fixed point scaling *manual*

prepared by **BUTLER MFG. CO.**

USERS' PROJECT NO. 293

TITLE:	Fixed Point Manual Scaling in G-15D
TYPE:	General
CATEGORY:	Class 1
EQUIPMENT AFFECTED:	G-15D
SUBROUTINES USED:	Refer to the contents of this project.

PREPARED BY: Butler Manufacturing Company
7400 East 13th Street
Kansas City 26, Missouri

DATE: 22 February 1959

PREFACE

This manual is hereby presented to the Bendix G-15D Users Exchange Organization. It may be used or published as the organization wishes.

The concept of fixed point scaling which is presented herein is a classic method, and is well suited to scientific use. There are other methods of handling fixed point operation, and this same method can be defined another way. This particular method is favored by the author as being straightforward and as requiring a minimum of study.

This manual covers the important features of fixed point storage, arithmetic, input and output. The coverage of some of the features is rather brief. However, enough information is provided to encourage thought and experimentation on the computer. With that understanding, then, this manual is a study outline for those who wish to program in fixed point.

The author wishes to thank Mr. Raymond Walls of Bendix Computer for instruction on the theory and practice of input conversion routines. For criticism of the text and helpful suggestions on the arrangement of descriptive matter, the author is indebted to Messrs. Laurence English and Donald Johnson of Butler Manufacturing Company and Messrs. Keith Blann and Harry Lorch of Bendix Computer.

Ralph T Smith

Ralph T. Smith
Computer Project Engineer
Butler Manufacturing Company

RTS MM
4-20-59

OUTLINE OF MANUAL FIXED POINT SCALING IN G-15D

PAGE

I Introduction

A. Standard Commands and Fixed Point Binary Scaling

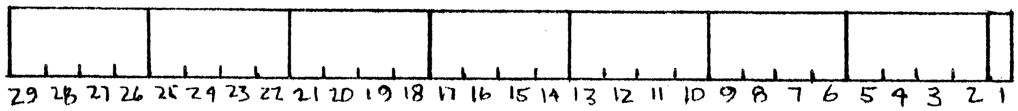
The Bendix Computer has an impressive standard command list, which, coupled with internal programming, makes it an extremely versatile and relatively fast machine. Standard commands directly control computer circuitry and go hand in hand with fixed point scaling. Most problems of an engineering or scientific nature are well suited to fixed point operation, since the range of each variable seldom requires the use of floating point arithmetic subroutines.

Fixed point binary scaling seems to be little understood or practiced. The author wishes to aid those who wish to use it in combination with standard commands by publishing this manual on the various phases of fixed point scaling.

II Basic Fixed Point Binary Scaling

A. Fixed Point Numbers in Storage

Fixed point binary scaling deals with numbers converted from decimal form to equivalent binary form, and with the method of storing such numbers in a cell of the computer's memory. The storage cell in the Bendix G-15D contains 29 bit positions, one bit for sign and 28 bits for numerical information. The storage cell may also be used for specially coded information, and it may be combined with another cell for double precision computation. For purposes of this manual, however, single precision number storage only will be discussed. The storage cell may be diagrammed thusly:

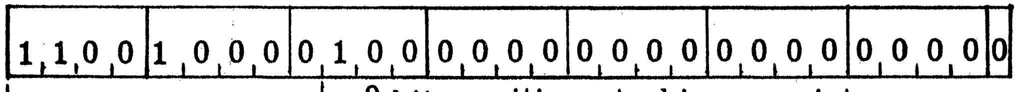


The bit positions are numbered as shown. Bit 1 is the sign and bit 29 is the most significant bit of the number being stored. The heavy vertical lines may be used as a convenience for planning purposes and are also shown because the input-output circuitry deals with the bits in these groups of four when operating under a standard format.

The concept of fixed point binary scaling which is presented here will be termed the Point Location or PL method. As illustrated in the storage cell above, 28 bit positions are available for data storage. Numbers may be stored in the cell with the binary point positioned to suit the size, or range, of each variable.

To illustrate this in everyday terms, a person using a desk claculator may position the decimal point on the keyboard to suit the numbers he will be handling. If large numbers are to be handled, the pointer or marker for the decimal point will have to be set far enough to the right to accomodate them, and vice versa for small numbers.

Suppose a number requires 9 bit positions above the binary point. It can be placed in the storage cell with the binary point nine bit positions from the left of the storage cell. An example of such a number is 400. 5. Its binary equivalent is 110010000. 1 and it would be placed in the storage cell thusly:



9 bit positions to binary point

This number is said to be scaled 2^{-9}. The number of bit positions from the left of the storage cell to the binary point is the negative power of two by which the number is scaled. A number whose binary point lies 14 positions from the left is scaled 2^{-14} etc.

Sometimes scaling is indicated like this: $400.5 \cdot 2^{-9}$ or $N \cdot 2^{-9}$. As defined by the PL method, the number in storage is not multiplied by 2^{-9}. It means the binary point is located 9 bit positions from the left of the storage cell.

It is also common terminology to say the stored number has a "scale factor of 2^{-9}". This is used in the following examples in Part II, arithmetic operations, and it should be understood that the scale factor referred to is a storage, or point location scale factor. This storage scale factor should not be confused with input-output scale factors discussed in Part III.

B. Multiplication of Fixed Point Numbers

The author presents the following information with the assumption that the reader is familiar with the PN, ID, and MQ registers, their associated 1 bit sign register, the IP flip-flop, and with the operation of the multiplication process. This information can be obtained from a Bendix schooled programmer if the reader so desires.

Before multiplication, the ID register holds the multiplicand and the MQ register holds the multiplier. After multiplication, the PN register holds the product. The multiplication process is a high-order process, meaning it takes place from the most to the least significant bit of the multiplier. For this reason, the number of bit positions above the binary point in the multiplicand must be added to those in the multiplier to obtain the number of bit positions above the binary point in the product.

In other words, the scale factor of the multiplicand must be multiplied by the scale factor of the multiplier to obtain the scale factor of the product. Suppose the multiplicand is scaled 2^{-14} and the multiplier is scaled 2^{-4}. The product will be scaled 2^{-18}. The following example illustrates this.

Multiplicand = $10 \cdot 2^{-14}$:

0,0,0,0	0,0,0,0	0,0,1,0	1,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	

14 bit positions

Multiplier = $5 \cdot 2^{-4}$:

0,1,0,1	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	

4 bit positions

Product = $50 \cdot 2^{-18}$:

0,0,0,0	0,0,0,0	0,0,0,0	1,1,0,0	1,0,0,0	0,0,0,0	0,0,0,0	

18 bit positions

C. Division of Fixed Point Numbers

The author presents the following information with the assumption that the reader is familiar with the two word registers, and with the operation of the division process.

Before division, the ID register holds the denominator and the PN register holds the numerator. After division, the MQ register holds the quotient. To obtain the number of bit positions above the binary point in the quotient, subtract the number of bit positions above the binary point in the denominator from the number of bit positions above the binary point in the numerator.

This is the same as dividing the scale factor of the numerator by the scale factor of the denominator. Suppose the numerator is scaled 2^{-14} and the denominator is scaled 2^{-8}; the quotient will be scaled 2^{-6}.

The following example illustrates this:

Numerator = $144 \cdot 2^{-14}$:

0,0,0,0	0,0,1,0	0,1,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	

14 bit positions

Denominator = $12 \cdot 2^{-8}$:

0,0,0,0	1,1,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	

8 bit positions

Quotient = $12 \cdot 2^{-6}$:

0,0,1,1	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	

6 bit positions

The division process differs from the multiplication process in two important features:

1. The scaling of the answer can be changed if the programmer wishes to do so.

2. Overflow is possible if, disregarding scaling, the contents of PN are equal to or greater than the contents of ID. Also, in cases of changing the scaling of the answer, erroneous answers are possible if, disregarding scaling, the contents of PN are equal to or greater than twice the contents of ID.

The scaling of an answer can be changed by either increasing or decreasing the operation time of the division command. For each two word times it is increased, the number of bit positions above the binary point in the quotient is decreased by 1. For each two word times it is decreased, the number of bit positions above the binary point in the quotient is increased by 1.

If, in the previous example, the division time is increased by two word times the scaling of the quotient becomes 2^{-5}.

5 bit positions

0, 1, 1, 0	0, 0, 0, 0	0, 0, 0, 0	0, 0, 0, 0	0, 0, 0, 0	0, 0, 0, 0	0, 0, 0, 0	

If the division time is decreased by two word times the scaling of the quotient becomes 2^{-7}.

7 bit positions

0, 0, 0, 1	1, 0, 0, 0	0, 0, 0, 0	0, 0, 0, 0	0, 0, 0, 0	0, 0, 0, 0	0, 0, 0, 0	

D. Addition - Subtraction of Fixed Point Numbers

The author presents the following information with the assumption that the reader is familiar with the AR, which is a 1 word adder.

The AR can be used only for addition and subtraction. As explained previously, the two word registers can be used for multiplication and division. It should be mentioned, however, that the PN register is the 2 word (double precision) accumulator. This explanation will be confined to the AR for simplicity, however.

In the addition or subtraction of fixed point numbers in the AR, the binary points must be aligned. This means that numbers to be added to or subtracted from each other must have the same scaling, i. e., the number of bit position above the binary point must be the same. If they are not, one of the numbers may be aligned with the other by left shifting it in MQ, or right shifting it in ID, whichever is required. Since loss of the most significant bits occurs during left shifting in MQ, it follows that some of these bits may be non-zero, causing loss of part of the number being shifted. There is no indication these bits are lost. The author prefers to left-shift in AR or PN by executing a block command to add the register to itself for the required number of word times to obtain the shift. The number must be in form of absolute value and sign, of course, and a 2 or 6 transfer characteristic should be used.

This process produces overflow if significant bits are lost, thereby providing the programmer with evidence of error.

Suppose two numbers scaled 2^{-6} are added:

Augend = $10 \cdot 2^{-6}$ = 001010.

0,0,1,0	1,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	

Addend = $16 \cdot 2^{-6}$ = 010000.

0,1,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	

Sum = $26 \cdot 2^{-6}$ = 011010.

0,1,1,0	1,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	

Suppose a number scaled 2^{-6} is to be added to a number scaled 2^{-8}.

Augend = $10 \cdot 2^{-6}$ = 001010.

0,0,1,0	1,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	

Addend = $16 \cdot 2^{-8}$ = 00010000.

0,0,0,1	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	

Shift Augend Right 2 Bit Positions

0,0,0,0	1,0,1,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	

Sum = $26 \cdot 2^{-8}$ = 00011010.

0,0,0,1	1,0,1,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	0,0,0,0	

III INPUT - OUTPUT SCALING

A. Output

1. Conversion from Binary to Decimal

General - Two steps are involved in output conversion. The first is applying a scale factor and the second is conversion of the resulting binary fraction from true binary to binary coded decimal.

a) Scale Factor Selection

Output scale factors must eliminate all powers of two by which a number is scaled and divide by enough powers of ten to keep the number to be converted less than 1. This may be accomplished by division by a scale factor containing negative powers of two and positive powers of ten or by multiplication by a scale factor containing positive powers of two and negative powers of ten.

Suppose a number is scaled 2^{-8}. Its value must be less than 2^{+8}, or 255.9999 maximum. According to the rule above, the number must be divided by enough powers of ten to keep the number to be converted less than 1. In this case, 10^3 must be used. Let $N \cdot 2^{-8}$ = the number to be converted from binary to decimal. If a division scale factor is used, it would have to be $2^{-8} \times 10^3$.

Proof: $\dfrac{N \cdot 2^{-8}}{2^{-8} \times 10^3} = N \cdot 10^{-3} = .2559999$ Maximum.

If a multiplication scale factor is used, it would have to be $2^8 \times 10^{-3}$.

Proof: $N \cdot 2^{-8} \; (2^8 \times 10^{-3}) = N \cdot 10^{-3} = .2559999$ Maximum.

Only one of the scale factors can be used, however. The division scale factor could never be contained in storage because 10^3 converted to binary contains 10 significant bits and cannot be scaled 2^{-8}.

Two significant bits would be left outside of the storage cell, and to change its scaling to 2^{-10} would make it unusable as a scale factor for a number scaled 2^{-8}.

Therefore, the multiplication scale factor must be used. From G-15 Card 10^{-3} = .004189375 hex. To make up $2^8 \cdot 10^{-3}$ scale factor, lay .004189375 out in binary.

```
  0    0    4    1    8    9    3    7    5
.0000 0000 0100 0001 1000 1001 0011 0111 0101
          .4    1    8    9    3    7    5
```

Move binary point to right 8 bit positions (multiplies by 2^8) and regroup bits in groups of four. Answer = $10^{-3} \cdot 2^8$ = .4189375 hex.

The simple way to determine which scale factor is usable is to evaluate them as shown below. The scale factor which is less than one must be used. In the above case:

$$2^{-8} \times 10^{3} = \frac{1000}{256} > 1 \text{ ; No Good}$$

$$2^{8} \times 10^{-3} = \frac{256}{1000} < 1 \text{ ; OK}$$

Here is another example. Find the scale factor for $N \cdot 2^{-15}$.

$N < 2^{+15} = 32,768$; Power of 10 is the fifth power $= 10^5$

Division S. F. $= \frac{10^5}{2^{15}}$ $\frac{100,000}{32,768} > 1$; No Good.

Mult. S. F. $= 2^{15} \times 10^{-5}$

$= \frac{32,768}{100,000} < 1$; O. K.

From G-15 card: 10^{-5} = .0000u7w5uw4 hex

```
  0    0    0    0    u    7    w    5    u    w    4
0000 0000 0000 0000 1010 0111 1100 0101 1010 1100 0100
                  . 5    3    Y    2    X    6    2
```

The scale factor is .53Y2X62 hex and must be multiplied by the number to be converted from binary to decimal coded binary.

If this process is used to determine the scale factor for the entire range of scalings from 2^{-1} to 2^{-28}, it will be found that the output scale factor always has to be multiplied by the number to be converted. Let us consider when an output scale factor could be divided into a number to be converted, and why this might be desirable in special cases. Suppose a number is scaled 2^{-7}. Its value must be less than 128. But suppose this number was always less than 100. This means its decimal form contains only two significant digits, and to use the multiply scale factor ($2^7 \times 10^{-3}$) would always leave a leading zero after conversion: 099.9999. In this special case, a scale factor of $2^{-7} \times 10^2$ could be divided into the number to be converted, making the number appear as 99.99999 after conversion. The advantage is elimination of the leading zero in front, and an additional digit of accuracy will be converted below the decimal point. The scale factor for $2^{-7} \times 10^2$ is .w800000 hex.

10^2 = 64 hex

```
  6    4
 0110 0100.
.w     8    0 0 0 0 0
```

Note that in this case the binary point was moved to the left to make up the scale factor. This is because the scale factor, $2^{-7} \times 10^2$, contains a negative power of two and is a division scale factor. In any case, the binary point must be somewhere to the left of the most significant bit of the scale factor. If any one bits lie to the left of the point, the scale factor is not usable in a standardized system.

b) Use of the Special Extract Command for Conversion

A method of converting binary fractions to decimal coded binary will be explained first. The method described applies to conversion of binary fractions to any base other than binary, but conversion to the base ten only will be discussed here.

The first step is to multiply the binary fraction by ten in binary. This will produce a whole number and a fraction, both in binary. The whole number is the first decimal digit of the fraction, and is in decimal coded binary. The remaining fractional part is again multiplied by ten in binary. The whole number produced this time is the second decimal digit of the fraction being converted. This process can be continued as many times as necessary to completely convert the binary fraction to decimal coded binary, or the process can be discontinued after a satisfactory accuracy is obtained.

Take .1111 binary as an example. The decimal value of this fraction is Z x 16^{-1} = $\frac{15}{16}$ = .9375.

STEP 1:

```
       .1111
        1010
       11110
     11110
    1001.0110
       9  =  first decimal digit
```

STEP 2: Remaining fractional part

```
=      .0110
     x 1010.
        1100
      1100
    0011.1100
       3  =  next decimal digit
```

STEP 3: Remaining fractional part

```
=      .1100
     x 1010
       11000
     11000
    0111.1000
       7  =  next decimal digit
```

STEP 4: Remaining fractional part

```
=      .1000
        1010
       10000
     10000
    0101.0000
       5  =  next digit
```

The remaining fractional part is zero, therefore further conversion is unnecessary. Collecting the decimal coded whole numbers in the order they were produced, the answer is .9375 which is the correct value. If the binary coding of these decimal digits were typed from AR, or line 19, the type-out would actually be .9375:

```
.1001  0011  0111  0101
   9     3     7     5
```

In presenting the following information, the author assumes the reader is familiar with the operation of the special extract command 3.23.31. The special extract command can be used to good advantage making this conversion, using the two word registers. The following extractors are necessary for single precision conversion.

- 0 Z Z Z Z Z Z + Z Z Z Z Z Z Z D_2
- 0 0 Z Z Z Z Z + Z Z Z Z Z Z Z D_3
- 0 0 0 Z Z Z Z + Z Z Z Z Z Z Z D_4
- 0 0 0 0 Z Z Z + Z Z Z Z Z Z Z D_5
- 0 0 0 0 0 Z Z + Z Z Z Z Z Z Z D_6
- 0 0 0 0 0 0 Z + Z Z Z Z Z Z Z D_7

The number after conversion will have 7 digits, thus: $D_1D_2D_3D_4D_5D_6D_7$. The first digit, D_1 can be formed quite simply by placing the binary fraction in ID_1 and 1010 in MQ1. After multiplying for eight word times, D_1 will be formed in the four most significant bits of PN_1, and the remainder of PN_1 holds the remaining binary fraction. If the special extract command is given using the D_2 extractor, D_1 remains in PN_1, while that portion of ID_1 is cleared to zero, and the binary fraction below D_1 is copied into ID_1 while that portion of PN1 is cleared to zero.

By placing another ten (1010) in MQ_1 and multiplying for eight more word times, D_2 is formed below D_1 in PN_1 and the remaining binary fraction can be extracted into ID_1 and cleared out of PN_1 using the D_3 extractor. This process continues in the order Multiply - Extract through use of the D_7 extractor after which one more multiplication forms D_7, and the binary coded decimal equivalent of the original binary fraction is now in PN_1.

Two improvements can be made on the process as explained here. The first is to use .V6XV680 in MQ_1 instead of 1010. This multiplier consists of seven 101 groups thus:

```
101 101 101 101 101 101 101
 V    6    X    V    6    8    0
```

By doing this, it is unnecessary to place 1010 in MQ_1 before each multiplication, and each multiply command lasts for six word times. The effect is the same as multiplication by 1010. The second improvement would be to carry out the conversion in PN_0, reducing the extractors to the following:

$$
\begin{array}{llllllll}
0 & Z & Z & Z & Z & Z & Z & D_2 \\
0 & 0 & Z & Z & Z & Z & Z & D_3 \\
0 & 0 & 0 & Z & Z & Z & Z & D_4 \\
0 & 0 & 0 & 0 & Z & Z & Z & D_5 \\
0 & 0 & 0 & 0 & 0 & Z & Z & D_6 \\
0 & 0 & 0 & 0 & 0 & 0 & Z & D_7
\end{array}
$$

in even word times only.

The previous example of .1111 = .9375 can again be given, using the PPR tracer on the Bendix output conversion subroutine. This appears on the next page.

```
pw00  s .1101000
sc5f020x0000002x03  ssc7fqz000000  81002zw  sima .z000000
rq/p0290  s

.90   .91.92.0.28.25   .z000000

.92   .93.95.0.02.24   .v6xv680

.95   .06.u2.0.24.31   .0000000  .9600000

.u2   .u5.u5.3.23.31

.u5   .06.10.0.24.31   .0000000  .93w0000

.10   .13.13.3.23.31

.13   .06.20.0.24.31   .0000000  .9378000

.20   .23.23.3.23.31

.23   .06.30.0.24.31   .0000000  .9375000

.30   .33.33.3.23.31

.33   .06.40.0.24.31   .0000000  .9375000

.40   .43.43.3.23.31

.43   .06.50.0.24.31   .0000000  .9375000

.50   .53.53.3.23.31

.53   .06.60.0.24.31   .0000000  .9375000

.60   .61.63.0.26.28   .9375000
```

These commands are on Page 7 of 9 of Input-Output Routine.

2. Output Format Selection

The following information is presented with the assumption that the reader is familiar with the functions of each of the format characters. If not, a detailed description of each may be found in the Bendix Coding Manual or in the newer Programmers Reference Books.

The format characters will be discussed here from the standpoint of type-out. Perhaps the best way to explain output format is to take a specific example.

Take $N \cdot 2^{-8}$ as an example. The output scale factor was $2^8 \times 10^{-3}$ and the converted number now is ready for type-out. It is presently divided by 10^3, so the period should be called for after the third digit is typed, which is where the decimal point actually belongs. The format would be

S D D D P D D D D T E

and, if the number converted was, say ± 189.9870, that is the way it would actually be typed out. The way to actually cause type-out of converted numbers is to place the compiled format in words 03 and 02 of line 03, put the number to be typed in AR, and give the command L2. N. 0. 08. 31. This will cause type-out of AR. The converted number can also be typed from line 19 by placing the compiled format in words 03 and 02 of line 02, clearing line 19, transferring the number to be typed into 19. U7, and executing the command L2. N. 0. 09. 31. If line 19 is not clear, the end code in the compiled format will be changed to a reload code, causing further type-out under the same format until line 19 has been emptied of all non-zero information.

The format can be compiled using the 03 x 08 code in Standard PPR or can be compiled by the individual. The procedure is simple.

Write the letter abbreviations first:

S D D D P D D D D T E

Then, using a table of format characters, write each three bit character in sequence. The G-15 reference card has a table on the back side.

```
  S   D    D    D   P  D     D    D    D    T E
 100 0,00 00, 0 000,011 0,00 00, 0 000,000 1,1,000 1
  8    0     0     6     0     0     1   -  1
```

The next step is to regroup the bits in groups of four until seven hex digits have been made up. Then allow one bit for sign, and continue with the same grouping (7 hex digits and sign) until the last format character has been taken care of. Fill out the remaining of the 7 hex digits and their sign as zero. The first seven hex digits are for storage in word 03 of line 03 or 02, and the next are for storage in word 02.

The format for line 19 may extend into words 01 and 00 of line 02, if required. This is very handy for typing up to four numbers at a time, and each 7 digit number could have a different format, if required.

One further difference between type-out of line 19 and AR should be mentioned. AR does not have to be empty before type-out will cease. If only two digits are desired, AR may be typed with DDE, and the five remaining digits in AR will not cause the format to reload, as would be the case for line 19.

If trailing digits are not desired, they can be omitted by replacing their digit format characters with wait format characters. Suppose the last two digits are not required in the previous example. The format could be changed to

S D D D P D D W W T E

The compiled format would become

```
  S      D   D     D P      D   D     W     W    T  E
 100 0, 00 00, 0 000, 011 0, 00 00, 0 111, 111 1, 1,0001,
  8     0      0        6     0       7     Z   -  1
```

and, using the previous example of ± 189. 9875, the type-out now would be ± 189. 98. No round-off occurs. However, by correlating the compiled format with a round-off factor, the number can be rounded off. This is discussed in the next section.

3. Round Off For Type-Out

It is often desirable to round off numerical data for type-out. The procedure is quite simple. The round off must be added to the number being converted after the scale factor has been applied to it and before use of the special extract command for conversion. The round off must be a 5 positioned after the last digit to be typed thus:

Number of digits to be typed	Round Off	Round Off converted to a binary fraction (hex)
.X	.05	.0wwwwwx
.XX	.005	.0147uy1
.XXX	.0005	.0020w49
.XXXX	.00005	.000346x
.XXXXX	.000005	.000053z
.XXXXXX	.0000005	.0000087
.XXXXXXX	.00000005	.000000x

Since the number being converted is a binary fraction scaled 2^0 at the time the round off is added, the round off must also be a binary fraction scaled 2^0.

Going back to the previous example of ± 189.9870, it was seen that if the last two digits were "waited", the type-out became ± 189.98. This could be rounded off to ± 189.99 by use of the proper round off factor, .000005 = .000346x. The round off would have to be added to the number before the conversion process, immediately after the scale factor was applied, when it was still in its binary fraction equivalent of .189 9870. The round off would make it the binary equivalent of .189 9920 and after conversion, it would appear in decimal coded binary as .189 9920. The format of

S D D D P D D W W T E

cuts off the last two digits, making the type-out ± 189.99.

4. Summary Of Output Procedure

The output procedure takes a scaled binary number, and converts it to decimal coded binary and types it out. The first step is to convert the number from whatever scaling it may have to a binary fraction. The reason for reducing it to a fraction is because the special extract command can be used to great advantage in converting binary fractions to decimal coded frations. The output format is selected, placing the period where the decimal point really belongs, and omitting trailing digits, if desired. In order to round off numbers being typed out, a round off can be added to the number after its reduction to a binary fraction by the scale factor. This round off should always be compatible with the output format. In other words, it should add the binary equivalent of a 5 in the position after the last digit typed, so that the last digit typed is rounded off.

The following example shows the Bendix output conversion subroutine converting and typing $N \cdot 2^{-8}$ where N = 189. 9870 = VX. ZWVW0 hex. The format compiled previously of SDDDPDDWWE is shown at the point marked (1) being transferred from ID to $02^{\underline{02.\ 03}}$. The scale factor of $2^8 \times 10^{-3}$ = . 4189375 hex is multiplied by the number being converted at the point marked (2), giving . 1899870 = . 30U2ZWZ hex. The round off of . 000005 = . 000053Z hex is added at the point marked (3), giving . 1899920 = . 30U350y hex. The combination of Multiply - Extract is used at the point marked (4) to convert D_1 through D_7. The rounded number is typed at the point marked (5), and the type-out is 189. 99, which is 189. 9870 rounded off to two digits below the decimal point.

For those who are interested in the typed information at the top of the page, the author did this while setting up the demonstration. The number 189. 987 is typed in to a fixed point input routine, and a standard command is typed into line 23 and executed, which brings the converted number into AR. It is typed from AR, using the a key. This is marked with the letter A. PPR is loaded next, and the Bendix output subroutine is read in to line 18, and the round off of . 00000005 = . 000000X hex in word 76 is replaced with . 000005 = . 000053Z hex. The 03 x 08 code of PPR is used to compile the SDDDPDDWWE format at point C. The 02 x 03 code transfers the subroutine, still in line 18, to line 02 from which it must be executed. This is marked D. The remainder of that line and the next line are type-in of format, scale factor, $N \cdot 2^{-8}$, and return command, and standard commands to place them in positions specified by the subroutine specifications. This is marked E. The scale factor was entered with a minus sign, which was a code to the output routine for multiplication by $N \cdot 2^{-8}$. The sign was a code only, and absolute value was used.

```
                    p101w0  s
101    189/987  s 189.9870                     } A
102    sc7fq810023w  sia .vxzwuw0               }
ppw00  s .1101000                                } B
sc5fv76  s                                       }
.76    .000000x  00000500  s .000053z           }
03x08  s                     } C
       .03                   }
40003007761                  }
-.800607z   .1000000  02x03  ssc7fq-800607z  1000000  0000000  -82002z9  siq  } E
                      (D)
-4189375  /8000275  siqvxzwuw0  81002z5  siq808021z  81002zw  sima .808021z    }
rq/p02u6  s
.u6    .02.05.4.25.02    .1000000 -.800607z    (1)
.05    .08.08.0.23.31
.08    .08.08.0.28.31
.09   u.10.15.0.26.19    .0000000
.15    .45.56.0.02.25    .yv0y393
.56    .65.34.0.25.02    .yv0y393
.34    .36.37.2.21.02    .808021z
.37    .39.44.2.28.25    .4189375               }
.44    .45.48.0.22.31                           } (2)
.49    .53.55.0.21.24    .vxzwuw0               }
.55    .56.04.0.24.31    .167x380   .30u2zwz    }
.04    .05.24.0.26.25    .30u2zwz
.24    .25.61.0.02.28    .w04139w
.61    .63.64.0.28.02    .w04139w
.64    .65.75.0.25.28    .30u2zwz  } (3)
.75    .76.84.0.02.29    .30u350y  }
.84    .90.90.0.23.31
.90    .91.92.0.28.25    .30u350y                  }
.92    .93.95.0.02.24    .v6xv680                  }
.95    .06.u2.0.24.31    .8000000 -.1y66128   D1   }
.u2    .u5.u5.3.23.31                              }
.u5    .06.10.0.24.31    .z000000  .18zzwv9   D2   }
.10    .13.13.3.23.31                              }
.13    .06.20.0.24.31    .x600000 -.189zxz3   D3   } (4)
.20    .23.23.3.23.31                              }
.23    .06.30.0.24.31    .y5w0000  .1899yv8   D4   }
.30    .33.33.3.23.31                              }
.33    .06.40.0.24.31    .8z98000  .1899933   D5   }
.40    .43.43.3.23.31                              }
.43    .06.50.0.24.31    .19vz000  .1899920   D6   }
.50    .53.53.3.23.31                              }
.53    .06.60.0.24.31    .1017600  .1899920   D7   }
.60    .61.63.0.26.28    .1899920
.63    .64.65.0.28.28    .1899920
.65    .u7.14.0.28.19    .1899920
.14    .26.36.0.09.31    189.99 —— (5)
.36   w.00.00.0.16.31
```

B. Input

1. Conversion From Decimal Coded Binary To Binary

a) Use of the special extract command for conversion

A method of converting decimal coded binary whole numbers to binary whole numbers will be discussed first. Take a 7 digit decimal coded binary whole number, $N = D_1\ D_2\ D_3\ D_4\ D_5\ D_6\ D_7$ as an example. To reduce it to a true binary whole number, simply multiply D_1 by ten in binary six times, D_2 for five times, D3 for four times, etc., and add the products. The effective multipliers are listed in the table below.

DIGIT	NUMBER OF TIMES MULTIPLIED BY TEN IN BINARY	EFFECTIVE MULTIPLIER
D_1	6	10^6
D_2	5	10^5
D_3	4	10^4
D_4	3	10^3
D_5	2	10^2
D_6	1	10^1
D_7	0	$10^0 = 1$

In the two word registers, this process can be accomplished very neatly by six successive Extract-Multiply operations, using these extractors:

E_1	=	z000000
E_2	=	zz00000
E_3	=	zzz0000
E_4	=	zzzz000
E_5	=	zzzzz00
E_6	=	zzzzzz0

and the same v6xv680 multiplier (7 u's) that is used on output conversion.

The process begins with N in PN_1 and the v6xv680 multiplier in MQ_1. The first Extract-Multiply operation works as follows. The extract command copies the first four bits of PN_1 into ID_1, leaving the remaining contents of PN_1 undisturbed and clearing the remainder of ID_1. These first four bits contain D_1 in decimal coded binary. The multiply command causes $D_1 \times 10$ in binary to be added into the first eight bits of PN_1. The first four of these bits were cleared when D_1 was extracted into ID_1. The next four bits contain D_2. Therefore, the first eight bits of PN_1 now contain $D_1 \times 10$ plus D_2.

The next Extract-Multiply operation works in like manner, taking the first eight bits of PN_1 into ID_1, which now contain $D_1 \times 10 + D_2$, and multiplying by 10, giving

$$(D_1 \times 10 + D_2)\ 10 \text{ plus } D_3$$
$$= D_1 \times 10^2 + D_2 \times 10 + D_3 \text{ in the first twelve bits of } PN_1.$$

After the sixth Extract-Multiply operation, PN_1 will contain $D_1 \times 10^6 + D_2 \times 10^5 + D_3 \times 10^4 + D_4 \times 10^3 + D_5 \times 10^2 + D_6 \times 10 + D_7$ which is the original number N converted as a decimal coded binary whole number to a binary whole number.

The following shows the PPR tracer on a series of six Extract-Multiply commands doing the conversion operation described above. The top half of the page was typed by the author when setting up the demononstration. The program which is traced is given below:

L	T	N	C	S	D	
04	06	07	3	23	31	Extract
07	06	14	0	24	31	MPY
14	16	17	3	23	31	Extract
17	06	24	0	24	31	MPY
24	26	27	3	23	31	Extract
27	06	34	0	24	31	MPY
34	36	37	3	23	31	Extract
37	06	44	0	24	31	MPY
44	46	47	3	23	31	Extract
47	06	54	0	24	31	MPY
54	56	57	3	23	31	Extract
57	06	64	0	24	31	MPY
64	66	64	0	16	31	Halt

05	z000000	E_1
15	zz00000	E_2
25	zzz0000	E_3
35	zzzz000	E_4
45	zzzzz00	E_5
55	zzzzzz0	E_6

The number being converted is 2222222. The answer, which appeared in hex as 217887 should be equal to 2222222 in decimal.

2×16^5	=	2, 097, 152
1×16^4	=	65, 536
$y \times 16^3$	=	57, 344
8×16^2	=	2, 048
8×16^1	=	128
$y \times 16^0$	=	14
		2, 222, 222

```
py04  s
.04   .8505000  x00  sy04  s
.04  060732331  s
.07  061402431  s
.14  161732331  s
.17  062402431  s
.24  262732331  s
.27  063402431  s
.34  363732331  s
.37  064402431  s
.44  464732331  s
.47  065402431  s
.54  565732331  s
.57  066402431  s
.64  666401631  s
.64   .424021z  z05  z000000  s
.05  z15  zz00000  s
.15  z25  zzz0000  s
.25  z35  zzzz000  s
.35  z45  zzzzz00  s
.45  z55  zzzzzz0  s
.55  x06  s-.54xzxw3
02x03  ssc7fq03002zz  siq2222222  81002zu  siqv6xv680  81002z8  siq
/p0204  s
.04   .06.07.3.23.31
.07   .06.14.0.24.31   .0000000  .1622222
.14   .16.17.3.23.31
.17   .06.24.0.24.31   .0000000  .0xy2222
.24   .26.27.3.23.31
.27   .06.34.0.24.31   .0000000  .08uy222
.34   .36.37.3.23.31
.37   .06.44.0.24.31   .0000000  .056wy22
.44   .46.47.3.23.31
.47   .06.54.0.24.31   .0000000  .03640y2
.54   .56.57.3.23.31
.57   .06.64.0.24.31   .0000000  .021y88y  <------  2, 222, 222 converted
.64   .66.64.0.16.31                                 to binary.  Type-out
                                                     in hex.
```

b) Input Scale Factors

The job of the input scale factor is to eliminate all powers of ten by which a number is scaled and divide by enough powers of two to effect the scaling which is intended for the number.

Suppose a number ranges from 0 to 500. A scaling of 2^{-9} could be used, since $2^9 = 512$, and the actual maximum value which could be stored with a 2^{-9} scaling is 511.9999. As explained under input conversion, the process described converts whole numbers only. In other words, this particular number would be converted as if it were 5,119,999. The input scale factor for this number must divide by 10^4, as well as by 2^9. This will reduce its value to 511.9999 (eliminate all powers of 10) and scale it 2^{-9}. The scale factor can be made up using the G-15 Reference Card.

```
          2     7     1     0      add 9 binary 0's = 10^4 · 2^9
10^4 = . 0010  0111  0001  0000 . 000000000 .
        0  4     y     2     0      0     0 .
```

Input scale factor = $10^4 \cdot 2^9$ = 04y2000. This input scale factor must be divided into the number, giving

$$\frac{5,119,999}{10^4 \cdot 2^9} = 511.9999\ (2^{-9})$$

An input scale factor which would be multiplied by the number could also be compiled:

```
          0     0     0     6     8     x     v     8     v     v
10^-4 = .0000  0000  0000  0110  1000  1101  1011  1000  1011  1011
```

add 9 more binary 0's = 2^{-9}

```
                                6     8
 0000  0000  0000  0000  0000  0011  0100
   0     0     0     0     0     3     2
```

= .0000032 hex. This is a very poor scale factor, since it is not exact, or even very accurate. In general, all input scale factors should be made up for division into the number to be scaled.

There are cases where input scale factors are not needed. Tally limits are an example. No fractional part of the number is needed, therefore scaling is unnecessary. Simply take the number from the decimal to binary conversion routine as a binary whole number and store without scaling.

Perhaps it would be a good idea to review the foregoing information in order to help the reader's understanding.

The decimal to binary conversion process explained in this manual converts whole numbers only. Or, to say it another way, it converts all numbers as if they were whole numbers. Thus, the number 511.9999 is converted as if it were 5,119,999. and is effectively multiplied by 10^4.

The intended scaling of the number, (in this case 2^{-9}) and elimination of the extra powers of ten (in this case 10^4) can be done in one process by dividing by $10^4 \cdot 2^9$. An example is given next, in the form of a traced program. The program being traced is given following the example.

The program is contained in one line and must be loaded with p key. Type in the decimal number to be converted followed by tab s. Then type the binary scale factor in hex followed by tab s. Typeout will be : $\overline{N}$ and N scaled. $\overline{N}$ is the decimal number converted to binary as if it were a whole number. N scaled is the result of dividing the scale factor into $\overline{N}$. If this division produced overflow, a bell ring will occur and a . wwwwwww flag will follow N scaled to indicate it is incorrect.

The example being traced is 200. 9375 being converted and scaled 2^{-9}. The scale factor has previously been compiled as $10^4 \cdot 2^9$ = 04y2000. The type-out shows $\overline{N}$ = 2, 009, 375 = 1yu91z and N = 200. 9375 (2^{-9}) = 6478000. N should equal 200. 9375:

```
   6    4    7    8
 0110 0100 0111 1000
   w    8   ,z
```

$w \times 16^1$	=	192
8×16^0	=	8
$z \times 16^{-1}$	=	.9375
		200.9375

The program was reloaded and used without the tracer at the bottom of the page. The first example is the same one that was traced. The second one is 189. 9870 being converted to binary and scaled 2^{-8}. The scale factor would be $10^4 \cdot 2^8$ =

```
   2    7    1    0 .  add 8 more zeros
 0010 0111 0001 0000.0000 0000
   2    7    1    0    0    0 .
```

The typeout shows $\overline{N}$ = 1, 889, 870 = 1wzx5y hex and N = 189. 9870 (2^{-8}) = vxzwuw0. N should equal 189. 9870:

```
   v    x    z    w    u    w    0
 1011 1101 1111 1100 1010 1100 0000
   v    x .  z    w    u    w    0
```

$v \times 16^1$	=	176
$x \times 16^0$	=	13
$z \times 16^{-1}$	=	.9375
$w \times 16^{-2}$	=	.046875
$u \times 16^{-3}$	=	.0024414
$w \times 16^{-4}$	=	.0001831
		189.9869995

The last example shows . 9375000 being scaled 2^0. The scale factor is $10^7 \cdot 2^0$ = 0989680 hex. $\overline{N}$ = 9, 375, 000 is 8z0x18 hex and N = . 9375000 (2^0) = . z000000 $\quad$. z = $z \times 16^{-1}$ = 15/16 = . 9375

```
ppsc5f1900  s
.00  u.01.01.0.19.02    8404uvz
.01  w.04.04.2.21.31
.04   .05.90.0.02.28   .4400000
.90   .03.29.0.28.03    8w00001
.29   .31.06.0.08.31

.06   12.07.0.29.23     0000000
.07   .07.07.0.28.31
.08   .10.09.0.12.31   2009375  s
.09   .12.12.0.23.31
.12   13.10.0.02.24     v6xv680
.10   .10.10.0.28.31
.11   .12.15.6.23.26    0000000  .2009375
.15   .12.17.0.29.23   .0000000
.17   19.16.0.29.31
.16   .18.18.0.12.31   04y2000  s
.18   18.18.0.28.31
.19   20.22.0.23.28    .04y2000
.22   .24.25.3.23.31
.25   .06.32.0.24.31   .0000000   1409375
.32   34.35.3.23.31
.35   .06.42.0.24.31    0000000  .0w89375
.42   44.45.3.23.31
.45   .06.52.0.24.31    0000000   07x9375
.52   54.55.3.23.31
.55   06.62.0.24.31    .0000000  .04y7x75
.62   .64.65.3.23.31
.65   06.72.0.24.31    .0000000  .0310y95
.72   .74.75.3.23.31
.75   06.82.0.24.31     0000000  .01yu91z
.82   .83.84.0.26.23   .01yu91z
.84   .85.87.2.28.25   .04y2000
.87   v6.96.1.25.31     0000001  .6478000
.96   .97.u1.0.24.28   .6478000
.u1   u2.u4.0.28.23     6478000
.u4   .00.20.4.29.23    0000000  .0000000
.20   .22.26.0.29.31
.26  u.27.u3.0.29.19    0000000
.u3  u.00.24.0.23.19    0000000
.24   .26.28.0.09.31    01yu91z   .6478000
.28   28.28.0.28.31
.29   .31.06.0.08.31

.06   .12.07.0.29.23    0000000
.07   .07.07.0.28.31
.08   10.09.0.12.31
```

Number type-in in decimal coded binary (or hex) [arrow to 2009375 s]

Binary scale factor type-in in hex [arrow to 04y2000 s]

$\bar{N}$ = 2, 009, 375 in binary (type-out in hex) [arrow to 01yu91z]

N = 200. 9375 (2^{-9}) in binary (typeout in hex) [arrow to .6478000]

Dec. No.	S. F.		p $\bar{N}$	N
2009375	s4y2000	s	01yu91z	.6478000
1899870	s271000	s	.01wzx5y	vxzwuw0
9375000	s989680	s	08z0x18	.z000000
Type-in			Type-out	

Butler Manufacturing Company
Kansas City 26, Mo.

Bendix G-15D Program Sheet

Page 1 of
Date: 2-22-59
Line 02

Prepared by R. T. Smith Problem Input Study Routine

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39
40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59
60	61	62	63
64	65	66	67
68	69	70	71
72	73	74	75
76	77	78	79
80	81	82	83
84	85	86	87
88	89	90	91
92	93	94	95
96	97	98	99
u0	u1	u2	u3
u4	u5	u6	u7

L	P	T	N	C	S	D	BP	NOTES
00	u	01	01	0	19	02		L19 → L02
01		03	04	2	21	31		NC: 2. 04
02		10	00	0	00	00		Format for
03		-8	w	0	00	00		SPDDDDDDDTE
04		05	90	0	02	28		02. 05 → ARc
05		44	00	0	00	00		Format for CE
90		03	29	0	28	03		AR → 03. 02
29		31	06	0	08	31		Type AR
06		12	07	0	29	23		Clear 23. 0
07		07	07	0	28	31		Ready → Test
08		10	09	0	12	31		Gate Type-in (N)
09		12	12	0	23	31		Clear
12		13	10	0	02	24		02. 13 → MQ1
13		V6	X	V	68	0		7 u's
10		10	10	0	28	31		Ready → Test
11		12	15	6	23	26		23. 0 CVA → PN1
15		12	17	0	29	23		Clear 23. 0 & Skip 1 rev
17		19	16	0	29	31		Reset 0' flo
16		18	18	0	12	31		Gate type-in (S. F.)
18		18	18	0	28	31		Ready → Test
19		20	22	0	23	28		23. 0 = S. F. → ARc

BUTLER

Butler Manufacturing Company
Kansas City 26, Mo.

Bendix G-15D Program Sheet

Page 2 of ___
Date: 2-22-59
Line 02

Prepared by R. T. Smith   Problem Input Study Routine

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39
40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59
60	61	62	63
64	65	66	67
68	69	70	71
72	73	74	75
76	77	78	79
80	81	82	83
84	85	86	87
88	89	90	91
92	93	94	95
96	97	98	99
u0	u1	u2	u3
u4	u5	u6	u7

L	P	T	N	C	S	D	BP	NOTES
22		24	25	3	23	31		Extract
23	<	z	00	0	00	0		E1
25		06	32	0	24	31		MPY
32		34	35	3	23	31		Extract
33	<	z	z0	0	00	0	>	E2
35		06	42	0	24	31		MPY
42		44	45	3	23	31		Extract
43	<	z	zz	0	00	0		E3
45		06	52	0	24	31		MPY
52		54	55	3	23	31		Extract
53	<	z	zz	z	00	0	>	E4
55		06	62	0	24	31		MPY
62		64	65	3	23	31		Extract
63	<	z	zz	z	z0	0		E5
65		06	72	0	24	31		MPY
72		74	75	3	23	31		Extract
73	<		zz	z	zz	z0	>	E6
75		06	82	0	24	31		MPY
82		83	84	0	26	23		PN1 → 23. 3
84		85	87	2	28	25		AR → ID1
87		v6	96	1	25	31		Divide
96		97	u1	0	28	23		MQ1 → AR_c
u1		u2	u4	0	28	23		AR → 23. 2
u4		00	20	4	29	23		Clear 23. 0, 1

BUTLER

Butler Manufacturing Company
Kansas City 26, Mo.

Bendix G-15D Program Sheet

Page 3 of ___
Date: 2-22-59
Line 02

Prepared by R. T. Smith Problem Input Study Routine

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39
40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59
60	61	62	63
64	65	66	67
68	69	70	71
72	73	74	75
76	77	78	79
80	81	82	83
84	85	86	87
88	89	90	91
92	93	94	95
96	97	98	99
u0	u1	u2	u3
u4	u5	u6	u7

L	P	T	N	C	S	D	BP	NOTES
20		22	26	0	29	31		0' flo → Test
26	u	27	u3	0	29	19		Clear L19
u3	u	00	24	0	23	19		L23 → 19. u4-u7
24		26	28	0	09	31		Type L19
28		28	28	0	28	31		Ready → Test
29		See Page 1						
		0' flo: Ring bell, set flag						
27		28	36	0	17	31		Ring bell
36		37	26	0	02	23		02. 37 → 23. 1
37	<	ww	w	w	ww	w	>	Flag

Type-out will be:

$\overline{N}$ (unscaled), N (scaled), Flag (if 0' flo)