∞ INFINITE BASIC ∞
FOR THE TRS-80
GENERAL DESCRIPTION

Written For RACET COMPUTES By

T.S. JOHNSTON

and

R.D. JOHNSTON

For Use On The Radio Shack® TRS-80™
Level II BASIC 16-48K Microcomputer System

IMPORTANT NOTICE

ALL RACET COMPUTES programs are distributed on an "AS IS" basis without warranty. Neither RACET COMPUTES nor the contributor makes any express or implied warranty of any kind with regard to this program material, including, but not limited to the implied warranties of merchantability and fitness for a particular purpose. Neither RACET COMPUTES nor the contributor shall be liable for incidental or consequential damages in connection with or arising out of the furnishing, use or performance of this program material.

INFINITE BASIC ™ SYSTEM DESCRIPTION

INTRODUCTION

INFINITE BASIC is a total system designed to provide the RADIO SHACK TRS-80(tm) user with a wide range of facilities not previously available. Total compatability with the existing BASIC system is maintained. Both TAPE as well as DISK oriented systems are supported.

Each facility provided with INFINITE BASIC(IB) can be selectively loaded, thus minimizing memory requirements. The use of IB functions in a users program can often dramatically decrease the execution time for many applications. Furthermore, the availability of preprogrammed complex functions will increase the users programming productivity.

The IB system is distributed as a basic "SYSTEM" module along with several independent "APPLICATION" modules. The SYSTEM module provides the BASIC interface, system routines, and loader required by the APPLICATION modules. The current application modules available are:

> INFINITE STRING: This module provides 50 different functions for string manipulation. This includes justification, truncation, rotation, translation, compression, and centering. Also included are two SORT string array functions.
>
> INFINITE MATRIX: This module provides 30 functions involving arrays and matrices. Included in this package are matrix inversion, simultaneous equations, dynamic array allocation, array copying, and matrix arithmetic routines. Also included are argument specification functions for creating generalized user written BASIC subroutines.
>
> INFINITE BUSINESS: This package provides 20 functions for business applications. Included in this series are an automatic printer pagination header/footer system, search string array, insert into sorted string array, and hash encoding functions. Also included is a packed decimal arithmetic system which can provide both increased accuracy(up to 500 decimal digits), as well as eliminating round-off errors associated with binary floating point numbers.

Additional INFINITE BASIC application systems will be available to further improve the usefulness of the TRS-80 system.

The descriptions of each of the above APPLICATION systems is provided in separate manuals. This manual documents the general system features, procedures, and use of the loader common to all uses of INFINITE BASIC.

USING INFINITE BASIC

Use of INFINITE BASIC is a two step process. The first step is the selection and loading of the desired components required by the users application. This is described in detail in later sections of this manual. The second step, described here, is the actual use of the selected functions in the users application program.

All facilities within the standard RADIO SHACK BASIC are available, along with the extensions provided by INFINITE BASIC. Each INFINITE BASIC function is recognized by the leading '&'. Documentation of available functions are contained in separate user manuals for each series available.

INFINITE BASIC functions can have from zero to ten arguments. Furthermore, some functions provide defaults for unspecified arguments. The documentation format of an INFINITE BASIC function is best illustrated with an example. Consider the Matrix Read Tape function - &MRDT as documented in the MATRIX section. The title line reads:

```
&MRDT(A <,LEN <,BN <,TN>>>)
```

This indicates that up to four arguments may be specified, A, LEN, BN, and TN. The use of '<', and its matching '>', indicate that the enclosed argument is optional. The '<' and '>' ARE NOT actually inserted in the users program line, they are for DOCUMENTATION PURPOSES ONLY. Only trailing arguments may be eliminated. The following forms are then valid:

a. &MRDT(A)
b. &MRDT(A,LEN)
c. &MRDT(A,LEN,BN)
d. &MRDT(A,LEN,BN,TN)

Suitable default values will be provided for missing arguments, as described in the documentation of each function. User variable names may be substituted for the argument names specified in the documentation. The following forms of the &MRDT function are also valid:

a. &MRDT(XE)
b. &MRDT(Z,LENGTH)
c. &MRDT(L5,I,J)
d. &MRDT(XX,I3,2,1)

Note that constants may also be used as arguments, as shown in Item(d) above.

Many INFINITE BASIC functions will accept arguments of different modes(integer, single precision, double precision, or string). Usually, modes are converted automatically where possible. Some functions, however, must have the correct mode(ie the matrix inversion function &MINV will not attempt to invert a character string matrix!). The documentation will indicate special mode requirements, or the mode is implied in the function being used. For example, the block number(BN) and tape number(TN) are implied integers in the above &MRDT function.

Each INFINITE BASIC function returns a value. The significance of this return value is documented for each function. Some functions will always have a zero or undefined return value. In all cases the function must either be on the right of an equal sign, an argument of another function or command, or appear in an input/output statement.

METHOD OF OPERATION

INFINITE BASIC system consists of the interface and application modules, along with a generalized loader. The loader assembles the modules that the user requires into a load module. This load module resides in protected memory, or can be dumped to tape or disk for reloading.

The general steps that a user follows in using INFINITE BASIC are as follows:

1. Determine which functions will be required in the application.

2. Execute the loader program from either tape or disk.

3. Specify the names of the functions determined in Step(1) above when requested.

4. Specify the memory location where the load module is to be relocated.

5. If operating from tape, the system will request loading of the application module tapes. If operating from disk, the user will be requested to enter the names of the application module files.

6. When all modules have been located and loaded, they will be relocated to the memory area requested. If operating from tape, the user will be given the option of saving the resulting load module to tape. If operating from disk, the DUMP command parameters are displayed allowing the user to save the load module to disk.

7. After the load module is created(or loaded by a SYSTEM or LOAD command), the user should enter BASIC and specify the MEMORY SIZE parameter to protect the load module.

8. INFINITE BASIC must be initialized by executing a "USR" statement. If in TAPE mode, all that is required is entering the command:

```
?USR(1)
```

In disk mode, the following sequence is used:

```
DEFUSR=&Hnnnn
?USR(1)
```

where "nnnn" is the entry "DEFUSR" value printed at the conclusion of Step(6) above. In both cases a message should be printed indicating proper initialization.

Upon completion of the above steps the program can now use the selected INFINITE BASIC functions. Note that ONLY Steps(7-8) are required once a load module has been created.

Once INFINITE BASIC is in memory it does not need to be reloaded unless power is turned off, or other action by the user is taken that violates protected memory. The initilization call specified in Step(8) can be placed at the front of a BASIC program and executed as often as desired. However, only the first initialization USR call is needed. Once initialized, INFINITE BASIC remains active until a "&NOIB" function is executed.

DESCRIPTION OF DISTRIBUTION TAPE CONTENTS

INFINITE BASIC is distributed on one or more tapes in a compatible format for both tape and disk users. The primary system distribution tape contains the following:

Cassette Side	File No.	Contents
TAPE VERSION	1.	IBLOAD - Tape version. This module contains the generalized INFINITE BASIC loader program. It is loaded using the standard "SYSTEM" command of BASIC.
	2.	MREL - Tape or disk version. This module contains all the relocatable matrix functions. This module is processed directly by IBLOAD, or loaded to disk and used with the disk version of IBLOAD (See DISK side)
	3.	SREL - Tape or disk version. This module contains all the relocatable string functions. This module can also be processed by IBLOAD or loaded to disk and used with the disk version of IBLOAD (See DISK side)
	4.	XREL - Tape or disk version. This module contains the BASIC system interface and service functions required for INFINITE BASIC. It is used by IBLOAD or loaded to disk and used with the disk version of IBLOAD (See DISK side)

Disk VERSION

1. IBLOAD - Disk version. This module contains the disk version of the INFINITE BASIC generalized loaded. It is in a standard "SYSTEM" format that can be loaded to disk using the Radio Shack TAPEDISK utility.

2. RELOAD - Disk version. This module is used for loading the application relocatable modules to disk. This includes modules contained on TAPE SIDE of this tape(MREL, SREL, XREL), or other INFINITE BASIC application modules distributed separately (such as the Business Module).

LOADING INFINITE BASIC TO DISC

The instructions below are given for use of INFINITE BASIC with disk based systems. The following steps should be used to load INFINITE BASIC to disk from the distribution tape. (Note that an ENTER key is assumed at the end of each input line).

Step #	Description
1.	Load the tape recorder with the DISK VERSION side of the distribution cassette. Reset the tape counter to zero, and position to the first file on the tape.
2.	Execute the TAPEDISK program by entering the following sequence: a. TAPEDISK b. C c. F IBLOAD/CMD:0 7000 7FFF 7082 d. (note tape counter here) e. C f. F RELOAD/CMD:0 7000 72FF 7000 g. E This sequence will load the IBLOAD and RELOAD programs to disk into files IBLOAD/CMD and RELOAD/CMD.
3.	Load the distribution Tape Side to the second file. This will be at the approximate location as noted in Step(2-d) above.
4.	Execute the following sequence: a. RELOAD b. SPECIFY OUTPUT FILESPEC ?MREL c. READY CASSETTE d. SPECIFY OUTPUT FILESPEC ?SREL e. READY CASSETTE

```
f.      SPECIFY OUTPUT FILESPEC ?XREL
g.      READY CASSETTE
h       SPECIFY OUTPUT FILESPEC ?
i       BREAK
```

The above process will load the Matrix package (MREL), String package (SREL) and the system package (XREL) into the corresponding files.

At this point all files for use with INFINITE BASIC should now be loaded to disk.

LOADER DESCRIPTION - IBLOAD

A. General Information.

Each INFINITE BASIC function required may be individually selected for loading into memory. The process of selecting and loading the desired functions into memory is performed by the IBLOAD program.

Two versions of the IBLOAD program are provided, one for TAPE and one for DISK based systems. The operations are essentially identical except for the source of input and output devices. The TAPE system is also relocated lower in memory, as required for smaller memory systems.

The naming conventions used in INFINITE BASIC are as follows:

a. Function names when used in the users BASIC program begin with an '&', ie &SRTV.

b. Names specified when using IBLOAD are identical, but begin with '@@', ie @@SRTV.

c. All string functions start with "S" (after the & or @@).

d. Most matrix functions start with "M" (after the & or @@). The exceptions are &PLUK, &PLUG, and &NOIB, which are also in MREL.

e. Other INFINITE BASIC application modules start use a different first letter("B" for the Business package, for example).

B. Description of Examples.

The use of IBLOAD is illustrated by two examples. The first is the TAPE version of IBLOAD, and the second the DISK version. Both examples assume that the user wishes to use three INFINITE BASIC functions:

Function	Name	Title
&SRTV	@@SRTV	Multivariable sort function.
&SRV$	@@SRV$	Random string generation.
&MSHP	@@MSHP	Matrix redimension and deletion.

The first two functions will be found in the string module(SREL) and the last one in the matrix module(MREL). In addition to the above two function modules, the system will require other service routines located in XREL.

All three modules must be scanned when using IBLOAD to retrieve the desired components. Other applications may require only scanning two of the modules. In any case XREL MUST BE SCANNED LAST! It is permissible to scan an unneeded module, which will only increase the time required for IBLOAD.

C. Specification of Memory Relocation.

IBLOAD creates a load module composed of the desired functions and required system support routines. The memory location of the resultant load module must be specified by the user as an input parameter to IBLOAD.

The Radio Shack BASIC system has the ability to protect an area of memory called "protected memory." This is done by setting the "MEMORY SIZE" parameter after power up, reset, or when loading BASIC from disk. The location of the resultant INFINITE BASIC load module must fall within "protected memory." The amount of space required for protected memory will depend upon the number of functions selected. The actual memory requirements will be displayed by IBLOAD after the load module has been created. This value can subsequently be used when setting the MEMORY SIZE.

Two options are provided when using IBLOAD for specifying memory assignment:

1. A minimum low address can be specified for the start of the load module (L option). Each compont selected will be placed in successively higher locations.

2. A maximum high address can be specified for the end of the load module (H option). Each component selected will be placed in successively lower locations.

Trial and error may be required for the optimum selection of memory parameters.

D. Tape Example.

The following steps are required to build a load module from tape for the three functions indicated in Section(B) above:

1. Load and position the "TAPE VERSION" side of the cassette tape to the first file.

2. Enter the following (Note that an ENTER key is assumed at the end of each line of input):

   a. SYSTEM
   b. IBLOAD
   c. / (after IBLOAD is loaded)

   The above process should load and execute IBLOAD from tape.

3. Enter the function names desired(@@ form) in response to the prompting message. These are entered one at a time as shown below:

   a. ENTER SUBROUTINE NAMES REQUIRED? @@SRTV
   b. @@SRR$
   c. @@MSHP
   d. (just an enter key)

4. The memory size parameters are specified next. For this example assume that assignment is to start from top of memory down in a 16K system (high address of 7FFF in hex, or 32767 in decimal). The following should be entered in response to the prompting messages:

   a. HIGH/LOW MEMORY ALLOCATION(H/L)? H
   b. ENTER STARTING ADDRESS? 32767

   The response to (b) could have been expressed in hex as 7FFFH. Note that the "H" appears after the number 7FFF.

5. The response to the following prompting message for tape users should be "T":

   a. DISK/TAPE INPUT(D/T)? T

6. The tape cassettee should already be in the correct position after Step(2) above. This is the start of the MREL module. The user should press the ENTER key in response to the following:

   a. READY CASSETTE

   After MREL has been scanned (and @@MSHP selected by IBLOAD), the program will list a series of entries not yet found. User specified modules can be identified by two '@@' symbols, all others are system entries that will be resolved in XREL. In this example @@SRTV and @@SRR$ will be listed as user entries.

The user will be prompted twice more with a "READY CASSETTE". The user should again respond by pressing the enter key. The tape should be in the appropriate location for the remaining SREL and XREL modules.

8. After Step(7) has been completed the system will display the "MEMORY" usage values in a single line as shown below:

```
MEMORY  START=X'ssss',END=X'eeee',
        TRA=X'402D',DEFUSR=X'dddd'
```

where:

ssss = Starting hex location of load module.
eeee = Ending hex location of load module.
402D = DOS return (Not used for tape).
dddd = Starting hex execution address.

The values of 'ssss' and 'eeee' should be within the desired "protected memory". Memory size must be protected before using

the load module. If the values are incorrect the above steps may need to be repeated, specifying different memory parameters.

The value of 'dddd' will automatically be placed at the USR transfer location 16526 when the load module is loaded.

9. The next prompting message in this example will be:

```
DUMP MEMORY TO TAPE(Y/N) ? Y
```

The response 'Y' will initiate the dumping of the created load module to tape. This step is not necessarily needed since the memory already contains the desired load module. However, the load module will be lost when the power is turned off, or "protected memory" is otherwise altered. It is recommended that the 'Y' response be used. In this case a fresh casssette tape should be loaded, and the ENTER key pressed in response to the "READY CASSETTE" message.

10. The load module above can be reloaded by executing the following sequence:

a. Set or reset memory size as required.

b. Load the cassette with the load module tape.

c. Enter the following:

```
SYSTEM
IB
/          (just a '/' followed by ENTER)
?USR(1)
```

The above process will load the users load module and initialize INFINITE BASIC for operation. The print of USR(1) should return '1' indicating successful initialization. The USR transfer address may be redefined as required, since INFINITE BASIC only uses the USR function for initialization purposes.

E. Disk Example.

The steps required for creating a load module in a disk based systems is essentially the same as for tape systems. The following discussion emphasizes the differences involved. The user should read Section(D) above before proceeding with this section.

The following steps are required to build a load module from disk for the three functions in Section(B) above:

1. Load the disk containing IBLOAD, MREL, SREL, and XREL. Refer to the section on LOADING INFINITE BASIC TO DISK for additional detail on these files.

2. Execute the IBLOAD program in DOS mode by entering:

   a. IBLOAD

3. Enter the function names desired as described in Step(3) of the tape instructions.

4. Enter the memory specification parameters as described in Step(4) of the tape instructions.

5. The response to the following prompting message for disk users should be "D":

   a. DISK/TAPE INPUT(D/T)? D

6. IBLOAD will request the name of the module to be scanned. In this example MREL is the first file to be scanned. The user should respond to the filespec prompting messaage as follows:

a. ENTER INPUT DISK FILESPEC? MREL

After MREL has been scanned (and @@MSHP selected by IBLOAD), the program will list a series of entries not yet found. User specified modules can be identified by two '@@' symbols, all others are system entries that will be resolved in XREL. In this example, @@SRTV and @@SRR$ will be listed as user entries.

7. The user will be prompted twice more with a "ENTER INPUT DISK FILESPEC". The user should respond by entering SREL and XREL, which contain the remaining modules requested by the user.

8. Refer to in Step(8) of the tape instructions for a discussion of the MEMORY usage values. The value 'dddd' in this case, however, will be used in a DEFUSR statement to initialize INFINITE BASIC.

9. Refer to Step(9) of the tape instructions for a discussion of dumping the load module just created to tape. An alternate, and most useful technique, is to dump the load module directly to disk using the DUMP command. The exact arguments for the DUMP command are displayed for the disk user. Replace the word 'MEMORY' with 'DUMP filespec', where filespec is the name of the file to contain the load module.

10. The load module will be in memory after the completion of Step(9) above, or it can be reloaded from tape or disk using the SYSTEM command(see tape instructions - Step(10)) or the DOS LOAD command. To initialize INFINITE BASIC the user must first enter BASIC in the normal fashion, secifying the MEMORY SIZE parameter. The following sequence should then be executed:

```
a. DEFUSR=&Hdddd        (see Step(8) for dddd)
b. ?USR(1)
```

This needs to be done only after reloading BASIC, although it can appear at the front of every program run. The DEFUSR value can also be redefined to some other value, since INFINITE BASIC uses the USR function only for initialization purposes.

INTRODUCTION

This section describes the use of the MATRIX and STRING functions distributed with INFINITE BASIC. The first part of this section provides general information for each logical group of functions. Contained in the last part of this section are detailed formal descriptions of each INFINITE BASIC function.

The General Information Manual contains information required for an understanding of the material presented in this manual. In particular, the user should fully understand the section "Using Infinite Basic" before preceeding with the discussion below.

The discussion below is divided into the MATRIX and STRING sections. The first group(Matrix Arithmetic Functions) provides a complete BASIC program example, including the IBLOAD sequence. Subsequent groups include just the BASIC examples needed to illustrate a typical usage of the functions involved.

The examples shown each have a DEFUSR=&Hxxxx command at Line #5, where xxxx is the value supplied by the IBLOAD program. The example for the first group shows a specific value for xxxx.

MATRIX FUNCTIONS — GENERAL DESCRIPTION

A. Matrix Arithmetic Functions(In order by index)
MAAD,MADV,MAMP,MCPY,MASB

1. Description - This group of functions perform the general operation:

A = A # B or A = B

where the # is the operator +, -, *, or /. The copy operation is also provided . The arithmetic and copy operations are performed element by element in order by corresponding index values.

Multi-dimensional arrays may be specified as arguments to these functions, along with maximum index values to be used. Furthermore, the number of dimensions of A and B need not be the same. Corresponding elements(by index) are involved in the operation, consistent with the actual dimensions of A, B, and the maximums specified by the user in the function. Maximum index values of "0" are assumed for missing dimensions in A or B, as shown in the examples below:

DIM(A)	DIM(B)	FUNCTION	MAXIMUM USED
1,2,3	2,3		1,2,0
3,4	1,6,3		1,4,0
3,4	1,6,3	2,3	2,3,0
3,4	1,6,3	8,8,8	1,4,0
3,4	2,3	4,4	3,3

All elements beyond the maximum index values are not used or changed during the operation.

The arithmetic operations are performed element by element. The * and / operations are scalar operations, NOT the al mathematical operations of matrix multiply or inverse(see MMPY and MINV for the mathematical functions).

2. Program Example - The example below illustrates the techniques required to perform the operation:

```
CD = (A + B) / C
```

where A, B, C, and CD are multi-dimensioned arrays. The BASIC program is shown below.

```
5    DEFUSR=&HE898: PRINT USR(1)
10   DIM A(1,2,3),B(3,2,1),C(3,3,3),CD(2,3)
20   FOR I=0 TO 1: FOR J=0 TO 2: FOR K=0 TO 3
30      A(I,J,K)=1                'INIT MATRIX A
40      B(K,J,I)=3                'INIT MATRIX B
50   NEXT: NEXT: NEXT
60   FOR I=0 TO 3: FOR J=0 TO 3: FOR K=0 TO 3
70     C(I,J,K)=2                  'INIT MATRIX C
      EXT: NEXT: NEXT
90   IR=&MCPY(CD,A):GOSUB 200     'COPY MATRIX A TO CD
100  IR=&MAAD(CD,B):GOSUB 200     'ADD MATRIX B TO CD
110  IR=&MADV(CD,C,2,1):GOSUB 200 'DIVIDE CD BY B
120  END
200  FOR I=0 TO 2: FOR J=0 TO 3
210     PRINT CD(I,J);
220  NEXT: PRINT "  ";: NEXT: PRINT
230  RETURN
```

Note in the above example different array sizes are used. Furthermore, the final divide step specifies that the maximum element sizes (2,1) are to be used. Lines #90-110 could be replaced by the more convenient form:

```
90      IR=&MCPY(CD,A) OR &MAAD(CD,B) OR
        &MADV(CD,C,2,2)
```

where the operations are performed left to right. For efficiency, the minimum subscript case(2,1) should have been forced on the first step, ie:

```
90      IR=&MCPY(CD,A,2,2) OR &MAAD(CD,B) OR
        &MADV(CD,C)
```

This produces the same result for the elements thru (2,2). Non-zero elements would appear in other elements of CD only in the first two forms above.

3. IBLOAD Example - Given below is the sequence required to build and load the INFINITE BASIC load module, and to execute the example BASIC program. The "Loader Description" section of the INFINITE BASIC General Description Manual provides step-by-step instructions for using IBLOAD with Tape(section D) or Disk(section E). Given below is the input and resulting outputs for the matrix example using both Tape and Disk systems. Step numbers noted in the comments correspond exactly the those givef in D and E.

a. Tape Sequence

```
>SYSTEM                                        (Step #'s 1-2)
*? IBLOAD
*? /
IBLOAD - TAPE VERSION 1.0 - COPYRIGHT 1979,
                  RACET COMPUTES
ENTER SUBROUTINE NAMES REQUIRED? @@MAAD
? @@MADV                                          (Step #3)
? @@MCPY
?
HIGH/LOW MEMORY ALLOCATION(H/L)? H                (Step #4)
ENTER STARTING ADDRESS? 7FFFH
DISK/TAPE INPUT(D/T)? T                           (Step #5)
READY CASSETTE                                    (Step #6)
OK OK OK OK OK OK OK OK OK
FOLLOWING SUBROUTINE(S) NOT YET FOUND
@AGETX  @AINCD  @AINTN  @APSHX  @APUTX
@APUTY  @ARGN   @ARGV   @ASUB
READY CASSETTE                                    (Step #7)
OK OK OK OK OK
MEMORY   START=X'7889',END=X'7FFF',TRA=X'402D'
                  DEFUSR=&H7897                   (Step #8)
DUMP MEMORY TO TAPE(Y/N)? Y                       (Step #9)
READY CASSETTE
XD   TAPE DUMP COMPLETED OK
XE   PROCESSING COMPLETED
READY
>SYSTEM                                          (Step #10)
*? IB
*? /
READY
>CLEAR
>CLOAD"A"                  (Load matrix example above)
READY
>5 PRINT USR(1)            (Don't use DEFUSR for tape)
RUN
INFINITE BASIC - VERSION 1.0 - COPYRIGHT 1979,
                  RACET COMPUTES
OK
 1  1  1  0    1  1  1  1    0  0  0  0
 4  4  4  0    4  4  4  0    3  3  3  0
 2  2  4  0    2  2  4  0    2  2  3  0
READY
>RUN
 1  1  1  0    1  1  1  1    0  0  0  0
 4  4  4  0    4  4  4  0    3  3  3  0
 2  2  4  0    2  2  4  0    2  2  3  0
READY
```

b. Disk Sequence

```
DOS READY                                         (Step #1)
IBLOAD                                            (Step #2)
IBLOAD - DISK VERSION 2.0 - COPYRIGHT 1979,
                  RACET COMPUTES
ENTER SUBROUTINE NAMES REQUIRED? @@MAAD
? @@MADV                                          (Step #3)
? @@MCPY
?
```

```
HIGH/LOW MEMORY ALLOCATION(H/L)? H          (Step #4)
ENTER STARTING ADDRESS? F000H
DISK/TAPE INPUT(D/T)? D                     (Step #5)
ENTER INPUT DISK FILESPEC? MREL             (Step #6)
FOLLOWING SUBROUTINES(S) NOT YET FOUND
@AGETX  @AINCD  @AINTN  @APSHX  @APUTX
@APUTY  @ARGN   @ARGV   @ASUB
ENTER INPUT DISK FILESPEC? XREL             (Step #7)
MEMORY   START=X'E88A',END=X'F000',TRA=X'402D'
               DEFUSR=&HE898
DUMP MEMORY TO TAPE(Y/N) N                  (Step #9)
XE   PROCESSING COMPLETED

DOS READY
DUMP MEX1/CMD:1 (START=X'E88A',END=-X'F000',
               TRA=X'402D')

DOS READY
MEX1                                        (STEP #10)
BASIC
HOW MANY FILES?
MEMORY SIZE 59500
RADIO SHACK DISK BASIC VERSION 1.1
READY
>LOAD "MEX1/BAS"
READY
>RUN
INFINITE BASIC - VERSION 1.0 - COPYRIGHT 1979
               RACET COMPUTES
OK
 1  1  1  0    1  1  1  0    0  0  0  0
 4  4  4  0    4  4  4  0    3  3  3  0
 2  2  4  0    2  2  4  0    2  3  3  0
READY
```

Note that in both examples above only the MREL and XREL files were scanned. For tape users this requires positioning the tape before responding to "READY CASSETTE". The exact location on tape can be determined by noting the tape counter while doing a full scan. It was not necessary to perform the load to memory operation(Step #10) in the examples above. Subsequent uses of the created load module would require the load as indicated(after a power off for example).

To determine memory size, the START parameter must be converted from hexadecimal to decimal. In the tape example above(Step #8) the resulting load module started at X'7889' which is expressed in hexadecimal. Similarly, the disk example(Step #7) indicates a start at X'E898'. BASIC requires a decimal value when being initialized. To convert, perform the following multiplication sequence:

```
 7*4096 + 8*256 + 8*16 +9  (30857 is the memory size)
14*4096 + 8*256 + 8*26 +10 (59530 is the memory size)
```

Remember that in hexadecimal, A B C D E F are 10 11 12 13 14 and 15.

The DEFUSR statement is not required when using the tape sequence, as shown in Line #5. The PRINT USR(1) will initialize INFINITE BASIC. Note that the first time the PRINT USR(1) is issued a full message is produced, as illustrated in the Tape example.

B. Matrix Arithmetic Functions(Sequential, B Repeated)

1. Description - This group of functions performs the general operation:

A = A # B or A = B

where the # is the operator +, -, *, or /. The copy operation is also provided. The arithmetic and copy operations are performed element by element in sequential order. The arrays A and B are processed as if they were singly dimensioned arrays, even if multi-dimensional. The elements of B are recycled until A is full or "N" elements have been processed.

Multi-dimensional arrays are stored in memory by BASIC with the first index value varying fastest. Consider the array B(I,J) with a DIM B(1,2):

Array B(I,J)		J--> 0	1	2
	0	1	3	5
I =	1	2	4	6

The data will be processed element-by-element as 1, 2, 3, 4, 5, 6, 1, 2, 3, ...etc. Similiarly, the Array A is processed with the first subscript varying fastest.

2. Program Example - The example below illustrates the techniques required to perform the operation:

CD = (A + B) / C

where A, B, C, are multi-dimensional arrays Of the same sizes, and CD is a singly dimensioned array. The BASIC program is shown below.

```
5    DEFUSR=&Hxxxx: PRINT USR(1)
10   DIM A(1,2),B(1,2),C(1,2),CD(14)
20   N=1
30   FOR J=0 TO 2: FOR I=0 TO 1
40     A(I,J)=N: B(I,J)=N: C(I,J)=N: N=N+1
50   NEXT: NEXT
60   IR=&MELC(CD,A): GOSUB 200  'COPY A TO CD
70   IR=&MELA(CD,B): GOSUB 200  'ADD B TO CD
80   IR=&MELD(CD,C): GOSUB 200  'DIVIDE CD BY C
90   END
200  FOR I=0 TO 14
210    PRINT CD(I);
220  NEXT: PRINT
230  RETURN
```

The results from running the above program are shown below.

```
>RUN
INFINITE BASIC - VERSION 1.0 - COPYRIGHT 1979,
                 RACET COMPUTES
OK
1  2  3  4  5  6  1  2  3  4  5  6  1  2  3
2  4  6  8 10 12  2  4  6  8 10 12  2  4  6
2  2  2  2  2  2  2  2  2  2  2  2  2  2  2
READY
```

Note that the arrays A, B, and C repeat 2 1/2 times to fill the array CD.

C. Matrix Scalar Arithmetic Functions
MSAD,MSMP,MSSB,MSDV

1. Description - This group of functions performs the general operation:

```
A = A # S
```

where # is the operator +, -, *, or /, and S is a scalar value. The same scalar "S" is applied to each element of A, or until N elements have been processed. The array A is procesed as if it were a singly dimensioned array, even if it is multi-dimensional. The first subscript varies fastest similar to the sequential matrix arithmetic functions(B above).

2. Program Example - Given below is an example of adding the scalar "2" to each element of a multi-dimensioned array A.

```
5   DEFUSR=&Hxxxx: PRINT USR(1)
10  DIM A(1,2,3)
20  N=0
30  FOR K=0 TO 3: FOR J=0 TO 2: FOR I=0 TO 1
40    A(I,J,K)=N: N=N+1
50  NEXT: NEXT: NEXT
60  IR=&MSAD(A,2)
70  FOR K=0 TO 3: PRINT "K=";K: FOR I=0 TO 1
80    FOR J=0 TO 2: PRINT A(I,J,K);: NEXT: PRINT
90  NEXT: NEXT:
100 END
```

The results from running the above program are shown below:

```
>RUN
INFINITE BASIC - VERSION 1.0 - COPYRIGHT 1979
                 RACET COMPUTES
OK
K=0
 2    4    6
 3    5    7
K=1
 8    10   12
 9    11   13
K=2
 14   16   18
 15   17   19
K=3
 20   22   24
 21   23   25
READY
```

D. Matrix Equal Function
MEQU

1. Description - This group consists of the single function MEQU. It is similar to MELC in that it copies elements from an array B to an array A in sequential order(See group(B) above). However, this routine differes from MELC in the following respects:

a. The elements of B are not repeated during the process. If B is smaller than A the ending elements of A will be unchanged. If B is larger than A then ending elements of B will not be copied.

b. Arrays A and B must both be of the same mode, either Integer, Single, or Double precision.

c. The processing speed of this function is faster than the corresponding MELC. It also requires less memory.

2. Program Example - The example below illustrates copying the multi-dimensioned array B into a larger array A.

```
5   DEFUSR=&Hxxxx: PRINT USR(1)
10  DIM A(30),B(1,2,3)
20  N=1
30  FOR K=0 TO 3: FOR J=0 TO 2: FOR I=0 TO 1
40    B(I,J,K)=N: N=N+1
50  NEXT: NEXT: NEXT
60  IR=&MEQU(A,B,22)
70  FOR I=0 TO 30
80    PRINT A(I);
90  NEXT: PRINT
100 END
```

Running the above program produces the following results:

```
>RUN
INFINITE BASIC - VERSION 1.0 - COPYRIGHT 1979,
                 RACET COMPUTES
1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
17 18 19 20 21 22 0  0  0  0  0  0  0  0  0
READY
```

In the above example there are 24 available elements in B. Of these 24 only 22 are copied as requested in Line #60.

E. Matrix Mathematical Functions
MIDT, MINV, MMPY, MTRN, MEQN

1. Description - This group of functions perform the formal mathematical operations of Identity, Inverse, Multiply, Transpose, and Simultaneous Equations solution. Functions MINV, MMPY, and MEQN are restricted to two-dimensional single or double precision arrays. MIDT and MTRN may be used with multi-dimensional arrays. Additional information on these functions can be obtained from standard mathematical texts.

2. Program Example - The example below illustrates the use MINV, MMPY, and MEQN. Assume that the set of equations below are to be solved:

```
2*X1 + 3*X2 +4*X3 = 5
3*X1 + 2*X2 +6*X3 = 7
8*X1 + 4*X2 +1*X3 = 3
```

This is expressed in the standard mathematical notation as:

```
A * X = B
```

where

```
        2  3  4          X1          5
A  =    3  2  6     X =  X2     B =  7
        8  4  1          X3          3
```

Two methods are possible to solve the set of equations using INFINITE BASIC. MINV and MMPY can be used to perform the operation of Inverse(A) * B to give X, or MEQU can be used directly to produce the desired result. The example below illustrates both methods.

```
5   DEFUSR=&Hxxxx: PRINT USR(1)
10  DIM A(10,10),AA(10,10),AI(10,10)
20  DIM B(10,0),X(10,0),BB(10)
30  DATA 2,3,4,5,3,2,6,7,8,4,1,3
40  FOR I=0 TO 2                    'READ ARRAYS A & B
50    FOR J=0 TO 2: READ A(I,J): NEXT
60    READ B(I)
70  NEXT
80  IR=&MEQU(AA,A): IR=&MEQU(BB,B)  'SAVE FOR MEQN
90  IR=&MINV(AA,AI,3)               'INVERT AA --> AI
100 IR=&MMPY(AI,B,X,2,2,0)          'MULTIPLY AI * B --> X
110 FOR I=0 TO 2: PRINT X(I,0);: NEXT: PRINT
120 IR=&MEQN(A,BB,3)                'USE MEQN TO SOLVE
130 FOR I=0 TO 2: PRINT BB(I);: NEXT: PRINT
140 END
```

The results from running the above program are shown below:

```
>RUN
INFINITE BASIC - VERSION 1.0 - COPYRIGHT 1979,
                 RACET COMPUTES
OK
 .146667    .2    1.02667
 .146667    .2    1.02667
READY
```

In the above example the correct choice of number of dimensions of X and B is critical. MMPY and MINV require two-dimensional arrays for B and X. MEQN requires a singly dimensioned array. Both MINV and MEQN destroy the matrix being inverted or solved. Line #80 was required to save Matrix A for use in Line #120. Additional saves would be required if A was need in later steps. The arrays specified above were dimensioned much larger than required for illustration purposes only. The value "10" could be replaced by "2" for this simple example.

Also note that the arguments used in MMPY are the highest index values rather than the number of elements.

The use of MEQN is recommended for solving simple equations. This function requires less memory, fewer user instructions, and less computer time than MINV/MMPY. However, MINV can be expected to be more accurate for larger or more sensitive systems of equations.

F. Matrix Input/Output Functions
MELR, MRST, MRDT, MWRT

1. Description - MELR is used to read data into a matrix from BASIC "DATA" statements. MRST is used to position BASIC to start at a specific DATA statement. MRDT and MWRT are used to input and output data to cassette tape.

MELR correspondes to the BASIC "READ" statement for matrices. This allows the user to conveniently initialize matrices. The MRST function performs a function similar to "RESTORE", but can be used to position to any DATA statement. This can be used for normal READ statements as well as with MELR.

MRDT and MWRT provide an efficient means of using cassettee tape for large amounts of data. These functions provide the capability of reading and writing entire blocks of data. In addition, block checksums are used to validate that the data read is correct. Block identification numbers are also provided to allow automatic selection of data to be read.

Both MRDT and MWRT process matrices sequentially similar to the arithmetic functions(See B, C, and D above). Multi dimensional arrays will be processed with the first index varying fastest. The number of dimensions and sizes of arrays can be different for a particular MWRT and subsequent MRDT. The user must, however, must be aware that positioning of data in multi-dimensional arrays may be different under these circumstances. MRDT also skips remaining data on tape that exceeds the size of the array being used.

The user must insure that the mode of the data being read from tape is identical to that originally written to tape. An Integer array written to tape must be read back into another Integer array. This also applies to Single Precision, Double Precision, and String arrays.

2. Program Example - The example below illustrates initializing several matrices using MELR and MRST, and then writing and subsequent reading of the data to/from tape. Single dimensioned arrays are used in the examples, although multi-dimensioned arrays could also be used as described above.

```
5   DEFUSR=&Hxxxx: PRINT USR(1): CLEAR 1000
10  DIM IA%(10),SP!(10),DP#(10),CH$(10)
20  0,1,2,3,4,5,6,7,8,9,10
30  0.0,1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9,10.10
40  0.0,1.0D01,2.0D02,3.0D03,4.0D04,5.0D05
    6.0D06,7.0D07,8.0D08,9.0D09,10.0D10
```

```
50  DATA "STRING #0", "STRING #1","STRING #2",
    "STRING #3","#4","","TEST #6","STRING #7",
    "TEST #8","TEST #9","TEST #10"
60  IR=&MELR(IA%)                 'READ IA% ARRAY
70  IR=&MRST(50)                  'POSITION TO STRINGS
80  IR=&MELR(CH$)                 'READ STRING ARRAY
90  IR=&MRST(30)                  'POSITION SP #'S
100 IR=&MELR(SP!)                 'READ SP #'S
110 IR=&MELR(DP#)                 'READ DP #'S
120 CMD "T"                       'TURN OFF CLOCK
130 IR=&MWRT(CH$,11,1)            'WRITE CH$ TO TAPE
140 IR=&MWRT(DP#,11,2)            'WRITE DP# TO TAPE
150 IR=&MWRT(SP!,11,3)            'WRITE SP! TO TAPE
160 IR=&MWRT(IA%,11,4)            'WRITE IA% TO TAPE
170 INPUT "REWIND TAPE, PRESS ENTER";A$
180 IR=&MRDT(CH$,11,1)            'READ CH$ FROM TAPE
190 IR=&MRDT(IA%,5,4)             'READ IA% FROM TAPE
200 INPUT "REWIND TAPE, PRESS ENTER";A$
210 IR=&MRDT(DP#,23,2)            'READ DP# FROM TAPE
220 IR=&MRDT(SP!,11,3)            'READ SP! FROM TAPE
230 CMD"R"                        'TURN CLOCK BACK ON
240 END
```

The results from running the above program are shown below:

```
>RUN
INFINITE BASIC - VERSION 1.0 - COPYRIGHT 1979,
                 RACET COMPUTES,
OK
REWIND TAPE, PRESS ENTER?
REWIND TAPE, PRESS ENTER?
READY
```

The MELR Function was used above to initialize the arrays. The arrays were read out of sequence by use of the MRST function, as shown in Lines #70 and #90.

The MWRT function was used to write the four arrays to tape. Block numbers #1-4 were assigned to each array written to tape. The use of block numbers allows automatic selection of the correct block by MRDT. In this example, block #4 was read after block #1, automatically skipping blocks 2-3 as shown in Line #190. This same statement also illustrates reading only the first 5 elements. Line #210 specifies more elements(23) to be read than are available. The return value(IR) would indicate only 11 were actually read.

G. Matrix Shape Function MSHP

1. Description - The matrix shape function allows the user to dynamically modify the size and number of dimensions of any array. Arrays may also be deleted from memory, freeing space for other use.

The total size of an array can be either increased or decreased. If the array size is increased only the added elements are initialized to zero, all others remain the same. Data elements will be permanently erased if the array size is decreased.

The array processed by MSHP overlays the original array. The data in the resulting array is NOT rearranged to conform to the new dimensions. The user should be aware that data is stored in memory with the first index varying fastest, as discussed in the sequential matrix functions(See B, C, and D above).

2. Program Example - The example below illustrates initializing a single dimensioned array , reshaping it to a two dimensional array for processing, and then deleting the array.

```
5   DEFUSR=&Hxxxx: PRINT USR(1)
10  DIM A(25)
20  FOR I=0 TO 25                'INITIALIZE A
30    A(I)=I
40  NEXT
50  IR=&MSHP(A,4,5)              'RESHAPE A
60  FOR I=0 TO 4                 'PRINT ARRAY
70    FOR J=0 TO 5: PRINT A(I,J);: NEXT: PRINT
80  NEXT
90  IR=&MSHP(A)                  'DELETE ARRAY A
100 END
```

The result from running the above program is shown below:.

```
>RUN
INFINITE BASIC VERSION 1.0 - COPYRIGHT 1979,
                  RACET COMPUTES
 0   5   10  15  20  25
 1   6   11  16  21  0
 2   7   12  17  22  0
 3   8   13  18  23  0
 4   9   14  19  24  0
READY
```

Note that the last four elements are zero since the new array was larger than the original array.

H. Matrix Call and Return Functions
MCAL, MRET

1. Description - This pair of functions allows the user to create user written BASIC subroutines with generalized calling arguments. The GOSUB command of BASIC allows the user to create a subroutine that can be called from different locations. Different variables may, however, not be processed directly by this technique. For example, a subroutine to sum the contents of an array must always have the data in the same array.

MCAL allows the user to temporarily specify variable names corresponding to those used in a subroutine. MRET restores the variable names to the original definitions. Scalar constants, scalar variables, and arrays may be passed as arguments to MCAL.

2. Program Example - The example below illustrates the use of a subroutine to calculate the sum and average of an array and to calculate percentages of each element. This subroutine is used with two different arrays.

```
5   DEFUSR=&Hxxxx: PRINT USR(1)
10  DIM A(10),B(10): N=0
20  FOR I=0 TO 10                  'INITIALIZE A & B
30    A(I)=I: B(I)=I+10
40  NEXT
50  P$="A,10": GOSUB 100           'CALL SUB200(A,10)
60  PRINT "AVERAGE=";AV
70  P$="B,5": GOSUB 100            'CALL SUB200(B,5)
80  PRINT "AVERAGE=";AV
90  END
100 Q$="&C,N,AV": IR=&MCAL(P$,Q$) 'RECEIVE ARGUMENTS
110 S=0.0                          'CALCULATE SUM
120 FOR I=0 TO N
130   S=S+C(I)
140 NEXT
150 AV=S/(N+1)                     'CALCULATE AVERAGE
160 FOR I=0 TO N                   'CALCULATE %
170   C(I)=100.0*C(I)/S
180   PRINT C(I);                  'PRINT %
190 NEXT: PRINT
200 IR=&MRTN(P$,Q$)                'RESTORE ARGUMENTS
210 RETURN                         'RETURN TO CALLER
```

The results from running the above program are shown below:

```
>RUN
INFINITE BASIC - VERSION 1.0 - COPYRIGHT 1979,
                 RACET COMPUTES
OK
 0   1.81818   3.63636   5.45455   7.27273   9.09091
 10.90901   12.7273   14.5455   16.3636   18.1818
AVERAGE= 5
 13.3333   14.6667   16   17.3333   18.6667   20
AVERAGE= 12.5
READY
```

In the above example note that array C was not dimensioned. It is a "dummy" array used only in the subroutine. The actual arrays used in place of C are A and B. Scalars passed as constants must have their "dummy" variable names initialized, as shown in Line #10. A scalar variables, such as "AA" and "AB" above, do not need to be initialized. Only one of the parameter specification strings (P$ or Q$ above) need explicitly indicate which variables are arrays. This is done by preceeding the array variable name with an "&", as shown in Line # 100.

I. Memory W· Fetch/Store Functions
PLUK, PLUG

1. Description - These two funtions allow the user to fetch or store a two byte integer to memory. This is similar to the PEEK and POKE which allow fetch and store of a single byte. The PLUK function simply fetches the two byte integer from a specified memory address. PLUG simultaneously fetches the old contents at a specified memory address and stores a new word at that address. Both functions assume data is in the standard low order byte first format.

2. Program Example - The example below shows how to replace the screen DCB pointer with the printer DCB pointer. This allows everything normally displayed on the screen to be diverted to a line printer. The pointer is restored after a single line has been printed.

```
5   DEFUSR=&Hxxxx: PRINT USR(1)
10  DEFINT I
20  I1=&PLUK(16422)              'FIND PRINTER PTR
30  I2=&PLUG(16414,I1)           'REPLACE SCREEN PTR
40  PRINT "THIS LINE GOES TO THE PRINTER"
50  I1=&PLUG(16414,I2)           'RESTORE SCREEN PTR
60  END
```

The results from running the above program are shown below:

```
>RUN
INFINITE BASIC - VERSION 1.0 - COPYRIGHT 1979,
                 RACET COMPUTES
OK
THIS LINE GOES TO THE PRINTER
READY
```

The PLUK and PLUG in Lines #20-30 could have been replaced by the single expression I2=&PLUG(16414,&PLUK(16422)).

STRING FUNCTIONS — GENERAL DESCRIPTION

A. String Manipulation Functions
SBJ, SDS, SIV, SLJ, SLR, SLS, SLT, SRJ
SRR, SRS, SRT, SSI, STC, STJ, STP

1. Description - These string functions are used to transform existing character strings into new forms. In general, one or more strings along with other parameters are arguments to each of these functions. The return value of each function is the resulting desired character string. All of these functions create new character strings, and do not modify the original character string arguments.

Several of the functions utilize a "Skip" character argument denoted by "X$". This is used to indicate which character is to be skipped or removed during the indicated operation. For example, the function:

```
&SBJ$(S$ <,X$>)
```

eliminates the "Skip" character from both the left and right sides of the character string S$. The default value for X$ is a blank character. Only the first character of X$ is used as the skip character. The results from those functions using the skip character is as follows:

a. SLJ, SRJ - The justification functions rotate the string left or right to the first non-skip character. The resulting string is the same length as the original string.
b. SBJ, SLT, SRT - The truncation routines remove skip characters resulting in shorter strings.
c. STJ - The text justification function removes skip characters from both ends of the string, but inserts skip characters between words. The resulting total length is specified by the user.
d. STP - The text pack functions removes redundant skip characters from the ends of the string and between words, resuting in shorter strings.
e. SLS, SRS - The string shift functions shift the the string left or right a specified number of locations. Skip characters are added to the front or end of the string to maintain the same length string.

The remaining functions in this group do not use the skip character argument. The length may, however, be modified as indicated below:

a. SLR, SRR, SIV - The string left and right rotate functions shift the contents of the string, inserting the shifted out characters at the other end of the string. The string invert function reverses the order of the characters in a string. The total length of the string remains the same for all three of these functions.

b. SDS - The substring delete function eliminates characters from a string, resulting in a shorter length.
c. SSI - The substring insert function increases the length of a string.

2. Program Example - The example below illustrates all the functions in this group. The skip character "." is used where applicable in the examples for illustration purposes.

```
5   DEFUSR=&Hxxxx: PRINT USR(1): CLEAR 1000
10  X$="."
20  S$=".....ABC..DE.FGH...IJ...."
30  A$=S$:                  PRINT "S$";
40  GOSUB 500
50  A$=&SBJ$(S$,X$):        PRINT "&SBJ$(S$,X$)";
60  GOSUB 500
70  A$=&SDS$(S$,7,2):       PRINT "&SDS$(S$,7,2)";
80  GOSUB 500
90  A$=&SIV$(S$):           PRINT "&SIV$(S$)";
100 GOSUB 500
110 A$=&SLJ$(S$,X$):        PRINT "&SLJ$(S$,X$);
120 GOSUB 500
130 A$=&SLR$(S$,7):         PRINT "&SLR$(S$,7);
140 GOSUB 500
150 A$=&SLS$(S$,7,X$):      PRINT "&SLS$(S$,7,X$)";
160 GOSUB 500
170 A$=&SLT$(S$,X$):        PRINT "&SLT$(S$,X$)";
180 GOSUB 500
190 A$=&SRJ$(S$,X$):        PRINT "&SRJ$(S$,X$)";
200 GOSUB 500
210 A$=&SRR$(S$,7):         PRINT "&SRR$(S$,7)";
220 GOSUB 500
230 A$=&SRT$(S$,X$):        PRINT "&SRT$(S$,X$)";
240 GOSUB 500
250 A$=&SSI$(S$,"12345",7):
                            PRINT "&SSI$(S$,"12345",7)";
260 GOSUB 500
270 A$=&STC$(S$,20,X$):     PRINT "&STC$(S$,20,X$)";
280 GOSUB 500
290 A$=&STJ$(S$,20,X$):     PRINT "&STJ$(S$,20,X$)";
300 GOSUB 500
310 A$=&STP$(S$,X$):        PRINT "&STP$(S$,X$)";
320 GOSUB 500
330 END
500 PRINT "=";TAB(20);"-->";A$;"<--"
510 RETURN
```

The results from running the above program are shown below:

```
>RUN
INFINITE BASIC - VERSION 1.0 - COPYRIGHT 1979,
                RACET COMPUTES
OK
S$=               -->.....ABC..DE.FGH...IJ....<--
&SBJ$(S$,X$)=     -->ABC..DE.FGH...IJ<--
```

```
&SDS$(S$,7,2)=    -->.....A..DE.FGH...IJ....<--
&SIV$(S$)=        -->....JI...HGF.ED..CBA.....<--
&SLJ$(S$,X$)=     -->ABC..DE.FGH...IJ.........<--
&SLR$(S$,7)=      -->C..DE.FGH...IJ.........AB<--
&SLS$(S$,7,X$)=   -->C..DE.FGH...IJ...........<--
&SLT$(S$,X$)=     -->ABC..DE.FGH...IJ....<--
&SRJ$(S$,X$)=     -->.........ABC..DE.FGH...IJ<--
&SRR$(S$,7)=      -->.IJ.........ABC..DE.FGH..<--
&SRT$(S$,X$)=     -->.....ABC..DE.FGH...IJ<--
&SSI$(S$,"12345",7)=
                  -->.....AB12345C..DE.FGH...IJ....<--
&STC$(S$,20,X$)= -->..ABC..DE.FGH...IJ..<--
&STJ$(S$,20,X$)= -->ABC...DE..FGH.....IJ<--
&STP$(S$,X$)=     -->ABC.DE.FGH.IJ<--
READY
```

B. String Translation Functions
SCL, SCU, STL

1. Description - These functions perform character by character translation of one character string into another. The functions SCL and SCU translate characters between lower case and upper case character sets. The STL function allows the user to specify not only the character translation to be performed, but also which characters are to be translated.

The SCL and SCU functions translate characters as shown below:

Orig Dec/(Hex)	SCL Dec/(Hex)	SCU Dec/(Hex)
0 - 63 (00 - 3F)	0 - 62 (00 - 3F)	0 - 63 (00 - 3F)
64 - 95 (40 - 5F)	96 - 127 (60 - 7F)	64 - 95 (40 - 5F)
96 - 127 (60 - 7F)	96 - 127 (60 - 7F)	64 - 95 (40 - 5F)
128 - 255 (80 - FF)	128 - 255 (80 - FF)	128 - 255 (80 - FF)

The string translate function STL can be used to translate selected characters. The general format of this function is:

```
&STL$(S$,T$ <,U$>)
```

T$ contains a list of characters that, if found within S$, will be translated to the corresponding character in U$. If U$ is not specified it is assumed to be the character string containing all 256 characters (0-255) in sequence.

2. Example Program - The example program below illustrates the use of the above translate functions:

```
5   DEFUSR=&Hxxxx: PRINT USR(1): CLEAR 1000
10  S$="ABCD efgh IJKL mnop 1234 #$%&"
20  LPRINT "S$=";TAB(10);"-->";S$;"<--"
30  LPRINT "&SCU$(S$)";" -->";&SCU$(S$);"<--"
40  LPRINT "&SCL$(S$)";" -->";&SCL$(S$);"<--"
50  LPRINT STRING$(45,"-")
60  S$="ABCD": T$="YAZD": U$="1234": GOSUB 500
70  S$="ABCD": T$="WXYZ": U$="1234": GOSUB 500
80  S$="ABCD": T$="DCB":  U$="1234": GOSUB 500
90  S$="AB":   T$="ABYZ": U$="1234": GOSUB 500
100 S$="ABCD": T$="YAZD": U$="4321": GOSUB 500
110 END
500 LPRINT "S$=";S$;TAB(9);"T$=";T$;TAB(17);"U$=";U$
510 LPRINT TAB(25);"&STL$(S$,T$,U$)=";
          TAB(42);&STL$(S$,T$,U$)
520 LPRINT TAB(42);&SX2$(&STL$(S$,T$))
530 RETURN
```

The results from running the above program are shown below:

```
>RUN
INFINITE BASIC - VERSION 1.0 - COPYRIGHT 1979,
                 RACET COMPUTES
OK
S$=       -->ABCD efgh IJKL mnop 1234 #$%&<--
&SCU$(S$) -->ABCD EFGH IJKL MNOP 1234 #$%&<--
&SCL$(S$) -->abcd efgh ijkl mnop 1234 #$%&<--
------------------------------------------
S$=ABCD T$=YAZD U$=1234 &STL$(S$,T$,U$)= 2BC4
                        &STL$(S$,T$)=    01 42 42 03
S$=ABCD T$=WXYZ U$=1234 &STL$(S$,T$,U$)= ABCD
                        &STL$(S$,T$)=    41 42 43 44
S$=ABCD T$=DCB  U$=1234 &STL$(S$,T$,U$)= A321
                        &STL$(S$,T$)=    41 02 01 00
S$=AB   T$=ABYZ U$=1234 &STL$(S$,T$,U )= 12
                        &STL$(S$,T$)=    00 01
S$=ABCD T$=YAZD U$=4321 &STL$(S$,T$,U$)= 3BC1
                        &STL$(S$,T$)=    01 42 43 03
READY
```

The character string in Line #10 of the BASIC program is shown in upper/lower case. Normally BASIC displays all keyboard input in upper case - even if the shift keys are used.

The first part of the example illustrates the upper/lower translation. In this example upper case characters are translated to lower case by SCL, and lower case to upper case by SCU. All other characters remain unchanged.

The second part of the example show the use of the STL function. The results of the STL function where U$ is not supplied is shown printed in Hex. This is accomplished by the use of function SX2$(See H. below). This was necessary since the default U$ contains the low ASCII codes 0-3 which are special non-printing characters(Note: A=41, B=42, C=43, and D=44 in Hex).

T$ and U$ in the first example of STL indicate the following translation is to be performed on S$:

From T$	To U$
Y	1
A	2
Z	3
D	4
Others	Unchanged

For S$="ABCD" only "A" and "D" are in the above translation table, giving the result as "2BC4". The other examples illustrate additional combinations.

C. String and Data Compression Functions
SC4, SC5, SC6, SC7, SCP, SCPM
SD4, SD5, SD6, SD7, SDP, SDPM

1. Description - The purpose of these functions is reduce the data storage requirements for certain types of information. The "C" mode routines compress data, while the corresponding "D" routines decompress the data back to the original format.

The SC4-SC7 and corresponding SD4-SD7 series compresses data by packing the data in groups smaller than the normal 8-bit byte. This requires that the data conform to certain character ranges. For example, SC4/SD4 assumes that only the characters "0","1", ... "9"," ","+","-",".","D","E" are in the strings to be compressed(numeric data). Any other characters in this case will be translated to blanks.

The SCP/SDP and SCPM/SDPM functions pack data by special encoding of repeated characters. The SCP/SDP are used with character strings. SCPM/SDPM work directly with data in memory.

The user should be aware that the compressed data may contain ASCII control codes. This prohibits use of sequential Input/Output for storing the compressed data on disk. Random I/O techniques, however, can still be used. Furthermore, compressing small strings may actually produce larger "compressed" strings. This is due to the fact that a certain amount of overhead is required for the compression codes.

2. Program Example - Given below are examples of using the compression/decompression routines.

```
5   DEFUSR=&Hxxxx: PRINT USR(1): CLEAR 1000
10  DEFINT I: DIM IA(600)
20  S1$="0123456789"
30  S2$=" +-.DE"
40  S3$="ABCFGHIJKLMNOPQRSTUVWXYZ"
50  S4$="&',?"
60  S5$="!#$%()*=@;><,"
70  S$=S1$+S2$+S3$+S4$+S5$
80  C$=&SC5$(S$): D$=&SD5$(C$): PRINT "SC5/SD5";:
    GOSUB 500
```

```
90  C$=&SC6$(S$):  D$=&SD6$(C$):  PRINT "SC6/SD6";:
    GOSUB 500
100 C$=&SC7$(S$):  D$=&SD  (C$):  PRINT "SC7/SD7";:
    GOSUB 500
110 '
120 S$="THIS CONTAINS      IMBEDDED      BLANKS      OR
     MULTIPLE ************ OTHER    $$$$$$$$
     CHARACTERS"
130  C$=&SCP$(S$):  D$=&SDP$(C$):  PRINT "SCP/SDP";:
     GOSUB 500
140  '
150  IR=&SCPM(15360,VARPTR(IA(0)),1024)
160  CLS
170  PRINT "SCPM/SDPM";TAB(15);"LEN(SCREEN)= 1024
     LEN(IA)=";IR
180  PRINT "PRESS ENTER TO RESTORE SCREEN FROM IA"
190  INPUT "AND AGAIN TO CONTINUE";A$
200  IS=&SDPM(VARPTR(IA(0)),15360,1024)
210  IF INKEY$="" THEN 230  ELSE CLS
220  END
230  '
500  PRINT TAB(17);"LEN(S$)=";LEN(S$);" LEN(C$)=";
     LEN(C$);" LEN(D$)=";LEN(D$)
510  PRINT "S$=";S$
520  PRINT "D$=";D$
530  PRINT STRING$(61,"-")
540  RETURN
```

The results from running Lines 5-190 are shown below. The screen display resulting from Line #200 is exactly the same as after executing Lines 5-190.

```
>RUN
INFINITE BASIC - VERSION 1.0 - COPYRIGHT 1979, RACET COMPUTES
OK
SC4/SD4          LEN(S$)= 58 LEN(C$)= 30 LEN(D$)= 58
C$=0123456789 +-.DEABCFGHIJKLMNOPQRSTUVWXYZ&',?!#$%()*=@;><,
D$=0123456789 +-.DE
-------------------------------------------------------------
SC5/SD5          LEN(S$)= 58 LEN(C$)= 38 LEN(D$)= 58
C$=0123456789 +-.DEABCFGHIJKLMNOPQRSTUVWXYZ&',?!#$%()*=@;><,
D$=              DEABCFGHIJKLMNOPQRSTUVWXYZ&',?!#$%()*=@;><,
-------------------------------------------------------------
SC6/SD6          LEN(S$)= 58 LEN(C$)= 45 LEN(D$)= 58
S$=0123456789 +-.DEABCFGHIJKLMNOPQRSTUVWXYZ&',?!#$%()*=@;><,
D$=0123456789 +-.DEABCFGHIJKLMNOPQRSTUVWXYZ&',?!#$%()*=@;><,
-------------------------------------------------------------
SC7/SD7          LEN(S$)= 58 LEN(C$)= 52 LEN(D$)= 58
S$=0123456789 +-.DEABCFGHIJKLMNOPQRSTUVWXYZ&',?!#$%()*=@;><,
D$=0123456789 +-.DEABCFGHIJKLMNOPQRSTUVWXYZ&',?!#$%()*=@;><,
-------------------------------------------------------------
SCP/SDP          LEN(S$)= 101 LEN(C$)= 73 LEN(D$)=101
S$=THIS CONTAINS      IMBEDDED      BLANKS      OR MULTIPLE **
********** OTHER    $$$$$$$$      CHARACTERS
D$=THIS CONTAINS      IMBEDDED      BLANKS      OR MULTIPLE **
********** OTHER    $$$$$$$$      CHARACTERS
-------------------------------------------------------------
SCPM/SDPM        LEN(SCREEN)= 1024  LEN(IA)=277
```

```
PRESS ENTER TO RESTORE SCREEN FROM IA
AND AGAIN TO CONTINUE?
READY
```

Only the displayable characters are illustrated above. A full conversion table may be created using the following program segement:

```
10  FOR I=0 TO 255
20    PRINT I,ASC(&SD4$(&SC4$(CHR$(I)))),
                ASC(&SD5$(&SC5$(CHR$(I)))),
                ASC(&SD6$(&SC6$(CHR$(I)))),
                ASC(&SD7$(&SC7$(CHR$(I))))
30  NEXT
```

D. String Copy Functions
SCPY, SEQU

1. Description - These functions can be used for copying arrays of strings from one location to another. These functions are similar to MELC and MEQU as described in Matrix sections B. and D. above. Both functions copy strings sequentially with the first subscript varying fastest in multi-dimensional arrays. SELC will repeat the source array strings(T$) until the destination array(S$) is full, or "N" elements have been processed. SCPY only copies available strings from T$, or until S$ is full.

2. Program Example - The program below illustrates the use of SCPY and SEQU. In the first two cases a 2 x 3 string array T$(DIM T$(1,2)) is copied to a singly dimensioned array S$ which is larger than T$. The last two cases show only the first four elements of T$ copied to S$.

```
5   DEFUSR=&Hxxxx: PRINT USR(1): CLEAR 1000
10  DIM S$(7),T$(1,2)
20  DATA A,C,E,B,D,F
30  FOR I=0 TO 7: S$(I)=".": NEXT
40  PRINT "T$(I,J)";
50  FOR I=0 TO 1:  PRINT TAB(25);
60    FOR J=0 TO 2
70      READ T$(I,J):  PRINT T$(I,J);"   ";
80    NEXT
90  NEXT: PRINT
100 N=&SCPY(S$,T$):  PRINT"&SCPY(S$,T$);   :GOSUB 500
110 N=&SEQU(S$,T$):  PRINT"&SEQU(S$,T$);   :GOSUB 500
120 N=&SCPY(S$,T$,4):PRINT"&SCPY(S$,T$,4)";:GOSUB 500
130 N=&SEQU(S$,T$,4):PRINT"&SEQU(S$,T$,4)";:GOSUB 500
140 END
500 PRINT TAB(16);"N=";N;TAB(25);
510 FOR I=0 TO 7
520   PRINT S$(I);"   ";: S$(I)="."
530 NEXT
540 PRINT: RETURN
```

The results from running the above program are shown below:

```
>RUN
INFINITE BASIC - VERSION 1.0 - COPYRIGHT 1979,
                 RACET COMPUTES
OK
T$(I,J)                    A   C   E
                           B   D   F
&SCPY(S$,T$)     N= 6      A   B   C   D   E   F   .   .
&SEQU(S$,T$)     N= 8      A   B   C   D   E   F   A   B
&SCPY(S$,T$,4)   N= 4      A   B   C   D   .   .   .   .
&SEQU(S$,T$,4)   N= 4      A   B   C   D   .   .   .   .
READY
```

Note that when T$ is smaller than S$ the results are identical.

E. String Count and Search Functions
SCNT, SMSK

1. These functions perform a search of a string and either counts the number of times a match is found, or the location of the first match. The SMSK function allows the specification of a "mask" character, which forces an equal compare wherever encountered during the search.

2. Program Example - The example below illustrates the use of SCNT and SMSK. In this case a character string is searched under several different conditions.

```
5   DEFUSR=&Hxxxx: PRINT USR(1): CLEAR 1000
10  S$="AAA AAB ABA BAA ABB BBA BBB"
20  M$="."
30  PRINT "S$=";S$;"     M$=";M$
40  PRINT "   T$     &SCNT(S$,T$) &SMSK(S$,T$)
    &SMSK(S$,T$,M$)"
50  PRINT "--------  ------------ ------------
    --------------
60  T$="A"    :GOSUB 500
70  T$="AA"   :GOSUB 500
80  T$=".B"   :GOSUB 500
90  T$="B."   :GOSUB 500
100 T$="BBA"  :GOSUB 500
110 T$="BB"   :GOSUB 500
120 T$="BB."  :GOSUB 500
130 T$="CCC"  :GOSUB 500
140 END
500 PRINT TAB(4);T$;
510 PRINT TAB(15);&SCNT(S$,T$);
520 PRINT TAB(28);&SMSK(S$,T$);
530 PRINT TAB(41);&SMSK(S$,T$,M$)
540 RETURN
```

The results from running the above program are shown below:

```
>RUN
INFINITE BASIC - VERSION 1.0 - COPYRIGHT 1979,
                RACET COMPUTES
OK
S$=AAA AAB ABA BAA ABB BBA BBB     M$=.
   T$     &SCNT(S$,T$) &SMSK(S$,T$) &SMSK(S$,T$,M$)
-------  ------------ ------------ ---------------
   A            11           1             1
   AA           4            1             1
   .B           0            0             6
   B.           0            0             7
   BBA          1            21            21
   BB           4            18            18
   BB.          0            0             18
   CCC          0            0             0
READY
```

Note that after SCNT finds the a match the search continues with the next character after the beginning of the sucessful match. This is illustrated in the second case where "4" occurrences of "AA" are found.

F. Screen Control Functions
SDHL, SDVL, SEHL, SEVL, SSCL, SSCR, SSDN, SSVP

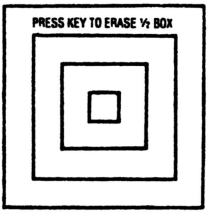
1. Description - These functions are used for drawing, erasing, and scrolling lines on the display screen. The draw/erase line functions(SDHL, SDVL, SEHL, SEVL) utililize the standard graphic X-Y coordinate system. Both horizontal and vertical lines can be manipulated with these functions. The scroll screen routines provide both multiple row and column shifting of contents on the screen.

2. Program Example - Given below is a program that illustrates both the line drawing/erasing capabilities as well as screen scrolling.

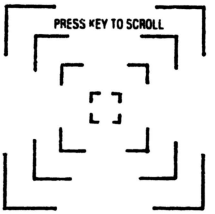
```
5   DEFUSR=&Hxxxx: PRINT USR(1): CLEAR 1000: CLS
10  FOR I=0 TO 3                    'DRAW BOXES
20    I1=I*6:   I2=47-I1:   LY=I2-I1+1
30    J1=I*16:  J2=127-J1:  LX=J2-J1+1
40    J=&SDHL(J1,I1,LX)             'DRAW TOP HORIZ
50    J=&SDHL(J1,I2,LX)             'DRAW BTM HORIZ
60    J=&SDVL(J1,I1,LY)             'DRAW LEFT VERT
70    J=&SDVL(J2,I1,LY)             'DRAW RIGHT VERT
80  NEXT
90  PRINT @84,"PRESS KEY TO ERASE 1/2 BOX";:GOSUB 500
100 FOR I=0 TO 3                    'ERASE 1/2 BOXES
110   I1=I*6:   I2=47-I1:   LY=(I2-I1+1)/2
120   J1=I*16:  J2=127-J1:  LX=(J2-J1+1)/2
130   J=&SEHL(J1+LX/2,I1,LX)        'ERASE 1/2 TOP
140   J=&SEHL(J1+LX/2,I2,LX)        'ERASE 1/2 BTM
150   J=&SEVL(J1,I1+LY/2,LY)        'ERASE 1/2 LEFT
160   J=&SEVL(J2,I1+LY.2,LY)        'ERASE 1/2 RIGHT
170 NEXT
```

```
180 PRINT @84,"   PRESS KEY TO SCROLL   ";:GOSUB 500
190 J=&SSCL(8):  GOSUB 500       'SCROLL LEFT 8
200 J=&SSCN(3):  GOSUB 500       'SCROLL DOWN 3
210 J=&SSCR(16): GOSUB 500       'SCROLL RIGHT 16
220 J=&SSUP(6):  GOSUB 500       'SCROLL UP 6
230 GOTO 190
500 IF INKEY$="" THEN 500 ELSE RETURN
```

The results from running the above program are shown below. Figure(1) shows the display after running Lines #5-90. Figure(2) is produced by Lines #100-180. Pressing a key for each of Lines #190-230 will step shift the screen counter clockwise and off the screen diagonally to the right.



Figure(1) - Draw/Erase Example



Figure(2) - Scroll example.

G. Absolute String Manipulation Functions
SABP, STIN, SVP$

1. Description - These functions are used for setting up and processing string data directly in memory. The SABP function initializes a string variable to point anywhere in memory. STIN stores data from one string into an existing string. SVP$ fetches a string from memory. Several important concepts about storage of strings in memory is first presented, followed by details of the functions.

Character strings in the Radio Shack Basic consist of the actual string data and a special 3-byte pointer. The pointer indicates both the length of the string and the actual starting location of the string data in memory. Consider the following two BASIC statements:

```
A$="THIS STRING DATA IS HERE"
B$=A$ + "IN STRING SPACE"
```

In both cases the three byte string pointers associated with A$ and B$ are maintained in the internal symbol table. The string data for A$ is the actual program text. The data for B$, however, is located in a special "string space" area. The reason for this is that the data was created during program exection.

A third type of string storage is used for strings associated with the FIELD statement. The string data associated with FIELD strings is located in the random I/O buffers.

It is important to realize that the actual location of string data in the "string space" area may change at any moment. The location of the other two types of string data are fixed The SABP and SVP functions extend the location of strings to anywhere within memory. "String Space" should, however, NOT be referenced by function SABP, since that area is under constant reorganization by BASIC.

SABP is used to initialize the string pointer associated with a string variable name to point anywhere in memory. The absolute memory location and length must be specified as arguments along with the variable name. The string variable may either be a scalar or an element of a string array. A typical example of the use of SABP is as follows:

```
10  A$=""
20  I=&SABP(A$,15360,64)
30  DIM B$(10)
40  I=&SABP(B$(3),15363,5)
```

This example specifies that A$ is overlayed on the first 64 characters of the display screen. Similarly, B$(3) will be in the same general area, but consists of only 5 characters. Note that the same area in memory can be associated with several strings at the same time. The string variable names(A$,B$) must be activated prior to use in the SABP function call. This is accomplished in Lines #10 and #30 above, although any executable statement would be acceptable. The return value from SABP is the address of the string pointer(same as VARPTR(A$) or VARPTR(B$(3) above).

Function SVP$ returns a character string containing data from anywhere in memory. The data is copied into the dynamic "String Space" area and is processed as any other character function such as MID$ and CHR$. For example:

```
10  A$=&SVP$(15360,64)
```

would also access the first 64 characters of the display screen. The major difference between SABP and SVP$ is that SABP creates a permanent association between a variable name and memory, while SVP$ is only transient reference to memory.

Function STIN provides the capability of storing string data directly in the same space occupied by an existing string. A BASIC statement of the form:

```
10  A$="ABCD..GHIJ"
20  A$=LEFT$(A$,4) + "XX" + RIGHT$(A$,4)
```

would create three different strings in "String Space"(LEFT$ part, LEFT$ + "XX", and the final result). This results in a considerable amount of overhead in the system. STIN, however,

can insert the data directly in the original string, as shown below:

```
30  I=&STIN(A$,"XX",5,2)
```

The use of this technique this eliminates string reorganization problems commonly encountered. Special care, however, must be taken where the data is stored. In the above example the string data for A$ is in the BASIC program area. The actual BASIC program will be changed. Illegal characters and quote marks(") when inserted into program space area may cause later difficulties.

2. Program Example - The example below illustrates the use of these functions for manipulating strings directly on the display screen. A name and address format is created, with string overlayed on the string. These strings are used for initializing the areas, reading, and storing input data.

```
5   DEFUSR=&Hxxxx: PRINT USR(1): CLEAR 5000
10  DEFINT A-Z
20  DIM NF$(4),ND$(4),DA(4)      'SCREEN LOCATIONS
30  DATA 15360,4,15365,25,15424,7,15432,25
40  DATA 15488,4,15493,15,15509,5,15515,2
50  DATA 15520,3,15525,5
60  BL$=STRING$(25," ")          'BLANKS STRING
70  FOR I=0 TO 4                 'INIT SCREEN LOC
80    READ IA,IL,JA,JL:  DA(I)=JA
90    J=&SABP(NF$(I),IA,IL): J=&SABP(ND$(I),JA,JL)
100 NEXT
110 CLS                          'WRITE NAME FIELDS
120 I=&STIN(NF$(0),"NAME"): I=&STIN(NF$(1),"ADDRESS")
130 I=&STIN(NF$(2),"CITY"): I=&STIN(NF$(3),"STATE")
140 I=&STIN(NF$(4),"ZIP")
150 PRINT @512,""                'INPUT & DISPLAY
160 INPUT "ENTER NAME";NM$:
    I=&STIN(ND$(0),NM$)
170 INPUT "ENTER ADDRESS";AD$:
    I=&STIN(ND$(1),AD$)
180 INPUT "ENTER CITY";CT$:
    I=&STIN(ND$(2),CT$)
190 INPUT "ENTER STATE";ST$:
    I=&STIN(ND$(3),ST$)
200 INPUT "ENTER ZIP";ZP$:
    I=&STIN(ND$(4),ZP$)
210 PRINT @384, &SVP$(DA(0),5)+&SVP$(DA(1),5)+
    &SVP$(DA(2),5)+&SVP$(DA(3),5)+&SVP$(DA(4),5)
220 INPUT "PRESS ENTER TO CLEAR DATA & REPEAT";X$
230 FOR I=0 TO 4: J=&STIN(ND$(I),BL$): NEXT
240 GOTO 150
```

The screen contents after processing an entry is shown below:

```
NAME JOHN J. SMITH
ADDRESS ANYWHERE USA
CITY ANY CITY USA    STATE MI    ZIP   12345
```

```
JOHN ANYWHANY CMI     12345
PRESS ENTER TO CLEAR DATA & REPEAT

ENTER NAME? JOHN J. SMITH
ENTER ADDRESS? ANYWHERE USA
ENTER STATE? MISSISSIPPI
ENTER ZIP? 123456789
```

Lines #10-100 read in the absolute memory locations of the name fields and associated data fields. Function SABP is used to point elements of arrays NF$ and ND$ to these locations. Lines #120-140 store the character data directly on the screen. Lines #160-200 allows the user to input data and then stores it in the pre-defined screen fields. Note that only the specified string sizes are moved. Line #210 shows the use of SVP$ to extract data directly from the screen. Line #230 is used to clear just the data fields to blanks(BL$).

H. String Utility Functions SRV$, SX1$, SX2$

1. Description - The string utility function perform the following:

SRV$	Creates a string composed of a random selection of characters.
SX1$	Creates the hexidecimal representation of the bytes in the source argument. Two hexadecimal characters are created for each byte in the original data.
SX2$	Creates the hexidecimal representation of the bytes in the source argument. Two hexadecimal characters followed by a blank are created for each byte of data in the original data.

The SX1$ and SX2$ functions can create hex representations for integer, single precision, double precision, or character string arguments. In addition, the contents of a specified location in memory can also be converted to hexadecimal. The resulting hexadecimal characters are in the exact order as contained in memory. For Integers this will be the lowest significant byte followed by the highest significant byte.

2. Program Example - Given below is a program illustrating these three functions.

```
5   DEFUSR=&Hxxxx: PRINT USR(1): CLEAR 1000

10  S$=&SRV$(6,65,67)
20  I%=12345
30  A!=1.23456
40  D#=1.23456789012345
50  PRINT "S$=";S$;"  I%=";I%;"  A!=";A!;"  D#=";D#
60  PRINT "&SX1$(S$)=";&SX1$(S$)
70  PRINT "&SX2$(S$)=";&SX2$(S$)
80  PRINT "&SX1$(I%)=";&SX1$(I%)
```

```
90  PRINT "&SX2$(I%)=";&SX2$(I%)
100 PRINT "&SX1$(A!)=";&SX1$(A!)
110 PRINT "&SX2$(A!)=";&SX2$(A!)
120 PRINT "&SX1$(D#)=";&SX2$(D#)
130 PRINT "&SX2$(D#)=";&SX2$(D#)
140 PRINT: PRINT "&SX1$(16416,2)=";&SX1$(16416,2)
150 PRINT        "&SX2$(16416,2)=";&SX2$(16416,2)
160 END
```

The results from running the above program are shown below.

```
>RUN
INFINITE BASIC - VERSION 1.0 - COPYRIGHT 1979,
                 RACET COMPUTES
OK
S$=AACCBCC I%= 12345 A!= 1.23456 D#= 1.2345678901245
&SX1$(S$)=41434342 4343
&SX2$(S$)=41 43 43 42 43 43
&SX1$(I%)=3930
&SX2$(I%)=39 30
&SX1$(A!)=0F061E81
&SX2$(A!)=0F 06 1E 81
&SX1$(D#)=E8CE621452061E81
&SX2$(D#)=E8 CE 62 14 52 06 1E 81
&SX1$(16416,2)=CF3F
&SX2$(16416,2)=CF 3F
READY
```

The results for S$ will vary from run to run since the random function &SRV is used to generate the string. Note that the integer 12345 is represented in lower significant byte first format(The value 12345 in Hex is 3039). When two arguments are used with &SX1$ or &SX2$, the first is an absolute memory address followed by a length. The last two examples show finding the absolute address of the screen cursor contained at memory location 16416(The cursor address will be between 3C00 to 3FFF in Hex).

I. Sort Functions
SRTC, SRTV

1. Description - These two functions are used for sorting data stored in memory. The character string sort function SRTC is used for sorting a single character string array. The multi-variable sort function SRTV is used for sorting a connected group of arrays of any type. Multiple sort-key fields, as well as ascending/descending sort order, may be specified for both SRTC and SRTV.

SRTC sorts a character string array where each element (record) contains one or more sort-key fields. The following record format is typical of many applications:

NAME (1)	ADDRESS (17)	CITY (33)	ST (41)	ZIP (46)
SMITH	AB 101 MAIN ST.	ORANGE	CA	92665
JONES	TA 222 1ST ST.	BLUE	MI	40229
:	:	:	:	:
:	:	:	:	:
WILLIAMS	AQ 321 CROSS ST.	PINK	NY	01022

For SRTC each data record is stored as a single element in a character string array. Five fields are shown in each data record in the above example. One or more of these fields could be selected as "sort-key" fields for a particular sorting application. For example, the sort sequence STATE(ascending), CITY(descending), followed by NAME(ascending) could be specified.

Sort-key field information for SRTC is contained in the integer array argument IE. The first element of IE indicates the number of sort fields. This is followed by pairs of elements in IE indicating the location, length, and sort ascending/descending sequqence for each sort field. The array IE for the above example of STATE(asc), CITY(desc), and NAME(asc) would be:

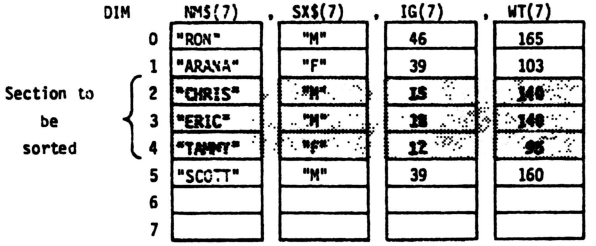
IE(I) I	Value	Description
0	3	Three sort fields specified.
1	41	Start location of STATE field(Asc).
2	2	Length of STATE field.
3	-33	Start location of CITY field(Desc).
4	8	Length of CITY field.
5	1	Start location of NAME field(Asc).
6	16	Length of NAME field.

Note that each sort field requires two entries. The first specifies the length and ascending/descending sort status. Positive values indicate ascending sort and negative values for descending sort sequence. The second entry of the pair indicates the number of characters in each field. All remaining unspecified fields will be carried along in the sort.

SRTV is used for sorting a group of singly dimensioned arrays, each connected element-by-element. One or more integer, single precision, double precision, or character string arrays can be in the group. For example, consider the case of four arrays as follows:

Array Name	Type	Description
NM$	Character	Name array
SX$	Character	Sex code(M/F) array
IG	Integer	Age array
WT	Single Prec	Weight array

These arrays can be connected to form a group where each sort record consists of corresponding elements from each array. Figure(1) below illustrates connecting the four arrays, each dimensioned to a maximum of eight elements (0 - 7):



Figure(1). Example of sort group.

Each array is considered to be a sort-key field. The order in which the arrays are specified in the group determines which sort-key has higher precedence.

SRTV requires the user to specify which elements(are to be grouped for sorting, and the ascending/descending sequence of each sort-key field. This is accomplished by use of a sort parameter specification string(S$). This single character string contains the names of the arrays to be grouped along with ascending(+) or descending(-) information. This is best illustrated with several examples:

S$	Description
+NM$,-SX$,+IG,WT	The NM$ field will be the primary sort field in ascending order(+). SX$(descending -), IG(ascending +) will be secondary sort-keys. WT will be carried along in the sort without order checking.
+SX$,-IG,NM$,WT	SX$ will be the primary (asc +) sort field, then IG(desc -). NM$ and WT are included in the sort.
-IG,-SX$	Only IG(desc -) and SX$(asc +) sorted. NM$ and WT were not used.

From the above it can be seen that it is only necessary to change the contents of S$ to perform a variety of sorts. In fact, S$ can be placed in an INPUT statement allowing specification at execution time.

The range of elements to be sorted by both SRTC and SRTV are specified by arguments II and JJ. These indicate the starting subscript and ending subscript for the sort. For the application shown in Figure(1) above, the records starting with "CHRIS" through "TAMMY" are indicated to be sorted. This corresponds to the range of II=2 through JJ=4. All elements preceeding II will be left unchanged. All elements after JJ will also be left unchanged, except for the last TWO elements in each array. These two elements are used internally by the sort program, and will be set to null strings(for character arrays) upon return. The user MUST dimension each array at least TWO larger than the maximum value of JJ to be used.

2. Program Example - Given below are several examples of using both SRTC and SRTV.

a. SRTC Example

```
5   DEFUSR=&Hxxxx: PRINT USR(1): CLEAR 10000 DEFINT I
10  DATA "         1         2         3         4
              5"
20  DATA "12345678901234567890123456789012345678901
          1234567890"
30  DATA "SMITH          AB101 MAIN ST.     ORANGE
          CA   92665"
40  DATA "JONES          TA222 1ST ST.      BLUE
          MI   40229"
50  DATA "BLACK          CD222 2ND ST.      GREEN
          MI   40229"
60  DATA "WHITE          AE222 2ND ST.      BLUE
          MI   40229"
70  DATA "DOE            J 102 MAIN ST.     ORANGE
          CA   92665"
80  DATA "SMITH         AB 123 NORTH ST.    GREEN
          KA   55346"
90  DIM NA$(10),I1(10),I2(10)
100 READ H1$,H2$                          'READ HEADERS
110 FOR I=0 TO 5                         'INIT SORT DATA
120   READ NA$(I)
130 NEXT
140 DATA 3,41,2,-33,8,1,16          'SORT-FIELD #1 INFO
150 FOR I=0 TO 5
160   READ I1(I)
170 NEXT
180 PRINT           "SORT BY +STATE, -CITY, ;NAME";
190 IR=&SRTC(NA$,0,5,I1): GOSUB 500
200 DATA 2,46,5,1,16                'SORT-FIELD #2 INFO
210 FOR I=0 TO 5
220   READ I2(I)
230 NEXT
240 PRINT                       "SORT BY +ZIP, +NAME";
250 IR=&SRTC(NA$,1,4,I2): GOSUB 500
260 END
500 PRINT TAB(31);"SORT RETURN CODE=";IR
510 PRINT H1$: PRINT H2$
520 FOR I=0 TO 5: PRINT NA$(I): NEXT
530 PRINT STRING$(50,"-")
540 RETURN
```

The results from running the above program are shown below:

```
>RUN

INFINITE BASIC - VERSION 1.0 - COPYRIGHT 1979,
                 RACET COMPUTES
OK
SORT BY +STATE, -CITY, +NAME   SORT RETURN CODE= 0
         1         2         3         4         5
12345678901234567890123456789012345678901234567890
DOE            J102 MAIN ST.     ORANGE   CA   92665
SMITH         AB101 MAIN ST.     ORANGE   CA   92665
SMITH         AB123 NORTH ST.    GREEN    KA   55346
```

```
BLACK          CD222 2ND ST.      GREEN    MI    40229
JONES          TA222 1ST ST.      BLUE     MI    40229
WHITE          AE222 2ND ST.      BLUE     MI    40229
-------------------------------------------------------
SORT BY +ZIP, -NAME              SORT RETURN CODE= 0
        1         2         3         4         5
12345678901234567890123456789012345678901234567890
DOE             J102 MAIN ST.     ORANGE   CA    92665
BLACK          CD222 2ND ST.      GREEN    MI    40229
JONES          TA222 1ST ST.      BLUE     MI    40229
SMITH          AB123 NORTH ST.    GREEN    KA    55346
SMITH          AB101 MAIN ST.     ORANGE   CA    92665
WHITE          AE222 2ND ST.      BLUE     MI    40229
-------------------------------------------------------
READY
```

The above example sorts the character string array NA$ in two different sequences. The first sorts all six initialized elements in order by STATE(ascending), CITY(descending), and finally by NAME(ascending). The second sequence sorts only the middle elements(1-4) in order by ZIP(ascending) and NAME(ascending). In this case note that elements 0 and 5 do not change.

b. SRTV Example.

```
5   DEFUSR=&Hxxxx: PRINT USR(1): CLEAR 10000
10  DATA RON,M,46,165
20  DATA ARANA,F,39,103
30  DATA CHRIS,M,15,140
40  DATA ERIC,M,18,140
50  DATA TAMMY,F,12,95
60  DATA SCOTT,M,39,160
70  DIM NM$(7),SX$(7),IG(7),WT(7)
80  FOR I=0 TO 5                'INIT DATA TO BE SORTED
90    READ NM$(I),SX$(I),IG(I),WT(I)
100 NEXT
110 PRINT      "SORT(2-4) BY";: S$="+NM$,-SX$,+IG,WT"
120 IR=&SRTV(S$,2,4): GOSUB 500
130 PRINT      "SORT(0-5) BY";: S$="+SX$,-IG,NM$,WT"
140 IR=&SRTV(S$,0,5): GOSUB 500
150 PRINT      "SORT(0-5) BY";: S$="-IG,-SX$"
160 IR=&SRTV(S$,0,5): GOSUB 500
170 END
500 PRINT "  ";S$;TAB(36);"RETURN CODE=";IR
510 PRINT " I  NM$(I)   SX$(I)   IG(I)    WT(I)"
520 PRINT "--- -------  -------  -------  -------"
530 FOR I=0 TO 7
540   PRINT I;TAB(5);NM$(I);TAB(14);SX$(I);TAB(23);
          IG(I);TAB(832);WT(I)
550 NEXT
560 PRINT STRING$(50,"-")
570 RETURN
```

The results from running the above program are shown below

```
>RUN
INFINITE BASIC - VERSION 1.0 - COPYRIGHT 1979,
                RACET COMPUTES
OK
SORT(2-4) BY  +NM$,-SX$,+IG,WT       RETURN CODE= 0
 I  NM$(I)    SX$(I)    IG(I)     WT(I)
--- -------  -------  -------  -------
 0  RON       M          46       165
 1  ARANA     F          39       103
 2  CHRIS     M          15       140
 3  ERIC      M          18       140
 4  TAMMY     F          12        95
 5  SCOTT     M          39       160
 6                       0         0
 7                       0         0
-------------------------------------------------
SORT(0-5) BY  +SX$,-IG,NM$,WT        RETURN CODE =0
 I  NM$(I)    SX$(I)    IG(I)     WT(I)
--- -------  -------  -------  -------
 0  ARANA     F          39       103
 1  TAMMY     F          12        95
 2  RON       M          46       165
 3  SCOTT     M          39       160
 4  ERIC      M          18       140
 5  CHRIS     M          15       140
 6                       0         0
 7                       0         0
-------------------------------------------------
SORT(0-5) BY  -IG,-SX$               RETURN CODE =0
 I  NM$(I)    SX$(I)    IG(I)     WT(I)
--- -------  -------  -------  -------
 0  ARANA     M          46       103
 1  TAMMY     M          39        95
 2  RON       F          39       165
 3  SCOTT     M          18       160
 4  ERIC      M          15       140
 5  CHRIS     F          12       140
 6                       0         0
 7                       0         0
-------------------------------------------------
READY
```

Lines #110-120 sort only elements 2-4 of the connected arrays NM$, SX$, IG, and WT. NM$ was the primary sort key as can be seen from the results. Secondary sort keys of descending SX$ and ascending IG were specified, although not needed since no identical NM$ elements were present in data. Note that elements 0,1, and 5 were not involved in the sort. Elements 6 & 7 were used by the sort for work space, but reset to zero(null) before returning to the user.

Lines #130-140 sort all initialized elements 0-5. In this case SX$ is the primary key. Since repeating elements of SX$ were found, the secondary keys of descending IG and ascending NM$ were used to determine the final order.

Lines #150-160 show only two of the arrays connected, with IG as the primary key and SX$ as the secondary key(both descending sequence). Arrays NM$ and WT were not involved in the sort, and therefore were not changed.

c. SRTV/Disk Sort Example

```
5   DEFUSR=&Hxxxx: PRINT USR(1): CLEAR 10000
10  DEFINT I-N
20  DIM R$(3),KY$(129),IX(129)
30  OPEN "R",1,"SORT/DAT:1"    'OPEN & SET FIELDS
40  FOR I=0 TO 3: FIEL1, I*64 AS D$,63 AS R$(I):NEXT
50  IF LOF(1)<>0 THEN 120      'SKIP IF FILE EXISTS
60  FOR I=0 TO 127             'CREATE AND WRITE
70    J=I/4: K=I-J*4           'RANDOM RECORDS
80    LSET R$(K)=&SRV$(9,65,69)+" "+&SRV$(10,78,82)+
      " "+STRING$(42,".")
90    IF K=3 THEN PUT 1,J+;1   'BLOCKED 4 REC/SECTOR
100 NEXT
110 IF K <> 3 THEN PUT 1,J+;1
120 PRINT "READ FILE & EXTRACT KEYS"
130 FOR I=0 TO 127             'READ & SAVE KEYS
140   J=I/4: K=I-J*4           'ALONG WITH CORRES-
150   GET 1,J+;1               'PONDING RECORD INDEX
160   KY$(I)=MID$(R$(K),10,10)
170   IX(I)=I
180 NEXT
190 PRINT "SORT KEYS, CARRYING ALONG RECORD INDEX"
200 IR=&SRTV("+KY$,IX",0,127)
210 PRINT "INDEX SORTED - RETURN CODE=";IR
220 PRINT "FETCH & PRINT RECORDS IN SORTED ORDER"
230 FOR I=0 TO 127             'SCAN INDEX FILE
240   II=IX(I)                 'CALCULATE CORRES-
240   J=II/4: K=II-J*4         'PONDING SECTOR/REC
240   GET 1,J+;1               'FETCH SECTOR
250   PRINT R$(K)              'PRINT RECORD
260 NEXT
270 CLOSE
280 END
```

The partial results from running the above program are shown below:

```
>RUN
INFINITE BASIC - VERSION 1.0 - COPYRIGHT 1979
                 RACET COMPUTES
OK
READ FILE & EXTRACT KEYS
SORT KEYS, CARRYING ALONG RECORD INDEX
INDEX SORTED - RETURN CODE= 0
ABDDBBCCB NNOPNRPPRR ......................  etc
DDDBEDACB NNPRPOQRQN ......................
DDBCAAAAD NNQPRQOQQO ......................
DDDBDACBA NNRQPRONOO ......................
BCCEACDDC NOOORQOQRQ ......................
ECDBDCABD NOQNQORRNP ......................
DEEACCDDE NORRQORORO ......................
    :         :                :
    :         :                :
CBEEBDBAE RRONPNQNRP ......................
```

```
DBCACECAC RRORPNPOPP .....
CEDCACCDE RRORPPPRQR .....
EDEEBBECD RRQOONQQRQ .....
READY
```

This more complicated example illustrates the use of SRTV for sorting a disk file. Large amounts of data encountered in many applications can be stored as a disk file. SRTV cannot be used directly if the file is larger than available memory. Usually, however, only a small portion of a data record is examined during sorting, such as a NAME, ZIP, or ACCT# fields. This example illustrates the use of an "indexed" sort technique, where only the sort fields are extracted from the record and sorted. Information is carried along in the sort to allow accessing the remainder of data in each record.

The example above creates a sample file containing 64 byte records packed 4/sector. The function &SRV$ is used to generate random strings for illustration purposes(Line #80). In this case it is assumed that the sort field consists of characters 10-19. Lines #10-110 create the file unless it already exists. Lines #120-180 reads each record and extracts just the sort key, saving it in array KY$. The corresponding record index number is also saved in array IX. The extra calculations in Lines 70 & 140 illustrate the techniques for packing records.

Line #200 performs the sort of the sort fields. In this case the sort parameter string ("+KY$,IX") specifies to sort the connected arrays KY$(ascending order) and carrying along array IX. Lines #230-280 selects the next element from the array IX to find the next record in sort sequence. This is used to access the original disk file and print the corresponding record.

Several variations of this technique could be used. For example, two(or more) sort fields such as NAME and ZIP could be extracted during the initial pass along with the corresponding index number. SRTV could then be used to create a sorted index in either NAME or ZIP order. The resulting sorted index files could also be written to a separate file in order to save rebuilding the index.