GENERAL UTILITIES

OPERATOR'S MANUAL

TABLE OF CONTENTS

SECTION 1 General Description
Introduction 1-1
Installation Instructions 1-1

SECTION 2 Magnetic Tape Utilities 2-1

SECTION 3 Array Utilities 3-1

SECTION 4 Binary Utilities 4-1

SECTION 5 General Utilities 5-1

SECTION 6 Edit Utilities 6-1

APPENDIX A Error Messages A-1

APPENDIX B Summary of Routines B-1

GENERAL DESCRIPTION

Introduction

The TransEra Utilities are firmware utility routines that enhance the capabilities of the Tektronix 4050 Series Graphic Systems. The five utility groups are as follows: Magnetic Tape Utilities, Array Utilities, Binary Utilities, General Utilities, and Edit Utilities.

These utility routines are accessed in a program by the CALL statement, but are written in assembly language for maximum execution speed. A detailed description for each routine is provided in the following sections. The format of the description consists of the CALL statement format, a definition of the parameters, and a general description of operation.

In the listing of the calling parameters, the letters O and I following the parameter name indicate whether the parameter is used as an input to the routine, an output from the routine, or both. In general, input parameters may be variables, expressions, or constants. Output parameters must be variables. Error checking is done by the routine to insure correct typing of the calling parameters.

Installation Instructions

The power to the 4050 should be turned off before the ROM Pack is installed. After the power is shut off, the ROM Pack may be inserted into a slot in the firmware backpack or into a slot of a ROM Expander unit. Press down gently until the edge card connector is seated in the receptacle connector.

MAGNETIC TAPE UTILITIES

Introduction 2-2

Naming Routines
NAME 2-3
TLI2 2-4

Status Routines
FILE? 2-5
TYPE? 2-6
STATUS 2-7
OPEN? 2-8

Error Recovery Routines
MARK2 2-9
TFRWRD 2-10
TBACK 2-10
TREAD 2-11
TWRITE 2-11

4051 Enhancement Routines
MTPACK 2-12
FIND 2-13
SETAPE 2-14

Summary of Magnetic Tape Utility Routines 2-15

MAGNETIC TAPE UTILITIES

Introduction

The Magnetic Tape Utilities are firmware routines that enhance the features of the 4050 internal magnetic tape drive. These routines include file naming and status utilities, error recovery routines, and routines that add some of the features of the 4052/54 tape drive to the 4051.

There are two naming routines and four status routines in the Magnetic Tape Utilities. They are NAME, TLI2, FILE?, STATUS, OPEN?, and TYPE?. NAME and TLI2 allow naming tape files and displaying the names. FILE?, STATUS, OPEN?, and TYPE? allow the user to determine the current status of the tape drive and file control system.

There are five error recovery routines. They are MARK2, TFRWRD, TBACK, TREAD, and TWRITE. MARK2 allows a file to be remarked without destroying the next file. TFRWRD and TBACK move the tape forward or backward one physical block. TREAD and TWRITE allow unformatted access to one physical block to repair damaged data blocks.

There are three 4051 enhancement routines. They are SETAPE, FIND, and MTPACK. SETAPE and FIND are also included in the 4052/54 version of the ROM Pack for compatibility, but MTPACK is not, since it is built in to the 4052 and 4054. SETAPE locates the beginning of the closest file on the tape. FIND finds and opens the desired tape file without first rewinding the whole tape. MTPACK duplicates the CALL "MTPACK" built in to the 4052 and 4054.

NAME

CALL "NAME",A$

A$:I DATA TO BE PUT IN HEADER OF TAPE FILE

NAME puts a name in the header record of the current tape file. The format used does not affect the operation of the standard TLIST, but allows TLI2 to see the names. The file to be named must have been opened just prior to calling NAME. The name is inserted into positions 45 through 71 of the header record which are normally unused. Strings longer than 27 characters are truncated.

Example:

```
100 FOR I=1 TO 10
110 FIND I
120 PRINT "ENTER NAME FOR FILE ";I;": ";
130 INPUT A$
140 CALL "NAME",A$
150 NEXT I
160 CALL "TLI2"
170 END
```

In this example, the user is prompted for a name for each file between 1 and 10. NAME puts the name in the file header for later access by TLI2.

TLI2

```
CALL "TLI2"[,D]

 D:I  DEVICE TO SEND LISTING TO
      (OPTIONAL - default is graphics screen)
```

TLI2 reads the file headers from the tape and prints them to the device specified. The default device is the graphics screen. Files named by the NAME routine are shown with their names, while unnamed files appear as in a normal TLIST.

Example:

```
100 CALL "TLI2"
110 CALL "TLI2",51
120 END
```

In this example, TLI2 sends a listing of tape files to the screen and the printer interface in slot 51.

FILE?

CALL "FILE?",A

A:O TARGET VARIABLE FOR FILE NUMBER (-1 IF NOT VALID)

FILE? returns the number of the file currently under the tape head. If for some reason, no file is valid, -1 will be returned.

Example:

```
100 INPUT N
110 FIND N
120 CALL "FILE?",A
130 PRINT "FILE: ";A
140 END
```

In this exmple, FILE? returns the number of the most recently found file. This should be the same as N.

TYPE?

CALL "TYPE?",A,B,C

```
A:O  TARGET VARIABLE FOR SECRET (1=SECRET)
B:O  TARGET VARIABLE FOR PROGRAM (1=PROGRAM)
C:O  TARGET VARIABLE FOR TYPE

         C

     0=LAST
     1=NEW
     2=ASCII
     3=BINARY
```

TYPE? returns the type of file last opened. OPEN? should be called first to determine if there is a valid opened file in the system. The first parameter, A, is returned with a value of one if the current file is secret, zero otherwise. The second parameter, B, is returned with a value of one if the current file is a program file, zero otherwise. The third parameter, C, is returned with a value of zero if the current file is the LAST file, one for a NEW file, two for an ASCII file, or three for a BINARY file.

Example:

```
100 CALL "TYPE?",A,B,C
110 A$=""
120 IF A=1 THEN 140
130 A$="NOT "
140 PRINT "THE CURRENT FILE IS ";A$;"SECRET."
150 C=C+1+(C>1)*2*B
160 FOR I=1 TO C
170 READ A$
180 NEXT I
190 PRINT "IT IS A ";A$;" FILE."
200 DATA "LAST","NEW"
210 DATA "ASCII DATA","BINARY DATA"
220 DATA "ASCII PROGRAM","BINARY PROGRAM"
230 END
```

In this example, TYPE? returns the type of the current file. Lines 110 through 140 print to the user whether the file is secret. Lines 150 through 220 print to the user whether the file is LAST, NEW, ASCII DATA, BINARY DATA, ASCII PROGRAM, or BINARY PROGRAM.

STATUS

CALL "STATUS",A,B,C

```
 A:O  TARGET VARIABLE FOR SHORT RECORD (1=SHORT RECORD)
 B:O  TARGET VARIABLE FOR NO CHECKSUM FORMAT (1=NO CHECKSUM)
 C:O  TARGET VARIABLE FOR NO HEADER FORMAT (1=NO HEADER)
```

STATUS returns the tape status as set by PRINT @33,0:. The target variables are in the same order and have the same meaning as in the PRI @33,0: statement that sets the status.

Example:

```
100 INPUT A,B,C
110 PRINT @33,0:A,B,C
120 CALL "STATUS",X,Y,Z
130 PRINT X,Y,Z
140 END
```

In this example, STATUS returns the same values into X, Y, and Z as were used in line 110 for A, B, and C, respectively.

OPEN?

CALL "OPEN?",A,B

A:O TARGET VARIABLE FOR TAPE IN STATUS
B:O TARGET VARIABLE FOR OPEN STATUS

A	B
0=TAPE NOT IN	0=FILE NOT OPEN
1=WRITE PROTECTED	1=JUST OPENED
2=WRITABLE	2=ACCESSED

OPEN? returns the current status of the tape drive and control system. The first parameter, A, is returned with a value of zero if there is no tape cartridge in the tape drive or if the tape drive has not been initialized since the cartridge was inserted, a value of one if there is a tape cartridge present, but it is write protected, or a value of two if there is a tape cartridge present and it is not write protected.

The second parameter, B, is returned with a value of zero if there is no open file, a value of one if the file has been opened, but not accessed yet, or a value of two if the file has been both opened and accessed.

Example:

```
100 CALL "OPEN?",A,B
110 RESTORE 150
120 FOR I=1 TO A+1
130 READ A$
140 NEXT I
150 DATA "NOT PRESENT.","WRITE PROCTECTED."
160 DATA "WRITABLE."
170 PRINT "THE TAPE IS ";A$
180 RESTORE 220
190 FOR I=1 TO B+1
200 READ A$
210 NEXT I
220 DATA "NOT OPEN.","JUST OPENED."
230 DATA "ACCESSED."
240 PRINT "THE CURRENT FILE IS ";A$
250 END
```

In this example, OPEN? determines whether the tape is not present, write protected, or writable. It also determines whether a file is not open, just opened, or has been accessed.

MARK2

CALL "MARK2",A

A:I NUMBER OF BYTES TO MARK

MARK2 is an enhancement of the MARK statement in BASIC. It allows the user to mark a tape file (usually to repair the file) without marking the next file as the dummy LAST file. To call MARK2, the tape must have just been positioned to the beginning of a file. MARK2 erases and formats the file, then rewinds the tape back to the beginning of the file. The file is marked NEW and all data is overwritten.

NOTE: MARK2 should ALWAYS be used to mark the file to the same number of records as the file was originally marked to. Failure to observe this may result in losing other files.

Example:

```
100 FIND 3
110 CALL "MARK2",2000
120 END
```

In this example, MARK2 marks file 3 to 2000 bytes without disturbing any other files.

TFRWRD and TBACK (Tape Forward and Tape Back Up)

CALL "TFRWRD"
CALL "TBACK"

TFRWRD moves the internal magnetic tape forward one physical record without reading or writing any data. TBACK moves the tape backward one physical record without reading or writing any data. These routines are intended to be used with TREAD and TWRITE to aid recovery of data from damaged tape files and repair of tape files. Since the motion of the tape is in physical records, no information should be in the buffer waiting to be written to the tape. Also, these routines will produce very unpredictable results if interspersed with the normal INPUT, PRINT, READ, and WRITE statements.

Example:

```
100 FIND 3
110 FOR I=1 TO 3
120 CALL "TFRWRD"
130 NEXT I
140 FOR I=1 TO 2
150 CALL "TBACK"
160 NEXT I
170 END
```

In this example, lines 110 through 130 position the tape head at the beginning of the fourth block of file 3. Lines 140 through 160 position the tape head back to the second block of the same file.

TREAD and TWRITE (Tape Read and Tape Write)

CALL "TREAD",A$

A$:O STRING TO PUT RECORD INTO (Must be dimensioned to 258 or larger)

CALL "TWRITE",A$

A$:I STRING TO WRITE TO TAPE (Must be dimensioned to 258 or larger, length must be 128 or 256)

TREAD and TWRITE are intended to be used with TFRWRD and TBACK to help in recovering data from damaged tape files and repairing files. TREAD reads one physical record from the internal magnetic tape. TWRITE writes one physical record on the tape.

The recommended procedure to follow in recovering data from damaged tape files is to position the tape head just before the unreadable record, call TREAD to read the physical record, call TBACK to move the tape back to the original position, modify the string to eliminate incorrect data, and call TWRITE to write the corrected string back to tape.

Caution must be exercised when using these routines, because they may cause additional damage to the tape file if complications arise. It is strongly recommended that these routines not be used unless the tape file is already damaged too badly to be read normally. Also, these routines should be used more cautiously on non-standard tape format files.

Example:

```
100 FIND 3
110 FOR I=1 TO 3
120 CALL "TFRWRD"
130 NEXT I
140 DIM A$(258)
150 CALL "TREAD",A$
160 CALL "TBACK"
170 CALL "TWRITE,A$
180 END
```

In this example, lines 110 through 130 position the tape head at the beginning of the fourth record in file 3. TREAD reads the block, TBACK backs up to the beginning of the block again, and TWRITE writes the same data back into the block.

MTPACK   Mag Tape Pack

CALL "MTPACK"

MTPACK is essentially the same routine built in to the 4052 and 4054. It is included only in the 4051 version for this reason. This routine should be called several times to break in a new tape before marking the files. It can also be used to keep the tape from packing unevenly.

Example:

```
100 FOR I=1 TO 5
110 CALL "MTPACK"
120 NEXT I
130 END
```

In this example, the tape is packed five times to break in a new tape. This will work the same on a 4051 with this ROM Pack installed as on a 4052 or 4054.

FIND

```
CALL "FIND",F

 F:I  FILE NUMBER TO FIND
```

FIND is essentially the same as the BASIC FIND statement in the 4052 and 4054. It allows the 4051 to find a tape file without the neccessity of rewinding the entire tape first. When FIND is called, it rewinds the tape looking for the start of a file. When a file is found, the header is read to locate the position of the tape. FIND then runs the tape in the appropriate direction to find the desired file. FIND is included in the 4052/54 version to provide BASIC software compatibility.

Example:

```
100 CALL "FIND",3
110 END
```

In this example, file 3 is found and opened. This will work the same on all 4050 series machines. It is the same as the FIND statement on the 4052/54.

SETAPE

CALL "SETAPE"

SETAPE rewinds the tape until the start of a file is seen. It then reads the file header to find the position of the tape.

Example:

```
100 CALL "SETAPE"
110 FIND 3
120 END
```

In this example, SETAPE locates the current position of the tape, so the FIND statement can find file 3 without rewinding the whole tape to the beginning. This will work in both the 4051 and 4052/54.

Summary of Magnetic Tape Utility Routines

```
CALL "NAME",A$
CALL "TLI2",D
CALL "FILE?",A
CALL "TYPE?",A,B,C
CALL "STATUS",A,B,C
CALL "OPEN?",A,B
CALL "MARK2",A
CALL "TFRWRD"
CALL "TBACK"
CALL "TREAD",A$
CALL "TWRITE",A$
CALL "MTPACK"
CALL "FIND",F
CALL "SETAPE"
```

ARRAY UTILITIES

Introduction 3-2

Subscripting Routines
- ROW 3-3
- COL 3-3
- POS 3-4
- NDOUT 3-5
- NDIN 3-5

Array Transfer Routines
- SEND 3-6
- ALOAD 3-7
- WRITE$ 3-8
- READ$ 3-9
- SWAP 3-10
- ASORT 3-11

Array Math Routines
- ARRAY 3-12
- OPERATOR 3-13
- COMPARE 3-14
- MAX 3-15
- MIN 3-15
- SUM 3-16
- ARYSET 3-17

Summary of Array Utility Routines 3-18

ARRAY UTILITIES

Introduction

The Array Utilities are firmware routines that complement the array functions built into the 4050 Series. These routines include subscripting functions, transfer and formatting functions, math functions, and multi-dimensional subscripting.

There are five subscripting routines in the Array Utilities. They are ROW, COL, POS, NDOUT, and NDIN. ROW and COL return the dimensioned size and shape of an array. POS returns the row and column subscripts of an array element given its linear position in the array. NDOUT and NDIN allow subscripting of an array as if it had up to five dimensions.

There are six array transfer routines. They are SEND, ALOAD, WRITE$, READ$, SWAP, and ASORT. SEND moves data from a source array to a target array. ALOAD inputs data from a specifiec input device into a target array. WRITE$ formats data from an array into a string variable using an IMAGE string similar to the PRINT USING statement. READ$ transfers data from a string to an array. SWAP exchanges the contents of two arrays, scalars, or strings. ASORT performs a shell sort on an array by rows or columns. Most of these routines allow the user to specify a starting location in the array and an increment.

There are seven array math routines. They are ARRAY, OPERATOR, COMPARE, MAX, MIN, SUM, and ARYSET. ARRAY performs +, -, *, /, and ^ arithmetic operations on array data. OPERATOR performs monadic (SIN, COS, SQR, erc.) operations on array data. COMPARE finds the first element less then, greater than, equal to, or not equal to a given value. MAX and MIN find the largest and smallest value in an array. SUM sums the elements in an array. ARYSET defines an array with a starting value and an incremental value for each successive array element.

ROW and COL (Row and Column Dimension)

```
CALL "ROW",A[,I1]
CALL "COL",A[,I1]

 A:I  ARRAY
I1:O  TARGET (OPTIONAL)
```

The ROW and COL routines return the row and column dimensions of the specified array to the target variable or the Graphics System screen. A one dimensional array has a column dimension of zero.

Example:

```
100 DIM A(3,2)
110 CALL "ROW",A
120 CALL "COL",A,C
130 PRINT C
```

Output:

```
3
2
```

POS (Position)

```
CALL "POS",A,I2[,I,J]

 A:I  ARRAY
I2:I  POSITION
 I:O  TARGET FOR ROW (OPTIONAL)
 J:O  TARGET FOR COLUMN (OPTIONAL)

      NOTE:  I AND J MUST BOTH BE PRESENT OR ABSENT
```

The POS routine returns the row and column subscript of an element in an array, given its linear position in the array. The subscripts are returned to the target variables, if passed, or the Graphics System screen. The row subscript is printed to the screen first.

Example:

```
100 DIM A(3,3)
110 CALL "POS",A,6,I,J
120 PRINT I,J
```

Output:

```
 2            3
```

NDOUT and NDIN (n-Dimensional Input and Output)

CALL "NDOUT",A,S,D,I

```
A:I   SOURCE ARRAY
S:O   TARGET SCALAR
D:I   VECTOR OF PSEUDO DIMENSIONS
I:I   VECTOR OF PSEUDO INDICES
```

CALL "NDIN",A,S,D,I

```
A:O   TARGET ARRAY
S:I   SOURCE SCALAR
D:I   VECTOR OF PSEUDO DIMENSIONS
I:I   VECTOR OF PSEUDO INDICES
```

The NDOUT and NDIN routines allow subscripting into an array as if it had up to five dimensions. The pseudo dimensions and pseudo indices are passed in separate singly dimensioned arrays. There may be from 1 to 5 pseudo dimensions. The dimensions are in lexicographic order, that is, the first element of the pseudo dimension and pseudo index arrays is weighted the greatest in determining the position of the specified element of the sequentially stored array. NDOUT transfers data from the array to the scalar. NDIN transfers data from the scalar to the array.

Example:

```
100 DIM A(720),D(5),I(5)
110 READ D
120 DATA 2,3,4,5,6
130 READ I
140 DATA 1,1,3,5,2
150 FOR X=1 TO 720
160 A(X)=X
170 NEXT X
180 CALL "NDOUT",A,S,D,I
190 PRINT S
200 CALL "NDIN",A,3,D,I
210 END
```

Output:

```
86
```

In this example, array A is set up to look like a five dimensional array with dimensions (2,3,4,5,6). NDOUT puts element 86 from array A into scalar S. NDIN puts a value of 3 into the same element.

SEND

```
CALL "SEND",A(X,Y),I,B(X1,Y1),I1,[N]

 A:I  SOURCE ARRAY
 X:I  ROW IN A
 Y:I  COLUMN IN A
 I:I  INCREMENT FOR A
 B:O  TARGET ARRAY
X1:I  ROW IN B
Y1:I  COLUMN IN B
I1:I  INCREMENT FOR B
 N:I  NUMBER OF ELEMENTS   (OPTIONAL)
```

The SEND routine moves data from the source array to the target array. The array elements are used starting at the specified starting position and incremented by the specified increment. The routine stops when the specified number of elements have been transfered or the end of either array encountered.

Example:

```
100 DATA 1,2,3,4,5,6,7,8,9
110 DIM A(3,3)
120 DIM B(5)
130 READ A
140 B=0
150 CALL "SEND",A(1,1),2,B(2),1,3
160 PRINT B
```

Output:

```
0            1            3            5
0
```

ALOAD (Array Load)

```
CALL "ALOAD",D,A,X,I[,N]

 D:I  DEVICE TO INPUT FROM
 A:O  TARGET ARRAY
 X:I  STARTING POINT IN INPUT
 I:I  INCREMENT (INPUT)
 N:I  NUMBER TO INPUT (OPTIONAL)
```

The ALOAD routine inputs data from the specified input device and puts the data in the target array. Data up to the specified input starting point is disregarded. The incoming data is put into the target array according to the specified increment. Data incremented past is disregarded. The target array is loaded sequentially starting with the first element of the array. The routine stops when an end of file condition is encountered, the specified number of numbers have been loaded into the target array, or the end of the target array is encountered.

Due to complexities in the I/O process, if the specified input device is nonexistent or does not send enough data, the routine may hang waiting for input. This will require pressing the BREAK key twice to regain control of the computer.

Example:

```
100 DIM A(5)
110 A=0
120 FIND 1
130 CALL "ALOAD",33,A,3,2,4
140 PRINT A
```

Output:

```
 3          5          7            9
 0
```

Tape file 1 contains integers starting with 1 and incrementing by 1.

WRITE$ (Write to String)

CALL "WRITE$",F$,A$[,I1],A(X,Y)[,I,N]

F$:I FORMAT FOR PACKING
A$:O TARGET STRING
I1:I STARTING POSITION IN TARGET STRING (OPTIONAL)
A:I SOURCE ARRAY
X:I ROW IN A
Y:I COLUMN IN A
I:I INCREMENT (OPTIONAL)
N:I NUMBER TO DO (OPTIONAL)

NOTE: I MUST BE PRESENT IF N IS.

NOTE: IF THE IMAGE STRING IS TO BE USED MORE THAN ONCE, THERE MUST BE NO NON-DATA ITEMS ON THE END OF THE IMAGE STRING.

The WRITE$ routine loads the contents of the source array into the string variable using the format string. Data is put into the target string starting with the specified starting position. If the starting position is greater than 1, leading spaces are inserted in A$. The syntax of the format string is the same as for the PRINT USING statement in BASIC. The format string is recycled and used again if there is more data to process than format specifiers in the format string. Note, however, that if this is the case, there must be no trailing non-data items in the format string, or an error message will be issued.

The routine loads the contents of the source array starting with the specified starting position and increments through the source array by the specified increment. If the increment specification is not passed, the default increment of 1 is used. The routine stops when the specified number of elements have been loaded, the end of the array is encountered, or the dimensioned length of the target string is exceeded. Note that the increment specifier and the number of elements to process must both be passed, or neither be passed.

Example:

```
100 DATA 1,2,3,4,5
110 DIM A(5)
120 READ A
130 CALL "WRITE$","3D.1D",A$,4,A(2),1,3
140 PRINT A$
```

Output:

```
   2.0  3.0  4.0
```

READ$ (Read from String)

```
CALL "READ$",A$[,I1],A(X,Y)[,I,N]

A$:I   SOURCE STRING
I1:I   STARTING POSITION IN A$ (OPTIONAL)
 A:O   TARGET ARRAY
 X:I   ROW IN A
 Y:I   COLUMN IN A
 I:I   INCREMENT (OPTIONAL)
 N:I   NUMBER TO DO (OPTIONAL)

       NOTE:  I MUST BE PRESENT IF N IS.
```

The READ$ routine loads data from the source string into the target array starting with the specified position in the source string. The data is interpreted the way the BASIC VAL function would interpret the data. The target array is loaded starting with the specified starting position. The increment refers to the target array. The routine stops when the specified number of elements have been transfered, the end of the target array is encountered, or the data is exhausted.

Example:

```
100 A$="9 1 2 3"
110 DIM A(7)
120 A=0
130 CALL "READ$",A$,3,A(2),2,3
140 PRINT A
```

Output:

```
 0           1           0           2
 0           3           0
```

SWAP

CALL "SWAP",A,B

A:IO SOURCE/TARGET VARIABLE (SCALAR, STRING, OR ARRAY)
B:IO TARGET/SOURCE VARIABLE (SCALAR, STRING, OR ARRAY)

The SWAP routine exchanges the contents of a scalar variable, a string variable, or an array variable. If two string variables are to be exchanged which are incompatible, string A having a longer current length than the dimension of string B, error message 21 is issued. If arrays of different sizes are exchanged, the routine sequentially exchanges the elements of the arrays until the end of either array is encountered. No error is produced when the end of an array is encountered, and the rest of the larger array is left unaltered.

Example:

```
100 A$="THIS IS A$"
110 B$="THIS IS B$"
120 CALL "SWAP",A$,B$
130 PRINT A$
140 PRINT B$
```

Output:

```
THIS IS B$
THIS IS A$
```

ASORT (Array Sort)

```
CALL "ASORT",A,D$,X1[,X2[,X3]]

 A:IO ARRAY TO BE SORTED
D$:I  ROW OR COLUMN SORT
X1:I  PRIMARY ROW/COLUMN NUMBER
X2:I  SECONDARY ROW/COLUMN NUMBER (OPTIONAL)
X3:I  TERTIARY ROW/COLUMN NUMBER (OPTIONAL)
```

The ASORT routine performs a shell sort on the specified array. The array is sorted either by rows or by columns as specified. Row integrity is maintained for column sorts, and column integrity is maintained for row sorts. If secondary and/or tertiary row/column numbers are passed, the array is sorted by these if the primary row/column contains identical values.

Example:

```
100 DIM A(5,4)
110 DATA 5,3,7,7
120 DATA 9,8,8,2
130 DATA 4,7,5,2
140 DATA 1,5,3,5
150 DATA 3,8,2,2
160 READ A
170 PRINT A
180 CALL "ASORT",A,"C",4,2,3
190 PRINT A
200 END
```

Output:

```
5       3       7       7
9       8       8       2
4       7       5       2
1       5       3       5
3       8       2       2

4       7       5       2
3       8       2       2
9       8       8       2
1       5       3       5
5       3       7       7
```

In this example, ASORT performs a column-based sort on array A with primary column 4, secondary column 2, and tertiary column 3.

ARRAY

```
CALL "ARRAY",A(X,Y),I1,O$,B(X1,Y1),I2,C(C1),[N]

 A:I  SOURCE ARRAY 1
 X:I  ROW IN A
 Y:I  COLUMN IN A
I1:I  INCREMENT IN ARRAY A
O$:I  ARITHMETIC OPERATION
 B:I  SOURCE ARRAY 2
X1:I  ROW IN B
Y1:I  COLUMN IN B
I2:I  INCREMENT IN B
 C:O  TARGET ARRAY
C1:I  STARTING POINT IN C
 N:I  NUMBER OF OPERATIONS    (OPTIONAL)
```

The ARRAY routine performs the indicated operation on elements of two arrays and puts the results in a third array. Allowable arithmetic operators are "+", "-", "*", "/", and "^". The source arrays are used starting at the element indicated and incremented by the specified increment. The arrays may be of different size and shape. The routine stops when the specified number of elements have been operated on or the end of any of the three arrays is encountered.

Example:

```
100 DIM A(9),B(3,2),C(3)
110 DATA 1,2,3,4,5,6,7,8,9
120 DATA 11,12,13,14,15,16
130 READ A
140 READ B
150 CALL "ARRAY",A(2),3,"+",B(1,1),2,C(1),5
160 PRINT C
```

Output:

```
 13            18              23
```

OPERATOR

```
CALL "OPERATOR",F$,A(X,Y),I1,B(X1,Y1),I2,[N]

F$:I  FUNCTION NAME
 A:I  SOURCE ARRAY
 X:I  ROW IN A
 Y:I  COLUMN IN A
I1:I  INCREMENT IN ARRAY A
 B:O  TARGET ARRAY
X1:I  ROW IN B
Y1:I  COLUMN IN B
I2:I  INCREMENT IN ARRAY B
 N:I  NUMBER OF OPERATIONS   (OPTIONAL)
```

The OPERATOR routine subjects elements of the source array to the specified named monadic operator and puts the results in the target array. The arrays are used starting with the specified starting position and incremented by the specified increment. The arrays may be of different sizes and shapes.

Allowable operators are "ABS", "ACS", "ASN", "ATN", "COS", "EXP", "INT", "LGT", "LOG", "SGN", "SIN", "SQR", "TAN" as explained in the Tektronix literature. Since these operators are uniquely determined by their first two letters, only the first two letters need be passed in the string. Any extra characters are disregarded.

Example:

```
100 DATA 1,2,4,2,9,2,16,2,25
110 DIM A(9)
120 DIM B(9)
130 B=0
140 READ A
150 CALL "OPERATOR","SQR",A(1),2,B(2),1,5
160 PRINT B
```

Output:

```
0          1          2          3
4          5          0          0
0
```

COMPARE

```
CALL "COMPARE",A(X,Y),I,O$,C,R[,X1,Y1][,N]

 A:I   ARRAY TO SEARCH
 X:I   ROW NUMBER OF STARTING POINT
 Y:I   COLUMN NUMBER OF STARTING POINT
 I:I   INCREMENT
O$:I   RELATIONAL OPERATOR
 C:I   NUMBER TO COMPARE WITH
 R:O   VALUE FOUND
X1:O   ROW NUMBER OF R (OPTIONAL)
Y1:O   COLUMN NUMBER OF R (OPTIONAL)
```

The COMPARE routine permits the search of arrays. The array is searched starting with the element indicated. Allowable relational operators are "<" for less than, ">" for greater than, "=" for equal, and "N" for not equal. The first element of the array satisfying the relational operator is returned for the "=" and "N" cases. The element of the array satisfying the relational operator closest in value to the value to compare with is returned in the "<" and ">" cases.

The value of the element found is returned. Zero is returned if no value was found to satisfy the relational operator. The row and collumn subscript of the element found are also returned if X1 and Y1 are passed in the CALL statement. The array is searched until the specified number of elements have been tested or the end of the array is found. If N is not passed, the routine stops when the end of the array is encountered.

Example:

```
100 DATA 5,4,3,2,1,2,3,4,5
110 DIM A(3,3)
120 READ A
130 CALL "COMPARE",A(1,2),1,"<",2.5,R,X1,Y1,7
140 PRINT R,X1,Y1
```

Output:

```
 2       2       1
```

MAX and MIN (Maximum and Minimum)

CALL "MAX",A[,I1,X]
CALL "MIN",A[,I1,X]

A=I	ARRAY
I1=O	TARGET FOR POSITION (OPTIONAL)
X=O	TARGET FOR VALUE (OPTIONAL)

NOTE: I1 AND X MUST BOTH BE PRESENT OR ABSENT

The MAX and MIN routines return the maximum and minimum values in the specified array to the target variables or the Graphics System screen. The position is printed on the screen first. If the array is two-dimensional, the POS routine may be used to convert from the position to the row and column subscripts.

Example:

```
100 DATA 9,8,7,6,5,4,3,2,1
110 DIM A(9)
120 READ A
130 CALL "MAX",A,I1,X
140 PRINT I1,X
150 CALL "MIN",A
```

Output:

```
1          9
9
1
```

SUM

```
CALL "SUM",A(X,Y),I,S[,N]
```

```
 A:I  ARRAY TO SUM
 X:I  ROW NUMBER OF STARTING POINT
 Y:I  COLUMN NUMBER OF STARTING POINT
 I:I  INCREMENT IN A
 S:O  SUM TARGET
 N:I  NUMBER OF ELEMENTS TO SUM (OPTIONAL)
```

The SUM routine sums all the elements in the array starting at the indicated element and incrementing by the specified increment until the specified number of elements has been summed or the end of the array is encountered. The sum is returned in the target variable. If the number of elements to sum is not passed, the routine stops when the end of the array is encountered.

Example:

```
100 DATA 1,2,3,4,5,6,7,8,9
110 DIM A(9)
120 READ A
130 CALL "SUM",A(2),2,S,3
140 PRINT S
```

Output:

```
 12
```

ARYSET (Array Set)

CALL "ARYSET",A,M,B

A:O TARGET ARRAY
M:I VALUE FOR A(1)
B:I INCREMENTAL VALUE

ARYSET defines an array with a starting value and an incremental value for each successive array element. The array must have been previously DIMENSIONed to the desired number of elements. It may be one or two dimensional. The value of M is assigned to the first element in the array. M+B is assigned to the second element, and so on. ARYSET stops when it encounters the dimensioned end of the array.

Example:

```
100 DIM A(5)
110 CALL "ARYSET",A,100,10
120 PRINT A
130 END
```

Output:

```
110     110     120     130
140
```

In this example, ARYSET assigns the value 100 to the first element of the array A, 110 to the second element, and so on.

Example 2:

```
100 DIM X(100),Y(100)
110 CALL "ARYSET",X,0,1
120 CALL "ARYSET",Y,0,PI*2/99
130 Y=SIN(Y)
140 WINDOW 0,99,-1,1
150 MOVE X(1),Y(1)
160 DRAW X,Y
170 END
```

In this example, ARYSET assigns values 0 through 99 to elements of X and value 0 through PI*2 to elements of Y. Line 130 computes the sine of each element of Y. Lines 140 through 160 plots the sine wave defined by X and Y to the screen.

Summary of Array Utility Routines

```
CALL "ROW",A[,I1]
CALL "COL",A[,I1]
CALL "POS",A,I2[,I,J]
CALL "NDOUT",A,S,D,I
CALL "NDIN",A,S,D,I
CALL "SEND",A(X,Y),I,B(X1,Y1),I1,N
CALL "ALOAD",D,A,X,I[,N]
CALL "WRITE$",F$,A$,A(X,Y)[,I,N]
CALL "READ$",A$,A(X,Y)[,I,N]
CALL "SWAP",A,B
CALL "ASORT",A,D$,X1[,X2[,X3]]
CALL "ARRAY",A(X,Y),I1,O$,B(X1,Y1),I2,C(C1),N
CALL "OPERATOR",F$,A(X,Y),I1,B(X1,Y1),I2,N
CALL "COMPARE",A(X,Y),I,O$,C,R[,X1,Y1][,N]
CALL "MAX",A[,I1,X]
CALL "MIN",A[,I1,X]
CALL "SUM",A(X,Y),I,S[,N]
CALL "ARYSET",A,M,B
```

BINARY UTILITIES

Introduction 4-2

I/O and Graphics Routines
GPIBIN 4-3
PLOT$ 4-4
UNLEAV 4-5
MAXI 4-6
MINI 4-6
BSWAP 4-7

Data Conversion Routines
PACK 4-8
UNPACK 4-8
DECBIN 4-9
BINDEC 4-10

Arithmetic and Logic Routines
ADDB 4-11
SUBB 4-12
MULB 4-13
ANDB 4-14
ORB 4-15
EORB 4-16
ROLB 4-17
RORB 4-18
ASLB 4-19
LSRB 4-20

Summary of Binary Utility Routines 4-21

BINARY UTILITIES

Introduction

The Binary Utilities are firmware routines for manipulating binary string format data. These include I/O and graphics routines, data conversion routines, and arithmetic and logic routines.

There are six I/O and graphics routines in the Binary Utilities. They are GPIBIN, PLOT$, UNLEAV, MAXI, MINI, and BSWAP. GPIBIN allows input from the GPIB. Data taken by this routine is stored in a string variable. PLOT$ draws a graph of data in a string variable. UNLEAV separates multiplexed data from a string variable. MAXI and MINI find the value and position of the largest and smallest datum in a string.

There are four data conversion routines. They are PACK, UNPACK, DECBIN, and BINDEC. PACK and UNPACK convert between floating point data in an array variable and binary string data in a string variable. DECBIN and BINDEC convert between numeric data and ASCII 1's and 0's.

There are 10 arithmetic and logic routines. They are ADDB, SUBB, MULB, ANDB, ORB, EORB, ROLB, RORB, ASLB, and LSRB. ADDB, SUBB, and MULB perform arithmetic operations on binary string data. ANDB, ORB, and EORB perform logical operations on binary string data. ROLB, RORB, ASLB, and LSRB perform shift and rotate operations on binary string data.

GPIBIN

```
CALL "GPIBIN",A$,N

A$:O  Target for data
 N:I  Number of bytes to take
```

The GPIBIN routine takes data from the GPIB and puts it in a string variable. This allows two computers to communicate with each other over the GPIB. The data should be sent with the WBYTES statement or the PRINT statement. All items received are put into the string, including addresses.

A$ is a string variable. It must be dimensioned large enough to contain the data to be read. If not previously dimensioned, it will be dimensioned to the default length of 72. N is the number of bytes to take. It may be a simple variable, an expression, or a literal value.

Example, sender:

```
100 PRINT @15:"THIS IS A TEST."
110 END
```

Example, reciever:

```
100 CALL "GPIBIN",A$,50
110 A$=SEG(A$,3,LEN(A$))
120 PRINT A$
130 END
```

Output:

```
THIS IS A TEST.
```

In this example, the computers are tied together with the GPIB cable. The sender outputs the primary listen address for device 15, which is 47, the secondary address for print, which is 108, and the string "THIS IS A TEST." The reciever puts all this data into A$. Line 110 strips off the addresses, and line 120 prints the received string.

PLOT$

```
CALL "PLOT$",A$,D,I,S,N,B

A$:I  String to be graphed
 D:I  Device number for graphing
 I:I  Interval
 S:I  Starting point
 N:I  Number of points to plot
 B:I  Number of bytes per sample
```

The PLOT$ routine draws a graph of data in a string. A$ is the string containing the data to be graphed. D is the device to draw to. It must be a legal value. I is the incremental value between plotted points. It should be 1 for unmultiplexed data. S is the point in the string to begin graphing at. N is the Number of points to plot. B is the number of bytes (1 or 2) per sample.

PLOT$ draws a graph of every I\`th point starting at number S. It draws until N points have been graphed. The graph is drawn to device D. A$ must be a defined string. All other parameters may be simple variables, expressions, or literal values. Before calling PLOT$, the WINDOW should be set to N in the horizontal direction and the largest expected sample in the vertical direction.

Example:

```
100 DIM A$(100),A(50)
110 FOR I=1 TO 50
120 A(I)=32768*(1+SIN(PI*I/25))
130 NEXT I
140 CALL "PACK",A$,A,50,2
150 WINDOW 1,50,0,65535
160 CALL "PLOT$",A$,32,1,1,50,2
170 END
```

In this example, lines 100 through 140 create a string of 50 2-byte words of data. Line 150 sets the plotting window to the correct size. Line 160, PLOT$, plots the data to the screen using an interval of 1 and a starting location of 1. This plots all 50 2-byte words and draws one cycle of a sine wave on the screen.

UNLEAV

```
CALL "UNLEAV",A$,B$,I,S,N,B
```

```
A$:I  Source string
B$:O  Target string
 I:I  Interval
 S:I  Starting Point
 N:I  Number of points to extract
 B:I  Number of bytes per point
```

The UNLEAV routine extracts every I`th point in A$ starting at point S. It puts the extracted points in B$. It extracts until N points have been extracted. B is the number of bytes per point. B should be either 1 or 2. This is useful in separating multiplexed data.

Example:

```
100 A$="1234567890"
110 CALL "UNLEAV",A$,B$,3,2,2,1
120 PRINT B$
130 END
```

Output:

```
25
```

In this example, A$ contains ten characters. UNLEAV extracts two 1-byte points into B$ starting with the second point in A$ and using an interval of three. This extracts the digits 2 and 5. A$ is not changed.

MAXI/MINI

```
CALL "MAXI",A$,M,P,B
CALL "MINI",A$,M,P,B
```

A$:I Data string
M:O Target for value
P:O Target for position
B:I Number of bytes per sample

MAXI finds the value and position of the largest point in A$. MINI finds the value and position of the smallest point in A$. The number of bytes per sample (1 or 2) is specified by B. B may be a simple variable, an expression, or a literal value. The value of the extreme point is returned in M. The position in A$ is returned in P. M and P must be simple variables.

Example:

```
100 A$="123ABC0a123ABC"
110 CALL "MAXI",A$,M,P,1
120 PRINT "MAX",M,P
130 CALL "MINI",A$,M,P,1
140 PRINT "MIN",M,P
150 END
```

Output:

```
MAX     97     8
MIN     48     7
```

In this example, MAXI and MINI work on 1 byte data. MAXI returns the ASCII value and position of the character "a" and MINI returns the ASCII value and position of the character "0".

BSWAP

CALL "BSWAP",A$

A$:IO Data string

The BSWAP routine exchanges bytes in a string. It exchanges the first and second byte, the third and fourth bytes, and so on. The main purpose of this is to allow the other routines to manipulate data in which the least significant byte is first in the data string. If the length of the string is odd, the last byte is unaffected.

Example:

```
100 A$="1234ABCDE"
110 CALL "BSWAP",A$
120 PRINT A$
130 END
```

Output:

```
2143BADCE
```

In this example, BSWAP exchanges every 2-byte pair in A$. The length is odd, so the last byte is unaffected.

PACK/UNPACK

```
CALL "PACK",A$,A,N,B
CALL "UNPACK",A$,A,N,B
```

```
A$:IO Target or source string
 A:IO Target or source array
 N:I  Number of samples to convert
 B:I  Bytes per sample
```

The PACK routine compresses data from a floating point numeric array A to 1 or 2 byte integers and stores the binary result into string variable A$. The UNPACK routine takes data thus represented in string variable A$ and converts it to the 8 byte floating point format and stores each number in an element of the numeric array A.

N is the number of samples to convert. B is the number of bytes per sample. B should be either 1 or 2. Array A should be dimensioned to at least N elements. String A$ should be dimensioned to at least N*B bytes and the length should be at least N*B for UNPACK. If B is 1, the values PACKed can be no larger than 255. If B is 2, the values PACKed can be no larger than 65535.

Example:

```
100 DIM A(5),A$(10),B(10)
110 FOR I=1 TO 5
120 A(I)=I
130 NEXT I
140 CALL "PACK",A$,A,5,2
150 CALL "UNPACK",A$,B,10,1
160 PRINT B
170 END
```

Output:

```
0       1       0       2
0       3       0       4
0       5
```

In this example, PACK packs five elements from array A into A$ using two bytes for each sample. UNPACK unpacks ten 1-byte samples from A$ into array B. This also demonstrates that the most significant byte is first.

DECBIN

CALL "DECBIN",E$,F$,X

E$:O Target string for the most significant byte
F$:O Target string for the least significant byte
 X:I Number to convert

The DECBIN routine takes a floting point number and creates a two byte binary number represented as a most significant and a least significant byte in two 8 character string variables with ASCII 1's and 0's.

Example:

```
100 A=3*256+34
110 CALL "DECBIN",E$,F$,A
120 PRINT E$;" ";F$
130 END
```

Output:

```
00000011 00100010
```

In this example, DECBIN converts 3*256+34 into its ASCII binary representation. The most significant byte is assigned to E$, and the least significant byte is assigned to F$.

BINDEC

CALL "BINDEC",E$,X

E$:I String to convert
 X:O Target variable

The BINDEC routine performs the opposite function of the DECBIN routine. The input string E$ can be any length up to 16 characters. The binary number represented by this string of ASCII 1's and 0's is converted into a decimal number and stored in variable X.

Example:

```
100 CALL "BINDEC","100010",X
110 PRINT X
120 END
```

Output:

```
 34
```

In this example, BINDEC converts the string "100010" to decimal and assigns the result, 34, to X.

ADDB

```
CALL "ADDB",B$,I,B
CALL "ADDB",B$,C$,B
```

```
B$:IO Target and string source 1
 I:I  Decimal source 2
C$:I  String source 2
 B:I  Bytes per sample
```

The ADDB routine allows adding a single integer value to an array of packed 1 or 2 byte binary numbers. If also allows two equal lengthed strings to be summed together. In the first case, I is the integer number that will be added to each word in B$. In the second case, C$ is a string of the same length as B$. Corresponding elements of C$ and B$ are added together. In both cases, the result is stored back in B$ and B is the number of bytes per sample and should be 1 or 2.

Example:

```
100 A$="123ABC"
110 CALL "ADDB",A$,2,1
120 PRINT A$
130 END
```

Output:

```
345CDE
```

In this example, ADDB adds 2 to each byte of A$. This increases each numeric character by 2 and changes each alphabetic character to the letter 2 later in the alphabet.

SUBB

```
CALL "SUBB",B$,I,B
CALL "SUBB",B$,C$,B
```

```
B$:IO Target and string source 1
 I:I  Decimal source 2
C$:I  String source 2
 B:I  Bytes per sample
```

The SUBB routine is used in the same way as the ADDB routine. The SUBB routine subtracts I or C$ from B$ and stores the difference back into B$.

Example:

```
100 A$="567XYZ"
110 CALL "SUBB",A$,2,1
120 PRINT A$
130 END
```

Output:

```
345VWX
```

In this example, SUBB subtracts 2 from each byte in A$. This decreases each numeric character by 2 and changes each alphabetic character to the letter 2 earlier in the alphabet.

MULB

```
CALL "MULB",B$,I,B
CALL "MULB",B$,C$,B

B$:IO Target and string source 1
 I:I  Decimal source 2
C$:I  String source 2
 B:I  Bytes per sample
```

The MULB routine is used to multiply a series of binary numbers in a string by an integer or by corresponding binary numbers in another string. As with ADDB and SUBB, the each element in B$ is multiplied by I or by the corresponding element in C$ and put back into B$. B is the number of bytes per sample and should be 1 or 2.

Example:

```
100 A$="12345"
110 CALL "MULB",A$,2,1
120 PRINT A$
130 END
```

Output: bdfhj

In this example, MULB multiplies each byte in A$ by 2. This doubles the ASCII value of each character.

ANDB

CALL "ANDB",B$,C$

B$:IO Target and source string 1
C$:I Source string 2

The ANDB routine allows two string variables to be ANDed together. Each byte of B$ is ANDed with the corresponding byte of C$, and the result is put back into B$.

Example:

```
100 A$="abcdqrst"
110 CALL "ANDB",A$,"^^^^>>>>"
120 PRINT A$
130 END
```

Output:

```
@BBD0224
```

In this example, ANDB performs a logical AND operation on the strings "abcdqrst" and "^^^^>>>>". This zeroes bits 6 and 1 of the first four characters of A$ and bits 7 and 1 of the last four.

ORB

CALL "ORB",B$,C$

B$:IO Target and source string 1
C$:I Source string 2

The ORB routine is used in the same way as the ANDB routine, but is performs a logical OR between the bytes in B$ and C$.

Example:

```
100 A$="1234ABCD"
110 CALL "ORB",A$,"@@@@    "
120 PRINT A$
130 END
```

Output: qrstabcd

In this example, ORB performs a logical OR operation on the strings "1234ABCD" and "@@@@   ". This sets bit 7 of the first four characters in A$ and bit 6 in the last four.

EORB

CALL "EORB",B$,C$

B$:IO Target and source string 1
C$:I Source string 2

The EORB routine is also used in the same way as the ANDB and the ORB routines with the exception that is performs a logical EXCLUSIVE OR function.

Example:

```
100 A$="QRST5678"
110 CALL "EORB",A$,"````````"
120 PRINT A$
130 END
```

Output:

```
1234UVWX
```

In this example, EORB performs an EXCLUSIVE OR function on the strings "QRST5678" and "````````". This inverts bits 6 and 7 of each byte of A$ exchanging numbers and upper case letters.

ROLB

CALL "ROLB",B$,C,N,B

```
B$:IO Data string to be rotated.
 C:IO Initial carry value and target for final carry
 N:I  Number of bit places to shift
 B:I  Number of bytes per sample
```

The ROLB routine is used to perform a Rotate Left bit shift over 1 or 2 byte binary words. The rotate function causes the carry generated from the previous shift to be shifted into the least significant bit position. The carry may be initially set or cleared by argument C which should be 1 or 0. The value of C will be used for the carry on the first shift in each word but will be returned with the actual value of the carry resulting from the final shift on the last binary word. N is the number of bit places to shift. B is the number of bytes per binary word over which shifting is to take place and should be 1 or 2.

Example:

```
100 A$="123"
110 C=1
120 CALL "ROLB",A$,C,1,1
130 PRINT A$,C
140 END
```

Output:

```
ceg     0
```

In this example, ROLB performs a left rotate operation on the string "123" with an initial carry of 1. This essentially multiplies the ASCII value of each character by 2 and adds 1.

RORB

```
CALL "RORB",B$,C,N,B
```

```
B$:IO Data string to be rotated.
 C:IO Initial carry value and target for final carry
 N:I  Number of bit places to shift
 B:I  Number of bytes per sample
```

The RORB routine is the same as the ROLB routine except that the direction of the shifting is to the right.

Example:

```
100 A$="ceg"
110 C=0
120 CALL "RORB",A$,C,1,1
130 PRINT A$,C
140 END
```

Output:

```
123     1
```

In this example, RORB performs a right rotate operation on the string "ceg" with an initial carry of 0. This is the inverse operation of the example for ROLB. It essentially divides the ASCII value of each character by 2.

ASLB

```
CALL "ASLB",B$,C,N,B
```

```
B$:IO Data string to be shifted
 C:O  Target for final carry
 N:I  Number of bit places to shift
 B:I  Number of bytes per sample
```

The ASLB routine is an Arithmetic Shift Left operation the same as the ROLB routine except that on each shift a 0 is shifted into the least significant bit position instead of the result from the carry. The variable C is returned with the value of the carry from the last shifting operation. N is the number of bit places to shift. B is the number of bytes per word over which the shifting is to take place and should be 1 or 2.

Example:

```
100 A$="123"
110 CALL "ASLB",A$,C,1,1
120 PRINT A$,C
130 END
```

Output:

```
bdf      0
```

In this example, ASLB performs a left shift operation on the string "123". This essentially multiplies the ASCII value of each character by 2.

LSRB

CALL "LSRB",B$,C,N,B

```
B$:IO Data string to be shifted
 C:O  Target for final carry
 N:I  Number of bit places to shift
 B:I  Number of bytes per sample
```

The LSRB routine is a Logical Shift Right operation that is the same as the RORB routine except that it shifts a 0 into the most significant bit position rather than the result of the carry. It is also the same as the ASLB routine except that the direction of the shifting is reversed.

Example:

```
100 A$="bdf"
110 CALL "LSRB",A$,C,1,1
120 PRINT A$,C
130 END
```

Ouput:

```
123     0
```

In this example, LSRB performs a right shift operation on the string "bdf". This essentially divides the ASCII value of each character by 2. Note that this is the inverse of the example for ASLB.

Summary of Binary Utility Routines

```
CALL "GPIBIN",A$,N
CALL "PLOT$",A$,D,I,S,N,B
CALL "UNLEAV",A$,B$,I,S,N,B
CALL "MAXI",A$,M,P,B
CALL "MINI",A$,M,P,B
CALL "PACK",A$,A,N,B
CALL "UNPACK",A$,A,N,B
CALL "DECBIN",A$,B$,X
CALL "BINDEC",A$,X
CALL "ADDB",A$,I,B
CALL "ADDB",A$,B$,B
CALL "SUBB",A$,I,B
CALL "SUBB",A$,B$,B
CALL "MULB",A$,I,B
CALL "MULB",A$,B$,B
CALL "ANDB",A$,B$
CALL "ORB",A$,B$
CALL "EORB",A$,B$
CALL "ROLB",A$,C,N,B
CALL "RORB",A$,C,N,B
CALL "ASLB",A$,C,N,B
CALL "LSRB",A$,C,N,B
CALL "BSWAP",A$
```

GENERAL UTILITIES

Introduction 5-2

Array Routines
INTERP 5-3
INTEG 5-5
DERIV 5-6
PACK 5-8
UNPACK 5-9

String Routines
SET$ 5-10
DIM$ 5-11
LIST$ 5-12
$SORT 5-13
EDIT 5-15

I/O Routines
GETCHR 5-16
GPIBIN 5-17
PLOT$ 5-18

Miscellaneous Routines
LREF 5-19
XREF 5-20
RUN 5-21
GETRET 5-21
SWAP 5-23
BEEP 5-24

Summary of General Utility Routines 5-25

GENERAL UTILITIES

Introduction

The General Utilities are general purpose firmware utility routines. They include routines that perform a variety of useful functions not otherwise possible from BASIC ranging from cross reference utilities to special math and string functions.

There are five array routines in the General Utilities. They are INTERP, INTEG, DERIV, PACK, and UNPACK. INTERP performs linear interpolation on array data. INTEG performs integration. DERIV performs first or second order derivation. PACK converts array data to packed binary string data. UNPACK converts packed binary string data to array data.

There are five string routines. They are SET$, DIM$, LIST$, $SORT, and EDIT. SET$ fills a string with a specified character and length. DIM$ returns the dimensioned length of a string variable. LIST$ displays a string to the screen using underline format for control characters. $SORT performs a shell sort on the data in a string variable. EDIT allows the user to modify a string using the standard line editor keys.

There are three I/O routines. They are GETCHR, GPIBIN, and PLOT$. GETCHR returns the next character from the type-ahead buffer if one is present. GPIBIN reads binary data from the GPIB into a string variable. PLOT$ graphs packed binary string data directly.

There are six miscellaneous routines. They are LREF, XREF, RUN, GETRET, SWAP, and BEEP. LREF prints a cross-reference listing of lines in the current BASIC program and lines that refer to them. XREF prints a cross-reference listing of lines that refer to a specified variable or a table of references to each defined variable. RUN allows a variable to specify the line number where a program is to begin running. GETRET gets the return line number from the last user key pressed and stores it in a variable that can be saved and later recalled to restart a program from the point it was aborted by the user key by RUN. SWAP exchanges the contents of two strings, scalars, or arrays. BEEP outputs a tone of a specified frequency and duration.

INTERP (Interpolate)

CALL "INTERP",X,Y,B,C[,C1]

```
 X:I  SOURCE ARRAY, OLD INDEPENDENT VARIABLE
 Y:I  SOURCE ARRAY, OLD DATA
 B:I  SOURCE ARRAY, NEW INDEPENDENT VARIABLE
 C:O  TARGET ARRAY, NEW DATA
C1:I  INITIAL FILL VALUE (OPTIONAL)
```

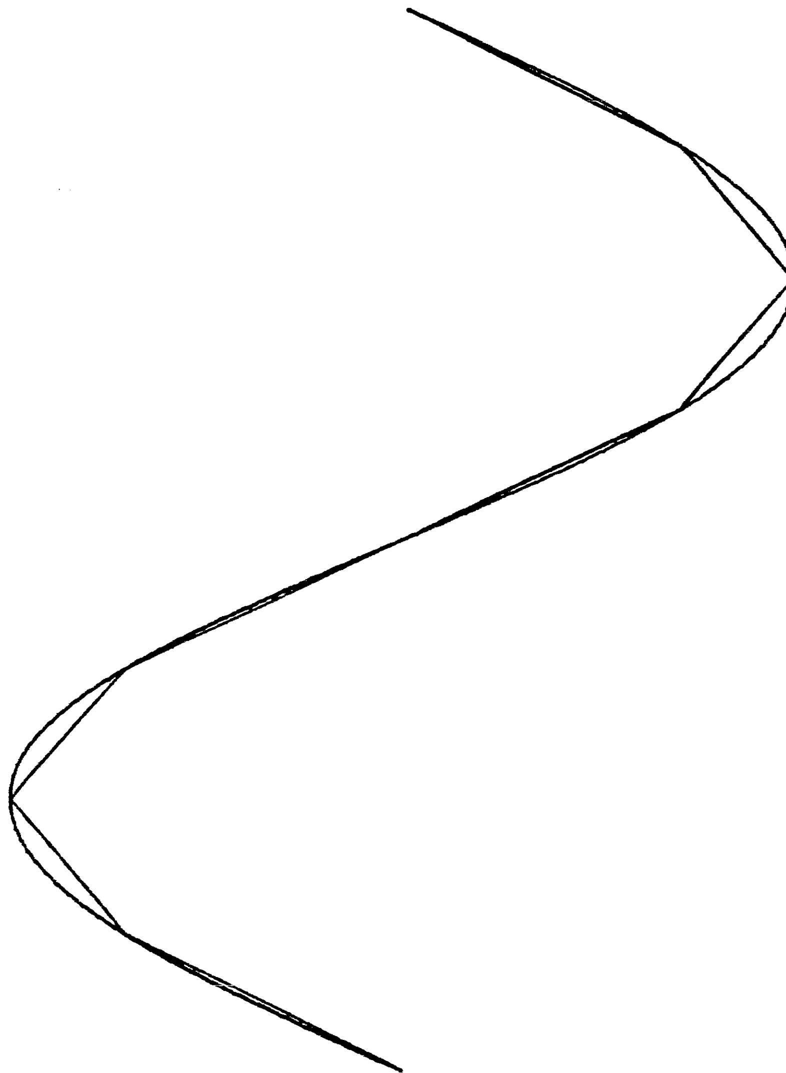
INTERP performs linear interpolation on the original data in array Y with the independent variable in array X onto the new independent variable in array B. The new data is stored in array C. The range of B should be less than or equal to the range of X. C1 is the optional fill value for C and has a default value of zero. INTERP is implemented from the following BASIC program:

```
2480 REM START OF BASIC INTERP
2490 C=C1
2500 J1=1
2510 K=X(N5)>X(1)
2520 FOR I=1 TO N4
2530 L0=B(I)
2540 IF L0<X(K+N5*(1-K)) OR L0>X(N5*K+1-K) THEN 2610
2550 FOR J=J1 TO N5-1
2560 J1=J
2570 IF L0<X((J+1)*K+J*(1-K)) AND L0=>X(J*K+(J+1)*(1-K)) THEN 2590
2580 NEXT J
2590 J=J-(J>N5-1)
2600 C(I)=(Y(J+1)-Y(J))/(X(J+1)-X(J))*(L0-X(J))+Y(J)
2610 NEXT I
2620 REM END OF BASIC INTERP
```

Example:

```
100 N5=100
110 N4=9
120 DIM X(N5),Y(N5),B(N4),C(N4)
130 FOR I=1 TO N5
140 X(I)=I
150 Y(I)=SIN((I-1)*2*PI/(N5-1))
160 NEXT I
170 FOR I=1 TO N4
180 B(I)=1+(I-1)*(N5-1)/(N4-1)
190 NEXT I
200 CALL "INTERP",X,Y,B,C
210 WINDOW 1,N5,-1,1
220 MOVE X(1),Y(1)
230 DRAW X,Y
240 MOVE B(1),C(1)
250 DRAW B,C
260 END
```

In this example, arrays X and Y approximate a sine wave in 100 points. INTERP interpolates this curve into arrays B and C, which then approximate a sine wave in 9 points. This is what the plot looks like:



INTEG (Integrate)

CALL "INTEG",X,Y,B

X:I SOURCE ARRAY, DATA
Y:I SOURCE ARRAY, INDEPENDENT VARIABLE
B:O TARGET ARRAY, INTEGRAL OF DATA

INTEG performs integration of the data in array X over the independent variable in array Y. The result for each element is stored in array B. It is implemented from the following BASIC program:

```
2840 REM BASIC INTEG
2850 S=0
2860 K=2*(X(N1)>X(1))-1
2870 K1=K<>1
2880 B(1)=0
2890 B(N1)=0
2900 FOR I=1 TO N1-1
2910 J=I*K+(N1+1)*K1
2920 S=S+0.5*(Y(J)+Y(J+K))*(X(J+K)-X(J))
2930 B(J+K)=S
2940 NEXT I
2950 REM END OF BASIC INTEG
```

Example:

```
100 N1=10
110 DIM X(N1),Y(N1),B(N1)
120 FOR I=1 TO N1
130 X(I)=I^2
140 Y(I)=I
150 NEXT I
160 CALL "INTEG",X,Y,B
170 PRINT X,Y,B
180 END
```

Output:

```
1       4       9       16
25      36      49      64
81      100

1       2       3       4
5       6       7       8
9       10

0       4.5     17      41.5
82      142.5   227     339.5
484     664.5
```

In this example, INTEG integrates the data in array X over the independent variable in array Y and stores the result for each point in array B.

DERIV (Derivative)

CALL "DERIV",A,Z,A1,N2

```
 A:I  SOURCE ARRAY, DATA
 Z:I  SOURCE ARRAY, INDEPENDENT VARIABLE
A1:O  TARGET ARRAY, DERIVATIVE OF DATA
N2:I  ORDER OF DERIVATIVE
```

DERIV performs the derivative of the data in array A with respect to the independent variable in array Z. The result for each element is stored in array A1. N2 determines the order of the derivative taken and should be 1 or 2. The first and last point of a second order derivative are not calculated and are set to zero. DERIV is implemented from the following BASIC program:

```
3120 REM START OF BASIC DERIV
3130 IF N2=2 THEN 3230
3140 FOR I=2 TO I2-1
3150 D1=Z(I)-Z(I-1)
3160 D4=Z(I+1)-Z(I)
3170 D3=1/(D1*D4*(D1+D4))
3180 A1(I)=(D4*D4*(A(I)-A(I-1))+D1*D1*(A(I+1)-A(I)))*D3
3190 NEXT I
3200 A1(1)=(A(2)-A(1))/(Z(2)-Z(1))
3210 A1(I2)=(A(I2)-A(I2-1))/(Z(I2)-Z(I2-1))
3220 GO TO 3320
3230 REM SECOND DERIVATIVE
3240 FOR I=2 TO I2-1
3250 D1=Z(I)-Z(I-1)
3260 D4=Z(I+1)-Z(I)
3270 D3=1/(D1*D4*(D1+D4))
3280 A1(I)=2*(D4*(A(I-1)-A(I))+D1*(A(I+1)-A(I)))*D3
3290 NEXT I
3300 A1(1)=0
3310 A1(I2)=0
3320 REM END OF BASIC DERIV
```

Example:

```
100 I2=10
110 DIM A(I2),Z(I2),A1(I2),A2(I2)
120 FOR I=1 TO I2
130 A(I)=I^2
140 Z(I)=I
150 NEXT I
160 CALL "DERIV",A,Z,A1,1
170 CALL "DERIV",A,Z,A2,2
180 PRINT A,Z,A1,A2
190 END
```

Output:

1	4	9	16
25	36	49	64
81	100		
1	2	3	4
5	6	7	8
9	10		
3	4	6	8
10	12	14	16
18	19		
0	2	2	2
2	2	2	2
2	0		

In this example, DERIV performs the derivative of the data in array A with respect to the independent variable in array Z. The result of the first order derivative is put in array A1, and the result of the second order derivative is put in array A2.

PACK

CALL "PACK",A$,A,N,B

A$:O TARGET STRING
A:I SOURCE ARRAY
N:I NUMBER OF POINTS TO CONVERT
B:I BYTES PER POINT

PACK converts floating point data in an array to packed binary data in a string variable. PACK takes data from array A, converts it, and puts the converted data into string variable A$. N is the number of points to convert. B is the number of bytes of A$ to use for each converted point and may be either 1 or 2. Array A must contain at least N defined elements. String variable A$ must be dimensioned to at least B*N bytes.

PACK rounds each element to be converted to the nearest whole number and packs it into binary format. If B is 1, values of 0 to 255 are valid. If B is 2, values of 0 to 65535 are valid.

Example:

```
100 DIM A(5),A$(5)
110 READ A
120 DATA 48,49,50,51,52
130 CALL "PACK",A$,A,5,1
140 PRINT A$
150 END
```

Output:

```
01234
```

In this example, PACK converts five elements from A to 1-byte binary format and puts the converted data in A$. A$ thus contains 5 characters when pack has finished.

UNPACK

CALL "UNPACK",A$,A,N,B

```
A$:I  SOURCE STRING
 A:O  TARGET ARRAY
 N:I  NUMBER OF POINTS TO DO
 B:I  BYTES PER POINT
```

UNPACK converts packed binary data from string A$ and puts the converted values into floating point array A. It is essentially the reverse of the PACK routine. N is the number of points to convert. B is the number of bytes to take from A$ for each conversion and may be either 1 or 2. Array A must be dimensioned to at least N bytes and string A$ must contain at least N*B characters.

UNPACK converts each binary value to the corresponding integer in floating point format. If B is 1, values from 0 to 255 may be produced. If B is 2, values from 0 to 65535 may be produced.

Example:

```
100 DIM A(5)
110 CALL "UNPACK","01234",A,5,1
120 PRINT A
130 END
```

Output:

```
48        49        50        51
52
```

In this example, UNPACK takes its data from the literal string "01234" and puts the converted data into array A. Five points are converted, using one byte from the source string for each point.

SET$ (Set String)

```
CALL "SET$",A$,A[,N[,I]]

A$:O  TARGET STRING
 A:I  ORDINAL VALUE TO FILL WITH
 N:I  NUMBER OF CHARACTERS TO FILL   (OPTIONAL)
 I:I  STARTING POSITION IN STRING   (OPTIONAL)
```

The SET$ routine fills a string with a specified ordinal value. This is intended to be primarily used for preparing a string for the $SORT routine. Any ordinal value may be passed up to 65535, but the value is taken mod 256 before filling the string. If the string is intended to be used for the $SORT routine, the ordinal value passed should be zero.

The string length is set to the lesser of N+I-1 and the dimensioned length of the string. Characters are filled starting at position I unless I is greater than the dimensioned length.

Example:

```
100 CALL "SET$",A$,65,10
110 PRINT A$
120 CALL "SET$",A$,66,5,8
130 PRINT A$
```

Output:

```
AAAAAAAAAA
AAAAAAABBBBB
```

DIM$ (Dimensioned Length of a String)

CALL "DIM$",A$,I

A$:I STRING IN QUESTION
I:O TARGET FOR DIMENSIONED LENGTH

The DIM$ routine returns the dimensioned length of a string variable. This is the argument of the most recently executed DIM statement.

Example:

```
100 DIM A$(500)
110 CALL "DIM$",A$,I
120 PRINT I
130 END
```

Output:

```
 500
```

LIST$

CALL "LIST$",A$

A$:I STRING TO PRINT

LIST$ prints the string A$ to the screen in list format. This displays all control characters except control M (carriage return) as the corresponding upper case letter with an underline.

$SORT (String Sort)

```
CALL "$SORT",A$,S,X1[,X2[,X3]]
```

```
A$:IO STRING TO BE SORTED
 S:I  RECORD SIZE
X1:I  OFFSET TO PRIMARY SUBRECORD (STARTS AT 0)
X2:I  OFFSET TO SECONDARY SUBRECORD (OPTIONAL)
X3:I  OFFSET TO TERTIARY SUBRECORD (OPTIONAL)
```

The $SORT routine performs a shell sort on the data in a string variable. The string is assumed to have a fixed record size, which is passed into the routine. It is also assumed to be filled with padding characters. A record may contain any number of subrecords or fields, which may be of mixed lengths. The offset used to index into the subrecords starts at zero, that is, the first subrecord has an offset of zero. There must be at least one null (ASCII value of zero) character at the end of any subrecord used in the sort to act as a delimiter.

The sort is done by comparing the subrecord indexed by the primary offset. If subrecords contain identical data, the secondary offset, if passed, is used, and so on. When ordering the records, entire records are moved, so record integrity is maintained. Upper case and lower case are treated as different. The sorted string is in increasing order of ASCII values.

Example:

```
100 DIM A$(250)
110 CALL "SET$",A$,0
120 FOR I=1 TO 250 STEP 10
130 READ B$
140 CALL "SET$",B$,0,1,LEN(B$)+1
150 A$=REP(B$,I,LEN(B$))
160 NEXT I
170 DATA "A","Z","3","8","BILL"
180 DATA "B","Y","1","8","FRED"
190 DATA "C","X","2","8","JOE"
200 DATA "D","W","6","7","SAM"
210 DATA "E","V","4","5","GEORGE"
220 CALL "$SORT",A$,50,30,20,40
230 FOR I=1 TO 250 STEP 50
240 FOR J=0 TO 40 STEP 10
250 B$=SEG(A$,I+J,10)
260 PRINT B$;" ";
270 NEXT J
280 PRINT
290 NEXT I
300 END
```

```
Output:         E V 4 5 GEORGE
                D W 6 7 SAM
                B Y 1 8 FRED
                C X 2 8 JOE
                A Z 3 8 BILL
```

In this example, $SORT uses a record lentgh of 50 characters. Each record is made up of five subrecords of 10 characters each. Lines 100 through 210 initialize A$. $SORT uses a primary offset of 30, a secondary offset of 20, and a tertiary offset of 40. This sorts A$ according to the fourth, third, then fifth subrecords.

EDIT

CALL "EDIT",Z$

Z$:IO DATA STRING TO BE EDITED

The EDIT routine displays the string to be edited, allows the user to modify the string using the standard line editor keys on the 4050, and stores the resulting string when the carriage return is pressed.

The maximum size of the data string to be edited is 72 characters. An attempt to edit a longer string will result in an error message. If the string variable is dimensioned smaller than the length of the string after editing, which can only happen if the string is dimensioned smaller than 72, the string is truncated when it is put into the variable. A line number may be printed to the left of the string, but the line number will not be reprinted by expand, compress, or reprint.

The data string may be extracted from the file string by SUBSTR, and put back in the file string by REPLACE. EDIT may also be used to insert material by passing the null string to EDIT and using INSERT to put the line into the file string. In this case, EDIT works much like the BASIC string INPUT statement.

The string passed to EDIT should not contain any carriage returns in the line string. Cursor motion is unpredictable if the line contains a carriage return.

Example:

```
100 A$="THIS IS A TEST"
110 CALL "EDIT",A$
```

This example prints "THIS IS A TEST" on the graphics system screen, allows the user to modify the string, and returns the modified string in A$.

GETCHR (Get Character)

CALL "GETCHR",A$

A$:O TARGET FOR CHARACTER (LENGTH=0 IF NOTHING THERE)

GETCHR tests for the presence of a character from the keyboard, returns that character if present, or returns the null string if there is no character present. This is intended to be used to aid cursor generation and command input procedures.

Example:

```
100 CALL "GETCHR",A$
110 PRINT A$;
120 IF A$<>"@" THEN 100
```

This example echos the input from the keyboard until the at sign is pressed. When the at sign is pressed, the program drops out. When no key has been pressed, A$ is empty, so line 110 does nothing.

GPIBIN

```
CALL "GPIBIN",A$,N

A$:O  Target for data
 N:I  Number of bytes to take
```

The GPIBIN routine takes data from the GPIB and puts it in a string variable. This allows two computers to communicate with each other over the GPIB. The data should be sent with the WBYTES statement or the PRINT statement. All items received are put into the string, including addresses.

A$ is a string variable. It must be dimensioned large enough to contain the data to be read. If not previously dimensioned, it will be dimensioned to the default length of 72. N is the number of bytes to take. It may be a simple variable, an expression, or a literal value.

Example, sender:

```
100 PRINT @15:"THIS IS A TEST."
110 END
```

Example, reciever:

```
100 CALL "GPIBIN",A$,50
110 A$=SEG(A$,3,LEN(A$))
120 PRINT A$
130 END
```

Output:

```
THIS IS A TEST.
```

In this example, the computers are tied together with the GPIB cable. The sender outputs the primary listen address for device 15, which is 47, the secondary address for print, which is 108, and the string "THIS IS A TEST." The reciever puts all this data into A$. Line 110 strips off the addresses, and line 120 prints the received string.

PLOT$

```
CALL "PLOT$",A$,D,I,S,N,B
```

```
A$:I  String to be graphed
 D:I  Device number for graphing
 I:I  Interval
 S:I  Starting point
 N:I  Number of points to plot
 B:I  Number of bytes per sample
```

The PLOT$ routine draws a graph of data in a string. A$ is the string containing the data to be graphed. D is the device to draw to. It must be a legal value. I is the incremental value between plotted points. It should be 1 for unmultiplexed data. S is the point in the string to begin graphing at. N is the Number of points to plot. B is the number of bytes (1 or 2) per sample.

PLOT$ draws a graph of every I`th point starting at number S. It draws until N points have been graphed. The graph is drawn to device D. A$ must be a defined string. All other parameters may be simple variables, expressions, or literal values. Before calling PLOT$, the WINDOW should be set to N in the horizontal direction and the largest expected sample in the vertical direction.

Example:

```
100 DIM A$(100),A(50)
110 FOR I=1 TO 50
120 A(I)=32768*(1+SIN(PI*I/25))
130 NEXT I
140 CALL "PACK",A$,A,50,2
150 WINDOW 1,50,0,65535
160 CALL "PLOT$",A$,32,1,1,50,2
170 END
```

In this example, lines 100 through 140 create a string of 50 2-byte words of data. Line 150 sets the plotting window to the correct size. Line 160, PLOT$, plots the data to the screen using an interval of 1 and a starting location of 1. This plots all 50 2-byte words and draws one cycle of a sine wave on the screen.

LREF (Line Cross-reference)

CALL "LREF"[,D],A

```
 D:I  DEVICE TO OUTPUT TO
 A:I  1 - TABLE OF REFERENCES
      2 - LIST OF DEAD-END POINTERS
      4 - LIST OF DEAD CODE
      FUNCTIONS CAN BE CONBINED, 7=DO EVERYTHING
```

LREF produces a listing of line references in the current BASIC program. This listing can be sent to any device with the optional device number D. The default for D is 32, the Graphics System screen. Parameter A determines which of the three available functions is performed. Any combination of functions can be selected.

If A is 1, 3, 5, or 7, LREF produces a table showing each line and the lines that refer to it. If A is 2, 3, 6, or 7, LREF produces a list of lines containing references to non-existent lines. If A is 4, 5, 6, or 7, LREF produces a list of lines that cannot be reached during program execution. Lines 4 through 80 may appear in this list and should be disregarded if they are reached through a User-Definable Key. If multiple functions are selected, they will be performed in the same order as they are explained in this paragraph.

XREF (Line Cross-reference)

CALL "XREF",D[,A]

D:I DEVICE TO PRINT TO
A:I VARIABLE TO SEARCH FOR (OPTIONAL)

XREF sends a listing of the variables referenced in the current program and the lines they are used in. This listing may be sent to any I/O device including the mag tape. If a variable A is specified, only references to it are printed. If no variable is specified, all references to all variables are printed.

RUN and GETRET (Get Return Line Number)

CALL "RUN",L

L:I LINE NUMBER TO RUN FROM

RUN allows a variable to specify the line number where a program is to begin running. This is intended primarily to restart a program when the current line number was saved by GETRET. When CALLed from immediate mode, RUN is equivalent to the RUN statement. When CALLed from a program line, RUN is equivalent to the GO TO statement.

CALL "GETRET",L

L:O TARGET FOR THE RETURN LINE NUMBER

GETRET gets the return line number from the last user key pressed and stores it in target variable L. If no return line is found, it returns a value of zero. GETRET is intended to be used with CALL "RUN" to make overlaid programs run more effectively. The suggested procedure is as follows:

1. The user presses a key that forces a GOSUB to a line between 4 and 80.

2. GETRET puts the return line number into a variable.

3. The program saves the return line number and any important data and OLDs in the appropriate overlay.

4. The overlay runs to completion.

5. The overlay OLDs in the original program.

6. The original program reads back in the saved data and performs a CALL "RUN" to the return line number.

Since CALL "RUN" cannot reconstruct the stack entries for FOR/NEXT loops and GOSUB statements, it is recommended that the user keys be disabled during FOR/NEXT loops and subroutines. Also, it is recommended that the user keys be disabled when executing GO TO statements since GETRET will not return the correct line number if the user key is pressed while executing a GO TO statement.

Example:
(main program)

```
1 GO TO 200
4 GO TO 400
100 N=0
110 GO TO 300
200 FIND 2
210 READ @33:N,L
220 CALL "RUN",L
300 N=N+1
310 PRINT "N EQUALS ";N
320 CALL "WAIT",1
330 GO TO 200
400 CALL "GETRET",L
410 FIND 2
420 WRITE N,L
430 FIND 3
440 OLD
```

(overlay)

```
100 PRINT "OVERLAY RUNNING"
120 FIND 1
130 OLD
```

In this example, the main program (on file 1) is started by typing RUN 100. This initializes N. Lines 300 through 330 print sequential numbers to the screen. When user key 1 is pressed, lines 400 through 440 get the return line number, saves N and L, and brings in the overlay from file 3. The overlay runs and brings the original program back in. Lines 200 through 220 read N and L back in and resume operation where the user key was pressed.

SWAP

```
CALL "SWAP",A,B

 A:IO SOURCE/TARGET VARIABLE (SCALAR, STRING, OR ARRAY)
 B:IO TARGET/SOURCE VARIABLE (SCALAR, STRING, OR ARRAY)
```

The SWAP routine exchanges the contents of a scalar variable, a string variable, or an array variable. If two string variables are to be exchanged which are incompatible, string A having a longer current length than the dimension of string B, error message 21 is issued. If arrays of different sizes are exchanged, the routine sequentially exchanges the elements of the arrays until the end of either array is encountered. No error is produced when the end of an array is encountered, and the rest of the larger array is left unaltered.

Example:

```
100 A$="THIS IS A$"
110 B$="THIS IS B$"
120 CALL "SWAP",A$,B$
130 PRINT A$
140 PRINT B$
```

Output:

```
THIS IS B$
THIS IS A$
```

BEEP

```
CALL "BEEP",F,D

 F:I  FREQUENCY OF BEEP (Hz.)
 D:I  DURATION OF BEEP (SEC.)
```

BEEP generates a tone of the specified frequency and duration on the Graphics System speaker. BEEP will do its best to provide whatever tone is requested, so it is recommended that parameters be within reason.

Summary of General Utility Routines

```
CALL "INTERP",X,Y,B,C[,C1]
CALL "INTEG",X,Y,B
CALL "DERIV",A,Z,A1,N2
CALL "PACK",A$,A,N,B
CALL "UNPACK",A$,A,N,B
CALL "SET$",A$,A[,N[,I]]
CALL "DIM$",A$,I
CALL "LIST$",A$
CALL "$SORT",A$,S,X1[,X2[,X3]]
CALL "EDIT",Z$
CALL "GETCHR",A$
CALL "GPIBIN",A$,N
CALL "PLOT$",A$,D,I,S,N,B
CALL "LREF"[,D],A
CALL "XREF",D[,A]
CALL "RUN",L
CALL "GETRET",L
CALL "SWAP",A,B
CALL "BEEP",F,D
```

EDIT UTILITIES

Introduction 6-2
File String Format 6-3

File Routines
- DIM$ 6-4
- EDREAD 6-5
- LINES 6-6
- SUBSTR 6-7
- REPLAC 6-8
- INSERT 6-9
- DELETE 6-10

Line Routines
- FNDLIN 6-11
- LINNUM 6-12
- LINLEN 6-13
- CURRLN 6-14
- BACKLN 6-15
- NEXTLN 6-16

User Interface Routines
- GETCHR 6-17
- EDIT 6-18
- LIST$ 6-19

Summary of Edit Utility Routines 6-20

EDIT UTILITIES

Introduction

The Edit Utilities are firmware utility routines designed to complement the string functions supplied in the Tektronix 4050 Series Graphics Systems and allow a supervisory program in BASIC to accept commands and manipulate a string variable using these routines to perform the editing functions on data inside the string.

There are seven file routines in the Edit Utilities. They are DIM$, EDREAD, LINES, SUBSTR, REPLACE, INSERT, and DELETE. DIM$ returns the length of the file string. EDREAD reads an entire file into the file string from tape or a GPIB device. LINES counts the number of lines in a file string. SUBSTR extracts a line from the file string without disturbing the file string. REPLACE replaces a line in the file string with another line. INSERT inserts a line into the file string. DELETE removes a line from the file string.

There are six line routines. They are FNDLIN, LINNUM, LINLEN, CURRLN, BACKLN, and NEXTLN. FNDLIN locates the position of a given line in a file string given its number. LINNUM returns the number of the line at a given position in the file string. LINLEN returns the length of a line given its position. CURRLN returns the position of the beginning of a line given its position. BACKLN returns the position of the beginning of the previous line. NEXTLN returns the position of the beginning of the next line.

There are three user interface routines. They are GETCHR, EDIT, and LIST$. GETCHR returns the next character in the type-ahead buffer if one is present. EDIT modifies a string using the standard line editor keys. LIST$ prints a string on the screen using underline format for control characters.

File String Format

The format used by the edit utility routines is as follows. The entire file is contained in one large string variable. Separate lines within the file are delimited by carriage returns. This allows the file string to be printed to the 4050 graphics screen, magnetic tape drive, disk, or other I/O device.

There is no overhead within the file string for line numbers or line lengths. The number of a line is found by scanning through the string and counting the carriage returns. The length of a line is found by scanning for the next carriage return after the start of a line.

Lines within the file string are terminated by carriage returns. Thus, the first character of a file will not be a carriage return unless the first line is empty. The last character of a file should be a carriage return.

The position in the file string as used by these routines is the same as that used by the BASIC POS, SEG, and REP statements. Thus, the position of the first character in the file string is one. The position of a line is the position of the first valid character of that line. The position of an empty line is the position of the carriage return that terminates that line.

DIM$ (Dimension of a String)

CALL "DIM$",A$,A

A$:I FILE STRING
A:O TARGET FOR DIMENSIONED LENGTH OF A$

DIM$ returns the dimensioned length of a string variable. This is intended to be used in editing programs to help with the task of monitoring the size of file strings. The value returned by DIM$ minus the current length of the string equals the number of characters left in the string. The value returned by DIM$ is the argument used in the most recently executed DIM statement.

Example:

```
100 DIM A$(500),B$(100),C$(3)
110 CALL "DIM$",A$,A
120 CALL "DIM$",B$,B
130 CALL "DIM$",C$,C
140 PRINT A,B,C
150 END
```

Output:

```
500     100     3
```

In this example, DIM$ returns the dimensioned size of the three strings A$, B$, and C$.

EDREAD (Edit Read)

```
CALL "EDREAD",A$,D

A$:O  TARGET STRING TO READ INTO
 D:I  DEVICE TO GET STRING FROM
```

EDREAD performs multiple inputs from an ASCII tape file, or other device. Strings are input into the file string and separated by carriage returns until the end of file on the I/O device is encountered or the dimensioned length of the file string is exceeded. It is essentially the same as a loop in which a string is input and concatenated onto the file string followed by a carriage return.

Example:

```
100 FIND 1
110 CALL "EDREAD",A$,33
120 FIND 2
130 PRINT @33:A$
```

This example finds file 1 on the internal magnetic tape drive and inputs the file into A$. The operation stops when the data in the tape file is exhausted or A$ is filled. The entire file of data is then written on tape file 2. The PRINT statement outputs the entire file, while the INPUT statement would only input a single line.

LINES

CALL "LINES",A$,A

A$:I FILE STRING
 A:O TARGET FOR LINE COUNT

The LINES routine counts the number of lines in a file string. If the last line in the file is not terminated by a carriage return, it is not counted. Thus, LINES returns the number of carriage returns in the file string. Since LINES must scan the entire file string, it takes approximately 1 second worst case. This is for a file string length of 25,000 in the 4051 and 50,000 in the 4052/54.

Example:

```
100 FIND 1
110 CALL "EDREAD",A$,33
120 CALL "LINES",A$,X
130 PRINT X
```

This example reads a file from tape and prints the number of lines in the file.

SUBSTR (Substring)

CALL "SUBSTR",A$,A,B$

A$:I FILE STRING (IS NOT CHANGED)
A:I POSITION OF LINE TO GET
B$:O TARGET FOR LINE

The SUBSTR routine extracts a line from the file string, leaving the file string unchanged. This may be used for displaying the file string a line at a time with line numbers added by the BASIC program. It may also be used to extract a line for the EDIT routine.

If the position is greater than the length of the file string, SUBSTR returns the null string. SUBSTR puts characters from A$ into B$ until a carriage return or the end of A$ is encountered. B$ will not contain the carriage return. If an attempt is made to put more characters in B$ than will fit, an error message will be issued. If the position passed is not the first character of a line, only the part of the line from the specified position onward is transferred to B$.

Example:

```
100 FIND 1
110 CALL "EDREAD",A$,33
120 CALL "SUBSTR",A$,1,B$
130 CALL "EDIT",B$
```

This example inputs a file from tape file 1, extracts the first line from the file, and allows the user to edit the line.

REPLAC (Replace)

```
CALL "REPLAC",A$,A,B$

A$:IO FILE STRING
 A:I  POSITION TO PUT LINE
B$:I  LINE TO PUT IN
```

REPLAC replaces a line in a file string. Characters are deleted from the file string, starting from the specified position until a carriage return or the end of the file string is detected. Then, the line to be inserted is put in the file string, starting at the specified position. No carriage returns are added or taken away. The number of lines in the file string is not changed by REPLAC. A position more than one past the end of the file string will cause an error message to be issued, as will exceeding the dimensioned length of the file string. If the position passed is one greater than the length of the file string, the line is effectively concatenated onto the end of the file string.

Example:

```
100 FIND 1
110 CALL "EDREAD",A$,33
120 CALL "SUBSTR",A$,1,B$
130 CALL "EDIT",B$
140 CALL "REPLAC",A$,1,B$
```

This example inputs a file from tape file 1, extracts the first line from the file, and allows the user to edit the line. Then, the modified line is put back into the file string in the same position in the file.

INSERT

```
CALL "INSERT",A$,B$,A

A$:IO FILE STRING
B$:I  LINE TO INSERT
 A:I  POSITION TO INSERT AT
```

INSERT inserts a new line into an file string. It is the equivalent of using the BASIC REP statement to insert a string containing a single carriage return into A$ at position A, then inserting B$ at the same position. Since a carriage return is inserted into the file string, the number of lines in the file string is increased by one.

If the position is one greater than the length of the file string, the new line is concatenated onto the end of the file string and terminated by a carriage return. If the position is more than one greater than the length of the file string, an error message is issued.

Example:

```
100 FIND 1
110 CALL "EDREAD",A$,33
120 CALL "FNDLIN",A$,3,B
130 CALL "INSERT",A$,"HI",B
```

This example reads a file from tape and puts in a new line 3. The old line 3 is now line 4. If line 3 does not exist, INSERT issues an error message.

DELETE

CALL "DELETE",A$,A

A$:IO FILE STRING
 A:I POSITION TO DELETE FROM

DELETE removes a line and its terminating carriage return from the file string. Data is deleted from the file string, starting with the character pointed to by the position, until a carriage return or the end of the file string is encountered. Since the rest of the file string is compressed into the space originally taken by the deleted line, multiple lines may be deleted by repeatedly calling DELETE with the same position. However, for large files, this type of multiple line deletion is slower than finding the position of the line after the last line to be deleted and using the BASIC REP statement to delete the entire mass of data in one statement.

Example:

```
100 FIND 1
110 CALL "EDREAD",A$,33
120 CALL "FNDLIN",A$,3,B
130 CALL "DELETE",A$,B
```

This example reads a file from tape and deletes the third line. If line 3 does not exist, DELETE issues an error message. The old line 4 is now line 3.

FNDLIN (Find Line)

CALL "FNDLIN",A$,A,B

```
A$:I  FILE STRING
 A:I  LINE NUMBER
 B:O  POSITION TARGET
```

FNDLIN finds the position of a line in the file string, given the line number. The position returned by FNDLIN is the position within the file string of the first valid character in the line requested. If the line number passed to FNDLIN is less than one, FNDLIN returns a value of one. If it is greater than the number of lines in the file string, FNDLIN returns the character position one past the end of the file string. Since FNDLIN might need to scan the entire file string, it could take approximately 1 second worst case. This is for a position of 25,000 in the 4051 and 50,000 in the 4052/54.

Example:

```
100 FIND 1
110 CALL "EDREAD",A$,33
120 CALL "FNDLIN",A$,3,B
130 CALL "SUBSTR",A$,B,B$
140 CALL "EDIT",B$
150 CALL "REPLAC",A$,B,B$
```

This example reads a file string from tape file 1, finds line 3 in the file string, allows the user to edit it, and puts the modified line 3 back in the file string. If line 3 does not exist, SUBSTR will issue an error message.

LINNUM (Line Number)

CALL "LINNUM",A$,A,B

A$:I FILE STRING
A:I POSITION OF LINE TO FIND
B:O LINE NUMBER TARGET

LINNUM returns the line number of a given position within the file string. The string is scanned for carriage returns until the first carriage return beyond the indicated position is encountered. This count is then returned as the line number of the position in question. LINNUM is essentially the reverse of FNDLIN. If the position is greater than the length of the file string, the number of the last line plus one is returned. Since LINNUM might need to scan the entire file string, it could take approximately 1 second worst case. This is for a position of 25,000 in the 4051 and 50,000 in the 4052/54.

Example:

```
100 FIND 1
110 CALL "EDREAD",A$,33
120 CALL "LINNUM",A$,LEN(A$),B
```

This example finds the line number of the last character in A$. If the last line is properly terminated, this is the same as that returned by LINES. If the last line is unterminated, this is one greater than that returned by LINES.

LINLEN (Line Length)

```
CALL "LINLEN",A$,A,B

A$:I  FILE STRING
 A:I  CURRENT POSITION
 B:O  TARGET FOR THE LENGTH OF THE LINE
```

LINLEN returns the length of the line pointed to by a given position. The carriage return that terminates the line is not counted. If the given position is greater than the length of the file string, the length of the last line in the file string is returned.

Example:

```
100 FIND 1
110 CALL "EDREAD",A$,33
120 CALL "LINLEN",A$,1,B
```

This example returns the length of the first line in the file string.

CURRLN (Current Line)

```
CALL "CURRLN",A$,A,B

A$:I  FILE STRING
 A:I  CURRENT POSITION
 B:O  TARGET FOR PRECEDING CR (CAN STAY)
```

CURRLN gives the position of the beginning of the line pointed to by a given position. It essentially moves the pointer to the beginning of the current line. If the position given is already at the beginning of a line, CURRLN returns the same position. If the given position is greater than the length of the file string, the position of the start of the last line in the file string is returned. CURRLN stops on the character after a carriage return or the first position in the file string, whichever is encountered first. CURRLN returns zero if the file string is empty.

Example:

```
100 FIND 1
110 CALL "EDREAD",A$,33
120 CALL "CURRLN",A$,LEN(A$),B
```

This example returns the position of the beginning of the last line in the file string.

BACKLN   (Back Up One Line)

CALL "BACKLN",A$,A,B

A$:I FILE STRING
A:I CURRENT POSITION
B:O TARGET FOR PRECEDING LINE START (MUST MOVE)

BACKLN returns the position of the beginning of the line prior to that pointed to by a given position. If the position points to the first line in the file, a value of 1 is returned by BACKLN. If the given position is greater than the length of the file string, the position of the last line in the file is returned. BACKLN returns zero if the file string is empty.

Example:

```
100 FIND 1
110 CALL "EDREAD",A$,33
120 CALL "BACKLN",A$,LEN(A$),B
```

This example returns the position of the beginning of the second to last line in the file string.

NEXTLN (Next Line)

CALL "NEXTLN",A$,A,B

A$:I FILE STRING
A:I CURRENT POSITION
B:O TARGET FOR NEXT LINE START (MUST MOVE)

NEXTLN returns the position of the beginning of the next line in the file string. If the given position is greater than the length of the file string or points to the last line in the file string, the position one past the last character of the file string is returned. NEXTLN returns zero if the file string is empty.

Example:

```
100 FIND 1
110 CALL "EDREAD",A$,33
120 CALL "NEXTLN",A$,1,B
```

This example returns the position of the beginning of the second line in the file string.

GETCHR (Get Character)

CALL "GETCHR",A$

A$:O TARGET FOR CHARACTER (LENGTH=0 IF NOTHING THERE)

GETCHR tests for the presence of a character from the keyboard, returns that character if present, or returns the null string if there is no character present. This is intended to be used to aid cursor generation and command input procedures.

Example:

```
100 CALL "GETCHR",A$
110 PRINT A$;
120 IF A$<>"@" THEN 100
```

This example echos the input from the keyboard until the at sign is pressed. When the at sign is pressed, the program drops out. When no key has been pressed, A$ is empty, so line 110 does nothing.

EDIT

```
CALL "EDIT",Z$
```

Z$:IO DATA STRING TO BE EDITED

The EDIT routine displays the string to be edited, allows the user to modify the string using the standard line editor keys on the 4050, and stores the resulting string when the carriage return is pressed.

The maximum size of the data string to be edited is 72 characters. An attempt to edit a longer string will result in an error message. If the string variable is dimensioned smaller than the length of the string after editing, which can only happen if the string is dimensioned smaller than 72, the string is truncated when it is put into the variable. A line number may be printed to the left of the string, but the line number will not be reprinted by expand, compress, or reprint.

The data string may be extracted from the file string by SUBSTR, and put back in the file string by REPLACE. EDIT may also be used to insert material by passing the null string to EDIT and using INSERT to put the line into the file string. In this case, EDIT works much like the BASIC string INPUT statement.

The string passed to EDIT should not contain any carriage returns in the line string. Cursor motion is unpredictable if the line contains a carriage return.

Example:

```
100 A$="THIS IS A TEST"
110 CALL "EDIT",A$
```

This example prints "THIS IS A TEST" on the graphics system screen, allows the user to modify the string, and returns the modified string in A$.

LIST$

CALL "LIST$",A$

A$:I STRING TO PRINT

LIST$ prints the string A$ to the screen in list format. This displays all control characters except control M (carriage return) as the corresponding upper case letter with an underline.

Summary of Edit Utility Routines

```
CALL "DIM$",A$,A
CALL "EDREAD",A$,D
CALL "LINES",A$,A
CALL "SUBSTR",A$,A,B$
CALL "REPLAC",A$,A,B$
CALL "INSERT",A$,B$,A
CALL "DELETE",A$,A
CALL "FNDLIN",A$,A,B
CALL "LINNUM",A$,A,B
CALL "LINLEN",A$,A,B
CALL "CURRLN",A$,A,B
CALL "BACKLN",A$,A,B
CALL "NEXTLN",A$,A,B
CALL "GETCHR",A$
CALL "EDIT",Z$
CALL "LIST$",A$
```

ERROR MESSAGES

MESSAGE NUMBER	ERROR MESSAGE
12	There is an error in the parameter list in the CALL statement.
89	XREF or LREF has detected a pre-existent condition in the BASIC program indicating corruption of system memory. A system error 0 may occur with this condition present.

SUMMARY OF ROUTINES

Magnetic Tape Utility Routines

```
CALL "NAME",A$
CALL "TLI2",D
CALL "FILE?",A
CALL "TYPE?",A,B,C
CALL "STATUS",A,B,C
CALL "OPEN?",A,B
CALL "MARK2",A
CALL "TFRWRD"
CALL "TBACK"
CALL "TREAD",A$
CALL "TWRITE",A$
CALL "MTPACK"
CALL "FIND",F
CALL "SETAPE"
```

Summary of Array Utility Routines

```
CALL "ROW",A[,I1]
CALL "COL",A[,I1]
CALL "POS",A,I2[,I,J]
CALL "NDOUT",A,S,D,I
CALL "NDIN",A,S,D,I
CALL "SEND",A(X,Y),I,B(X1,Y1),I1,N
CALL "ALOAD",D,A,X,I[,N]
CALL "WRITE$",F$,A$,A(X,Y)[,I,N]
CALL "READ$",A$,A(X,Y)[,I,N]
CALL "SWAP",A,B
CALL "ASORT",A,D$,X1[,X2[,X3]]
CALL "ARRAY",A(X,Y),I1,O$,B(X1,Y1),I2,C(C1),N
CALL "OPERATOR",F$,A(X,Y),I1,B(X1,Y1),I2,N
CALL "COMPARE",A(X,Y),I,O$,C,R[,X1,Y1][,N]
CALL "MAX",A[,I1,X]
CALL "MIN",A[,I1,X]
CALL "SUM",A(X,Y),I,S[,N]
CALL "ARYSET",A,M,B
```

Summary of Binary Utility Routines

```
CALL "GPIBIN",A$,N
CALL "PLOT$",A$,D,I,S,N,B
CALL "UNLEAV",A$,B$,I,S,N,B
CALL "MAXI",A$,M,P,B
CALL "MINI",A$,M,P,B
CALL "PACK",A$,A,N,B
CALL "UNPACK",A$,A,N,B
CALL "DECBIN",A$,B$,X
CALL "BINDEC",A$,X
CALL "ADDB",A$,I,B
CALL "ADDB",A$,B$,B
CALL "SUBB",A$,I,B
CALL "SUBB",A$,B$,B
CALL "MULB",A$,I,B
CALL "MULB",A$,B$,B
CALL "ANDB",A$,B$
CALL "ORB",A$,B$
CALL "EORB",A$,B$
CALL "ROLB",A$,C,N,B
CALL "RORB",A$,C,N,B
CALL "ASLB",A$,C,N,B
CALL "LSRB",A$,C,N,B
CALL "BSWAP",A$
```

Summary of General Utility Routines

```
CALL "INTERP",X,Y,B,C[,C1]
CALL "INTEG",X,Y,B
CALL "DERIV",A,Z,A1,N2
CALL "PACK",A$,A,N,B
CALL "UNPACK",A$,A,N,B
CALL "SET$",A$,A[,N[,I]]
CALL "DIM$",A$,I
CALL "LIST$",A$
CALL "$SORT",A$,S,X1[,X2[,X3]]
CALL "EDIT",Z$
CALL "GETCHR",A$
CALL "GPIBIN",A$,N
CALL "PLOT$",A$,D,I,S,N,B
CALL "LREF"[,D],A
CALL "XREF",D[,A]
CALL "RUN",L
CALL "GETRET",L
CALL "SWAP",A,B
CALL "BEEP",F,D
```

Summary of Edit Utility Routines

```
CALL "DIM$",A$,A
CALL "EDREAD",A$,D
CALL "LINES",A$,A
CALL "SUBSTR",A$,A,B$
CALL "REPLAC",A$,A,B$
CALL "INSERT",A$,B$,A
CALL "DELETE",A$,A
CALL "FNDLIN",A$,A,B
CALL "LINNUM",A$,A,B
CALL "LINLEN",A$,A,B
CALL "CURRLN",A$,A,B
CALL "BACKLN",A$,A,B
CALL "NEXTLN",A$,A,B
CALL "GETCHR",A$
CALL "EDIT",Z$
CALL "LIST$",A$
```