the number of bytes processed) then using 30KB tapes, we would have only to save four bytes of information to break even, and we would gain 28.4 microseconds for each additional byte saved.

$$44.4N = 102 + 16N$$
$$28.4N = 102$$
$$N \approx 4 \text{ bytes to break even}$$

Of course, each device and CPU must be evaluated for its impact on thruput speed under this assumption. We must also remember that in many jobs, not every record is active and need not be decoded, providing a pure thruput bonus. Also, records that are decoded but not altered need not be encoded to be returned to the data set.

V. Application Areas Impacted

Below are suggested some of the more obvious application areas that could be impacted by reduction of 40 to 45% in alpha storage requirement. The readers will probably think of many others.

1. Text Storage
   a. Administrative Terminal Systems
   b. Computer Aided Instruction

2. Name and Address Files

3. The Insurance Alpha Index (MIB) File

4. Historical Data Storage

5. Information Retrieval Systems

6. Data Transmission (to increase effective line speed)

VI. Conclusions and Recommendations

IBM should seriously consider implementing an alphabetic encode and decode instruction on the System/360. This could give us a strong competitive edge in thruput and direct access storage capacity. It could also open some new application areas that are currently marginal because of the cost of storage of alphabetic text.

Usage of the Alphabet in English Language Text

	Usage per 1000	Cumulative Usage per 1000
E	131	
T	90	221
O	82	303
A	78	381
N	73	454
I	68	522
R	67	589
S	65	654
H	59	713
D	44	757
L	36	793
C	29	822
F	28	850
U	28	878
M	26	904
P	22	926
Y	15	941
W	15	956
G	14	970
B	13	983
V	10	993
K	04	997
X	01	998
J	01	999
Q	01	1000
Z	01	1001

Source: See Reference I

Figure 1

TABLE DESIGNATOR CODE

HEXIDECIMAL CODE

	0	1	2	3	4	5	6	7	8	9	A	B	C
0	U	E	E	E	E	E	E	E	E	E	E	E	E
1	E	M	T	T	T	T	T	T	T	T	T	T	T
2	T	T	P	O	O	O	O	O	O	O	O	O	O
3	O	O	O	Y	A	A	A	A	A	A	A	A	A
4	A	A	A	A	W	N	N	N	N	N	N	N	N
5	N	N	N	N	N	G	I	I	I	I	I	I	I
6	I	I	I	I	I	I	B	R	R	R	R	R	R
7	R	R	R	R	R	R	R	V	S	S	S	S	S
8	S	S	S	S	S	S	S	S	K	H	H	H	H
9	H	H	H	H	H	H	H	H	H	X	D	D	D
A	D	D	D	D	D	D	D	D	D	D	J	L	L
B	L	L	L	L	L	L	L	L	L	L	L	Q	C
C	C	C	C	C	C	C	C	C	C	C	C	C	Z
D	F	F	F	F	F	F	F	F	F	F	F	F	F
E	b	b	b	b	b	b	b	b	b	b	b	b	b

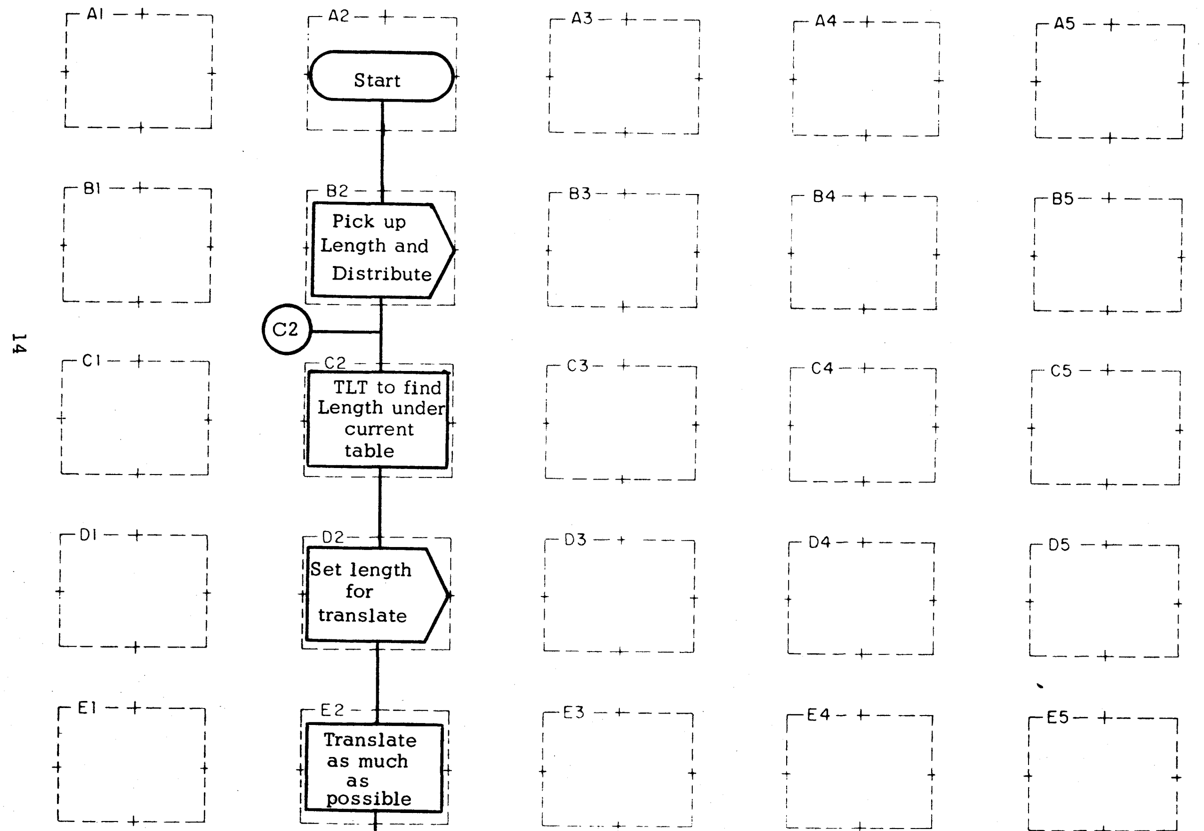
Figure 2

TABLE DESIGNATOR

HEXIDECIMAL CODE (rows) / TABLE DESIGNATOR (columns); column F and the right-hand entries: OTHER SPECIAL CHARACTERS

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	U	E	E	E	E	E	E	E	E	E	E	E	E	0	E	
1	E	M	T	T	T	T	T	T	T	T	T	T	T	1	T	
2	T	T	P	O	O	O	O	O	O	O	O	O	O	2	O	
3	O	O	O	Y	A	A	A	A	A	A	A	A	A	3	A	
4	A	A	A	A	W	N	N	N	N	N	N	N	N	4	N	
5	N	N	N	N	N	G	I	I	I	I	I	I	I	5	I	
6	I	I	I	I	I	I	B	R	R	R	R	R	R	6	R	
7	R	R	R	R	R	R	R	V	S	S	S	S	S	7	S	
8	S	S	S	S	S	S	S	S	K	H	H	H	H	8	H	
9	H	H	H	H	H	H	H	H	H	X	D	D	D	9	D	
A	D	D	D	D	D	D	D	D	D	D	J	L	L	,	L	
B	L	L	L	L	L	L	L	L	L	L	L	Q	C	-	C	
C	C	C	C	C	C	C	C	C	C	C	C	C	Z	.	F	
D	F	F	F	F	F	F	F	F	F	F	F	F	F	sp	b	
E	b	b	b	b	b	b	b	b	b	b	b	b	b	b	sp	
F	NOT USED															sp

Figure 3



Fold under at dotted line.

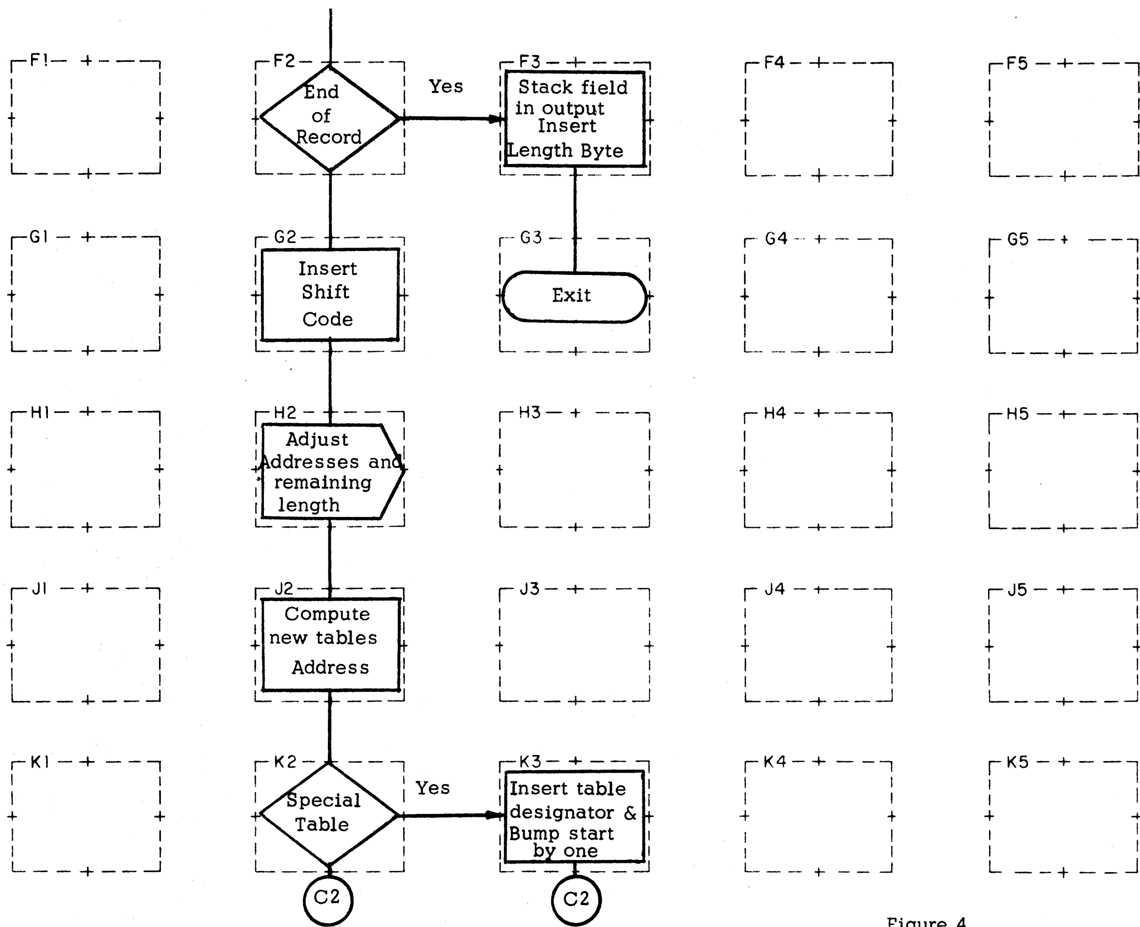


Figure 4

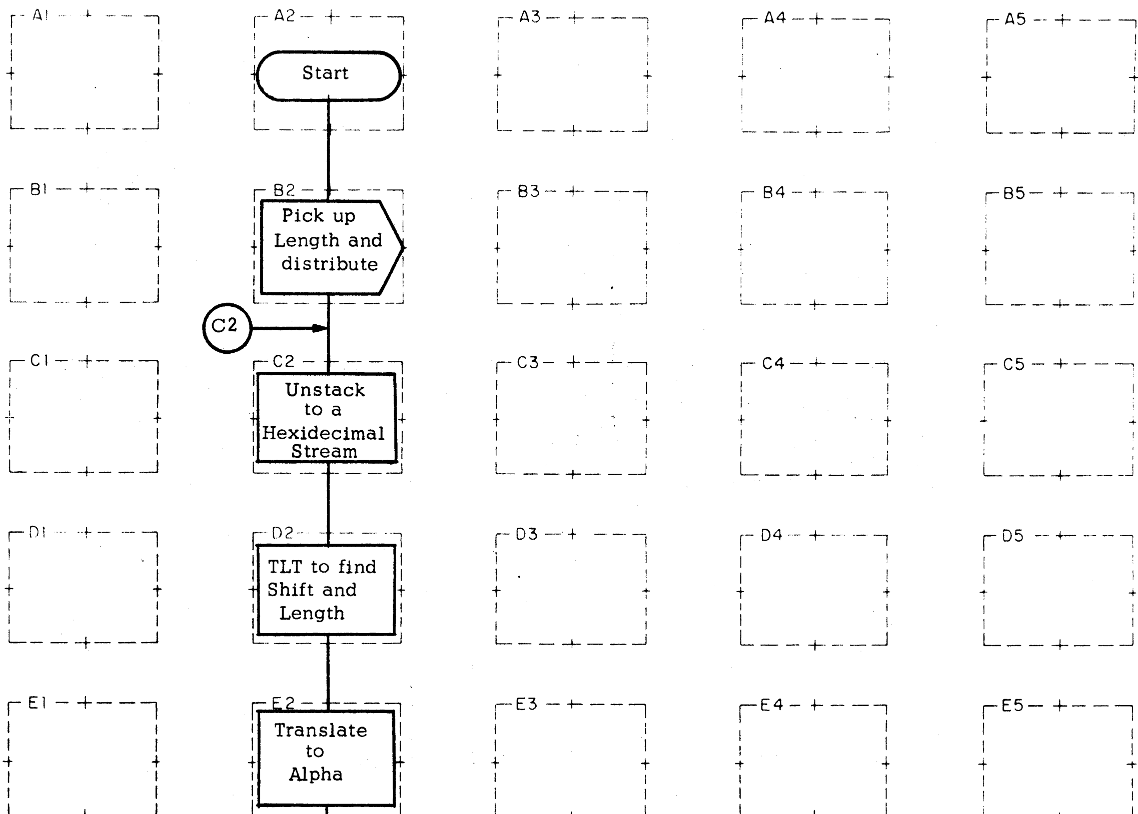
15

IBM Flowcharting Worksheet

PRINTED IN U.S.A.

Programmer: Program No.: Date: Page:

Chart ID: Chart Name: Decode Routine Program Name:



16

Fold under at dotted line.

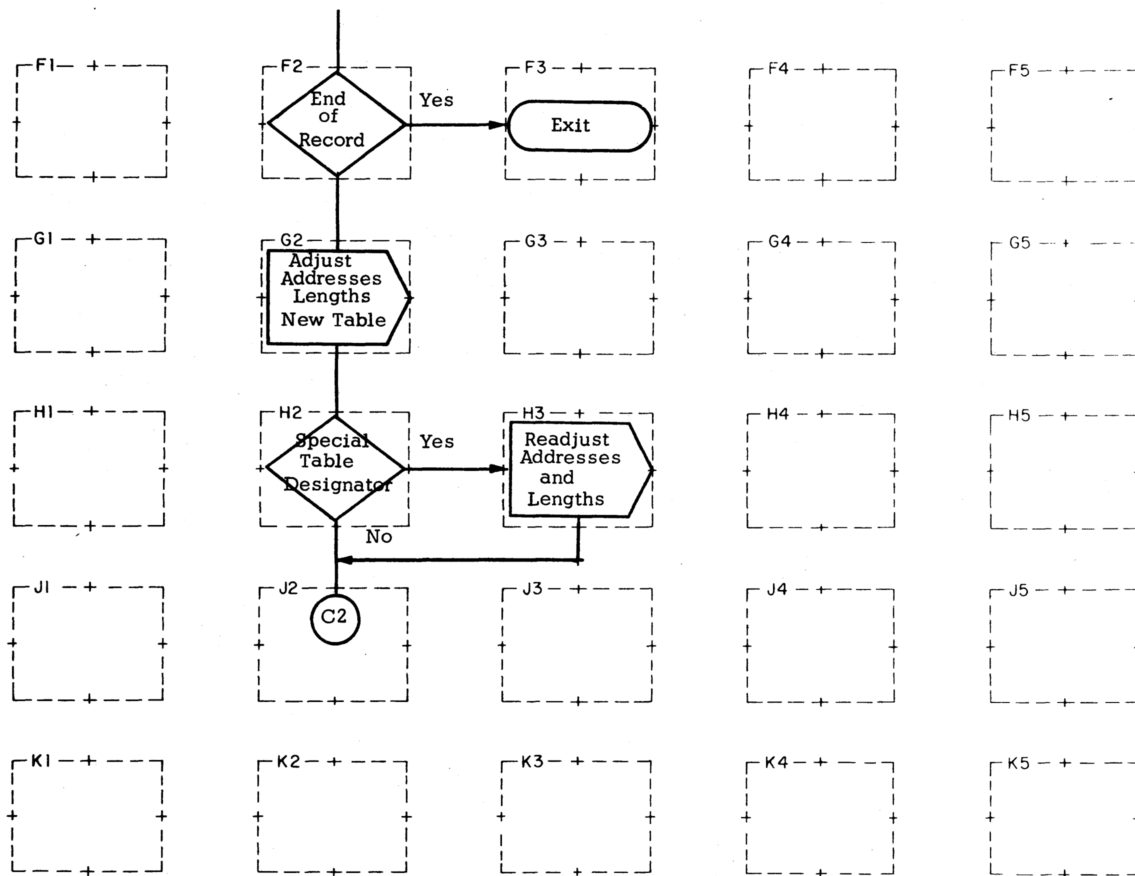


Figure 5

VIII. Cited References

1. W. W. Rouse Ball, Mathematical Reactions and Essays, The MacMillan Co., New York, Eleventh Edition, 1962, Library of Congress No. 39-2762.

Last Page