

TECHNICAL NOTE #2602

Author: Bruce Patterson  
Date: July 7, 1976  
Subject: BASIC-2 TEXT ATOMIZATION

In order to conserve memory and optimize program line interpretation, BASIC-2 atomizes program text when RETURN (EXEC) is pressed. Most BASIC words are replaced by single byte codes, called text atoms (see following page). The text atom is an 8-bit code with high order bit on; the lower 7-bits specify the particular BASIC word. Line numbers and line number references are stored in pack decimal form (2 bytes) preceded by the text atom FF<sub>16</sub>.

Most atoms can be used to enter the associated BASIC word for Console Input or INPUT operations. However, E5<sub>16</sub> and E6<sub>16</sub> are interpreted differently. E5<sub>16</sub> represents line erase for CI, INPUT, and LINPUT operations. E6<sub>16</sub> represents the statement number key and causes a new line number to be generated for CI mode; E6<sub>16</sub> is ignored by INPUT and LINPUT.

Programs saved on disk are stored in atomized form.

BASIC-2 KEY WORDS

80 LIST	90 TRACE	A0 PRINT
81 CLEAR	91 LET	A1 LOAD
82 RUN	92 FIX(	A2 REM
83 REMEMBER	93 DIM	A3 RESTORE
84 CONTINUE	94 ON	A4 PLOT
85 PAUSE	95 STOP	A5 SELECT
86 LIMITS	96 END	A6 COM
87 COPY	97 DATA	A7 PRINTUSING
88 KEYIN	98 READ	A8 MAT
89 DEKIP	99 INPUT	A9 REWIND
8A AND	9A GOSUB	AA SKIP
8B OR	9B RETURN	AB BACKSPACE
8C XOR	9C GOTO	AC SCRATCH
8D TEMP	9D NEXT	AD MOVE
8E DISK	9E FOR	AE CONVERT
8F TAPE	9F IF	AF [SELECT] PLOT

B0 STEP	C0 FN	D0 [ARC] SIN(
B1 THEN	C1 ABS(	D1 [ARC] COS(
B2 TO	C2 SQR(	D2 HEX(
B3 EEC	C3 COS(	D3 STR(
B4 OPEN	C4 EXP(	D4 ATN(
B5 [SELECT] CI	C5 INT(	D5 LEN(
B6 [SELECT] R	C6 LOG(	D6 RE
B7 [SELECT] D	C7 SIN(	D7 [SELECT] #
B8 [SELECT] CO	C8 SGN(	D8 % [image]
B9 LGT(	C9 RND(	D9 [SELECT] P
BA OFF	CA TAN(	DA BT
BB DBACKSPACE	CB ARC	DB [SELECT] G
BC VERIFY	CC #PI	DC VAL(
BD DA	CD TAB(	DD NUM(
BE BA	CE DEFFN	DE BIN(
BF DC	CF [ARC] TAN(	DF POS(

E0 LS=	F0 LINPUT
E1 ALL	F1 VER(
E2 PACK	F2 ELSE
E3 CLOSE	F3 SPACE
E4 INIT	F4 ROUND
E5 HEX	F5 AT(
E6 UNPACK	F6 HEXOF(
E7 BOOL	F7 MAX(
E8 ADD	F8 MIN(
E9 ROTATE	F9 MOD(
EA * [stmt]	FA reserved
EB ERROR	FB reserved
EC ERR	FC reserved
ED DAC	FD reserved
EE DSC	FE reserved
EF SUB	FF packed-line-number

TECHNICAL NOTE #2602

Author: Bruce Patterson  
Date: July 7, 1976  
Subject: BASIC-2 TEXT ATOMIZATION

In order to conserve memory and optimize program line interpretation, BASIC-2 atomizes program text when RETURN (EXEC) is pressed. Most BASIC words are replaced by single byte codes, called text atoms (see following page). The text atom is an 8-bit code with high order bit on; the lower 7-bits specify the particular BASIC word. Line numbers and line number references are stored in pack decimal form (2 bytes) preceded by the text atom FF<sub>16</sub>.

Most atoms can be used to enter the associated BASIC word for Console Input or INPUT operations. However, E5<sub>16</sub> and E6<sub>16</sub> are interpreted differently. E5<sub>16</sub> represents line erase for CI, INPUT, and LINPUT operations. E6<sub>16</sub> represents the statement number key and causes a new line number to be generated for CI mode; E6<sub>16</sub> is ignored by INPUT and LINPUT.

Programs saved on disk are stored in atomized form.

BASIC-2 TEST ATOMS

80 LIST  
 81 CLEAR  
 82 RUN  
 83 REMEMBER  
 84 CONTINUE  
 85 SAVE  
 86 LIMITS  
 87 COPY  
 88 KEYIN  
 89 SKIP  
 8A AND  
 8B OR  
 8C XOR  
 8D TEMP  
 8E DISK  
 8F TAPE

90 TRACE  
 91 LET  
 92 FIX(  
 93 DIM  
 94 ON  
 95 STOP  
 96 END  
 97 DATA  
 98 READ  
 99 INPUT  
 9A GOSUB  
 9B RETURN  
 9C GOTO  
 9D NEXT  
 9E FOR  
 9F IF

A0 PRINT  
 A1 LOAD  
 A2 REM  
 A3 RESTORE  
 A4 PLOT  
 A5 SELECT  
 A6 COM  
 A7 PRINTUSING  
 A8 MAT  
 A9 REWIND  
 AA SKIP  
 AB BACKSPACE  
 AC SCRATCH  
 AD MOVE  
 AE CONVERT  
 AF [SELECT] PLOT

B0 STEP  
 B1 THEN  
 B2 TO  
 B3 BEG  
 B4 OPEN  
 B5 [SELECT] CI  
 B6 [SELECT] R  
 B7 [SELECT] D  
 B8 [SELECT] CO  
 B9 LGT(  
 BA OFF  
 BB DBACKSPACE  
 BC VERIFY  
 BD DA  
 BE BA  
 BF DC

C0 FN  
 C1 ABS(  
 C2 SQR(  
 C3 COS(  
 C4 EXP(  
 C5 INT(  
 C6 LOG(  
 C7 SIN(  
 C8 SGN(  
 C9 RND(  
 CA TAN(  
 CB ARC  
 CC #PI  
 CD TAB(  
 CE DEFFN  
 CF [ARC] TAN(  
 D0 [ARC] SIN(  
 D1 [ARC] COS(  
 D2 HEX(  
 D3 STR(  
 D4 ATN(  
 D5 LEN(  
 D6 RE  
 D7 [SELECT] #  
 D8 % [image]  
 D9 [SELECT] P  
 DA BT  
 DB [SELECT] G  
 DC VAL(  
 DD NLM(  
 DE BIN(  
 DF POS(  
 E0 LS=  
 E1 ALL  
 E2 PACK  
 E3 CLOSE  
 E4 INIT  
 E5 HEX  
 E6 UNPACK  
 E7 BOOL  
 E8 ADD  
 E9 ROTATE  
 EA \* [stmt]  
 EB ERROR  
 EC ERR  
 ED DAC  
 EE DSC  
 EF SUB  
 F0 LINPUT  
 F1 VER(  
 F2 ELSE  
 F3 SPACE  
 F4 ROUND  
 F5 AT(  
 F6 HEXOF(  
 F7 MAX(  
 F8 MIN(  
 F9 MOD(  
 FA reserved  
 FB reserved  
 FC reserved  
 FD reserved  
 FE reserved  
 FF packed-line-number

D0 [ARC] SIN(  
 D1 [ARC] COS(  
 D2 HEX(  
 D3 STR(  
 D4 ATN(  
 D5 LEN(  
 D6 RE  
 D7 [SELECT] #  
 D8 % [image]  
 D9 [SELECT] P  
 DA BT  
 DB [SELECT] G  
 DC VAL(  
 DD NLM(  
 DE BIN(  
 DF POS(  
 E0 LS=  
 E1 ALL  
 E2 PACK  
 E3 CLOSE  
 E4 INIT  
 E5 HEX  
 E6 UNPACK  
 E7 BOOL  
 E8 ADD  
 E9 ROTATE  
 EA \* [stmt]  
 EB ERROR  
 EC ERR  
 ED DAC  
 EE DSC  
 EF SUB  
 F0 LINPUT  
 F1 VER(  
 F2 ELSE  
 F3 SPACE  
 F4 ROUND  
 F5 AT(  
 F6 HEXOF(  
 F7 MAX(  
 F8 MIN(  
 F9 MOD(  
 FA reserved  
 FB reserved  
 FC reserved  
 FD reserved  
 FE reserved  
 FF packed-line-number

E0 LS=  
 E1 ALL  
 E2 PACK  
 E3 CLOSE  
 E4 INIT  
 E5 HEX  
 E6 UNPACK  
 E7 BOOL  
 E8 ADD  
 E9 ROTATE  
 EA \* [stmt]  
 EB ERROR  
 EC ERR  
 ED DAC  
 EE DSC  
 EF SUB

F0 LINPUT  
 F1 VER(  
 F2 ELSE  
 F3 SPACE  
 F4 ROUND  
 F5 AT(  
 F6 HEXOF(  
 F7 MAX(  
 F8 MIN(  
 F9 MOD(  
 FA reserved  
 FB reserved  
 FC reserved  
 FD reserved  
 FE reserved  
 FF packed-line-number

TECHNICAL NOTE #2601

Author: Bruce Patterson  
Date: July 7, 1976  
Subject: BASIC-2 MATRIX INVERSION

BASIC-2 performs matrix inversion by the Gauss-Jordan Elimination method done in place with maximum pivot strategy. The determinant is calculated during the inversion. If specified, the normalized determinant is also calculated. The technique is outlined on p. 3 and a BASIC emulation of the technique is provided on p. 5.

Several types of errors can occur during matrix inversion:

1. singular matrix
2. computation errors (overflow or underflow)
3. round-off error accumulation
4. ill-conditioned matrix

If a matrix is singular (i.e., has no inverse), the determinant of that matrix is zero. If a determinant variable is not included in the MAT INV statement, the system will stop with an ERR 72. If a determinant variable is included in the MAT INV statement, that variable is set equal to zero and program execution continues; the resultant matrix contains meaningless values. It is the program's responsibility to check the determinant after each inversion.

Computational errors can occur during the calculation of the inverse. Generally, it is best to let underflow default to zero; however, overflow should be detected (i.e., SELECT ERROR > 60). If overflow is detected the matrix being inverted should be scaled down by dividing each element by some constant before the inversion is performed.

Round-off error accumulation results from the successive calculations performed during the inversion. By utilizing the maximum pivot strategy, BASIC-2 reduces this type of error; however, some loss of precision will occur, especially, with large matrices. Round-off error can be detected by calculating the residuals defined by:

$$R = I - A C$$

where: I = identity matrix  
A = matrix being inverted  
C = computed inverse

For an exact inverse each element of R will be zero. If C is a good approximation to the inverse, then each element of R will be small.

There are matrices for which the residual does not provide a reasonable measure of the accuracy of the resultant inverse. Such matrices are said to be ill-conditioned. Loss of accuracy is not due to round-off error accumulation or the algorithm chosen to perform the inversion but rather to the nature of the data itself. Typically, the size of the determinant is used to detect ill-conditioned matrices (small determinants indicating potential problems). However, for an accurate measure of the condition of the matrix, the determinant must be normalized relative to the matrix being inverted; thus, BASIC-2 provides the normalized determinant. The normalized determinant for a matrix A is defined as follows:

$$\text{if } A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \cdot & & & \\ \cdot & & & \\ \cdot & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}$$

$$\text{and } \alpha_k = \sqrt{a_{k1}^2 + a_{k2}^2 + \dots + a_{kn}^2}$$

$$\text{then NORM } |A| = \begin{vmatrix} a_{11}/\alpha_1 & a_{12}/\alpha_1 & \dots & a_{1n}/\alpha_1 \\ a_{21}/\alpha_2 & a_{22}/\alpha_2 & \dots & a_{2n}/\alpha_2 \\ \cdot & & & \\ \cdot & & & \\ \cdot & & & \\ a_{n1}/\alpha_n & a_{n2}/\alpha_n & \dots & a_{nn}/\alpha_n \end{vmatrix}$$

$$= \frac{|A|}{\alpha_1 \alpha_2 \dots \alpha_n}$$

If the normalized determinant of a matrix A is small relative to one, then A is nearly singular and ill-conditioning can be expected.

## Outline of BASIC-2 Inversion Technique

END

Inversion is performed by the Gauss-Jordan Elimination Method done in place with maximum pivot strategy. The determinant is calculated during the inversion. If a norm variable is specified, the normalized determinant is calculated.

METHOD for MAT c = INV(a)

- A. MAT c = MAT a
- B. If norm variable is specified, calculate determinant normalizer.
- C. determinant = 1
- D. N passes are made through the matrix. On the ith pass, the following is done.
  1. Find maximum pivot element (maximum element in lower right subarray whose element (1,1) is element (I,I) of entire array. Matrix is singular if maximum pivot element = 0.
  2. Move pivot element to pivot position:
    - a. If pivot element is in row J switch rows I & J.
    - b. If pivot element is in column K, switch columns I & K.
  3. Normalization: divide each element of pivot row by pivot element except element in pivot position = 1 / pivot element. Multiply determinant by pivot element.
  4. Elimination: eliminate all elements in pivot column except pivot element by multiplying the pivot row by the element in the pivot column to be eliminated and subtracting the result from the row containing the element to be eliminated. But let element to be eliminated = - ( that element ) \* (pivot element).
- E. Unscramble inverse: switch columns and rows of matrix in reverse order than they were switched by step D2.
- F. norm = determinant / normalizer

REFERENCES (INVERSION METHOD):

1. Kuo, Shan S., "COMPUTER APPLICATIONS OF NUMERICAL METHODS", Addison-Wesley, 1972, p. 176-203.
2. Carnahan, Luther, Wilkes, "APPLIED NUMERICAL METHODS", Wiley, 1969, p. 269-296..
3. Conte, S. D., "ELEMENTARY NUMERICAL ANALYSIS", McGraw-Hill, 1965, p. 156-177.

REFERENCES (ILL-CONDITIONED MATRICES, ROUND-OFF ERRORS, OTHER TECHNIQUES):

1. Forsythe, Moler, "COMPUTER SOLUTION OF LINEAR ALGEBRAIC SYSTEMS", Prentice-Hall, 1967.



0010 REM %

# MATINV 2/17/76 -- MATRIX INVERSION

```
0020 REM AUTHOR; BRUCE PATTERSON
0030 REM
0040 REM EMULATION OF BASIC-2 MATRIX INVERSION MICROCODE.
0050 REM
0060 REM METHOD; GAUSS-JORDAN ELIMINATION DONE IN PLACE WITH
0070 REM . MAXIMUM PIVOT STRATEGY.
0080 REM VARIABLES;
0090 REM . A() - MATRIX BEING INVERTED
0100 REM . P() - SUBSCRIPTS OF PIVOT ELEMENTS
0110 REM . M - ORDER OF MATRIX
0120 REM . D - DETERMINANT
0130 REM . N - NORMALIZED DETERMINANT
0140 REM . R - ROW INDEX
0150 REM . C - COLUMN INDEX
0160 REM . P - PIVOT POSITION (PASS #)
0170 REM . B() - ORIGINAL MATRIX
0175 REM
0180 SELECT PRINT 005(64), INPUT 001
: PRINT HEX(03)
: PRINT , "*** MATRIX INVERSION ***"
: PRINT
0190 DIM A(15,15), P(15,2), B(15,15), T$64, C(15,15)
0200 REM %
```

## READ MATRIX

```
0210 READ M
REM M = ORDER OF MATRIX
0220 FOR R=1 TO M
: FOR C=1 TO M
:- READ A(R,C)
: B(R,C)=A(R,C)
: NEXT C
: NEXT R
: REM A() & B() = MATRIX TO INVERT
0230 REM %
```

## DISPLAY MATRIX BEFORE INVERSION

```
: PRINT HEX(03)
: GOSUB 15("*** MATRIX BEING INVERTED ***")
0240 REM %
```

## CALC. DET. NORMALIZER, N

```
0250 N,D=1
0260 FOR R=1 TO M
: X=0
: FOR C=1 TO M
: X=X+A(R,C)*A(R,C)
: NEXT C
N=N*SQR(X)
NEXT R
0270 REM %↑
```

\*\*\*\*\* CALCULATE INVERSE \*\*\*\*\*

```
0280 FOR P=1 TO M
: REM MAKE M PASSES THROUGH MATRIX
0290 REM Z
```

PIVOT ELEM. =MAX. ELEM. IN SUBARRAY

```
0300 X=0
: FOR R=P TO M
: FOR C=P TO M
: IF ABS(A(R,C))>ABS(X) THEN 310
: R1=R
: C1=C
: X=A(R,C)
0310 NEXT C
: NEXT R
: REM PIVOT ELEMENT , A(R1,C1) = X
0320 P(P,1)=R1
: P(P,2)=C1
: REM SAVE PIVOT ELEMENT SUBSCRIPTS
0330 IF A(R1,C1) <> 0 THEN 340
: PRINT "MATRIX SINGULAR"
: D,N=0
: END
```

```
0340 REM Z
```

MOVE PIVOT ELEM. TO PIVOT POSITION

```
0350 IF P=R1 THEN 360
: GOSUB '50(P,R1)
: D=-D
: REM SWITCH ROWS
0360 IF P=C1 THEN 370
: GOSUB '51(P,C1)
: D=-D
: REM SWITCH COLUMNS
0370 REM Z
```

NORMALIZE ROW P

```
: X=A(P,P)
: D=D*X
: IF D<>0 THEN 380
: N=0
: PRINT "MATRIX SINGULAR"
: END
0380 A(P,P)=1
: FOR C=1 TO M
: A(P,C)=A(P,C)/X
: NEXT C
0390 REM Z
```

ELIMINATION

```
: FOR R=1 TO M
: IF R=P THEN 410
```

```

: X=A(R,P)
: A(R,P)=0
0400 FOR C=1 TO M
: A(R,C)=A(R,C)-X*A(P,C)
: NEXT C
0410 NEXT R
: NEXT P
: P=0
0420 REM %

```

## UNSCRAMBLE INVERSE

```

: FOR R=M TO 1 STEP -1
: IF P(R,2)=R THEN 430
: GOSUB '50(R,P(R,2))
: REM SWITCH ROWS
0430 IF P(R,1)=R THEN 440
: GOSUB '51(R,P(R,1))
: REM SWITCH COLUMNS
0440 NEXT R
0450 REM %

```

\*\*\*\*\* INVERSION COMPLETE \*\*\*\*\*

```
0460 REM %
```

## DISPLAY MATRIX

```

: GOSUB '15("*** INVERSE ***")
0470 N=D/N
PRINT "DET =";D,"NORM =";N
END

```

```
0480 REM %
```

## SWITCH ROWS

```

0490 DEFFN'50(R2,R3)
: IF R2=R3 THEN 500
: FOR J=1 TO M
: X=A(R2,J)
: A(R2,J)=A(R3,J)
: A(R3,J)=X
: NEXT J
0500 RETURN

```

```
0510 REM %
```

## SWITCH COLUMNS

```

0520 DEFFN'51(C2,C3)
: IF C2=C3 THEN 530
: FOR I=1 TO M
: X=A(I,C2)
: A(I,C2)=A(I,C3)
: A(I,C3)=X
: NEXT I
0530 RETURN

```

```
0540 REM %
```

# PRINT MATRIX

```
0550 DEFFN IS(T#)
      :PRINT
      PRINT T#
      PRINT
0560 FOR R2=1 TO M
      : FOR C2=1 TO M
      : PRINT A(R2,C2),
      : NEXT C2
      : PRINT
      : NEXT R2
      :PRINT
      :RETURN
```

```
0570 REM %
```

## MATRIX TO BE INVERTED

```
0580 REM ORDER OF MATRIX
      : DATA 3
0590 REM ROW 1
      : DATA -3,8,5
0600 REM ROW 2
      : DATA 2,-7,4
0610 REM ROW 3
      : DATA 1,9,-6
```

**TECHNICAL NOTE #2606**

**Author:** 2200 Development group  
**Date:** May 21, 1979  
**Subject:** VP/MVP Random Numbers

Several people have asked recently about the algorithms used for random numbers on the 2200 VP and MVP. This memo will describe the implementation as of VP Release 2.0 and MVP Release 1.7. Details are subject to change on future releases if we think necessary.

The programming definition of a RND function in BASIC-2 is that it should produce successive terms from a linearly distributed (between 0 and 1) psuedo random sequence. RND(0) initializes the sequence and RND(1) gets the next term of the sequence. Master Initialization, CLEAR, and LOAD RUN "randomize" the sequence. The purpose here is to describe specifically what is done, so that the random numbers may be used with more confidence.

The heart of the random number generator is a binary generator called BINRAN. BINRAN generates a sequence of 32 bit integers as follows:

B(0) = HEX(10DF5D09)  
B(n+1) = B(n) \* HEX(00010005)  
( all arithmetic done with 32 bit binary integers )

This binary sequence is set to B(0) by RND(0). The exact process of randomizing for Master Initialization, CLEAR, and LOAD RUN will not be detailed in this memo, as a change in a future release is under consideration at present.

Each decimal random number (13 digits) is generated by repeated calls to the binary generator BINRAN. The number is created one digit at a time. First the odd digits (1,3,5, ... 13) are generated, and then the even. For each odd digit, one or more calls to BINRAN are made until the second hex-digit of the 32 bit binary value is a valid BCD digit. After all odd digits are placed in the value, then the even digits are similarly built, using the first hex-digit of the binary value.

I have written a program, called RNDGEN, which emulates (in BASIC-2) the method described above. The listing of the program is enclosed with this memo.

```
010 REM RNDTEST - CHECK OUT RANDOM NUMBER ALGORITHM ON A
    NGEL 10/11/79
0020 DIM A$(8),A1$(8),S$,CS$,S$14
0030 PRINT HEX(05); "RANDOM NUMBER DEMONSTRATION PROGRAM"
    : PRINT
0040 PRINT "    THE FOLLOWING PROGRAM DEMONSTRATES THE RAND
    OM NUMBER GENERATION"
0050 PRINT "    ON THE 2200VP (RELEASE 2.0) AND THE 2200MV
    P (RELEASE 1.7). "
    : PRINT
0060 PRINT "    THERE IS A SEQUENCE OF 32 BIT BINARY RANDOM
    NUMBERS, STARTING AT A "
0070 PRINT "    SPECIFIC BINARY RANDOM SEED, FROM WHICH TH
    E DECIMAL ALGORITHM "
0080 PRINT "    EXTRACTS BCD DIGITS. THESE DIGITS ARE ASSE
    MBLD INTO 13 DIGIT "
0090 PRINT "    FRACTIONS, THEN NORMALIZED AND RETURNED TO
    THE RND FUNCTION."
0100 PRINT "    FIRST THE ODD PLACES ARE FILLED FROM THE SE
    COND DIGITS OF EACH "
0110 PRINT "    TERM IN THE BINARY SEQUENCE, THEN THE EVEN
    PLACES ARE FILLED FROM "
0120 PRINT "    THE FIRST DIGITS. ANY NON-BCD TERM IS DISC
    ARDED IN THE PROCESS."
0130 PRINT
0140 A$=HEX(10DF5D09)
    : PRINT HEXOF(A$); "    = BINARY RANDOM SEED"
0150 G=RND(0)
    : GOTO 170

0160 G=RND(1)
0170 PRINT "    PRESS EXFC TO CONTINUE";
    : KEYIN CS
    : PRINT HEX(00); "*****BINARY*****DECIMAL*****"
    : PRINT "*****";
0180 S$=" .-----"
    : FOR I=1 TO 13 STEP 2
0190 GOSUB 240
    : X=VAL(A1$(2))-48
    : IF X<=9 THEN 200
    : PRINT
    : PRINT A1$();
    : GOTO 190

0200 A1$(2)= OR HEX(80)
    : STR$(S$,I+1,1)=BIN(128+48+X)
    : PRINT
    : PRINT A1$().S$;
    : S$=AND ALL(7F)
    : NEXT I
    : FOR I=2 TO 12 STEP 2
0210 GOSUB 240
    : X=VAL(A1$(1))-48
    : IF X<=9 THEN 220
    : PRINT
    : PRINT A1$();
    : GOTO 210
```

```
0220 A1$(1) = OR HEX(80)
      : STR(C$,I+1,1)=BIN(128+48*X)
      : PRINT
      : PRINT A1$(1),S$;
      : S$=AND ALL(7F)
```

```
      : NEXT I
      : PRINT " <-- NEW RANDOM NUMBER"
0230 CONVERT S$ TO S
      : IF S=0 THEN 160
      : STOP "ERROR!!!"#
      : END
```

```
0240 REM %
```

~~BINRAN BINARY RANDOM NUMBER GENERATOR~~

```
0250 B$=STR(A$) & HEX(0000)
      : C$=A$ ADDC A$ ADDC A$ ADDC A$ ADDC A$ ADDC A$
      : A$=C$
      : HEXUNPACK A$ TO A1$(1)
      : RETURN
```

RANDOM NUMBER DEMONSTRATION PROGRAM

THE FOLLOWING PROGRAM DEMONSTRATES THE RANDOM NUMBER GENERATION ON THE 2200VP (RELEASE 2.0) AND THE 2200MVP (RELEASE 1.7).

THERE IS A SEQUENCE OF 32 BIT BINARY RANDOM NUMBERS, STARTING AT A SPECIFIC BINARY RANDOM SEED, FROM WHICH THE DECIMAL ALGORITHM EXTRACTS BCD DIGITS. THESE DIGITS ARE ASSEMBLED INTO 13 DIGIT FRACTIONS, THEN NORMALIZED AND RETURNED TO THE RND FUNCTION. FIRST THE ODD PLACES ARE FILLED FROM THE SECOND DIGITS OF EACH TERM IN THE BINARY SEQUENCE, THEN THE EVEN PLACES ARE FILLED FROM THE FIRST DIGITS. ANY NON-BCD TERM IS DISCARDED IN THE PROCESS.

10DF5D09 = BINARY RANDOM SEED  
PRESS EXEC TO CONTINUE

*BINARY*	*****	DECIMAL	*****
B165D12D		.1-----	
482A15E1		.1-8-----	
7EB36D65			
E6E622F9		.1-8-6-----	
A577AEDD		.1-8-6-5-----	
EA336A51			
FD521395			
062F61E9		.1-8-6-5-6----	
66D5E98D		.1-8-6-5-6-0-	
6DBA8FC1			
B465CEC5		.1-8-6-5-6-0-4	
54C289D9		.158-6-5-6-0-4	
F1A3313D			
A96CF631			
55516E55		.15846-5-6-0-4	
298E0AC9		.1584625-6-0-4	
DA8F35ED			
7A6989A1		.158462576-0-4	
733E4425		.15846257670-4	
845C54B9		.1584625767084	<-- NEW RANDOM NUMBER

PRESS EXEC TO CONTINUE

*BINARY*	*****	DECIMAL	*****
EA86A79D			
3C3E4611			
73485E55		.3-----	
9EBED7A9			
F163364D		.3-1-	
ED3D0F81			
B1B24D85		.3-1-1-----	
66008399		.3-1-1-6-----	
619E91FD		.3-1-1-6-1----	
7A06D9F1			
3C134185			
6E154889			
6EF36AAD			
956E1561		.3-1-1-6-1-5-	
00876AE5		.3-1-1-6-1-5-0	
6D8A1679		.361-1-6-1-5-0	
3A2E795D		.36131-6-1-5-0	
33631D1		.3613196-1-5-0	
11DFF915		.361319611-5-0	
5274D965		.36131961155-0	
79B1E30D		.3613196115570	<-- NEW RANDOM NUMBER

PRESS EXEC TO CONTINUE



---

### USING THESE FEATURES

These features can be used to increase the speed and efficiency of your system. However, their use is completely dependent on your system's requirements and applications. Before using them fully, you should check and adjust them in a controlled way during a normal system load.

Wang's System Activity Monitor (SAM) or SAM-II products can help pinpoint the busiest areas for your system. Using some of these new features can help you reduce redundant work.

---

### 2200 SYSTEMS: ALGORITHMS USED FOR RANDOM NUMBERS BY BASIC-2

This article provides some insight into the algorithms used to generate random numbers by BASIC-2 for Release 2.5 of the VP and Release 2.6 of the MVP Operating Systems. Although no changes to the algorithm are currently anticipated, details are subject to change on future releases of the operating systems.

The programming definition of a RND function in BASIC-2 is that it should produce successive terms from a linearly distributed (between 0 and 1) pseudo random sequence. RND(0) initializes the sequence and RND(1) gets the next term of the sequence. Master Initialization, CLEAR, and LOAD RUN randomize the sequence. This article describes specifically what is done so that the random numbers may be used with more confidence.

The heart of the random number generator is a binary generator called BINRAN. BINRAN generates a sequence of 32-bit integers as follows:

B(0) = HEX(10DF5D09)  
B(n+1) = B(n) \* HEX(00010005)  
(all arithmetic is done with 32-bit binary integers)

This binary sequence is set to B(0) by RND(0). The exact process of randomizing for Master Initialization, CLEAR, and LOAD RUN varies between the VP and the MVP operating systems. While the operating systems are idle, random numbers are generated.

Each decimal random number (13 digits) is generated by repeated calls to the binary generator, BINRAN. The number is created one digit at a time. First, the odd digits (i.e., digits in the 1/10s, 1/1000s, etc., positions) are generated, and then the even (i.e., 1/100s, 1/10,000s, etc., positions). For each odd digit, one or more calls to BINRAN are made until the second hex-digit of the 32-bit binary value is a valid binary coded decimal (BCD) digit. After all odd digits are placed in the value, then the even digits are similarly built, using the first hex-digit of the binary value.

A following listing of a BASIC-2 program, called RNDGEN, which emulates the method previously described is included.

```
0010 REM RNDGEN - CHECK OUT RANDOM NUMBER ALGORITHM
0020 DIM A$4, A1$(8)1, B$6, C$4, S$14
0030 PRINT HEX(03), "RANDOM NUMBER DEMONSTRATION PROGRAM"
      : PRINT
0040 PRINT "THE FOLLOWING PROGRAM DEMONSTRATES THE RANDOM
      NUMBER"
0050 PRINT " GENERATION BY BASIC-2."
      : PRINT
0060 PRINT "THERE IS A SEQUENCE OF 32 BIT BINARY RANDOM
      NUMBERS,"
0070 PRINT " STARTING AT A SPECIFIC BINARY NUMBER SEED,
      FROM WHICH THE"
0080 PRINT "DECIMAL ALGORITHM EXTRACTS BCD DIGITS. THESE
      DIGITS ARE"
0090 PRINT "ASSEMBLED INTO 13 DIGIT FRACTIONS, THEN
      NORMALIZED AND"
0100 PRINT "RETURNED TO THE RND FUNCTION. FIRST THE ODD
      PLACES ARE"
0110 PRINT "FILLED FROM THE SECOND DIGITS OF EACH TERM IN
      THE BINARY"
0120 PRINT "SEQUENCE, THEN THE EVEN PLACES ARE FILLED FROM
      THE FIRST"
0125 PRINT "DIGITS. ANY NON-BCD TERM IS DISCARDED IN THE
      PROCESS."
0130 PRINT
0140 A$=HEX(10DF5D09)
      : PRINT HEXOF(A$); " = BINARY RANDOM SEED"
0150 Q=RND(0)
      : GOTO 170
0160 Q=RND(1)
0170 PRINT " PRESS EXEC TO CONTINUE";
      : KEYIN C$
      : PRINT HEX(0D); "**BINARY*****DECIMAL*****"
      : *****";
0180 S$=" .-----"
      : FOR I = 1 TO 13 STEP 2
0190 GOSUB 240
      : X=VAL(A1$(2))-48
      : IF X<=9 THEN 200
      : PRINT
      : PRINT A1$();
      : GOTO 190
0200 A1$(2) = OR HEX(80)
      : STR(S$,I+1,1) = BIN(128+ 48+ X)
      : PRINT
      : PRINT A1$(),S$;
      : S$ = AND ALL(7F)
      : NEXT I
      : FOR I = 2 TO 12 STEP 2
0210 GOSUB 240
      : X=VAL(A1$(1))-48
      : IF X<=9 THEN 220
      : PRINT
      : PRINT A1$();
      : GOTO 210
```

```

0220 A1$(1) = OR HEX(80)
      : STR(S$,I+1,1) = BIN(128+ 48+ X)
      : PRINT
      : PRINT A1$( ),S$;
      : S$ = AND ALL(7F)
      : NEXT I
      : PRINT " <-- NEW RANDOM NUMBER"
0230 CONVERT S$ TO S
      : IF S=Q THEN 160
      : STOP "ERROR"#
      : END
0240 REM BINRAN - BINARY RANDOM NUMBER GENERATOR
0250 B$ = STR(A$) & HEX(0000)
      : C$ = A$ ADDC A$ ADDC A$ ADDC A$ ADDC A$ ADDC B$
      : A$=C$
      : HEXUNPACK A$ TO A1$( )
      : RETURN

```

RANDOM NUMBER DEMONSTRATION PROGRAM

THE FOLLOWING PROGRAM DEMONSTRATES THE RANDOM NUMBER GENERATION BY BASIC-2.

THERE IS A SEQUENCE OF 32 BIT BINARY RANDOM NUMBERS, STARTING AT A SPECIFIC BINARY NUMBER SEED, FROM WHICH THE DECIMAL ALGORITHM EXTRACTS BCD DIGITS. THESE DIGITS ARE ASSEMBLED INTO 13 DIGIT FRACTIONS, THEN NORMALIZED AND RETURNED TO THE RND FUNCTION. FIRST THE ODD PLACES ARE FILLED FROM THE SECOND DIGITS OF EACH TERM IN THE BINARY SEQUENCE, THEN THE EVEN PLACES ARE FILLED FROM THE FIRST DIGITS. ANY NON-BCD TERM IS DISCARDED IN THE PROCESS."

```

10DF5D09      = BINARY RANDOM SEED
      PRESS EXEC TO CONTINUE
*BINARY*****DECIMAL*****
B165D12D      .1-----
482A15E1      .1-8-----
7ED36D65
E6E622F9      .1-8-6-----
A577AEDD      .1-8-6-5-----
EA336A51
FD521395
062F61E9      .1-8-6-5-6----
80D5E98D      .1-8-6-5-6-0--
6DBA8FC1
B465CEC5      .1-8-6-5-6-0-4
54C209D9      .158-6-5-6-0-4
B1A3313D
A96CF631
4551CEF5      .15846-5-6-0-4
298E0AC9      .1584625-6-0-4
DA8F35ED
7AB90DA1      .158462576-0-4
733E4425      .15846257670-4
845C54B9      .1584625767084 <-- NEW RANDOM NUMBER
      PRESS EXEC TO CONTINUE

```

TECHNICAL NOTE #2607

Author: 2200 Development Group  
Date: June 25, 1979  
Subject: Disk Error Recovery

A. Classes of Disk Errors

1. Operator/Hardware Errors (I90, I91, I92, I94, I95, I98)

These errors generally require some specific user or Wang service action. The user should assure that the disk has been properly cabled, powered on, platters mounted, and ready for the operations to be performed. If the operation then generates an error, it should be retried several times. On a 2200T, the system should be RESET between each attempt. (The 2200VP/MVP automatically resets the disk whenever a disk I/O error occurs). If the error persists, Wang service may have to be notified since the error may indicate a malfunction in the disk processing unit (DPU) or disk drive itself.

2. Media/Data Errors (I93, I96, I97, I99)

These errors generally indicate a failure in the transmission or recording of data. The failing operations should be retried several times. If the error persists, the platter may have to be reformatted or discarded. If the above procedures fail, Wang service may need to be notified.

B. Recovery Procedures for Specific Disk Errors

I90 (2200T ERR 61) Disk Hardware Error

This error indicates that the disk processing unit did not respond properly to the 2200 at the beginning of a disk operation.

Recovery:

1. Retry the disk operation.
2. Make sure the disk is powered on and properly cabled.

I91 (65, 83) Disk Hardware Error

Generally, this error means that the DPU cannot communicate with the disk drive.

Recovery:

1. Retry the disk operation.
2. Make sure the disk is powered on, properly cabled, and ready (in RUN mode).

## I92 (61) I/O Timeout

The DPU did not respond to the 2200 within the expected time period.

## Recovery:

1. Retry the disk operation.

## I93 (67) Disk Format Error

This error indicates that the DPU could not locate the specified sector on the disk platter. The error may result from the sector address having been improperly recorded, perhaps due to a media defect. If the error occurs when accessing a platter formatted on a different drive, one of the drives may not be properly aligned.

## Recovery:

1. Retry the disk operation.
2. Reformat the platter. Caution, all data will be erased.
  - a. If the removable cartridge or diskette does not format properly, it should not be used.
  - b. If the fixed media does not format properly, Wang service should be notified.
3. Certain disk units permit the sector control information of a particular sector to be rewritten (see following pages). The data in the sector is not rewritten so that it may be possible to recover the data by reformatting a sector that causes a format error. However, the format sector operation should be considered as a last resort measure. There is no guarantee that the data will be preserved.

## I94 (67) Format Key Engaged

## Recovery:

Turn off format key.

## I95 (71) Device Error

This error generally indicates a fault at the disk drive itself. Some disk units have a fault light that may be illuminated. Some disks also require the fault to be manually reset.

## Recovery:

1. If writing, make sure the platter is not protected.
2. Reset the fault if the disk has a fault switch.
3. Retry the disk operation.

## I96 (72) Data Error (CRC or ECC)

This error usually indicates that the sector being accessed was improperly recorded, perhaps due to a media defect. If the error occurs when accessing a platter formatted on a different drive, one of the drives may not be properly aligned.

## Recovery:

1. Retry the disk operation.
2. Certain disk units (see the following pages) support a read bad sector command. The data read (even though in error) will be transferred to the 2200. It is then the user's responsibility to verify the data and correct bad information. Once corrected, the data can be written. This operation should be considered as a last resort measure since there will most likely be errors in the data.

## I97 (68) LRC Data Error

This error usually indicates an error in the transmission of the data between the 2200 and the DPU.

## Recovery:

1. Retry the disk operation.
2. Make sure the disk is properly cabled to the 2200.

## I98 (64) Illegal Sector Address or Platter not Mounted

The disk sector being accessed is not on the disk, or the disk platter has not been mounted.

## Recovery:

1. Correct the program if it is attempting to access a sector beyond the maximum legal limit for the platter.
2. Make sure the platter has been properly mounted on the drive.
3. Make sure the correct drive is being accessed.

## I99 (85) Read After Write Error

Reading the sector after it was written indicates that the data was not properly written.

## Recovery:

1. Retry the operation.
2. Make sure the platter has not been write protected.

## C. Suggested \$GIO Sequences for 2260C/BC Disk Recovery Commands

The use of \$GIO for controlling the disk is not generally recommended. The recovery measures below should only be considered to be last resort methods for recovering data. Wang in no way guarantee that data will always be able to be recovered or that additional data will not be lost. It is the user's responsibility to verify any data recovered using the measures below since in most cases some of the data will be incorrect. It is advised that a backup of the platter be made before attempting any recovery operations. Furthermore, it is recommended that these operations be incorporated into a standalone utility, rather than including them in application programs.

The following list of \$GIO sequences uses the following parameters.

error return = HEX(abcdef)

where:

abcdef = 000000 if no errors

abcdef = 000004 if echo error, should retry command.

ab = 01 if illegal sector address (ERR I98)  
 02 if disk hardware error (ERR I91)  
 04 if format key engaged (ERR I94)

cd = 01 if device error (ERR I95)  
 02 if format error (ERR I93)  
 04 if data error (ERR I96)

/xyz = disk address (e.g., /320, /360)

variables used: DIM G\$10, G\$(4) 64

1. Read Bad Sector (also available on later versions of the 2270A).

Entry parameters:

STR(G\$,1,1) = HEX(00) if fixed platter  
 HEX(10) if removable platter

STR(G\$,2,2) = binary sector address

\$GIO /xyz (0600 0700 70A0 68C0 7040 6A10 6A20 6230 8705 1704  
 1156 1576 4270 8367 C640 8605, G\$) G\$()

Return Parameters:

STR(G\$,6,3) = error return

STR(G\$,5,1) = LRC

G\$() = data (256 bytes)

2. Format Sector

Entry parameters:

STR(G\$,1,1) = HEX(20) if fixed platter  
                  HEX(30) if removable platter

STR(G\$,2,2) = binary sector address

\$GIO /xyz (0600 0700 70A0 68C0 7040 6A10 6803 6800 6A20 6230  
          8705 1704 1156 1576 4000 8B67, G\$)

Return Parameters:

STR(G\$,6,3) = error return



\*\*\* MEMO \*\*\*

To: Lee Collett

From: Roger Droz

Date: August 11, 1982

Subject: 2200 Math Package Precision  
Ref: Mr. Zeskind's Letter

I have composed an answer to Mr. Zeskind's letter. I am finding it easier these days to write in first person, so I have composed this letter as I would if I were discussing this matter in person. I have not mailed this letter to Mr. Zeskind. You may decide whether to re-write the letter, mail it as is, or even claim credit for it. (Three years ago, Dave Angel was John Neuman's ghost writer to answer a similar question.) I will be happy to supply the letter in machine readable form.

I am not getting much time to write articles for the user society newsletter. This is an excellent subject. Perhaps you can compose a suitable article, based on my letter and the Science News article supplied by Mr. Zeskind.

Roger I. Droz  
2200 Microcode Group  
MS 1383  
August 24, 1982

Jeffrey A. Zeskind  
c/o Design & Administration Corp.  
Miami Shore, Florida 33153-0563

Dear Mr. Zeskind:

The problems associated with round-off errors in finite precision arithmetic are better known to engineers and scientists than they are to business people. The subject should be discussed much more than it is, not to get people worried about it, but simply to educate them about those rare situations where unexpected results can occur. As the Science News article points out on page 73, "calculators and computers work with numbers having a definite number of digits and the results of calculations must be rounded or chopped off, introducing unavoidable error." Mr. Kahan is overly optimistic when he implies that it is possible to build a computer that never delivers misleading answers, but we certainly have made an effort to minimize the problems on the Wang 2200. I am happy to see that you are interested in the subject of arithmetic errors, and will spend a few pages discussing our strategy on the Wang 2200.

The Wang 2200 performs arithmetic in a straight-forward manner, which is described below. We have had few problems with the microcode for arithmetic operations, but we appreciate concerned users bringing problems to our attention. The specific problem you present in your letter is not serious in most applications. Most users do not require the full 13 digit accuracy available in 2200 BASIC, and are in fact sometimes introducing the type subtle errors discussed in the Science News article by rounding or truncating at bad times while attempting to minimize memory requirements. These errors are typically much more serious than arithmetic errors in the 2200 arithmetic microcode. After describing how the 2200 math package works, I shall discuss programming techniques to avoid accumulating errors.

#### 2200 Floating Point Arithmetic

All Wang 2200s do arithmetic in binary coded decimal floating point. The mantissa is 13 digits. The exponent is 2 digits. BASIC-2 and BASIC-3 offered on the 2200VP, 2200MVP, and 2200LVP employ guard digits and other techniques to improve both speed and accuracy over the 2200T.

Using base 10 arithmetic has the advantage of doing strange things when people expect it to. For instance,  $(1/3)*3$  is .999999999999, but  $(1/10)*10$  is 1.0. On many computers that use binary floating point (such as the Wang VS running BASIC),  $(1/10)*10$  is .99999999. This is not to say that the 2200 is more correct. The 2200 simply comes closer to living up to the expectations of a base 10 oriented world. Proponents of base 2 or base 16 floating point representations argue that using a number base that is a multiple of 2 is more storage efficient, and is faster in many cases. (The Wang 2200 hardware is a counter example to the speed argument, being equally adept at binary and decimal operations.) It turns out that there is no good solution. Neither the 2200 nor the VS are very good at  $(1/7)*7$ .

There are two common solutions to the problem: guard digits and "fuzz". Often, both solutions are employed. Guard digits are fairly familiar. More digits are stored than are displayed. The displayed result uses the guard digits to round the displayed answer.

Fuzz solves the problem that 1 and  $(1/3)*3$  should be equal. Systems with fuzz often allow the user to specify how close to equal is close enough to be called equal. For instance, on the 2200T 0 and  $((1/3)*3)-1$  are equal within a fuzz (plus or minus) of  $1E-12$ . In BASIC-2 (supported on the 2200VP, 2200MVP, 2200LVP),  $((1/3)*3)-1$  is  $1E-13$ . In the absence of user settable fuzz, the programmer is left with the rule "there is no such thing as equal in floating point". This rule serves the engineer or scientist quite well. Commercial users are either perplexed or ignore the problem. The solution to the problem of almost equal on the 2200 involves rounding results to fewer than 13 digits before comparing them, or checking a range instead of testing for strict equality. More on this later.

For the most part, the 2200 philosophy is not to keep invisible guard digits. The goal is to return the answer of each addition, subtraction, multiplication, division, or built-in function to 13 digit accuracy. The user concerned about arithmetic errors should use the last few digits as guard digits and round answers himself before displaying. BASIC-2 and BASIC-3 provide the ROUND built-in function to aid in this process. Later on, I shall discuss when the user should be concerned about possible arithmetic errors.

The 2200 never automatically rounds results to fewer than 13 digits. For instance,

```
10 PRINT USING 11, 1.59
11% +#. #
```

gives: +1.5

Similarly,

A(1.9) is interpreted as A(INT(1.9)), or A(1)

Wang BASIC (supported on the 2200T) makes no use of guard digits. This explains why  $1/3*3-1$  and  $1/3*3-.5-.5$  give different results. BASIC-2 and BASIC-3 calculate 1 guard digit, so  $1/3*3-1$  and  $1/3*3-.5-.5$  give the same result. The primary purpose for the guard digit is to support the optional rounding of answers to 13 digits. By default, BASIC-2 and BASIC-3 round the result of each computation to 13 digits, such that  $2/3$  is .6666666666667. Optionally, the user may SELECT NO ROUND, and  $2/3$  then gives .6666666666666.

Three examples:

.999 - 1, using no guard digit  
express problem in floating point:  $+9.99 \text{ E-1} - +1.00 \text{ E+0}$   
match exponents:  $+0.99 \text{ E+0} - +1.00 \text{ E+0} = -0.01 \text{ E0}$   
normalize result:  $-1.00 \text{ E-2}$  or  $-.01$

.999 -.5 -.5, using no guard digit  
express problem in floating point:  $+9.99 \text{ E-1} - +5.00 \text{ E-1} = +4.99 \text{ E-1}$   
perform second subtraction:  $+4.99 \text{ E-1} - +5.00 \text{ E-1} = -0.01 \text{ E-1}$   
normalize result:  $-1.00 \text{ E-3}$  or  $-.001$

.999 - 1, using 1 guard digit  
express problem in floating point:  $+9.990 \text{ E-1} - +1.000 \text{ E+0}$   
match exponents:  $+0.999 \text{ E-0} - +1.000 \text{ E+0} = -0.001 \text{ E+0}$   
normalize result; truncate to 3 digits:  $-1.00 \text{ E-3}$ , or  $-.001$

BASIC-2 and BASIC-3 use even more guard digits when computing trig functions. The trig functions on the 2200T may be considered accurate to 10 digits; by using several guard digits, BASIC-2 trig functions are accurate plus or minus 1 in the 13th digit.

Intermediate results in expressions are stored to 13 digits. The order of expression evaluation can thus be significant. For instance  $(4/3) - (2/3)$  is different than  $(4-2)/3$ . If intermediate results during expression evaluation were stored with a guard digit, the two answers would be the same, but if the calculation was spread out over several statements, the discrepancy would reappear. You can't win!

Many computers use logarithms to perform exponentiation. BASIC-2 and BASIC-3 use multiplication to raise a number to an integer power, for reasons of speed and accuracy. The square of an integer is always an integer in BASIC-2. This is not the case on computers that always use logarithms for exponentiation. Even on the 2200, inaccuracies in performing LOG and anti-LOG calculations cause 2 to the 4th power to give a different answer than 2 to the 3.5 times 2 to the .5.

## When to Worry About Arithmetic Errors

The Wang 2200 math package behaves as expected for most common business calculations. As stated earlier, this is because most business applications do not require the full 13 digit precision of 2200 BASIC. The extra digits serve as guard digits, protecting the user from subtle accumulating errors.

Most errors occur during division and subtraction. Examples such as  $(1/7)*7$  and  $1-(1/3)*3$  have already been given. Division and subtraction both end up with fewer digits in the answer than were in the operands. (Consider  $19-11$  and  $25/5$ .) Subtracting two numbers that are very nearly equal generally introduces considerable error:

$5280.637005678 - 5280.000000002$  gives  $.6370056760000$

Both operands are accurate to 13 digits, but the result is only accurate to 9 digits.

A Civil Engineer once tried to use the Wang 2200 to account for the error that the curvature of the earth introduced into his measurement of the length of a table. He rearranged the expression several ways and got several radically different answers, primarily because the calculation always involved subtracting two numbers that were almost equal. Each different formula came up with a different "almost zero" (sometimes called a "dirty zero") that was subsequently multiplied and divided by several large numbers. The final answers differed by several orders of magnitude, because the "dirty zeros" differed by several orders of magnitude.

Subtraction does not usually present a problem in business applications, because the numbers involved are either integers (quantities) or fixed point (dollars and cents).

Multiplication and division can present problems in business applications if intermediate results exceed 13 digits.  $1234567890123 * 11 / 2$  is not the same as  $1234567890123 / 2 * 11$  because in the first case, the intermediate result exceeds 13 digits.

Division often results in fractions that cannot be represented exactly in a finite number of base 10 digits. Division by 3 and 7 commonly produce this problem.

## What to Do About Arithmetic Errors

In scientific applications, the typical technique is to keep numbers to the highest precision available on the machine and round off just before displaying. Engineers and scientists are sometimes misled by the problem of "dirty zeroes" mentioned above. They also learn to only trust 11 or 12 digits of the answer because of the "truncation of intermediate results" problem. (The theorems of numerical analysis are sometimes used to calculate the amount of error that may be present, but experimental error in measuring the input data usually dominate computational errors.)

In business, results are generally rounded sooner, to avoid dealing in fractions of a cent. Consistent rounding rules are important in business, but somehow the problems of fractions of a cent are hard to impress upon people until some strange result occurs:

It used to be fairly common to see bank statements and payroll check stubs where the printed total was one cent higher or lower than the sum of the printed figures. The problem is that  $\text{ROUND}(A,2) + \text{ROUND}(B,2)$  is not equal to  $\text{ROUND}(A+B,2)$ . ( $\text{ROUND}(\text{expression},2)$  is the BASIC-2 built-in function that rounds a result to 2 decimal places.)

Sometimes programmers put in a quick fix to make the check stub balance, but forget that rounding problems effect other calculations later in the program: Social security withholding results in an answer in terms of fractions of a cent. (Gross pay time 8.6%, last time I checked.) In order to get the paycheck stub to balance, the result is rounded to the nearest cent. Later, when it comes time to pay Uncle Sam, many people are tempted to multiply the total gross pay for all employees by 8.6%. This answer is virtually always less than the sum of the social security withholding items on the paycheck stubs. This inconsistent rounding scheme is to the employer's advantage, as long as he doesn't drive his accountant crazy looking for an error that turns out to be hidden in the undisplayed fractions of a cent.

### Rounding Functions for the 2200T

BASIC-2 and BASIC-3 have a built-in ROUND function that can be used to round results to fewer than 13 digits. On the 2200T, the ROUND function may be simulated using the following user-defined function:

```
100 REM ROUND result to two decimal places.  
200 DEF FNR(X)=SGN(X) * INT((ABS(100 * X)+ .5)/100)
```

The above function rounds -5.236 to -5.24; some people prefer to round -5.236 to -5.23, which is accomplished by:

```
200 DEF FNS(X)=INT((100 * X + .5)/100)
```

These functions are convenient to use, because the rounding algorithm need be coded only once in the program. Any expression may be rounded by coding something like:

```
1000 Q = FNR(A+B*C)
```

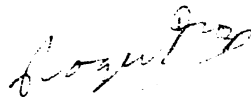
## Conclusion

To summarize, arithmetic errors are inevitable in finite precision computer arithmetic. Errors can be avoided by using integer or fixed point operands with fewer than the maximum number of digits allowed on the machine and avoiding division by numbers like 3 and 7, that produce fractions that cannot be expressed in a finite number of digits. It is easier, however, to use suitable rounding conventions and consider at least 1 of the 13 digits the 2200 calculates to be a guard digit. On the 2200T, the above user-defined rounding functions are almost as convenient to use as the ROUND function built into BASIC-2 and BASIC-3.

I am glad that your letter gave me the excuse I needed to write all of this down. As I said earlier, problems of arithmetic errors are not discussed often enough. Though we have improved the arithmetic on later Wang 2200s, the members of the 2200 microcode group believe that the arithmetic available on the 2200T is accurate enough for most any application, given a little knowledge of the problems associated with finite precision arithmetic.

During the last year or so, there has been new interest in computers solving problems algebraically, instead of numerically. This allows solutions in terms of rational numbers, which completely solves the problem of  $(1/3)*3$ . The December 1981 issue of Scientific American has an excellent article on computer algebra. I have not heard of an algebra program for the 2200, but I hope somebody writes one some day.

Sincerely,



Roger L. Droz

M E M O R A N D U M

TO: 2200 Support

FROM: B. Ciolek  
J. Cizmadia

DATE: March 2, 1983

SUBJ: 2200 Disk Error Meeting

---

The meeting on 3/1/83 was conducted by Jerry Sevigny who discussed disk errors and recovery procedures. The information should help us differentiate between some solid hardware errors and hardware/software errors.

The following is a copy of a handout that summarized most of the error codes discussed. This can be used with any BASIC-2 error code manual as well as the following helpful hints.

I94 refers to 2230 or 2260 disk with format key

I99 this is a hardware error that can be caused when a DATASAVE statement includes the "\$" option which causes a read-after-write.

X74 May indicate a hardware error on the 2280 caused by the loss of a bit when storing information in the upper portion of memory.

D88 indicates the possibility that the hardware dropped a bit

Example: When seeking a numeric variable type yet finding an alphanumeric variable type. This is just a possibility, usually the error indicates a programming problem.

Rumor Corner:

1. A 5 1/4 DSDD for the MVP is forthcoming.
2. If PEDM errors are found in the "C" chassis, have the CE check to make sure both 32K memory boards are present. If not, there may be too much noise causing the error to occur.



**Buildalt (cont.)**

With BUILDALT 5.03.08, however, the following happens: A level 2 alternate key record is *not* equal to the highest alternate key value in a level 1 block. This problem is most likely to be noticed after a COPY (with REORG = YES) has been performed against the file. Files with a large number of records (typically in the tens of thousands) or files with a relatively large number of records (thousands or high hundreds) *plus* a large alternate key size are especially susceptible.

A corrected version of BUILDALT (5.03.09) will be distributed to Customer Engineering ATOMs with VS operating system maintenance releases 5.03.52 and 5.03.70.

**VERIFY 5.03.03**

VERIFY version 5.03.03, distributed with OS 5.03.50, contains a bug which, under certain conditions, causes a file to be reported as damaged when in fact there is nothing wrong with the file. A corrected version of VERIFY (5.03.04) will be distributed to Customer Engineering ATOMs with VS operating system maintenance releases 5.03.52 and 5.03.70.

The corrected versions of DATENTRY (4.02.04), BUILDALT (5.03.09) and VERIFY (5.03.04) should replace the existing modules in library @SYSTEM@ on operating system 5.03 install packages and should be installed on any customer systems running a 5.03 operating system release and experiencing the problems described above.

## **2200 SYSTEM ARCHITECTURE**

by 2200 R&D

Wang 2200 computer systems employ a direct-execution, high-level-language (HLL) architecture. With direct-execution HLL systems, the HLL is effectively the machine language of the computer. Unlike more conventional architectures, where the source code is transformed into a distinct object code before processing, the direct execution system processes the source code directly.

The direct execution system provides a number of advantages over more traditional architectures, not the least of which is its conceptual simplicity. The more conventional layers of software including assemblers, linkage editors, compilers and loaders are eliminated. The inherent conversational nature of the system facilitates programming and debugging. The debug run and the execution run are identical. Error messages can easily include a listing of the actual source code. Program execution can be halted, single-stepped, and restarted. Since there is no compilation phase, the system responds immediately to program entries and modifications. Programmers can understand the language semantics by observing the direct response of the system.

The 2200 provides the user with a single high-level language, BASIC-2, which is used for all programming. Proficiency in system use is easily achieved since there is only one language to learn. A fundamental design criterion in the development of BASIC-2 was to provide a self-sufficient language that would be as flexible as conventional general-purpose computer instruction sets. I/O and data handling language extensions provide the user with flexibility not usually found in a high-level-language.

The 2200 is not a pure direct-execution machine since the source code is preprocessed into a form which conserves memory and is more efficiently interpreted. However, source and object differences are such that the preprocessor transformation is nearly completely reversible. As a result, only the condensed code is stored in the machine. The preprocessing function eliminates gross inefficiencies in memory, timing, and logic requirements.

(continued)

## 2200 Hardware

2200 computers consist of a microprogrammed MSI (medium scale integrated circuit) processor coupled with a number of special purpose LSI (large scale integrated circuit) I/O processors and controllers. The operating system and language interpreter reside in a large computer storage memory which is independent from user data memory; this microprogram directs the execution of the CPU and coordinates communication with the I/O processors. The independent I/O processors permit the overlap of the CPU and I/O processing. The CPU is relieved of the responsibility for controlling peripherals that would otherwise require frequent or dedicated CPU attention.

The 2200 CPU is a pseudo 16-bit processor using a 3-bus architecture for interconnecting a bank of general-purpose, status, and I/O 8-bit registers and the ALU (arithmetic and logical unit). A microinstruction can address these registers as double, single, or half registers for performing 16, 8 or 4-bit operations. In addition, a bank provides quick access to major system pointers. The extensive microinstruction set consisting of 24-bit words provides decimal and binary arithmetic, logical operations, and a wide variety of conditional branching instructions.

In a single CPU cycle, a 24-bit microinstruction can be fetched, 16-bits of data memory can be fetched, and a 16-bit operation can be performed. The wide memory path, 600 nanosecond cycle time, and rich microinstruction set provides a highly effective processor for implementing direct-execution languages.

User programs and system controllers are kept in data memory, of which 512K can be installed. Since the CPU's address space is limited to 64K, however, data memory is divided into 64K banks. In order to provide the microprogram with access to control tables without switching memory banks, the lower 8K of the address space always refers to bank 1. The lower 8K of banks 2, 3, 4, 5, 6, 7, and 8 is not used.

## Multi-User Operating System

The 2200 multi-user operating system allows several users to share a single computer effectively. To accomplish this, the operating system divides the resources of the computer—memory, peripherals, and CPU time—among the users. Once each user has been allocated a share of the computer resources, the operating system acts as a monitor, allowing each user to utilize the system in turn, while preventing the various users from interfering with each other's computations.

The multi-user operating system employs a fixed partition memory scheme. User memory is divided into a number of sections or "partitions", each of which can store a separate program. From the user's point of view, each partition functions independently from the other partitions in the system. Each user may LOAD and RUN BASIC software, compose a program, or perform Immediate Mode operations. As in a single-user environment, the user has complete control over his or her partition. No user on the system may halt execution in, or change the program text of, a partition controlled by another user.

Each terminal may control several partitions executing independent jobs. At any given time, however, only one of these partitions is in control of the terminal and thus capable of interacting with the operator. The partition in control of the terminal is said to be in the "foreground." Other partitions assigned to the terminal may continue to execute in the "background" until operator intervention becomes necessary.

(continued)

Although partitions function independently of one another in general, there are situations in which it is useful for two or more partitions to cooperate. Cooperating partitions may share program text and/or data. The sharing of commonly used programs and data by several partitions eliminates needless duplication and produces more efficient use of available memory. The integrity and independence of a partition which contains shared programs or data is maintained by requiring the partition to explicitly declare itself to be global (sharable) before it can be accessed by other partitions. Correspondingly, a partition wishing to access shared text or data in a global partition must identify the desired global partition.

To the programmer who regards the multi-user operating system as a whole, it appears that all partitions are executing simultaneously. Because all partitions share a single CPU, however, only one partition can be executing at any given moment. The operating system creates the illusion of simultaneous execution of several programs by rapidly switching from one to the other in turn.

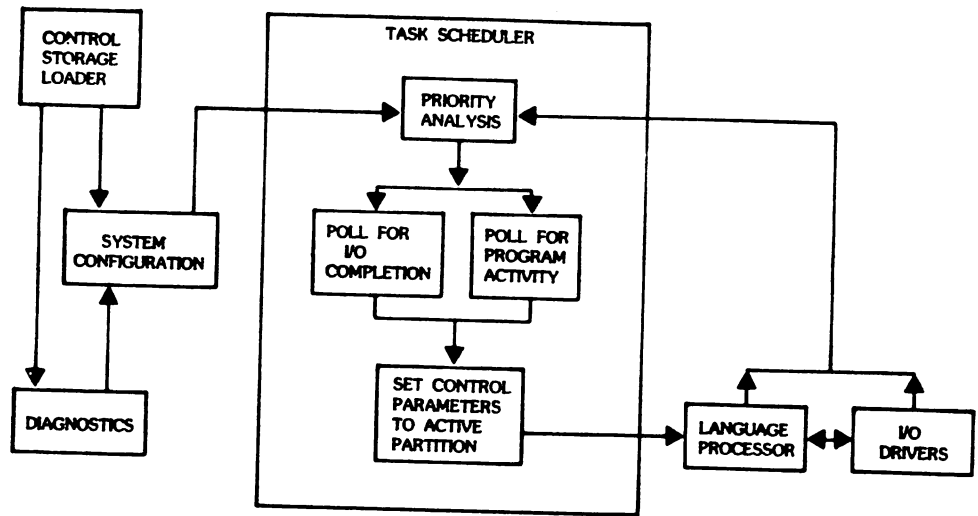


FIGURE 1. BLOCK DIAGRAM OF 2200 MULTI-USER OPERATING SYSTEM

Partitions in the 2200 multi-user operating system are serviced by the CPU in a "round-robin" fashion, with priority given to I/O operations. Each partition in turn is given a "timeslice" 30 milliseconds (ms) in duration. The CPU has a 30 ms timer which is set at the beginning of the timeslice; at the completion of each BASIC statement (and at various points in the middle of long statements and I/O operations), the clock is checked to see whether the 30 ms timeslice has been exhausted. When a partition's timeslice has expired, the operating system saves the status of that partition so that it may be restored later when that partition's turn comes around again. The operating system then loads the status of the next partition in line and begins its 30 ms timeslice. The process of halting execution of a partition at the end of its timeslice is called a "breakpoint."

(continued)

Timeslices do not always last exactly 30 ms. Unlike many operating systems, the 2200 multi-user system switches users (breakpoints) whenever it is convenient rather than strictly by the clock. This technique reduces the amount of status information that must be saved, giving the 2200 multi-user system low operating system overhead when compared with most other multi-user systems. More importantly, breakpoints may occur in the middle of BASIC I/O statements. If, for instance, the current partition attempts a disk access and the disk is hogged by another partition, this condition is quickly detected and breakpoint occurs. I/O breakpoints differ from program breakpoints in that the partition is specifically marked as "waiting for I/O". When the partition's turn comes around again, the system takes only a few microseconds to decide whether processing may proceed or whether the partition is still waiting for the I/O device and may be bypassed. Thus, if a printer goes "busy" while it performs some mechanical function or if a partition that does not currently control the terminal attempts to write to the CRT, the system bypasses that partition almost as effectively as if it were removed from the system until the I/O device becomes available.

## OIS BASIC PROBLEMS AND SOLUTIONS

by the Technical Support Center

Below are some common questions relating to OIS BASIC that are received from the field. The solutions to the problems do not necessarily show the best or only way of solving the given questions, but they will illustrate some BASIC statements that are not often used. The reader should also try the programs to see how they work.

**"I would like to right justify my output to the screen. How can I do this?"**

BASIC will print to the screen left justified starting at a certain column on the screen, so if a column is printed on the screen of eight character fields it might look like the following.

```
abcdefgh
ab
abcd
```

What the programmer must do to achieve right justification is to put the trailing spaces in each of the fields before the letters. For example, the field "abcd" is really "abcd space space space space". To right justify this field, put the four "spaces" before the "abcd". The following program will do this. It is a general solution which will work no matter what the dimensioned length of the variable a\$ is.

```
0010 dim a$8
0020 a$ = "abcde"
0030 for x=1 to len (a$)
0040 a$ = rotate c (a$,8)
0050 next x
0060 print a$
```

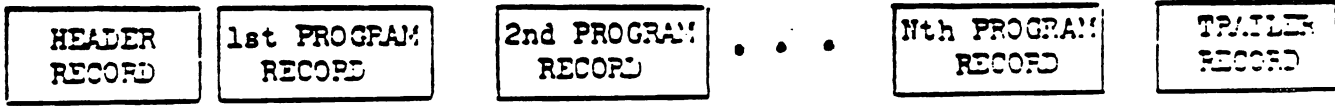
(continued)

### I. 2200 CASSETTE/DISK FILE FORMAT

The 2200 provides the capability to record both programs and data onto cassette and disk. Both programs and data are recorded in 256 byte physical records. To insure data integrity, each physical record is recorded twice on tape. Dual recording and read-back is done automatically by the system, and requires no special user considerations.

#### PROGRAM FILES:

When programs are recorded, it is not sufficient to merely record the program lines and nothing else. It is important for the 2200 system to tell where the beginning and ending records of a program are. Therefore, everytime a program is recorded, the 2200 system automatically records a header record before the program, and a trailer record after the program. Each recorded program thus becomes a program file. The figure below illustrates a program file.



#### HEADER RECORD:

This is a physical record (256 bytes) which contains a control byte identifying it as the header (or beginning record) of a program. It also contains 8 bytes which can be used to store the name of the program, if the program is named when saved. The remainder of the record is not used.

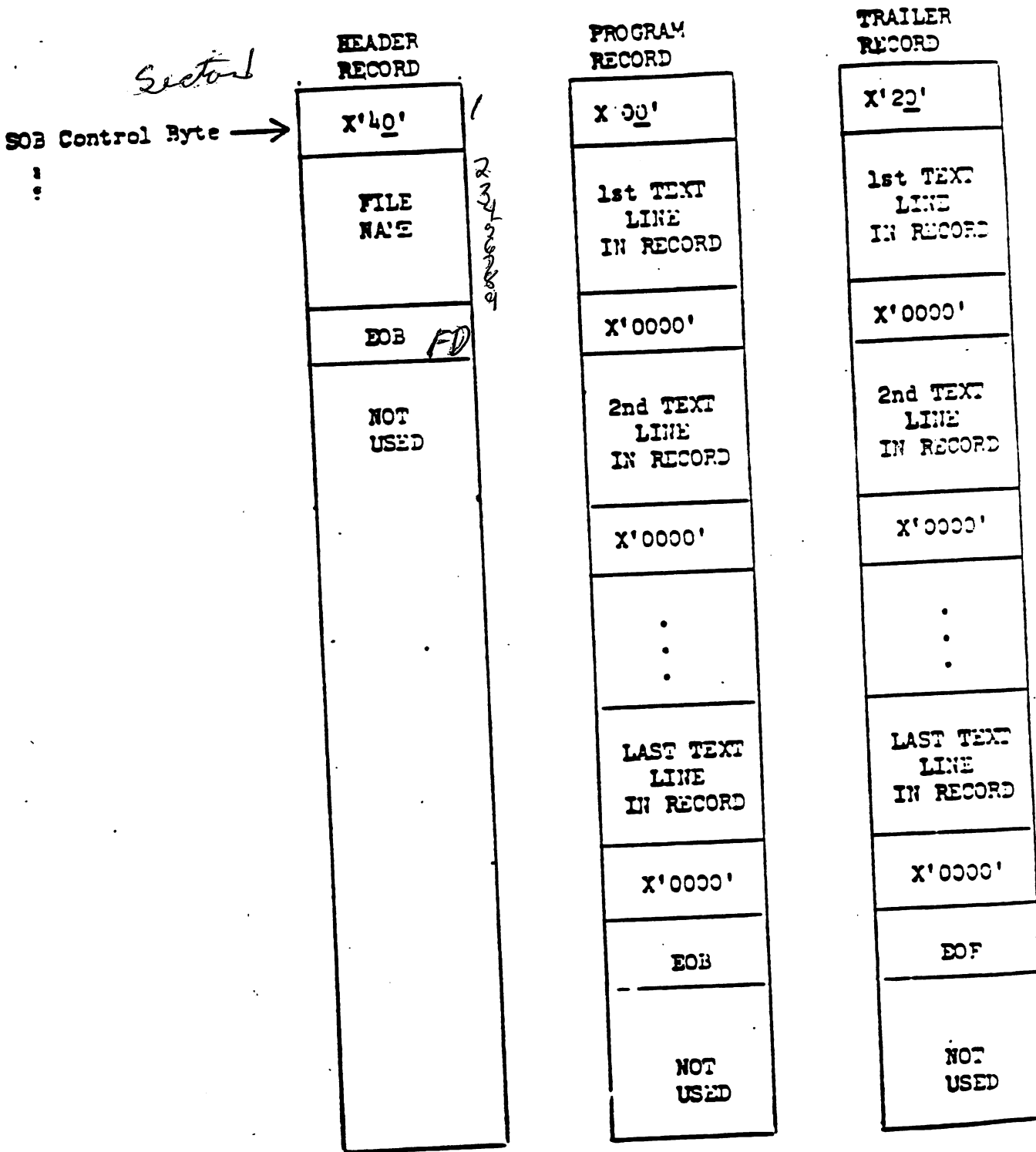
#### PROGRAM RECORD:

Each program record is a 256 byte physical record containing a portion of the saved program. It also contains a control byte identifying it as a physical record which contains part of a program, (i.e., a Program Record).

#### TRAILER RECORD:

The trailer record is similar to a program record except that it has a control byte identifying it as the final physical record of the current program file. (i.e., the Trailer Record).

PROGRAM FILE FORMAT



For protected programs, SOB control bytes are X'50', X'10', and X'30' respectively.

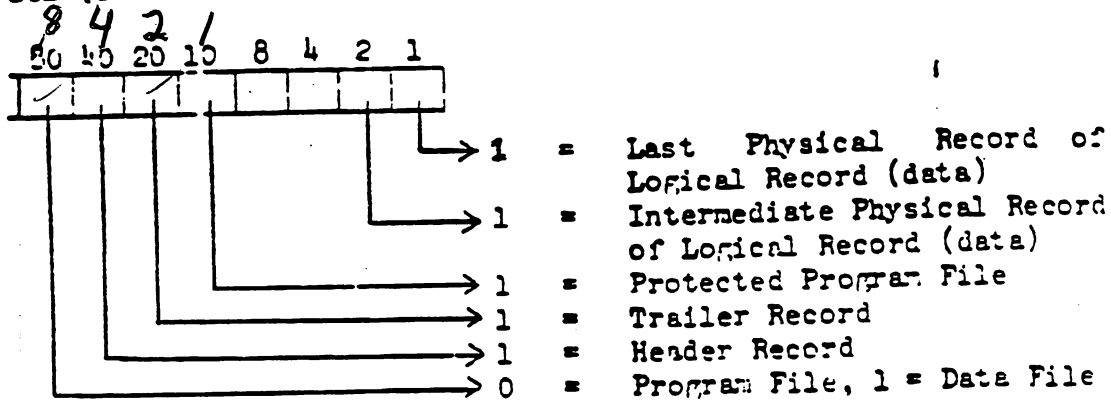
EOB (end-of-block) = X'FD'

EOF (end-of-file) = X'FE'

3/469

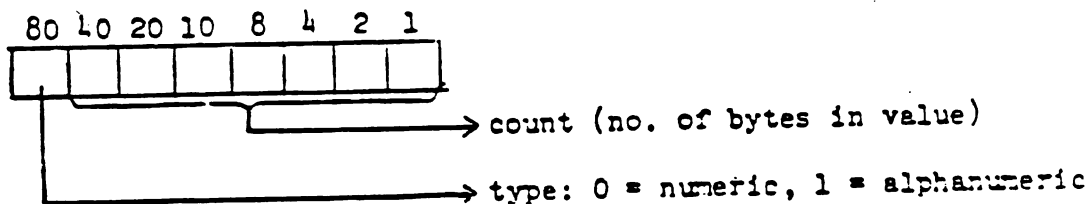
CONTROL BYTES:

1. SOB (start of block) - identifies type of record



2. PRN (physical record number) - Data (nonheader and nontrailer) records have a second control byte specifying the physical record number within the logical record.

3. SOV (start of value) - precedes each value in a data record.



4. EOB (end of block) - indicates the end of valid data in a physical block.

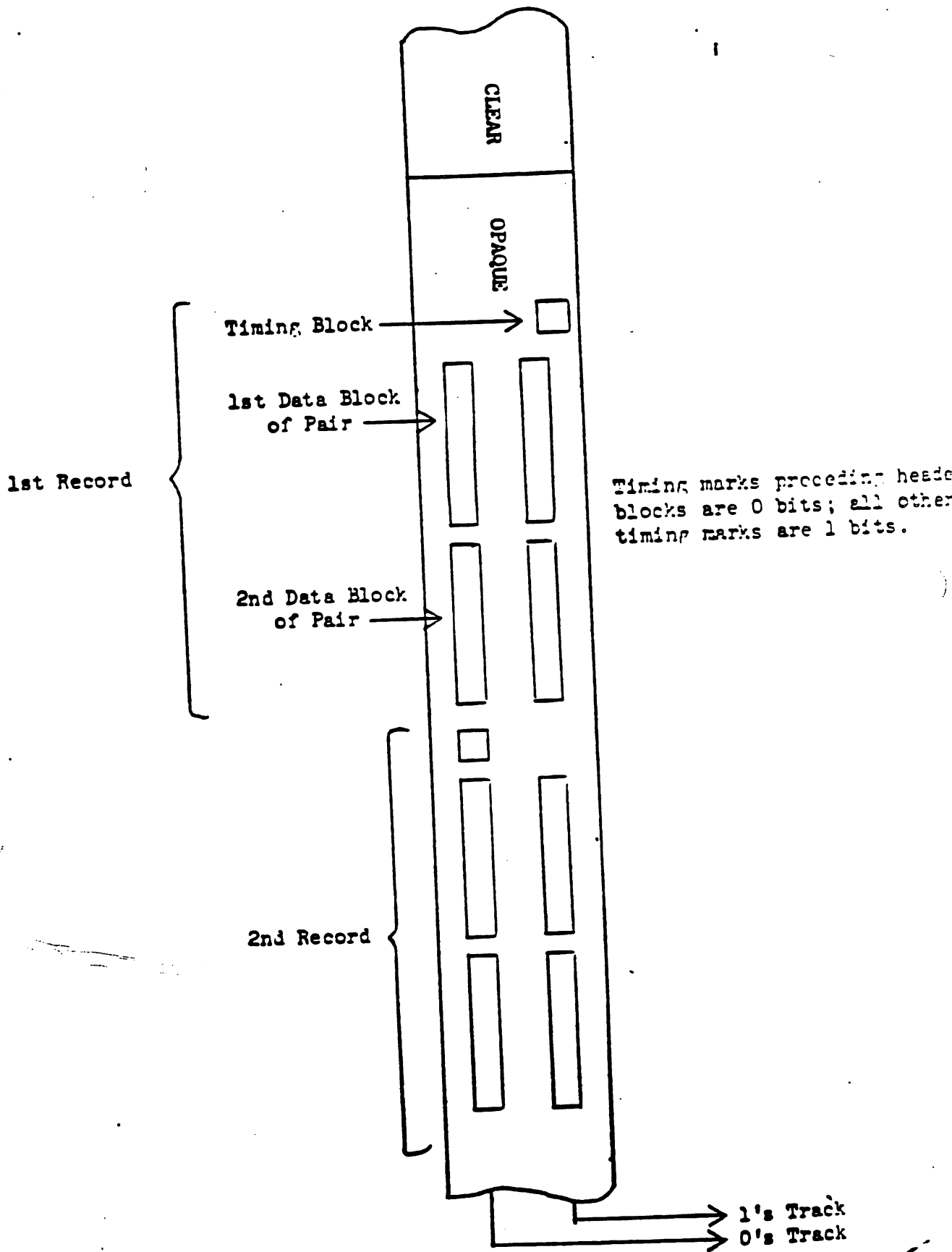
EOB = X'FD'

5. EOF (end of file) - indicates the end of valid data in a program trailer block.

EOF = X'FE'

PGM TEXT

2200 PHYSICAL TAPE FORMAT





DATA FILE FORMAT

*sector*

1

X'CO'
FILE NAME
EOB
NOT USED

2

X'80'
PRN
SOV
1st DATA VALUE
.
.
.
SOV
LAST DATA VALUE
EOB
NOT USED

3

X'A0'
NOT USED

Note, when values of alphanumeric variables are stored, the value is filled in with spaces on the right up to the maximum length of the variable.

EOB = X'FD'

*End of significant data*

## II. 2200 DISK INDEX STRUCTURE

The disk catalogued area is set up by the user with a SCRATCH DISK statement. The catalogued area of each disk platter is divided two sections; (1) an index which keeps track of where each file is currently stored and (2) the file area where the files are actually written.

### The INDEX:

The index can vary from 1 to 255 sectors. The index is always stored on the first sectors of each platter. Each sector of the index can hold 16 file entries except the first sector which can hold 15. The first sector uses the first file entry to store the index size, the current end of the catalogue area, and the upper limit of the catalogue area.

The layout of the catalogue index is given below:

		SECTOR 0							
ENTRY 1	}	Z0	XX	AA	AA	BB	BB	NU	NU
		NU	NU	NU	NU	NU	NU	NU	NU
ENTRY 2	}	US	TO	CC	CC	DD	DD	NU	NU
		8 BYTES OF FILE NAME							

Entry 1 of Sector 0 has information for the entire catalogued area.

Z0XX

- = Number of sectors in the index  
 Z = 0 if on the fixed platter.  
 Z = 8 if on the removable platter.  
 XX = binary number from 1 to 255.

AAAA

- = The next sector in the catalogued area which is available for storing program or data files. The high order bit is 0 if on the fixed platter; 1, if on the removable platter.

BBBB

- = The first sector beyond the catalogued area. Again, the high order bit is 0, if on the fixed platter; 1, if on the removable platter.

NU

- = The rest of the 10 bytes is unused and should contain 0.

Entry 2 of Sector 0 is representative of every other entry in the index.

US	=	00 Free or unused entry.
US	=	10 File currently stored.
US	=	11 File currently stored here is scratched. Area is still reserved but is not being utilized.
US	=	21 The index entry has been used by a file, but currently has no information stored here. This status exists when a scratched file's area is being used by a new file.
TO	=	00 Data file.
TO	=	80 Program file.
CCCC	=	Sector address where <del>last</del> <sup>FIRST</sup> record of file is stored in catalogued area.
DDDD	=	Sector address where last record of file is stored in catalogued area.

This last sector of each file is set up as a dummy trailer record with the second and third byte of the record used to specify the number of sectors currently used by the file. This value stored there is equal to (SECTOR ADDR+1 of TRAILER RECORD) - (BEGINNING SECTOR ADDR)+1. With DATA files it is initially set to 1.

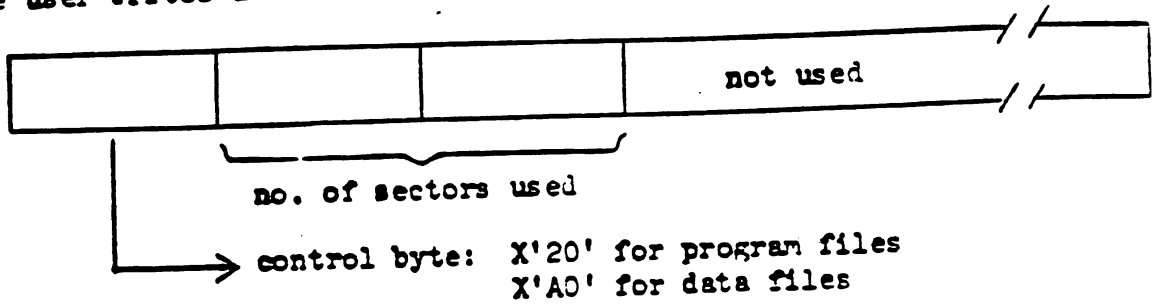
The last 8 bytes of the entry is the ASCII codes for the name of the file.

#### HASHING:

The entries to the index are selected by using a HASH function. The HASH is calculated from the 8 byte program name. The Exclusive OR of the 8 bytes is calculated (S).  $3 * S$  equals a 16 bit binary value W X Y Z.  $X + Z$  is set to Z and  $W + Y$  is set to Y. The hash Y Z is then reduced to a number less than the number of index sectors. This value is set equal to the index sector where the file is going to be stored. If there is no room at this sector (i.e., all 16 entries are filled), the value is decremented by 1 and that sector is searched for a free entry. The process continues until a free position has been found or all the sectors have been scanned; in which case an error is given.

### III. DISK SYSTEM EOF RECORD

When a file is created on disk under DC mode, the last sector of the file is reserved for system use. The system creates an artificial EOF record which includes a count of the number of sectors actually used in the file. Note, the number of sectors used is only updated when the user writes a trailer record in the file.



MAP OF DISK PLATTER

Sector 0

00AA    BBBB    CCCC

Index Area  
up to 255 sectors

Sector AA

Currently Used Catalogue Area where  
programs and data are stored

BBBB

Unused Catalogue Area  
(Next file will be  
stored at BBBB)

CCCC

Free Area  
(Used by TEMP Files  
or any DA instructions)

ASCII CHARACTER SET

00	-	NUL	20	-	SPACE	40	-	@	60	-	'
01	-	SOH	21	-	!	41	-	A	61	-	a
02	-	STX	22	-	"	42	-	B	62	-	b
03	-	ETX	23	-	#	43	-	C	63	-	c
04	-	EDT	24	-	\$	44	-	D	64	-	d
05	-	ENG	25	-	%	45	-	E	65	-	e
06	-	ACK	26	-	&	46	-	F	66	-	f
07	-	BEL	27	-	'	47	-	G	67	-	g
08	-	BS	28	-	(	48	-	H	68	-	h
09	-	HT	29	-	)	49	-	I	69	-	i
0A	-	LF	2A	-	*	4A	-	J	6A	-	j
0B	-	VT	2B	-	+	4B	-	K	6B	-	k
0C	-	FF	2C	-	,	4C	-	L	6C	-	l
0D	-	CR	2D	-	-	4D	-	M	6D	-	m
0E	-	SO	2E	-	.	4E	-	N	6E	-	n
0F	-	SI	2F	-	/	4F	-	O	6F	-	o
10	-	DLE	30	-	0	50	-	P	70	-	p
11	-	DC1	31	-	1	51	-	Q	71	-	q
12	-	DC2	32	-	2	52	-	R	72	-	r
13	-	DC3	33	-	3	53	-	S	73	-	s
14	-	DC4	34	-	4	54	-	T	74	-	t
15	-	NAK	35	-	5	55	-	U	75	-	u
16	-	SYN	36	-	6	56	-	V	76	-	v
17	-	ETB	37	-	7	57	-	W	77	-	w
18	-	CAN	38	-	8	58	-	X	78	-	x
19	-	EM	39	-	9	59	-	Y	79	-	y
1A	-	SUB	3A	-	:	5A	-	Z	7A	-	z
1B	-	ESC	3B	-	;	5B	-	[	7B	-	{
1C	-	FS	3C	-	<	5C	-	\	7C	-	
1D	-	GS	3D	-	=	5D	-	]	7D	-	}
1E	-	RS	3E	-	>	5E	-	^	7E	-	~
1F	-	US	3F	-	?	5F	-	_	7F	-	CLEAR E

32  
3  
96

TEXT ATOMS (HEX CODE ORDER)

80 - LIST	A0 - PRINT	C0 - FN	E0 - LS=
81 - CLEAR	A1 - LOAD	C1 - ABS(	E1 - ALL
82 - RUN	A2 - REM	C2 - SQR(	E2 - PACK
83 - REMEMBER	A3 - RESTORE	C3 - COS(	E3 - CLOSE
84 - CONTINUE	A4 - PLOT	C4 - EXP(	E4 - INIT
85 - SAVE	A5 - SELECT	C5 - INT(	E5 - HEX
86 - LIMITS	A6 - COM	C6 - LOG(	E6 - UNPACK
87 - COPY	A7 - PRINTUSING	C7 - SIN(	E7 - BCCL
88 - KEYIN	A8 - MAT	C8 - SGN(	E8 - ADD
89 - DSKIP	A9 - REWIND	C9 - RND(	E9 - ROTATE
8A - AND	AA - SKIP	CA - TAN(	EA - (stmt)
8B - OR	AB - BACKSPACE	CB - ARC	EB - ERROR
8C - XOR	AC - SCRATCH	CC - #PI	EC - ERR
8D - TEMP	AD - MOVE	CD - TAN(	ED - DAC
8E - DISK	AE - CONVERT	CE - DEFFN	EE - DSC
8F - TAPE	AF - [SELECT] PLOT	CF - [ARC] TAN(	EF - SUB
90 - TRACE	B0 - STEP	DO - [ARC] SIN(	FO - LINPUT
91 - LET	B1 - THEN	D1 - [ARC] COS(	F1 - VER(
92 - FIX(	B2 - TO	D2 - HEX(	F2 - ELSE
93 - DIM	B3 - BEG	D3 - STR(	F3 - SPACE
94 - ON	B4 - OPEN	D4 - ATN(	F4 - not used
95 - STOP	B5 - [SELECT] CI	D5 - LEN(	F5 - AT(
96 - END	B6 - [SELECT] R	D6 - RE	F6 - HEXOF(
97 - DATA	B7 - [SELECT] D	D7 - [SELECT] #	F7 - MAX(
98 - READ	B8 - [SELECT] CO	D8 - % [IMAGE]	F8 - MIN(
99 - INPUT	B9 - LGT(	D9 - [SELECT] F	F9 - MOD(
9A - GOSUB	BA - OFF	DA - BT	FA - ROUND
9B - RETURN	BB - DBACKSPACE	DB - [SELECT] G	FB - not used
9C - GOTO	BC - VERIFY	DC - VAL(	FC - not used
9D - NEXT	BD - DA	DD - NUM(	FD - not used
9E - FOR	BE - BA	DE - BIN(	FE - not used
9F - IF	BF - DC	DF - POS(	FF - line num

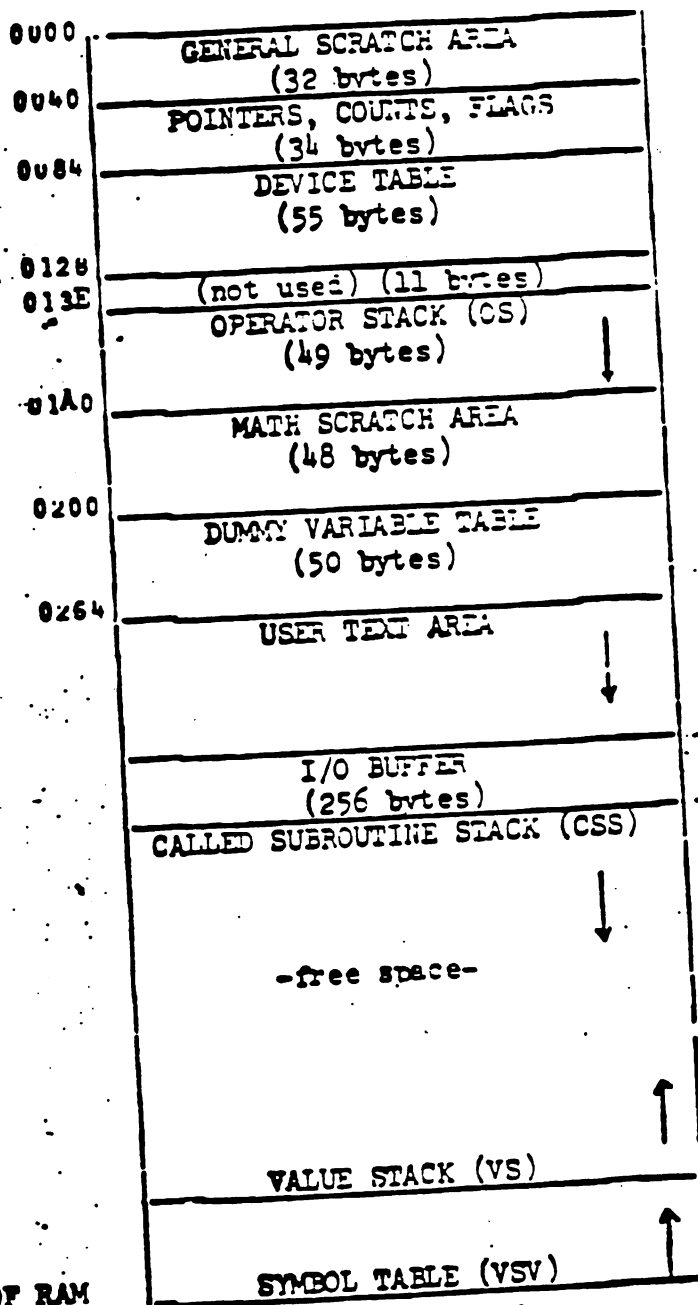
The following atoms are valid only on the 2200VP.

92 - FIX(	F2 - ELSE
B9 - LGT(	F3 - SPACE
EB - ERROR	F5 - AT(
EC - ERR	F6 - HEXOF(
ED - DAC	F7 - MAX(
EE - DSC	F8 - MIN(
EF - SUB	F9 - MOD(
FO - LINPUT	FA - ROUND
F1 - VER(	

The following atoms are not used (or stored on the disk) by either 2200T or 2200VP within a line of BASIC text.

FA FB FC FD FE

RAM MAP



END OF RAM



## INTRODUCTION

The 2200 BASIC calculator is a single-user, noninterrupt micro-programmed system which allows a user to execute programs written in 2200 BASIC language. 2200 BASIC conforms to most Dartmouth conventions but is expanded to include many other features. The BASIC compiler is interpretive, in that it operates directly on the entered user text, saving it in RAM where required.

## 2200 BASIC MEMORY

The micro-programmed BASIC compiler (2200A) occupies 8K of 20-bit ROM. In addition, 1.5K of 8-bit ROM are used to store math constants, text-word lists, a text atom table, timing constants, console device information, and other constants. The users RAM area is expandable from 4K (8-bit byte) up to 32K in 4K increments. Note, the BASIC compiler uses a portion of the first 4K RAM module for stacks, a device table, and scratch areas.

## USER PROGRAM PROCESSING

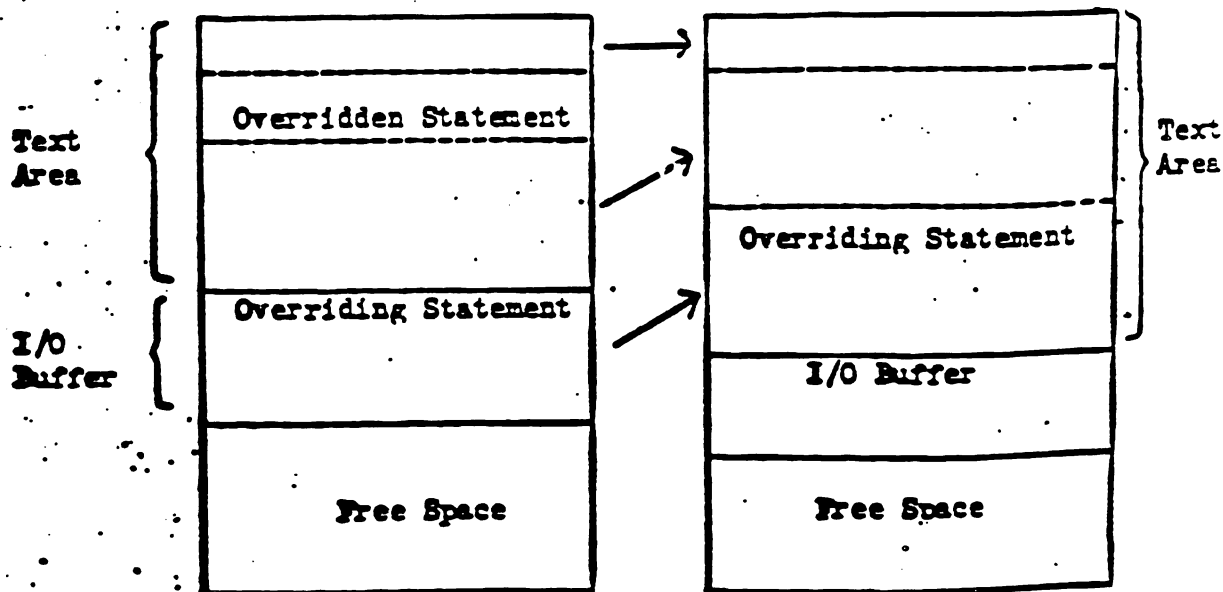
When the 2200 BASIC system is operating, it is in one of three phases: text entry, variable and line number resolution, or program execution.

### TEXT ENTRY PHASE

Text entry phase is identified by a colon (or '?' for INPUT) being output; the system then waits for the user to input text. Input characters are placed into the I/O buffer (at end of text area) until a carriage return is encountered. The text line is then syntactically analyzed. Syntax errors cause an error message to be output. If there are no syntax errors and the text line is a system command, the command is executed; if it is a statement without a line number, it is executed as a one line user program (immediate execution). If it is a statement with a line number, it is threaded into the users present text program. That is, two bytes are reserved before each text line in the program as a pointer (thread) to the next highest program text line. Hence, when a new text line is added, the next lowest text line is found and its thread is set to point to the new line. If the line number of the new text line is equal to the line number of a previously entered line, then the old text line is removed from the text area and the new statement is threaded into the program. (If the overriding text line consists solely of a line number and a carriage return, the overridden text line is removed but the new line is not threaded in - this is line deletion). Note, lines with syntax errors are threaded into the users program. Removal of redundant statement lines is illustrated below.

a). prior to repacking

b). after repacking



DEVICE TABLE

INDEVICE (Input device for text entry)  
 OUTDEVICE (Output device for text entry)  
 PRINTFLAG (PRINT flag)  
 DEVTYP (Current device type)  
 LATDEV (Last active tape device)  
 PRNTLC (PRINT line count)  
 TLMCF (telecommunications flag)

0091
0096
0098
0099
009A
009C
009E

DEVICE DEVICE CARRIAGE  
 TYPE ADDRESS WIDTH

INPUT	0090		
PRINT	0096		
LIST	009C		
CONSOLE INPUT	00A2		
CONSOLE OUTPUT	00B0		
PLOT	00AE		
TAPE	00B4		

disk only

DISK	disk only			
	DEVICE TYPE	STARTING ADDR.	CURRENT ADDRESS	ENDING ADDRESS
#1	0095			
#2	0098			
#3	009B			
#4	00F8			
#5	0103			
#6	0116			

DEVICE TYPES:

- 0 = parallel ASCII
- 1 = serial 2200 cassette
- 2 = parallel ASCII with combined CR/LF
- 3 = disk
- 4 = parallel ASCII with no carriage return generated at end of line (plotter, etc.)

VARIABLE AND VALUE FORMATS

When a variable is defined by a user, it is allocated space in the symbol table during resolution phase. Numeric variables are initially given a value of 0; alphanumeric variables are given a value of one blank character. Alphanumeric variable values have a default maximum length of 16 characters which may be overridden by the user with a DIM statement (the user may set the maximum length from 1 to 64 characters).

Each symbol table entry consists of 2 parts: symbol prefix (name, atom, dimensions, thread to next symbol, etc.) and the symbol data (i.e., variable values).

**SYMBOL TABLE ENTRY FORMAT**

LETTER	
DIGIT OR F16	ATOM
THREAD TO NEXT SYMBOL	
DIM #1	
DIM #2	
MAX. STRING LENGTH	
SYMBOL DATA	

} last entry in table has thread = 0

} arrays only (for vectors DIM #2 = 1)

} alphanumeric variables only

**SYMBOL ATOM**



numeric = 0, alphanumeric = 1

00 = scalar, 01 = vector, 11 = array

## VARIABLE AND LINE NUMBER RESOLUTION PHASE

Resolution phase is entered just prior to execution phase; it is triggered by a RUN, ~~STEP~~, ~~CONTINUE~~, or ~~special function command~~. Its function is to build the variable symbol table, allocate value areas for user variables, and assure that referenced user line numbers do exist. It consists of a complete scan of the users program. If the entire pass is error free, then the resolution phase transfers control to the execution phase directly. If an error is detected, a message is output and execution is inhibited; control returns to the entry phase.

The resolution phase scan verifies the presence of valid line numbers and user defined functions that have been referenced, and constructs a variable symbol/value table. As each referenced line number or user defined function is encountered, the body of the user text is scanned for a match with the current element. As each referenced variable is encountered, the symbol table is scanned for a match with the current variable name. If a match is found, then the scan of the program continues. If no match is found, then the variable is entered into the table and assigned a value of zero (numerics) or one blank character (alphanumerics). The symbol table is built up from high address to low address.

## EXECUTION PHASE

Execution phase is entered only after a successful resolution phase. During execution phase, each statement is executed as it is scanned. Three pushdown stacks are now active: the called subroutine stack (CSS), the value stack (VS), and the operator stack (OS). The CSS is used principally to store subroutine return addresses for recursive subroutines. The value stack is used for operand storage during expression evaluation and for such purposes as storing loop and subroutine information. The OS stores operator atoms for expression analysis as well as atoms for looping and subroutines, etc.

## RECURSION

The 2200 CPU has a 16-level, circular subroutine return stack. Hence, for recursive subroutines, which could overflow this stack, a special called subroutine stack (CSS) is set up in RAM. By recursion we mean that elements of the language are defined as containing the elements themselves. For example, an expression may contain within itself smaller elements which are themselves expressions. Thus, when the syntax scan encounters an expression, it will call the expression processor, which may, at some later part of the scan, itself call the expression processor.

The called subroutine stack is a pushdown stack that operates on a last-in/first-out basis. Before entry into a recursive routine, a call is made to PUSHR which stores the return address in the CSS. Exit from the recursive routine is made by a branch to POPR which removes the return address from the CSS and branches to that point.

Recursive 2200 BASIC subroutines:

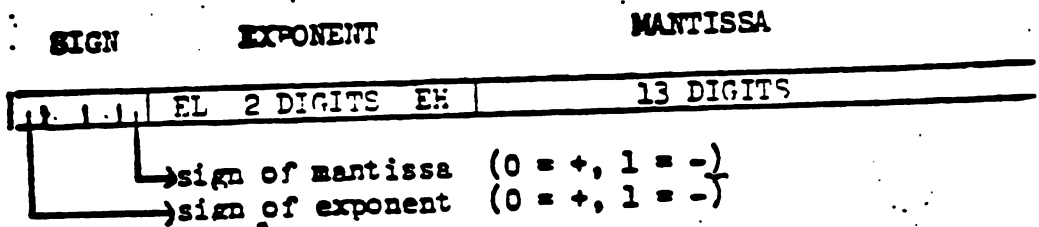
- VAR - process a numeric variable
- EXPR - expression processor
- TERM - evaluate a term
- FUNC - process a function

## EXPRESSION EVALUATION

Expression evaluation is performed by the EXPR and TERM routines. When an operand is encountered, its value is stored in the value stack (VS). As each operator is encountered, it is compared with the last operator already in the operator stack. If the last operator in the stack is of higher priority than the current operator, it is removed from the OS and the specified operation is performed on the last two values in the VS. The result replaces the two values in the value stack. Stack operator execution is continued until an operator with lower priority than the current operator is found; at this point the current operator is added to the OS and the scan of the expression continues. Note if a

The thread to the next symbol is a pointer to the next symbol in the symbol table used to speed up searching for a particular variable. The values of arrays are stored row by row from left to right.

Numeric values are normalized (leading zeroes removed) and stored in floating point format. The decimal point is assumed to be after the first mantissa digit. (scientific notation).



Alphanumeric values are character strings which are left justified and filled in with blanks on the right up to the maximum length of the value. The end of the value is assumed to be the last nonblank character (except when the value is all blanks, in which case, the value is assumed to be one blank). Hence, trailing blanks are not part of alphanumeric values. For example, the program

```

10 A$ = "ABC  "
20 PRINT A$
  
```

would print 'ABC' with no trailing spaces.

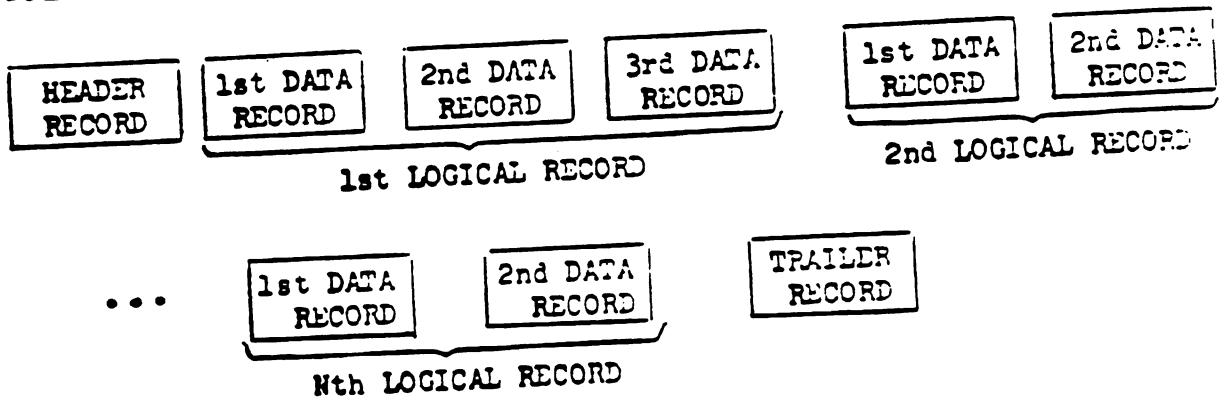
NOTE:  
 EL = Low digit of exponent  
 EH = High digit of exponent

## DATA FILES:

A series of logical data records can be made into a Data File, similar to a Program File, by preceding the records with a HEADER record and following the records with a TRAILER record. Unlike Program files however, the header and trailer record are not automatically generated by the 2200 system. They must be generated by the user's program using special forms of the DATASAVE statement.

DATASAVE OPEN "FILE1" (Write a Data File header record on tape and name the file "FILE1"; Data Files must be named)  
DATASAVE END (Write a Data File trailer record on tape)

Therefore a Data File constructed by a series of DATASAVE statements would be as follows:



The header, data records, and trailer record are similar to those in a Program file except that the control information in the records identifies them as Data File records.

## LOGICAL DATA RECORDS:

Since all programs and data are recorded in 256 byte physical records, it is possible for the values of the variable list of a DATASAVE statement to exceed 256 bytes. In this case two or more physical records are written. The one or more physical records written by the execution of one DATASAVE statement is called a LOGICAL RECORD. When data is read back by a DATALOAD statement, the entire LOGICAL RECORD is read, reading physical records sequentially one at a time. If there are more values on a logical record than are called for in a variable list of a DATALOAD statement, the unused values will be bypassed.



## SUBROUTINES (GOSUB/RETURN)

A subroutine call is made when a GOSUB statement is executed, and a return from the subroutine is made when a RETURN statement is executed. When a GOSUB statement is encountered, the following steps are taken:

1. The address of the statement immediately following the GOSUB statement is put into the VS.
2. A subroutine atom is placed into the OS.

When a RETURN statement is encountered, the value stack is scanned for a subroutine return address. Any incompleted FOR groups encountered are flushed from the VS. If there are no subroutine return addresses in the VS, an error message is issued. If a return address is found, processing proceeds at the specified statement.

<u>Operator</u>	<u>Priority</u>
Operations within parentheses	4
↑	3
• /	2
+ -	1

## LOOPING (FOR/NEXT)

A BASIC program loop is initiated by a FOR statement and terminated by a NEXT statement. When the FOR statement is encountered, the following steps occur:

1. The scalar variable (index variable) is set equal to the value of the initial expression.
2. The address of the index variable is placed in the value stack.
3. The address of the statement following the FOR statement is placed into the VS.
4. The values of the limit and step expressions are placed into the VS.
5. A 'FOR' atom is placed into the OS.

The 20 bytes of information in the VS is known as the 'FOR group'. The FOR group is not touched until the companion NEXT statement is executed (or the FOR group is flushed, see SUBROUTINES (GOSUB/RETURN)).

When the NEXT statement is encountered, the system scans the VS (which now only contains FOR groups and GOSUB return addresses) to find the FOR group to which the current NEXT applies. When the correct FOR group is found, the index variable is retrieved and its value is incremented by the step value. If the new index value is greater than or equal to the limit value, the current FOR group is flushed from the VS and processing proceeds in line number order. If the index value is less than the limit, processing continues at the address stored in the FOR group.

Note, scanning for the FOR group in the VS proceeds only until the stack is empty, or a subroutine return address is found at which time an error message is output. Hence, the following sequence is illegal:

```

10 FOR I = 1 TO 10
20 GOSUB 30
30 NEXT I

```

### Search For A User Defined Function

During Resolution phase, the address of the first user defined function is stored in memory. When a reference to a user defined function made, memory is searched starting at the location of the first user defined function.

Therefore, to optimize your program, you should bunch them together. This would minimize the amount of memory to be searched.

## Search For A Special Function (DEFFN')

During the Resolution phase, the address of the first 16 special functions are stored in a table.

When your program references a special function, this table is used to determine if it contains a pointer to the special function key the program is searching for. If so, the address is quickly retrieved and program execution will begin there.

If the special function is not in this table, it uses the address of the last entry as a starting point for a search through memory for the desired special function.

Therefore, if you were to define your special functions in a bunch the search time required would be minimal. Additionally, defining them in the order in which they are most frequently used will improve the search time more.

In order to determine the impact of DEFFN'/GOSUB' in a program, I created the following 1600 line BASIC program (line numbered 0001 - 1600).

```
0001 GOTO 1550
0002 DEFFN'33: X=33: RETURN
0003 DEFFN'34: X=34: RETURN
.
.
.
0018 DEFFN'49: X=49: RETURN
0019 REM
.
.
.
1444 REM
1445 DEFFN'50: X=50: RETURN
1446 REM
.
.
.
1549 REM
1550 FOR I= 1 TO 10000: GOSUB'33: NEXT I
1551 PRINT "*"
1552 REM
.
.
.
1600 REM
```

Using the above program, the time to complete the loop was approximately 9 seconds. Modifying Line 1550 with GOSUB '39 (the 7th defined special function) took 10 seconds. Modifying Line 1550 with GOSUB '49 (the 17th defined special function) took 14 seconds. Modifying Line 1550 with GOSUB '50 (the 18th defined special function) took 2 minutes 55 seconds.

## Search For The Value Of A Variable

During the Resolution phase, memory is allocated for each variable encountered in your program. Also during this phase a table is constructed which contains a pointer to the first variable whose name begins with a particular letter (there are 26 pointers, one for each letter of the alphabet). This pointer is the starting point of a "linked list" of the variables which all have a common first letter.

When a value is needed, the first letter of the alphabet is used to get the address of the first variable. Then the "linked list" forward pointer is used to search for the particular variable.

The variables within the "linked list" are listed in the reverse order in which they are encountered in memory.

In order to optimize your program, you may wish to make use of all 26 letters of the alphabet when naming variables. Additionally, you may wish to define your variable in the reverse order of their frequency of use.

## Search For A Line Number

When the user executes a program in memory, the BASIC-2 Interpreter performs many functions which are grouped together in a phase called Resolution. One of these functions "threads" your program in line sequence order. It is the result of this function that you do not have to enter your program line in order.

Another function of this phase, divides your program into up to 16 "groups" of lines. Each "group" contains the  $\text{INT}(L/16+1)$  number of lines where L is the total number of lines in memory. The starting address for each group is stored in a table.

Whenever, the program must search for a line number, it uses the group threads to quickly identify the block where the line exists. The group pointer is then used as the starting point for a memory search for the line number.

This technique was implemented so that a search for a particular line number must not always start at the beginning of the program.

In order to determine the impact of GOTO/GOSUB in a program, I created the following 1600 line BASIC program (line numbered 0001 - 1600).

```
0001 REM
.
.
.
0099 GOTO 1550
0100 REM
0101 RETURN
0102 REM
.
.
.
1549 REM
1550 FOR I= 1 TO 10000: GOSUB 101: NEXT I
1551 PRINT "*"
1552 REM
.
.
.
1600 REM
```

Expecting Line 101 to be the 1st line of the 2nd group (and therefore within the group table), I expected this to be very quick. The time to complete this loop took 10 seconds. I modified Lines 101, 150 and 1550 so that the GOSUB was now at Line 150 (expecting it to be at the midpoint of the threading for a group), the time to complete the loop then took 7.7 seconds (WHY??). Again modifying Line 150, 199 and 1550 so that the GOSUB was at line 199 (expecting it to be the last thread in a group), the time to complete the loop was again 10 seconds.

NOTE: That after each modification I saved the program, cleared memory and re-loaded the program so that the line were in sequence in memory. This should not of mattered, however.

## 2200 DISK INDEX HASHING TECHNIQUE

THE LOCATION IN THE DISK INDEX TO STORE A NAME IS DETERMINED BY USING A HASH FUNCTION. THE HASH IS CALCULATED FROM THE 8 BYTE NAME AS FOLLOWS. THE EXCLUSIVE OR OF THE 8 BYTES IS CALCULATED (S).  $3*S$  EQUALS A 2 BYTE RESULT, R. THE 2 BYTES OF R ARE ADDED TOGETHER IN BINARY TO PRODUCE THE HASH VALUE. THE HASH VALUE IS THEN REDUCED TO A NUMBER LESS THAN THE NUMBER OF INDEX SECTORS BY REPEATEDLY SUBTRACTING THE NUMBER OF INDEX SECTORS FROM THE HASH VALUE. THIS VALUE IS THE LOCATION IN THE INDEX WHERE THE NAME IS TO BE STORED. IF THERE IS NO ROOM AT THIS SECTOR (I. E., ALL 16 ENTRIES ARE FILLED), THE VALUE IS DECREMENTED BY 1 AND THAT SECTOR IS SEARCHED FOR A FREE ENTRY. THIS PROCESS CONTINUES UNTIL A FREE POSITION HAS BEEN FOUND OR ALL THE SECTORS HAVE BEEN SCANNED. IN WHICH CASE AN ERROR IS GIVEN.

THE BASIC PROGRAM BELOW CALCULATES THE HASH VALUE FOR ANY GIVEN NAME GIVEN THE NUMBER OF INDEX SECTORS:

```
:CLEAR

READY
:LOAD
:LIST
10 DIM N$8, L$1, H$2
20 PRINT
30 INPUT "NO. OF INDEX SECTORS", S
40 INPUT "NAME", N$
50 XOR (STR(N$, 2), N$) :REM -- EXCLUSIVE OR EACH BYTE OF NAME
60 L$=STR(N$, 8, 1) :REM -- L$=EXCLUSIVE OR OF EACH BYTE
70 H$=HEX(0000)
80 ADDC(H$, L$): ADDC(H$, L$): ADDC(H$, L$) :REM -- H$=3*L$
90 ADD(STR(H$, 1, 1), STR(H$, 2, 1)) :REM -- ADD 2 BYTES OF H$
100 H=VAL(H$) :REM -- CONVERT RESULT TO NUMERIC
110 H=H-INT(H/S)*S :REM -- REDUCE HASH VALUE < NO. OF SECTORS
120 PRINT "HASHES TO SECTOR": H
:RUN
```

```
NO. OF INDEX SECTORS? 24
NAME? JOHN
HASHES TO SECTOR 9
```